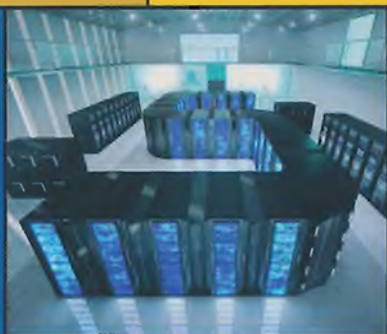


Университетский учебник

А. О. Лацис

# ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ



Прикладная математика  
и информатика

## **Редакционный совет серии**

### **Председатели совета:**

академик РАН Ю. И. Журавлев,  
академик РАН В. А. Садовничий

### **Члены совета:**

О. М. Белоцерковский (академик РАН),  
В. П. Дымников (академик РАН),  
Ю. Г. Евтушенко (академик РАН),  
И. И. Еремин (академик РАН),  
В. А. Ильин (академик РАН),  
П. С. Краснощеков (академик РАН),  
Е. И. Моисеев (академик РАН),  
А. А. Петров (академик РАН),  
Л. Н. Королев (член-корреспондент РАН),  
Д. П. Костомаров (член-корреспондент РАН),  
Г. А. Михайлов (член-корреспондент РАН),  
Ю. Н. Павловский (член-корреспондент РАН),  
К. В. Рудаков (член-корреспондент РАН),  
Е. Е. Тыртышников (член-корреспондент РАН),  
И. Б. Федоров (член-корреспондент РАН),  
Б. Н. Четверушкин (член-корреспондент РАН)

### **Ответственный редактор серии**

доктор физико-математических наук  
Ю. И. Димитриенко

УНИВЕРСИТЕТСКИЙ УЧЕБНИК

---

Серия «Прикладная математика и информатика»

А. О. ЛАЦИС

# ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

*Допущено*

*Учебно-методическим объединением*

*по классическому университетскому образованию*

*в качестве учебного пособия для студентов высших учебных заведений,  
обучающихся по специальности «Прикладная математика и информатика»*



Москва

Издательский центр «Академия»

2010

УДК 519.254(075.8)  
ББК 32.98я73  
Л302

Рецензенты:

д-р техн. наук, проф., академик РАН *В. К. Левин*  
(научный руководитель ФГУП НИИ «Квант»);  
д-р физ.-мат. наук, доц. *М. В. Яковлевский*  
(заведующий сектором Института математического  
моделирования РАН)

**Лацис А. О.**

**Л302** Параллельная обработка данных : учеб. пособие для студ. вузов / А. О. Лацис. — М. : Издательский центр «Академия», 2010. — 336 с. — (Университетский учебник. Сер. Прикладная математика и информатика).

ISBN 978-5-7695-5951-8

В учебном пособии дан углубленный систематический обзор технологий параллельной обработки данных. Основное внимание уделено традиционным программным технологиям параллельного программирования на кластерных вычислительных системах, причем не только программистским моделям, но и их отображениям на вычислительном оборудовании; объяснено происхождение программистских технологий и отражение в них существенных для системного и прикладного программиста свойств оборудования.

Описан ускоряющийся в последние годы переход на новые нетрадиционные аппаратные архитектуры, причем как в области технологий объединения процессоров в параллельную вычислительную систему, так и в области внутренней организации самих вычислителей. Рассмотрены вопросы пользования высокоскоростных коммуникационных магистралей Hypertransport и PCI Express, проблемы создания и внедрения реконфигурируемых вычислителей на базе технологий программируемой логики. Кратко затронута проблематика метакомпьютинга, технологий отбора неиспользуемой вычислительной мощности из Интернета.

Для студентов высших учебных заведений.

УДК 519.254(075.8)  
ББК 32.98я73

*Оригинал-макет данного издания является собственностью  
Издательского центра «Академия», и его воспроизведение любым способом  
без согласия правообладателя запрещается*

ISBN 978-5-7695-5951-8

© Лацис А. О., 2010  
© Образовательно-издательский центр «Академия», 2010  
© Оформление Издательский центр «Академия», 2010



## ПРЕДИСЛОВИЕ

Несколько лет назад среди назойливо мелькающих на телеэкранах рекламных слоганов был примерно такой: «Есть люди, которые разбирают машины, а есть люди, которые в них разбираются». Речь шла, конечно, не о вычислительных машинах, а об автомобилях. Точнее, о новом автомобильном журнале. Видеоряд, помнится, демонстрировал впечатляющую разницу между слесарем со страшным взглядом и паровозным гаечным ключом в руках — и очевидно успешным, импозантным владельцем нового автомобиля. Словом — не тратьте сил на ерунду, доверьте пачкать руки профессионалам, лучше разберитесь во всей палитре возможностей, которую являют вам современные технологии, и потом сделайте правильный выбор.

Достигнув поры зрелости, многие отрасли нуждаются в неизмеримо большем числе «разбирающихся», чем «разбирающих». То же самое можно было бы сказать и про индустрию, специализирующуюся на разработке и производстве массовых компьютеров и предназначенного для них программного обеспечения. Но верно ли это в отношении людей, претендующих на карьеру в той чрезвычайно узкой области, что занимается созданием программных продуктов, инженерных решений и производством «штучных» изделий, именуемых суперкомпьютерами?

Время летит быстро. Мы — поколение тех, кто еще помнит, что такое «пульт центрального процессора» и как он выглядит, оглянуться не успели, как в офисы компьютерных фирм пришло совершенно новое поколение молодых профессионалов. Последнее слово сказано безо всякой иронии. Конечно, ни один руководитель коллектива разработчиков не возьмет на работу сотрудника, плохо разбирающегося в особенностях множественного полиморфизма и наследования, не владеющего хотя бы одним языком запросов к базе данных. Но спросите у современных выпускников профильных вузов, что такое прерывание? Что такое процесс? Что такое машина фон Неймана, наконец? Именно о ней я недавно спросил наугад нескольких своих студентов. По-

лучив три раза подряд неуверенный ответ, будто это, наверное, что-то вроде машины Тьюринга, я понял, что наш мир окончательно стал другим.

Нравится мне это или нет, но это нормально. Люди знают и умеют то, что им надо знать и уметь. И это здорово, что в современном компьютере утомительные подробности внутреннего устройства, которыми нас мучили на младших курсах, надежно спрятаны за каменной стеной ОС и системы программирования, так что разработчик может сосредоточиться на решении исключительно творческих задач. Но это здорово лишь там и тогда, когда эти самые надежно спрятанные технические подробности остаются неизменными. Когда же устоявшийся, вечный и незыблемый фундамент традиционных технологий начинает меняться, «разбирающийся, но не разбирающий» профессионал оказывается в роли деревенского жителя слаборазвитой страны, который умеет сменить провайдера сотовой связи, но не умеет ремонтировать собственный трактор.

Казалось бы, пример с трактором неудачен. Ничто в нашем мире не меняется так стремительно, как компьютерные технологии. Но парадокс заключается в том, что их фундамент — основы архитектуры процессоров и операционных систем — чудовищно консервативны. В главном они не меняются уже почти 50 лет. Именно поэтому их и удалось, в итоге, так надежно «спрятать».

Есть, однако, одна отрасль компьютерного мира, которая именно сейчас, на исходе первого десятилетия XXI века, стоит перед реальной перспективой не просто быстрых и значительных, но глубинных перемен, в буквальном смысле этого слова — сотрясения основ. И отрасль эта — высокопроизводительные вычисления, все то, что связано с суперкомпьютерами.

Да, массовой нашу отрасль, при всем желании, не назовешь. Но она существует, и нуждается в специалистах, способных ответить на вызов времени. Когда на смену процессору Pentium-3 приходит процессор Pentium-4, особенности различий между ними можно смело оставить специалистам соответствующего отдела фирмы Intel. Когда на смену процессору приходит реконфигурируемая логическая матрица, понимание того, что такое процессор, как он вообще бывает устроен, становится вопросом жизни или угасания суперкомпьютерной отрасли. Начинаешь лихорадочно искать тех немногих, кто все еще помнит, что там, у этого самого процессора, внутри.

И каждый раз таких специалистов находишь все с большим трудом. А то и не находишь вовсе, поскольку их уже нет с нами.

Когда три года назад мы начинали работу над принципиально новым суперкомпьютером, совсем не похожим на машины из тогдашнего списка Тор-500, нам не раз и не два мучительно хотелось посоветоваться со своими учителями. С теми, кто на заре отечественной информатики еще не делил работу компьютерного инженера на «математику», «системное программирование» и «электронику», а умел видеть задачу в целом, во всей сложности взаимосвязей, не всегда очевидных узкому специалисту. На что способен не просто инженер, а Инженер. Одним из таких Инженеров для меня был и навсегда останется мой первый учитель компьютерного дела, мой первый руководитель в ИПМ им. М. В. Келдыша, Всеволод Серафимович Штаркман.

Он разбирал и собирал все, что могут сделать человеческие руки и придумать человеческая голова. Он всегда искренне недоумевал и огорчался, видя, как мы, тогда еще молодые системные программисты, не понимали аппаратуру. А зачастую, просто боялись ее. Лично меня он в свое время буквально заставил изучить основы логической схемотехники, освоить на практике азы наладки и диагностики. В работе системного программиста, имеющего дело с опытными образцами новой техники, эти знания и навыки за многие годы работы в ИПМ выручали меня не раз и не два. До дня, когда я, спустя долгие 20 лет, сел за консоль схемотехнической САПР, чтобы впервые самому «нарисовать» макет векторного вычислителя, Всеволод Серафимович не дожид двух недель.

Ему, и всем тем, кто не только разбирался —

уникальному профессионалу, давшему старт одному из первых суперкомпьютерных проектов России, первому Главному инженеру ИПМ им. М. В. Келдыша Анатолию Николаевичу Мямлину;

первому программисту в истории СССР Любови Борисовне Морозовой, проработавшей в ИПМ много лет;

моему коллеге по работе в ИПМ Андрею Борисовичу Ходулеву, а также другим замечательным сотрудникам ИПМ, многих из которых, увы, уже нет с нами, посвящается эта книга.

## ВВЕДЕНИЕ

Что такое параллельная обработка данных? Попытка задуматься над этим, казалось бы, довольно отвлеченным вопросом поможет нам сделать некоторые вполне практические выводы — например, построить план этой книги. Вопрос же сам по себе вовсе не так прост, как кажется.

Нам хорошо известно, что почти все устройства цифровой обработки данных — в частности, компьютеры общего назначения — оперируют данными в двоичной системе счисления. При этом они являются устройствами дискретного действия, и работают на некоторой рабочей частоте. Следовательно, параллельной, строго говоря, является любая обработка данных, в процессе которой за один такт обрабатываются более двух бит информации, т.е. за один «квант» времени выполняется более одного «кванта» обработки. Значит, любая обработка данных — параллельная? Теоретически — да. О чем же конкретно тогда эта книга?

Жанр введения до поры избавляет нас от необходимости соблюдать математический уровень строгости рассуждений. Поэтому прежде, чем переходить ко вполне формальным определениям, развернутым пояснениям и примерам, попробуем внести ясность в некоторые общие вопросы. Здесь и далее первое, определяющее вхождение в текст нового термина будем выделять *жирным курсивом*.

Для начала попробуем связать в логическую цепочку такие часто употребляемые сегодня понятия, как параллельная обработка данных, суперкомпьютер, высокопроизводительные вычисления, параллельная вычислительная система. Для этого нам понадобится ответить на совсем уже, казалось бы, простой вопрос — а что такое «обычный компьютер»?

Примерно 60 лет назад, на заре современной информатики, был придуман некоторый способ объединения логических элементов в вычислительную систему, которой можно было пользоваться, не задумываясь о физике ее функционирования. Способ

этот получил название *фоннеймановской машины* (см. далее), и оказался настолько удачным, что практически без изменений дожил до сегодняшнего дня. Мы привыкли к нему настолько, что называем построенные этим способом машины «просто компьютер», «обычный компьютер». Действительно, до тех пор, пока других компьютеров не было или почти не было, всякие рассуждения о том, что обычный компьютер — фоннеймановский, носили налет некоторого академизма или даже неуместного подчеркнутого педантизма. Совсем иначе обстоит дело теперь, когда существование других компьютеров перестало быть редкостью.

Фоннеймановский компьютер, строго говоря — параллельная вычислительная система. В развернутой формулировке своих знаменитых принципов [1] фон Нейман настаивал на том, что операции над числами в его машине должны выполняться параллельно на уровне двоичных разрядов, т. е. не разряд за разрядом, а, по возможности, со всеми разрядами сразу. В то время это был труднодостижимый идеал. Необходимое для такой организации арифметического устройства количество ламп и диодов трудно было собрать в одну схему и заставить работать без ошибок.

Так или иначе, поразрядный параллелизм был заложен в архитектуру уже тем фактом, что операции над числами задавались как единые, неделимые машинные команды.

Шли годы, и все настолько привыкли к фоннеймановскому компьютеру, что перестали считать его параллельным — заложенный в него уровень параллелизма стал подразумеваться как нечто, само собой разумеющееся, перестал быть достойным особого упоминания.

Количество логических элементов, технологически доступных для объединения в единую вычислительную систему, со времен фон Неймана до наших дней выросло в миллиарды раз, и этот рост продолжается. Примерно в конце 80-х годов XX в. фоннеймановские принципы, оставаясь фундаментальными, стали уже недостаточными. Математики же требовали от разработчиков вычислительных систем все большего и большего быстродействия. Возникла настоятельная потребность дополнить, «надстроить» фоннеймановские принципы, т. е. научиться строить машины гораздо «более параллельные», чем фоннеймановский компьютер. Именно такие системы получили название *параллельных вычислительных систем*, а способы их построения

и использования стали называться **технологиями параллельной обработки данных**. Чтобы подчеркнуть отличие вновь создаваемых компьютеров от традиционных машин, последние стали называть **последовательностными** (или даже просто **последовательными**) **компьютерами**, что, строго говоря, не совсем точно, но отражает факт поочередного выполнения отдельных команд, из которых состоит программа.

Тут надо заметить, что еще задолго до того, как возникла настоятельная необходимость дополнения фоннеймановских принципов, компьютеры разделились на машины массового (по тогдашним понятиям, конечно) производства и машины рекордной производительности, выпускаемые отдельными экземплярами или малыми сериями. Последние стали называть **суперкомпьютерами**, а способы их построения и использования — **технологиями высокопроизводительных вычислений**. До некоторых пор эти технологии развивались в рамках фоннеймановских принципов, но в конце 80-х, когда это перестало быть возможным, **суперкомпьютеры стали параллельными вычислительными системами**, а **технологии высокопроизводительных вычислений стали неотделимыми от технологий параллельной обработки данных**.

Важно отметить, что «мотором» развития суперкомпьютерных (и вообще компьютерных) архитектур является именно рост числа доступных для объединения в систему логических элементов (часто говорят о числе транзисторов, подразумевая, что каждый логический элемент строится из вполне определенного и небольшого числа транзисторов). Радикально новая архитектура может быть придумана и чисто умозрительно, может даже пройти стадию опытных образцов, но фактом компьютерной промышленности она становится тогда, когда возросшее число доступных элементов перестает «умещаться» в старую. Каждый такой технический переворот просто-таки мучителен для пользователей, поэтому идут на него лишь тогда, когда все средства работы в старых рамках действительно исчерпаны.

В последние два-три года технологии параллельной обработки данных начали применяться в классе компьютеров массового выпуска. Во второй части книги мы об этом обязательно расскажем. Пока же будем считать, что компьютер массового выпуска — это последовательная (фоннеймановская) машина, а суперкомпьютер — это параллельная вычислительная система, и сосредоточимся на суперкомпьютерах.

В своем развитии параллельные суперкомпьютеры прошли два четко различимых этапа.

На первом этапе они строились, в основном, как системы из большого числа интегрированных в единую систему последовательных машин, или же последовательных машин с небольшими, четко очерченными параллельными расширениями, такими, как команды векторной обработки данных. На этом этапе существует множество различных способов построения системы из составляющих ее узлов, но узлом в любом случае является «обычный» или «почти обычный» компьютер. Именно эти технологии подробно рассматриваются в части I книги.

Данный этап был длительным и в настоящее время завершается.

Завершение первого и переход ко второму этапу обусловлены теми же причинами, что и наметившийся в последние два-три года переход к технологиям параллельной обработки данных в классе компьютеров массового выпуска. Предел числа элементов, пригодного для «упаковки» в фоннеймановскую архитектуру, в настоящее время уже далеко перекрыт не только в рамках вычислительной системы в целом, но и в рамках отдельного микропроцессора. Для превращения этого невообразимого количества элементов в реально достижимое вычислительное быстродействие уже не достаточно архитектуры из так или иначе взаимодействующих фоннеймановских узлов. Надо менять «конструкцию» самого узла.

Процесс появления соответствующих технологий, как отмечалось ранее, только начинается. Почти наверняка его результаты по своей революционности и неожиданности превзойдут все мыслимые и немыслимые ожидания и прогнозы, причем в самое ближайшее время. Круг вопросов, подлежащих обсуждению при систематическом изложении предмета, не устоялся и стремительно меняется. По этой причине текст учебного пособия разделен на две части.

**В части I** «Освоенные технологии» предмет представлен так, как его следовало излагать, например, два-три года назад с учетом линейного, количественного развития, имевшего место за эти два-три года в рамках традиционных подходов. В начале изложения об освоенных технологиях высокопроизводительных вычислений некоторое внимание уделено краткому обзору основ архитектуры ЭВМ и операционных систем. Опыт показывает, что у значительной части читателей знание этих основ имеет

слишком общий характер, не позволяющий проследить конкретную логику изложения некоторых вопросов организации высокопроизводительных вычислений.

**В части II** «Новые технологии» представлены принципиально новые технологии, методы и подходы.

Цель такого разделения материала — не только естественное желание автора «подстраховаться», поскольку стремительность развития отрасли может буквально завтра сделать нынешние суждения автора неверными. Гораздо важнее то, что часть II подразумевает рассмотрение предмета в терминах, обычно не используемых в программистской литературе. Это само по себе тяжело для читателей, и «размазывать» эти термины по всему тексту не хотелось бы. Именно поэтому достаточно сложный и непривычный материал части II гораздо легче излагать, опираясь на уже законченное рассмотрение освоенных на данный момент технологий.

Небольшое замечание о списке литературы, использованной при подготовке этой книги. Около двух третей этого списка составляют ссылки на сетевые ресурсы, и в современных условиях это вряд ли могло бы быть иначе. К сожалению, сетевые ресурсы иногда исчезают, или меняется их адрес. К большинству использованных в этой книге ссылок это, скорее всего, не относится — вряд ли portalу [www.ibm.com](http://www.ibm.com), например, грозит изменение названия в ближайшем будущем. Тем не менее, хотелось бы напомнить читателям, что основной метод поиска дополнительной информации на сегодня — это поиск, в первую очередь — в англоязычном сегменте Интернет, по ключевым словам. Включенную в список литературы ссылку на сайт, который к моменту чтения данной книги перестал существовать, следует понимать как указание, что данную информацию следует искать (и легко найти) в Интернете.



# ЧАСТЬ I

## ОСВОЕННЫЕ ТЕХНОЛОГИИ

---

### Глава 1

## ТАКСОНОМИЯ СУПЕРКОМПЬЮТЕРОВ И ПРИМЕНЯЕМЫХ В СВЯЗИ С НИМИ ПРОГРАММИСТСКИХ ТЕХНОЛОГИЙ

Приведенное ниже определение суперкомпьютера не бесспорно, как и все многочисленные определения данного термина, встречающиеся в специальной литературе. В целом оно согласуется с пояснениями, данными во введении.

***Суперкомпьютером*** называется вычислительная система, вычислительное быстродействие которой многократно выше, чем у современных ей компьютеров массового выпуска [2].

Первые компьютеры, изготовленные в середине XX в., все без исключения были суперкомпьютерами, поскольку являлись машинами для вычислительных приложений, выпускаемые единичными экземплярами.

Из определения автоматически следует, что суперкомпьютер — изделие мелкосерийное или даже штучное, причем при его производстве применяются технологии и подходы высокой стоимости. Какими же факторами определяется конкретный выбор тех или иных технологических подходов к созданию суперкомпьютера?

На самом деле, господствующий в то или иное время облик «типичного суперкомпьютера» в решающей мере определяется лишь одним очень важным фактором, о котором незаслуженно часто забывают. Попробуем вывести это утверждение из приведенного выше Определения.

Сотни, в лучшем случае — тысячи, разработчиков суперкомпьютеров во всем мире пытаются создавать вычислительные системы, гораздо более быстрые, чем компьютеры серийного выпуска. Между тем над совершенствованием компьютеров серийного выпуска работают сотни тысяч, если не миллионы, человек. Как первым обогнать вторых? Как добиться того, чтобы за вре-

мя реализации новых суперкомпьютерных разработок — пусть даже совершенно гениальных — в магазине еще не успели появиться компьютеры серийного выпуска с еще бóльшим быстродействием? Для решения проблем, затронутых в перечисленных вопросах, разработчикам суперкомпьютеров необходимо работать очень **быстро**. Это возможно лишь в случае, если они будут максимально заимствовать решения, технологии и компоненты из компьютерной индустрии серийного выпуска, с тем, чтобы скомбинировать их по-новому, **точечно** вложить собственные силы в изготовление лишь **недостающих** фрагментов системы, и получить желанное новое качество — суперкомпьютер.

Но заимствуемые решения, компоненты и технологии очень быстро прогрессируют, и время от времени количество изменений переходит в качество — **технически оправданные способы заимствования меняются**. Вместе с ними очень быстро — буквально в течение двух-трех лет — меняется и облик суперкомпьютеров, т.е. происходит очередная суперкомпьютерная революция.

Данный фактор, который можно условно назвать **правилом экономии сил на разработку суперкомпьютера**, был существенным всегда. С переходом (в 80-х годах XX в.) компьютера крупносерийного выпуска в категорию действительно массовых изделий, выпускаемых многомиллионными тиражами, этот фактор стал решающим. Об этом необходимо постоянно помнить, анализируя и сравнивая на бумаге те или иные изящные умозрительные конструкции, претендующие на гордое звание суперкомпьютерных архитектур.

Необходимость следовать правилу экономии сил проходит «красной нитью» через всю иерархию суперкомпьютерных технологий, затрагивая решающим образом даже такие, казалось бы, далекие от экономики области, как модели и парадигмы параллельного программирования и численные методы параллельного решения задач математической физики. Рассматривая и сравнивая модели и технологии параллельного программирования, мы увидим это особенно ясно.

## 1.1. Архитектура фон Неймана

Первые же попытки автоматизировать процесс вычислений в конце 40-х годов XX в. показали, что быстродействие определя-

ется не только и не столько тем, как быстро машина выполняет арифметические операции, сколько тем, как именно организован процесс «объяснения» машине, что именно вычислять.

Так, вычислительная производительность лаборанта, считающего на арифмометре, определяется не скоростью срабатывания колес арифмометра, а скоростью, с которой лаборант, соблюдая заданную последовательность (программу) действий, набирает числа и команды и записывает промежуточные результаты.

Для автоматизации управления процессом вычислений к арифметическому устройству пришлось добавить устройство программного управления и память для хранения программ и промежуточных данных. Наиболее общие принципы программного управления были сформулированы Джоном фон Нейманом. Кратко суть их в следующем: однородная адресуемая память хранит данные, используемые арифметическим устройством, и команды, оперирующие отдельными числами и выполняющиеся в определенной последовательности. Принципы эти используются уже более 60 лет: то, что мы называем «обычным универсальным процессором», в действительности — «фоннеймановская машина» [1, 3].

## **1.2. Этапы развития суперкомпьютерных технологий**

Примерно до конца 80-х годов XX в. удавалось строить суперкомпьютеры в рамках фоннеймановской архитектуры. Усложняя внутреннее устройство процессора, можно было заставить его выполнять примерно (или даже в точности) такие же последовательности машинных команд, что и «обычный» процессор, во много раз быстрее. Иными словами, фоннеймановская архитектура оставалась адекватной формой организации растущего числа транзисторов, пригодных для интеграции в единую вычислительную систему.

На этом этапе:

- программист почти не задумывался о том, для суперкомпьютера или для «обычного» компьютера он пишет программу;
- разработчик оборудования суперкомпьютера разрабатывал «суперпроцессор»;
- суперкомпьютер таким образом представлял собой обычную фоннеймановскую машину, оснащенную «суперпроцессором».

Сама возможность значительно ускорять процессор по сравнению с серийно выпускаемыми образцами, совершенствуя лишь его внутреннее устройство, объяснялась дефицитом транзисторов, используемых при изготовлении «обычного» процессора. Разработчики суперкомпьютеров имели в своем распоряжении гораздо большее число транзисторов, чем разработчики серийно выпускаемых машин, что позволяло им применять технические решения, просто недоступные для последних. «Суперпроцессор» работал быстрее «обычного» процессора, в конечном итоге, именно за счет того, что на его изготовление удалось потратить гораздо больше транзисторов.

Примерно в конце 80-х годов разработчики СБИС смогли разместить на одном кристалле кремния миллион транзисторов (это — примерно технологический уровень процессора Intel 80486 [67]). Имея в своем распоряжении такое количество «деталей», разработчики процессоров получили возможность ускорить практически все, что можно было ускорить в принципе. Конечно, попытки «добыть» дополнительное быстродействие на микроуровне продолжаются и сейчас — на кристалле микропроцессора Itanium-2, например, транзисторов уже миллиард [7, 67]. Но уже тогда, на миллионном уровне, стало очевидно, что многократного (в десятки и сотни раз) ускорения работы, выполняемой в среднем за один такт, не получить. Эпоха «супер-процессоров» кончилась — отныне каждый процессор является одновременно и «обычным», и «супер», существенной разницы нет.

Единственным способом изготовления суперкомпьютера стало изготовление *параллельной вычислительной системы*, т.е. машины, в которой многие процессоры совместно трудятся над одной задачей. Такие машины строились из серийно выпускаемых микросхем, в частности, микропроцессоров, но на оригинальной схемотехнической базе (шкафы, платы, линии связи) [66]. Этот технологический переворот можно считать *первой суперкомпьютерной революцией*.

На этом этапе:

- программист, чтобы получить суперкомпьютерный уровень быстродействия, был поставлен перед необходимостью писать специальные — параллельные — программы;

- разработчик оборудования суперкомпьютера переключился с создания «суперпроцессоров» на разработку средств объединения многих стандартных, серийно выпускаемых микропроцессоров в единую систему;

— таким образом, суперкомпьютер представлял собой уже не фоннеймановскую машину, а параллельную вычислительную систему.

Возможность получать суперкомпьютер с помощью разработки собственных средств интеграции стандартных микропроцессоров в единую систему существовала до тех пор, пока серийно выпускаемые средства интеграции, т. е. оборудование локальных сетей, были развиты сравнительно слабо, и для интеграции процессоров «на суперкомпьютерном уровне» не годились. По мере совершенствования оборудования локальных сетей и интерфейсов их подключения к серийно выпускаемым компьютерам и эта ситуация изменилась. Примерно в конце 90-х годов серийно выпускаемое сетевое оборудование позволило использовать в качестве суперкомпьютера обычную локальную сеть, компактно расположенную и оснащенную соответствующим программным обеспечением. Технический и экономический смысл разработки и изготовления собственных плат и шкафов для интеграции стандартных микропроцессоров в единую систему пропал. Подавляющее большинство суперкомпьютеров стало строиться по технологии ***кластеров выделенных рабочих станций***, т. е. собираться методами системной интеграции из готовых, выпускаемых серийно, компьютеров общего назначения, плат и коммутаторов локальных сетей [68]. Этот переворот будем называть ***второй суперкомпьютерной революцией***.

На этом этапе:

— для программиста, по сравнению с предшествующим этапом, ничего принципиально не изменилось, кроме того, что таких программистов стало многократно больше, как и самих суперкомпьютеров;

— разработчик суперкомпьютерного оборудования вынужден был стать системным интегратором — разрабатывать стало нечего, все используемые компоненты и средства их интеграции стали выпускаться крупными сериями;

— суперкомпьютер, в результате, из параллельной вычислительной системы на оригинальной схмотехнической базе стал ***кластером выделенных рабочих станций*** [2, 3].

Обратим внимание на значение слова «выделенных» в данном случае. Сама по себе идея использовать локальную сеть в качестве суперкомпьютера значительно старше второй суперкомпьютерной революции. В промежутке между первой и второй суперкомпьютерными революциями локальные сети бурно развива-

лись, в том числе в организациях, заинтересованных в высокопроизводительных вычислениях. Естественно, делались попытки использовать локальные сети общего назначения, например, во время ночных простоев, для параллельных вычислений. Такие локальные сети по своим коммуникационным способностям значительно уступали «настоящим» суперкомпьютерам, но на некоторых, не требующих особенно интенсивных коммуникаций между узлами, параллельные программы удавалось успешно выполнять. Именно тогда появилось само название «вычислительный кластер», и первые варианты программного обеспечения, позволяющего использовать локальную сеть в этом качестве. Это программное обеспечение было ориентировано на параллельное использование локальной сети в обоих режимах: в основном — как локальной сети общего назначения, и лишь иногда — как вычислительного кластера, т. е. оборудование **не выделялось** специально для использования в режиме вычислительного кластера. Вторая суперкомпьютерная революция, как мы только что видели, заключалась в том, что при необходимости **выделить** оборудование для использования исключительно в качестве вычислительного кластера это оборудование стало оформляться также в виде локальной сети. В отличие от кластеров предыдущего этапа, такие кластеры стали снабжаться совершенно особым программным обеспечением, часто — исключая использование узлов как машин общего назначения (на узле современного вычислительного кластера электронную почту не отправляют, и научные статьи к публикации не готовят).

Чтобы различать два способа организации параллельных вычислений на оборудовании локальных сетей и, соответственно, два принципиально разных способа организации программного обеспечения, «новые» кластеры стали называть кластерами выделенных рабочих станций, а старые, соответственно, **кластерами невыделенных рабочих станций**. Программные технологии организации кластеров невыделенных рабочих станций существуют и сегодня. Далее, если явно не оговорено противное, будем называть кластерами именно кластеры выделенных рабочих станций — продукт второй суперкомпьютерной революции.

Технология кластеров выделенных рабочих станций господствует в суперкомпьютерной отрасли, фактически, определяя ее облик, уже почти 10 лет. На подходе третья суперкомпьютерная революция. Правильнее было бы сказать, что она уже началась. Масштаб изменений, которые внесет ее завершение в мир техно-

логий параллельной обработки данных, будет беспрецедентным. Именно это мешает нам охарактеризовать ее суть «в двух словах», как было сделано в случае первой и второй суперкомпьютерных революций. Оставим этот увлекательный рассказ до части II книги, а пока остановимся на этой точке зрения: **сегодня (вчера?) весь мир считает, в основном, на кластерах.**

### **1.3. Способы объединения многих процессоров в единую систему**

Многопроцессорные вычислительные комплексы принято делить на *мультипроцессоры* — системы с общей, или *разделяемой* (между процессорами), памятью, и *мультикомпьютеры* — системы из самостоятельных компьютеров, каждый — со своей, локальной, памятью, объединенных коммуникационной сетью. Мультикомпьютеры чаще называют системами с *распределенной* (между процессорами) памятью, или *MPP-системами* (*Massively Parallel Processing*, массово-параллельная обработка), или просто системами без общей памяти. На данный момент такая классификация не может быть признана достаточной. Как класс мультипроцессоров, так и класс мультикомпьютеров подразделяются на подклассы, качественно различающиеся по способам интеграции процессоров. В итоге способы интеграции процессоров в единую вычислительную систему образуют не только иерархию, но и непрерывный спектр, от способов с наиболее сильными взаимосвязями между процессорами из класса мультипроцессоров до способов с наиболее слабыми взаимосвязями из класса мультикомпьютеров (рис. 1.1).

Рассмотрим спектр способов интеграции в традиционном порядке: сначала опишем крайние случаи, а затем заполним промежуток между ними. Для краткости здесь и далее будем говорить о качественно различных способах интеграции процессоров как о «сильных» и «слабых».

На самом «сильном» краю спектра в верхней части рисунка находятся *SMP-системы* (*Symmetric Multi Processing*). В этих мультипроцессорах вся имеющаяся в системе оперативная память общая, и вся она равно доступна всем процессорам.

Самый «слабый» край спектра в нижней части рисунка представлен *MPP-системами на базе сетей двустороннего обмена данными*. В этих системах для передачи данных из

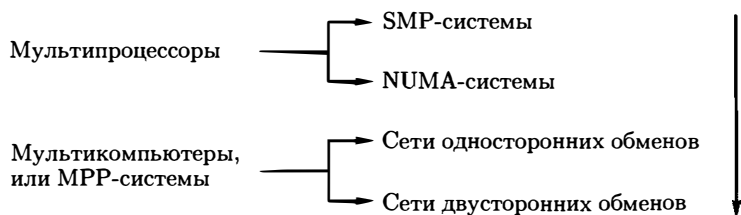


Рис. 1.1. Иерархия и спектр способов интеграции процессоров в вычислительную систему (вертикальная стрелка справа показывает направление ослабления взаимосвязей между процессорами)

памяти одного процессора в память другого необходимы специальные, явно записанные в программе действия на обоих концах канала. Тот, кому требуются «чужие» данные, должен выполнить запрос на прием данных, а тот, у кого эти данные хранятся — запрос на их передачу. Только при таком совместном желании обоих участников обмена данные могут быть переданы от одного процессора к другому (отсюда название: «сети двусторонних обменов»).

Теперь остановимся на промежуточных по «силе» решениях.

Второй по «силе» способ интеграции процессоров — **NUMA-системы** (*Non-Uniform Memory Access*). Это системы, в которых память общая, но не обязательно вся, а главное — не все области памяти доступны всем процессорам на равных правах. Общая память в такой системе поделена (обычно — на блоки). Каждый из таких блоков принадлежит конкретному процессору, т.е. обращения к нему из этого процессора происходят быстро. Все остальные, кроме «своего», блоки общей памяти также доступны процессору для прямой адресации обычными командами обращения к памяти, но выполняются эти команды гораздо медленнее, чем при доступе в «свой» блок. В зависимости от конкретной реализации, слово «гораздо» в предыдущей фразе может означать «вдвое медленнее», а может — и «в 300 раз медленнее».

Наконец, третий по «силе» способ интеграции процессоров представлен **МРР-системами на базе сети односторонних обменов**. В этих системах для передачи данных из памяти одного процессора в память другого достаточно записать соответствующий запрос в программе лишь одного процессора. Например, один процессор может «насильно» положить некоторые



данные в указанное им место памяти другого процессора. Другой процессор физически не участвует в этом обмене — всю работу делает его сетевой адаптер, получивший данные и запрос на их запись по сети от сетевого адаптера — инициатора обмена. В итоге, каждый процессор работает с памятью любого другого процессора примерно так же, как с собственным жестким диском, самостоятельно контролируя весь процесс передачи данных.

Краткое описание классификации способов, которыми процессоры интегрируются в единую вычислительную систему, завершено. Теперь посмотрим на нее глазами программиста.

Легко обнаружить, что перечисленные способы интеграции, действительно, образуют непрерывный спектр в смысле удобства записи и эффективности реализации доступа к «чужим» данным.

В случае SMP-системы нет самого понятия «чужих» данных. Все данные для всех процессоров «свои». В NUMA-системе доступ к «чужим» данным записать так же просто, как к «своим», но требуется учитывать в структуре программы, что доступ этот — гораздо более медленный. В MPP-системе на базе сети односторонних обменов доступ к «чужим» данным не просто гораздо более медленный, чем к «своим», но и записывается в программе специальным образом, в виде запроса на обмен, подобно запросу на обмен с жестким диском или другим устройством внешней памяти. Наконец, в MPP-системе на базе сети двусторонних обменов доступ к «чужим» данным не просто записывается специальным образом, но и требует «ответной любезности» от партнера — «хозяина» данных.

Естественно предположить, что этим четырём способам должны соответствовать четыре качественно различных способа записи доступа к «чужим» данным в прикладной программе, т.е. четыре семейства систем параллельного программирования — языков и/или библиотек. В дальнейшем будет показано, что это действительно так. Более того, способы записи доступа к «чужим» данным в программе, как и породившие их способы построения многопроцессорной вычислительной системы, бывают «сильными» и «слабыми». Лучше всего составлять программу способом, «сила» которого соответствует «силе» используемой аппаратуры, но так бывает далеко не всегда. «Слабые» способы легко реализуются на «сильной» аппаратуре, но не полностью используют ее возможности. «Сильные» способы реализуются на «слабой» аппаратуре с большим трудом, зачастую — весьма

неэффективно. Об этом мы подробно поговорим в главе, посвященной моделям и технологиям параллельного программирования.

Теперь посмотрим на классификацию способов интеграции процессоров в систему глазами разработчика оборудования.

SMP-система — наиболее удобный для программиста, но наиболее дорогой в технической реализации способ. Конечно, при таком способе программист делит между процессорами только работу, но не данные, причем делить работу можно очень мелкими частями (порядка десятков команд), что бывает очень удобно. Однако, помимо высокой стоимости, такие системы очень плохо масштабируются: даже в очень дорогих системах число объединяемых процессоров не дотягивает до сотни. Качество объединения часто оставляет желать лучшего: чем больше процессоров, тем более они склонны конкурировать за доступ к общей памяти, взаимно замедляя работу друг друга.

Отметим, что все популярные в последнее время многоядерные процессоры являются, с логической точки зрения, SMP-системами.

MPP-системы значительно дешевле в расчете на один процессор, прекрасно масштабируются (до десятков тысяч узлов), но на них труднее программировать. Делить между процессорами приходится не только работу, но и данные. Причем работу приходится «нарезать» крупными частями (порядка сотен тысяч команд и более), а передачи данных, расположенных в одном процессоре, но необходимых для вычислений в другом, приходится явно записывать в программу.

Вторая суперкомпьютерная революция усилила и закрепила господство этой технологии: с появлением возможности строить MPP-системы вообще без специальных аппаратных разработок ценовая пропасть между ними и SMP-системами сравнимого размера стала еще гораздо более широкой, чем раньше. Весь прогресс в удешевлении и росте доступности суперкомпьютеров, достигнутый второй суперкомпьютерной революцией, пришелся на долю MPP-систем.

Довольно часто узлом MPP-системы бывает не единичный однопроцессорный компьютер, а SMP-система сравнительно небольшого размера.

Еще совсем недавно средние позиции выстроенного нами спектра были заполнены слабо. Среди мультипроцессоров было мало NUMA-систем, и стоимость их зачастую практически не отлича-

лась от стоимости SMP-систем. Единственным преимуществом (кстати, очень важным) была значительно более высокая масштабируемость.

С другой стороны, подавляющее большинство MPP-систем были системами на базе сетей двустороннего обмена данными. За годы, прошедшие после второй суперкомпьютерной революции, именно в этих промежуточных позициях спектра наблюдался устойчивый и значительный прогресс. NUMA-системы росли в размерах и дешевели, оставаясь, тем не менее, суперкомпьютерами на оригинальной схемотехнической базе (то есть не кластерами). При этом в классе сетевого оборудования для кластеров выделенных рабочих станций стало появляться все больше решений, поддерживающих на аппаратном уровне односторонние обмены данными. Третья суперкомпьютерная революция, о которой пойдет речь в части II книги, скорее всего приведет к слиянию этих двух промежуточных классов. В самом деле, NUMA-систему можно рассматривать как «усовершенствованную» сеть односторонних обменов, в которой односторонний обмен запускается автоматически, на аппаратном уровне, при выполнении команды, адресующей «медленную», т. е. «чужую», память. Очевидно, существование этих двух родственных, но не равноценных промежуточных классов — NUMA-систем и сетей одностороннего обмена данными — возможно лишь в случае, если системы «менее удобного» класса гораздо дешевле и доступнее. До совсем недавнего времени это так и было. NUMA-системы можно было строить исключительно на основе оригинальной схемотехники уровня плат и шкафов, причем с применением дорогостоящих заказных СБИС, в то время как сети односторонних обменов были кластерами на базе серийно выпускаемого оборудования.

Изменения в архитектуре и технологии выпускаемых крупными сериями материнских плат, речь о которых пойдет в части II книги, привели, фактически, к стиранию этой грани. **NUMA-кластеры** уже появляются, и коммуникационного оборудования в них не больше, а меньше, чем в «обычных» кластерах, а значит — буквально завтра они станут дешевыми и доступными. Скорее всего, это приведет не только к слиянию двух промежуточных классов представленного спектра, но и к частичному поглощению вновь образовавшимся классом «обычных» кластеров на базе сетей двустороннего обмена.

## 1.4. Альтернативные архитектуры. Классификация Флинна

До сих пор молчаливо полагалось, что параллельная вычислительная система, в любом случае, строится из универсальных (фоннеймановских) процессоров. Это не всегда так. Например, векторный процессор — это также параллельная не фоннеймановская архитектура, но полученная не объединением в одну систему многих фоннеймановских процессоров, а другим способом — добавлением к универсальному процессору команд работы с векторами [3, 4]. Мы не будем сейчас специально обсуждать вопросы векторных вычислений. Однако отметим, что узел как SMP-, так и MPP-системы может быть и векторным процессором. Также, как отмечалось ранее, узел MPP-системы может быть SMP- или NUMA-системой, размер которой заметно меньше размера установки в целом.

Тот факт, что параллельная вычислительная система, вообще говоря, не обязана строиться на базе именно фоннеймановских процессоров, означает возможность (и необходимость) классификации таких систем, образно говоря, по «виду параллелизма». Таких классификаций существует много [3], и самую известную из них — классификацию Флинна — хотелось бы упомянуть. Эта классификация строится на понятии единственности или, наоборот, множественности одновременно обрабатываемых вычислительной системой потоков команд и данных. Поскольку имеется два типа потоков (команды и данные), и вариантов для каждого — также два (один поток или много), всего существует четыре варианта структуры вычислительной системы.

1. Один поток команд, один поток данных — *SISD* (*Single Instruction, Single Data*). Это — фоннеймановский компьютер.

2. Один поток команд, много потоков данных — *SIMD* (*Single Instruction, Multiple Data*). Это — векторный компьютер, процессор которого может, например, одной командой сложить покомпонентно два массива чисел.

3. Много потоков команд, много потоков данных — *MIMD* (*Multiple Instruction, Multiple Data*). В этот класс попадают все упомянутые нами выше многопроцессорные вычислители на базе фоннеймановских процессоров, независимо от способа объединения этих процессоров между собой — SMP, NUMA, MPP.

4. Много потоков команд, один поток данных — *MISD* (*Multiple Instruction, Single Data*). Некоторые исследователи компью-

терных архитектур считают, что отдельные, экспериментальные вычислительные установки, известные им, могли бы претендовать на включение в этот класс. Большинство авторов полагает, что этот класс пуст, и вряд ли будет заполнен в обозримом будущем. Иногда практикуемое отнесение к этому классу систолических массивов [3] и даже векторно-конвейерных структур, на взгляд автора, имеет характер некоторой «терминологической натяжки».

## **1.5. Простейшая модельная программа**

Рассказывать о составе и назначении программного обеспечения суперкомпьютеров лучше всего на конкретном примере. Выберем модельную программу, спланируем «на бумаге» ее параллельную реализацию, а затем — посмотрим, какого рода программное обеспечение нам потребуется, чтобы выполнить эту параллельную реализацию на суперкомпьютере вообще, и на кластере рабочих станций — в частности.

### **1.5.1. Решение двумерной краевой задачи для уравнения теплопроводности методом Якоби**

Физика задачи: рассмотрим однородный прямоугольный параллелепипед («кирпич»), к боковым граням которого плотно прислонены «утюги» бесконечной массы и заданной температуры. По мере прогрева «кирпича» в нем установится некоторое распределение температуры, которое нам и требуется найти. Будем рассматривать двумерное приближение задачи. Это значит, что кирпич предполагается очень высоким, и нас будет интересовать распределение температуры в его горизонтальном срезе, достаточно удаленном как от верхней, так и нижней граней. Для краткости будем называть эту задачу «задачей о прогреве кирпича». Подчеркнем, что ни физическая, ни вычислительно-математическая сторона этой прекрасно изученной задачи нас не интересует — мы рассматриваем ее как пример заданной и не обсуждаемой вычислительной процедуры, для которой надо построить параллельную реализацию [2].

Данная задача является краевой задачей для уравнения в частных производных (уравнения теплопроводности), которую можно приблизить задачей на равномерной четырехугольной

сетке. Сеточную задачу, в свою очередь, можно решать итерационным методом — мы выберем метод Якоби. Численный метод, к которому мы приходим, двигаясь упомянутым путем, очень прост.

Имеется двумерный массив  $F$ , значения элементов которого — нормализованные значения температуры в узлах сетки, «наброшенной» на срез кирпича. По краям массива находятся значения температуры утюгов — граничные условия. Они в процессе расчета не изменяются. Со всеми остальными значениями поступаем так:

- для каждой ячейки вычисляем новое значение как среднее арифметическое значений четырех ее соседей;

- после того, как новые значения вычислены для всех ячеек, заменяем старые значения новыми, попутно вычисляя максимум абсолютной величины отклонения новых значений от старых;

- повторяем эти два шага до тех пор, пока максимум абсолютной величины отклонения не станет меньше заданного порога малости.

Для упрощения наших модельных рассуждений исключим из алгоритма вычисление отклонений — будем просто выполнять фиксированное число итераций. Текст последовательной (для одного процессора) реализации данной вычислительной процедуры представим в виде:

```
include <stdio.h>
/**/
#define MX 640
#define MY 480
#define NITER 10000
#define STEPITER 100
    static float  f[MX][MY];
    static float df[MX][MY];
/**/
int main( int argc, char **argv )
{
    int i, j, n, m;
    FILE *fp;
/**/
    printf( "Solving heat conduction task
           on %d by %d grid\n", MX, MY );
    fflush( stdout );
```

```

/* Initial conditions: */
for ( i = 0; i < MX; i++ )
{
    for ( j = 0; j < MY; j++ )
    {
        f[i][j] = df[i][j] = 0.0;
        if      ( (i == 0)
                || (j == 0) ) f[i][j] = 1.0;
        else if ( (i == (MX-1))
                || (j == (MY-1)) ) f[i][j] = 0.5;
    }
}

/* Iteration loop: */
for ( n = 0; n < NITER; n++ )
{
    if ( !(n%STEPITER) )
        printf( "Iteration %d\n", n );
/* Step of calculation starts here: */
    for ( i = 1; i < (MX-1); i++ )
    {
        for ( j = 1; j < (MY-1); j++ )
        {
            df[i][j] = ( f[i][j+1] +
                        f[i][j-1] + f[i-1][j] +
                        f[i+1][j] ) * 0.25 - f[i][j];
        }
    }
    for ( i = 1; i < (MX-1); i++ )
    {
        for ( j = 1; j < (MY-1); j++ )
        {
            f[i][j] += df[i][j];
        }
    }
}

/* Calculation is done, F array is a result: */
fp = fopen( "progrex.dat", "w" );
for ( i = 1; i < (MX-1); i++ )
    fwrite( f[i]+1, MY-2, sizeof(f[0][0]),
            fp );
fclose( fp );
return 0;
}

```

### 1.5.2. Параллельная реализация «на бумаге»

Простейшая параллельная реализация очевидна. Поделим массивы F и DF на горизонтальные полосы (группы строк) равной высоты, по числу процессоров. Пусть каждый процессор обрабатывает свою полосу на каждой итерации (рис. 1.2).

Под словом «обрабатывает» здесь понимаются те строки, элементам которых процессор присваивает новые значения. Конкретный смысл слова «поделим» (строки между процессорами) зависит от того, имеется ли в системе общая память.

Если наша машина — SMP-система, то параллельная реализация построена; если это NUMA-система, то мы ее построили, при условии, что нам удастся расположить каждую полосу массивов F и DF в памяти, «своей» для обрабатывающего эти полосы процессора. В обоих этих случаях **весь** массив F, изображенный на рисунке, располагается в общей памяти, доступной **всем** процессорам. При этом каждый процессор, зная собственный номер, выполняет цикл обхода строк не всего массива, а только своей полосы: нулевой процессор модифицирует строки с 1-й по 4-ю, первый — с 5-й по 8-ю, и т. д. Пределы изменения переменной цикла по строкам у каждого процессора свои. Нулевая и 13-я строки массива содержат граничные условия и не обрабатываются. Обработка 12 строк поделена между процессорами поровну, каждому досталось по четыре строки.

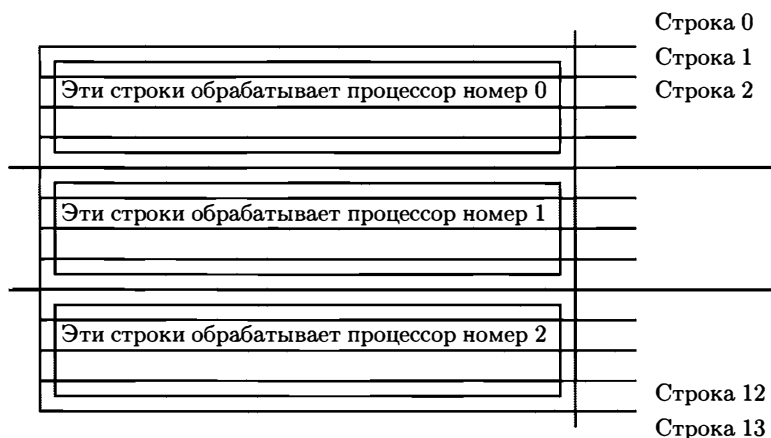


Рис. 1.2. Массив F из 14 строк, поделенный между тремя процессорами (строки 0 и 13 — граничные условия)



Сейчас нас больше интересует случай, когда расчет выполняется на МРР-системе (например, на вычислительном кластере). Общей памяти у процессоров кластера нет, следовательно, нет и единого массива  $F$  (точнее, он есть, но не в тексте программы, а только в голове программиста). Вместо общего для всех процессоров массива  $F$ , у **каждого** процессора теперь имеется **свой** массив  $F$ , размером с «доставшуюся» этому процессору полоса. Каждый процессор проходит этот массив целиком — пределы изменения переменной цикла по строкам у всех процессоров одни и те же. Обработка каждого из таких частичных массивов в каждом процессоре должна быть построена так, чтобы к концу расчета во всей этой совокупности частичных массивов оказались в точности те же значения, которые оказались бы в едином массиве  $F$  при расчете на одном процессоре или на системе с общей памятью. Например, если сначала выполнить расчет на одном процессоре и записать итоговый массив  $F$  в файл на диске, а затем повторить расчет на четырех процессорах и записать в файл на диске друг за другом содержимое массивов  $F$  каждого из процессоров, в порядке их номеров, то файлы должны получиться идентичными.

В этом случае задачу построения параллельной реализации нельзя считать законченной, поскольку непонятно, что делать с краями полосы массива  $F$ . В самом деле, вычисляя первую и последнюю строки полосы, процессор, вообще говоря, вынужден воспользоваться крайними строками соседних полос, а эти строки расположены на соседних процессорах. Следовательно, необходимо отводить полосы с «запасом» в одну строку в начале и одну в конце, а перед началом каждой итерации предусмотреть в программе передачу данных из краев полосы каждого процессора в «запасные» строки соседних процессоров, чтобы каждый

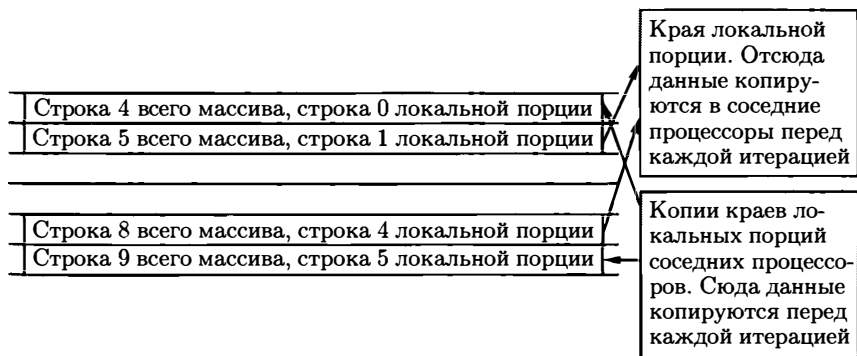


Рис. 1.3. Локальная порция массива  $F$  на первом процессоре

из процессоров располагал «свежей» копией «соседских» краев, насчитанной на прошлой итерации. Разумеется, при записи окончательных результатов счета на диск «запасные» строки записывать не следует (рис. 1.3).

Для нулевого и последнего процессоров, не имеющих «соседа сверху» и «соседа снизу», соответственно роль копий соседских краев выполняют строки граничных условий. Их не надо ниоткуда копировать, поскольку они в процессе расчета не изменяются.

Теперь параллельную реализацию «на бумаге» можно считать построенной и для случая кластера. Далее будем, если явно не оговорено противное, говорить только о кластерах (точнее, об MPP-системах).

### **1.5.3. Параллельная реализация на практике**

Попытаемся реализовать на практике то, что так просто и логично выглядит на бумаге. С одной стороны, имеется кластер, например, из восьми процессоров. Это компьютеры, каждый со своей ОС, и они связаны сетью. С другой стороны, имеется понимание, какая программа должна выполняться на каждом из них, и общее представление о том, как (на содержательном уровне) эти программы должны обмениваться данными друг с другом.

Означает ли это, что мы должны написать восемь исходных текстов, разложить по узлам кластера восемь исполняемых модулей, а затем, быстро перебегая от клавиатуры к клавиатуре, запустить все это на восьми компьютерах? И как эти исполняемые модули потом свяжутся друг с другом для обмена данными? Вопрос риторический.

Прежде всего, разберемся с гипотетической необходимостью написания восьми исходных текстов. Некоторые системы параллельного программирования предусматривают возможность написания отдельных исходных текстов программы для разных узлов, но в большинстве случаев все же поступают иначе.

Программисту предлагается написать текст программы для  $i$ -го узла, где значение  $i$  станет известным лишь во время выполнения программы. В данном случае это не только не трудно, но и весьма желательно. С одной стороны, тексты программ для всех узлов, очевидно, очень похожи, с другой — не хотелось бы что-то менять в тексте, если мы захотим выполнить программу не на восьми, а на десяти или сорока узлах кластера.

Написанная нами программа для обобщенного узла номер  $i$  называется *ветвью параллельной программы*, а вся совокупность ветвей, каждая из которых запускается на своем узле — *параллельной программой*. Часто для краткости параллельной программой называют именно ее ветвь. Мы и сами будем иногда так поступать, хотя, строго говоря, это и не вполне верно.

Запуск указанного пользователем количества ветвей параллельной программы на соответствующем количестве узлов выполняется из единой точки, например, из самого первого узла, или вообще из специальной управляющей машины, не являющейся узлом. Довольно часто команда запуска — это командный файл (скрипт), который просто выполняет команду дистанционного (по сети) запуска исполняемого файла ветви на каждом из задействованных узлов кластера.

Однако просто запустить ветвь дистанционно на указанном узле недостаточно. В процессе запуска необходимо выполнить еще, как минимум, два действия:

- каким-то образом сообщить каждой из запущенных ветвей ее порядковый номер (то самое значение  $i$ ) и общее число запущенных ветвей;

- позаботиться о том, чтобы запущенные ветви могли пересылать друг другу сообщения, пользуясь номерами в качестве адресов.

Рассмотрим простейший (и довольно редко применяемый, но вполне работоспособный) вариант решения первой из проблем.

В большинстве операционных систем (и уж тем более — в Unix, обычно используемой на узлах вычислительного кластера) имеется понятие командной строки (то, что передается при запуске исполняемого файла в качестве аргументов в функцию `main`). Договоримся, что упомянутый выше скрипт — команда запуска — добавляет к имеющимся аргументам командной строки каждой ветви еще два, представляющих собой собственный номер ветви и общее число запущенных ветвей. Зная об этой договоренности, некоторая инициализирующая функция, вызванная пользователем в начале работы ветви, могла бы получить эти значения и запомнить внутри себя, попутно скорректировав командную строку, которая в программе, возможно используется. Например, в библиотеке MPI это обращение к инициализирующей функции выглядит примерно так:

```
int main( int argc, char *argv )  
{
```

```
.....  
MPI_Init( &argc, &argv );  
.....
```

Когда программа пользователя обратится к другой библиотечной функции — «узнать собственный номер» — запомненное таким образом число будет ей сообщено. Например, в MPI:

```
MPI_Comm_rank( MPI_COMM_WORLD, &my_number );
```

```
// my_number теперь содержит номер ветви.
```

Конкретный способ передачи значений, на самом деле, не важен. Интересующие нас два числа могут передаваться через командную строку, через временные файлы или как-то еще, в командной строке дополнительные параметры могут быть первыми или последними. Важно здесь то, что ветвь параллельной программы должна быть запущена с соблюдением некоторых правил, причем правила эти соблюдаются на двух сторонах: на стороне команды запуска и на стороне библиотеки функций, которыми пользуется ветвь параллельной программы.

Пусть, например, у нас имеются два варианта библиотеки MPI, изготовленных разными авторами, в первом из которых служебные параметры решено было добавлять в начало командной строки, а во втором — в конец (или вообще передавать совсем иным способом). Тогда бессмысленно собирать исполняемый файл ветви с библиотекой от первого варианта, а запускать его — командой запуска от второго. То же касается и передачи более сложно организованных данных, которые необходимы для начальной настройки обмена сообщениями между ветвями по ходу счета. Конкретных способов организации и передачи таких данных может быть много, и программист вовсе не обязан их знать, но передача этих данных из команды запуска и трактовка переданных данных внутри программы должны быть согласованы.

Комплект из команды запуска параллельной программы на счет и согласованной с ней библиотеки функций, к которым обращается ветвь параллельной программы, чтобы общаться с себе подобными, называется **системой параллельного программирования**. В действительности в систему параллельного программирования могут входить и другие компоненты, согласованные по внутренним интерфейсам с первыми двумя (специальные трансляторы, отладчики и т. п.), но два компонента, вынесенные в сформулированное здесь определение, обязательны.

Многие (хотя и не все) системы параллельного программирования подразумевают наличие доступа процессоров, на которых запускаются ветви параллельной программы, к общей файловой системе. В самом деле, предусмотренные, скажем, в Unix средства дистанционного запуска исполняемых файлов предусматривают передачу по сети запроса на запуск, но вовсе не самого исполняемого файла, который требуется запустить. Если исполняемый файл, который требуется запустить дистанционно, доступен на удаленной стороне через общую файловую систему, процесс запуска сильно облегчается.

Теперь, когда мы представляем себе в общих чертах как трудности, возникающие при необходимости запустить параллельную программу на кластере, так и подходы к их преодолению, можно приступить к формальному перечислению необходимых компонентов программного обеспечения вычислительного кластера.

## 1.6. Общий состав программного обеспечения вычислительного кластера

Приведем перечень компонентов программного обеспечения вычислительного кластера.

1. Каждый узел кластера должен быть оснащен *ОС узла*, включая необходимую *сетевую поддержку*, возможно, специального, применяемого только в вычислительных кластерах, сетевого оборудования и/или специальных сетевых протоколов.

2. Желательно, чтобы все узлы имели доступ к общей *сетевой файловой системе*.

3. Для записи передачи данных из узла в узел необходимо предусмотреть *коммуникационную библиотеку* (обычно — реализация MPI), часто опирающуюся на специальную сетевую поддержку.

4. С коммуникационной библиотекой должна быть интегрирована *система запуска параллельных программ*. Это — именно тот компонент, который «раскидывает» (автоматически, с помощью стандартных сетевых возможностей) команду запуска или сам исполняемый модуль по всем компьютерам, заботясь при этом о том, чтобы, начав выполняться, эти программы установили связь друг с другом, и могли обмениваться данными.

5. Если кластер используется как машина коллективного пользования, с системой запуска интегрируется *система очередей и контроля доступа*.

6. В свою очередь, над базовыми коммуникационными библиотеками типа MPI нередко надстраиваются, для удобства разработки прикладных программ, более высокоуровневые **библиотеки и языки параллельного программирования**.

7. Если, помимо всего прочего, предусмотрено использование кластера как части распределенного вычислительного ресурса, объединяющего многие, независимо администрируемые суперкомпьютеры, придется добавить еще и средства метакомпьютинга (Grid-технологий) [4]. Эти средства неизбежно затрагивают системы запуска, очередей и контроля доступа, но могут включать в себя и программные надстройки над коммуникационной библиотекой.

В дальнейшем изложении обратим внимание на то, какие из упомянутых технологий при построении кластера заимствуются в готовом виде из мира компьютерных технологий массового применения, а какие разрабатываются специально для вычислительных кластеров.

## **1.7. Программистская модель передачи данных**

При записи в программе обменов сообщениями между ветвями могут использоваться различные **программистские модели передачи данных**. Наиболее часто используется модель **рандеву**, при которой данные передаются от передатчика к приемнику тогда и только тогда, когда передатчик выдает запрос на передачу данных, а приемник, соответственно, на прием. Порядок срабатывания передатчика и приемника не существен — синхронизация, т. е. «ожидание опоздавшего», происходит автоматически, внутри коммуникационной библиотеки.

Помимо всевозможных высокоуровневых программных надстроек, облегчающих запись рандеву в программе, имеются также базовые модели, принципиально отличные от рандеву, например, модель односторонней передачи сообщений [2, 3, 4].

## **1.8. Технологии параллельной обработки данных без использования суперкомпьютеров**

Возможность организовать как можно более тесное взаимодействие между ветвями параллельной программы в процессе

совместной работы ветвей над решением единой задачи принципиально важна для программиста, и разработчики вычислительных кластеров стараются оснастить свои изделия как можно более быстрыми сетями. Однако, существует (хотя и не является преобладающим в процентном отношении) довольно обширный класс задач, при решении которых взаимодействие между ветвями параллельной программы вообще не требуется. Точнее, оно требуется в начале и в конце решения задачи, но объем его пренебрежимо мал по сравнению с объемом самого расчета. Это так называемый *вариантный счет*, когда необходимо выполнить очень много одинаковых расчетов с различными исходными данными, а интеграция полученных результатов сравнительно не сложна. Поскольку взаимодействие между ветвями не требуется, ветви могут выполняться в разном темпе, например, на однопроцессорных машинах с разным быстродействием. Интеграция результатов откладывается «на потом», когда посчитанных вариантов накопится много. В отличие от преобладающих суперкомпьютерных технологий, обозначаемых термином *НРС* (*High Performance Computing* — высокопроизводительные вычисления), технологии вариантного счета принято обозначать термином *НТС* (*High Throughput Computing* — вычисления с высокой пропускной способностью, высокоинтенсивные вычисления — более удачного русского термина пока нет). Возможно, что для таких расчетов нет нужды строить суперкомпьютер — достаточно разработать программные *средства отбора неиспользуемой вычислительной мощности* из сетей общего назначения. Это — совершенно особая область, которую мы не будем рассматривать в нашем курсе специально. Здесь речь идет о *задаче метакомпьютинга* [4] в ее наиболее общем виде. Периодически делаются попытки, наложив некоторые ограничения на дисциплину отбора мощности, все же воспользоваться технологиями метакомпьютинга для решения задач в дисциплине НРС, т.е. организовать выполнение «кластерных», параллельных программ не на выделенном оборудовании, а на системах программного отбора мощности. В общем виде это сложнейшая технологическая задача. Некоторые частные варианты ее решения могут быть просты и, до некоторой степени, полезны (одно из таких решений — *проект МВС-900* — будет рассмотрен далее [2, 3, 4]).

# НЕКОТОРЫЕ ОСНОВНЫЕ ПОНЯТИЯ АРХИТЕКТУРЫ ПРОЦЕССОРОВ И ОС

Материал данной главы не имеет суперкомпьютерной специфики. Такие сведения раньше принято было излагать во вводных курсах программирования для профильных специальностей.

Однако, как показывает опыт, традиции преподавания меняются. Подобно тому, как из программы современных автошкол постепенно исчезли разделы, посвященные самостоятельному ремонту и обслуживанию автомобиля, об «азах» устройства компьютерного «железа» в современных вводных курсах программирования преподаватели говорят все меньше, а студенты слушают об этом все менее охотно.

Современные технологии разработки программ позволяют большинству студентов сочетать вполне удовлетворительный для работы программистом уровень практических знаний с неоправданно абстрактным, поверхностным представлением об основах работы компьютера как такового.

Там, где требуется быстро и эффективно программировать в рамках устоявшихся технологий, это оправданно. Для разработчика банковских приложений знание о различиях между системами команд Pentium и Itanium — непозволительная роскошь и «засорение мозгов». Полезнее знать о потаенных особенностях последней версии какой-нибудь конкретной реализации C++.

Для разработчика суперкомпьютера, главная работа которого — создавать и помогать осваивать принципиально новые технологии использования оборудования, напротив, незнание того, что процессоры бывают с разной системой команд, а команды обращения к памяти — это совсем не то же самое, что команды работы с портами ввода-вывода, — совершенно непозволительная роскошь. Начнем с определения некоторых основных понятий, которые зачастую воспринимаются как почти гуманитарные эпитеты, в то время как в действительности они имеют строгий инженерный смысл.



## 2.1. Некоторые основные определения

*Архитектурой* называется функциональное описание объекта, т.е. описание того, как он «виден» пользователю.

Например, система команд процессора относится к его архитектуре, размер кэш-памяти — тоже (подробнее об этом — ниже), а технология изготовления кристалла, на котором выполнен микропроцессор — не относится. Это, в противоположность архитектуре, уже **реализация** процессора. Архитектура — **что** сделано, реализация — **как** сделано.

*Виртуальное* (нечто) — это нечто, не существующее, но представляющееся существующим.

Виртуальная сеть — так организованная дисциплина передачи данных, как будто бы это был отдельный сегмент сети, в то время как никакого отдельного сегмента в действительности нет.

*Прозрачное* (нечто) — это нечто, существующее, но представляющееся несуществующим.

При попытке пойти по ссылке на интересующий Вас сайт Интернета конкретная структура Интернета для Вас прозрачна и Ваши действия не зависят от того, расположен ли сайт физически в соседней комнате или в Австралии.

Замечание. Приведенные определения свойств виртуальности и прозрачности заимствованы автором по памяти из курса программирования, прочитанного в МГУ им. М.В. Ломоносова в 1975 г. профессором Э.З. Любимским. В профессиональной литературе эти термины употребляются именно в этом смысле, и ни в каком другом. С появлением Интернета словом «виртуальный» стало обозначаться практически все, относящееся к компьютерам. Еще большее число разнообразных трактовок смысла этого слова можно найти в словарях общей лексики. Тем не менее далее по тексту мы будем иметь в виду именно то значение этого слова, которое приведено в данном выше определении.

Для иллюстрации еще нескольких важных понятий нам потребуется базовое представление об устройстве компьютера на уровне системы команд. Цель его предстоящего обзора в том, чтобы сформулировать минимальный и исчерпывающий перечень возможностей оборудования, на котором строится все многообразие программистских технологий. Именно **исчерпывающий**. Важно понимать не столько то, что излагаемые ниже

возможности присущи аппаратуре, наряду с множеством других возможностей, сложных и загадочных, сколько то, что **ничего принципиально другого в аппаратуре нет**, и именно необходимость опираться на этот очень узкий набор базовых возможностей делает программное обеспечение таким, каким оно является. В качестве критерия полноты наших базовых представлений предлагается мысленный эксперимент. Начав с простейшего примера фоннеймановского процессора, будем постепенно обогащать его функции до тех пор, пока не станет ясно, что на имеющемся ункции набора можно построить ОС Windows (или Linux) в известном нам виде.

## 2.2. Пример системы команд: обработка данных в памяти

В 60-х годах XX в. в СССР выпускалась ламповая вычислительная машина М-20. Позднее она была переведена на транзисторную элементную базу и стала называться БЭСМ-3, после некоторой модернизации — БЭСМ-4 (а также М-220). Машина эта замечательна тем, что ее система команд, по крайней мере, в части, касающейся обработки данных внутри оперативной памяти, была исключительно простой и понятной. Именно это подмножество системы команд М-20 будет кратко рассмотрено в данном подразделе. При этом мы сосредоточимся на командах, выполняющих обработку данных как таковую. Оставим пока в стороне вопрос о том, как команды и данные попадают в память, а результаты вычислений из нее выводятся [5].

Автор полностью отдает себе отчет в том, что текст данного подраздела не является исчерпывающим пособием по программированию в машинных кодах. Для более подробного изучения предмета применительно к современным процессорам рекомендуется обратиться, например, к [6].

Также не следует рассматривать работу [5] как рекомендуемое сегодняшнему читателю введение в программирование. В настоящем тексте эта работа упомянута только как источник сообщаемых ниже сведений о системе команд М-20.

**Оперативная память** состоит из *слов*, занумерованных от 0 подряд. В слове 45 разрядов, занумерованных справа от 1. Из слова номер 0 всегда читается двоичный 0. Номер ячейки в памяти называется ее **адресом**.

Слово может хранить число или команду.

*Формат числа:*

П 3 EEEEEEE M..... M

45 44 43 37 36 1 (номера разрядов).

M — мантисса,

E — код порядка, (64+порядок),

3 — знак числа (0-плюс),

П — признак (арифметического смысла не имеет, при арифметических операциях наследуется).

Число имеет значение:  $0.M \cdot 2^{**}(E-64)$ , со знаком 3.

Видим, что речь идет о числах с плавающей точкой, причем в прямом коде. Последнее означает, что машинное представление чисел «А» и «-А» отличается только битом знака числа.

*Формат команды:*

ППП КKKKKK A1.. A1 A2.. A2 A3..A3

45 43 42 37 36 25 24 13 12 1 (номера разрядов).

ППП — 3-битный код признаков,

KKKKKK — 6-битный код операции,

A1, A2, A3 — 12-битные адреса операндов.

Код операции и адреса — неотрицательные, целые двоичные числа. Обычно первые два адреса указывают исходные данные, третий — результат. Например, в команде сложения содержимое слова с адресом A1 складывается с содержимым слова с адресом A2, а результат помещается по адресу A3. При этом к адресу, указанному в соответствующем поле команды, прибавляется значение индекс-регистра, если соответствующий этому адресу бит кода признаков равен 1. Для занесения значений в индекс-регистр служат специальные команды.

Архитектурно в процессор входят три целочисленных регистров:

- счетчик команд;
- триггер «омега»;
- индекс-регистр.

Замечание. **Регистром** называют отдельную, не входящую в оперативную память, ячейку памяти, а **триггером** — регистр, состоящий из единственного двоичного разряда.

Содержимое счетчика команд всегда равно адресу следующей команды, т.е. команды, на которую надо перейти, выполнив текущую команду. Если текущая выполняемая команда — не команда перехода, то при выполнении команды содержимое

счетчика команд автоматически увеличивается на единицу, т. е. следующая команда берется из следующего слова памяти. Команда перехода заносит в счетчик команд указанное в команде значение, что означает: «продолжить выполнение программы с указанного места (памяти)», или, другими словами, «перейти в указанное место».

Триггер «омега» — дополнительный признак результата выполнения операции, например, признак того, что при выполнении сложения получилось отрицательное число.

Вопросы на понимание:

— сколько слов оперативной памяти может быть в этой машине?

— сколько команд может быть в этом процессоре?

— какова разрядность счетчика команд?

— можно ли выполнить число?

— можно ли прибавить единицу к команде?

Ширина полей команды кратна трем разрядам, поэтому будем использовать при записи значений этих полей восьмеричную систему счисления (восьмеричная цифра — это как раз три разряда). Нетрудно показать, что при такой группировке разрядов значение записываемого целого числа не меняется: 001 101 010 в двоичной системе и 152 в восьмеричной — одно и то же число, поскольку восьмерка есть степень двойки.

Основные команды (в восьмеричном виде):

01 — сложение,

02 — вычитание,

03 — вычитание модулей,

04 — деление,

05 — умножение,

44 — квадратный корень.

Некоторые команды, помимо вычисления результата, изменяют значение триггера «омега». Например, при выполнении команд группы сложения (сложение, вычитание) триггер «омега» становится равным 1, если результат не отрицателен, иначе — равным 0.

У команд группы умножения  $\text{омега}=1$ , если порядок положителен.

06 — сложение порядка с адресом,

46 — вычитание порядка из адреса,

13 — сложение адресных частей команд,  
33 — вычитание адресных частей команд,  
омега = 1, если произошло переполнение или заем из несуществующего старшего разряда.

Эти команды предназначены для вычисления команд, которые сами же потом будут выполняться. Система команд этой машины столь аскетична, что для организации, например, сложных циклов или косвенной адресации (такой, которая в С записывается унарным оператором «\*») приходилось писать программы, модифицирующие (вычисляющие) собственный код. С развитием в современных процессорах режимов адресации писать программы таким образом, по крайней мере, в кругу приличных людей, стало не принято. Но на описываемой машине поступать приходилось именно так. И отнюдь не разработчикам вирусов.

00 — пересылка слова из первого адреса в третий,  
56 — безусловный переход по второму адресу с пересылкой из первого адреса в третий,  
76 — переход по омега, равному 0,  
36 — переход по омега, равному 1,  
77 — стоп (остановить процессор),  
52 — засылка адреса в индекс,  
72 — засылка кода в индекс.

Из всего изложенного здесь принципиально важно то, что процессор оснащен некоторым набором арифметических и логических операций, а также командами условного перехода, т.е. возможностью выбрать дальнейший путь выполнения программы в зависимости от промежуточных результатов вычислений.

Пока можно сделать два важных вывода:

1) **алгоритмически** приведенных скромных возможностей достаточно для реализации любой вычислительной процедуры, которая может быть записана, например, на С или Фортране;

2) **технологически** мы еще очень далеки от понимания того, как работает, например, ОС Windows. Мы не представляем себе, как программа попадает в память и получает управление, как организуется параллельное выполнение нескольких программ на одном процессоре, как управляются внешние устройства, и т.п.

Для начала все же приведем пример программы, вычисляющей абсолютную величину значения  $A*x + B*y$ .

Для начала требуется **распределить память**. Договоримся, что:

А имеет адрес 100,  
В имеет адрес 101,  
Х имеет адрес 102,  
У имеет адрес 103.

Результат необходимо поместить по адресу 104.

В качестве рабочих ячеек будем использовать слова со 105 по 107.

Сама программа будет располагаться в памяти по адресу 110.

Теперь можно **писать программу**:

```
110: 0 05 0100 0102 0105   A*x => 105
111: 0 05 0101 0103 0106   B*y => 106
112: 0 01 0105 0106 0104   105+106 => 104
113: 0 36 0000 0115 0000   если «омега»= 1, идти на конец
114: 0 02 0000 0104 0104   0-104 => 104 (из 0-й ячейки
                           читается 0)
115: 0 77 0000 0000 0000   стоп
```

Как это все работало:

На пульте набирали (тумблерами, в двоичной системе счисления) адрес ввода, клали колоду перфокарт в устройство считывания, и нажимали кнопку «ввод». Потом на пульте набирали адрес первой команды, нажимали кнопку «пуск» и ждали команды «77». Конечно, было предусмотрено программное управление внешними устройствами (здесь мы это опустили).

### 2.3. Основные сложности и пути их преодоления

Из изложенного в подразд. 2.2 видим, что работать так крайне трудно. Из совершенно очевидных шагов по преодолению этих трудностей выведем основные понятия архитектуры операционных систем, которые остались в принципе неизменными с тех самых пор. Каковы же эти трудности?

1. *Очень трудно программировать.* Попробуйте написать  $a*x^{**2}+b*y^{**2}+c*z^{**2}$ . И основная трудность даже не в том, что надо выписывать коды операций — к этому привыкаешь. Трудность в том, что надо указывать адреса переходов, а они «плывут» при малейших изменениях в программе. В самом деле, если в программе обнаружена ошибка, для исправления которой необходимо добавить хотя бы одну команду, адреса всех команд,

следующих за добавленной, увеличатся. Чтобы ранее написанная программа продолжала работать, надо подправить значения адреса во всех командах перехода, адресующих переместившуюся часть программы. Для этого адреса этих команд надо помнить. Но ведь при исправлениях они тоже меняются!

Для решения этой проблемы, борьба с которой «вручную» отбирала львиную долю рабочего времени тогдашних программистов, были придуманы первые трансляторы — *ассемблеры* или, как их тогда называли, «автокоды один в один». Программа на этом языке представляла собой символическую запись последовательности команд, в которой вместо кодов команд присутствовали их символические названия, а вместо адресов — имена ячеек в виде идентификаторов. В процессе трансляции ассемблер сам вычислял, какому идентификатору какой адрес соответствует. Программирование на языке ассемблера по-прежнему требовало знания системы команд на уровне битовых полей, но было во много раз более легкой работой, чем непосредственное кодирование, как в приведенном выше примере.

Потом появились трансляторы алгоритмических языков высокого уровня. Первым не зависящим от системы команд конкретной машины языком программирования был Фортран. Язык С — гораздо моложе.

Одновременно эволюционировали сами системы команд. Одно из важнейших направлений эволюции — избавление от необходимости модифицировать программу по мере ее работы. Для этого требовалось усложнять и разнообразить способы адресации. Например, в БЭСМ-6 был предусмотрен уже не один, а 16 индексных регистров. Тем не менее, важно понимать следующее.

*Все системы команд процессоров общего назначения избыточны и алгоритмически эквивалентны, т. е. не может быть вычислительной процедуры, выражимой в системе команд одного процессора и не выражимой в системе команд другого.*

2. *Невероятно трудно запускать программу.* Пользуясь современными операционными системами, мы привыкли непринужденно оперировать множеством окошек на экране, которые все открыты и все активны. Для запуска программы достаточно «кликнуть» мышью пиктограмму ее исполняемого файла. А как это получается? Простейший ответ: «Это получается, «потому что есть операционная система». А у нее получается?

3. *Совершенно не понятно, как организован ввод-вывод.* Это объяснить проще всего, так что с этого и начнем.

### 2.3.1. Общее представление об управлении внешними устройствами

С точки зрения программы, любое внешнее устройство — это набор **регистров**, обладающих, помимо способности хранить и/или выдавать записанные в них данные, некоторыми специальными свойствами. Рассмотрим такое простейшее внешнее устройство, как Com-порт. Предположим для конкретности, что извне к нему подключен алфавитно-цифровой терминал.

Замечание. Алфавитно-цифровым терминалом когда-то называли устройство в виде телетайпа или экрана с клавиатурой, выполняющее те же функции, что программа HyperTerminal в Windows.

Com-порт состоит из двух **каналов: вывода и ввода**. Каждый из каналов управляется двумя регистрами: **регистром данных** и **регистром состояния**.

Регистр данных имеет ширину, например 8 бит, простейший регистр состояния состоит из одного бита.

Рассмотрим **работу канала вывода**.

В начальный момент регистр состояния содержит «1», что означает «линия вывода свободна». При записи в регистр данных некоторого числа единица в регистре состояния гаснет (линия вывода занята передачей), а само число передается по линии вывода на терминал. Если это число — код буквы «А», на терминале в текущей позиции курсора появится буква «А». Это будет означать, что вывод символа из регистра данных завершен, и в регистре состояния снова появится «1» — линия свободна для вывода следующего символа.

**Канал ввода работает «зеркально»:** появление «1» в регистре состояния означает, что линия положила в регистр данных некоторое значение (код клавиши, нажатой на клавиатуре терминала), которое программа теперь должна забрать. При чтении программой регистра данных ввода единица в регистре состояния погаснет, и снова появится там лишь при поступлении с клавиатуры очередного символа.

Осталось наполнить конкретным содержанием понятия «запись в регистр» и «чтение из регистра». Очевидно, это какие-то команды процессора, аналогов которым мы в нашем беглом обзоре системы команд не видели. В разных системах команд эта проблема решается одним из всего двух способов (иногда — обоими):



1) регистры внешних устройств (обычно их называют *портами*, чтобы не путать с регистрами общего назначения в составе процессора) занумерованы, и для обмена данными между ними и процессором предусмотрены специальные команды. В нашем примере процессора это могли бы быть, например, команды: «записать содержимое слова по адресу А1 в порт, номер которого — А3», и, соответственно, «прочитать содержимое порта номер А1 в слово по адресу А3». (Замечание. «Могли бы быть» — не означает «действительно были». Здесь и далее, формулируя те или иные расширения в системе команд процессора, не будем следовать реально существовавшей системе команд М-20);

2) часть адресов памяти не используется для адресации именно памяти, а используется для доступа к портам внешних устройств. Свойство ячейки памяти с определенным адресом быть, в действительности, не ячейкой памяти, а портом некоторого внешнего устройства, задается, грубо говоря, при изготовлении машины аппаратно. Порты просто «маскируются» под слова памяти, что позволяет работать с ними обычными процессорными командами. Например, если в команде сложения указать в качестве адресов слагаемых адреса слов памяти, содержащих, соответственно, код буквы «А» и единицу, а в качестве адреса результата — адрес «слова памяти», которое на самом деле — регистр данных терминальной линии вывода, на терминале появится буква «В».

В процессоре “Pentium”, например, применяются оба способа, а в выпускавшихся еще недавно процессорах “Alpha” — только второй.

Более сложные внешние устройства, такие, как сетевые адаптеры или контроллеры дисков, включают в себя большее количество портов с более сложной логикой их обслуживания, но практически не отличаются от разобранного только что тривиального примера.

Замечание о видах памяти в многопроцессорных вычислительных системах. Теперь наше представление об алгоритмике системы команд процессора стало более или менее замкнутым — мы можем представить себе, как именно (конечно, в общих чертах, а не в деталях) превращается в процессорные команды и затем выполняется любая программа, включающая в себя не только внутреннюю переработку данных, но и ввод-вывод. Вернемся немного назад и посмотрим, как выглядят с этой точки зрения рассуждения, приведенные в подразд. 1.3. Говоря, что процессо-

ру доступна некоторая память, имеем в виду, что существуют такие значения адресов, использование которых в качестве адресных полей в процессорных командах приведет к адресации слов именно этой памяти. Как выглядит в этом смысле, например, MPP-система? Общей памяти в ней нет. Построим мысленно кластер из двух узлов, соединив их Com-порты нуль-модемным кабелем. Такое соединение означает, что данные, выдаваемые в Com-порт одного узла, поступают на ввод Com-порта второго узла, и наоборот. Легко видеть, что в этой системе способы доступа к «своей» и «чужой» памяти действительно радикально различаются. Для доступа к «своей» ячейке достаточно использовать ее номер в качестве адреса в любой процессорной команде. Для доступа к «чужой» надо, во-первых, чтобы процессор, находящийся по другую сторону линии связи, выдал содержимое этой ячейки в линию, «просовывая» его, байт за байтом, через регистр данных линии вывода, и не забывая правильно реагировать на значение регистра состояния вывода. Во-вторых, нужно «подобрать» это значение на своей стороне линии, «вынув» его, байт за байтом, из регистра данных ввода, по мере появления единиц в регистре состояния ввода. Следовательно, в отличие от обращения к своей памяти, требуются согласованные действия на двух сторонах, и действия эти — сложные и длительные, сводящиеся к выполнению особых, часто очень длинных последовательностей команд, предназначенных исключительно для управления коммуникационным оборудованием.

В терминах классификации, приведенной в подразд. 1.3, получен мультимикомпьютер на базе сети двусторонних обменов. Ничего другого мы получить и не могли, поскольку стандартный адаптер Com-порта односторонних обменов не поддерживает, т. е. «не умеет» ничего делать без ведома «своего» процессора.

### 2.3.2. Программа-диспетчер

Очередная трудность, которую требуется преодолеть, чтобы проложить дорогу к принципиальной возможности реализации ОС типа Windows —упрощение запуска программ на счет. Как работает программа, уже находящаяся в памяти, включая ввод и вывод, мы теперь в принципе понимаем, а вот как она туда попадает?

Очевидно, в результате «холодного пуска» — с перфокарт (в описанной выше М-20), или из микросхемы постоянного запо-

минающего устройства (в современных машинах) в оперативную память попадает лишь очень простая и короткая программа — **начальный загрузчик**. Затем она читает с диска более сложную программу, и выполняет команду перехода на ее начало. Попав в память, эта прочитанная с диска программа остается в ней постоянно, выполняя запросы пользователя на чтение с диска и запуск (в оставшейся части памяти) других программ. В зависимости от сложности упомянутой программы, такие запросы пользователя на запуск других программ могут быть оформлены как набор команд на клавиатуре, как «кликание» мышью пиктограммы или как-то еще. Такую программу будем (пока) называть **программой-диспетчером**. Было бы логично включить в состав диспетчера также набор стандартных подпрограмм, которыми могли бы располагать программы пользователей для управления внешними устройствами. Например, полезна была бы подпрограмма «выдать в com-порт заданную строку символов». Для общения программ пользователей с устройствами, более сложными, чем com-порт, такая **базовая библиотека ввода-вывода** просто жизненно необходима. Ведь устройства ввода-вывода могут быть не просто сложными, но и разными (на уровне конкретных регистров управления), а базовая библиотека могла бы скрыть всю эту утомительную специфику от прикладного программиста.

### 2.3.3. Прерывания

На машине, оснащенной программой-диспетчером и базовой библиотекой ввода-вывода, уже можно работать, но это еще далеко не Windows. Как современные операционные системы обеспечивают псевдопараллельное выполнение многих программ? Хочется привычно ответить: «Процессор выполняет несколько программ короткими порциями, постоянно переключаясь с одной на другую». Как это выглядит конкретно, в командах? Существуют ли команды типа «переключиться на другую программу», и, если да, то что будет с программой-диспетчером, если пользователь запустит программу, в которой таких команд нет? Наконец, что такое «с точки зрения» процессора эти самые «разные программы»? Пока мы видели, что процессор в состоянии выполнять ровно одну программу.

В действительности, никаких «команд переключения» в процессоре нет, а «разные программы» существуют лишь в голове

пользователя. То, что для пользователя выглядит как выполнение нескольких программ одновременно, для процессора есть выполнение единой программы, единой последовательности команд, в строгом соответствии с принципами фон Неймана.

Здесь есть серьезная проблема. Нам известно, что процессор способен менять порядок выполнения команд в результате выполнения команд перехода, условного или безусловного. Таким образом действительно можно «переключать» процессор между несколькими последовательностями команд, расположенными в различных местах памяти, про которые пользователю удобно думать, что это разные программы. Требуется лишь, чтобы в каждой из этих «разных программ» были достаточно часто расставлены такие директивы переключения, скорее всего, не непосредственного, а с помощью программы-диспетчера. Но это требование не реалистично — всегда найдется программист, который напишет программу безо всяких переключений. Если такая программа «зависнет», программа-диспетчер никогда не получит управления.

Для решения этой (и не только) проблемы в процессоре аппаратно реализован механизм ***прерываний***. Прерывание — это принудительный (никак не записанный в программе) переход по определенному адресу в результате наступления некоторого события. Событие может быть **внутренним** (для процессора) — например, деление на 0, или **внешним** — например, нажатие клавиши на клавиатуре. Практически всегда процессор снабжается специальным источником внешних прерываний — таймером, поставляющим, независимо ни от чего, внешние события, например, 1000 раз в секунду.

Переход по определенному адресу при прерывании сопровождается аппаратным сохранением (также по определенному адресу) того минимального объема сведений о состоянии процессора, который необходим, чтобы из прерывания можно было вернуться, а прерванная программа «ничего не заметила». Например, абсолютно необходимо сохранение счетчика команд в момент прерывания — иначе неизвестно, куда возвращаться.

Программа, находящаяся по адресу обработки прерывания, называется ***обработчиком прерывания*** и, конечно же, является частью нашей программы-диспетчера. Скорее всего, она тоже довольно многое прячет на входе и восстанавливает на выходе, но уже программно.

Как правило, внешние устройства генерируют прерывания, устанавливая в «1» бит готовности в том или ином регистре со-

стояния. Именно прерывания (от таймера и внешних устройств) и дают возможность программе-диспетчеру переключать процессор между разными программами теми самыми «мелкими порциями». Важно понимать, что прерывания в большинстве своем прозрачны для программы пользователя — результат ее выполнения никак не зависит от того, сколько и каких прерываний, например, от таймера произошло во время счета, и в каких именно местах программу временно останавливали, переключая процессор на другую. Логически программа выполнялась так, как если бы она выполнялась непрерывно на отдельном процессоре.

Способность внешних устройств генерировать прерывания при завершении передачи очередной порции данных позволяет организовать параллельную (не псевдопараллельную, а действительно параллельную в терминах астрономического времени) работу процессора и внешних устройств. В самом деле, рассмотрим работу com-порта по передаче большого массива данных из процессора во внешнее устройство — например, на принтер. Com-порт может передавать данные со скоростью примерно 16 кб/с. Это означает, что от записи очередного символа в регистр данных до появления готовности порта к записи следующего символа проходит немногим более 60 мкс. Процессор с частотой 3 ГГц способен выполнить за это время до 360 тыс. команд (считая по 2 команды за такт). Если программа-диспетчер, выдающая данные на принтер, будет после записи каждого символа в регистр данных вставать в цикл ожидания готовности, опрашивая регистр состояния, ей придется выполнить в этом цикле ожидания как раз эти 360 тыс. команд. Никаких других команд в это время процессор выполнить уже не сможет.

С другой стороны, программа-диспетчер могла бы поступить иначе. Записав очередной символ в регистр данных, она могла бы предоставить процессор какой-нибудь программе пользователя. Когда, по прошествии примерно 60 мкс, передача символа через com-порт завершится, произойдет прерывание. В этот момент программа-диспетчер снова получит управление и сможет записать в регистр данных следующий символ. Обработка прерывания, включая запись очередного символа в регистр данных, занимает гораздо меньше времени, чем 60 мкс. Программа пользователя, во время работы которой происходит такая печать файла в фоновом режиме, скорее всего, даже не очень сильно замедлится по сравнению со случаем, когда никакой печати нет.

Такое совмещение работы процессора и внешних устройств во времени широчайшим образом используется в операционных системах, и возможно оно именно благодаря наличию в аппаратуре процессора механизма прерываний.

Оснатив программу-диспетчер правильно настроенным аппаратом обработки прерываний и на этой основе возможностью псевдопараллельного выполнения прикладных программ, каждая из которых написана как последовательная, мы вправе назвать, наконец, такую программу-диспетчер *операционной системой*. Теперь она уже может (в принципе) делать все, что делает Windows или Linux. За двумя небольшими исключениями. Наша операционная система (очень похожая, кстати, на MS DOS по набору принципиальных возможностей) не имеет аппарата виртуальной памяти и не обеспечивает защиту себя и прикладных программ друг от друга. Эти две недостающие возможности строятся на общем фундаменте, что позволяет нам рассмотреть их вместе.

#### 2.3.4. Защита памяти и многорежимность процессора

Любая прикладная программа, выполняющаяся под управлением операционной системы в том виде, в каком мы ее сейчас понимаем, может разрушить саму себя, операционную систему или любую другую программу. Нереально построить защиту от этого без дополнительных аппаратных средств, которые называются *системой защиты памяти*, и, в простейшем случае, имеют вид аппарата страничной приписки.

Предлагаемая ниже схема защиты памяти **условно** привязана к рассмотренному нами подмножеству системы команд М-20. Это **не** означает, что в реально существовавшей М-20 действительно была защита памяти, причем именно такая, которую мы сейчас сконструируем.

Рассмотренная выше система команд, как мы видели, оперирует 12-разрядными адресами. Разделим (сначала мысленно) 12-разрядный адрес на два поля: старшие три разряда назовем **номером страницы**, а оставшиеся младшие девять разрядов — **смещением в странице**.

Теперь добавим к аппаратуре процессора **таблицу приписки**, состоящую из 8-**страничных регистров**. Пока будем считать, что регистр приписки состоит из трех битов, как и номер страницы.

Смысл таблицы приписки — в том, чтобы модифицировать адреса команд и данных, с которыми процессор обращается в память.

Рассмотрим работу регистра приписки номер 5, и предположим, что в нем находится значение 3.

Регистр приписки номер 5 срабатывает всякий раз, когда адрес, с которым процессор обратился к памяти, содержит в старших трех разрядах значение 5, т.е. попадает в пятую страницу. Характер обращения к памяти не важен — это может быть обращение с записью результата некоторой команды, обращение за самой командой и т.п. При любом таком обращении с адресом происходит одно и то же: его старшие разряды (а именно — число «5», номер страницы) заменяются на содержимое пятого регистра приписки (в данном случае — на число 3). Пятая страница как бы «перенаправляется» на третью. Для программы это перенаправление прозрачно — она работает так, как если бы никакой приписки не было, а пятая страница памяти действительно была в ее распоряжении (рис. 2.1).

Адреса, которыми оперирует программа, называются *виртуальными*, а адреса, уже преобразованные системой защиты памяти — *физическими*.

В использовании этого аппарата имеется прямая аналогия с аппаратом прерываний. Сам аппарат защиты памяти — свойство

Адрес: |PPP|OOOOOOOOO|, например:  
1 0 1 0 0 0 0 0 1 1 1 – 7-е слово в 5-й странице.

Таблица приписки:

Номер регистра	Значение
000	...
001	...
010	...
011	...
100	...
101	011
110	...
111	...

Рис. 2.1. Простейшая система страничной приписки памяти: (PPP — трехбитное поле номера страницы; OOOOOOOOO — девятибитное поле смещения в странице; пятый регистр таблицы приписки равен 3, т.е. пятая виртуальная страница памяти отображена на третью физическую)

**аппаратуры**, реализовать его программно нельзя. Использование этого аппарата (занесение осмысленных значений в регистры приписки) происходит **программно**, и это — обязанность операционной системы. Например, переключение процессора на обслуживание очередной программы сопровождается, скорее всего, занесением настроенных для нужд именно этой программы значений в регистры приписки.

Правильно организовав значения регистров приписки для каждой из одновременно выполняющихся программ, операционная система добивается того, что, например, пятая страница у каждой из таких программ — своя, а разрушить «чужую пятую страницу» программа не может в принципе, как и вообще получить к ней какой-либо доступ.

Приведем несколько технических замечаний. В современных машинах, с их громадными объемами памяти, таблицы приписки хранятся, конечно, не в специальных блоках регистров, а в самой памяти — специальные регистры лишь указывают, где именно. Сами таблицы обычно имеют иерархическое строение, что позволяет уменьшить их объем [7,67]. Однако общий принцип остается тем же — имеется таблично заданное отображение виртуальных адресов в физические, прозрачное для прикладной программы, которое должна настроить операционная система.

В некоторых специализированных процессорах этот аппарат может отсутствовать, но в универсальных процессорах он есть всегда.

В регистре приписки, помимо уже рассмотренного нами поля **адреса страницы**, есть еще поле **атрибутов страницы**, позволяющее, например, сделать страницу доступной для чтения, но не для записи.

Легко видеть, что страничная приписка, действительно, способна защитить разные программы от «затирания памяти» друг другом. Но как защитить их от «затирания» самой таблицы приписки? Для этого придется ввести еще одну аппаратную возможность — многорежимность процессора. Обычно в процессоре предусматривают два режима: **режим супервизора** и **режим пользователя**. Операционная система выполняется в первом режиме, прикладные программы — во втором. В режиме пользователя некоторые команды не работают (точнее, вызывают **прерывание по запрещенной команде**). Такие команды называются **привилегированными**. К числу привилегированных относятся команды, с помощью которых можно изменить при-



писку. Таким образом, сделать это может только операционная система. Сама команда смены режима процессора — также привилегированная. Как же процессор попадает из режима пользователя в режим супервизора? В результате прерываний. При прерывании автоматически включается режим супервизора. При возврате из обработчика прерывания операционная система его выключает.

Замечание об управлении внешними устройствами. Очевидно, что команды доступа к портам внешних устройств должны быть привилегированными (или, если эти порты отображены в адреса памяти, эта память должна быть защищена от записи в режиме пользователя). В противном случае программа пользователя могла бы, например, разрушить файловую систему на диске, выдав устройству управления дисководом некорректную команду на запись данных.

Как же в таком случае программа пользователя пишет, например, данные на диск? Очевидно, она делает это не непосредственно, а путем обращения к операционной системе «с просьбой» сделать это «за нее». Такое обращение прикладной программы к операционной системе за услугой (обычно — за вводом-выводом) называется **системным запросом**. При исполнении системного запроса должно произойти переключение процессора из режима пользователя в режим супервизора. Команда (или команды) в составе программы пользователя такого переключения выполнить не могут. Как же конкретно (на уровне команд) выглядит системный запрос?

Известно, что переход в режим супервизора происходит при прерывании. Обычно это происходит «неожиданно» для программы, по наступлению некоторого события, в самой программе явно не предусмотренного. В данном же случае мы столкнулись с необходимостью сделать все то же самое, но не «неожиданно», а по явному запросу программы. Для этого предназначен специальный вид прерываний — **программные прерывания**. В отличие от «неожиданных» прерываний, программные прерывания явно иницируются специальными командами процессора — командами программных прерываний. Смысл программного прерывания состоит в том, чтобы программа могла явным запросом активировать некоторый код внутри операционной системы, выполняющийся в режиме супервизора. Чтобы выполнить системный запрос, прикладная программа заносит в некоторое фиксированное место (например, на регистры общего на-

значения процессора) код запроса и параметры, говорящие о том, в чем конкретно заключается запрос, а затем выполняет команду программного прерывания. При этом происходит самое настоящее прерывание, т. е. переход на соответствующий обработчик в составе ОС с одновременным включением режима супервизора. Выполнив запрос, ОС возвращает управление прикладной программе в режиме пользователя. Конечно, системный запрос не обязан касаться только управления внешними устройствами. Это может быть, например, запрос на изменение таблицы приписки («дай страницу»).

### 2.3.5. Виртуальная память

Рассмотрим, как с помощью только что описанных возможностей реализуется такая часто упоминаемая функция ОС, как предоставление прикладной программе *виртуальной памяти*. Для того чтобы понять, как прикладная программа может, например, пользоваться памятью большего объема, чем реально есть в компьютере, достаточно мысленно добавить в регистр атрибутов страницы в таблице приписки всего один бит — бит, говорящий о том, что данная страница отсутствует. Когда в ходе выполнения программы происходит обращение к такой странице, происходит *прерывание по отсутствующей странице*. Обработывая это прерывание, ОС находит на диске временно «откачанную» туда копию этой страницы, читает ее с диска в некоторое место в памяти, настраивает регистр приписки на это место в памяти и возвращает управление прикладной программе, причем на ту самую команду, которая не смогла выполниться, вызвав прерывание. Таким образом, прикладная программа, в итоге, «ничего не замечает».

При подкачке страницы с диска может возникнуть необходимость откатить из памяти на диск какую-то другую страницу. Когда она потребуется — ОС подкачает ее снова. Так реализуется виртуальная память, по объему превышающая физическую.

## 2.4. Кэш-память и прямой доступ к памяти (DMA)

Практически в любом современном компьютере суммарное время обращения к памяти при выполнении команды значительно превышает время выполнения самой команды внутри про-

цессора. Доступ к памяти, таким образом, является наиболее «узким местом», важнейшей причиной снижения реальной производительности. При этом данные, которыми оперируют команды, часто требуются повторно, уже при выполнении следующей команды, или несколько команд спустя. Учитывая это, процесс выполнения программы можно значительно ускорить, если предусмотреть в процессоре специальную сверхбыструю память — кэш-память.

**Кэш-память** — чисто аппаратное образование, часть процессора, работающая без вмешательства операционной системы. Когда процессор обращается к оперативной памяти за значением, это значение по пути из оперативной памяти в процессор попадает в кэш-память и там запоминается. При следующем обращении к этому значению оно будет доставлено в процессор прямо из кэш-памяти — обращения к оперативной памяти не произойдет. Естественно, при записи значения из процессора в оперативную память значение записывается сначала в кэш-память. Размер кэш-памяти значительно меньше размера оперативной памяти: например, в БЭСМ-6 кэш-память данных состояла всего из восьми слов, а объем оперативной памяти равнялся 32 768 словам. В современных процессорах соотношение размеров аналогичное [67].

При очередной попытке записать в кэш-память новое значение может оказаться, что места в ней больше нет. В этом случае самое старое значение из кэш-памяти (то, к которому не обращались дольше всего) вытаскивается в оперативную память, а новое значение записывается на его место. Здесь просматривается некоторая аналогия с описанной выше организацией страничной подкачки в системе виртуальной памяти. Главное различие — в том, что кэш-память работает полностью аппаратно. Команд процессора, помогающих управлять манипуляциями с кэш-памятью, в системе команд может вообще не быть. В этом случае кэш-память логически прозрачна для программиста. Почему же мы включили обзор этого понятия в подраздел, посвященный архитектуре (а не реализации) процессора? Причин тому, как минимум, две.

Наличие и размер кэш-памяти видны программисту хотя бы в силу того ускорения, которое получается за счет ее работы. Допустим, например, что наша модельная программа (см. подразд. 1.5) выполняется на процессоре, кэш-память которого целиком вмещает все ее данные. По сравнению со случаем, когда

размер кэш-памяти в несколько раз меньше размера массива  $f$ , программа будет работать в несколько раз быстрее. Если попытаться варьировать размер обрабатываемой сетки, зафиксировав размер кэш-памяти, то, скорее всего, будет найдено некоторое почти «пороговое» значение, при выходе за которое работа программы резко замедлится. Замедление это будет нам вполне заметно, а значит, его механизм — элемент не только реализации, но и архитектуры процессора.

Вторая причина отнесения понятия кэш-памяти не к реализации, а к архитектуре напрямую относится именно к параллельным вычислительным системам. Наш краткий обзор принципа работы кэш-памяти молчаливо предполагает, что «хозяин» у оперативной памяти один — процессор. А что будет, если процессоров на общей оперативной памяти несколько, но кэш-память у каждого процессора своя? Например, один из процессоров в составе SMP-системы записал значение «1» в некоторую управляющую переменную, в которой раньше был «0», и ждет, пока второй процессор, «увидев» это, запишет туда значение «2». Второй процессор опрашивает значение этой переменной в ожидании, что там эта единица появится. Ждать они оба будут очень долго — в первом процессоре новое значение (единица) «застряло» в кэш-памяти, и до общей оперативной памяти пока не дошло. Во втором — старое значение (ноль) заняло свое место в кэш-памяти, и, сколько его в общей оперативной памяти ни модифицируй усилиями первого процессора, доступная второму процессору через его кэш-память копия не изменится. Чтобы такая прискорбная ситуация не возникала на практике, кэш-память должна быть «умной» — как минимум, при всякой записи в оперативную память, выполненной другим процессором, она должна автоматически обновлять свою копию записанного значения (при ее наличии). В действительности все несколько сложнее, но не будем вдаваться в детали.

Реализация SMP- или NUMA-системы, в которой кэш-память логически полностью прозрачна, т. е. аппаратура совершенно самостоятельно справляется с неприятностями вроде описанной только что, называется *кэш-когерентной*. Альтернативой является многопроцессорная система, в процессорах которой предусмотрены специальные команды управления кэшированием, что-то вроде «повторно взять из оперативной памяти в кэш-память указанное значение». Необходимость правильно расставлять в программе такие команды естественно возлагается на программиста.

Говоря о кэш-памяти, часто подчеркивают, что она представляет собой *ассоциативную память*. Так называются устройства памяти, в которых значения хранятся вместе с адресами. В самом деле, если в обычной (не ассоциативной) памяти значения имеют фиксированные при изготовлении памяти адреса, то в кэш-памяти набор адресов присутствующих в памяти значений не является сплошным и меняется со временем. Такая память похожа на ассоциативную базу данных, в которой доступ к значению происходит по ключу, и называется аналогично.

В заключение нашего архитектурного обзора нам потребуется несколько расширить уже имеющиеся у нас представления о том, как работают устройства ввода-вывода.

Обсуждая в подразд. 2.3.1 общие принципы управления внешними устройствами, мы рассмотрели один из возможных режимов, применяемый на практике лишь для довольно медленных устройств. Рассмотренный нами режим называется *PIO* (*Program Input-Output*, программный ввод-вывод). Для устройств сравнительно быстрых, таких, как адаптер жесткого диска или высокоскоростной локальной сети, обычно применяется другой режим, называемый *DMA* (*Direct Memory Access*, прямой доступ в память). Суть этого режима в том, что передача данных происходит не байтами или словами, а блоками, возможно из десятков тысяч байт, и процесс выборки блока из памяти (при выводе) или его записи в память (при вводе) устройство выполняет самостоятельно, без помощи процессора. Процессор лишь «сообщает» устройству (занося соответствующие значения в порты управления контроллером) следующие сведения:

- где именно находится в памяти передаваемый блок;
- какова его длина;
- вывести его надо или ввести;
- когда начать.

Запустив таким образом обмен, процессор может заниматься полезной вычислительной работой, пока устройство, самостоятельно взаимодействуя с памятью, этот обмен выполняет. Когда обмен завершится — произойдет прерывание.

Высокопроизводительные локальные сети, применяемые в современных вычислительных кластерах, реализуют интересный частный случай режима DMA, называемый *RDMA* (*Remote DMA*, дистанционный прямой доступ в память).

При обменах данными с жестким диском или аналогичным устройством в режиме DMA устройство самостоятельно выпол-

няет «перекачку» данных в память (или из памяти), но инициатором каждой порции такой «перекачки» является процессор. Жесткий диск пассивен — он не выполняет никакой программы и не может «сам собой захотеть», например, записать данные в оперативную память. Несколько иначе обстоит дело в случае адаптера локальной сети («сетевой карты»). Сетевая карта связана линиями связи с другими сетевыми картами, каждая из которых подключена к процессору, а процессор, в отличие от жесткого диска, является источником самостоятельной активности. Например, пусть в сетевую карту по линии связи извне поступил пакет, адресованный именно данному компьютеру. В принципе сетевая карта могла бы вполне самостоятельно, не только без вмешательства, но и без ведома процессора, записать содержимое этого пакета в оперативную память, воспользовавшись режимом DMA — если, конечно, в пакете указано, куда писать. Возможна передача данных и в обратном направлении. В сетевую карту поступает пакет с запросом на выдачу в линию некоторых данных, расположенных в оперативной памяти. Сетевая карта, опять — самостоятельно, без участия и, возможно, без ведома процессора, читает эти данные из оперативной памяти в режиме DMA, и отправляет инициатору запроса.

Именно такой режим обмена данными в локальной сети и называется режимом RDMA. Его называют также режимом односторонних обменов, подчеркивая, что для обмена данными между двумя узлами локальной сети достаточно инициативы только одной стороны, т.е. достаточно, чтобы только на одном из процессоров выполнялись команды, организующие обмен. Второй участник обмена получит (или отдаст) данные «насильно», не выполняя никаких задающих обмен данными команд в своей программе. В подразд. 1.3 упоминался такой режим организации коммуникаций между процессорами.

Использование режима RDMA для обмена данными в вычислительных кластерах будет рассмотрено подробно в последующих главах. Сделаем несколько уточняющих замечаний.

Наличие или отсутствие режима RDMA является аппаратным свойством сетевой технологии (свойством «конструкции» сетевой карты). В каждой конкретной сетевой технологии либо предусмотрен на аппаратном уровне протокол одностороннего обмена данными, либо нет. В сетях, где этот режим работы аппаратуры не предусмотрен, режим односторонних обменов данными можно организовать, моделируя логику RDMA программно, с по-

мощью прерываний. При этом неизбежны значительные накладные расходы.

При использовании режима односторонних обменов в прикладных программах возникает много вопросов, ответ на которые не очевиден. С первого взгляда может возникнуть впечатление, что осмысленное использование этого режима просто невозможно. Это впечатление не соответствует действительности. Вопрос о месте дисциплины односторонних обменов в спектре технологий параллельного программирования, о достоинствах и недостатках этой дисциплины, будет подробно рассмотрен в гл. 5.

## 2.5. Еще несколько определений

Итак, обзор принципиальных возможностей процессора общего назначения завершен. Легко видеть, что рассмотренных возможностей, действительно, достаточно для реализации операционной системы в том виде, в каком мы ее знаем.

Это — **исчерпывающий** перечень базовых архитектурных принципов, на которых строится любая операционная система, как бы сложна и многообразна она ни была. **Ничего принципиально другого в процессоре нет.** В свою очередь, наиболее базовые понятия операционной системы, которые строятся на основе этих возможностей процессора, включают в себя понятие *процесса* — такого способа выполнения программы, как если бы она выполнялась на отдельном процессоре, и *виртуальной машины*, предоставляемой программе операционной системой. В системе команд виртуальной машины подмножество команд реальной машины (все не привилегированные команды) расширено набором специальных «рукотворных» команд — системных запросов. Память виртуальной машины, конечно, тоже виртуальная, и может превосходить по размеру память физической машины или ее часть, реально используемую данным процессом.

Технически системные запросы реализуются обычно через команды программных прерываний. Процесс выполняется в рамках виртуальной машины.

Постоянно находящаяся в памяти программа, предназначенная для запуска других программ и обслуживания их потребностей в общении с внешним миром, называется *операционной системой*.

Часть операционной системы, которая выполняется в режиме супервизора, содержит обработчики прерываний и предоставляет прикладной программе виртуальную машину, называется **ядром**. Сменные модули ядра, предназначенные для обслуживания отдельных внешних устройств, называются **драйверами**. Вспомогательные части операционной системы, которые сами выполняются в рамках процессов и тем самым не являются частью ядра, а являются «служебными программами», обычно называют **демонами** (в Unix) или **сервисами** (в Windows).

Мы видели, что виртуальная машина, предоставляемая пользователю, реализуется как аппаратными возможностями процессора, так и программным строением операционной системы. Именно по этой причине исполняемые модули программ пользователя, разработанные в одной операционной системе, вообще говоря, не могут исполняться на том же самом процессоре, но под управлением другой операционной системы, а исполняемые модули для одной и той же операционной системы не могут исполняться на разных по системе команд процессорах, даже если работающие на них ОС одинаковы. В обоих случаях системы команд виртуальной машины, в рамках которой предстоит выполнение программы пользователя, не совпадают. В первом случае они не совпадают в части «рукотворных» команд — системных запросов, во втором — в части аппаратно реализованных процессором команд.

В современной программистской терминологии термин «виртуальная машина» часто употребляется в связи с конкретными программистскими технологиями, например, применительно к трансляторам языка java, или к технологии виртуальных машин в узком смысле этого слова, реализованной в программных продуктах фирмы VMWare [18]. Такое употребление данного термина не следует смешивать с его общим, изначальным смыслом, описанным в данной главе.

Замечание. Объявленной целью настоящей главы было формулирование исчерпывающего перечня принципиальных возможностей современного процессора. Важно понимать разницу между возможностями **принципиальными** и возможностями **важными** (и даже **очень важными**). В самом деле, мы ни слова не сказали о таких, безусловно, важнейших свойствах архитектуры и реализации современных процессоров, как конвейерный и спекулятивный режимы исполнения команд, и о многом другом [66, 67]. Из этого не следует, что эти вопросы не важны.



Например, за незнание того, что такое спекулятивное исполнение команд, вполне можно получить «два балла» на экзамене по архитектурам современных микропроцессоров. Однако, не рассмотренные нами здесь, хотя и очень важные сами по себе, вопросы не представляются автору принципиальными с точки зрения предстоящего рассмотрения структуры и свойств программного и аппаратного обеспечения параллельной обработки данных.

# КРИТЕРИИ ЭФФЕКТИВНОСТИ ПРОЦЕССОРА И КОММУНИКАЦИОННОЙ СЕТИ

### 3.1. Из чего складывается вычислительная производительность

В последующих главах будет проведен сравнительный обзор того материала, из которого строятся вычислительные кластеры: процессоров (точнее, рабочих станций, построенных на их базе) и оборудования локальных сетей высокой производительности. Чтобы сравнивать, необходимо сначала сформулировать критерии эффективности. Критериям эффективности коммуникационных сетей посвящен подразд. 3.2. В данном подразделе остановимся на критериях эффективности процессоров. Казалось бы, отдельного подраздела для рассмотрения такого простого вопроса слишком много. В самом деле, если мы собираемся вычислять как можно с более высокой скоростью, критерий эффективности один — вычислительная производительность. Но что это такое, как она определяется, чему равна и в чем измеряется? Как и многое в технологиях высокопроизводительных вычислений, вопросы эти гораздо сложнее, чем кажется на первый взгляд.

С одной стороны, синонимом быстродействия процессора сегодня принято считать его рабочую частоту, с другой — вычислительное быстродействие принято измерять во *флопсах*, т. е. в количестве арифметических операций с плавающей точкой в секунду (*flops* — *f*loating point *o*perations per *s*econd). А еще есть «пиковое быстродействие». Как все это связано между собой?

Процессор является синхронным устройством дискретного действия, квантом времени, в котором он функционирует, является один такт рабочей частоты. Быстрее, чем за один такт, в процессоре никакая обработка данных не происходит. Объем работы, которую процессор способен выполнить за один такт, определяется его логической схемой, точнее — степенью внутреннего параллелизма в выполнении команд, которую разработчикам удалось реализовать в схеме процессора. Достижимая степень внутреннего параллелизма, в свою очередь, зависит от имеющегося в распоряжении разработчика числа транзисторов.

Чем больше транзисторов способны переключаться одновременно, тем больше полезной работы в принципе может быть выполнено за время, необходимое для одного переключения.

Примерно 20 лет назад, когда суммарное число транзисторов в составе процессора было гораздо меньше, чем в данный момент, разработчики процессоров были вынуждены «растягивать» выполнение команд на несколько тактов. При этом простые команды, вроде пересылки значения из одного регистра в другой, занимали, например, один или два такта, а сложные, вроде умножения чисел с плавающей точкой — десятки тактов. Время выполнения типичной программы вычислительного характера если и не полностью, то в значительной степени определялось временем выполнения операций над числами с плавающей точкой, которые встречались в этой программе. Отсюда пошла традиция измерять быстродействие процессора во флопсах. По мере роста числа транзисторов на кристалле росла и степень внутреннего параллелизма в реализации процессорных команд. Сегодня она такова, что практически любая команда процессора выполняется за одно и то же время, не превышающее времени процессорного такта. Более того, появилась возможность выполнять несколько следующих подряд команд за один такт. Современные версии Pentium [7] и Opteron [8] выполняют, при благоприятных условиях, две команды за такт, в том числе, возможно, и две команды, каждая из которых есть операция над числами с плавающей точкой. Упомянутый здесь коэффициент, равный для Pentium и Opteron двум, известен для каждого процессора и определяется его логической схемой. Умножая его на рабочую частоту, получаем *пиковое быстродействие*.

По старой традиции, пиковое быстродействие принято измерять во флопсах. Однако если 20 лет назад измеренное во флопсах пиковое быстродействие, действительно, давало некоторое представление о том, сколько полезных арифметических операций процессор может выполнить за секунду, то сегодня эта величина не означает практически ничего. В самом деле, чтобы вычислять с плавающей точкой со скоростью, равной пиковой производительности, надо совсем не выполнять команд, связанных с формированием адресов вычисляемых значений. Раньше временем выполнения такого рода команд пренебрегали, но теперь любая манипуляция с адресом или простая пересылка значения из регистра в регистр выполняется столько же времени, сколько занимает умножение чисел с плавающей точкой. По-

сколько в любых сколько-нибудь реальных программах такого рода «вспомогательных» команд больше, чем «полезных» арифметических действий, достижение хоть чего-то похожего на пиковую производительность становится проблематичным.

Но это еще не все. Ранее обсуждались способности процессора выполнить две команды за такт **при благоприятных условиях**. Что это за условия? Если процессор выполняет две команды каждый такт, значит каждую половину такта он выполняет по одной команде ... стоп. Мы ведь начали с того, что ни за половину, ни за 0,99 такта процессор ничего сделать просто не может — он работает дискретно. Следовательно, он действительно выполняет за такт две команды, а не за половину такта — по одной, т.е. он выполняет их **одновременно**. Но ведь не всякие две команды, записанные в программе подряд, можно выполнить одновременно! Если следующая команда использует результат предыдущей, то, вообще говоря, выполнять их в одном такте нельзя. Именно по этой причине следует с большой осторожностью относиться к величинам пикового быстродействия процессоров, выполняющих за такт не две, а, например, четыре команды. Обеспечить «благоприятные условия», т.е. «хорошо упаковать» смесь команд, транслятору используемого Вами языка программирования для такого процессора будет очень и очень непросто.

Серьезнейшим источником отклонения от «благоприятных условий» являются задержки при доступе в оперативную память. Если данные не находятся в кэш-памяти, то адресующая их команда будет выполняться гораздо дольше, чем в идеальном случае. Тем более когда за право доступа к этой памяти «борются» несколько процессоров (ядер)...

На сегодня известна лишь одна программа вычислительного характера, выполняющая осмысленный расчет, которая способна показывать быстродействие, близкое к пиковому. Это — тест Linpack [4,9], решающий систему линейных алгебраических уравнений методом Гаусса. И дело здесь не в том (точнее, не только в том), что именно метод Гаусса как-то особенно хорош для современных процессоров. Дело в том, что именно этот тест принято использовать для оценки производительности, особенно для оценки производительности многопроцессорных вычислительных систем. По этой причине на написание специальных, очень сильно оптимизированных на уровне отдельных команд, версий этого теста (точнее, не самого теста, а библиотек, которыми он

пользуется, но сути дела это не меняет) разработчики процессоров тратят громадные силы и средства. Ни одну другую реальную программу никто не оптимизирует так долго и тщательно — это просто практически невозможно. Конечно, метод Гаусса, просто написанный полностью «с нуля» на языке С или Фортране и пропущенный даже через очень хороший транслятор, покажет многократно меньшую производительность, чем специально «выжатые» демонстрационные варианты теста.

Для оценки полезной производительности как отдельного процессора, так и многопроцессорной вычислительной установки в целом, в более или менее реальных условиях, принято использовать так называемые тесты NASA (NAS parallel benchmarks) [10]. Это фрагменты реальных численных методов, используемых в задачах математической физики. Получаемые на этих тестах значения производительности во флопсах очень далеки от пикового, и это совершенно нормально.

Тесты NASA предназначены, в первую очередь, для измерения эффективности параллельной реализации типовых вычислительных алгоритмов. В зависимости от объема обрабатываемых данных, с одной стороны, и свойств коммуникационной среды многопроцессорной установки, с другой, реально достигаемая производительность может колебаться в широких пределах. Но даже верхний порог этих пределов, т.е. уровень производительности, достигаемый на отдельно взятом процессоре, безо всяких накладных расходов на распараллеливание, редко бывает выше 15 % от пикового. Таким образом, для реальных задач львиная доля потерь производительности кроется не в накладных расходах на распараллеливание, а в локальной обработке данных процессором, по причинам, кратко охарактеризованным выше.

В качестве самостоятельного упражнения можно рекомендовать читателю измерить производительность упоминавшейся выше модельной программы «прогрев кирпича». Автору ни разу не приходилось видеть для этой задачи значения производительности в «полезных флопсах», превышающего 10 % от «пика», при использовании одного процессора. И это при том, что данная программа использует регулярную, или индексную, сетку, при работе с которой количество команд, связанных с адресацией обрабатываемых данных, сравнительно мало. Для численных методов на сложно организованных, так называемых «нерегулярных», сетках даже 3 % от «пика» — явление совершенно

нормальное, хотя, конечно, очень многое зависит от транслятора.

### 3.1.1. Коэффициент распараллеливания. Закон Амдала

В предыдущем подразделе было показано, что измерение производительности вычислительных приложений и даже выбор подходящей единицы измерения — дело не простое. Гораздо проще обстоит дело, если мы хотим оценить скоростной выигрыш при переходе от выполнения расчета на однопроцессорном компьютере к выполнению того же расчета на многопроцессорной вычислительной системе. В самом деле, полагая процессоры одинаковыми, мы, очевидно, хотели бы всякий раз получать ускорение в  $N$  раз, где  $N$  — число процессоров. По причине накладных расходов на организацию взаимодействия мы, как правило, получим несколько меньшее ускорение. Сравнивая ожидаемое «идеальное» ускорение с реально получившимся, мы можем совершенно точно сказать, насколько хороша параллельная реализация расчета.

Пусть расчет выполняется на однопроцессорной машине за время  $T_1$ , а на системе из  $N$  процессоров — за время  $T_N$ . Величину  $S=T_N/T_1$  будем называть **ускорением** для данной параллельной реализации расчета.

Нам бы хотелось, чтобы на системе из  $N$  процессоров расчет выполнялся за время, равное  $T_1/N$ . За меру качества параллельной реализации естественно взять отношение этих величин:

$$P = T_1/(N \cdot T_N)$$

Величину  $P$  принято называть **коэффициентом распараллеливания** для данной параллельной реализации расчета. Это — своего рода «коэффициент полезного действия» параллельной реализации. Если на десяти процессорах мы получили время, лишь в пять раз меньшее, чем на одном, то  $P = 0,5$ .

Верно ли, что  $P$  не может быть больше 1? Нет, не верно. Такие ситуации бывают, и не только на тестах, но и в реальных задачах. Тому есть несколько причин.

Первая и очевидная — эффект использования кэш-памяти. При параллельной реализации объем данных, обрабатываемых одним процессором, уменьшается примерно в  $N$  раз. Если при этом преодолевается порог эффективного использования кэш-

памяти, например, все или почти все данные, обрабатываемые в критическом цикле, начинают умещаться в кэш-памяти, это дает дополнительное ускорение, и  $P$  может оказаться больше единицы.

Второй, несколько менее заметный источник дополнительного ускорения — возможные изменения в способе адресации данных. Пусть, например, некоторая программа на Фортране обрабатывает трехмерный массив. При переходе к параллельной реализации каждый слой такого массива было решено поместить на отдельный процессор, и массив в каждом процессоре теперь двумерный. Доступ к элементам массива стал гораздо более эффективным, и это также дало дополнительное ускорение.

Ситуации, когда  $P$  заметно меньше 1, встречаются на практике гораздо чаще. Рассмотрим ситуацию, когда источники дополнительного ускорения отсутствуют, и зададимся вопросом: как будет меняться значение  $P$  по мере роста числа используемых процессоров? Интуиция подсказывает нам, что при уменьшении объема обрабатываемых данных будут расти удельные накладные расходы на организацию параллельной обработки. Пойдем еще дальше — рассмотрим идеальный случай, т. е. предположим, что накладные расходы равны нулю. Предельное значение  $P$ , на которое можно рассчитывать в этом случае, как ни странно, строго меньше единицы и определяется соотношением, которое называется *законом Амдала* [3].

Пусть в однопроцессорном варианте программы имеется цикл прохода по некоторому массиву, с его поэлементной обработкой. Разделив массив между  $N$  процессорами, написав для каждого процессора аналогичный цикл и пренебрегая накладными расходами на организацию взаимодействия, мы, казалось бы, вправе рассчитывать на ускорение расчета в  $N$  раз, поскольку мы поделили выполняемую работу на  $N$  частей. К сожалению, мы в действительности, почти наверняка поделили на  $N$  частей **не всю** работу. В реальной программе, скорее всего, выполняется итерационный процесс, в котором проход по массиву выполняется на каждой итерации. Следовательно, имеется цикл по итерациям, внешний по отношению к циклу прохода по массиву, который мы рассматриваем. На каждой итерации этого внешнего цикла проверяется, надо ли выполнять внутренний цикл еще раз, или расчет пора заканчивать. Эту работу мы не поделили — напротив, мы размножили ее по процессорам, т. е. **каждый** из  $N$  процессоров теперь выполняет действия по организации **своего** внешне-

го цикла, в который вложен сократившийся в  $N$  раз внутренний цикл. Совокупная работа, действительно, поделена на  $N$  частей, но ее стало больше, чем было в однопроцессорном случае, и будет становиться тем больше, чем больше значение  $N$ . Пусть в однопроцессорном варианте программы работа по организации цикла занимала 0,001 суммарного времени. Это означает, что одна тысячная исходной совокупной работы осталась не поделенной, т.е. добавление каждого нового процессора увеличивает совокупную работу на одну тысячную от исходной. Как минимум, отсюда следует, что ускориться в 1 000 раз мы точно не сможем, даже используя 10 тыс. процессоров.

Совершенно аналогичный по затратам времени эффект наблюдается в том случае, когда некоторая вычислительная работа не размножается по процессорам, а просто выполняется последовательно, т.е. один процессор ее выполняет, а все остальные — ждут. Чтобы охватить оба эти варианта, будем говорить, в общем случае, о распараллеленной и нераспараллеленной вычислительной работе.

В реальных параллельных программах объем нераспараллеленной вычислительной работы изменяется в широких пределах. Зависимость идеального коэффициента ускорения от доли нераспараллеленной исходной работы и устанавливается законом Амдала.

Пусть доля нераспараллеленных вычислений в программе равна  $F$ . Тогда время работы программы на  $N$  процессорах не может быть меньше величины

$$(T_1 - T_1 * F) / N + T_1 * F$$

Соответственно, ускорение не может быть больше, чем

$$1 / (F + (1 - F) / N)$$

Именно это неравенство представляет собой формулировку закона Амдала.

### 3.1.2. Масштабирование приложений

**Масштабированием** вычислительных приложений называют их ускорение при выполнении на все большем числе процессоров. Как отмечалось в подразд. 3.1.1, даже в идеальном случае, т.е. при отсутствии накладных расходов на организацию межпроцессорного взаимодействия, масштабирование приложе-



ния при росте числа используемых процессоров не всегда бывает линейным. С другой стороны, предпринятое в подразд. 3.1.1 рассмотрение идеального случая вовсе не означает, что действительность к нему близка. В подавляющем большинстве реальных приложений накладные расходы на организацию межпроцессорного взаимодействия более чем заметно влияют на общую картину, и, если их доля в общих затратах времени растет, масштабирование быстро теряет смысл.

Под масштабированием приложений обычно понимают, не оговаривая этого специально, одно из двух совершенно разных действий.

**1. Масштабирование по числу процессоров при фиксированном объеме исходных данных.** Пусть мы масштабируем решение системы из 1 000 линейных уравнений. Нарастив число процессоров, и наблюдая за значением коэффициента ускорения, мы увидим, что почти линейный рост сменится сначала более медленным ростом, а затем — падением. По мере роста числа процессоров приложение пройдет через две точки деградации: в первой добавление новых процессоров перестанет сокращать время счета, во второй добавление новых процессоров приведет к замедлению по сравнению с однопроцессорным вариантом. Чем больше то число процессоров, при котором наступает первая точка деградации, тем лучше масштабируемость приложения. Почти всегда масштабируемость приложения сильно зависит от свойств коммуникационного оборудования.

Такой способ масштабирования является наиболее «честным», или «жестким» способом оценки пригодности многопроцессорной вычислительной системы для решения конкретной задачи. Он соответствует следующей постановке вопроса: «У меня есть конкретная задача, до какой степени я могу ускорить именно ее решение на данной системе?»

**2. Масштабирование по числу процессоров с наращиванием объема исходных данных.** Пусть мы решаем систему линейных уравнений произвольного размера. Пусть мы умеем измерять (не важно, в каких именно единицах) достигаемое при этом вычислительное быстродействие. Поставим задачу максимизировать это быстродействие, используя столько процессоров, сколько удастся. Сначала зададимся некоторым размером системы, и начнем решать ее на все большем числе процессоров. Заметив признаки деградации, не будем прекращать наши усилия — вместо этого просто увеличим число уравнений в системе.

Для подавляющего большинства реально применяемых параллельных численных методов это приведет к тому, что точка деградации «отодвинется». Будем поступать так до тех пор, пока не исчерпается оперативная память системы, после чего заявим, что наша система имеет именно такое быстроедействие на приложении «решение системы линейных уравнений».

Такой способ масштабирования также не лишен смысла (отметим, что именно он традиционно применяется для ранжирования суперкомпьютеров при включении в известный список «Тор-500» [10]). Он соответствует следующей постановке вопроса: «Если Ваша задача плохо ускоряется на нашем суперкомпьютере, значит, она просто мала для него. На таком большом суперкомпьютере, как наш, надо решать действительно большие задачи».

Рассмотрим с этой точки зрения нашу модельную программу. На рис. 1.3 легко видеть, что объем пересылаемых из процессора в соседние процессоры данных, при выбранном нами методе параллельной реализации, пропорционален размеру расчетной сетки. Объем же вычислительной работы в промежутках между пересылками пропорционален квадрату размера сетки. Естественно ожидать, что для как угодно «плохой» коммуникационной сети по мере роста размера сетки объем накладных расходов на пересылки можно сделать как угодно малым в процентном отношении. Вопросов только два: хватит ли оперативной памяти, и нужны ли пользователю сетки такого размера?

Параллельные вычислительные приложения, которые хорошо масштабируются «жестким» способом, даже на плохих сетях, называют **коммуникационно нечувствительными**. Формализованному определению смысла слов «плохая» и «хорошая» применительно к коммуникационным сетям посвящен подраздел 3.2.

### 3.2. Критерии эффективности коммуникационной сети

Материал этого подраздела частично заимствован из [2]. В предшествующем изложении нам уже приходилось упоминать такие понятия, как, например, «как можно более быстрая сеть». Здесь будет показано, что само понятие «быстрая сеть» можно понимать, как минимум, двумя разными способами. В то же

время, помимо «быстроты», сети обладают еще многими важными свойствами, по наличию или отсутствию которых их можно сравнивать между собой.

### **Производительность, латентность и цена обмена.**

**Производительностью** канала «точка — точка» между узлами А и В будем называть количество данных, передаваемых по каналу в единицу времени, в среднем за некоторый большой промежуток астрономического времени, например за минуту. Производительность канала можно представить себе как среднюю (за длительное время) скорость передачи данных. Очевидно, производительность канала сильно зависит от того, какой длины сообщения используются при передаче данных, т.е. от того, как часто канал «останавливается», с тем, чтобы потом снова «разогнаться».

Пусть узел А передает узлу В сообщение длиной  $X$  байтов, и при этом никаких других обменов в сети не происходит. Время  $T$ , затрачиваемое на такую передачу, довольно точно оценивается формулой:

$$T = X/S + L,$$

где  $L$  не зависит от  $X$ .

В этой формуле, очевидно,  $S$  — **пропускная способность канала «точка — точка» на пустой сети**, или, попросту, **мгновенная скорость** передачи данных;  $S$  измеряется в мегабайтах в секунду.

Величина  $L$ , в свою очередь, представляет собой **время запуска обмена**, не зависящее от длины сообщения, и измеряется в микросекундах (мкс). На профессиональном языке принято называть эту величину **латентностью**, и мы будем впредь поступать так же, чтобы не нарушать традицию, хотя это, возможно, и не совсем правильно с точки зрения терминологической строгости.

Иногда удобно оперировать **латентностью, приведенной к скорости**, или **ценой обмена**, которую мы обозначим как  $P$ :

$$P = L \cdot S.$$

Эта величина измеряется в байтах и имеет несколько полезных «физических» интерпретаций. Прежде всего, цена обмена — это то число байт, которое канал «точка — точка» мог бы передать за время своего запуска, если бы «умел» запускаться мгновенно. Иными словами, за счет «инертности» канала, к каждому пере-

даваемому им сообщению «как бы добавляется», с точки зрения скорости передачи,  $P$  байт.

Таким образом, производительность канала зависит от длин сообщений, используемых при передаче данных. Если  $X$  много больше  $P$ , т. е. длины сообщений много больше цены обмена, производительность близка к пропускной способности. Напротив, если  $X$  много меньше  $P$ , то производительность практически полностью определяется латентностью, а не пропускной способностью. Наконец, при  $X$ , равном  $P$ , производительность равна в точности половине пропускной способности канала. Тем самым, **цена обмена — это такая длина сообщения, при использовании которой производительность канала равна половине его пропускной способности.**

В итоге мы сформулировали два независимых критерия эффективности: пропускную способность канала «точка — точка» на пустой сети и латентность, и один производный критерий — цену обмена. Очевидно, сеть тем лучше, чем выше пропускная способность, и чем ниже латентность. Так, в SMP-системе (машине с общей, симметрично адресуемой памятью) пропускная способность бесконечна, а латентность, теоретически, равна нулю.

Замечание о соотношении терминов «латентность» и «время запуска обмена». Давая определение понятия «латентность», мы упоминали некоторую терминологическую неточность этого определения. В строгом терминологическом смысле латентностью коммуникационного канала «точка — точка» следует называть время доставки короткого сообщения от отправителя к получателю. Измеряют ее так. Время многократной ( $N$  раз) посылки короткого сообщения «туда-обратно» делится на  $2N$ , что дает среднее астрономическое время доставки одного сообщения. В нашем же определении речь шла о локальном времени запуска обмена. Эта величина измеряется несколько иначе: время многократной ( $N$  раз) посылки короткого сообщения в одну сторону делят на  $N$ . Какова же причина традиции, действительно имеющей место в профессиональном жаргоне, согласно которой время запуска обмена называют латентностью?

Причина тривиальна. Для подавляющего большинства локальных сетей эти две величины **практически совпадают**, поскольку время собственно доставки сообщения пренебрежимо мало по сравнению со временем запуска обмена. В последнее время начинают появляться коммуникационные среды, в которых время запуска обмена заметно меньше латентности.

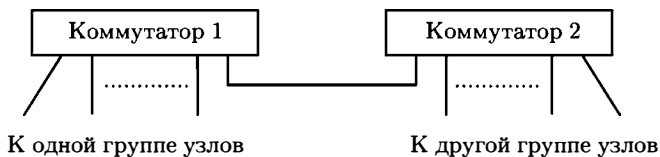


Рис. 3.1. Бисекционно неполная сеть

Для таких сетей значения «время запуска обмена» и «латентность» следует рассматривать по отдельности. Применительно к таким сетям популярен термин «message rate» — «ритм выдачи сообщений», т.е. величина, обратная ко времени запуска обмена.

**Полнота сети.** Содержательно, полнота сети есть мера того, насколько несколько одновременно происходящих обменов «мешают» друг другу. Простейшее определение полноты сети — это понятие *бисекционной полноты*: сеть называется бисекционно полной, если **любые** обмены между **разными** парами узлов, происходящие одновременно и в любом количестве, **совсем** не мешают друг другу. Или, что то же самое, при любом разделении сети пополам (бисекции) пропускная способность потока данных из одной половины в другую есть сумма пропускных способностей независимых каналов, ведущих из одной половины сети в другую. Типичный пример бисекционно неполной сети — сеть на базе двух однородных коммутаторов, объединенных между собой единственной линией (рис. 3.1).

Следует отметить, что при практической оценке пропускной способности реальной сети в режиме интенсивных перекрестных обменов модель бисекционной полноты недостаточна. Если сеть построена на базе центрального коммутатора (а так бывает чаще всего), то реальное время выполнения серии одновременных обменов, некоторые из которых пересекаются по участвующим процессорам, зависит от особенностей внутренней реализации коммутатора. Достаточно типична ситуация, когда два коммутатора разных моделей, каждый из которых реализует попарно независимые обмены без потери быстродействия, сильно отличаются по времени выполнения одной и той же тестовой серии частично пересекающихся обменов.

Следует также отметить, что информацию о внутренней логике поведения коммутатора, по которой можно было бы предсказать его скоростные характеристики в этом режиме, добыть обычно не удастся: производители ее не сообщают. При

этом в реальных задачах почти всегда имеет место именно такой режим, когда одни и те же процессоры участвуют одновременно в нескольких разных обменах. Поскольку достоверной информации о внутренней логике коммутатора у нас нет, практически мало целесообразно строить на эту тему какие-то сложные формализованные модели, основанные на выдуманных допущениях.

При необходимости оценить качество коммутатора, или сравнить несколько коммутаторов между собой, разумно воспользоваться каким-либо простым эталонным тестом. Такой тест должен циклически, много раз подряд осуществлять обмены всех узлов со всеми сообщениями фиксированной длины  $X$ , причем  $X$  должно быть заметно больше цены обмена. Все обмены на одном узле должны запускаться одновременно, чтобы исключить их зависимость друг от друга по порядку выполнения.

Если среднее время выполнения витка такого теста равно  $T$ , то легко подсчитать суммарный объем переданных за это время одним узлом данных  $D$ :

$$D = X \cdot (N - 1),$$

где  $N$  — число процессоров. Если бы сеть была идеальна, т.е. никакой обмен не мешал никакому другому, то этот объем данных мог бы быть передан из узла за время  $T_i$ :

$$T_i = D/S = (X \cdot (N - 1))/S.$$

Предполагая канал, связывающий узел с сетью, дуплексным, т.е. способным одновременно с передачей принимать данные с той же скоростью, и помня, что узлы работают одновременно, видим, что  $T_i$  представляет собой теоретически идеальное время срабатывания витка нашего теста. В действительности, тест будет выполнять виток за экспериментально измеренное время  $T_e$ , очевидно, большее, чем  $T_i$ . Поделив одно на другое, получим *экспериментальный коэффициент полноты*  $F_e$ :

$$F_e = T_i/T_e.$$

Вне всякого сомнения, приведенное здесь определение экспериментального коэффициента полноты нельзя назвать определением в математическом смысле этого слова — было сделано слишком много неформализованных допущений. Конечно, значение коэффициента будет, при прочих равных, зависеть и от

длины сообщения  $X$ , и от особенностей тестовой программы. Однако приведенная здесь процедура экспериментальной оценки полноты сети обладает одним несомненным достоинством: она позволяет на практике измерить, на сколько процентов, в буквальном смысле этого слова, идеальной является наша сеть, с точки зрения ее полноты.

Опыт показывает, что для сравнения между собой различных сетевых коммутаторов такой способ измерения весьма полезен.

Конечно, если полученные на разных коммутаторах значения  $Fe$  отличаются на четверть, то это мало о чем говорит, но на практике не редкостью являются отличия в несколько раз.

**Потребность сети в вычислительной мощности.** Пусть некоторое вычисление (например, перемножение двух матриц) занимает время  $T_m$ .

Выполним то же самое вычисление на фоне запущенного обмена, время завершения которого заведомо превосходит  $T_m$ . Теперь это же вычисление займет время  $T_c$ .

Коэффициент замедления счета на фоне обмена  $C$  вычисляется по формуле:

$$C = T_m / T_c.$$

Отметим, что для «более или менее разумной» сети  $C$  должно мало отличаться от единицы, т.е. сеть не должна заметно потреблять вычислительную мощность узла.

Этот критерий эффективности важен, поскольку в правильно организованных параллельных программах обмен данными стараются совмещать во времени с обработкой других данных. Например, передав операционной системе запрос на посылку в другой процессор некоторого сообщения, программа может заняться вычислениями, не затрагивающими содержимое посылаемого сообщения. ОС, скорее всего, использует для этой передачи режим DMA.

С точки зрения сформулированного критерия эффективности, чем меньше работа сетевого адаптера в режиме DMA «нагружает» память, замедляя работу процессора, тем лучше.

В итоге можно сформулировать четыре независимых критерия эффективности сети, важных с точки зрения ее практического использования:

- пропускная способность канала «точка — точка»;
- латентность;

- экспериментальный коэффициент полноты;
- потребность сети в вычислительной мощности.

**Существует ли интегральный критерий эффективности?** Четыре независимых критерия эффективности — это хорошо, но много. С практической точки зрения, хотелось бы иметь что-то вроде единого универсального критерия качества коммуникационной сети, способного количественно выразить, насколько одна сеть лучше другой при использовании в реальной работе. Как ни странно, такой интегральный критерий эффективности можно, хотя и с некоторыми существенными оговорками, построить.

В данном случае коммуникационная сеть волнует нас как источник помех для программиста в его попытках организовать совместную работу многих процессоров над единой задачей. Чем этих помех меньше, тем, очевидно, легче программисту разрабатывать параллельные приложения.

Интегрально уровень неудобства, вносимого коммуникационной сетью в работу программиста, можно выразить через ***характерный размер зерна параллелизма***.

Пусть в алгоритме имеется 100 независимых арифметических операций со 100 парами чисел. Имеет ли смысл поручать их 100 процессорам — по одной операции на процессор? Очевидно, нет: последующая синхронизация, включающая взаимный обмен результатами, уничтожит эффект такого распараллеливания.

Так же очевидно и то, что 100 независимых фрагментов по миллиону арифметических операций в каждом можно и нужно выполнять параллельно на 100 процессорах, поскольку накладной расход на последующую синхронизацию, скорее всего, будет оправдан 100-кратным ускорением расчета.

Где-то между одной операцией и миллионом операций проходит «грань осмысленности дробления работы от обмена до обмена». Эта грань и называется размером зерна параллелизма.

В литературе по параллельным вычислениям принято говорить о «мелкозернистых» (fine-grained), «среднезернистых» (middle-grained) и крупнозернистых (coarse-grained, буквально — грубозернистый) параллельных вычислительных системах и, соответственно, параллельных алгоритмах.

Мелкозернистыми, точнее, допускающими мелкозернистый параллелизм, обычно называют системы с общей памятью,



среднезернистыми — кластеры, крупнозернистыми — метакомпьютеры, т. е. системы из слабо связанных, сравнительно независимых машин.

Довольно очевидно, что чем меньше допускаемый данной вычислительной системой размер зерна параллелизма, тем больше у программиста свободы в выборе способа организации совместной работы многих процессоров над одной задачей, и тем больше вероятность того, что такая организация вообще окажется возможной.

# ОБЗОР ПРОЦЕССОРНЫХ И СЕТЕВЫХ РЕШЕНИЙ, ПРИМЕНЯЕМЫХ В СОВРЕМЕННЫХ КЛАСТЕРАХ

Данная глава посвящена сравнению основных процессорных и сетевых технологий, доступных для построения вычислительных кластеров в настоящее время. Для наглядности изложение построено в терминах выбора подходящих технических решений воображаемым разработчиком вычислительного кластера.

## 4.1. Выбор процессора

При построении вычислительного кластера разработчик, конечно, выбирает не процессор, а готовую рабочую станцию, которая будет использоваться в качестве узла. Неизбежно возникающие при этом технические и экономические вопросы, связанные с ценами на материнские платы различных производителей, надежностью вентиляторов и прочими факторами, так важными в реальной жизни, рассматривать не будем. Ограничимся сравнительным анализом тех свойств выбираемой в качестве узла рабочей станции, которые напрямую вытекают из используемого в ней процессора. Именно поэтому данный подраздел и называется «Выбор процессора».

Видов процессоров (по используемой системе команд), на базе которых строятся рабочие станции серийного выпуска, пригодные в качестве узлов вычислительного кластера, на сегодня существует, в основном, три:

Pentium — совместимые процессоры [7, 8],  
PowerPC [11],  
Itanium — 2 [7].

Эти три вида процессоров имеют совершенно разные системы команд, никак не совместимые друг с другом. Для каждой из этих систем команд существуют реализации ОС Linux, обычно применяемой для построения вычислительных кластеров.

Очевидной целью построения вычислительного кластера является максимально высокая производительность выполняемых

прикладных программ, которая очень сильно зависит от качества применяемых трансляторов. Трансляторы gnu, входящие в состав Linux [30], по качеству генерируемого кода оставляют желать лучшего. Применение профессиональных трансляторов весьма желательно, но они редко поставляются бесплатно. Поэтому важно понимать, на какой из систем команд потери от применения трансляторов gnu (если на профессиональный транслятор денег не нашлось) будут минимальными. Опыт показывает, что потери качества кода (для трансляторов gnu по сравнению с профессиональными, коммерчески доступными трансляторами) минимальны для Pentium — совместимых процессоров (примерно 30—50 %), а на менее распространенных процессорах могут возрастать кратно. То же верно и в отношении качества реализации самой ОС (количества еще не обнаруженных ошибок). Поэтому действует правило: **если нет явных, специальных оснований выбрать какой-либо совершенно конкретный процессор, выбирать надо Pentium — совместимый.**

Под «явными, специальными основаниями» понимаются особые требования к создаваемому кластеру — например, необходимость очень плотной компоновки, или снижения суммарного энергопотребления, или, наконец, просто способность выполнять готовые программы, приобретаемые в виде исполняемых модулей для совершенно конкретного процессора и операционной системы.

Часто рекламируемое превосходство процессоров Itanium-2 над Pentium — совместимыми процессорами, как показывает опыт, на практике зачастую не наблюдается, имеет место в основном при выполнении специально подобранных тестов, а не реальных задач.

К выбору типа процессора, на основе которого строится кластер, вплотную примыкают еще два важных решения: выбор числа процессоров (и/или процессорных ядер) в составе узла создаваемого кластера и выбор типа операционной системы (64-разрядной или 32-разрядной). Начнем с разрядности.

**Говоря о разрядности процессора**, часто имеют в виду, что 64-разрядный процессор быстрее выполняет вычисления над числами с двойной точностью, чем его 32-разрядный аналог. Это неверно. 32-разрядный Pentium или PowerPC выполняет операции над 64-разрядными числами ничуть не медленнее, чем аналогичный по частоте 64-разрядный. В данный момент 64-разрядность процессора означает, практически, только то, что

адреса команд и данных в программе являются 64-разрядными, т. е. отсутствует «32-разрядное» ограничение на суммарный размер программы и всех ее данных.

Вопрос на понимание: с учетом того, что современные микропроцессоры адресуют память с точностью до байта, каков, теоретически, максимальный суммарный размер программы и всех ее данных для 32-разрядного процессора?

Сказанное автоматически означает, что 64-разрядный и аналогичный ему 32-разрядный процессоры имеют разные системы команд. Верно ли это? Да, но не совсем.

Как Pentium — совместимые процессоры, так и процессоры PowerPC когда-то были 32-разрядными, а сегодня выпускаются в 64-разрядной версии. Однако из соображений совместимости программного обеспечения новая (64-разрядная) система команд была реализована не вместо, а в дополнение к уже существующей, 32-разрядной. Получился двухрежимный процессор (не путать с упоминавшейся выше многорежимностью в смысле «режим супервизора — режим пользователя»), т. е., фактически, два разных процессора в одном кристалле. Одновременно эти два режима работать не могут — в каждый конкретный момент времени процессор находится либо в 32-разрядном режиме (реализует старую систему команд), либо в 64-разрядном (реализует новую систему команд). Переключение режимов возможно через прерывания.

Если на машину, оснащенную таким процессором, устанавливается 32-разрядная операционная система, «ничего не знающая» о новой системе команд, процессор будет всегда работать в старом, 32-разрядном, режиме. Можно, однако, установить 64-разрядную операционную систему. Тогда сама ОС будет работать в 64-разрядном режиме, но исполняемые файлы прикладных программ она сможет запускать как «старые», так и «новые» (мы ведь уже знаем, что переключения процессора между разными прикладными программами, а также ядром, происходят через прерывания). Внутри прикладной программы переключение режимов невозможно: нельзя, например, оттранслировать свой код для 32-разрядного режима и собрать его с 64-разрядной библиотекой. Это то же самое, что пытаться собрать код для Itanium-2 с библиотекой объектных модулей для PowerPC.

Старая (32-разрядная) система команд Pentium называется IA-32. Новая (64-разрядная) называется x86-64, и была первоначально придумана и реализована AMD в процессорах Opteron

[8], но сейчас принята также и фирмой Intel для процессоров Pentium [7]. Таким образом, процессоры AMD на сегодня от Pentium — совместимых процессоров Intel по системе команд не отличаются. Система команд Itanium-2 — 64-разрядная, называется IA-64, и с системой команд x86-64 не имеет ничего общего, кроме разрядности. Itanium-2 — процессор одnoreжимный (32-разрядной системы команд у него нет). Современные версии PowerPC — двухрежимные, как и Pentium, разработанные для них версии Linux — тоже. Ни с Itanium-2, ни с Pentium/Opteron они по системе команд не совместимы [11].

**Выбирая число процессоров (и/или ядер)** на узле создаваемого кластера, следует выдерживать баланс между стремлением уменьшить размер и стоимость создаваемой установки, с одной стороны, и минимизацией конкуренции ядер за общую память и общие сетевые ресурсы — с другой.

Казалось бы, желание обеспечить как можно более тесную связь между процессорами (ядрами) в кластере диктует использование узлов с максимально возможным числом ядер на узле. В действительности все в точности наоборот.

С одной стороны, увеличивая связность внутри узла выше некоторого предела, мы автоматически снижаем связность между узлами (вряд ли 8-ядерный сервер будет оснащаться восемью адаптерами высокоскоростной коммуникационной сети для связи с себе подобными).

С другой стороны, как отмечалось ранее, объединение нескольких процессоров на общей памяти приводит, вообще говоря, к замедлению работы каждого из процессоров по причине их конкуренции за доступ к памяти. Эффект максимально проявляется в SMP-системах. Таковыми являются все многопроцессорные серверы Intel (Pentium) и IBM (PowerPC), а также все многоядерные процессоры (ядра в пределах одного процессора всегда конкурируют за память). Замедление может быть очень существенным (в 1,5, а то и почти в 2 раза для двух конкурирующих ядер), и сильно зависит от алгоритмической специфики задач и от транслятора.

Многопроцессорные серверы на базе AMD Opteron — это NUMA — системы, конкуренция за доступ к общей памяти при выполнении независимых процессов на разных процессорах (но не на разных ядрах одного процессора!) в них отсутствует, точнее — может отсутствовать, если сервер правильно построен и снабжен надлежащей версией ОС. Это заметное преимущество

при изготовлении вычислительного кластера на базе многопроцессорных узлов. Судя по заявлениям официальных представителей Intel на недавних научных конференциях, Intel планирует в самом скором времени также перейти на NUMA-архитектуру многопроцессорных серверов.

## 4.2. Выбор коммуникационной сети

Кластерная технология построения многопроцессорных вычислителей породила такое невиданное ранее явление, как производящееся серийно сетевое оборудование, предназначенное специально для суперкомпьютеров, т. е. не применяемое в сетевых технологиях общего назначения. Сети такого рода предназначены не для конкретной модели суперкомпьютера, а для «суперкомпьютера вообще», что и позволяет выпускать соответствующее оборудование пусть не такими громадными тиражами, как оборудование для офисных сетей, но все же серийно. Как правило, оборудование каждой из таких сетей выпускается единственным производителем (исключение — Infiniband [14]).

В то же время далеко не все кластеры оснащаются специальной высокоскоростной сетью — довольно часто делают кластеры на сетевом оборудовании общего назначения. Объясняется это, в первую очередь, высокой стоимостью специализированных сетевых решений — от 700—800 до нескольких тысяч долларов за узел.

Наиболее дешевое и доступное решение — *Gigabit Ethernet* (сеть общего назначения). Она характеризуется производительностью около 100 Мб/с, латентностью около 20—70 мкс, приемлемой потребностью в вычислительной мощности. Полнота сети зависит от используемого коммутатора и изменяется в широких пределах.

В середине 2007 г. появились сообщения о начале выпуска сетевых плат *10 Gigabit Ethernet*. О распространенности этой технологии на практике на момент написания настоящей главы говорить рано.

Самое старое и распространенное решение из специализированных — *Myrinet* [12]. До недавнего времени выпускался лишь вариант со скоростью 200 Мб/с, сейчас предлагается вариант с учетверенной производительностью в четыре раза выше и латентностью 3—6 мкс. Сеть оснащается коммутатором с извест-

ными и опубликованными характеристиками, полнота сети достаточно высока, потребность в вычислительной мощности — пренебрежимо мала.

Несколько лет назад серьезную конкуренцию Myrinet составляла сеть *SCI* [13]. Она имела примерно такую же, как старый вариант Myrinet, производительность, но примерно вдвое меньшую латентность. Сеть не предусматривает наличия коммутатора: узлы связываются в топологию (например, тор), в которой каждый узел является коммутатором транзитных сообщений. Сейчас эта технология мало заметна на рынке.

Главным конкурентом Myrinet сегодня является сеть *Infiniband* [14]. Основные ее параметры сходны с Myrinet, но производительность масштабируется выбором более «толстых» (с большим числом линий) сетевых кабелей. Самый «тонкий» (с одной линией) вариант кабеля дает производительность, как у старого варианта Myrinet. Число линий часто равно 4 или 8 (стандарт допускает до 32). Сеть коммутируемая, алгоритмы работы коммутаторов опубликованы.

В последнее время появились реализации Infiniband с удвоенной (DDR) и учетверенной (QDR) частотой в линиях передачи данных. Старый вариант, работающий на одинарной (как у старого варианта Myrinet) частоте, называется SDR. Соответственно, если производительность одной линии SDR Infiniband — около 200 Мб/с, то у DDR Infiniband она в 2 раза, а у QDR Infiniband — в 4 раза выше.

Такие сети, как *Giganet* и *Quadrics* [4], применяются довольно редко, в основном — фирмами, поставляющими кластеры «под ключ» и традиционно связанными именно с этими технологиями. Причина — высокая цена, но не очень существенные преимущества по сравнению с решениями, упомянутыми выше.

Помимо свойств сети как таковой, важно учитывать интерфейс подключения сети к компьютеру. В самом деле, если 32-разрядный вариант PCI с частотой 33 МГц в принципе не способен пропустить через себя больше 132 Мб/с, стоит ли подключать к нему Myrinet? Сейчас как раз происходит переход с традиционных, шинных интерфейсов PCI/PCI-X на современные интерфейсы PCI Express [7] и Hypertransport [8], которые сами по себе являются сетями с масштабируемым числом линий в канале. Одна линия PCI Express (Hypertransport в качестве периферийного разъема встречается гораздо реже) эквивалентна по скорости одной линии канала Myrinet или SDR Infiniband.

Чаще всего на материнских платах встречаются разъемы с числом линий 1, 4, 8 и 16. В последнее время начался переход на PCI Express удвоенной частоты. В этом случае одна линия равна по скорости линии канала DDR Infiniband.

Замечание. В английской терминологии линия в канале интерфейса переменной «толщины» называется «lane» (не «line»!), что в дословном переводе означает «переулок», или «дорожка». Об этом следует помнить, общаясь со специалистами на английском языке.

Кроме повышенных, по сравнению с Gigabit Ethernet, показателей по традиционным критериям эффективности, все специализированные сети обладают дополнительными функциональными возможностями, отсутствующими в сетях общего назначения, например, аппаратной поддержкой односторонних обменов, или RDMA (подробнее об односторонних обменах рассказано в разделе, посвященном технологиям параллельного программирования). 10G Ethernet также поддерживает RDMA. Сеть SCI, кроме того, обеспечивает аппаратный доступ к памяти удаленного узла в режиме прямой адресации (RMA, или NUMA), хотя и с некоторыми ограничениями.

На *программном обеспечении* специализированных сетей следует остановиться особо. Низкая, по сравнению с сетями общего назначения, латентность, а также поддержка RDMA (NUMA) достигаются использованием совершенно специальных системных запросов, не являющихся стандартными в ОС и специфичными только для конкретного вида сети. Программист может выдавать такие запросы, обращаясь к специальным низкоуровневым библиотекам, которые поставляются с каждой из специализированных сетей. Это принципиально отличается от ситуации с Ethernet, доступ к которой из прикладной программы всегда осуществляется через стандартную во всех ОС программную «настройку» — стек tcp/ip. Эта программная настройка рассчитана на наиболее общий случай использования локальной сети в составе Интернета, и сама по себе вносит очень заметный (если не решающий) вклад в латентность. Именно желанием обойти эту громоздкую настройку и продиктовано решение производителей специализированных сетей предоставить специальные, нестандартные библиотеки специальных же, нестандартных системных вызовов. Нестандартность их состоит не только в том, что эти системные вызовы не предусмотрены в Unix, но и в том, что для разных специализированных сетей они разные. Попытки стандартизации



таких интерфейсов для разных специализированных сетей принимаются, но проблема пока еще далека от решения.

Таким образом, с программистской точки зрения каждое сетевое устройство специализированной сети — это два устройства: одно быстрое, с низкой латентностью, но совершенно нестандартное, второе — обычный сетевой интерфейс под управлением стека tcp/ip. Конечно, второй вариант отличается гораздо большей латентностью, чем первый, но он стандартен. Можно ли сделать то же самое для Gigabit Ethernet? В принципе, можно, и такие попытки предпринимались, но проблема в том, что сетевые карты Ethernet от разных производителей не совместимы друг с другом по программистским интерфейсам к аппаратуре, и различных модификаций очень много. Разработчики «сокращенных протоколов» для Ethernet катастрофически не успевают за разнообразием сетевых плат, разработки затягиваются, а тем временем планируемый выигрыш в латентности достигается сам собой, просто в результате повышения частоты процессора, выполняющего стек tcp/ip.

О наличии двух «программистских входов» — стандартного, но медленного, и нестандартного, но быстрого — следует помнить, приобретая лицензионное прикладное программное обеспечение для кластеров в виде готовых исполняемых модулей. Как правило, оно собирается с коммуникационной библиотекой, рассчитанной на стандартный программистский интерфейс (tcp/ip), что приводит к значительно меньшему выигрышу от свойственной таким сетям низкой латентности, чем хотелось бы ожидать. В полной мере заявленные преимущества специализированных сетей, таким образом, могут рассчитывать ощутить лишь те, кто сам разрабатывает программы. Вариант библиотеки параллельного программирования MPI, работающий по «быстрому» входу, поставляется всегда.

Таким образом, краткие рекомендации по выбору коммуникационной сети выглядят примерно так: если Вы уверены, что Ваши задачи мало чувствительны к латентности, или вообще не очень чувствительны к производительности сети, то Ваш выбор — Gigabit Ethernet.

В противном случае следует ориентироваться на Myrinet или Infiniband, если требуется очень высокая производительность, то на Infiniband. В любом случае необходимо обращать внимание на сбалансированность пропускной способности сети и пропускной способности интерфейса сетевого адаптера с компьютером.

Современные периферийные интерфейсы (PCI Express, Hypertransport) предпочтительнее, чем PCI/PCI-X.

Наконец следует помнить о том, что оснащение кластера единственной сетью не желательно (и тем менее желательно, чем крупнее кластер). До сих пор молчаливо предполагалось, что сеть нашего кластера будет использоваться коммуникационной библиотекой. В действительности сеть вычислительного кластера выполняет еще, как минимум, две функции: *управление кластером* и *параллельный ввод-вывод* (поддержку доступа к общей для всех узлов файловой системе). Использование для реализации всех трех перечисленных функций единственной сети нежелательно по ряду причин, хотя в небольших кластерах так иногда и поступают. Дело не только в непредсказуемости производительности такой сети, и не только в том, что для разных функций желательны разные сетевые протоколы (для управления и ввода-вывода — tcp/ip, для коммуникационной библиотеки — специальные высокопроизводительные протоколы, упомянутые выше). Дело еще и в том, что сетям со стандартными протоколами присущи *сетевые перегрузки* [2], т.е. «плавающие» отказы в выполнении сетевых запросов в отсутствие неисправностей отдельных компонентов. Сетевые перегрузки органически присущи ip — сетям, их нельзя искоренить, но можно и нужно всемерно снижать их вероятность. По этой причине очень полезно выделить для каждой из трех сетевых функций кластера — поддержки коммуникационной библиотеки, управления и поддержки ввода-вывода — по одной физически отдельной сети. Даже если все три сети будут сетями Ethernet. По крайней мере, напряженный трафик в одной сети не «сломает» никакую другую.

Наконец — замечание о **SAN/NAS**. Доступ к общей файловой системе в вычислительном кластере, зачастую, является даже более узким местом, чем коммуникации между узлами. Это довольно естественно — большие машины обрабатывают большие объемы данных, которые надо откуда-то брать и куда-то класть. Для этого есть готовые, предлагаемые на рынке аппаратные решения — специальные сети, интегрированные с мощным файловым сервером, применяемые для поддержки параллельного ввода-вывода и ни для чего другого. Это решение идеально для вычислительных кластеров, во всех отношениях, кроме одного — стоимости. Такая сеть может оказаться заметно дороже всего остального оборудования, из которого состоит кластер.

Альтернативой такому оборудованию является программная организация распределенной по локальным дискам вычислительных узлов файловой системы. В принципе это очень хороший способ расширения узкого места при доступе узлов к единой файловой системе. Однако он технически очень сложен в реализации, особенно с учетом возможности временного вывода узла из эксплуатации для ремонта, сетевых перегрузок и прочих «мелочей жизни». Существуют, хотя и очень дорого стоят, коммерчески доступные реализации таких систем весьма высокого качества. Свободно распространяемые реализации, такие, как PVFS [31], обладают рядом недостатков с точки зрения отказоустойчивости, что мешает применять их на кластерах с большим числом узлов.

# МОДЕЛИ И ТЕХНОЛОГИИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

К этому моменту мы накопили вполне достаточно представлений о том, из чего строятся вычислительные кластеры, чтобы перейти к главному: к рассмотрению собственно программного обеспечения параллельной обработки данных. Как и программное обеспечение «обычных» компьютеров, программное обеспечение вычислительных кластеров распадается на две довольно автономные части: программное обеспечение для разработки параллельных программ (системы программирования) и программное обеспечение управления ресурсами вычислительной установки в процессе работы (операционные системы). Обе части программного обеспечения сильно отличаются от того, что мы привыкли видеть на «обычных» компьютерах, и не могут быть поняты без всех тех сведений об аппаратуре, изложению которых мы уделили так много места в предшествующих главах.

В этой главе изучим системы параллельного программирования. Системы управления ресурсами будут рассмотрены в гл. 6.

На тему, обсуждаемую в данной главе, написано довольно много хороших и подробных учебников, в том числе на русском языке, например [62].

**Моделью параллельного программирования** называется набор понятий, которыми оперирует программист, записывая параллельную программу. Например, «модель обмена сообщениями между процессами в рамках рандеву».

**Технологией параллельного программирования** называется конкретное программное воплощение модели, например: «технология программирования с помощью библиотеки MPI (или PVM)».

Одна модель может быть представлена несколькими технологиями.

Прежде чем переходить к рассмотрению моделей и технологий, убедимся в их праве на существование. В самом деле, идеальная система параллельного программирования должна была бы уметь

автоматически строить параллельный код по тексту последовательной (однопроцессорной) программы. В этом случае обсуждение всех прочих технологий имело бы чисто исторический интерес. **Основной постулат параллельного программирования для систем с распределенной памятью** звучит так: «Для реальных программ, написанных на традиционных языках программирования высокого уровня, автоматическое порождение параллельной программы по тексту ее последовательного прототипа невозможно».

Для демонстрации верности этого утверждения достаточно сопоставить всего несколько чисел.

В системах без общей памяти латентность обмена данными имеет порядок микросекунд. Следовательно, время работы процессора между обменами должно быть, в среднем, гораздо больше — иначе накладные расходы на коммуникации будут слишком велики.

Современный процессор за 10 мкс (добавим хотя бы один порядок к «идеальной латентности») выполняет примерно 30—50 тыс. операций. Для того чтобы выделить фрагмент программы такой длины в качестве независимого куска, не связанного с вычислениями в других процессорах, наш гипотетический распараллеливающий транслятор должен его проанализировать и убедиться в отсутствии зависимостей. Однако в реальной программе, содержащей ввод-вывод, обращение к внешним процедурам и прочую динамику, возможен анализ лишь линейных фрагментов кода (30 тыс. команд — это примерно 300—500 операторов). Такой длины линейных фрагментов в программах не бывает. Демонстрация верности основного постулата завершена. Мы провели ее только для систем без общей памяти, но очень грубо и с большим запасом, просто чтобы показать, чем в принципе определяется возможность решения проблем такого рода. В действительности на данный момент и для SMP-систем практических результатов полностью автоматического «распараллеливания» реальных программ неизвестно.

Проще говоря, операторы современных языков программирования слишком малы (по времени их выполнения) по сравнению с латентностью современных сетей, чтобы можно было надеяться автоматически «закрыть» эту латентность линейным фрагментом программы, а фрагмент, не являющийся линейным, не поддается должному автоматическому анализу. Следовательно, мы вынуждены признать, что построение параллельной про-

граммы при современном уровне техники — задача не механическая, а осмысленная, требующая участия человека. Технологии, которые будут рассмотрены, как раз и являются конкретными формами записи такого участия.

Для сравнительного анализа моделей и технологий параллельного программирования у нас недостаточно примеров того, как именно пишутся параллельные программы. В подразд. 1.5 был приведен один пример модельной программы и построена «на бумаге» ее параллельная реализация. В настоящем подразделе, по мере необходимости, этот пример будет использован в иллюстративных целях. Однако способ параллельной реализации алгоритма Якоби обманчиво прост, и не хотелось бы создавать ложного впечатления, что все или почти все техники параллельной реализации вычислений в главном на него похожи. Далее будет приведено несколько примеров технологий параллельного программирования, с помощью которых метод Якоби записывается легко и изящно, в то время как целый класс широко применяемых методов параллельных вычислений не записывается вообще. В следующем подразделе приводится пример модельного алгоритма, принципиально отличающегося от метода Якоби по технике параллельной реализации.

## **5.1. Способы параллельной обработки данных**

В фундаментальной монографии «Параллельные вычисления» [3] выделяется два широко распространенных, принципиально различных способа параллельной обработки данных — собственно параллелизм и конвейер. Пример собственно параллелизма был приведен — это параллельный вариант метода Якоби. В критическом цикле программы выделяются независимые вычисления, и они «раздаются» равными порциями разным процессорам, работающим «в фазе»: каждый выполнил очередной шаг обработки — все, кому надо, обменялись недостающими данными — все перешли к следующему шагу. В каждый момент времени каждый процессор выполняет одну и ту же итерацию, но с разными данными, обработка которых на этой итерации не зависит от обработки других порций данных на этой же итерации.

Конвейерная обработка данных принципиально отличается от этого простого шаблона. Рассмотрим пример альтернативно-

го численного метода решения той же (по физическому смыслу) задачи математической физики, т. е. задачи Дирихле для уравнения Лапласа. Это — метод верхней релаксации, и записывается он так:

```
#include <stdio.h>
/**/
#define MX 640
#define MY 480
#define W 0.5
#define NITER 10000
#define STEPITER 100
static float f[MX][MY];
/**/
int main( int argc, char **argv )
{
    int i, j, n;
    FILE *fp;
/**/
    printf( "Solving heat conduction task on %d
            by %d grid\n", MX, MY );
    fflush( stdout );
/* Initial conditions: */
    for ( i = 0; i < MX; i++ )
    {
        for ( j = 0; j < MY; j++ )
        {
            f[i][j] = df[i][j] = 0.0;
            if      ( (i == 0)
                    || (j == 0) ) f[i][j] = 1.0;
            else if ( (i == (MX-1))
                    || (j == (MY-1)) ) f[i][j] = 0.5 ;
        }
    }
/* Iteration loop: */
    for ( n = 0; n < NITER; n++ )
    {
        //      if ( !(n%STEPITER) )
            printf( "Iteration %d\n", n );
/* Step of calculation starts here: */
        for ( i = 1; i < (MX-1); i++ )
        {
            for ( j = 1; j < (MY-1); j++ )
```

```

        {
            f[i][j] = W * ( f[i][j+1] +
                           f[i][j-1] + f[i-1][j] +
                           f[i+1][j] )
                        * 0.25 + (1-W) * f[i][j];
        }
    }
/* Step is over: */
}
fp = fopen( "progre.dat", "w" );
for ( i = 1; i < (MX-1); i++ )
    fwrite( f[i]+1, MY-2, sizeof(f[0]
[0]), fp );
fclose( fp );
return 0;
}

```

Как и в случае рассмотрения метода Якоби, не будем обсуждать ни физическую, ни вычислительно-математическую сторону вопроса: нас волнует параллельная реализация записанного здесь алгоритма, и ничего более.

При всем кажущемся сходстве с методом Якоби, этот алгоритм обладает крайне неприятным для нас свойством: в его итерации, на первый взгляд, нет независимых вычислений. Расчет движется «слева направо и сверху вниз» по полю величин, и каждое новое значение этого поля зависит от всех, вычисленных ранее. Зачем математики придумали такой неудобный для параллельной реализации метод? Затем, что он быстрее сходится. Иногда на порядок быстрее. Как выполнить параллельную реализацию, если все данные зависят друг от друга?

Поделим данные между процессорами таким же способом, что и в случае метода Якоби. Если поделить тем же способом и работу, то процессоры будут работать строго по очереди, и никакого скоростного выигрыша не получится. Однако, ничто не мешает процессору, вычислив последнюю строку своей порции поля величин, обменяться со следующим процессором и сразу же приступить к следующей итерации. В итоге получим примерно такую картину:

- третий процессор приступил к первой итерации;
- в это время второй процессор уже выполняет вторую;
- а первый — третью;
- наконец, нулевой — четвертую.



Цель достигнута — процессоры работают одновременно, правда, не сразу, а лишь начиная с момента, когда «фронт вычислений» дойдет до последнего процессора. Если число итераций много больше числа процессоров (а это практически всегда так), этой неприятностью можно пренебречь.

Возможность одновременной работы процессоров достигнута за счет того, что нам удалось выделить вычисления, не зависящие друг от друга, не просто для разных данных, а еще и **на разных фазах расчета**. Такой способ параллельной обработки данных и называется конвейером. Многократно повторяющийся фрагмент вычислений (в данном случае — итерация) разбивается на последовательно сменяющие друг друга фазы (расчет каждой полосы — фаза). Независимость вычислений с разными порциями обрабатываемых данных достигается сдвигом по фазе — в точной аналогии со сборочным конвейером.

Проще говоря, если параллелизм независимых вычислений можно назвать **параллелизмом в пространстве**, то конвейерный параллелизм естественно называть **параллелизмом во времени**.

В завершение отметим, что правильная параллельная реализация метода верхней релаксации, допускающая расчет также и на многообластной сетке, выглядит несколько сложнее, и реализует параллельный расчет в рамках одной итерации, как и параллельная реализация метода Якоби. При этом все равно используется параллелизм конвейерного типа — другого в этом алгоритме просто нет. Тексты обоих вариантов параллельной реализации метода верхней релаксации, снабженные необходимыми пояснениями, приводятся в прил. 1.

## 5.2. Аспекты программистской модели

Модель параллельного программирования, как и любая программистская модель, имеет два аспекта, или два «лица».

С одной стороны, она призвана быть возможно более удобной для программиста — это «лицо, обращенное к программисту». В этом смысле программистская модель — чистая «игра ума», логическая абстракция, созданная одним программистом для другого.

С другой стороны, программистская модель должна быть адекватной абстракцией коммуникационного оборудования, т. е.

допускать эффективную реализацию («лицо, обращенное к аппаратуре»). Если модель элегантна и легка в постижении, но создает у программиста неверное представление об используемом коммуникационном оборудовании, не способствуя эффективному использованию этого оборудования, то толку от нее мало.

Таким образом, ключ к пониманию моделей и технологий параллельного программирования — в понимании их соответствия аппаратуре. Всего несколько лет назад коммуникационная аппаратура суперкомпьютеров была логически очень однородна. Подавляющее большинство составляли МРР-системы на базе сетей двусторонних обменов. Этим обусловлено богатство моделей и технологий параллельного программирования, ориентированных именно на такое оборудование. Как отмечалось в подразд. 1.3, сейчас ситуация изменилась, но для того, чтобы проследить эти изменения, у нас пока мало конкретных примеров. Приведем их.

### **5.3. Модель: явный двусторонний обмен сообщениями**

Явный двусторонний обмен сообщениями — самая старая и самая «ручная» (наименее автоматизированная, наиболее близкая к аппаратуре на базе сетей двусторонних обменов) модель. В основном она соответствует модели взаимодействующих процессов Хоара (CSP) [32]. В предшествующем изложении эта модель неоднократно упоминалась, но сколько-нибудь формализованных определений не было дано. Приведем краткое формализованное описание модели.

Пусть имеется набор из  $N$  процессов, занумерованных от 0 до  $N-1$ . Каждый из процессов связан с каждым из остальных парой противоположенных каналов передачи сообщений. Таким образом, с каждым процессом соединена, в общей сложности,  $(N - 1)$  пара каналов. Пары каналов в каждом из процессов занумерованы номерами тех процессов, с которыми связаны противоположные концы каналов. Так, в процессе номер 7 пара каналов, один из которых идет к процессу номер 5, а второй — от процесса номер 5, имеет номер 5. Соответственно, в процессе номер 5 эта пара каналов имеет номер 7 (рис. 5.1).

Каждый из процессов способен выполнить операцию передачи сообщения в канал, идущий к другому процессу, и операцию

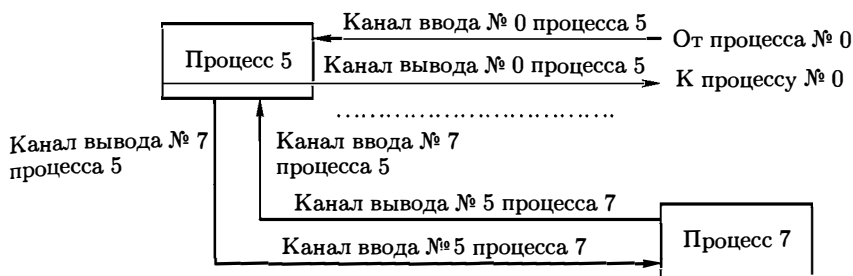


Рис. 5.1. Нумерация процессов и каналов при связи каждого с каждым (полностью показаны только связи процессов 5 и 7)

приема сообщения из канала, идущего от другого процесса. Операндами операции являются:

- номер канала (он же номер процесса, которому надо передать или от которого надо принять сообщение);
- длина сообщения;
- адрес (передаваемого сообщения или места в памяти, куда требуется сообщение принять).

Для того чтобы передача сообщения состоялась, необходимо и достаточно, чтобы процесс на передающем конце канала выполнил операцию передачи сообщения, а процесс на приемном конце — операцию приема. Требуется, чтобы длины сообщений при этом совпадали: иначе состояние канала становится не определенным.

Во время передачи сообщения канал автоматически обеспечивает синхронизацию в режиме *рандеву*, или «ожидания опоздавшего».

Например, если процесс номер 7 выполнил операцию выдачи сообщения процессу номер 5, а процесс номер 5 еще не выполнил операцию соответствующего приема, передача данных от процесса номер 7 в канал номер 5 автоматически задерживается. При этом, с точки зрения процесса номер 7, операция выдачи сообщения «идет, но бесконечно медленно». Она завершится, когда процесс номер 5 выполнит операцию приема, и данные действительно будут, как минимум, переданы в канал. Операция приема в процессе номер 5 завершится, когда данные будут доставлены из канала по месту назначения. При этом не важно, был ли процесс номер 7 уже готов их выдавать, когда процесс номер 5 запустил прием, или процессу номер 5 пришлось долго ждать, пока процесс номер 7 начнет передачу.

Вспомнив наш гипотетический простейший кластер (см. подразд. 2.3.1), полученный соединением com-портов двух компьютеров с помощью нуль-модемного кабеля, заметим, что описанная модель в высшей степени адекватна свойствам такого коммуникационного оборудования. Рандеву можно обеспечить, программно отслеживая значение бита готовности в регистре состояния com-порта. Для передачи данных требуется «взаимное согласие» двух сторон: передающей и принимающей. Поведение программы, обеспечивающей синхронизацию, никак не зависит от того, началась ли уже передача данных на противоположном конце, или нет. Она просто ждет отправки или поступления очередного байта данных, и только.

Аппаратура локальных сетей устроена, с точки зрения регистров управления сетевым адаптером, гораздо сложнее, чем com-порт, а каналы «от каждого к каждому» в коммутируемой сети — сущности скорее виртуальные, чем физические. Однако и это оборудование, особенно в сочетании с программным обеспечением tcp/ip в составе ОС, на логическом уровне хорошо описывается в терминах двустороннего обмена сообщениями. Наиболее принципиальны два свойства модели: двусторонность, т. е. необходимость согласованных действий на двух сторонах для любой передачи данных, и рандеву, т. е. автоматическая принудительная синхронизация с ожиданием опоздавшего.

В «докластерную эру» эта программистская модель была представлена многочисленными библиотеками, каждая из которых была ориентирована на конкретную модель суперкомпьютера. Затем появилась технология PVM [4, 33, 62], предназначенная, в основном, для кластеров невыделенных рабочих станций.

Разработчики этой технологии изначально не стремились к высокой эффективности, поскольку сетевое оборудование все равно было довольно слабым, и уделяли основное внимание простоте программистских интерфейсов. В результате получилась система, не допускающая действительно эффективной реализации. В силу исторических причин она используется до настоящего времени в некоторых старых программистских коллективах, в основном — в Европе.

Затем появился стандарт *MPI* (*Message Passing Interface* — Интерфейс передачи сообщений) [4,62], на сегодня — главная технология этой программистской модели. Конкретных реализаций *MPI* существует много, в том числе — как минимум две

свободно распространяемых в виде исходных текстов реализации весьма высокого качества.

### 5.3.1. Технология: MPI

Среди всех технологий, которые нам предстоит рассмотреть, технология MPI занимает особое место. Как отмечалось выше, модель двусторонних обменов в течение довольно долгого времени рассматривалась как единственная модель, в полной мере ориентированная на аппаратуру. Считалось (и это было недалеко от истины), что любые другие программистские модели могут быть только производными от этой базы высокоуровневыми надстройками. Конкретно это означает, что именно модель двусторонних обменов способна выразить в программистских понятиях истинную природу коммуникационного оборудования, а любая другая модель имеет право на существование лишь постольку, поскольку она сводится к модели двусторонних обменов без существенных потерь эффективности. Впоследствии развитие возможностей коммуникационного оборудования изменило ситуацию, но модель двусторонних обменов осталась той отправной точкой, в терминах сходства с которой или отличия от которой описываются и объясняются все остальные модели и технологии. Словом, модель двусторонних обменов — «классика» среди прочих моделей.

С другой стороны, технологию MPI можно с полным на то основанием считать «классикой» среди технологий модели двусторонних обменов. MPI является не одной из стихийно сложившихся библиотек, используемых многими в силу традиции, а планомерно продуманным стандартом. Разработчики стандарта сознательно старались включить в него все лучшее из более старых, стихийно сложившихся технологий, и им это во многом удалось.

MPI-стандарт на внешний интерфейс коммуникационной библиотеки, к функциям которой обращается ветвь параллельной программы. Наиболее распространенные варианты этой библиотеки рассчитаны на обращение из программ, написанных на C и Фортране. Ниже приводится краткий обзор возможностей в терминах C.

Важно понимать, что приводимое ниже рассмотрение MPI — это именно обзор, а не краткое описание стандарта. Будем стремиться не столько к полноте перечисления функций MPI, сколько к пониманию внутренней логики набора возможностей.

**Минимальное подмножество функций.** Стандартный файл заголовков, который должна включать программа, использующая MPI, называется `mpi.h`. В нем, в частности, определяются многочисленные символические константы, которые будут встречаться в дальнейшем изложении.

В отличие от системы PVM, MPI стандартизует только интерфейс коммуникационной библиотеки, но не пользовательский интерфейс системы запуска программ. Конкретный вид команд, с помощью которых запускаются на счет программы с использованием MPI, стандартом не определяется, и в разных реализациях MPI может быть различным.

Все упомянутые в приведенном далее обзоре функции MPI, для которых явно не оговорено противное, имеют тип `int` и возвращают код ответа, причем успешному срабатыванию отвечает значение `MPI_SUCCESS`.

Первая из функций MPI, которую должна вызвать программа пользователя, является функция инициализации MPI.

Обращение к ней имеет вид:

```
MPI_Init( &argc, &argv );
```

Здесь `argc` и `argv` — стандартные аргументы функции `main`. Цели их передачи в функцию инициализации MPI были рассмотрены в подразд. 1.5.3. Реализация MPI не обязана использовать `argc` и `argv` тем способом, который был описан в подразд. 1.5.3, но может это делать. Соответственно программа, использующая MPI, не должна как-либо использовать `argc` и `argv` до того, как отработает `MPI_Init`.

В конце работы с MPI программа должна вызвать функцию `MPI_Finalize()`. Аргументов у нее нет. После обращения к этой функции программа не должна вызывать каких-либо функций MPI, включая `MPI_Init`.

Как отмечалось в подразд. 1.5.3, первое, что должна сделать ветвь параллельной программы — это узнать свой номер и общее число себе подобных. Это делается обращениями к функциям:

```
MPI_Comm_rank( MPI_COMM_WORLD, &rank );  
MPI_Comm_size( MPI_COMM_WORLD, &size );
```

Здесь `rank` и `size` — собственный номер и число ветвей в параллельной программе, соответственно. Это возвращаемые целочисленные величины. Ветви нумеруются с нуля и подряд. В процессе работы программы ветви можно объединять в подмноже-

ства, внутри каждого из которых определена собственная нумерация. Такие подмножества называются **коммуникаторами**. Соответственно все функции MPI, в числе аргументов которых так или иначе фигурирует номер ветви или количество ветвей, подразумевают указание коммуникатора, чтобы было понятно, о номере в каком именно подмножестве идет речь. По умолчанию имеется коммуникатор, включающий все ветви параллельной программы, и обозначается он константой MPI\_COMM\_WORLD.

Пусть ветвь параллельной программы желает послать другой ветви в качестве сообщения массив двойной точности `dparr` длиной `n` элементов. Это можно сделать обращением к функции:

```
MPI_Send( dparr, n, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD );
```

Здесь MPI\_DOUBLE — константа, обозначающая тип данных передаваемого массива, `dest` — целочисленный номер ветви — получателя (кому послать), `tag` — тэг сообщения, последний аргумент — коммуникатор (см. выше).

Зачем указывать тип данных передаваемого массива, не проще ли стандартным для C способом указать длину сообщения в байтах? Проще, но при одном условии: узел кластера, которому мы посылаем сообщение, должен совпадать с узлом — отправителем по машинному формату представления чисел. В подавляющем большинстве кластеров выделенных рабочих станций это так, но библиотека MPI, вообще говоря, рассчитана на гетерогенные кластеры, т.е. на кластеры, состоящие из узлов с разной системой команд и, возможно, разным машинным представлением чисел. Кластер невыделенных рабочих станций, например, вполне может быть гетерогенным. Также вполне может оказаться гетерогенным комплекс из нескольких кластеров выделенных рабочих станций, объединенных в единый «мета — кластер».

Тэг сообщения — это входной целочисленный аргумент, значение которого передается вместе с сообщением и позволяет фильтровать сообщения при приеме (см. ниже).

Для приема сообщения ветвь параллельной программы должна воспользоваться функцией:

```
MPI_Recv( dparr, n, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, &status );
```

Здесь `dragg` — массив, в который следует принять сообщение, `n` — его длина в компонентах, `MPI_DOUBLE` — обозначение типа принимаемых данных, `src` — номер ветви отправителя (от кого принять). В отличие от случая посылки сообщения (и от классического варианта модели двусторонних обменов), `src` можно задать равным специальной константе `MPI_ANY_SOURCE`, что будет означать «принять от любого, кто мне послал». Значение тэга — входной аргумент, и смысл его — «принять сообщение только в том случае, если оно было послано с указанным значением тэга». Указав в качестве этого аргумента специальную константу `MPI_ANY_TAG`, можно принять сообщение с произвольным тэгом.

Как мы видим, аппарат тэгов позволяет принимать сообщения от заданного узла не в том порядке, в котором они были посланы. Это серьезнейший источник накладных расходов в реализации — ведь сообщения «не с теми тэгами», которые отправитель посылает, а получатель пока не хочет принимать, библиотека должна накапливать внутри себя, ничего не зная ни об их количестве, ни об их суммарной длине, чтобы пропустить вперед те сообщения, которые получатель, возможно, принять захочет. Хорошая реализация `MPI` не должна включать в себя заметных накладных расходов времени на обработку тэгов для «правильных» случаев, т. е. для программ, которые таким переупорядочиванием сообщений реально не пользуются.

Таким образом, при успешном срабатывании `MPI_Recv` имеются как минимум два потенциально неизвестных значения — номер отправителя и тэг отправленного сообщения. В действительности их три — `MPI` позволяет принимать сообщения не обязательно той длины, которая указана во втором аргументе, но и меньшей (то есть можно попытаться принять сообщение длиной 100, а потом убедиться, что пришло сообщение длиной всего 55). Все эти потенциально неизвестные значения «оседают» при завершении операции приема в специальной структуре типа `MPI_Status`. Именно этот тип имеет последний возвращаемый аргумент.

Чтобы извлечь из структуры типа `MPI_Status` длину прошедшего сообщения в целочисленную переменную `count`, надо обратиться к функции:

```
MPI_Get_count( &status, MPI_DOUBLE, &count );
```

Два других потенциально неизвестных значения доступны как поля структуры: `status.MPI_SOURCE` и `status.MPI_TAG`.



При отладке эффективности параллельных программ приходится постоянно измерять затраченное в тех или иных местах программы время. Для этого в MPI существует функция `MPI_Wtime()` (без аргументов). Эта функция имеет тип `double` и возвращает время в секундах, прошедшее с некоторого начального момента. Вычитая друг из друга значения, полученные последовательными обращениями к `MPI_Wtime()`, можно узнать, сколько времени прошло от одного обращения до другого.

Само значение, возвращаемое этой функцией, никакого «физического смысла» не имеет. Совершенно неверно было бы полагать, например, что это — некоторое глобальное, единое для всех узлов, время.

Подведем промежуточный итог.

В нашем распоряжении имеются семь функций, которые, казалось бы, полностью покрывают наши представления о модели двусторонних обменов сообщениями. Создается впечатление, что в первом приближении их достаточно для написания любой параллельной программы в терминах этой модели. Правда, внешний интерфейс этих функций кажется несколько переусложненным, причем сложность эта все равно не приводит к полной ясности: что будет, например, если послать массив как целочисленный, а принять — как массив чисел двойной точности?

Впрочем, способов поставить программное обеспечение в тупик «хитрой» ошибкой немало в любом программистском интерфейсе — специфика параллельной обработки данных здесь ни при чем. Интереснее другое — действительно ли рассмотренных нами семи функций практически достаточно для реальных приложений? По мере получения ответа на этот вопрос мы, возможно, поймем, зачем в MPI предусмотрено еще около 120 функций.

Попытаемся написать с использованием минимального подмножества MPI реализацию нашей модельной задачи о прогреве кирпича методом Якоби и реализовать ту общую схему, которая была разобрана в подразд. 1.5.2.

```
#include <stdio.h>
#include <mpi.h>
/****/
#define MX 640
#define MY 480
#define NITER 10000
#define STEPITER 100
```

```

        static float  f[MX][MY];
        static float df[MX][MY];

/****/
        int main( int argc, char **argv )
        {
            int i, j, n, m, mx, size, rank;
            MPI_Status status;

/****/
            MPI_Init( &argc, &argv );
            MPI_Comm_size( MPI_COMM_WORLD, &size );
            MPI_Comm_rank( MPI_COMM_WORLD, &rank );
            mx = ((MX-2)+size-1)/size;
            if ( rank == (size-1) )
            {
                mx = (MX-2) - (size-1)*mx;
            }
            mx += 2;
            printf( "%d of %d is solving heat
                    conduction task on %d
                    by %d grid\n",rank, size, mx, MY );
            fflush( stdout );

/* Initial conditions: */
            for ( i = 0; i < mx; i++ )
            {
                for ( j = 0; j < MY; j++ )
                {
                    f[i][j] = df[i][j] = 0.0;
                    if      ( ((i == 0) && (rank == 0))
                        || (j == 0) ) f[i][j] = 1.0;
                    else if ( ((i == (mx-1)) &&
                        (rank == (size-1)))
                        || (j == (MY-1)) ) f[i][j] = 0.5;
                }
            }

/* Iteration loop: */
            for ( n = 0; n < NITER; n++ )
            {
                if ( !(n%STEPITER) )
                    printf( "Iteration %d\n", n );
/* EXCHANGE THE FIRST AND LAST ROWS HERE!!!! */
                .....
/* Step of calculation starts here: */
                for ( i = 1; i < (mx-1); i++ )

```

```

    {
        for ( j = 1; j < (MY-1); j++ )
        {
            df[i][j] = ( f[i][j+1] + f[i][j-1] +
                        f[i-1][j] + f[i+1][j] )
                        * 0.25 - f[i][j];
        }
    }
    for ( i = 1; i < (mx-1); i++ )
    {
        for ( j = 1; j < (MY-1); j++ )
        {
            f[i][j] += df[i][j];
        }
    }
}
/* Calculation is done, F array is a result: */
if ( rank == 0 )
{
    fp = fopen( "progrex.dat", "w" );
    fclose( fp );
}
else
{
    MPI_Recv( &n, 1, MPI_INT, rank-1,
              MPI_ANY_TAG, MPI_COMM_WORLD,
              &status );
}
fp = fopen( "progrex.dat", "a" );
for ( i = 1; i < (mx-1); i++ )
    fwrite( f[i]+1, my-2, sizeof(f[0][0]), fp );
fclose( fp );
if ( rank < (size-1) )
    MPI_Send( &n, 1, MPI_INT, rank+1, 100,
              MPI_COMM_WORLD );
MPI_Finalize();
return 0;
}

```

Значение `mx` в нашей программе — это число строк суммарного массива, доставшееся данному процессу. Вычисление этого значения записано перед первым в программе обращением к `printf` и имеет следующий смысл:

— уменьшаем суммарное число строк на 2, чтобы получить число обрабатываемых строк (первая и последняя строки не обрабатываются);

— делим число обрабатываемых строк нацело на число процессов, с округлением в большую сторону, получая число строк, обрабатываемых одним процессом, кроме, быть может, последнего;

— поскольку число строк могло нацело не поделиться на число процессов, для последнего процесса вносим поправку: обрабатываемое им число строк должно быть равно общему числу обрабатываемых строк, минус суммарное число строк, обрабатываемых всеми остальными;

— наконец, добавляем в каждом процессе к числу обрабатываемых строк две строки — буферные зоны для копирования краев или, если процесс первый или последний, для граничных условий.

Теперь обратим внимание на изменения, по сравнению с однопроцессорным вариантом, в конце программы, связанные с записью результатов расчета в файл. Смысл их состоит в том, чтобы обеспечить поочередную запись в один и тот же файл данных, насчитанных каждым процессом, строго в порядке номеров процессов. С этой целью процессы синхронизируются, передавая друг другу по цепочке значение переменной *n*. Само значение *n* при этом совершенно не волнует — передача осуществляется только ради самого факта синхронизации.

Остальной текст программы, как легко видеть, мало отличается от однопроцессорного варианта. Впрочем, программа не дописана: вместо многоточия после комментария:

```
/* EXCHANGE THE FIRST AND LAST ROWS HERE!!!! */  
.....
```

следует вставить обращения к функциям MPI, которые, собственно, и осуществляют обмен сообщениями. При этом должны произойти те обмены, которые изображены стрелками в правой части рис. 1.3 (см. подразд. 1.5.2). Выпишем их сначала по отдельности, сопровождая необходимыми пояснениями. Встречающиеся в этих пояснениях упоминания «верхних» и «нижних» краев и буферных зон следует понимать в смысле их расположения на упомянутом рисунке. Итак:

— во всех процессах, кроме нулевого, следует послать предыдущему процессу верхний край своей локальной порции:

```
MPI_Send( &f[1][1], MY-2, MPI_DOUBLE, rank-1,
          100, MPI_COMM_WORLD );
```

а также принять в верхнюю буферную зону посланный им нижний край его локальной порции:

```
MPI_Recv( &f[0][1], MY-2, MPI_DOUBLE, rank-1,
          MPI_ANY_TAG, MPI_COMM_WORLD,
          &status );
```

— во всех процессах, кроме последнего, следует послать следующему процессу нижний край своей локальной порции:

```
MPI_Send( &f[mx-2][1], MY-2, MPI_DOUBLE,
          rank+1, 100, MPI_COMM_WORLD );
```

а также принять в нижнюю буферную зону посланный им верхний край его локальной порции:

```
MPI_Recv( &f[mx-1][1], MY-2, MPI_DOUBLE,
          rank+1, MPI_ANY_TAG,
          MPI_COMM_WORLD, &status );
```

Вроде бы ничего другого в приведенную выше заготовку текста программы добавлять не требуется. Надо лишь расположить выписанные здесь обращения к функциям MPI в правильном порядке. А какой порядок правильный? Так ли важно, с каким соседом процесс обменивается сначала — с предыдущим или с последующим, а с каким — потом? Совершенно не важно — но при одном условии. Необходимо, как минимум, обеспечить саму возможность randevу. Здесь нас поджидает проблема, казалось бы, на совершенно ровном месте. В самом деле, покажем, что единого для всех процессов «правильного порядка», в котором следовало бы расположить приведенные выше обращения к функциям MPI, **не существует**.

Вспомним еще раз, что программа, которую мы сейчас пишем — это текст ветви параллельной программы, т.е. каждый процесс ведет себя так, как в ней написано. Итак, пусть **каждый** процесс начинает обмены данными с попытки передать предыдущему (кроме нулевого — у него предыдущего нет):

```
if (rank > 0)      MPI_Send( &f[1][1], MY-2,
                          MPI_DOUBLE, rank-1, 100,
                          MPI_COMM_WORLD );
```

Рассмотрим выполнение этого оператора в процессе, номер которого равен, например, 5. Он посылает сообщение процессу

номер 4, и для того, чтобы эта посылка завершилась, процесс номер 4 должен был бы выполнить соответствующий прием. Пока процесс номер 4 этого не сделает, процесс номер 5 будет «висеть» внутри обращения к MPI\_Send. Но процесс номер 4 совершенно аналогичным образом ждет «ответной любезности» от процесса номер 3, тот, в свою очередь, от процесса номер 2, и т.п. Единственный процесс, который не выполняет посылку предыдущему — это процесс номер 0. Если в нем сразу после этой посылки записан прием от предыдущего в нижнюю буферную зону, дело, наконец, сдвинется с места:

```

if (rank > 0)          MPI_Send( &f[1][1], MY-2,
                           MPI_DOUBLE, rank-1, 100,
                           MPI_COMM_WORLD );
if (rank < (size-1)) MPI_Recv( &f[mx-1][1], MY-2,
                           MPI_DOUBLE, rank+1,
                           MPI_ANY_TAG,
                           MPI_COMM_WORLD, &status );

```

В результате все передачи из верхнего края локальной порции в нижнюю буферную зону предыдущего процесса выполняются **строго по очереди**, сначала — из первого процесса в нулевой, только затем — из второго в первый, и т. д. Оставшиеся два обмена лучше записать в обратном порядке: сначала — прием от предыдущего в верхнюю буферную зону, затем — передача следующему нижнего края собственной локальной порции. Это позволит нулевому и первому процессам, передавая данные «вверх», тут же приступить к передаче «вниз»:

```

if (rank > 0)  MPI_Send( &f[1][1], MY-2,
                        MPI_DOUBLE, rank-1, 100,
                        MPI_COMM_WORLD );
if (rank < (size-1)) MPI_Recv( &f[mx-1][1], MY-2,
                        MPI_DOUBLE, rank+1,
                        MPI_ANY_TAG,
                        MPI_COMM_WORLD, &status );

if (rank > 0)  MPI_Recv( &f[0][1],
                        MY-2,
                        MPI_DOUBLE, rank-1,
                        MPI_ANY_TAG,
                        MPI_COMM_WORLD,
                        &status );

```

```

if (rank < (size-1)) MPI_Send( &f[mx-2][1],
                                MY-2,
                                MPI_DOUBLE, rank+1,
                                100,
                                MPI_COMM_WORLD );

```

Если мы поделим время совершения всех необходимых обменов на фазы, равные длительности передачи одного сообщения, то распределение обменов по фазам для первых шести фаз будет выглядеть так:

Номер фазы	Обмены на этой фазе
1.	1 -> 0
2.	2 -> 1
3.	3 -> 2, 0 -> 1
4.	4 -> 3, 1 -> 2
5.	5 -> 4, 2 -> 3
6.	6 -> 5, 3 -> 4

.....

С точки зрения параллельного использования имеющегося оборудования для ускорения расчета, результат просто безобразный. На каждой фазе выполняется не более двух обменов, т. е. практически все сетевое оборудование практически все время простаивает, поскольку обмены, которые по смыслу алгоритма вполне могли бы выполняться одновременно, выполняются по очереди, друг за другом. Суммарное время обменов на каждой итерации алгоритма пропорционально числу используемых процессов. По мере роста этого числа, от ускорения расчета за счет его параллельной реализации очень быстро ничего не останется. Мы, конечно, могли бы утешаться тем, что написанная таким образом программа вообще способна доработать до конца. Например, легко убедиться, что, всего лишь поменяв местами второй и четвертый обмены, мы получили бы программу, которая вообще не будет работать, поскольку в ней содержится *дедлок*, или *системный тупик*. Однако хотелось бы понять, как же все-таки следовало написать эту программу правильно.

Для данного, конкретного алгоритма существует простое и вполне удовлетворительное решение. Надо сделать так, чтобы процессы с четными номерами выполняли обмены в одном порядке, а процессы с нечетными номерами — в другом. Напри-

мер, каждый процесс с четным номером сначала выполняет прием данных от следующего процесса, затем — передачу данных ему же, потом — прием данных от предыдущего и, наконец, передачу данных предыдущему. Процесс с нечетным номером, в свою очередь, выполняет обмены в таком порядке: передача данных предыдущему — прием от предыдущего — передача данных следующему — прием от следующего. В результате все множество необходимых обменов разбилось на четыре группы, выполняющиеся друг за другом. Внутри каждой группы обмены между соседними процессами не зависят друг от друга, и могут выполняться одновременно, а не по очереди. Число строго последовательных фаз обмена данными равно четырем, независимо от того, сколько процессов участвует в обмене. Учитывая, что канал связи узла кластера с сетью, как правило, один, и он почти всегда дуплексный, т. е. способен выполнять прием и передачу одновременно, мы, в данном случае, используем коммуникационное оборудование в среднем на 50 %. Это гораздо лучше, чем в предыдущем варианте, хотя все еще довольно плохо.

**Системные причины возникшей трудности.** С одной стороны, имеется набор обменов данными, которые, по смыслу алгоритма, не зависят друг от друга, с другой — некий объем коммуникационного оборудования, способного эти обмены совершить с некоторой степенью параллелизма. Конкретная степень параллелизма, посильная для оборудования, нам, вообще говоря, не известна. Впрочем, мы знаем, что она может быть довольно высокой. Например, будь в нашем распоряжении транспьютерная система с топологией «двумерная решетка» — типичная MPP — система конца 80-х — все обмены, необходимые на каждой итерации нашего алгоритма, могли бы быть выполнены физически одновременно. На кластере рабочих станций с топологией сети «звезда» и достаточно хорошим центральным коммутатором все наши обмены могли бы быть выполнены в два приема, т. е. до половины требуемых обменов аппаратура могла бы выполнять одновременно.

Следовательно, в общем случае мы должны сообщить ОС и аппаратуре обо всех необходимых нам обменах, чтобы ОС, в свою очередь, зная конкретную аппаратную конфигурацию, могла сама упорядочить обмены наилучшим образом, заставив все коммуникационное оборудование работать настолько параллельно, насколько это возможно.



Сложность в том, что имеющихся в нашем распоряжении средств записи обменов для такой дисциплины недостаточно. Пользуясь традиционными языками программирования и моделью двусторонних обменов на базе рандеву, мы вынуждены записывать наши, не зависящие друг от друга по смыслу алгоритма, обмены в определенном порядке, а всякий такой порядок порождает «лишние», не нужные нам зависимости между обменами, по принципу: «не приступай к следующему, не завершив предыдущего».

Отсюда с очевидностью следует, что нам следует несколько расширить наши представления о логике двусторонних обменов и, возможно, добавить еще функций MPI в наше минимальное подмножество.

Способов решения описанных трудностей на логическом уровне в технологиях двустороннего обмена сообщениями известно, как минимум, три (все они представлены в MPI).

*Первый способ* (заимствован из более старых систем, таких, как PVM) — способ принудительной буферизации на передающем конце. Потребуем, чтобы коммуникационная библиотека при запросе на отправку сообщения первым делом копировала это сообщение в некий внутренний буфер, в котором сообщение могло бы дожидаться момента действительной отправки. В этом случае у программы пользователя создается впечатление, что любая отправка сообщения завершается немедленно. У MPI и ОС, в свою очередь, появляется возможность рассмотреть заявки на отправку сразу нескольких сообщений, и организовать выполнение этого «пакета заявок» наилучшим с точки зрения использования аппаратуры образом. При такой дисциплине избегать лишних зависимостей в записи последовательностей обменов совсем просто. Достаточно выписать сначала все передачи, а затем — все приемы, желательно — в том же порядке, что и соответствующие передачи на другом конце канала. Способ этот замечателен своей простотой для программиста. Можно вообще не задумываться о рандеву и прочих сложностях, на иллюстрацию которых мы потратили столько места и умственных усилий, а просто следовать формальному правилу, не задумываясь о том, почему делать надо именно так. В ранних версиях PVM, а также в библиотеке BLACS [4,23], этот способ был единственным именно по причине своей простоты и минимума необходимых пояснений. Недостаток способа состоит в том, что он не допускает эффективной реализации. Разрешая программисту «отправлять»

любые объемы данных, не задумываясь о том, были ли они отправлены в действительности, коммуникационная система вынуждена динамически выделять буфера неизвестного заранее размера, а также тратить время на копирование сообщений в эти буфера.

В MPI этим режимом можно воспользоваться двумя методами.

Первый («неправильный») метод заключается в том, чтобы исследовать конкретную реализацию MPI и установить, в каких условиях функция `MPI_Send` осуществляет буферизацию на передающем конце. Например, многие реализации MPI буферизуют на передающем конце сообщения, длина которых меньше некоторого порога, и даже имеют нестандартные, «подпольные» возможности задавать значение этого порога, скажем, при генерации системы. Разработка прикладной программы в расчете на такие нестандартные и не всегда документированные особенности реализации рискованна по совершенно очевидным соображениям, и не рекомендуется.

Второй («правильный») метод состоит в использовании вместо функции `MPI_Send` функции `MPI_Bsend` с теми же аргументами. Эта функция, в отличие от `MPI_Send`, всегда пытается выполнить буферизацию передаваемого сообщения и, следовательно, завершиться безусловно, не дожидаясь запуска приема на другом конце канала. Работа этой функции основана на более реалистичных допущениях, нежели работа функций «послать» в PVM и подобных ей старых системах. Вместо того чтобы обещать программисту невозможное — потенциально бесконечный буфер — MPI предлагает программисту самому определить, какого размера буфер исходящих сообщений ему необходим, и даже выделить этот буфер. Выделяется он обращением к функции:

```
MPI_Buffer_attach( buffer, size );
```

где `buffer` — выделяемый буфер (массив произвольного типа), `size` — его размер в байтах. В нашей программе вполне хватило бы буфера размером  $2*(MY-2)*sizeof(double)$ .

Ранее выделенный буфер можно освободить обращением к функции:

```
MPI_Buffer_detach( &addr, &size );
```

где `addr` — возвращаемый указатель на освобождаемый буфер, `size` — возвращаемое целочисленное значение — размер освобож-

даемого буфера. Освобождение буфера автоматически приводит к предварительному «выталкиванию» накопленных в нем и еще не переданных сообщений. Одновременно может быть выделен только один буфер, и используется он циклически. Если место в нем исчерпано, обращение к MPI\_Bsend завершится аварийно. По умолчанию MPI\_Bsend ведет себя так, как если бы был выделен буфер нулевой длины, т. е. не работает.

Помимо MPI\_Bsend, функция «послать» в MPI имеет еще две модификации: MPI\_Ssend и MPI\_Rsend. Эти функции не будем ни обсуждать, ни рекомендовать к использованию. Цель этих функций — экономия времени, необходимого для реализации randеву, которая достигается, если синхронизация выполнена каким-то другим способом, например, через обмены двух процессоров с третьим. Тогда появляется возможность говорить о том, что «к моменту запуска передачи из А в Б прием из Б в А уже запущен», и этим можно воспользоваться, исключив часть «обмена любезностями» между А и Б, необходимого в общем случае. На взгляд автора, это не очень хорошая практика программирования, вряд ли способная дать существенный выигрыш на современном оборудовании. Возможно, использование этих функций оправдывает себя на очень высоколатентном оборудовании, например, в случае двух физически удаленных кластеров, объединенных в единое вычислительное поле через спутниковый канал.

*Второй способ* уйти от проблем с лишними зависимостями между обменами — использование **коллективных операций**. Этот способ наиболее непосредственно воплощает идею о том, чтобы сообщить коммуникационной библиотеке сразу обо всех необходимых обменах, предоставив ей возможность «обслужить пакет заявок» самостоятельно и наилучшим образом. Необходимые с точки зрения каждого процесса обмены просто сводятся в таблицу, а все процессы логически одновременно обращаются к функции «обменяться согласно таблице». Конкретную логику элементарных обменов и их совмещения во времени выстраивают коммуникационная библиотека и ОС. От программы требуется, чтобы таблицы необходимых обменов во всех процессах были согласованы: если 5-й заказывает передачу 3-му, то 3-й, соответственно, обязан заказать прием от 5-го, иначе коллективная операция «зависнет». Такое «обобщенное randеву», на которое выходят не 2, а N процессов, и называется коллективной операцией. Обычно в качестве коллективных операций оформляются

специальные виды коммуникаций, вроде широковещательной рассылки всем процессам от одного или *редукции*, т.е., например, вычисления максимума по набору массивов, разбросанных по разным процессам. Однако упомянутая выше в качестве умозрительной возможности коллективная операция «обменяться согласно таблице» в MPI также есть. В этой функции предполагается, что все сообщения, как передаваемые, так и принимаемые в процессе выполнения коллективной операции, являются частями одного и того массива. При этом они могут иметь различную длину и располагаться относительно начала массива с произвольными смещениями. Обращение к функции имеет вид

```
MPI_Alltoallv( sendbuf, sendcounts, sdispls,
               sendtype, recvbuf, recvcounts,
               rdispls, recvtype,
               MPI_COMM_WORLD );
```

Здесь `sendbuf` — массив, в котором располагаются все отправляемые из данного процесса сообщения, `sendcounts` — целочисленный массив длин отправляемых сообщений (`sendcounts[i]` — длина сообщения, отправляемого *i*-му процессу, в компонентах типа `sendtype`), `sdispls` — целочисленный массив смещений отправляемых сообщений относительно начала массива `sendbuf`, тоже в компонентах, `recvbuf`, `recvcounts`, `rdispls`, `recvtype` — аналогичные величины для принимаемых сообщений.

Здесь мы видим, что наилучшим «кандидатом» на включение в список минимально необходимых возможностей MPI для простых случаев являются вовсе не функции «послать — принять», как обычно полагают, а, скорее, функция `MPI_Alltoallv`. Чтобы убедиться в этом, приведем вариант нашей модельной программы, записанный с использованием этой функции:

```
#include <stdio.h>
#include <mpi.h>
/****/
#define MX 640
#define MY 480
#define NITER 10000
#define STEPITER 100
    static float  f[MX][MY];
    static float  df[MX][MY];
    static int  r_lengthes[MX];
    static int  r_displacements[MX];
```

```

        static int s_lengthes[MX];
        static int s_displacements[MX];

    /***/
    int main( int argc, char **argv )
    {
        int i, j, n, m, mx, size, rank;
        MPI_Status status;

    /***/
        MPI_Init( &argc, &argv );
        MPI_Comm_size( MPI_COMM_WORLD, &size );
        MPI_Comm_rank( MPI_COMM_WORLD, &rank );
        mx = ((MX-2)+size-1)/size;
        if ( rank == (size-1) )
        {
            mx = (MX-2) - (size-1)*mx;
        }
        mx += 2;
        printf( "%d of %d is solving heat
                conduction task on %d by %d grid\n",
                rank, size, mx, MY );
        fflush( stdout );

    /* Initial conditions: */
        for ( i = 0; i < mx; i++ )
        {
            for ( j = 0; j < MY; j++ )
            {
                f[i][j] = df[i][j] = 0.0;
                if      ( ((i == 0) && (rank == 0))
                        || (j == 0) ) f[i][j] = 1.0;
                else if ( ((i == (mx-1)) &&
                           (rank == (size-1)))
                        || (j == (MY-1)) ) f[i][j] = 0.5;
            }
        }

    /* Prepare the lengthes and displacements: */
        for ( i = 0; i < size; i++ )
        {
            if      ( i==(rank-1) )
            {
                r_lengthes[i] = s_lengthes[i] = MY-2;
                r_displacements[i] = 1;
                s_displacements[i] = MY+1;
            }
        }
    }

```

```

else if ( i==(rank+1) )
{
    r_lengthes[i] = s_lengthes[i] = MY-2;
    r_displacements[i] = MY*(mx-1)+1;
    s_displacements[i] = MY*(mx-2)+1;
}
else
    r_lengthes[i] = s_lengthes[i] =
        r_displacements[i] =
        s_displacements[i] = 0;
/* Iteration loop: */
for ( n = 0; n < NITER; n++ )
{
    if ( !(n%STEPITER) )
        printf( "Iteration %d\n", n );
/* Do all the transfers: */
    MPI_Alltoallv( &f[0][0], s_lengthes,
                    s_displacements,
                    MPI_DOUBLE,
                    &f[0][0], r_lengthes,
                    r_displacements,
                    MPI_DOUBLE,
                    MPI_COMM_WORLD );
/* Step of calculation starts here: */
    for ( i = 1; i < (mx-1); i++ )
    {
        for ( j = 1; j < (MY-1); j++ )
        {
            df[i][j] = ( f[i][j+1] + f[i][j-1] +
                          f[i-1][j] + f[i+1][j] )
                        * 0.25 - f[i][j];
        }
    }
    for ( i = 1; i < (mx-1); i++ )
    {
        for ( j = 1; j < (MY-1); j++ )
        {
            f[i][j] += df[i][j];
        }
    }
}
/* Calculation is done, F array is a result: */
if ( rank == 0 )

```

```

    {
        fp = fopen( "progrex.dat", "w" );
        fclose( fp );
    }
else
    {
        MPI_Recv( &n, 1, MPI_INT, rank-1,
                  MPI_ANY_TAG,
                  MPI_COMM_WORLD, &status );
    }
fp = fopen( "progrex.dat", "a" );
for ( i = 1; i < (mx-1); i++ )
    fwrite( f[i]+1, my-2, sizeof(f[0][0]), fp );
fclose( fp );
if ( rank < (size-1) )
    MPI_Send( &n, 1, MPI_INT, rank+1, 100,
              MPI_COMM_WORLD );
    MPI_Finalize();

return 0;
}

```

Несколько замечаний о коллективных операциях.

Коллективные операции в MPI — мощный и, в полном объеме, весьма непростой аппарат, описание которого мы здесь не приводим по причине недостатка места (вообще, более или менее полное описание MPI с комментариями и примерами заняло бы примерно столько же страниц, сколько вся эта книга). Основная цель его разработки заключалась в том, чтобы повысить возможную эффективность реализации специальных видов множественных обменов. Например, при широковещательной рассылке сообщения от одного процесса всем, реализуемого функцией MPI\_Bcast(), можно было бы не посылать это сообщение последовательно всем процессам, а построить логику рассылки по принципу дерева: один процесс посылает двум, каждый из двух, в свою очередь, еще двум и так далее. Такого рода оптимизация групповых рассылок, скрытая в реализации MPI, способна значительно повысить эффективность выполнения прикладных программ по сравнению с «наивными» способами реализации, к которым тяготеет большинство прикладных программистов. Однако как мы видели на приведенном только что примере, польза от применения механизма коллективных операций такими применениями не исчерпывается. Иными словами, коллек-

тивные операции могут и должны использоваться не только как средство сделать программу, быть может, более сложной, но зато более эффективной, но и как средство сделать ее более простой, чем в случае использования функций «послать — принять».

При этом не следует забывать, что упрощение программы за счет использования коллективных операций неразрывно связано с параллелизмом типа «независимые вычисления». Для организации конвейерного параллелизма, напротив, функции «послать — принять» обычно гораздо более естественны.

*Третий способ* решения проблем лишних зависимостей, обычно применяемый опытными программистами, наиболее универсален и приводит к наибольшей эффективности реализации прикладных программ.

Акт выдачи запроса на обмен сообщением (не важно, на посылку или на прием) согласно этому способу разбивается на два: «запустить обмен» и «дождаться конца обмена, запущенного ранее». В MPI имеется две модификации выполнения обменов таким способом.

Например, запустить передачу сообщения без ожидания его конца можно, обратившись к функции:

```
MPI_Isend( dparr, n, MPI_DOUBLE, dest, tag,  
MPI_COMM_WORLD, &rq );
```

По сравнению с кратким описанием MPI\_Send (см. подразд. 5.3.1) здесь появился дополнительный аргумент (последний) — выходной параметр: rq — структура типа MPI\_Request, т.е. *запрос*. Функция MPI\_Isend сохранит в этой структуре информацию о запросе на обмен, который был принят к исполнению. Впоследствии программе придется предъявить этот запрос при обращении к функции «дождаться конца обмена», чтобы понятно было, о каком именно из запущенных ранее обменов идет речь.

В данном примере обращение к функции «дождаться конца» могло бы выглядеть так:

```
MPI_Wait( &rq, &st );
```

где rq — структура типа MPI\_Request, а st — уже знакомая нам структура типа MPI\_Status. Для случая посылки в возвращении статуса особого смысла нет, но дело в том, что MPI\_Wait и аналогичные функции ожидания конца запущенного ранее обмена одинаковы для запуска посылок и запуска приемов. Например,



если бы мы запустили без ожидания конца обмена не посылку, а прием, обратившись к функции:

```
MPI_Irecv( dparr, n, MPI_DOUBLE, src, tag,  
           MPI_COMM_WORLD, &rq );
```

то для ожидания конца нам потребовалось бы обратиться к той же самой функции `MPI_Wait()`, и в структуре `st` после этого обращения была бы накоплена та же самая информация, что и в случае `MPI_Recv()`.

Помимо функции «дождаться конца данного обмена», имеется функция «проверить, завершен ли данный обмен». Как для ожидания, так и для проверки завершения обмена имеются функции, оперирующие не одним запросом, а массивом запросов. Отдельные функции позволяют подождать (или проверить) завершение любого обмена из представленных массивом запросов, или же всех обменов, представленных массивом запросов.

Следует иметь в виду, что обращения к функциям ожидания и/или проверки завершения обмена, запущенного таким способом, должны быть предусмотрены в программе обязательно. Нельзя, например, запустить посылку сообщения некоторому процессу, а потом ждать от него ответного сообщения, не проверяя, завершена ли запущенная посылка. Она вполне может быть не завершена никогда. Это неприятное свойство запущенных без ожидания конца обменов является не недочетом «плохих» реализаций, а свойством системы, явно предусмотренным в стандарте.

Вторая модификация запуска обменов без ожидания конца, предусмотренная в MPI, связана с понятием *постоянных запросов* на обмен. При использовании функций `MPI_Isend()`/`MPI_Irecv()` запросы на обмен порождаются автоматически при запуске обмена и автоматически же исчезают, как только та или иная функция ожидания или проверки сообщает вызывающей программе о завершении обмена. Альтернативная дисциплина состоит в том, чтобы сформировать постоянный запрос на обмен, после чего пользоваться им многократно. Например, обращение к функции «сформировать постоянный запрос на посылку сообщения» выглядит так:

```
MPI_Send_init( dparr, n, MPI_DOUBLE, dest, tag,  
              MPI_COMM_WORLD, &rq );
```

Аргументы у этой функции те же, что и у `MPI_Isend()`, но обращение к этой функции не запускает обмена. Оно лишь фор-

мирует в структуре `rq` всю необходимую для скорейшего запуска такого обмена информацию. Чтобы запустить такой заранее подготовленный обмен, следует обратиться к функции:

```
MPI_Start( &rq );
```

или к ее аналогу `MPI_Startall()`, запускающему сразу целый массив заранее подготовленных обменов. При завершении обмена запрос `rq` никуда не девается и может быть использован повторно. Когда запрос больше не нужен, его можно освободить обращением к

```
MPI_Request_free( &rq );
```

`MPI_Start()` и `MPI_Startall()` запускают обмены без ожидания конца. Ожидание или опрос их завершения выполняется так же, как в случае `MPI_Isend()/MPI_Irecv()`.

Использование аппарата постоянных запросов позволяет программисту легко и непосредственно создавать собственные «коллективные операции». В самом деле, для оформления любой группы обменов как единого действия достаточно:

- объявить в программе массив запросов требуемой длины, и массив статусов такой же длины;

- в инициализирующей части программы сформировать в каждом из элементов массива запросов соответствующий запрос;

- в рабочем цикле программы запускать всю группу обменов обращением к `MPI_Startall()`, а ожидать завершения обмена — обращением к `MPI_Waitall()`. Пример написания в таком стиле нашей модельной программы приводится ниже.

```
#include <stdio.h>
#include <mpi.h>
/****/
#define MX 640
#define MY 480
#define NITER 10000
#define STEPITER 100
    static float  f[MX][MY];
    static float  df[MX][MY];
/****/
int main( int argc, char **argv )
{
    int i, j, n, m, mx, size, rank, nrq;
```

```

FILE *fp;
MPI_Request rq[4];
MPI_Status st[4];

/**/
MPI_Init( &argc, &argv );
MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
mx = ((MX-2)+size-1)/size;
if ( rank == (size-1) )
{
    mx = (MX-2) - (size-1)*mx;
}
mx += 2;
/* Prepare the transfer requests: */
nrq = 0;
#define LENG (MY-2)
if ( rank > 0 )
{
    MPI_Recv_init( &f[0][1], LENG,
                  MPI_FLOAT, rank-1, 100,
                  MPI_COMM_WORLD, rq+nrq );

    nrq++;
    MPI_Send_init( &f[1][1], LENG,
                  MPI_FLOAT, rank-1, 100,
                  MPI_COMM_WORLD, rq+nrq );

    nrq++;
}
if ( rank < (size-1) )
{
    MPI_Recv_init( &f[mx-1][1], LENG,
                  MPI_FLOAT, rank+1, 100,
                  MPI_COMM_WORLD, rq+nrq );

    nrq++;
    MPI_Send_init( &f[mx-2][1], LENG,
                  MPI_FLOAT, rank+1, 100,
                  MPI_COMM_WORLD, rq+nrq );

    nrq++;
}
/* Requests prepared */
printf( "%d of %d is solving heat
        conduction task on %d by %d grid\n",
        rank, size, mx, MY );
fflush( stdout );

```

```

/* Initial conditions: */
    for ( i = 0; i < mx; i++ )
    {
        for ( j = 0; j < MY; j++ )
        {
            f[i][j] = df[i][j] = 0.0;
            if      ( ((i == 0) && (rank == 0))
                    || (j == 0) ) f[i][j] = 1.0;
            else if ( ((i == (mx-1)) && (rank ==
                                   (size-1)))
                    || (j == (MY-1)) ) f[i][j] = 0.5;
        }
    }

/* Iteration loop: */
    for ( n = 0; n < NITER; n++ )
    {
        if ( !(n%STEPITER) )
            printf( "Iteration %d\n", n );
/* Start all the necessary requests: */
        MPI_Startall( nrq, rq );
/* Wait for them to be over: */
        MPI_Waitall( nrq, rq, st );
/* Step of calculation starts here: */
        for ( i = 1; i < (mx-1); i++ )
        {
            for ( j = 1; j < (MY-1); j++ )
            {
                df[i][j] = ( f[i][j+1] +
                             f[i][j-1] + f[i-1][j] +
                             f[i+1][j] )
                            * 0.25 - f[i][j];
            }
        }
        for ( i = 1; i < (mx-1); i++ )
        {
            for ( j = 1; j < (MY-1); j++ )
            {
                f[i][j] += df[i][j];
            }
        }
    }

/* Calculation is done, F array is a result: */
    if ( rank == 0 )

```

```

{
    fp = fopen( "progrev.dat", "w" );
    fclose( fp );
}
else
{
    MPI_Recv( &n, 1, MPI_INT, rank-1,
              MPI_ANY_TAG,
              MPI_COMM_WORLD, &status );
}
fp = fopen( "progrev.dat", "a" );
for ( i = 1; i < (mx-1); i++ )
    fwrite( f[i]+1, my-2, sizeof(f[0][0]), fp );
fclose( fp );
if ( rank < (size-1) )
    MPI_Send( &n, 1, MPI_INT, rank+1, 100,
              MPI_COMM_WORLD );
    MPI_Finalize();
return 0;
}

```

**Что еще может потребоваться.** Итак, мы получили довольно подробное представление о том, во что и как превращаются абстрактные понятия модели программирования при переходе к конкретной технологии. Рассматривать MPI полностью даже на том уровне подробности, который использовался ранее, мы не будем, а ограничимся лишь перечислением наборов принципиальных возможностей.

1. *Барьерная синхронизация.* В дисциплине двусторонних обменов синхронизация партнеров по обмену происходит автоматически. Это, казалось бы, избавляет от необходимости иметь отдельные от собственно обменов предписания синхронизации. И все же аппарат отдельной от обменов сообщениями синхронизации в MPI предусмотрен. Это — аппарат *барьерной синхронизации*. Обращение к функции:

```
MPI_Barrier( MPI_COMM_WORLD );
```

приводит к тому, что все процессы выполняют синхронизацию в режиме коллективной операции. Обращение к этой функции в **каждом** процессе завершится тогда и только тогда, когда **все** процессы указанного коммуникатора выполняют обращение к соответствующей функции. Барьерная синхронизация удобна,

например, для синхронизации при обращении к общей файловой системе, когда все процессы должны по очереди открыть некоторый файл и записать в него свою порцию результатов счета. Барьерная синхронизация широко применяется в технологиях параллельного программирования на базе моделей, отличных от модели двусторонних обменов (см. далее). В качестве примера использования приведем здесь завершающую часть нашей модельной программы, в которой результаты расчета записываются в файл, с применением барьерной синхронизации вместо передачи по цепочке значения  $n$ :

```
if ( rank == 0 )
{
    fp = fopen( "progre.dat", "w" );
    fclose( fp );
}
for ( j = 0; j < size; j++ )
{
    MPI_Barrier( MPI_COMM_WORLD );
    if ( j == rank )
    {
        fp = fopen( "progre.dat", "a" );
        for ( i = 1; i < (mx-1); i++ )
            fwrite( f[i]+1, my-2,
                    sizeof(f[0][0]), fp );
        fclose( fp );
    }
}
```

**2. Создание групп и коммутаторов.** В самом начале нашего обзора MPI мы упоминали понятие коммутатора, т.е. набора процессов, внутри которого:

- имеется отдельная нумерация процессов;
- обмены между процессами происходят независимо от обменов в других наборах.

Нигде в приводимых примерах мы не использовали коммутаторов, отличных от создаваемого системой по умолчанию коммутатора `MPI_COMM_WORLD`. Это в целом соответствует практике реального программирования — большинство программистов так и поступает. Тем не менее, создание коммутаторов, отличных от `MPI_COMM_WORLD`, в некоторых случаях бывает очень полезно. Например, при написании библиотек функций, вызываемых из ветвей параллельной программы, сообщения, циркулирующие внутри библиотечных функций,

могут «перепутаться» с сообщениями в вызывающей программе, если, скажем, при приеме используется адрес отправителя вида `MPI_ANY_SOURCE`. Простейший способ избежать неприятностей такого рода — создание для сообщений, передаваемых внутри библиотечных функций, отдельного коммуникатора.

Полное описание соответствующего набора возможностей в стандарте MPI четким и лаконичным не назовешь. Оно занимает более 40 страниц и изобилует довольно многословными, но не всегда понятными пояснениями. Попробуем выделить главное.

Любой обмен данными в MPI происходит в рамках некоторого коммуникатора, причем таким образом, как будто бы других коммуникаторов (и заданных в них обменов) не существовало.

По умолчанию в распоряжении программы имеется два коммуникатора: `MPI_COMM_WORLD`, включающий все процессы, и `MPI_COMM_SELF`, включающий только данный процесс.

Коммуникатор может быть интра-коммуникатором, или интер-коммуникатором.

**Интра-коммуникатор** — это коммуникатор общего вида, такой, как использованный нами в примерах, существующий по умолчанию коммуникатор `MPI_COMM_WORLD`. Основной способ порождения новых коммуникаторов такого вида — это расщепление имеющегося коммуникатора на несколько новых. Выполняется это действие коллективной (в расщепляемом коммуникаторе) операцией `MPI_Comm_split()`. Если программисту необходимо только это, то ничего другого из упомянутых выше сорока с лишним страниц описания групп и коммуникаторов изучать не требуется. Процессы в интра-коммуникаторе занумерованы от нуля и подряд, и могут взаимодействовать в любых режимах без ограничений.

**Интер-коммуникатор** — это специальный вид коммуникатора, в котором входящие в него процессы разбиваются на две не пересекающиеся *группы*. В таких коммуникаторах не бывает коллективных операций, а операции попарного взаимодействия, вроде «послать — принять», могут выполняться только между процессами из разных групп. Утверждается, что это удобно для программ с логикой «клиент — сервер», когда сервер представляет собой группу процессов: клиенты не взаимодействуют друг с другом, но взаимодействуют с процессами сервера. Сами процессы сервера, очевидно, взаимодействуют между собой через другой коммуникатор. Вопрос о том, действительно ли это удобно, обсуждать здесь не будем.

В отличие от интра-коммуникаторов, в интер-коммуникаторах в каждой группе имеется своя, независимая нумерация процессов с нуля и подряд. Номера процессов — получателей и отправителей в функциях «послать — принять», трактуются как номера процессов в «чужой» группе данного коммуникатора.

Такая логика требует введения, наряду с понятием коммуникатора, понятия группы как специального программного объекта. Строится довольно сложная и разветвленная система манипуляций с группами как таковыми, а также с группами и коммуникаторами.

Таким образом, понятие группы, по существу, необходимо только в связи с интер-коммуникаторами. Интра-коммуникаторами вполне можно пользоваться, вообще не зная слова «группа». Ситуация несколько осложняется тем, что формально группа (единственная) приписывается также любому интра-коммуникатору, а в тексте стандарта в объяснении функций работы с интра-коммуникаторами термин «группа» интенсивно используется, зачастую — без острой необходимости, что заметно запутывает изложение.

**3. Производные типы данных.** Нам известно, что аппарат типов передаваемых данных необходим в MPI для поддержки гетерогенных вычислительных кластеров. Для чего нужно уметь создавать свои типы данных?

В применявшуюся нами до сих пор систему типизации не укладываются две категории передаваемых данных, сравнительно часто встречающиеся на практике:

- структуры из разнотипных полей;
- данные, расположенные в памяти регулярным, но не сплошным образом, например, столбец (в C) или строка (в Фортране) двумерного массива.

Для исключения лишних копирований и обеспечения возможно более высокой эффективности передачи такого рода данных, в том числе — в гетерогенных кластерах, служит аппарат *производных типов*, т. е. типов данных, создаваемых пользователем.

Элемент данных производного типа представляет собой обобщенную структуру и характеризуется двумя одномерными массивами:

- массивом типов полей структуры;
- байтовыми смещениями этих полей внутри структуры.

Легко видеть, что столбец двумерного массива в C (строка в Фортране) легко укладывается в эту схему. В MPI имеется набор



функций построения типов, которые позволяют программе пользователя создавать производные типы на базе уже существующих, а затем использовать их на общих основаниях в функциях обмена данными.

4. *Топологии.* Это наиболее экзотическое и наименее применяемое подмножество функций MPI. Формально топология есть свойство интра-коммуникатора. По смыслу оно отражает тот факт, что коммуникационная сеть, на которой работает MPI, может быть существенно неоднородной. Например, в двумерной решетке или торе узлов, каждый из которых оснащен четырьмя линиями связи, некоторые пары узлов связаны непосредственно, а некоторые — через большое количество «пересадок». Формально связь каждого с каждым имеется, но скорости и латентности могут различаться очень сильно. Для того чтобы реализация MPI могла работать в этих условиях наиболее эффективно, и существует аппарат топологий. При современном оборудовании, основанном почти всегда на центральном коммутаторе, особой нужды пользоваться этими возможностями нет. Кроме того, топологии вводят дополнительную нумерацию процессов, отличную от линейной. Например, можно организовать процессы в многомерный массив. Эта возможность может рассматриваться как дополнительное удобство для программиста, безотносительно к эффективности аппаратной реализации.

#### 5. *Новые возможности стандарта MPI-2.*

Через несколько лет после публикации стандарта MPI он был уточнен и значительно расширен. Старый набор возможностей стал называться MPI-1, а новый — MPI-2. В действительности, во многих реализациях MPI, объявленных как реализации MPI-1, присутствуют многие из расширений MPI-2.

Перечислим основные направления расширения стандарта.

*Динамическое создание процессов.* В MPI-1 принята очень простая как в понимании, так и в реализации схема порождения процессов. Считается, что все процессы, образующие в совокупности коммуникатор MPI\_COMM\_WORLD, рождаются в момент запуска программы. Процесс, конечно, может завершиться, но родиться в ходе выполнения программы он не может. Информация о том, сколько процессов всего имеется, задается не в программе, а в команде ее запуска на счет, т. е. приходит в программу извне. Эта схема удобна своей простотой, но не всегда достаточна (в PVM, например, применяется совсем другая схема, что сильно усложняет именование процессов, но дает дополни-

тельную гибкость). В MPI-2 было решено разрешить создавать новые процессы динамически. Рассматривать этот аппарат подробно здесь не будем.

*Односторонние коммуникации.* В MPI-2 были добавлены возможности организации односторонних обменов данными между процессами. Мы кратко обсудим эти возможности ниже, в разделе, посвященном модели односторонних обменов.

*Параллельный ввод-вывод.* Файловый ввод-вывод в параллельных программах для MPP-систем осложнен, вообще говоря, двумя факторами.

Во-первых, некоторые (или все) процессоры (и, следовательно, процессы) могут не иметь доступа к общей файловой системе.

Во-вторых, даже если такой доступ имеется, одновременное открытие файла на запись из многих процессов, как правило, ни к чему хорошему не приводит, даже если разные процессы пишут в разные места одного и того же файла. Два примера преодоления этой трудности мы видели в различных вариантах текста нашей модельной программы.

Очевидно, обе проблемы можно решить, если включить в состав MPI специальную, отличную от стандартной, библиотеку функций работы с файлами. Такая библиотека могла бы обеспечить общение всех процессоров с единой файловой системой, даже если в стандартных терминах ОС эта файловая система не доступна, причем в дисциплине, допускающей параллельное открытие файлов на запись. Эта дисциплина, конечно, должна быть специально спроектирована.

В состав MPI-2 включена именно такая библиотека параллельного ввода-вывода. Ниже приводится пример записи с ее помощью вывода результатов расчета в нашей модельной программе.

```
MPI_File_open( MPI_COMM_WORLD, "progre.dat",
               MPI_MODE_WRONLY | MPI_MODE_
CREATE,
               MPI_INFO_NULL, &fp );
MPI_File_set_view( fp, (MPI_Offset)
                  (my_number*(mx-2) *
                  (my-2)*sizeof(f[0][0])),
                  MPI_FLOAT, MPI_FLOAT, "native",
                  MPI_INFO_NULL );
for ( i = 1; i < (mx-1); i++ )
```

```

{
    MPI_File_write( fp, f[i]+1, my-2,
                   MPI_FLOAT, st );
}
MPI_File_close( &fp );

```

Используемая здесь переменная `fp` имеет тип `MPI_File` и должна быть объявлена выше в тексте программы. Запись в файл, выполняемая таким образом, уже не нуждается в дополнительной синхронизации с помощью обменов по цепочке или барьеров. Всю необходимую синхронизацию библиотека параллельного ввода-вывода берет на себя.

В заключение — несколько слов о доступных для использования реализациях MPI.

Самой старой и наиболее распространенной реализацией MPI, свободно распространяемой в виде исходных текстов, является MPICH [34]. Вторая по распространенности реализация — LAM [35], которая сейчас переведена в «режим поддержки старых версий», поскольку большая часть разработчиков перешла в новый проект — Open MPI [36]. Все перечисленные реализации поддерживают специализированные высокоскоростные сети, а также, частично или полностью, расширения MPI-2. Опыта использования Open MPI у автора нет, но с MPICH и LAM приходилось работать много. Это весьма надежные и качественные реализации.

Обзор возможностей технологии MPI завершен. Как мы уже неоднократно отмечали, технология эта традиционно является низкоуровневой базой при работе с оборудованием двусторонних обменов сообщениями. Дальнейшее рассмотрение моделей и технологий параллельного программирования возможно по трем направлениям.

Можно рассматривать аналогичные низкоуровневые технологии той же модели. За многие годы господства в суперкомпьютерном мире того класса оборудования, на который они ориентированы, таких технологий накопился не один десяток. Мы этого делать не будем. Все эти технологии принципиально похожи друг на друга, как системы команд разных процессоров, и все основные идеи, накопленные их разработчиками, технология MPI интегрировала в себе. Двигаясь этим путем, мы не узнаем ничего нового и интересного.

Можно рассматривать более высокоуровневые надстройки над MPI, призванные повысить уровень автоматизации параллель-

ного программирования при работе с коммуникационным оборудованием того же класса.

С другой стороны, можно рассмотреть модели и технологии, ориентированные на другие классы коммуникационного оборудования.

Мы продолжим изложение именно в таком порядке, после чего проведем сравнительный анализ всех приведенных технологий.

### **5.3.2. Модели, производные от явного двустороннего обмена сообщениями**

Как отмечалось ранее, в течение весьма длительного времени преимущественно применяемое в вычислительных кластерах оборудование не способствовало разнообразию базовых программистских моделей. Выбор для программиста был прост: либо MPI (PVM), т.е. ручное выписывание двусторонних обменов, либо программная надстройка, повышающая уровень абстракции при записи программы, но не меняющая качественного характера рассуждений при ее (программы) построении. При проектировании такого рода моделей, зачастую, весьма элегантно логически, главным было постоянно помнить, что в реализации это все будет опираться на MPI, и модель должна допускать высокую эффективность такой реализации. В целом это семейство моделей можно рассматривать как *модели частичной автоматизации разработки параллельных программ, выполняющихся в среде двусторонних обменов*.

**Модель библиотек коллективных операций над распределенными массивами.** Наиболее известная и типичная технология этой модели — библиотека ScaLAPACK [23]. Это версия изначально однопроцессорной библиотеки линейной алгебры LAPACK, ориентированная на многопроцессорные вычислители без общей памяти.

На практике прикладные программы часто выполняют вычислительно емкие фрагменты обработки данных путем обращения к стандартным библиотечным функциям. Например, в программе, работающей с векторами и матрицами, скорее всего, будут присутствовать обращения к библиотечным функциям: «перемножить две матрицы», «найти собственные числа», «обратить матрицу» и т. п. Если программа в основном использует

массивы, в которых хранятся матрицы и векторы, в качестве аргументов, передаваемых в функции, почему бы не «спрятать» внутри этих функций весь параллелизм, т. е. обращения к MPI? Массивы, вычисляемые в одних функциях и тут же передаваемые в качестве аргументов в другие, теперь будут представлять собой не матрицы и векторы, а их локальные порции, хранящиеся в данном процессе. Когда все процессы «хором» (в режиме коллективной операции) обращаются к функции «перемножить матрицы», соответствующая функция в каждом процессе, зная свой номер и общее число участников, «соображает», какая часть работы ей досталась, какие обмены данными с себе подобными следует выполнить. С точки зрения прикладного программиста текст ветви параллельной программы становится очень похожим на текст обычной, последовательной программы (добавляются лишь некоторые «настроечные» обращения к библиотечным функциям в начале текста). Вся логика параллельного взаимодействия, включая всевозможные проверки условий, связанных с передачей тех или иных сообщений, скрыта от прикладного программиста. Массивы, в которых хранятся локальные порции векторов и матриц, можно рассматривать как абстрактные объекты (в духе объектно-ориентированного программирования), библиотечные функции — как методы. Называется этот абстрактный объект, конечно же, *распределенным массивом*, а метод, реализуемый логически одновременным обращением к одной и той же библиотечной функции на всех процессах, — конечно же, коллективной операцией. В обзоре MPI мы уже сталкивались с коллективными операциями, правда, выполняющими более простые действия.

На практике потребовалось некоторое усложнение этой схемы. Дело в том, что не все распределенные массивы распределены по процессам одинаково. Например, программисту может потребоваться один массив, распределенный блоками по всем процессам, кроме нулевого, и еще один, распределенный по всем процессам, кроме последнего. При выполнении коллективной операции в первом случае нулевой процесс должен «сообразить», что ему ничего делать не надо, во втором случае то же самое должен «понять» последний процесс. Значит, в распределенном массиве должна храниться информация о конкретном виде его распределения по процессам. Будь библиотека написана на объектно-ориентированном языке, например C++, и предназначена для использования в программах, написанных на нем же, об этой

совершенно естественной детали не стоило бы даже говорить. Довольно естественно, что внутри объекта много всего полезного хранится, на то он и объект, чтобы скрывать в себе детали реализации. Но библиотека ScaLAPACK написана на Фортране и ориентирована, преимущественно, на использование в программах, написанных на нем же, причем проектировалась еще в те времена, когда ни структур, ни указателей в Фортране не было. Поэтому о передаче дополнительных сведений о характере распределения аргумента — массива в коллективную операцию должна заботиться прикладная программа. Сделано это очень простым и понятным для «Фортрановского» программиста способом. По сравнению с функциями LAPACK (однопроцессорными) в аналогичные им функции ScaLAPACK (параллельные) к каждому аргументу — массиву добавляется еще один аргумент — **дескриптор распределенного массива**. По смыслу это — структура, в которой хранится информация о конкретном формате распределения предыдущего аргумента. В Фортране, в котором структур нет, эта структура стала одномерным целочисленным массивом (длиной 9). В начале работы программы дескрипторы всех распределенных массивов инициализируются обращением к специальной функции «создать распределенный массив». Затем пользователь должен самостоятельно следить, чтобы при использовании в качестве аргумента коллективной операции локальной порции распределенного массива вместе с ней передавался соответствующий дескриптор. Если несколько массивов распределены одинаково, дескриптор у них будет один и тот же.

В связи с эскизно изложенной здесь технологией возникают две проблемы.

Первая очевидна. Пусть в библиотеке не нашлось стандартных функций для некоторых действий, которые предстоит выполнить прикладной программе. Означает ли это, что программист вынужден немедленно приступить к форсированному изучению MPI, и, заодно, внутренних форматов представления распределенных массивов, чтобы восполнить этот досадный пробел? Есть ли в мире распределенных массивов и коллективных операций какое-нибудь более или менее универсальное средство передачи данных между процессами, не связанное с конкретными действиями вроде «найти собственные числа матрицы»?

Конечно, есть. Средство это — всего лишь одна коллективная операция, которая обычно присутствует в библиотеках такого

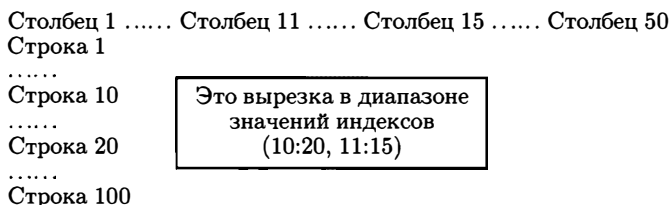


Рис. 5.2. Пример вырезки размером  $11 \times 5$  из двумерного массива размером  $100 \times 50$

рода. Называется она — «копирование произвольной вырезки из одного распределенного или локального массива в другой распределенный или локальный массив». Это и есть средство произвольного перемещения данных между процессами.

**Замечание.** Понятие *вырезки из массива* хорошо известно программистам, пишущим на Фортране. Под вырезкой понимается часть массива, задаваемая сплошным диапазоном значений индекса по каждому из измерений. Например, пусть имеется двумерный массив из строк, занумерованных от 1 до 100, и столбцов, занумерованных от 1 до 50. Тогда часть этого массива, находящаяся на пересечении, например, строк, с 10-й по 20-ю, и столбцов, с 11-го по 15-й, — это вырезка из исходного массива. Она представляет собой сплошной двумерный «подмассив» из 11 строк и 5 столбцов (рис. 5.2).

Вторая проблема коренится в самой природе понятия коллективной операции. Она связана с соображениями о двух способах параллельной обработки данных (см. подразд. 5.1). Легко видеть, что «логически одновременный вызов коллективной операции во всех процессах» как нельзя лучше отвечает первому способу — собственно параллельной обработке, или, применительно к нашим модельным задачам, методу Якоби. Конвейерный параллелизм (в наших терминах — метод верхней релаксации) в коллективных операциях записывается очень плохо, если записывается вообще — ведь там весь смысл организации передачи данных состоит в том, что все происходит «не в фазе», т. е. не одновременно по всем процессам. Это не страшно, если вся работа конвейера, от запуска до остановки, спрятана внутри одной коллективной операции. Если же мы хотим самостоятельно выписать конвейерную логику в прикладной программе, опираясь на технологию более мелких коллективных операций, получится, в лучшем случае, довольно тяжело.

Это — известная проблема всех технологий, основанных на понятии коллективной операции, и хорошего решения она не имеет.

Библиотеки коллективных операций обычно сочетают в себе в некоторой пропорции проблемно-ориентированные (например, перемножить матрицы) и универсальные (скопировать вырезку из распределенного массива) средства. Бывают чисто проблемно-ориентированные библиотеки — как правило, «решатели» разреженных матриц специального вида. К решению систем линейных алгебраических уравнений с такими матрицами сводятся многие численные методы задач математической физики. Бывают библиотеки коллективных операций с высокоразвитой логикой низкоуровневых, универсальных возможностей — всевозможных манипуляций с векторами, матрицами, строками, столбцами и т. п. Наиболее известной на сегодня библиотекой такого рода является библиотека PETSc [24]. В противоположность ScaLAPACK, она написана на C, для представления обрабатываемых данных используется развитая система специальных типов — тип «вектор», тип «матрица» и т. п. Часто складывается ощущение, что со сложностью специально разработанной системы типов данных разработчики несколько перестарались. Для первоначального освоения библиотеки надо обладать заметной общепрограммистской культурой, «Фортрановскому программисту» освоить ее тяжело.

Как ScaLAPACK, так и PETSc распространяются свободно в виде исходных текстов. Обе они реализованы на базе MPI. Для ScaLAPACK существуют старые реализации на базе других библиотек двусторонних обменов.

**Модель параллельных трансляторов в стиле HPF.** Основная идея этой программистской модели предельно проста. Ранее мы обосновали основной постулат параллельного программирования для систем с распределенной памятью тем, что гипотетический «распараллеливающий транслятор» не может извлечь из текста программы для одного процессора информацию о том, что и как «распараллеливать», и потому не в состоянии автоматически превратить ее в параллельную программу. Так может быть, если ему немного помочь, предоставив необходимую дополнительную информацию, он справится, построит параллельную программу по тексту для одного процессора?

Наиболее известной технологией этой модели является **HPF** (**H**igh **P**erformance **F**ortran, Фортран высокой производительности) [4,26,62].



Программа на HPF пишется как однопроцессорная, и представляет собой правильную программу на обычном, «однопроцессорном» Фортране. Она снабжается дополнительными *директивами параллельного выполнения*, которые с точки зрения транслятора HPF являются операторами языка, а с точки зрения обычного Фортрана — комментариями. Так, в Фортране комментарием является любая строка, содержащая в первой позиции символ «C». В HPF директивой параллельного выполнения является любая строка, содержащая в первых пяти позициях текст «CHPF\$».

Помимо хотя и далеко не полной, но все же весьма значительной степени автоматизации распараллеливания, такое построение языка обладает дополнительными технологическими преимуществами. Очень удобно, что файловый ввод-вывод распределенных массивов автоматизируется полностью. Также немаловажно, что исчезает необходимость сопровождать отдельно два варианта программы — последовательный и параллельный, поскольку теперь оба варианта содержатся в едином тексте.

Первоначально предполагалось, что директивы параллельного выполнения будут просты и немногочисленны. В самом деле, если попросить программиста специфицировать необходимое ему распределение обрабатываемых массивов по процессорам, то транслятор самостоятельно «сообразит», как построить параллельную обработку этих массивов (ведь ясно, что данные надо стараться обрабатывать там, где они расположены). Так появились две первые директивы — CHPF\$ DISTRIBUTE и CHPF\$ ALIGN, и был построен пример решения задачи о прогреве кирпича методом Якоби:

```

      PROGRAM LAPLACE
C Solving the laplace pde
      PARAMETER ( MX = 300, MY = 400 )
      REAL F( 0:MX+1, 0:MY+1 ), DF( 1:MX, 1:MY )
C Массив DF распределяется блоками,
C а массив F — так, чтобы элементы
C F и DF с одинаковыми значениями
C индексов попадали на один и тот
C же процессор:
CHPF$      DISTRIBUTE DF (BLOCK, BLOCK)
CHPF$      ALIGN F(I,J) WITH DF(I,J)
C
C Витки следующего цикла независимы,
```

С выполнять на том процессоре, где  
 С расположена левая часть присваивания:

```

CHPF$ INDEPENDENT
      DO 2 J = 0, MY+1
        F( 0, J ) = 1.0
        F( MX+1, J ) = 1.0
      2 CONTINUE
CHPF$ INDEPENDENT
      DO 3 I = 1, MX
        F( I, 0 ) = 1.0
        F( I, MY+1 ) = 1.0
      3 CONTINUE
CHPF$ INDEPENDENT
      DO 4 I = 1, MX
CHPF$ INDEPENDENT
        DO 5 J = 1, MY
          F( I, J ) = 0.0
        5 CONTINUE
      4 CONTINUE
CHPF$ INDEPENDENT
      DO 6 I = 1, MX
CHPF$ INDEPENDENT
        DO 7 J = 1, MY
          DF( I, J ) = 0.0
        7 CONTINUE
      6 CONTINUE
C Iteration
      DO 8 K = 1, 10000
CHPF$ INDEPENDENT
        DO 9 I = 1, MX
CHPF$ INDEPENDENT
          DO 10 J = 1, MY
            DF(I, J) = ( F(I, J+1) + F(I, J-1)
              &          +F(I-1, J) + F(I+1, J) )
              &          *0.25 - F(I, J)
          10 CONTINUE
        9 CONTINUE
CHPF$ INDEPENDENT
        DO 11 I = 1, MX
CHPF$ INDEPENDENT
          DO 12 J = 1, MY
            F(I, J) = F(I, J) + DF(I, J)
          12 CONTINUE

```

```

11    CONTINUE
8     CONTINUE
      OPEN (10, FILE='F')
      WRITE (10,*) F
      CLOSE (10)
END

```

С этой модельной программой любой транслятор HPF справится просто блестяще. С реальными программами, к сожалению, все несколько сложнее.

Уже в этом примере присутствует намек на проблему, с которой пришлось столкнуться впоследствии — директива `CHPF$ INDEPENDENT`. Эта директива говорит транслятору, что витки данного цикла независимы и, следовательно, их можно «разбросать» по процессам, вслед за элементами обрабатываемых в них массивов. В данном, простом, случае транслятор мог бы и сам понять, что витки цикла независимы — соответствующие методы анализа программ существуют. Но в реальной программе, например, в цикле могло бы встретиться обращение к подпрограмме, транслируемой отдельно. Вносит ли она зависимость между витками цикла? Это знает программист, транслятор этого знать не может. Значит, нужна директива `CHPF$ INDEPENDENT`, утверждающая, что витки этого цикла — независимы. Постепенно таких дополнительных директив, необходимость которых не была очевидна на первоначальном этапе, накопилось довольно много. Язык усложнился.

Невзирая на оговорки, касающиеся директивы `CHPF$ INDEPENDENT`, приведенный выше текст программы несколько похож на чудо. В самом деле, мы написали вполне разумную последовательную программу, добавили к ней несколько директив, и получили в результате параллельную программу, причем для такого тяжелого в программировании варианта оборудования, как MPP-система на базе сети двусторонних обменов. Если это возможно, что мы тогда вообще обсуждаем в этой главе? И почему так мало слышно о практическом использовании этой потрясающей технологии?

Начнем с главного. Правильно написанная программа на HPF, действительно, является последовательной программой, к которой добавлено некоторое количество директив. Из этого, однако, вовсе не следует, что для любой или почти любой последовательной программы можно подобрать такие директивы, которые превратят ее в программу на HPF. В последовательной програм-

ме, например, может в принципе не быть параллелизма. В большинстве же программ параллелизм в принципе есть, или автору программы кажется, что он есть, но этот параллелизм надо выразить понятным транслятору HPF образом, т.е. где-то что-то подправить, немного изменить форму записи циклов. И конечно, расставить подходящие директивы.

Работа эта вовсе не простая, и программист попытается выполнить ее, как и любую коренную переработку программы, не в один прием, а по шагам, т.е. выполнить *пошаговое распараллеливание*. В самом деле, вполне естественно начать с распараллеливания того цикла в программе, который потребляет максимальное время, потом распараллелить следующий по затратам времени цикл и так далее. Как узнать, правильно ли распараллелился очередной цикл, не напутал ли программист что-нибудь в директивах? Конечно же, по ускорению программы. Если частично распараллеленная программа демонстрирует ускорение, хотя бы очень грубо соответствующее закону Амдала, значит, программист — на верном пути.

Многие, если не все, начинающие пользователи HPF рассуждали и действовали именно так. К сожалению, пошаговое распараллеливание с помощью HPF практически невозможно. Попробуем понять, почему это так. Для краткости в приведенном ниже рассуждении будем называть вычислительные узлы, каждый из которых оснащен локальной памятью, просто процессорами, имея в виду, в действительности, совокупность процессора и его памяти.

В частично распараллеленной программе присутствует уже распараллеленная часть, т.е. циклы, витки которых уже распределены по процессорам, и еще не распараллеленная часть, т.е. циклы, операторы которых должны выполняться строго последовательно, поскольку они, вообще говоря, произвольным образом зависят друг от друга по данным. Будь у нас система с общей памятью, в этом не было бы никакой проблемы, но HPF делает код для вычислителя, процессоры которого общей памяти не имеют. Для того чтобы распараллеленная часть выполнялась быстро, данные требуется распределить по процессорам определенным образом, а именно — так, чтобы обработка затрагивала в основном локальные данные. С этой задачей HPF, под руководством программиста, справится. Но как теперь обеспечить строго последовательную, оператор за оператором, обработку данных в еще не распараллеленной части, если данные эти

распределены по разным процессорам? Выход из этого тупика, в который транслятор сам себя поставил, справляясь с уже распараллеленной частью программы, только один. Код, изменяющий значения распределенных данных, следует выполнять в том процессоре, где эти данные находятся, а межпроцессорные обмены всеми изменившимися данными, включая одиночные переменные, которые транслятор не распределил, а просто размножил по процессорам, надо выполнять **после каждого оператора**. Иначе обработка не будет строго последовательной, и есть риск «потерять» зависимости по данным. В результате для еще не распараллеленной части программы получится крайне неэффективный код с громадным количеством крайне «плохих» из-за латентности коротких обменов.

Распараллеленная часть программы, действительно, ускорится. Не распараллеленная часть замедлится, возможно, в тысячи раз. С точки зрения программиста, экспериментирующего с распараллеливанием, программа просто «зависнет» — настолько медленным станет ее выполнение на многих процессорах. Увеличение числа процессоров будет только ухудшать картину.

Интуитивное ожидание программиста, что в еще не распараллеленной части программы транслятор «оставит все, как есть», обмануто самым вероломным образом. Совершенно аналогичная ситуация будет иметь место также в той части программы, про которую программист думает, что она распараллелена, но в распараллеливании которой он допустил ошибку.

Вполне возможно, что программист даже понимает, до некоторой степени, упомянутую выше проблему, и ожидает от транслятора построения в еще не распараллеленной части «разумного» кода. Например, программист может полагать, что транслятор сгруппирует последовательную обработку распределенных данных в крупные, локально независимые куски, и выстроит эти куски в цепочку, заставив процессоры работать поочередно, обмениваясь при этом сообщениями лишь изредка. Но ждать от транслятора таких способностей было бы неразумно. В самом деле, ведь HPF — инструмент распараллеливания, а не глубокой структурной перестройки программы с целью замедлить ее не очень сильно.

Наконец, программист мог бы выполнить эту гипотетическую работу по «не очень сильному замедлению» сам — ведь в его частной, конкретной программе это так просто сделать! Однако выполнить этого он не сможет — HPF обладает слишком высоким

уровнем абстракции, не дает программисту управлять ветвями программы вручную. Принятая в HPF модель логически единого процесса не знает такого понятия, как «разумная структурная перестройка последовательной программы, которую распараллелить не удалось». В языке просто нет таких директив.

В итоге мы получили типичный случай автомата, который прекрасно работает, если работает, и не дает переключиться на ручное управление, если отказал. Такое поведение транслятора страшно раздражает программиста тем, что оно контринтуитивно. В самом деле, общаясь с традиционными трансляторами для одного процессора, программист привык совсем к другому.

Во-первых, он привык к тому, что транслятор полностью изолирует его от системы команд процессора. Можно годами писать программы на Фортране или С и понятия не иметь о том, сколько в процессоре регистров, и чему равен код команды «сложить с двойной точностью». Если же транслятор и справляется с генерацией кода хуже, чем «идеальный программист на языке ассемблера», то это ухудшение — на проценты, от силы — вдвое, второе, но не в сотни же раз! HPF не обладает этим свойством. Он избавляет программиста от необходимости выписывать обмены данными с помощью MPI, но заставляет его думать о своей программе фактически в терминах MPI. А ведь было обещано распараллелить автоматически!

Во-вторых, аналогичным свойством «разумной аддитивности» обладают опции режимов оптимизации, предусмотренные в трансляторе. Их можно подбирать, не совсем понимая, что именно они делают, в надежде получить ускорение программы, иногда — кратное, но никогда — не замедление в сотни раз. Можно постепенно наращивать уровень оптимизации, добавлять все новые опции, и, обнаружив в некоторый момент замедление вместо ускорения, понять, что для данного алгоритма данная опция не подходит. Директивы HPF, как мы видим, свойством этим совершенно не обладают. Их можно написать либо все сразу, причем правильно, либо программа практически не будет работать.

Таким образом, невозможность пошагового (или, как иногда говорят, *инкрементального*) распараллеливания означает, в переводе на обиходный язык, что программу на HPF **очень трудно отлаживать**. При этом необходимо знать и уметь почти все то, что знает и умеет программист, использующий MPI. Не удивительно, что на практике пользуются этим языком мало.

Программисту, понимающему MPI, такой язык не очень нужен, не понимающему же — бесполезен, он вряд ли сумеет им воспользоваться.

Серьезным недостатком HPF является отсутствие возможности организации конвейерного параллелизма. Организовать же конвейер «на низком уровне» практически невозможно по причинам, упомянутым выше — HPF «не пускает» программиста на низкий уровень.

Последнее утверждение, строго говоря, не вполне верно. Средство перехода на «совсем низкий уровень» в HPF имеется. В этой роли выступает аппарат *extrinsic* — функций.

Замечание. Автор не берет на себя смелость предложить русский перевод этого термина. В оригинале он, очевидно, был придуман по аналогии с широко применяемым в языках программирования термином «*intrinsic*», который относится к стандартным функциям и обычно переводится как «встроенная (функция)». Если «*extrinsic*» — это антоним к «*intrinsic*», то рискнем выбрать из предлагаемых в словаре синонимов вариант «несобственная (функция)».

Несобственная функция — это функция, написанная на обычном, последовательном языке программирования, например, на «обычном» Фортране. Вызов такой функции из программы на HPF означает «расщепление» единого процесса, в котором логически выполняется программа на HPF, на хорошо знакомые нам ветви параллельной программы, которые выполняются независимо. Пока все ветви не выполнят возврат из функции, их работа, в том числе — взаимодействие, ничем не отличается от хорошо знакомой нам ситуации с MPI, например. Ветвь может узнать собственный номер, послать сообщение другой ветви и т. п. В описании языка подробно рассматривается передача параметров в такие функции. Конечно, при передаче в качестве параметра распределенного массива несобственная функция получит его локальную порцию. Для использования внутри несобственных функций предусмотрены специальные операторы двустороннего обмена сообщениями. Наконец, можно написать несобственную функцию на «обычном» Фортране с использованием MPI.

Таким образом, конвейерный параллелизм в программе на HPF записать можно, но для этого необходимо «убрать» весь конвейер внутрь одной несобственной функции, и запрограммировать его в терминах двусторонних обменов сообщениями. Это дает формальную возможность выхода из тупика, но снова при-

водит к уже знакомой нам парадоксальной ситуации: для того чтобы получить возможность не задумываться о технических особенностях параллельной реализации, надо предварительно очень тщательно эти особенности продумать.

Следует также отметить, что выбор в качестве базового языка для этой программистской модели именно Фортрана (а не С) не случаен. Наличие в С, в отличие от Фортрана, развитой адресной арифметики (полноценных указателей) сильно усложняет автоматический анализ текста программы.

Аналогичная российская разработка DVM [27], выполненная в Институте прикладной математики им. М. В. Келдыша РАН, включает в себя транслятор с языка, внешне очень похожего на HPF, но построенного на других принципах.

Как мы уже видели, серьезным недостатком HPF является то, что по тексту программы нельзя однозначно сказать, как именно транслятор ее «распараллелит». Пусть программист не указал для некоторого цикла `CHPF$ INDEPENDENT`. «Поймет» ли транслятор, что этот цикл — параллельный? Зависит это, вообще говоря, от конкретного транслятора и от того, правильно ли программист понимает его возможности. Если программист полагает, что транслятор его «поймет», но ошибается, получится крайне неэффективный параллельный код. Вместо желаемого ускорения программа замедлится, возможно, в тысячи раз. Зная об этой проблеме, и справедливо считая ее основной, разработчики DVM спроектировали такой набор директив, который **однозначно** определяет поведение транслятора. Транслятор DVM не ищет параллелизм в программе, он лишь выполняет то, что программист написал в форме директив. Если перед циклом не написано `CDVM$ PARALLEL`, транслятор не будет «догадываться», что этот цикл можно было бы исполнить параллельно.

В систему DVM входят трансляторы Фортран DVM, С DVM, небольшой транслятор HPF, средства отладки производительности и многое другое.

При разработке программ на С DVM настоятельно рекомендуется придерживаться «Фортрановского стиля программирования». Языки включают специальную форму параллельного цикла для организации конвейерного параллелизма. В настоящее время система интенсивно развивается в сторону гибридных моделей параллелизма, предназначенных для использования на установках с многопроцессорными узлами, в которых внутри узла общая память присутствует, а между узлами — нет.



Языкам DVM в полной мере присущи такие технологические преимущества HPF, как отсутствие необходимости сопровождать отдельно однопроцессорный и параллельный варианты текста и полная автоматизация файлового ввода — вывода для распределенных массивов.

**Модель непроецедурных языков.** Ярким представителем технологии в рамках этой модели является язык Норма [28], также разработанный в ИПМ им. М. В. Келдыша РАН.

Эта программистская модель предполагает преодоление Основного постулата с неожиданной стороны. Если, как мы отмечали выше, при обсуждении Основного постулата, операторы традиционного алгоритмического языка высокого уровня слишком мелки, чтобы программа была обозримой для автоматического анализа, может быть, можно решить проблему, значительно увеличив «размер» оператора? Способ увеличения «размера» известен — надо от записи алгоритма (последовательности **действий**, которые надлежит выполнить при решении задачи) перейти к записи **соотношений** между входными данными и результатами, доверив транслятору построить алгоритм самостоятельно, причем сразу — параллельный. Поступая так, мы освобождаем транслятор от решения «обратной задачи» — восстановления смысла программы по ее однопроцессорному варианту. Смысл записан в виде соотношений, точнее, функций, преобразующих входные данные в результат. Поэтому Норму называют неалгоритмическим, функциональным языком. Норма — несомненно, язык более высокого уровня, чем традиционные алгоритмические языки.

С точки зрения операторного программирования форма записи программ очень непривычная. Программа на Норме скорее напоминает текст из учебника по математическому анализу, чем текст программы. Например, присваивать значение данной переменной можно только один раз. Запись « $a = b + 1$ » означает не «сделать  $a$  в данном месте программы равным текущему значению  $b + 1$ », а «принять к сведению, что  $a$  равно  $b + 1$  (всегда)». Понятия «сейчас», «потом», «в данном месте программы» не существуют, как и понятие времени, в котором разворачивается выполнение программы. Поэтому операторы присваивания можно писать в любом порядке. Словом, Норма — действительно язык записи соотношений, а не алгоритмов. Ниже приводится пример записи двух функций, вычисляющих, соответственно, сумму и произведение двух матриц.

```

PART Add.
    a,b RESULT x
BEGIN
    DOMAIN PARAMETERS n=4.
    ijs: (is: (i=1..n); js: (j=1..n)).
    VARIABLE a,b,x DEFINED ON ijs.
    FOR ijs ASSUME x = a + b.
END PART.
PART Mult.
    a,b RESULT x
BEGIN
    DOMAIN PARAMETERS n=4.
    ijks: (ijs: (is: (i=1..n); js: (j=1..n));
           ks: (k=1..n)).
    VARIABLE a,b,x DEFINED ON ijs.
    VARIABLE t DEFINED ON ijks.
    FOR ijks/k=1
        ASSUME t = a[i,j=k] * b[i=k,j].
    FOR ijks/k=2..n
        ASSUME t[i,j,k] = t[i,j,k-1] +
            a[i,j=k] * b[i=k,j].
    FOR ijs ASSUME x = t[i,j,k=n].
END PART.

```

По такой функциональной, а не операторной, записи транслятор Норма успешно строит весьма эффективные параллельные программы.

Среди всех технологий, которые мы рассмотрели и еще рассмотрим, язык Норма занимает **совершенно особое место**. Эта технология не ориентирована ни на какой конкретный класс коммуникационного оборудования, как, впрочем, и на то, что узел вычислительной системы является фоннеймановской машиной. Строго говоря, она одна заслуживает названия **«действительно высокоуровневой»**, поскольку ее уровень абстракции от конкретного оборудования беспрецедентно высок. Как мы увидим ниже, во второй части курса, это свойство технологии программирования с наступлением третьей суперкомпьютерной революции становится особенно ценным.

В заключение перечислим наиболее доступные реализации рассмотренных технологий.

Системы DVM и Норма распространяются их разработчиками через сайт ИПМ им. М. В. Келдыша РАН.

Имеется, как минимум, одна реализация HPF весьма высокого качества, распространяемая в виде исходных текстов — система HPF Adaptor [37].

На этом краткий обзор моделей и технологий, производных от модели двустороннего обмена сообщениями завершен.

## 5.4. Модель: односторонний обмен сообщениями

Как нам уже известно, эта программистская модель является низкоуровневой базой для третьего по счету класса коммуникационного оборудования из спектра, построенного в подразд. 1.3.

При попытке построить формализованное описание этой модели бросаются в глаза две проблемы, аналогичных которым в случае предыдущей модели не было:

1) необходима система глобальной (общей для всех процессов) адресации данных. В самом деле, если обмены односторонние, то инициатор обмена обязан иметь «систему координат» для адресации памяти партнера по обмену;

2) необходима система синхронизации, отличная от самих обменов, поскольку односторонний обмен сам по себе никакой синхронизации не подразумевает, а организовать взаимодействие процессов без синхронизации вряд ли возможно.

Сейчас строить общее решение этих двух проблем в терминах программистской модели не будем, хотя такое решение и существует. Вместо этого посмотрим сначала, как эти проблемы решаются в рамках конкретной технологии.

### 5.4.1. Технология: *shmem*

*Shmem* (*Shared Memory* — разделяемая память) — как и MPI, представляет собой библиотеку функций, вызываемых из ветви параллельной программы [4,20]. Эта технология гораздо меньше известна и распространена, чем MPI. Тем не менее, она в течение многих лет служила низкоуровневой базой систем параллельного программирования — правда, только на суперкомпьютерах Cray и SGI [20, 51]. В частности, реализация MPI для этих машин построена на базе *shmem*. Впоследствии была выполнена реализация *shmem* для кластерной сети Quadrics [38]. Эту технологию можно считать такой же «классической» среди технологий одно-

сторонних обменов, как MPI — среди технологий двусторонних обменов. В отличие от MPI, `shmem` принципиально рассчитана на полностью гомогенные многопроцессорные вычислители. Предполагается не только общность системы команд и тем самым машинного представления чисел во всех узлах вычислителя, но и одинаковая на всех узлах операционная система, а также один и тот же исполняемый файл ветви параллельной программы во всех процессах.

Программа, использующая `shmem`, может быть написана на Фортране или С, здесь будем приводить примеры на С.

Программа на С, использующая `shmem`, должна включать файл заголовков `shmem.h`.

До обращения к любым другим функциям `shmem` следует вызвать функцию инициализации: `shmem_init()`. Аргументов у нее нет, возвращаемого значения — тоже. Для того чтобы узнать собственный номер и общее число процессов, следует вызвать функции `my_pe()` и `num_pes()` соответственно. Обе они не имеют аргументов и возвращают целочисленное значение.

Основой библиотеки являются функции односторонних обменов. Односторонняя запись в чужую память называется «put», одностороннее чтение из чужой памяти — «get».

Ориентация библиотеки на вычислители с одинаковым машинным представлением однотипных данных во всех узлах позволяет, в принципе, не учитывать в интерфейсе функций «put» и «get» типы передаваемых данных, ограничившись указанием длины передаваемой порции данных в байтах. Тем не менее, каждая из этих функций имеет в `shmem` несколько десятков частично перекрывающихся модификаций, причем имена этих вариантов функций строятся регулярным образом в зависимости от следующих свойств запроса:

- тип передаваемых данных;
- одно значение этого типа передается или массив;
- сплошной массив или расположенный в памяти с шагом (например, столбец двумерного массива в С или строка в Фортране).

Например, для передачи единственного значения типа «double» следует обратиться к функции:

```
shmem_double_p( addr, value, pe ),
```

а для передачи сплошного массива типа `float` — к функции:

```
shmem_float_put( addr, src, len, pe ).
```

Здесь `value` и `src` — передаваемое значение типа `double` и передаваемый массив типа `float`, соответственно, `len` — длина массива в компонентах, `pe` — номер процесса, в память которого осуществляется передача. Значение `addr` в обоих случаях — адрес в памяти процесса номер `pe` (указатель), по которому следует разместить передаваемое значение.

В отношении этого аргумента возникает совершенно естественный вопрос, непосредственно связанный с упоминавшейся выше проблемой глобальной адресации. В самом деле, откуда процесс — отправитель данных может получить осмысленное значение указателя в память совершенно другого процесса? Здесь нам на помощь приходит гомогенность вычислительной установки. Если все узлы одинаковы, и все исполняемые файлы одинаковы, то, по крайней мере, адреса всех переменных и массивов, объявленных статически, также одинаковы во всех процессорах (и процессах). Следовательно, указав собственный, локальный адрес переменной или массива, инициатор обмена, тем самым, указывает адрес одноименного объекта в процессе — получателе данных. На словах это можно сформулировать так: «Мою переменную `A` положить в переменную `B` процесса `B`».

Ясно, что таким образом категорически нельзя обращаться к данным, размещаемым в стеке (т. е. объявленным внутри функции без квалификатора `static`). Одностороннего доступа к таким данным `shmem` не поддерживает. Что же касается данных, размещаемых с помощью обращения к функции `malloc()` или ее производным, то их адресация указанным способом возможна, если написать специальный вариант функции `malloc()`, который будет выполняться как коллективная операция и специально обеспечивать равенство выделяемых адресов во всех процессах. В реализации `shmem` для `Quadrics`, например, такой функцией можно воспользоваться, но она не является частью `shmem` (предоставляется отдельно). Можно использовать и стандартный вариант `malloc()`, если организовать рассылку между участвующими в обмене процессами значений соответствующих адресов.

Вторая важная группа функций `shmem` — операции синхронизации. Ранее отмечалось, что в модели односторонних обменов наличие возможности синхронизироваться отдельно от собственно обмена принципиально, поскольку обмен сам по себе выполняется безо всякой синхронизации.

Основная операция синхронизации — барьер (барьерная синхронизация уже встречалась нам при рассмотрении `MPI`).

В shmem нет понятия, аналогичного понятию коммуникатора в MPI, но есть возможность выполнить барьерную синхронизацию не всех, а лишь указанных процессов.

Барьерную синхронизацию всех процессов выполняют функции `barrier()` и `shmem_barrier_all()`, без аргументов и возвращаемых значений. Для синхронизации подмножества процессов надо обратиться к функции:

```
shmem_barrier( start, stride, size, sync )
```

Здесь `start` — номер первого из процессов, участвующих в барьере, `size` — число процессов. Номера процессов, участвующих в барьере, начинаются со значения `start`, и идут с некоторым положительным шагом. Например, если значение `start` равно 3, а шаг — 4, в барьере участвуют процессы с номерами: 3, 7, 11, 15, ... Дополнительно требуется, чтобы шаг был степенью двойки. Аргумент `stride` задается равным как раз этой степени, т.е. двоичному логарифму требуемого шага. В приведенном только что примере значение `stride` должно было быть задано равным 2 ( $\log_2(4) = 2$ ). Наконец, `sync` — это рабочий массив типа `long`, длиной `_SHMEM_BARRIER_SYNC_SIZE` компонентов. Его адрес должен быть одинаковым во всех процессах, участвующих в барьере, и перед обращением к функции `shmem_barrier()` он должен быть расписан нулями. Расписать нулями достаточно перед первым обращением — после завершения барьера в массиве будут нули, если перед началом барьера они там были.

При выполнении барьерной синхронизации `shmem` гарантирует, что все выданные до барьера запросы типа `put` будут предварительно завершены.

Помимо барьерной синхронизации, имеется специфическая для модели односторонних обменов *операция ожидания изменения значения переменной*. Эти операции представлены семейством функций `shmem_wait()`. Так, при обращении к функции:

```
shmem_wait( &var, value )
```

процесс будет ждать, пока некоторый другой процесс не изменит значение переменной `var` (типа `long`) таким образом, чтобы оно стало не равным значению `value`. Переменная `var` — локальная для процесса, вызвавшего `shmem_wait()`, другой процесс изменяет ее, например, выполняя `shmem_put()`. Можно усложнить условие сравнения. Например, обращение к функции:

```
shmем_int_wait_until( &var, SHMEM_CMP_GT, value )
```

приведет к ожиданию, пока значение var не станет больше (GT — Greater Than) значения value.

Имеются варианты упомянутых функций для целочисленных типов int, short, long и longlong.

Наконец, к функциям синхронизации относятся две функции без аргументов и возвращаемых значений — shmем\_fence() и shmем\_quiet().

Выполняя запросы типа shmем\_put(), shmем может накапливать их внутри себя и, вообще говоря, переупорядочивать, что, конечно же, недопустимо, если используется синхронизация типа shmем\_wait().

Обращение к функции shmем\_fence() гарантирует, что все выданные до него запросы типа put к любому данному процессу будут завершены. Например, при последовательности обращений:

```
shmем_double_p( &a, 5, 7 );  
shmем_fence();  
shmем_double_p( &b, 4, 7 );
```

гарантируется, что в процессе номер 7 к моменту, когда переменная b получит значение 4, переменная a уже будет иметь значение 5, а не наоборот. Попросту говоря, shmем\_fence() «проталкивает буфер» обращений типа put к каждому из процессов.

Обращение к функции shmем\_quiet() действует аналогично, но не по отношению к каждому из процессов в отдельности, а по отношению ко всем процессам сразу. Например, при последовательности обращений:

```
shmем_double_p( &a, 5, 7 );  
shmем_quiet();  
shmем_double_p( &b, 4, 8 );
```

гарантируется, что к моменту, когда переменная b в процессе номер 8 получит значение 4, переменная a в процессе номер 7 уже будет иметь значение 5.

К возможностям синхронизации вплотную примыкают **атомарные операции** shmем.

Для читателей, не очень сведущих в системном программировании, поясним, что это такое. Для простоты пояснения будем говорить не о системе коммуникаций на базе односторонних обменов, а о системе общей памяти.

Пусть в общей памяти имеется переменная — счетчик, который считает события, фиксируемые несколькими процессорами. При наступлении события в процессоре А этот процессор увеличивает счетчик на единицу, при наступлении события в процессоре Б он, в свою очередь, делает то же самое. На аппаратном уровне увеличение счетчика на единицу в любом случае распадается на три действия:

- 1) прочитать значение счетчика;
- 2) прибавить к нему единицу;
- 3) записать новое значение счетчика.

Пусть теперь счетчик равен 5, и очередные события происходят одновременно в процессорах А и Б. Каждый из процессоров, фиксируя событие, читает значение 5, прибавит к нему единицу, и запишет новое значение, равное 6. Одно из двух произошедших одновременно событий осталось не посчитанным. Этой ошибки не произошло бы, если бы аппаратура умела выполнять сформулированную выше последовательность из трех действий атомарно, т. е. как одну неделимую операцию. При попытке выполнить одновременно две такие неделимые операции на двух процессорах, контроллер памяти вынужден был бы выстроить их в некоторую последовательность. В данном случае не важно, в какую именно — в любом случае ошибки удалось бы избежать.

Shmem реализует атомарное приращение счетчика, атомарный обмен данными между локальной и удаленной переменной, и некоторые другие атомарные действия. Например, обращение к функции:

```
shmem_int_fadd( &a, incr, 5 );
```

приведет к тому, что к целочисленной переменной а в процессе номер 5 будет прибавлено целочисленное значение incr, причем атомарным образом. Функция возвращает в качестве своего значения старое значение изменяемой переменной. Если требуется атомарный доступ к переменной в том числе и из процесса, в котором переменная расположена, то процесс — хозяин переменной обязан работать с ней как с удаленной, указывая в качестве третьего аргумента собственный номер. В приведенном только что примере переменная а находится в процессе номер 5, и, казалось бы, в этом процессе мы могли бы написать просто:



```
a += incr;
```

но атомарности это не обеспечит. Чтобы ее обеспечить, придется и в 5-м процессе, как и во всех прочих, написать:

```
shmem_int_fadd( &a, incr, 5 ).
```

Обзор возможностей синхронизации в shmem завершен.

В shmem, как и в MPI, имеются коллективные операции. В отличие от MPI, они охватывают лишь специальные виды обменов данными, такие, как операции редукции или широковещание. Среди коллективных операций shmem нет ничего подобного функции MPI\_Alltoallv(), т.е. нет средств организации таблично заданного набора обменов общего вида.

В заключение приведем текст нашей модельной программы, написанной с использованием shmem.

```
#include <stdio.h>
#include <stdlib.h>
#include <shmem.h>
/****/
#define MX 640
#define MY 480
#define NITER 10000
#define STEPITER 100
    static double f[MX][MY];
    static double df[MX][MY];
    static int size, rank, mx;
    static FILE *fp;
/****/
int main( int argc, char **argv )
{
    int i, j, n, m;
/****/
    shmem_init();
    rank = my_pe();
    size = num_pes();
    mx = (MX-2)+size-1)/size;
    if ( rank == (size-1) )
    {
        mx = (MX-2) - (size-1)*mx;
    }
    mx += 2;
    if ( !my_number )
```

```

    {
printf( "Solving heat conduction
        task on %d by %d grid\n", MX, MY);
        fflush( stdout );
    }
/* Initial conditions: */
    for ( i = 0; i < mx; i++ )
    {
        for ( j = 0; j < MY; j++ )
        {
            f[i][j] = df[i][j] = 0.0;
            if      ( (i == 0)
                    || (j == 0) ) f[i][j] = 1.0;
            else if ( (i == (mx-1))
                    || (j == (MY-1)) ) f[i][j] = 0.5;
        }
    }
/* Iteration loop: */
    for ( n = 0; n < NITER; n++ )
    {
/* Do all the transfers: */
        shmem_barrier_all();
        if ( rank > 0 )
            shmem_double_put( &f[mx-1][1],
                              &f[1][1], MY-2, rank-1 );
        if ( rank < (size-1) )
            shmem_double_put( &f[0][1],
                              &f[mx-2][1], MY-2, rank+1);
        shmem_barrier_all();
        if ( rank == 0 )
        {
            if ( !(n%STEPITER) )
                printf( "Iteration %d\n", n );
        }
/* Step of calculation starts here: */
        for ( i = 1; i < (mx-1); i++ )
        {
            for ( j = 1; j < (MY-1); j++ )
            {
                df[i][j] = ( f[i][j+1] + f[i][j-1] +
                             f[i-1][j] + f[i+1][j] )
                           * 0.25 - f[i][j];
            }
        }
    }

```

```

    }
    for ( i = 1; i < (mx-1); i++ )
    {
        for ( j = 1; j < (MY-1); j++ )
        {
            f[i][j] += df[i][j];
        }
    }
}
if ( !my_number )
{
    fp = fopen( "progre.dat", "w" );
    fclose( fp );
}
for ( j = 0; j < n_of_nodes; j++ )
{
    shmem_barrier_all();
    if ( j == rank )
    {
        fp = fopen( "progre.dat", "a" );
        for ( i = 1; i < (mx-1); i++ )
            fwrite( f[i]+1, MY-2,
                    sizeof(f[0][0]), fp );
        fclose( fp );
    }
}
shmem_barrier_all();
return 0;
}

```

#### 5.4.2. Технология: Global Arrays

Когда-то давно, во времена малых объемов оперативной памяти и коротких машинных адресов, программисты использовали для работы с большими объемами данных «библиотеки организации виртуальных массивов». Такой массив хранился в файле на жестком диске и отображался на массив в оперативной памяти порциями в режиме «скользящего окна», по явным запросам прикладной программы. В библиотеке ***Global Arrays*** (Глобальные массивы) [19] применяется та же дисциплина, но «виртуальные массивы» не хранятся на жестком диске, а «размазаны» по оперативным памятьям вычислительных узлов. Если массив очень большой, то допу-

скается также «размазывание» по оперативным памятьям и локальным жестким дискам узлов. Каждый узел работает с глобальным массивом, запрашивая отображение (фактически — одностороннее копирование) произвольной части массива в свою память. Распределенные массивы создаются в режиме коллективной операции и идентифицируются при работе символическими именами. Имеются отдельные запросы для явной синхронизации. Дополнительно предоставляется набор коллективных операций линейной алгебры: например, перемножение матриц, каждая из которых хранится в глобальном массиве.

Рассматривать эту технологию подробнее здесь не будем.

### 5.4.3. Технология: MPI-2

Во второй версии стандарта MPI среди прочих расширений присутствует также набор возможностей организации односторонних обменов. MPI, как мы знаем, является стандартом очень высокой степени общности, допускающим сосуществование в одной системе узлов с разными машинными представлениями чисел, не говоря уже о разных версиях ОС. В этих условиях проблема глобальной адресации данных уже не может быть решена (точнее, проигнорирована) так просто и напрямую, как это было сделано в `shmem`. Требуются более аккуратные способы, допускающие гетерогенность узлов.

В MPI-2 проблема глобальной адресации решается с помощью **окон**, т.е. областей, в пределах которых возможна глобальная адресация относительно начала области. Создание окна — коллективная операция. Она выполняется обращением к функции:

```
MPI_win_create( base, size, disp_unit,  
                info, comm, &win );
```

здесь:

`base` — указатель на область данных окна (адрес массива),

`size` — размер области в байтах,

`disp_unit` — размер элемента данных, из которых состоит область, в байтах (например, `sizeof(double)`),

`info` — дополнительная информация о режимах использования окна,

`comm.` — коммуникатор,

`win` — возвращаемый параметр типа `MPI_Win`, дескриптор окна, по которому программа впоследствии будет обращаться к созданному окну.

После окончания использования окно может быть уничтожено обращением к функции:

```
MPI_Win_free( &win );
```

Если некоторый процесс выполняет операцию put в некоторый другой процесс, он указывает, среди прочих аргументов, окно и смещение в нем. Данные будут доставлены в процесс — получатель и размещены в области данных, связанной с этим окном в процессе — получателе, с указанным смещением от начала. Операция get работает аналогично.

Для обеспечения атомарности процесс может захватить окно для монопольного доступа (MPI\_Win\_lock()) и освободить (MPI\_Win\_unlock()).

Аппарат синхронизации устроен, на взгляд автора, довольно тяжеловесно, а его описание в тексте стандарта вызывает поначалу некоторую оторопь. Обычная барьерная синхронизация из MPI-1 не годится, т. е. не приводит к завершению всех запущенных ранее односторонних обменов. Для синхронизации односторонних обменов применяются специальные функции. Больше всего похожа на барьер (хотя и отличается некоторыми деталями) функция MPI\_Win\_fence(). Это коллективная операция, подразумевающая исполнение барьера внутри себя в большинстве режимов применения.

Для синхронизации не по всему коммуникатору, в котором создано окно, а только по некоторым взаимодействующим процессам, используются функции MPI\_Win\_start(), MPI\_Win\_complete(), MPI\_Win\_post(), MPI\_Win\_wait(). Использование этих функций позволяет синхронизироваться в стиле, похожем на стиль двусторонних коммуникаций. Фактически, можно организовать randevu, но не на одном обмене, как в классическом «двустороннем» случае, а на произвольной серии односторонних обменов. Об использовании этих функций лучше читать в тексте стандарта. Приведенные там многочисленные рисунки многое поясняют.

В заключение, как всегда, о доступных реализациях.

Библиотека shmem по определению является инструментом очень низкого уровня, и реализована она, фактически, аппаратно. На том оборудовании, разработчики которого решили использовать shmem в качестве набора системных вызовов, эта библиотека входит в комплект поставки оборудования.

Библиотека Global Arrays распространяется свободно в виде исходных текстов.

Обзор технологий односторонних обменов на этом завершен, и мы продолжим наше движение по спектру классов коммуникационного оборудования. Теперь нам предстоит рассмотреть модель программирования, являющуюся наиболее непосредственной абстракцией NUMA-системы.

## 5.5. Модель: PGAS

Эта программистская модель является низкоуровневой базой для второго по счету класса коммуникационного оборудования из спектра, построенного в подразд. 1.3, а именно — для NUMA-систем.

Впервые в нашем обзоре мы рассматриваем модель, ориентированную на оборудование с общей памятью. Принято считать, что ориентация на общую память, в первую очередь, отражается на модели порождения и синхронизации процессов. Например, для систем SMP часто применяется модель динамически порождаемых *легковесных процессов*, или *нитей* (pthreads), синхронизирующихся с помощью расположенных в общей памяти семафоров. Хотя в технологиях модели PGAS термин «threads» и используется для обозначения процессов, модель легковесных процессов, динамически порождаемых в общем адресном пространстве, обычно не применяется. Вместо этого используется способ управления процессами, хорошо знакомый нам по предыдущим моделям. Как и в случае MPI или shmem, параллельная программа состоит из статически порождаемых ветвей, каждая из которых есть процесс с собственным адресным пространством. Отличие от уже рассмотренных моделей — в том, что адресные пространства ветвей параллельной программы частично перекрываются, и вместо средств передачи данных из одной ветви в другую основной упор делается на средства управления перекрестным доступом к этим перекрывающимся частям. Для синхронизации применяются, в основном, те же способы, что и в модели односторонних обменов, например, барьеры.

Также обычно предполагается, что многопроцессорный вычислитель — гомогенный, а исполняемый файл всех ветвей параллельной программы — один и тот же.

Описанная здесь модель статически порождаемых процессов с частично перекрывающимися адресными пространствами и, возможно, со значительной временной асимметрией доступа к

общим областям памяти из разных процессов, получила название *PGAS* (*P*artitioned *G*lobal *A*ddress *S*pace, Разделенное глобальное адресное пространство).

### 5.5.1. Технология: UPC

В отличие от технологий, рассмотренных выше, *UPC* (*U*nified *P*arallel *C*, Унифицированный параллельный C) [21] — это не библиотека, а язык программирования, на котором пишется ветвь параллельной программы. Этот язык отличается от C некоторыми дополнительными языковыми конструкциями. В отличие от рассмотренных выше HPF и DVM, эти дополнительные конструкции не оформлены как комментарии специального вида, т. е. программа на UPC, вообще говоря, не является правильной последовательной программой на «обычном» C.

UPC — язык явного задания параллелизма. Он ни в коей мере не претендует на то, чтобы автоматически превращать написанную пользователем последовательную или «почти последовательную» программу в ветвь программы параллельной. Как и в случае MPI или shmem, пользователь пишет именно ветвь параллельной программы, которая будет взаимодействовать с себе подобными буквально так, как напишет пользователь, безо всякого «автоматического распараллеливания» или «логически единого процесса».

Дополнительных, по сравнению с традиционным C, возможностей в UPC совсем немного. Приведем их полный список:

- 1) встроенные константы, равные собственному номеру ветви и общему числу ветвей;
- 2) дополнительный квалификатор «shared», позволяющий объявлять данные в общей памяти и указатели на них;
- 3) оператор барьерной синхронизации;
- 4) оператор параллельного цикла, позволяющий компактно записывать циклы, разные витки которых выполняются независимо в разных процессах;
- 5) дополнительные возможности библиотеки файлового ввода-вывода, по смыслу аналогичные тем, которые имеются в MPI-2 (см. ранее).

Важнейшей из этих дополнительных возможностей является, конечно, вторая.

Опишем кратко эти дополнительные возможности UPC, в основном следуя приведенному плану.

Программа на UPC должна включать один из двух файлов заголовков — `upc_relaxed.h` или `upc_strict.h` (но не оба). Разница будет разъяснена ниже. В каждом из этих файлов определяются встроенные целочисленные константы `MYTHREAD` и `THREADS`, равные собственному номеру ветви параллельной программы и общему числу ветвей, соответственно.

Адресное пространство процесса в UPC состоит из двух частей — *локальных данных* и *разделяемых данных*.

Локальные данные — это «обычные» данные языка C. У каждого процесса свой набор локальных данных, т.е. если в программе написано:

```
int a;
```

то это означает, что в каждом процессе имеется своя переменная «a».

Разделяемые данные, напротив, являются общими для всех процессов. При объявлении элементов разделяемых данных или указателей на них используется квалификатор `shared`. Таким образом, если в программе написано:

```
shared double b;
```

то это означает, что имеется одна, общая для всех процессов, переменная «b».

В UPC не предусмотрена, даже на логическом уровне, специальная общая память, отличная от локальных памяти процессов. Разделяемые элементы данных доступны из всех процессов, но каждый из них находится в локальной памяти ровно одного процесса, или *принадлежит* ему. Понятие принадлежности (в оригинале описания называемое «affinity») — это понятие логическое, архитектурное, а не реализационное.

Чтобы выяснить подробнее, как это все работает, нам надо научиться для любого адресуемого элемента данных отвечать на следующие вопросы:

- где этот элемент находится;
- где он доступен;
- каковы способы доступа?

Перечислим все возможные случаи.

Локальные данные размножены по процессам, в каждом процессе находится свой экземпляр таких данных. Каждый такой экземпляр доступен тому процессу, в котором он находится. Способ доступа — по имени, или по локальному («обычному») указателю.



Местонахождение разделяемых данных зависит от того, как они объявлены или выделены динамически. Например, одиночные разделяемые переменные все находятся в нулевом процессе. При объявлении массива с квалификатором «shared» или динамическом выделении разделяемой области памяти, аналогичном malloc, имеется возможность распределить массив или выделяемую область по процессам регулярным образом, например, в порядке номеров, начиная с нулевого, блоками равного размера. Эти данные не размножены, существуют в единственном экземпляре, и доступны всем процессам. Доступ возможен либо по имени, либо по указателю типа «shared\*». Индексированный доступ к элементу массива — частный случай доступа по указателю.

Разделяемые данные, принадлежащие процессу, в этом и только в этом процессе могут адресоваться по локальному («обычному») указателю. Например, если массив из 100 элементов распределен по 10 процессам блоками по 10 элементов, то элементы с 10-го по 19-й включительно могут адресоваться из процессора номер 1 по локальному указателю.

В отношении трех различных способов доступа к разделяемым данным имеется следующая временная асимметрия:

- доступ по указателю типа «shared\*» к элементу данных, который не принадлежит данному процессу — самый медленный;

- доступ тем же способом к элементу данных, который данному процессу принадлежит — гораздо более быстрый;

- доступ к элементу, принадлежащему данному процессу, через локальный указатель — еще более быстрый, по скорости не отличим от доступа к локальным данным.

Эта трехуровневая временная асимметрия — свойство языка, а не конкретной реализации. Программист обязан учитывать ее в структуре своей программы. В критическом по времени выполнении цикле программа должна осуществлять доступ преимущественно к тем элементам разделяемых данных, которые принадлежат данному процессу, и, по возможности, пользоваться для этого локальными указателями.

Как конкретно все это записывается в UPC?

Одиночные переменные, объявленные с квалификатором «shared», как мы знаем, всегда принадлежат нулевому процессу. Массивы (или области, адресуемые указателем), объявленные как «shared», распределяются по принадлежности

**блочно-циклическим** способом, причем размер блока задается при объявлении. Например, рассмотрим такое объявление массива:

```
shared [10] double b[100];
```

В этом случае массив `b` типа `double` и длиной 100 будет, во-первых, единственным и доступным для всех ветвей, а во-вторых — распределенным блоками по 10 элементов между процессами, по порядку номеров. Например, если процессов 10, получим такую картину:

Процесс № 0:	Процесс № 1:	...	Процесс № 9:
Быстро	Быстро		Быстро
доступные	доступные		доступные
элементы	элементы		элементы
<code>b[0] — b[9]</code>	<code>b[10] — b[19]</code>		<code>b[90] — b[99]</code>

Если бы массив был объявлен как имеющий длину, например, 75, то двум последним процессам его элементов не хватило бы. Для этих процессов все элементы этого массива были бы доступны медленно. Такое распределение массива по процессам, когда массив поделен на сплошные блоки, каждый из которых «достался» отдельному процессу, называется **блочным**. Это частный случай блочно-циклического распределения.

А если бы массив был объявлен имеющим длину 200? Тогда распределение блоков длиной 10 пришлось бы циклически повторить для имеющихся процессов, и мы бы получили такую картину:

Процесс № 0:	Процесс № 1:	...	Процесс № 9:
Быстро	Быстро		Быстро
доступные	доступные		доступные
элементы	элементы		элементы
<code>b[0] — b[9]</code>	<code>b[10] — b[19]</code>		<code>b[90] — b[99]</code>
<code>b[100] — b[109]</code>	<code>b[110] — b[119]</code>		<code>b[190] — b[199]</code>

Это общий случай рассматриваемого нами блочно-циклического распределения.

Наконец, пусть размер блока в объявлении массива указан равным 1, или, что означает то же самое, не указан вообще. При этом мы получим второй частный случай — **циклическое** распределение массива, при котором нулевому процессу «достанутся» элементы с номерами: 0, 10, 20, 30, ..., 90.

Блочно-циклическое распределение общего вида широко применяется в параллельных алгоритмах линейной алгебры. А для нашей модельной задачи, например, наиболее естественным является блочное распределение.

Из всего изложенного пока не ясно, как именно процесс, которому принадлежат некоторые разделяемые данные, может получить локальный указатель на эти данные. Это делается простым присваиванием разнотипных указателей. Например, пусть в программе написано:

```
shared [10] double b[100];  
double *lb;  
double a;
```

Тогда в процессе номер 1 можно написать:

```
lb = b+10;  
....  
a = lb[7];
```

Если бы мы, вместо этого, написали:

```
lb = b+30;
```

или:

```
a=lb[20];
```

то это было бы ошибкой типа «эффект не предсказуем», поскольку мы попытались бы адресовать локальным указателем данные, не расположенные в локальном адресном пространстве данного процесса.

Краткое описание строения адресного пространства в UPC завершено.

Оператор барьерной синхронизации в UPC имеет вид:

```
upc_barrier;
```

К возможностям синхронизации вплотную примыкают возможности управления когерентностью доступа к разделяемым переменным из разных процессов. Выше, в главе, посвященной обзору основных понятий архитектуры процессоров, мы говорили о когерентности кэш. В данном случае понятие когерентности означает примерно то же самое. Доступ к разделяемым данным называется когерентным, если в любой момент времени любой процесс видит одинаковые значения всех разделяемых переменных. Нарушения когерентности могут происходить (и происхо-

дять) за счет того, что физически между локальными памятьми процессоров находится некоторая коммуникационная среда, время передачи запросов по которой отлично от нуля. Например, если процесс А изменил значение, принадлежащее процессу В, и начал взаимодействовать с процессом В, то процесс В, обратившись к процессу В, может обнаружить там еще старое значение — новое из процесса А еще «не дошло». Реализация полной когерентности доступа к разделяемым переменным почти на любой аппаратуре бывает очень не эффективной. В реальных программах обычно считают, что общая память когерентна, например, сразу после барьерной синхронизации, но не обязана быть когерентной в любой промежуточный момент.

Общая память UPC становится когерентной, если в качестве стандартного файла заголовков выбрать вариант `upc_strict.h` (вместо рекомендуемого варианта — `upc_relaxed.h`). В большинстве реализаций это приведет к очень неэффективным программам. Синхронизация между процессами будет происходить автоматически буквально при каждом доступе к какому-либо элементу разделяемых данных. Когерентность также можно включить в какой-то части программы (с помощью директивы `#pragma upc strict`) и выключить (с помощью директивы `#pragma upc relaxed`). Наконец, можно сделать когерентным доступ к отдельным переменным или массивам, объявив их со специальным дополнительным квалификатором, например:

```
strict shared long a;
```

В UPC нет атомарных операций над разделяемыми данными, подобных атомарным операциям `shmem`, но есть возможность более общего вида, позволяющая легко реализовать, в частности, произвольные атомарные последовательности действий. Эта возможность, широко используемая во многих технологиях взаимодействующих легковесных процессов с общим адресным пространством, называется аппаратом **семафоров взаимного исключения**, или **замков** (`lock`).

Над переменными специального типа «замок» определены две операции — «захватить» и «освободить». Захватить один замок может только один процесс одновременно — все остальные желающие будут ждать («висеть» в операции «захватить»), пока замок не освободится.

Если в программе есть последовательность действий, которая должна выполняться атомарно, с ней надо связать переменную

типа замок. Эта переменная будет использоваться для защиты данной последовательности действий от одновременного выполнения ее несколькими процессами. Чтобы обеспечить такую защиту, достаточно во всех процессах перед началом этой последовательности выполнить захват этого замка, а по завершении последовательности — его освобождение.

Для работы с замками применяются не специальные операторы языка, а стандартные библиотечные функции.

Тип переменных «замок» называется в UPC `upc_lock_t`. Переменные эти порождаются обращением к специальной функции и доступны через указатели. Например:

```
.....
upc_lock_t *lock;
.....
lock = upc_all_lock_alloc();
if ( lock != NULL ) .....
.....
.....
upc_lock_free( lock );
```

Здесь `lock` — указатель на замок, `upc_all_lock_alloc()` — библиотечная функция порождения замка, `upc_lock_free()` — библиотечная функция уничтожения замка. В промежутке между обращениями к этим функциям замок существует, и им можно пользоваться, например:

```
upc_lock( lock );           // захват замка
.....
// атомарная последовательность действий
upc_unlock( lock ); // освобождение замка
```

Обзор основных понятий UPC, связанных с синхронизацией, завершен.

Оператор параллельного цикла в UPC очень похож на оператор «for» традиционного С. Вместо «for» пишется «`upc_forall`», а в скобках указываются, через точку с запятой, не три выражения, а четыре. Смысл первых трех совпадает со смыслом трех выражений оператора «for», а четвертое выражение вычисляется в начале каждого витка цикла и определяет, будет ли данный виток выполняться в данном процессе. Это выражение может быть либо целочисленным, либо указателем на «`shared`». В первом случае виток цикла выполняется в том процессе, номер которого равен значению выражения. Во втором случае он выполняется в

том процессе, которому принадлежит адресуемое указателем значение. Второй случай кажется чем-то сложным и неестественным, но только на первый взгляд. Рассмотрим простейший пример:

```
shared [10] double a[100];
shared [10] double b[100];
int i;
....
upc_forall( i = 0; i < 100; i++;
            &a[i] ) a[i] = b[i];
```

Здесь один «shared» массив переписывается в другой, причем присваивания элементов массивов выполняются в тех процессах, которым соответствующие элементы принадлежат.

UPC имеет мощную библиотеку стандартных функций, в которую входят, в частности, оптимизированные версии коллективных обменов данными и копирования блоков памяти между разными процессами. Важнейшей частью этой библиотеки являются функции параллельного ввода-вывода, аналогичные тем, которые предусмотрены в MPI-2. Они используются, в частности, в варианте реализации нашей модельной программы на UPC, текст которого приводится ниже.

```
#include <upc_io.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <upc_relaxed.h>
/****/
#define MX 1000
#define MY 1000
#define NITER 100
#define STEPITER 10
    static shared [MX*MY/THREADS]
        double f[MX][MY];
    static shared [MX*MY/THREADS]
        double df[MX][MY];
    static FILE *fp;
/****/
int main( int argc, char **argv )
{
```

```

int i, j, n, m;
struct timeval ts_st, ts_end;
double howlong;
upc_file_t *fp;
upc_hint_t hint;
/***/
if ( !MYTHREAD )
{
printf( "Solving heat conduction
        task on %d by %d grid
        by %d processors\n",
        MX, MY, THREADS );
fflush( stdout );
}
/* Initial conditions: */
upc_forall ( i = 0; i < MX; i++;
            &f[i][0] )
{
for ( j = 0; j < MY; j++ )
{
    f[i][j] = df[i][j] = 0.0;
    if      ( (i == 0)
              || (j == 0) ) f[i][j] = 1.0;
    else if ( (i == (MX-1))
              || (j == (MY-1)) ) f[i][j] = 0.5;
}
}
if ( !MYTHREAD )
    gettimeofday( &ts_st, NULL );
/* Iteration loop: */
for ( n = 0; n < NITER; n++ )
{
if ( !MYTHREAD )
{
    if ( !(n%STEPITER) )
        printf( "Iteration %d\n", n );
}
}
/* Step of calculation starts here: */
upc_forall ( i = 1; i < (MX-1);
            i++; &f[i][0] )
{
for ( j = 1; j < (MY-1); j++ )
{

```

```

        df[i][j] = ( f[i][j+1] + f[i][j-1] +
                     f[i-1][j] + f[i+1][j] )
                     * 0.25 - f[i][j];
    }
}
upc_barrier;
upc_forall ( i = 1; i < (MX-1);
             i++; &f[i][0] )
{
    for ( j = 1; j < (MY-1); j++ )
    {
        f[i][j] += df[i][j];
    }
}
upc_barrier;
}
if ( !MYTHREAD )
{
    gettimeofday( &ts_end, NULL );
    howlong = ts_end.tv_sec +
              (ts_end.tv_usec / 1000000.0);
    howlong -= ts_st.tv_sec +
              (ts_st.tv_usec / 1000000.0);
    printf( "Elapsed time: %f sec\n",
            howlong );
}
fp = upc_all_fopen( "progreve_upc_1.dat",
                   UPC_COMMON_FP | UPC_WRONLY |
                   UPC_CREATE | UPC_TRUNC,
                   0, &hint );

if ( !fp )
{
    printf( "Node %d failed to open a file\n",
            MYTHREAD );
    return( -1 );
}
upc_all_fwrite_shared( fp, &f[0][0],
                      MX*MY/THREADS, sizeof( f[0][0] ),
                      MX*MY, UPC_IN_ALLSYNC |
                      UPC_OUT_ALLSYNC );
upc_all_fclose( fp );
return 0;
}

```



Бросается в глаза некоторое внешнее сходство данной технологии с HPF. Естественно, хотелось бы понять, в какой мере технологии UPC присущи методические проблемы, препятствующие широкому использованию HPF, о которых мы довольно подробно рассказывали выше. Рассмотрим этот вопрос подробнее, отталкиваясь от приведенного только что примера модельной программы.

Существенных отличий от HPF в рассматриваемой технологии два. Во-первых, UPC — язык гораздо более низкоуровневый, чем HPF. Программист управляет распределением работы, с одной стороны, и доступом к чужим для процесса данным, с другой, непосредственно, в терминах номеров ветвей, а не косвенно, выписывая директивы в рамках логически единого процесса. Во-вторых, сама дисциплина доступа к «чужим» данным гораздо проще, чем в HPF. Доступ этот является односторонним, т. е. структурно не отличается от доступа к локальным данным, а лишь занимает больше времени. Сочетание этих двух факторов дает основания для некоторого оптимизма. В самом деле, выше мы видели, что именно сочетание высокого уровня абстракции языка, с одной стороны, и структурной сложности организации «неудобных» доступов к чужим данным, с другой, препятствуют инкрементальному распараллеливанию в случае HPF. В случае UPC эта трудно преодолимая «пропасть» сокращается с обеих сторон. Удастся ли осуществить инкрементальное распараллеливание? Насколько плохо будут вести себя еще не полностью распараллеленные версии программы? Какие неожиданные сюрпризы, подобно рассмотренному нами замедлению еще не распараллеленной части программы в сотни раз, способен преподнести программисту транслятор UPC?

Сначала уточним, как в данном случае следует понимать инкрементальное распараллеливание. Ведь логически единого процесса нет, ветви программы порождаются статически, и программа, в любом случае, является формально полностью параллельной, от начала и до конца. Этот факт, тем не менее, не может помешать программисту размножить данные и вычисления, т. е. выполнить одну и ту же работу одновременно в разных процессах. Легко убедиться, что в смысле скоростного выигрыша это то же самое, что и последовательное выполнение соответствующей части программы. Так что об инкрементальном распараллеливании говорить вполне можно. Тем более — об инкрементальном совершенствовании параллельной реализации, которая, как и

всякая программа, поначалу может оказаться очень не эффективной.

Как мы помним, в случае HPF транслятор не мог разумно организовать взаимодействие ветвей создаваемой параллельной программы, если исходная программа не была распараллелена до конца. Это приводило к реализациям настолько неэффективным, что их практически невозможно было отлаживать, а главное — нелегко было понять, в каком месте, в каком конкретно операторе исходной программы кроется катастрофическое замедление. В случае UPC программисту это не грозит, просто потому, что транслятор UPC организацией взаимодействия ветвей не занимается. UPC — язык явного задания параллелизма. Транслятор UPC не генерирует неожиданных для программиста неэффективных обменов данными просто потому, что он вообще не генерирует никаких обменов данными, кроме тех, которые программист записал явно и непосредственно. Значит, если программа выполняется медленнее, чем хотелось бы, всегда можно «указать пальцем» совершенно конкретный оператор в исходной программе, в котором и кроется замедление, причем оператор этот будет содержать в себе достаточно часто выполняемое обращение к распределенным данным.

Таким образом, в языке просто нет средств, неумелое использование которых заставило бы программу замедлиться в сотни раз «на ровном месте». Похоже, инкрементальная разработка программ на этом языке возможна. Продемонстрируем ее на примере нашей модельной программы. Приведенный выше текст будем считать первой, «наивной» попыткой реализации. Реализация эта полностью параллельная, поскольку в данном случае это было совсем не трудно, но, возможно, не очень эффективная. В чем именно может быть причина низкой эффективности? Например, мы знаем, что доступ к данным, даже локальным, выполняемый с использованием указателей типа «shared\*», более медленный, чем доступ с использованием локальных указателей. В нашей программе доступ к локальным данным никак не оптимизирован, поскольку мы совершенно не представляем себе, насколько велики такого рода потери именно на нашей вычислительной установке, и решили для начала написать программу «в лоб», наиболее простым способом. Необходимые изменения будем вносить постепенно, ровно в том количестве, в каком это потребуется. Это и называется инкрементальной разработкой.

Приводимые далее конкретные показатели быстродействия были получены на реальной вычислительной установке. Расчет проводился на восьми процессорах. Выполнение однопроцессорного варианта программы заняло 1,8 с. Выполнение на восьми процессорах аналогичной программы, написанной с использованием MPI, заняло 0,28 с. Это — «идеал», к которому следует стремиться.

Выполнение на том же числе процессоров нашего «наивного» варианта заняло 15,7 с. По сравнению с однопроцессорным вариантом, программа замедлилась почти в 10 раз, но не в 1000. Программу уже можно отлаживать, а большего нам пока и не требуется.

Попробуем перейти к локальным указателям в обработке массива приращений. Это — совсем простая оптимизация, ведь массив приращений вообще никак не участвует в обменах данными между процессорами. Получится такая программа:

```
#include <upc_io.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <upc_relaxed.h>
/****/
#define MX 1000
#define MY 1000
#define NITER 100
#define STEPITER 100
    static shared [MX*MY/THREADS]
        double f[MX][MY];
    static shared [MX*MY/THREADS]
        double df[MX][MY];
    static FILE *fp;
/****/
int main( int argc, char **argv )
{
    int i, j, n, m;
    struct timeval ts_st, ts_end;
    double howlong;
    upc_file_t *fp;
    upc_hint_t hint;
    double *lf, *ldf;
```

```

/****/
    if ( !MYTHREAD )
    {
        printf( "Solving heat conduction
                  task on %d by %d grid
                  by %d processors\n",
                  MX, MY, THREADS );
        fflush( stdout );
    }
/* Initial conditions: */
    upc_forall ( i = 0; i < MX; i++; &f[i][0] )
    {
        for ( j = 0; j < MY; j++ )
        {
            f[i][j] = df[i][j] = 0.0;
            if      ( i == 0 )
                || ( j == 0 ) ) f[i][j] = 1.0;
            else if ( i == (MX-1))
                || ( j == (MY-1)) ) f[i][j] = 0.5;
        }
    }
    if ( !MYTHREAD )
        gettimeofday( &ts_st, NULL );
/* Iteration loop: */
    for ( n = 0; n < NITER; n++ )
    {
        if ( !MYTHREAD )
        {
            if ( !(n%STEPITER) )
                printf( "Iteration %d\n", n );
        }
    }
/* Step of calculation starts here: */
    upc_forall ( i = 1; i < (MX-1);
                  i++; &f[i][0] )
    {
        for ( j = 1; j < (MY-1); j++ )
        {
            df[i][j] = ( f[i][j+1] + f[i][j-1] +
                          f[i-1][j] + f[i+1][j] )
                        * 0.25 - f[i][j];
        }
    }
    upc_barrier;

```

```

upc_forall ( i = 1; i < (MX-1);
            i++; &f[i][0] )

{
    lf = &f[i][0];
    ldf = &df[i][0];
    for ( j = 1; j < (MY-1); j++ )
    {
        lf[j] += ldf[j];
    }
}
upc_barrier;
}
if ( !MYTHREAD )
{
    gettimeofday( &ts_end, NULL );
    howlong = ts_end.tv_sec +
              (ts_end.tv_usec / 1000000.0);
    howlong -= ts_st.tv_sec +
              (ts_st.tv_usec / 1000000.0);
    printf( "Elapsed time: %f sec\n",
            howlong );
}
fp = upc_all_fopen( "progreve_upc_1.dat",
                   UPC_COMMON_FP | UPC_WRONLY |
                   UPC_CREATE | UPC_TRUNC, 0,
                   &hint );

if ( !fp )
{
    printf( "Node %d failed to open a file\n",
            MYTHREAD );
    return( -1 );
}
upc_all_fwrite_shared( fp, &f[0][0],
                      MX*MY/THREADS,
                      sizeof( f[0][0] ),
                      MX*MY, UPC_IN_ALLSYNC |
                      UPC_OUT_ALLSYNC );
upc_all_fclose( fp );
return 0;
}

```

Время выполнения этого варианта составляет уже 12,9 с. Похоже, дело не в недостаточной эффективности работы с локаль-

ными данными через указатели, рассчитанными на данные распределенные, по крайней мере, не только в ней. Все же постараемся полностью изъять из критического цикла программы использование распределенных указателей для работы с локальными данными. Текст этого (третьего) варианта программы выглядит так:

```
#include <upc_io.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <upc_relaxed.h>
/****/
#define MX 1000
#define MY 1000
#define NITER 100
#define STEPITER 100
    static shared [MX*MY/THREADS]
        double f[MX][MY];
    static shared [MX*MY/THREADS]
        double df[MX][MY];
    static FILE *fp;
/****/
int main( int argc, char **argv )
{
    int i, j, n, m;
    struct timeval ts_st, ts_end;
    double howlong;
    upc_file_t *fp;
    upc_hint_t hint;
    double *lf[MX], *ldf[MX];
    int ib, ie;
/****/
    if ( !MYTHREAD )
    {
        printf( "Solving heat conduction
                task on %d by %d grid
                by %d processors\n",
                MX, MY, THREADS );
        fflush( stdout );
    }
}
```

```

/* Initial conditions: */
upc_forall ( i = 0; i < MX; i++; &f[i][0] )
{
    lf[i] = (double*)(&f[i][0]);
    ldf[i] = (double*)(&df[i][0]);
    for ( j = 0; j < MY; j++ )
    {
        f[i][j] = df[i][j] = 0.0;
        if ( (i == 0) || (j == 0) ) f[i][j] = 1.0;
        else if ( (i == (MX-1))
                  || (j == (MY-1)) ) f[i][j] = 0.5;
    }
}

if ( !MYTHREAD )
    gettimeofday( &ts_st, NULL );

/* Iteration loop: */
ib = MX/THREADS*MYTHREAD;
ie = MX/THREADS*(MYTHREAD+1);
for ( n = 0; n < NITER; n++ )
{
    if ( !MYTHREAD )
    {
        if ( !(n%STEPITER) )
            printf( "Iteration %d\n", n );
    }
}

/* Step of calculation starts here: */
i = ib;
for ( j = 1; j < (MY-1); j++ )
    ldf[i][j] = ( lf[i][j+1] + lf[i][j-1] +
                  f[i-1][j] + lf[i+1][j] )
                * 0.25 - lf[i][j];

i = ie - 1;
for ( j = 1; j < (MY-1); j++ )
    ldf[i][j] = ( lf[i][j+1] + lf[i][j-1] +
                  lf[i-1][j] + f[i+1][j] )
                * 0.25 - lf[i][j];
for ( i = ib+1; i < (ie-1); i++ )
{
    for ( j = 1; j < (MY-1); j++ )
    {
        ldf[i][j] = ( lf[i][j+1] + lf[i][j-1] +
                      lf[i-1][j] + lf[i+1][j] )
                    * 0.25 - lf[i][j];
    }
}

```

```

    }
}
upc_barrier;
upc_forall ( i = 1; i < (MX-1);
            i++; &f[i][0] )
{
    for ( j = 1; j < (MY-1); j++ )
    {
        lf[i][j] += ldf[i][j];
    }
}
upc_barrier;
}
if ( !MYTHREAD )
{
    gettimeofday( &ts_end, NULL );
    howlong = ts_end.tv_sec +
              (ts_end.tv_usec / 1000000.0);
    howlong -= ts_st.tv_sec +
              (ts_st.tv_usec / 1000000.0);
    printf( "Elapsed time: %f sec\n",
            howlong );
}
fp = upc_all_fopen( "progrev_upc_3.dat",
                   UPC_COMMON_FP |
                   UPC_WRONLY |
                   UPC_CREATE |
                   UPC_TRUNC, 0, &hint );

if ( !fp )
{
    printf( "Node %d failed to open a file\n",
            MYTHREAD );
    return( -1 );
}
upc_all_fwrite_shared( fp, &f[0][0],
                      MX*MY/THREADS,
                      sizeof( f[0][0] ),
                      MX*MY,
                      UPC_IN_ALLSYNC |
                      UPC_OUT_ALLSYNC );
upc_all_fclose( fp );
return 0;
}

```



Время выполнения третьего варианта составляет 4,9 с. Теперь понятно, как распределяются накладные расходы при работе нашей программы. Примерно 15 с работы «наивного» варианта практически целиком представляют собой накладные расходы, две трети которых приходится на обращение к локальным данным с помощью указателей, которые применимы в общем случае, т.е. для обращения также и к удаленным данным. Оставшаяся треть — время выполнения третьего варианта программы — очевидно, приходится на обращение за удаленными данными слишком мелкими порциями (заодно мы узнали, что данная реализация UPC, очевидно, не выполняет оптимизацию таких обращений, т.е. буквально генерирует по одному доступу на каждый элемент «чужой» порции массива). Чтобы это исправить, придется еще раз переписать программу, воспользовавшись стандартной функцией копирования распределенных данных вместо обращений к отдельным элементам распределенного массива. Используем хорошо известное нам по предыдущим версиям модельной программы представление локальной порции с буферными зонами, для копирования же «чужих» данных в эти буферные зоны применим стандартную функцию `upc_tetrcpy()`. Этот (четвертый) вариант программы выглядит так:

```
#include <upc_io.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <upc_relaxed.h>
/****/
#define MX 1000
#define MY 1000
#define NITER 100
#define STEPITER 100
    static shared [MX*MY/THREADS]
        double f[MX][MY];
    static shared [MX*MY/THREADS]
        double df[MX][MY];
    static shared [MY] double prev[MY*THREADS];
    static shared [MY] double next[MY*THREADS];
    static FILE *fp;
/****/

int main( int argc, char **argv )
```

```

{
    int i, j, n, m;
    struct timeval ts_st, ts_end;
    double howlong;
    upc_file_t *fp;
    upc_hint_t hint;
    double *lf[MX], *ldf[MX];
    int ib, ie;

    /***/
        if ( MX%THREADS )
        {
            /* we want exactly one block of each array
               on a processor: */
            if ( !MYTHREAD )
                printf( "Matrix row count - MX - is %d,
                        not divisible by # of processors (%d),
                        so exiting\n",
MX, THREADS );
            return( -1 );
        }
        if ( !MYTHREAD )
        {
            printf( "Solving heat conduction
                    task on %d by %d grid
                    by %d processors\n",
MX, MY, THREADS );
            fflush( stdout );
        }

        /* Initial settings: */
        upc_forall ( i = 0; i < MX; i++; &f[i][0] )
        {
            /* Local row pointers to the rows of the shared
               arrays that are local: */
            lf[i] = (double*)(&f[i][0]);
            ldf[i] = (double*)(&df[i][0]);

            /* Shadow (ghost) row pointers point
               to the local portions of the special */
            /* shared arrays. These arrays should
               not be shared, but they have to, because */
            /* they are used in upc_memcpy, and the latter
               requires the shared arrays: */
            if ( (i > 0) && (upc_threadof(&f[i-1][0]) ==
(MYTHREAD-1)) )

```

```

    {
        ib = i - 1;
        lf[ib] = (double*) (&prev[MY*MYTHR EAD]);
    }
    if ( (i < (MX-1)) &&
        (upc_threadof(&f[i+1][0]) ==
         (MYTHR EAD+1)) )
    {
        ie = i + 1;
        lf[ie] = (double*) (&next[MY*MYTHR EAD]);
    }
/* Initial conditions for the calculation: */
for ( j = 0; j < MY; j++ )
{
    f[i][j] = df[i][j] = 0.0;
    if      ( (i == 0)
              || (j == 0) ) f[i][j] = 1.0 ;
    else if ( (i == (MX-1))
              || (j == (MY-1)) ) f[i][j] = 0.5 ;
}
}
if ( !MYTHR EAD ) gettimeofday( &ts_st, NULL );
/* Iteration loop: */
for ( n = 0; n < NITER; n++ )
{
    if ( !MYTHR EAD )
    {
        if ( !(n%STEPITER) )
            printf( "Iteration %d\n", n );
    }
}
/* Step of calculation starts here: */
upc_barrier;
/* Get the shadow rows
from the neighbor processors, if any: */
if ( MYTHR EAD > 0 )
    upc_memcpy( prev+MY*MYTHR EAD, f[ib],
                MY*sizeof( f[0][0]) );
if ( MYTHR EAD < (THR EADS-1) )
    upc_memcpy( next+MY*MYTHR EAD, f[ie],
                MY*sizeof( f[0][0] ) );
/* Perform the calculation step: */
upc_forall ( i = 1; i < (MX-1);
            i++; &f[i][0] )

```

```

{
    for ( j = 1; j < (MY-1); j++ )
    {
        ldf[i][j] = ( lf[i][j+1] + lf[i][j-1] +
                     lf[i-1][j] + lf[i+1][j] )
                     * 0.25 - lf[i][j];
    }
}
upc_forall ( i = 1; i < (MX-1);
            i++; &f[i][0] )
{
    for ( j = 1; j < (MY-1); j++ )
    {
        lf[i][j] += ldf[i][j];
    }
}
}
if ( !MYTHREAD )
{
    gettimeofday( &ts_end, NULL );
    howlong = ts_end.tv_sec +
              (ts_end.tv_usec / 1000000.0);
    howlong -= ts_st.tv_sec +
              (ts_st.tv_usec / 1000000.0);
    printf( "Elapsed time: %f sec\n",
            howlong );
}
fp = upc_all_fopen( "progreve_upc_4.dat",
                   UPC_COMMON_FP | UPC_WRONLY |
                   UPC_CREATE | UPC_TRUNC,
                   0, &hint );

if ( !fp )
{
    printf( "Node %d failed to open a file\n",
            MYTHREAD );
    return( -1 );
}
upc_all_fwrite_shared( fp, &f[0][0],
                      MX*MY/THREADS, sizeof( f[0][0] ), MX*MY,
                      UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );
upc_all_fclose( fp );
return 0;
}

```

Этот вариант программы выполняется за 0,29 с. Отличие от варианта на базе MPI — в пределах погрешности измерения.

Итоговый вариант программы совсем не похож на первоначальную «наивную» версию, хотя очень похож на версию с использованием MPI или shmem. Очевидно, нам досталась вычислительная установка с достаточно слабой сетью, ведь шагов оптимизации пришлось сделать много. Однако мы на каждом шаге имели в своем распоряжении пригодную к отладке программу, которая предсказуемо реагировала на наши попытки ее улучшить. Параллельную реализацию «MPI-ного качества» мы получили, совершенно не вникая в коренное различие между функциями MPI\_Bsend() и MPI\_Alltoallv(), и так и не узнав, что такое тег и коммуникатор. Мы оперировали исключительно понятиями, вполне очевидными любому более или менее опытному в написании последовательных программ программисту.

Как всегда, в заключение — о доступных реализациях. Вполне работоспособной является свободно распространяемая в виде исходных текстов реализация Berkeley UPC.

### 5.5.2. Технология: Co-array Fortran

Еще одним примером технологии программирования в рамках модели PGAS является язык Co-array Fortran [22]. Термин «co-array», давший название языку и используемый в этом языке для обозначения распределенного по процессам массива, не имеет хорошего русского перевода. Дословно он переводится как «со-массив», по аналогии с такими словами, как «собрат» или «со-автор». Мы будем использовать для обозначения этого понятия термин «распределенный массив».

Язык является расширением Фортрана, точнее — Фортрана-95. Главное смысловое отличие от UPC — в способе адресации распределенных массивов.

Если массив A объявлен как локальный, то в каждом из процессов существует и доступен свой, отдельный экземпляр массива A. Если же он объявлен как распределенный, то в каждом из процессов по-прежнему существует свой, отдельный экземпляр массива A, но каждому из процессов доступны все такие экземпляры, а не только свой. Собственно, совокупность таких экземпляров объявленного специальным образом массива, по одному на процесс, и образует распределенный массив. Чтобы различать экземпляры, расположенные в разных процессах, номер процес-

са указывается в отдельных квадратных скобках как дополнительный индекс. Например,  $A(i, j)[K]$  — элемент двумерного массива, с индексами  $(i, j)$ , на процессоре  $K$ . Если индекс в квадратных скобках не указан, подразумевается свой, локальный экземпляр массива. Как отдельные элементы, так и вырезки из таких массивов (строки, столбцы, прямоугольные области) можно присваивать, перемещая тем самым данные между процессами. Имеются средства синхронизации.

Более подробно с языком можно ознакомиться в [22].

Обзор технологий модели PGAS на этом завершен.

## 5.6. Модель: SMP

В этом подразделе рассмотрим программистскую модель, ориентированную на SMP-системы. Память в таких системах вся общая и доступна всем процессам совершенно симметрично. Ничего похожего на понятие принадлежности элемента данных процессу, иными словами, на понятие «своих» и «чужих» данных, в этой модели нет. Следовательно, нет и средств описания принадлежности данных процессам — есть только средства синхронизации процессов и обеспечения атомарности. Легко видеть, что такая ситуация практически не отличается от случая, когда много процессов выполняются псевдопараллельно на одном процессоре. В этом случае также необходимо синхронизировать процессы и обеспечивать атомарность, но нет необходимости в каких-либо средствах описания локальности данных, тем более передачи этих данных из процесса в процесс.

Технологии программирования в терминах псевдопараллельных процессов интенсивно развивались в системном программировании на протяжении многих лет, и технологии эти гораздо старше, чем сами SMP-системы. Таким образом, мы впервые имеем дело с моделью, которая не обязана своим происхождением высокопроизводительным параллельным вычислениям.

Конкретные технологии программирования для SMP-систем, тем не менее, сильно отличаются от технологий, традиционно применявшихся в системном программировании. Причина этих отличий кроется в «лице» технологии, обращенном к программисту, а не к аппаратуре. Традиционные, старые технологии программирования в терминах параллельных процессов вполне адекватны оборудованию SMP-систем, но не очень удобны для

тех задач вычислительного характера, которые решаются на SMP-системах. Они скорее ориентированы на программы вроде драйверов внешних устройств или серверов баз данных и, что немаловажно, на программистов, разрабатывающих именно такие программы. Когда появилась необходимость дать аналогичные по функциям средства разработки программистам численных методов, над традиционными, старыми технологиями пришлось изготовить сравнительно высокоуровневую надстройку, более простую в использовании.

Прежде чем приступить к изучению этой надстройки, скажем несколько слов о базовом уровне. Нам придется изучить два вопроса:

- чем, с точки зрения системного программиста, отличается SMP-система от однопроцессорной машины, в которой ОС реализует псевдопараллельные процессы;

- как именно выглядят старые, традиционные средства программирования в терминах параллельных процессов системного уровня?

Ответ на *первый вопрос* можно дать одним словом: **ничем**. В современных ОС однопроцессорная машина рассматривается как частный случай SMP-системы, в которой число процессоров равно 1. ОС поддерживает внутри себя две очереди — очередь процессов, желающих потребить очередной квант времени, т.е. «немного повыполняться», и очередь процессоров, готовых этот квант предоставить, т.е. «немного повыполнять» какой-либо процесс. В общем случае,  $K$  процессоров используются для выполнения  $N$  процессов, и на соотношение значений  $K$  и  $N$  в данный момент времени не накладывается никаких ограничений. В однопроцессорной машине просто значение  $K$  всегда равно 1. Организованные таким образом ОС для многопроцессорных машин с общей памятью называются *системами единого образа*, или **SSI** (*Single System Image*). Название подчеркивает тот факт, что и сама ОС, и выполняемые под ее управлением программы существуют в единственном экземпляре в единой общей памяти. Не существует такого понятия, как «привязка» процесса к процессору. Технология SSI применяется также для построения ОС NUMA-систем, при условии, что вся память в системе является общей. Например, в NUMA-серверах Silicon Graphics до 2048 процессорных ядер управляются в режиме SSI [51]. Конечно, реализация режима SSI для NUMA-систем гораздо сложнее, чем для SMP-систем, поскольку понятие локальности данных в

NUMA-системах все же присутствует. Впрочем, вернемся к SMP-системам.

Теперь обсудим *второй вопрос*. Системные программисты, вынужденные использовать модели программирования в терминах параллельных процессов, традиционно решали две задачи.

Первая задача — по возможности, изолировать процессы друг от друга, т. е. организовать выполнение под управлением единой ОС независимых, самостоятельных процессов. Типичная ситуация, приводящая к такой постановке задачи — работа интерпретатора команд ОС, набираемых пользователем на клавиатуре. Среди таких команд, конечно, есть команды запуска на счет других программ. Выполняя эти команды, интерпретатор команд, сам являющийся процессом в рамках ОС, формирует и запускает другие процессы — запускаемые пользователем программы. По умолчанию, никакой связи между такими процессами нет, чем меньше каждый из этих процессов «знает» о существовании других, тем лучше.

Вторая задача, в некотором смысле, противоположна первой. В рамках единой программы — например, драйвера внешнего устройства — может потребоваться организовать тесное взаимодействие нескольких процессов, имеющих доступ к общим данным. Типичный пример — обработчик прерываний, генерируемых устройством при завершении выполняемых этим устройством обменов. В таких приложениях изоляция процессов не просто не требуется — она вредна. Сам смысл организации нескольких процессов в том и состоит, что доступные им данные — общие.

Наличие этих двух задач в системном программировании привело к появлению в современных ОС двух базовых механизмов: механизма *полновесных процессов* и механизма *легковесных процессов*, или *нитей*, часто называемых, по англоязычному оригиналу термина, *тредами*.

Полновесным процессом принято называть такой процесс в рамках ОС, у которого имеется свое собственное адресное пространство. В терминах главы, посвященной основным понятиям архитектуры процессоров и ОС, это означает, что при переключении на этот процесс активизируется своя, свойственная именно этому процессу, копия таблицы приписки виртуальной памяти. Если у каждого из двух полновесных процессов имеется слово памяти номер 100, то, по умолчанию, это — разные слова. Если полновесным процессам все же требуется разделять некоторые общие данные, то необходимое для этого частичное совмещение



адресных пространств достигается выполнением специальных, довольно сложных действий со стороны этих процессов.

Легковесный процесс, напротив, существует в адресном пространстве некоторого полновесного процесса, поскольку своего адресного пространства у него нет. Как правило, все треды одного полновесного процесса разделяют все данные программы, за исключением данных, находящихся в стеке, т. е. описанных внутри функции без квалификатора «static». Треды могут порождать новые треды, и все они останутся в рамках адресного пространства своего полновесного процесса. При переключении процессора с одного треда на другой в пределах полновесного процесса смены таблицы приписки не происходит.

В настоящее время общепринятым программистским интерфейсом для работы с тредами является стандарт `pthread` [25]. Именно над этой базой обычно надстраивается программистский интерфейс более высокого уровня — `Open MP`, к рассмотрению которого мы и переходим.

### 5.6.1. Технология: `Open MP`

`Open MP` [4,39] — это набор расширений традиционных языков программирования высокого уровня — `C` и Фортрана.

Расширение языка в обоих случаях выполняется способом, уже известным нам на примере `HPF` и `DVM`, т. е. дополнительные языковые конструкции имеют вид директив параллельного выполнения и оформлены как комментарии специального вида (для Фортрана) или директивы `#pragma` (для `C`). В обоих случаях программа является (точнее, может являться) правильной последовательной программой на базовом языке.

Сходство с `HPF` и `DVM` в форме записи не означает, что `Open MP` следует принятой в этих языках модели логически единого процесса. Напротив, `Open MP` является языком явного задания параллелизма, т. е. программа представляет собой набор параллельно выполняющихся ветвей, и ветвь может узнать собственный номер. Таким образом, `Open MP` — технология низкоуровневая. Транслятор не «распараллеливает» программу, а выполняет в точности и непосредственно то, что написал программист.

Проще всего рассказать об основных возможностях `Open MP` в терминах его отличий от `UPC`.

Как известно, программа на `UPC` представляет собой набор статически создаваемых процессов, каждый из которых имеет до-

ступ к локальным (своим для каждого процесса) и разделяемым (общим для всех процессов) данным. При этом разделяемым данным свойственна локальность — про каждый элемент таких данных можно точно сказать, какому из процессов он принадлежит.

В Open MP понятия локальности разделяемых данных нет, но это — не единственное отличие. Важное отличие состоит в том, что процессы в Open MP порождаются динамически, согласно схеме «*разветвление — слияние*» (fork/join). Схема эта устроена так.

Программа начинает выполняться как один процесс, называемый *исходным*.

Любой процесс в составе программы, в том числе исходный, может «разветвиться», т. е. породить несколько своих копий. Это происходит, если процесс выполняет один из многочисленных вариантов директивы «parallel». Процесс, выполнивший директиву «parallel», становится *ведущим* (master) процессом образовавшейся таким образом *команды* (team) процессов. Процессы в команде получают номера, которые они могут опросить. Пользуясь знанием собственного номера, а также некоторыми дополнительными директивами и встроенными функциями, процессы команды могут настроиться на совместное выполнение общей работы, подобно процессам в моделях параллельного программирования, которые мы рассматривали выше. Когда эта параллельная работа окончена, процессы команды выполняют (по директиве «end parallel», или по выходу из блока — тела директивы «parallel») «слияние» — действие, обратное «разветвлению». Слияние заключается в барьерной синхронизации, после которой все процессы, кроме ведущего, завершаются.

Участок программы от разветвления до слияния называется *параллельной областью*.

Некоторые реализации Open MP не поддерживают вложенных разветвлений, т. е. разветвиться может только исходный процесс.

Во всем остальном Open MP принципиально похож на UPC. Имеется барьерная синхронизация, средства обеспечения атомарности, параллельные циклы. Специальных средств параллельного ввода-вывода нет, поскольку отсутствие локальности данных исключает необходимость доступа к файловой системе из нескольких процессов.

Как и в случае UPC, данные, доступные процессам, могут быть локальными (private) и разделяемыми (shared). Локальные дан-

ные размножаются по процессам, разделяемые существуют в единственном экземпляре, доступном всем процессам.

Однако, в отличие от UPC, решение о том, какие именно переменные и массивы являются локальными, а какие — разделяемыми, принимается не статически, в момент написания программы, а динамически, в процессе ее выполнения.

При входе в каждую параллельную область решение о том, какие переменные и массивы будут размножены по создаваемым в этой области процессам, а какие останутся в единственном экземпляре, принимается заново.

Один и тот же массив может в одной параллельной области быть локальным, а в другой — разделяемым.

Доступ к разделяемым данным в современных версиях Open MP не предполагается когерентным. Физически это связано с наличием в процессорах SMP-системы многоуровневых кэш. Для обеспечения когерентности памяти в те моменты времени, когда она требуется, программист должен предусмотреть специальные действия, явно записываемые в программе.

Как и в случае UPC, записывать разделение работы между процессами внутри параллельной области можно двумя способами.

*Первый способ* заключается в снабжении циклов, витки которых должны быть распределены по процессам, специальными директивами. Цикл при этом превращается в специальную управляющую конструкцию — параллельный цикл. В терминах UPC это аналогично использованию специального параллельного цикла `upc_forall` вместо обычного цикла `for`. Достоинством этого способа является сходство текста параллельной программы с текстом ее последовательного прототипа. Написанная таким образом параллельная программа остается правильной последовательной программой, т.е., будучи оттранслирована транслятором, «не понимающим» Open MP, будет выполнять на одном процессоре те же самые вычисления, что и в многопроцессорном варианте.

*Второй способ* состоит в явном опросе процессом собственного номера из числа себе подобных и явном же разделении работы между процессами в соответствии с опрошенными значениями. Программа, написанная таким способом, уже не является правильной последовательной программой для транслятора, «не понимающего» Open MP. Достоинством этого способа является необходимость изучения программистом значительно меньшего

числа директив, более непосредственная форма записи разделения работы.

Ниже приводится пример написания нашей модельной программы на Open MP C обоими способами.

*Первый способ:*

```
#include <stdio.h>
/****/
#define MX 640
#define MY 480
#define NITER 10000
#define STEPITER 100
    static float  f[MX][MY];
    static float df[MX][MY];
/****/
int main( int argc, char **argv )
{
    int i, j, n, m;
    FILE *fp;
/****/
    printf( "Solving heat conduction task
            on %d by %d grid\n",
            MX, MY );
    fflush( stdout );
/* Initial conditions: */
    for ( i = 0; i < MX; i++ )
    {
        for ( j = 0; j < MY; j++ )
        {
            f[i][j] = df[i][j] = 0.0;
            if      ( (i == 0)
                    || (j == 0) ) f[i][j] = 1.0;
            else if ( (i == (MX-1))
                    || (j == (MY-1)) ) f[i][j] = 0.5;
        }
    }
/* Iteration loop: */
    for ( n = 0; n < NITER; n++ )
    {
        if ( !(n%STEPITER) )
            printf( "Iteration %d\n", n );
/* Step of calculation starts here: */
```

```

#pragma omp parallel for shared(f) private(i,j)
    for ( i = 1; i < (MX-1); i++ )
    {
        for ( j = 1; j < (MY-1); j++ )
        {
            df[i][j] = ( f[i][j+1] + f[i][j-1] +
                        f[i-1][j] + f[i+1][j] )
                        * 0.25 - f[i][j];
        }
    }
#pragma omp parallel for shared(f,df) private(i,j)
    for ( i = 1; i < (MX-1); i++ )
    {
        for ( j = 1; j < (MY-1); j++ )
        {
            f[i][j] += df[i][j];
        }
    }
}

/* Calculation is done, F array is a result: */
fp = fopen( "progreiv.dat", "w" );
for ( i = 1; i < (MX-1); i++ )
fwrite( f[i]+1, MY-2, sizeof(f[0][0]), fp );
fclose( fp );
return 0;
}

```

В этом примере каждая из параллельных областей включает в себя строго по одному циклу `for`, и потому вместо двух директив — директивы параллельной области и директивы параллельного цикла — для каждой параллельной области была использована одна «синтетическая» директива `parallel for`.

Легко видеть, что эти директивы были действительно добавлены к вполне корректному варианту последовательной, однопроцессорной программы, сам текст которой не претерпел никаких изменений.

*Второй способ:*

```

#include <stdio.h>
#include <omp.h>
/***/
#define MX 640

```

```

#define MY 480
#define NITER 10000
#define STEPITER 100
    static float  f[MX][MY];
    static float df[MX][MY];
/***/
    int main( int argc, char **argv )
    {
        int i, j, n, m;
        FILE *fp;
        int mt, nt, ns;
/***/
        printf( "Solving heat conduction
                task on %d by %d grid\n", MX, MY );
        fflush( stdout );
/* Initial conditions: */
        for ( i = 0; i < MX; i++ )
        {
            for ( j = 0; j < MY; j++ )
            {
                f[i][j] = df[i][j] = 0.0;
                if      ( i == 0 )
                    || ( j == 0 ) ) f[i][j] = 1.0;
                else if ( i == (MX-1))
                    || ( j == (MY-1)) ) f[i][j] = 0.5;
            }
        }
/* Iteration loop: */
        for ( n = 0; n < NITER; n++ )
        {
            if ( !(n%STEPITER) )
                printf( "Iteration %d\n", n );
/* Step of calculation starts here: */
#pragma omp parallel schedule(static)
                shared(f) private(i,j)
            {
                nt = omp_get_num_threads();
                mt = omp_get_thread_num();
                ns = (MX-2)/nt;
                for ( i = mt*ns+1; i <= (mt+1)*ns; i++ )
                {
                    for ( j = 1; j < (MY-1); j++ )
                    {

```

```

        df[i][j] = ( f[i][j+1] + f[i][j-1] +
                    f[i-1][j] + f[i+1][j] )
                    * 0.25 - f[i][j];
    }
}
}
#pragma omp parallel schedule(static)
shared(f,df) private(i,j)
{
    nt = omp_get_num_threads();
    mt = omp_get_thread_num();
    ns = (MX-2)/nt;
    for ( i = mt*ns+1; i <= (mt+1)*ns; i++ )
    {
        for ( j = 1; j < (MY-1); j++ )
        {
            f[i][j] += df[i][j];
        }
    }
}
}
/* Calculation is done, F array is a result: */
fp = fopen( "progrev.dat", "w" );
for ( i = 1; i < (MX-1); i++ )
fwrite( f[i]+1, MY-2, sizeof(f[0][0]), fp );
fclose( fp );
return 0;
}

```

В этом примере текст программы уже отличается от однопроцессорного варианта не только наличием директив Open MP, зато распределение работы между процессами в пределах параллельной области записано совершенно непосредственным и очевидным образом.

Open MP — классический пример технологии разработки, ориентированной на инкрементальное распараллеливание. Отсутствие необходимости делить данные между процессорами и прямая, непосредственная запись разделения работы делает язык идеальным для такого режима использования.

Именно в этом многие видят преимущество данной технологии перед MPI и, соответственно, преимущество SMP-машин, на которые эта технология ориентирована, перед MPP-системами.

## 5.7. Парадокс неприятия новых технологий

Обзор технологий параллельного программирования завершен, и пора приступать к их сравнительному анализу. Прежде чем это делать, хотелось бы обратить внимание читателя на «подводные камни», подстерегающие разработчиков технологий параллельного программирования. Среди таких «подводных камней» почетное (точнее — печальное) первое место занимает феномен, о котором мы поговорим в этом разделе.

Многолетний опыт развития технологий параллельного программирования, «надстроенных» над рандеву (грубо говоря, над MPI), показывает, что технологии эти крайне неохотно и редко используются программистами на практике, причем тем реже, чем выше уровень технологии [2,16]. Из технологий, рассмотренных в подразд. 5.3.2, более или менее широко распространены, пожалуй, лишь проблемно-ориентированные библиотеки коллективных операций. Почему же пользователи, часто и помногу жалуясь на трудности освоения параллельного программирования, упорно игнорируют попытки системных программистов помочь им, уменьшив эти трудности?

В подразделе, посвященном HPF, довольно подробно обсуждались недостатки этой технологии, мешающие ей претендовать на роль «моста» между мирами последовательного и параллельного программирования. В действительности, проблема имеет несколько более общих, системный характер.

Гипотеза о том, что соответствующие программные системы просто плохо отлажены или порождают неэффективный код, не подтверждается. Все серьезные системы, такие, как DVM или Норма, проверялись обратным переносом больших комплексов реальных, производственных программ, написанных первоначально с использованием MPI. Документация этих систем также весьма качественная.

Действительный источник парадокса в том, что системы эти, в действительности, зачастую решают не ту задачу, автоматизируют не то, что в действительности трудно начинающим параллельным программистам.

Разработчики систем частичной автоматизации параллельного программирования — профессиональные системные программисты. Кроме того, им, как и всем людям, свойственно судить о других по себе. Они полагают, что прикладному программисту трудно в процессе освоения параллельного программирования



то, что было трудно в аналогичной ситуации им самим, т. е. кропотливо выписывать конкретные детали организации двусторонних обменов. Освоить же новую систему понятий, элегантную и остроумную, которая от этой кропотливой работы освобождает, гораздо легче.

В отношении очень многих прикладных программистов это верно. Очень многие прикладные программисты, к счастью, «видят» компьютер именно так, как «системщики», т. е. постоянно, не стремясь к этому специально, держат в голове хорошо структурированную, формализованную модель вычислительной системы, программы и процесса ее построения, охотно и с интересом осваивают придуманные специально для них новые системы понятий. Трудность лишь в том, что эта категория пользователей не нуждается в помощи систем автоматизации. Эти пользователи легко осваивают MPI, а затем столь же легко строят на его базе собственные «системы автоматизации» для конкретной задачи или класса задач. «Норму» или DVM они не осваивают не потому, что не могут, а потому, что для них это лишняя работа.

На немыслимую трудность освоения параллельного программирования жалуются пользователи иного рода. Это — те программисты (зачастую, кстати сказать, весьма опытные), которые воспринимают вычислительную систему как досадную, но неизбежную помеху своей главной производственной деятельности (физике, химии, биологии). Примерно так, как «системщик» относится к «наворотам» новой версии Word или Excel, когда приходится оформлять документацию. Эти пользователи видят вычислительную систему нерасчлененно и целостно, на уровне привычки, совершенно справедливо полагая, что, раз уж без нее не обойтись, она должна просто считать побыстрее, и не засорять голову всякими программистскими моделями и прочей ерундой. Целостное, невербализованное, во многом основанное на привычке представление об однопроцессорной машине у них есть, причем когда-то давно оно, возможно, далось нелегко. Любое другое — крайне нежелательно, почти враждебно.

Как поступает такой пользователь, осваивая, например, DVM? Пролистав описание до первого примера программы, он вставляет в **некоторые** места своей программы **некоторое** количество директив распараллеливания, **похожих** на те, что были в примере. При этом, например, один и тот же массив может оказаться в вызывающей программе локальным, а в подпрограмме —

распределенным по процессорам. Полученная программа запускается, в надежде, что если **некоторое** количество директив в нее вставлено, то **некоторое** ускорение, по сравнению с однопроцессорным вариантом, должно получиться. Получив вместо этого замедление в 10 000 раз, пользователь бывает искренне возмущен. Замечание, что написанная лишь частично программа не способна частично ускориться, поскольку элементарно не-правильна, отвергается как «демагогия» и «философствование». Указанная психологическая ловушка захлопывается тем скорее и надежнее, чем выше степень автоматизации, предлагаемая используемой технологией.

Для такого пользователя, в действительности, не важно, какой технологией пользоваться. При наличии достаточно сильной мотивации он может освоить любую, но двигаться будет строго индуктивно, отвергая всякую «простую и изящную», с точки зрения «системщика», общую логику, строя на конкретных примерах нерасчлененную и невербализованную, на уровне привычки, модель новой для него сущности — параллельной вычислительной системы. Скорее всего, цель будет достигнута ручным применением технологии МРІ, поскольку она элементарно проще сама по себе. Научиться копать лопатой проще, чем освоить управление экскаватором, хотя производительность рытья траншеи при использовании экскаватора неизмеримо выше.

Означает ли это, что технологии такого рода лишены права на существование, по крайней мере — с точки зрения реальной программистской практики? Ни в коем случае. Оборудование усложняется, становится все более разнородным, и к тому моменту, когда выписывание обменов вручную с учетом этой сложности и разнородности станет совершенно невыносимым для программиста, высокоуровневые технологии с неизбежностью будут востребованы. Произойдет это гораздо раньше, чем иногда кажется, поскольку оборудование развивается стремительно.

## **5.8. Сравнительный анализ моделей и технологий параллельного программирования**

Соображения, изложенные в предыдущем разделе, можно считать попыткой сравнительного анализа программистских моделей, производных от модели двустороннего обмена сообщениями. Мы знаем еще три базовых модели, ориентированных на

три других класса коммуникационного оборудования. Производных, высокоуровневых моделей мы ни для одного из этих классов оборудования не построили. Модель односторонних обменов сообщениями, модель PGAS и модель SMP представлены в нашем обзоре базовыми, низкоуровневыми технологиями. Попробуем сравнить эти модели между собой. Не сделав этого, мы вряд ли сможем в полной мере понять, почему некоторые пользователи приобретают вместо вычислительных кластеров дорогостоящие SMP- или NUMA-системы, переплачивая за узел сравнимой производительности в разы, а то и в десятки раз.

Вначале разработаем критерии сравнения:

1) *удобство программирования*. Рабочее время программиста, как и вычислительное оборудование, стоит денег, поэтому к данному критерию не следует относиться как к чему-то малосущественному и второстепенному;

2) *принципиальная пригодность для реализации типовых численных методов*. Вполне может оказаться, что некоторые виды распределения работы между процессорами просто не на всех классах коммуникационного оборудования осуществимы в принципе;

3) *пригодность для реализации на коммуникационном оборудовании других классов* (отличных от того класса, на который ориентирована модель). Это свойство технологий важно хотя бы с точки зрения переносимости программ.

Вводя в подразд. 1.3 классы коммуникационного оборудования, мы упорядочили их по тесноте связей между процессорами в системе. Этот линейный порядок дает нам ключ к сравнению классов оборудования и ориентированных на них моделей по всем трем критериям. В самом деле, все, что можно делать на более «слабом» классе, можно с тем же (и даже большим) успехом делать на более «сильном». Обратное неверно.

Например, на самом «слабом» классе коммуникационного оборудования — сети двусторонних обменов — естественно пользоваться соответствующей технологией, скорее всего, MPI. Можно ли реализовать эту же технологию на более «сильном» оборудовании, например, на самом «сильном» — на SMP-системе? Без сомнения, можно. SMP-систему можно использовать как «очень хорошую сеть двусторонних обменов», реализовав на ней MPI. Что будет с программой, перенесенной на такую систему с кластера на базе Gigabit Ethernet? Она будет работать гораздо лучше, ведь значения всех показателей эффективности обмена

сообщениями при таком переносе резко улучшатся (правда, могут возникнуть проблемы конкуренции за общую память, но этот вопрос здесь не рассматривается). Однако осуществив такой перенос, мы далеко не лучшим образом используем как оборудование, так и рабочее время прикладного программиста. С одной стороны, вполне может оказаться, что прямая адресация данных в рамках всей системы могла бы позволить написать гораздо более эффективную программу, чем с использованием MPI. С другой стороны, почти наверняка эта программа была бы и в разработке гораздо проще.

Теперь попробуем поступить наоборот. Напишем программу на Open MP, после чего реализуем Open MP программно на сети двусторонних обменов, и перенесем нашу программу с настоящей SMP-системы на программно реализованную. Вообще говоря, программа станет работать гораздо хуже — накладные расходы на выполнение синхронизации тредов, внутри которой теперь «спрятаны» пересылки данных между процессорами, возрастут во много раз. Из соображений симметрии следовало бы ожидать, что мы, тем не менее, достигли исключительно высокого качества использования как оборудования, так и рабочего времени прикладного программиста. К сожалению, это не совсем верно. Оборудование почти наверняка выполняет несколько больше (или гораздо больше) обменов данными или какой-то другой работы, чем это было бы необходимо при качественной реализации алгоритма с использованием MPI. Программист, действительно, сэкономил массу усилий, написав программу с использованием не MPI, а Open MP, но при этом вовсе не очевидно, что положительный результат вообще получен. Дело в том, что, экономя время и силы по сравнению с использованием MPI, программист почти наверняка использовал, сам того не осознавая, именно те преимущества «сильного» коммуникационного оборудования, которые в данном случае реализованы программно. Вероятность очень не эффективной реализации весьма и весьма высока. Программа может вообще не ускориться или даже замедлиться по сравнению с однопроцессорным вариантом.

Те же закономерности будут наблюдаться в отношении любой пары классов коммуникационного оборудования и соответствующей пары технологий. Чем ближе расположены соответствующие классы оборудования в спектре, тем меньше эти закономерности выражены количественно. Например, при реализации MPI на сети односторонних обменов улучшение параметров производи-

тельности будет не таким резким, как при его реализации на SMP-системе. С другой стороны, при реализации, например, технологии Global Arrays на не вполне свойственной ей сети двусторонних обменов и рост накладных расходов будет не таким значительным, как при попытке реализовать на такой сети Open MP.

Таким образом, модели программирования, которые мы до сих пор считали базовыми для соответствующих классов коммуникационного оборудования, могут также рассматриваться как производные, т. е. надстроенные в целях повышения удобства программирования, для более «слабых» классов. При этом чем больше «ступенек» по спектру мы пытаемся «преодолеть в один прыжок», тем более дает о себе знать недостаточная адекватность модели оборудованию.

Чтобы наполнить конкретикой эти общие рассуждения, следует ответить на два вопроса:

- насколько использование «чужих» базовых моделей в качестве производных встречается в реальной практике;
- в каких именно программах и численных методах преимущества более «сильных» моделей и технологий особенно заметны?

Реализации MPI известны для всех классов коммуникационного оборудования, реализации некоторых технологий односторонних обменов также. Например, технология Global Arrays [19] вообще сначала возникла как производная для «двусторонних» сетей, и лишь затем был выполнен перенос на сети односторонних обменов, более отвечающие этой технологии.

Реализации Open MP долгое время существовали только для систем с общей памятью, преимущественно — для SMP-систем, но в 2006 г. фирма Intel выпустила продукт под названием Cluster Open MP [7] — реализацию Open MP на базе сетей односторонних и двусторонних обменов.

Интересная ситуация сложилась с моделью PGAS. Свободно распространяемая в виде исходных текстов реализация UPC из Berkeley поддерживает все классы коммуникационного оборудования — кроме NUMA. Это, мягко говоря, не может не изумлять. Чтобы осмыслить этот парадокс, попробуем начать отвечать на наш второй вопрос. Попробуем сжато сформулировать, что именно получает программист на каждом шаге, «поднимаясь» по спектру базовых моделей.

При объяснении, что такое параллельная программа, обычно приводят очень простые во всех отношениях примеры. В отно-

шении организации обменов данными простота заключается в том, что пересылаемые данные расположены в памяти крупными кусками, не перемещающимися во времени. Наши две модельные программы в этом смысле — не исключение. Такие параллельные программы довольно легко пишутся в любой модели, и довольно эффективно выполняются на оборудовании любого класса. Для того чтобы ощутить преимущества более «сильных» программистских моделей, надо поставить себя на место программиста, пишущего другие программы.

Сегодня принято считать, что важнейшим классом прикладных программ такого рода являются программы расчетов на неструктурных сетках. При дискретизации расчетной области в численных методах сетки, покрывающие область, вовсе не всегда бывают четырехугольными — очень популярны, например, треугольные сетки. Хранение такой сетки в оперативной памяти — нетривиальная задача. В самом деле, в случае четырехугольной сетки поле величин хранится в двумерном массиве, соседство ячеек выражается в терминах индексов. Например, «верхняя» граница некоторой области — это всегда нулевая строка двумерного массива. «Правый сосед» ячейки — это элемент массива со вторым индексом, на единицу большим, чем у данной ячейки, и т. п. А как быть, если сетка треугольная? В этом случае значения величин в ячейках хранятся в одномерном массиве, значение индекса в котором никак не отражает соседства ячеек. Информация о соседстве хранится в других массивах, тоже одномерных. Если мы захотим выполнить для такого представления сетки нечто подобное параллельной реализации метода Якоби, мы обнаружим, что «буферная зона на границе двух процессоров», которая раньше была строкой двумерного массива, теперь превратилась в совокупность отдельных чисел, весьма причудливо разбросанных по памяти обоих процессоров. Числа эти было бы здорово передать в соседний процессор (или принять из него) по одному, да еще так, чтобы не оба процессора, а лишь один — инициатор обмена — мучительно вычислял, откуда именно что взять и куда именно положить. Задача — в точности для аппаратуры, поддерживающей односторонние обмены. Потом, когда серия обменов закончится, можно и синхронизироваться отдельным запросом из всех взаимодействующих процессоров. Еще лучше такие численные методы реализуются на NUMA-оборудовании.

Легко видеть, что в этом случае использование аппаратной возможности односторонних обменов (или, в случае NUMA, одно-

стороннего доступа) одновременно и повышает быстродействие программы, и упрощает ее написание.

Аналогично обстоит дело при использовании адаптивных сеток, когда структура данных, в которой хранится сетка, не только сложно и «неудобно» устроена, но и меняется в ходе расчета.

Как сложность написания программ такого рода, так и их алгоритмическая специфика часто заставляют их авторов задуматься об инкрементальном распараллеливании. Мы уже знаем, что идеальным инструментом для инкрементального распараллеливания является Open MP, что UPC в принципе пригоден для этого способа разработки программ, а MPI и его производные — практически не пригодны. В действительности легко убедиться, что для осуществимости инкрементального распараллеливания критичны две вещи: односторонняя дисциплина доступа к «чужим» данным и возможно более низкая латентность, чтобы большое число коротких обменов не уничтожило эффективность полностью. Однако, этими двумя свойствами обладают не только NUMA-системы, на которые, казалось бы, ориентирован UPC, но и MPP-системы на базе сетей односторонних обменов. Можно утверждать, что эти два класса оборудования отличаются лишь количественно (уровнем латентности и временем синхронизации), но сходны по стилю использования. Различия в форме записи, как мы видим на примере UPC, могут быть «прикрыты» несложным транслятором. Интересно отметить, что в суперкомпьютерах Cray, являющихся NUMA-системами, низкоуровневой базой для разработки коммуникационных библиотек и трансляторов является библиотека shmem. Это означает, что NUMA-оборудование используется как «очень хорошая сеть односторонних обменов», даже при реализации, например, UPC, который, казалось бы, следовало отображать на NUMA-оборудование непосредственно. Вопрос о том, действительно ли оборудование NUMA, грубо говоря, ни на что больше не годится, т.е. отличается от сети односторонних обменов лишь количественно, и не порождает собственного стиля написания программ, является, на взгляд автора, открытой технологической проблемой в разработке систем параллельного программирования. Одного опыта фирмы Cray для уверенного положительного ответа на этот вопрос недостаточно. Впрочем, даже положительный ответ на этот вопрос не означает, что NUMA-оборудование не имеет права на существование. Как раз наоборот. Количественный выигрыш в латентности и времени синхронизации в данном случае

имеет принципиальное значение, и может оказаться, что именно NUMA позволяет снизить эти времена так сильно, как никакая другая реализация сети односторонних обменов.

Окончательный ответ на этот вопрос может дать только программистская практика, и она, скорее всего, даст его в самое ближайшее время. Как будет показано в части II курса, проявившиеся в последние два-три года тенденции развития средств коммуникации обещают сделать системы NUMA такими же дешевыми и доступными, как системы на базе сетей односторонних обменов. Разработка систем программирования для этого оборудования, таким образом, выйдет за рамки «экзотики», практикуемой лишь в сообществе пользователей Стау, и станет достоянием всего мирового суперкомпьютерного сообщества. Тогда мы и узнаем ответ на наш вопрос.

Пока же можно довольно уверенно констатировать следующее.

Длительное развитие преимущественно двух крайних (по нашему спектру) классов коммуникационного оборудования — SMP и сетей двусторонних обменов — сформировало в головах у пользователей суперкомпьютеров ряд широко распространенных явных и неявных убеждений, которые сегодня уже не соответствуют действительности.

Важнейшим из таких устаревших убеждений является представление о том, что единственной реальной альтернативой использования MPI при работе на кластере является переход с кластера на SMP-систему. В действительности использование, например, UPC предоставляет программисту кластера практически все те преимущества в разработке программы, которые раньше принято было ассоциировать исключительно с SMP-системами, и которые мы кратко охарактеризовали выше. При этом реализации UPC для специализированных «кластерных» сетей являются вполне удовлетворительными по эффективности. По мере распространения и удешевления оборудования NUMA, которое, фактически, уже началось, технологии модели PGAS, скорее всего, выйдут на первый план по распространенности, значительно «потеснив» MPI.

С другой стороны, реальная применимость такой системы, как Cluster Open MP, претендующей на практически полное «стирание граней» между классами коммуникационного оборудования, вызывает сомнения у многих специалистов, и нуждается в серьезной проверке на практике. Впрочем, с развитием ап-



паратуры NUMA и на этом направлении можно прогнозировать значительные продвижения.

В качестве примера рассмотрим реально проводившееся автором изучение масштабируемости нашей модельной программы на Cluster Open MP. Ранее (см. подразд. 3.1.2) отмечалось, что наша модельная программа может как угодно хорошо ускоряться на как угодно плохой сети, если только проводить расчет на достаточно большой по размеру сетке. Минимальный размер сетки, на котором достигаются заранее заданные величины параллельного ускорения, можно, таким образом, рассматривать как интегральную «меру качества» используемой коммуникационной технологии. Чем лучше коммуникационная технология, тем на меньшей по размеру сетке, при прочих равных, удастся достичь заданного параллельного ускорения. Фактически, минимальный допустимый в данном расчете размер сетки предлагается использовать в качестве численной меры зерна параллелизма.

Рассматриваемая ниже серия измерений проводилась на вычислительной установке, оснащенной двухпроцессорными узлами на базе Opteron с рабочей частотой 2,4 ГГц, и сетью Myrinet. Ставилась задача грубо оценить минимальный размер сетки, для которого в диапазоне числа процессоров от 1 до 16 наращивание числа процессоров будет приводить к монотонному падению времени счета. При этом сравнивались две коммуникационные технологии: MPI для Myrinet и Cluster Open MP. Цель исследования — выяснить, насколько Cluster Open MP хуже, чем «эталонный» для данного оборудования вариант — MPI для Myrinet.

Сначала изучим вариант на базе MPI. Программа написана с использованием MPI\_Alltoallv(), выполняет 5 000 итераций. Для сетки размером 1 000 × 1 000 времена выполнения расчета получились следующие:

Число процессоров .....	1	2	4	8	16
Время, с .....	51	30	15	5,8	3,4

Пока задача демонстрирует вполне приемлемые показатели параллельного ускорения. Уменьшим сетку вдвое, до 500 × 500, и получим:

Число процессоров .....	1	2	4	8	16
Время, с .....	14	6,1	2,1	1,6	1,9

Похоже нам удалось «поймать» момент начала деградации масштабирования. На восьми процессорах ситуация не намного лучше, чем на четырех, а на шестнадцати — хуже, чем на восьми.

У нас есть все основания считать, что технология MPI на данной вычислительной установке позволяет выполнять наш модельный расчет на 16 процессорах для сеток размером  $500 \times 500$  или больше, но не меньше.

Прежде чем переходить к сравнению полученного результата с результатом для OpenMP, исследуем и сравним базовые уровни коммуникационной поддержки обеих систем.

Использованная версия MPI является специализированной версией для Myrinet, т.е. опирается на очень эффективную низкоуровневую библиотеку доступа к оборудованию.

Использованный вариант Cluster Open MP опирается на tcp/ip. Вариант на базе библиотеки односторонних обменов dapl запустить не удалось: этот вариант «соглашается» работать только на процессорах производства Intel, а такого кластера у автора в распоряжении не оказалось. Следовательно, для обеспечения наилучшей производительности имеющейся реализации Cluster Open MP ее необходимо запускать с использованием tcp/ip на базе Myrinet.

Сравнивая специализированную версию MPI для Myrinet и tcp/ip на базе Myrinet, легко увидеть, что производительность (скорость передачи данных) в обеих технологиях совпадают, но латентности отличаются в три раза. Для MPI латентность двустороннего обмена составляет 7 мкс, для tcp/ip на базе Myrinet — примерно 20.

Обе технологии используют Myrinet как сеть двусторонних обменов, т.е. игнорируют «односторонние» возможности оборудования.

Таким образом, перед нами в совершенно чистом виде стоит задача оценить, насколько именно применение технологии Open MP «испортит» сеть двустороннего обмена сообщениями. Однако по изложенным только что техническим причинам «соревнование» является не вполне честным — MPI имеет «фору» в виде втрое меньшей латентности. Зная тестовое приложение, можно легко внести в уже имеющиеся цифры для MPI необходимую поправку.

В самом деле, увеличим мысленно латентность MPI втрое, чтобы уравнивать условия. Это будет означать, что каждый обмен в действительности протекал на 14 мкс дольше, чем в наших измерениях. Для случая двух процессоров таких обменов на каждой из 5 000 итераций было два, для случая большего числа процессоров — 4. Следовательно, время счета на двух процессорах следует увеличить на  $14 \cdot 2 \cdot 5000 = 140\,000$  мкс, или 0,14 с, время же счета на большем числе процессоров — на 0,28 с. Для сетки

500 × 500 времена выполнения варианта MPI станут выглядеть примерно так:

Число процессоров .....	1	2	4	8	16
Время, с .....	14	6,3	2,4	1,9	2,2

Качественно картина не изменилась, только несколько ухудшился коэффициент распараллеливания.

Теперь попробуем выполнить тот же самый расчет с использованием Cluster Open MP. Для сетки 500 × 500 получим:

Число процессоров .....	1	2	4	8	16
Время, с .....	18	19	21	21	21

Ничего похожего на параллельное ускорение мы не наблюдаем, программа находится далеко за пределом деградации масштабирования. Удвоим сетку, и на сетке 1 000 × 1 000 получим:

Число процессоров .....	1	2	4	8	16
Время, с .....	49	39	51	42	28

Сетка все еще мала, удвоим ее еще раз. Для сетки 2 000 × 2 000 получим:

Число процессоров .....	1	2	4	8	16
Время, с .....	178	114	125	101	60

Очевидно, мы находимся на пороге деградации масштабирования. Для контроля удвоим размер сетки еще раз, до 4 000 × 4 000, и получим:

Число процессоров .....	1	2	4	8	16
Время, с .....	696	413	325	235	167

Мы получили хотя и довольно «пологое», но монотонное снижение времени счета с ростом числа процессоров. Для этого нам пришлось увеличить размер сетки в четыре раза по сравнению со случаем использования MPI.

Выводы из приведенной серии измерений следующие.

В данном подразделе нам уже приходилось говорить о том, что реализация программистской технологии, ориентированной на «сильный» класс оборудования, на более «слабом» классе рискует быть неэффективной. При этом потеря эффективности тем больше, чем больше «расстояние» между «сильным» и «слабым» классом по спектру классов оборудования. Приведенный пример это подтверждает. Преодолев максимально возможное «расстояние» по спектру классов оборудования, т. е. реализовав

самую «сильную» технологию на самом «слабом» классе, мы радикально упростили работу программиста, но лишили его возможности решать те задачи, с которыми данное оборудование прекрасно справилось бы, если бы программист использовал MPI.

Тем не менее, реальная работоспособность программы на Open MP была достигнута. Учитывая, что в реальных задачах соотношение объема вычислительной и обменной работы часто бывает гораздо более выгодным (счета на один обмен приходится больше), данную технологию можно уверенно рекомендовать для практического использования наряду с MPI (но не вместо MPI!).

В заключение обзора — несколько слов о программистских моделях *гибридного параллелизма*. Большинство современных кластеров представляют собой системы с иерархическим строением коммуникационной среды. В самом деле, рассмотрим кластер из двухпроцессорных рабочих станций на базе двухядерных процессоров Opteron, в котором рабочие станции связаны сетью Infiniband. В этом вполне реалистичном случае мы видим трехуровневую иерархию. Каждый из процессоров представляет собой SMP-систему, каждая рабочая станция — NUMA-систему из двух SMP-систем, а весь кластер в целом — это система на базе сети односторонних обменов, объединяющей NUMA-системы. Часто такие системы используют «по минимуму коммуникационной оснащенности», т. е. рассматривают как однородные образования, в котором каждое процессорное ядро обладает собственной памятью, а общение между ядрами происходит через сеть. Однако с ростом числа ядер на кристалле игнорирование факта наличия общей памяти, к которой эти ядра имеют доступ, становится все менее и менее желательным. В данном случае следует учесть сравнительно слабую разнородность доступа к памяти, свойственную рабочим станциям на базе процессоров Opteron. Это позволяет рассматривать каждый узел сети, с высокой степенью точности, как 4-процессорную SMP-систему. Число уровней иерархии, таким образом, сокращается до двух. Программист, желающий явно учесть в программе факт наличия общей памяти у каждой из четверок процессоров, скорее всего, воспользуется гибридной моделью Open MP+MPI. Эта модель на сегодня наиболее популярна, хотя есть и другие варианты, например, Open MP+DVM [27]. Некоторые исследователи систем параллельного программирования считают использование языков модели PGAS альтернативой гибридным моделям. Много обзоров на эту тему можно найти в Интернет по ключевым словам.

# СУПЕРКОМПЬЮТЕР МВС-1000 И МЕТАКЛАСТЕР МВС-900

В данной главе рассмотрим проблемы, возникающие при управлении ресурсами вычислительного кластера, на примере системы управления прохождением задач для машин МВС-1000. Программное обеспечение МВС-1000 — совместная разработка ИПМ им. М. В. Келдыша РАН, Института математики и механики Уральского отделения РАН, НИИ «Квант» и ряда других организаций РАН и промышленности. Именно это программное обеспечение используется на машинах Суперкомпьютерного центра РАН, открытых для коллективного использования через Интернет, со дня основания Центра. Кроме того, это программное обеспечение успешно работает на нескольких десятках вычислительных кластеров разного размера во многих организациях науки и промышленности [2].

## 6.1. Основные принципы организации МВС-1000

МВС-1000 — это Linux-кластер, определенным образом сконфигурированный [2,15]. Тип процессора и сети могут быть разными.

Общая структура МВС-1000:

- сервер доступа, он же инструментальный сервер;
- управляющая машина;
- вычислительное поле, состоящее из узлов;
- файловый сервер;
- три сети.

**Сервер доступа** — это машина, используемая для разработки программ. На эту машину пользователь «заходит» сеансом, копирует на нее свои файлы и т. п. На этой же машине он выполняет команды запуска параллельных программ.

Саму работу по подготовке узлов к запуску на них программ, учет числа занятых и свободных узлов, и прочие административные функции выполняет *управляющая машина*. Если она не

совпадает с сервером доступа, попасть на нее сеансом пользователь не может.

**Файловый сервер** предоставляет серверу доступа, управляющей машине и **вычислительным узлам** общую файловую систему, в которой располагаются директории пользователей, а также некоторые общие для всех директории, доступные пользователям для чтения. На файловый сервер, если он не совпадает с сервером доступа, пользователи также не «заходят».

Как запуск на счет параллельных программ и последующее управление ими, так и предоставление всем единой файловой системы, осуществляется в рамках технологии «клиент — сервер». Поэтому сервер доступа, управляющая машина и файловый сервер — это функции, а не реальные компьютеры. Распределение этих функций по реальным компьютерам может быть разным. В небольших кластерах часто все три функции возлагаются на один компьютер, для краткости называемый **хостом**. В больших кластерах файловый сервер может быть распределенным, управляющая машина — это отдельный компьютер, а серверов доступа может быть даже несколько (каждый — отдельный компьютер).

Три сети — это:

- **сеть управления**, с помощью которой управляющая машина передает узлам команды на запуск ветвей параллельной программы, а также другие управляющие воздействия;

- **сеть ввода-вывода** позволяет всем выделенным машинам и узлам общаться с файловым сервером;

- **сеть коммуникаций**, которую ветви параллельной программы используют при работе коммуникационной библиотеки.

Это также функции, распределение которых по конкретным сетям может быть разным. Это могут быть действительно три сети, или две (сеть коммуникаций выделена), или даже в небольших кластерах всего одна сеть (рис. 6.1).

Сеть ввода-вывода охватывает все машины, сеть управления — управляющую машину и узлы, сеть коммуникаций — только узлы. Выход в Интернет имеет только сервер доступа и, возможно, файловый сервер (не показано).

Само название «МВС-1000» относится прежде всего к **системе управления ресурсами**, функционирующей на управляющей машине. В первую очередь речь идет о процессорном ресурсе.

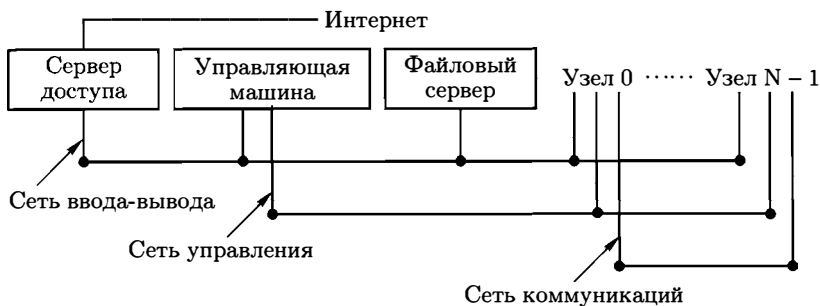


Рис. 6.1. Функциональная структура МВС-1000 (сеть ввода — вывода охватывает все машины, сеть управления — управляющую машину и узлы, сеть коммуникаций — только узлы. Выход в Интернет имеют только сервер доступа и, возможно, файловый сервер (не показано))

Системы параллельного программирования (обычно на базе MPI) сами по себе ресурсами вычислительной установки не управляют. При запуске параллельной программы в такой системе пользователь сообщает точный список узлов, на которых он бы хотел программу запустить. Сам процесс «разбрасывания» ветвей параллельной программы по узлам происходит с помощью команды `rsh` или `ssh`, соответственно, предполагается, что пользователь авторизован на узлах для выполнения этой команды без запроса пароля. Процесс прекращения работы программы вообще практически не поддерживается — всегда есть вероятность, что на узлах после аварийного (или просто принудительного) завершения программы останутся «обломки» — процессы, потребляющие память и процессорное время.

Довольно очевидно, что работать так можно (хотя и не без трудностей) лишь на установке, монопольно занятой одним пользователем. Попытка использовать при таких условиях систему в режиме коллективного пользования и удаленного доступа — это гарантированный кошмар. Разные параллельные программы будут пересекаться по используемым узлам, узлы будут постепенно «деградировать» за счет оставшихся на них «обломков» от прошлых запусков и т.п. Доступные системы управления ресурсами, вроде Open PBS [4], решают проблему лишь отчасти: они позволяют вести учет свободных и занятых узлов. Но ведь надо еще проследить, чтобы эта «бухгалтерия» соответствовала действительности. В самом деле, если пользователь всегда авторизован на узле для беспарольного входа, ничто не мешает ему запросить у PBS три узла на час, а занять — 100 узлов на сутки

(при этом про оставшиеся 97 узлов PBS будет «думать», что они свободны). По крайней мере, в то время, когда принималось решение о разработке системы управления ресурсами МВС-1000, среди свободно распространяемых систем управления ресурсами кластеров не удалось найти ни одной, свободной от этого недостатка.

Система управления ресурсами МВС-1000 была задумана для преодоления трудностей именно такого рода, а в конечном счете — для обеспечения устойчивой эксплуатации кластера в режиме коллективного пользования при удаленном доступе (т.е. без каких-либо «договоренностей», кто и когда «возьмет» те или иные узлы).

**Функциональные принципы** управления процессорным ресурсом таковы:

- *единая точка входа* — пользователь «видит» при сетевом доступе только сервер доступа;

- *императивность правил поведения пользователя и его программ* — все, что пользователю делать нельзя, запрещено технически, а не дисциплинарно. Пользователю нельзя попасть сеансом на узел, не занятый его задачей, — значит, он туда попасть **не сможет**. Программе пользователя можно, попав на узел, порождать другие процессы произвольным образом — значит, система от этого **не «сломается»**, как бы причудливо программа это ни делала;

- *выделение пользователю запрошенного им под задачу числа узлов на запрошенное им время в монопольное распоряжение* — любой вычислительный узел может быть в данный момент времени либо занят ровно одной задачей, либо свободен, либо заблокирован как негодный. Пересечение задач по используемым узлам не допускается. Если в момент выставления запроса требуемого числа свободных узлов нет, запрос на запуск задачи будет поставлен в очередь. Если в момент завершения задачи (пользователем, администратором или автоматически, по исчерпанию времени) системе управления не удалось уничтожить на узле все процессы данного пользователя и убедиться, что на узле их действительно не осталось, узел не будет считаться свободным (будет заблокирован как негодный).

**Реализационные принципы**, позволяющие эту функциональность обеспечить, весьма просты:

- *пассивность узла относительно процесса управления ресурсами* — вся работа по управлению процессорным ресурсом



сосредоточена в управляющей машине. На узле нет никаких специальных активных компонентов — серверов, демонов или модулей ядра — которые были бы специально разработаны для взаимодействия с управляющей машиной или для специфического отслеживания действий программы пользователя. Все управление осуществляется командами `rsh` или `ssh`, которые выдаются от имени суперпользователя `root` с управляющей машины;

- *выборочное разрешение доступа пользователей* только на те узлы, которые им выделены, с последующим запретом при освобождении узла;

- *контроль успешности освобождения узла* по отсутствию на нем процессов того пользователя, который этот узел ранее занимал, а не конкретно тех процессов, которые запустила система выполнения параллельных программ;

- *блокировка узла как негодного* при любой невозможности для системы запуска убедиться, что ее действия привели к ожидаемому результату, с сохранением целостности системы и степени пригодности к использованию всех прочих узлов.

Следует отметить, что сама система планирования очереди на включение задач в счет — компонент, часто ошибочно отождествляемый с системой управления в целом — также является весьма сложным программным изделием. Так, алгоритмы планирования гарантируют, при неизменности состава системы и наличии в ней в принципе необходимого для удовлетворения запроса числа узлов, конечность времени ожидания запроса в очереди.

Система управления процессорным ресурсом прозрачна относительно конкретной используемой системы параллельного программирования. Она не настроена жестко на какую-либо конкретную реализацию MPI (и вообще именно на MPI). Все известные и доступные в настоящее время разработчикам МВС-1000 системы параллельного программирования придерживаются единой схемы начальной раскрутки — с помощью беспарольного доступа на узлы командами `rsh` или `ssh`. Именно этот механизм и контролируется. Для настройки на конкретную систему параллельного программирования необходимо написать несколько несложных скриптов.

В качестве дополнительной возможности в систему управления может включаться управление еще одним ресурсом — **локальной дисковой памятью** (ЛДП). Мы уже отмечали, что общая файловая система кластера — одно из важнейших «узких мест» в системе. Альтернативой могло бы быть использование

локальных дисков вычислительных узлов, но для этого необходимо обеспечить возможность повторного попадания задачи, использующей локальные диски, именно на те узлы, на которых она считалась в прошлый раз. Также необходимо правильно настроить квотирование пространства на этих дисках, внести изменения в алгоритмы планирования очереди и т. п. Система управления ЛДП все это умеет: она может как включаться, так и не включаться в систему управления конкретным кластером и, конечно, ее действие распространяется только на те задачи, которые явно запрашивают ее услуги.

## **6.2. Как работает и зачем нужен метакластер MBC-900**

Важнейшая проблема практического применения вычислительных кластеров — проблема первоначального освоения технологии [2,16]. Ее часто ошибочно отождествляют с проблемой обучения параллельному программированию. В действительности практическое освоение вычислительного кластера, внедрение его в реально существующую производственную обстановку — сложнейшая комплексная инженерно-психологическая задача. Ее происхождение примерно такое же, как и у упоминавшегося феномена неприятия новых технологий, но задача эта комплексная и даже более сложная, чем внедрение новой системы параллельного программирования. Типичный новый пользователь вычислительного кластера — вовсе не программист-профессионал, а специалист по инженерным расчетам в своей предметной области. Даже применение традиционного программирования его тяготит, и не является, с его точки зрения, содержательной деятельностью, к которой следует стремиться, которую следует осваивать и изучать. Компьютер — неизбежное зло. К нему следует привыкнуть и приспособиться, если уж без этого никак не обойтись. Да, было бы хорошо, чтобы он считал раз в 20 или 200 быстрее, но... не настолько же хорошо, чтобы отвлекаться от проектирования турбин, реакторов или новых лекарств ради освоения всех этих Линуксов, серверов и демонов. Рассказ о новом, прекрасном с точки зрения разработчика суперкомпьютерах, программном обеспечении воспринимается, в лучшем случае, со снисходительной улыбкой занятого взрослого, общающегося с заигравшимся, увлеченным ребенком. Удастся ли действительно

прорваться сквозь этот детский лепет? Стоит ли это делать, поможет ли это лучше завершить расчеты к концу отчетного квартала? Надо попробовать. Но попробовать на практике, а не на бумаге. Получится — будем искать средства на покупку этой чудо-техники. А на чем пробовать, если самой техники еще нет? Порочный круг. Для его разрыва нужна техника, удовлетворяющая следующим требованиям:

— **«портретное» сходство** с настоящим, «железным» вычислительным кластером при работе на нем. Не «сходство в принципе», которое дает программное обеспечение кластеров невыделенных рабочих станций, вроде MPICH для Windows, а именно полное совпадение интерфейсов. Мы — люди серьезные, «в принципе» нам не подходит — как известно, «работать должно не в принципе, а в корпусе», как говорили когда-то опытные «электронщики»;

— **нулевая цена**. Мы же только пробуем, вдруг не понравится. Причем — действительно нулевая. Если я не буду платить за оборудование, но заставляю своего системного администратора работать за семерых, обеспечивая функционирование экспериментального программного обеспечения в дополнение к его штатным обязанностям — то лучше не надо;

— способность давать хоть какой-то **реальный скоростной выигрыш** хотя бы на некоторых задачах. Не на калькуляторе же вычислять «будущее возможное ускорение» — я желаю **видеть**, что мои усилия по изучению MPI не пропали даром, что стало лучше, и делу польза, и руководству есть, что показать.

Портретное сходство в сочетании с нулевой ценой способна дать программная модель суперкомпьютера (симулятор). Скоростной выигрыш способен дать MPICH для Windows, но ни портретного сходства, ни нулевой цены эксплуатации он не даст.

**Метакластер MBC-900** [2,17] способен удовлетворить **всем** трем указанным требованиям.

Устроен он следующим образом. На выделенном наборе машин под управлением Windows, объединенном локальной сетью, устанавливается (на каждой машине) программное обеспечение VMWare [18]. Под его управлением в каждой из реальных машин создается функционирующая в фоновом режиме виртуальная машина. На каждой из таких виртуальных машин устанавливается Linux, а на получившийся Linux-кластер устанавливается программное обеспечение MBC-1000 (отсюда название: 900 — это почти 1 000), рис. 6.2.

В реализации этой идеи присутствуют две трудности:

- как обеспечить нулевую цену эксплуатации, т.е. сделать систему администрируемой отдельно от Windows-класса, на оборудовании которого она установлена;

- будет ли, и за счет чего, достигнут реальный скоростной выигрыш от параллельного выполнения программ?

Первая проблема решается написанием совсем несложных скриптов, запускающих виртуальную машину в фоновом режиме как сервис, автоматически, при старте Windows. При этом в конфигурации виртуальной машины режим виртуализации жесткого диска для узлов (но не хоста!) выбирается «Independent, non-persistent», что позволяет прерывать работу узла на ходу, не выполняя всякий раз «shutdown». При этом узел в процессе работы обладает полноценным жестким диском, но при старте «не помнит» прошлых изменений — состояние жесткого диска узла при включении всегда эталонное и правильное. Следовательно, узлы дополнительной нагрузки на администратора «несущей» нашу систему Windows сети не создают, поскольку являются необслуживаемыми. Необслуживаемость крайне важна еще и потому, что объем работы по поддержанию «мирного сосуществования» программного обеспечения физических и виртуальных машин не увеличивается с ростом числа узлов.

Виртуальные узлы не создают и проблем со штатным функционированием оборудования и программного обеспечения.

Так, экспериментально установлено и неоднократно проверено, что виртуальная машина под управлением Linux, на которой не выполняется напряженной вычислительной работы, не создает заметной нагрузки на ресурсы физической машины. Узел виртуального кластера может быть постоянно запущен на штатно используемой машине в фоновом режиме — если на виртуальном кластере ничего не считается, физическая машина не будет работать медленнее или ощущать дефицит памяти.

Никакие аварии виртуальной машины не угрожают целостности «несущей» ее операционной системы физической машины. Интерфейс между виртуальной машиной и ОС физической машины прекрасно отлажен, и опасные сбои через него не распространяются.

Хост, к сожалению, нуждается в присмотре — несущую его физическую машину нельзя выключать, не выполнив shutdown виртуального хоста. Впрочем, нарушение этого правила опасно только для самого виртуального хоста, но не для ОС несущей его физической машины.

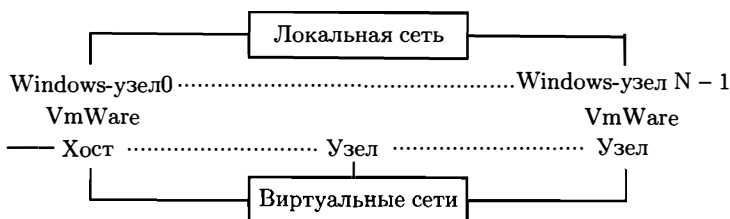


Рис. 6.2. Метакластер MBC-900 (широкой полосой обозначен выход виртуального хоста во внешнюю сеть для доступа пользователей)

Эта совокупность мер позволяет говорить о **раздельном администрировании** физической и виртуальной сетей, что и требовалось. Администрирование MBC-900 из дополнительной нагрузки на штатного администратора становится плановой учебной задачей будущего администратора будущего вычислительного кластера, т. е. совершенно закономерной частью «пробования, получится ли освоить эту технику».

Вторая проблема — откуда возьмется скоростной выигрыш? Чтобы ответить на этот вопрос, надо вспомнить, что такое виртуальная машина (в узком терминологическом смысле VmWare и подобных систем). Эмулятор? Но эмуляция процессором собственной системы команд, как известно, приводит к примерно 100-кратному замедлению в выполнении кода. До выигрыша ли тут? А если это не эмулятор, то как она вообще работает? Что происходит при выполнении процессором — не при эмуляции, а именно при честном выполнении — команды обращения к жесткому диску, сетевому адаптеру или ячейке памяти, которых нет в действительности, поскольку они виртуальны?

В гл. 2 подробно обсуждался данный, а также смежные с ним вопросы. Там же давалось определение виртуальной машины (в общем смысле этого термина), объяснялся механизм работы виртуальной памяти и специальных «виртуальных» команд — системных запросов. Смысл всей техники виртуализации — в том, что подавляющее большинство команд процессор выполняет напрямую, без всякой программной эмуляции, а переход от прямого выполнения кода к программной эмуляции той или иной виртуальной сущности происходит по прерыванию, т. е. без специальных предварительных проверок в выполняющейся программе. В случае виртуальной памяти это прерывание «неожиданное», а в случае обращения к ОС за услугой — запланированное, программное. В результате мы получаем виртуальную ма-

шину, отличающуюся по системе команд и набору внешних устройств от реальной. Чтобы получить то, что мы видим, наблюдая за работой виртуальной машины под управлением VmWare, нам достаточно распространить технику виртуализации памяти на виртуализацию привилегированных команд работы с аппаратурой. Еще раз — что происходит, когда ядро Linux на виртуальной машине обращается к несуществующему (виртуальному) адаптеру жесткого диска? Прерывание. Заменяем обработчик таких прерываний в ОС несущей машины на программный эмулятор виртуального адаптера жесткого диска, т. е. будем считать это прерывание не аварийным сигналом, а разновидностью системного запроса — и задача решена. Именно так работает VmWare. В выполняющейся под управлением VmWare виртуальной машине прерывания от команд обращения к несуществующему (виртуальному) оборудованию просто выполняют функцию системных запросов. В итоге получаем частный случай виртуальной машины, или виртуальную машину в узком смысле этого термина. А именно — виртуальную машину, которая совпадает с физической машиной по системе команд, но отличается от нее набором внешних устройств.

Что можно сказать о быстродействии такой виртуальной машины по сравнению с физической? Снижение производительности вычислительной работы, при которой ничего не эмулируется (прерываний почти нет), практически отсутствует — оно не больше, чем накладной расход на выполнение программы под управлением Linux или Windows по сравнению с выполнением на «голом» процессоре, т. е. пренебрежимо мало. Это дает надежду на получение реального скоростного выигрыша от параллельного выполнения программ. Будет ли он столь же весомым, как и на «железном» кластере — при условии, конечно, что несущие нашу систему физические машины не заняты в данный момент собственной вычислительной работой?

К сожалению, нет. Коммуникационная сеть виртуальной машины обладает примерно на порядок большей латентностью, чем та же сеть физической машины, поскольку при работе виртуального сетевого адаптера прерывания происходят не один раз — при системном запросе на сетевой обмен, а многократно — при выполнении этого системного запроса ядром, обращающимся внутри себя к виртуальному оборудованию. В итоге скоростной выигрыш от параллельного выполнения будет гораздо сильнее зависеть от «коммуникационной насыщенности» прикладных программ, чем

на физической машине. Но на «хороших» программах он может приближаться к случаю «железного» кластера.

Наконец, итоговое замечание о названии предложенной технологии. Слово «метакластер», казалось бы, претендует на принадлежность МВС-900 к области метакомпьютинга [4]. Что дает нам основание для такой претензии? Техническое назначение системы. МВС-900 можно рассматривать как вариант дисциплины отбора вычислительной мощности с вычислительной установки, изначально для этого не предназначенной. Это и есть одна из постановок задачи метакомпьютинга. Ничто не мешает применять МВС-900 в некоторых случаях не только для первоначального освоения технологий параллельной обработки данных, но и как систему отбора мощности на постоянной основе.

Программное обеспечение МВС-900 также рекомендуется для целей преподавания.

Технология VmWare является не единственной из доступных сегодня технологий виртуализации. В последнее время популярны такие системы виртуализации, как, например, coLinux или Xen, распространяемые свободно. Много информации на эту тему можно найти в Интернете по двум упомянутым ключевым словам.

# ТЕХНОЛОГИИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ И МЕТАКОМПЬЮТИНГ

Тема *метакомпьютинга* [4], очень популярная в последние годы, не является предметом специального рассмотрения в настоящем курсе. Это — совершенно отдельная, большая и интересная тема, на которую уже написано и еще будет написано очень много. Однако совсем не затронуть ее мы не можем. Проблемы метакомпьютинга требуют весьма своеобразных решений в областях, которые мы, так или иначе, рассматривали: как в технологиях параллельного программирования, так и, в особенности, в технологиях управления многопроцессорными вычислителями.

### 7.1. Две постановки задачи

Задача метакомпьютинга в общем виде — это задача построения в рамках Интернет единой, потенциально неограниченной, распределенной вычислительной среды. Как по масштабу, так и по степени ожидаемого влияния на развитие вычислительных технологий, решение этой задачи не имеет прецедентов. Сравнить ее можно, пожалуй, лишь с появлением самой сети Интернет. Продолжая аналогию с историей Интернет (конечно, в чем-то спорную, как и всякая аналогия), можно заметить, что метакомпьютинг находится сейчас в стадии развития, в которой Интернет был в самом начале 80-х годов XX в. На рубеже 70-х и 80-х годов XX в. сетевые технологии еще воспринимались как некоторая «экзотика». Всего несколько лет спустя уже не очень понятно было, как мир обходился до сих пор без Интернет, и для чего вообще может быть нужен компьютер, если не для подключения к Интернету. «Экзотикой» стали казаться все остальные компьютерные технологии. Нечто подобное должно произойти в ближайшие годы (и уже происходит) с технологиями метакомпьютинга. Многое уже достигнуто, еще больших результатов следует ожидать в самом ближайшем будущем, учитывая темп



развития этих технологий. Число действующих международных сетей метакомпьютинга уже сейчас измеряется десятками, и быстро растет. На Интернет-портале [4] поддерживается и постоянно обновляется богатый раздел, посвященный технологиям метакомпьютинга, где обо всем этом можно узнать подробнее.

На практическом уровне можно выделить две постановки задачи метакомпьютинга: «узкую» и «широкую».

**Под метакомпьютингом в узком смысле этого слова** понимается задача объединения территориально удаленных и независимо администрируемых вычислительных установок в единый вычислительный ресурс. Например, несколько суперкомпьютерных центров коллективного пользования могут договориться о создании общей очереди задач, с направлением задач из очереди на выполнение в тот суперкомпьютерный центр, где в настоящее время имеются свободные ресурсы. Или же можно попытаться объединить несколько суперкомпьютеров в единое вычислительное поле для решения особенно крупных задач. На решение преимущественно этой задачи ориентировано, в частности, такое программное обеспечение из области технологий Grid, как программный комплекс Globus [4, 40] — пожалуй, самой популярной в настоящее время технологии метакомпьютинга.

**Под метакомпьютингом в широком смысле этого слова** понимается задача «брать из сети Интернет вычислительную мощность так же легко, как мы сегодня берем из нее документы». Иными словами — создание средств возможно более прозрачного отбора вычислительной мощности с компьютеров, которые, вообще говоря, не являются вычислительным ресурсом коллективного пользования, т.е. установлены и сконфигурированы для решения совершенно других задач. Например, в простейшем частном случае, это всевозможные «скрин — сейверные технологии», вроде *seti@home*.

По аналогии с вычислительными кластерами будем называть эти задачи *задачей метакомпьютинга выделенных мощностей* и *задачей метакомпьютинга невыделенных мощностей*, соответственно.

Обе задачи невероятно сложны и масштабны, в обоих случаях организационные сложности преобладают над техническими. Вторая — более общая — задача гораздо сложнее первой и в организационном, и в техническом отношении. Настоящая глава ни в коей мере не претендует на систематический обзор этой проблематики, даже краткий. Хотелось бы лишь сформулировать в

общем виде сами проблемы и привести несколько примеров их конкретного решения, перекликающихся так или иначе с материалом настоящего курса.

## **7.2. Метакомпьютинг выделенных мощностей.**

### **Пример технологии**

Главная техническая проблема метакомпьютинга выделенных мощностей — это, конечно же, недостаток производительности коммуникационного канала, связывающего между собой части системы. Проблема эта стоит очень остро даже в том случае, когда территориально удаленные части системы не используются для организации единого вычислительного поля. Например, пусть имеется всего лишь общая очередь задач, каждая из которых по мере освобождения соответствующих ресурсов направляется целиком на один из суперкомпьютеров комплекса, но заранее не известно, на какой именно. Этой задаче, очевидно, потребуются какие-то исходные данные, а также место для записи результатов расчета. Файловый ввод-вывод зачастую является узким местом даже для классических, территориально сосредоточенных, кластеров. В случае же, когда между вычислительным полем и хранилищем данных, которые на нем обрабатываются, «внезапно» возникает спутниковый или другой канал, более узкий, нежели канал доступа к собственному файловому серверу, проблема рискует стать неразрешимой.

Задачи, в которых объем обмена данными с файловой системой особенно велик, в рамках этих технологий не решаются. Эта же проблема недостаточной производительности канала между составляющими комплекс суперкомпьютерами встает в случае, когда делается попытка организовать из узлов нескольких суперкомпьютеров единое вычислительное поле. При этом добавляется еще и проблема латентности — все высокоскоростные магистральные каналы передачи данных обладают очень высокой (по сравнению с собственной производительностью) латентностью.

Обычно программистам, пишущим программы для таких комплексов, приходится явно учитывать в структуре программы тот факт, что общее (формально) поле вычислительных узлов разбито в действительности на несколько сравнительно слабо связанных между собой подмножеств.

Организационные проблемы, возникающие при работе на метакомпьютерах, достаточно очевидны, и связаны они с фактом независимого администрирования составляющих метакомпьютер вычислительных установок. Сюда относятся не только проблемы авторизации или согласования политики безопасности, которая, как хорошо известно, у каждого системного администратора своя, причем, конечно же, «уникальная» и «лучшая в мире». Собственно, решению именно этих проблем и посвящено в основном программное обеспечение Globus. Есть и другие, не менее сложные проблемы. Например, если в единое вычислительное поле выделяются узлы двух установок, каждая из которых снабжена своей системой очередей, то очереди на выделение ресурсов для двух «половинок» одной большой задачи должны «подойти» примерно одновременно. Сделать это, конечно же, весьма нелегко. Ведь требуется, с одной стороны, сохранить системы планирования каждой из установок, с другой — объединить их на некоторое время.

Рассмотрим в качестве примера одну частную техническую проблему, возникающую в связи с задачей объединения узлов нескольких параллельных вычислителей в единое вычислительное поле. Как могла бы выглядеть система параллельного программирования для такого метакомпьютера? Очевидно, это должна быть система программирования с помощью MPI, поскольку канал связи, соединяющий отдельные суперкомпьютеры в этой системе, почти наверняка гораздо более медленный, чем хотелось бы, и совершенно определенно — очень высоклатентный, т.е. мы имеем дело с системой, в которой узлы связаны слабо. Рассмотрим проблемы, возникающие при попытке построить такую реализацию MPI. Как и большинство проблем метакомпьютинга, проблемы эти находятся «на стыке» организационной и технической областей.

Как известно, помимо реализаций MPI на базе протоколов tcp/ip, существуют реализации MPI на базе специализированных, нестандартных протоколов, присущих тем или иным высокоскоростным сетям. Например, MPI для Myrinet, или MPI для Infiniband. Высокая эффективность этих реализаций именно тем и достигается, что построены эти реализации не на базе tcp/ip. Почти наверняка объединяемые в метакомпьютер установки оснащены именно такими реализациями MPI, и естественно полагать, что реализации эти разные. Например, одна из объединяемых машин построена на базе Myrinet, а вторая — на базе

Кластер на базе Myrinet ——— Спутниковый канал (tcp/ip) ——— Кластер на базе Infiniband

Рис. 7.1. Разнородный метакомпьютер

Infiniband. Объединяющий их Internet-канал, вероятно, спутниковый, поддерживает tcp/ip (рис. 7.1).

Проще всего было бы использовать на объединенной установке MPI на базе tcp/ip, но это привело бы к падению эффективности коммуникаций в пределах каждой из объединяемых установок. Хотелось бы построить такую реализацию MPI, чтобы в пределах каждой из объединяемых установок сообщения пересылались по «внутренним» протоколам, а между установками — по tcp/ip. Принципиально ничего невозможного в этом нет, но технически за такую реализацию вряд ли кто возьмется — слишком уж велико число возможных сочетаний из разных сетей. А вдруг к метакомпьютеру решено будет присоединить еще и машину на базе Quadrics?

Для удовлетворительного решения этой задачи необходимо научиться «вживлять» в работу функций уже существующей реализации MPI некоторые дополнительные действия, например проверку, по Myrinet отправлять сообщение, или же по tcp/ip, но при этом не транслировать заново ни саму библиотеку MPI, ни вызывающую программу.

Исторически эта проблема встала перед разработчиками MPI впервые в связи с необходимостью добавлять к имеющимся реализациям MPI средства профилирования и отладки. Хотелось бы иметь возможность создать «обертку» для каждой функции MPI, которая работает так:

- принимает все аргументы данной функции;
- делает нечто (например, засекает время);
- вызывает данную функцию;
- делает еще что-то, например, еще раз засекает время, вычисляет время работы функции и что-то с этим делает;
- возвращает все результаты, которых вызывающая программа ждет от данной функции.

Понятно, что в эту логику укладывается не только измерение времени, но и, например, построение «связки» из двух систем без нарушения целостности MPI в каждой из них, и многое другое.

Во всем этом не было бы ни малейшей проблемы, если бы не еще одно, совершенно естественное, требование: и «обертка», и исходная функция MPI **должны называться одинаково!** Ведь тексты не только самой библиотеки MPI, но и прикладной программы могут быть не доступны для повторной трансляции.

Решение этой проблемы, являющееся частью стандарта MPI, основывается на так называемых «слабых символах (weak symbols)» — некогда дополнительной, а сегодня — практически обязательной возможности транслятора C и компоновщика (линкера).

От транслятора требуется наличие директивы `#pragma weak`, позволяющей снабдить имя функции синонимом, объявив его «слабым». Например, функция может иметь основное имя “abc” и слабый синоним “def”, обозначающие одну и ту же точку входа в функцию. Реализация такой возможности ничего не стоит разработчику транслятора, поскольку на уровне формата объектного модуля этот аппарат всегда был. Например, на языке ассемблера любая функция может иметь как угодно много равноправных названий. На Фортране запись синонимов, кстати, также допускается. Мы, впрочем, хотим, чтобы каждый из синонимов не просто обозначал одну и ту же функцию, но и имел дополнительный атрибут — признак «слабости».

Для вызывающей программы наличие синонима означает, что функцию можно с равным успехом вызвать и как “abc(...)”, и как “def(...)”, причем работать будет в обоих случаях один и тот же код.

От компоновщика — линкера — требуется лишь одна, также довольно простая, возможность. Если, в процессе связывания объектных модулей в загрузочный, компоновщик встречает повторное определение имени функции, т. е. функцию, имя одного из синонимов которой уже встречалось, и вновь встреченный синоним — «слабый», то это — не ошибка. Слабый синоним можно молча игнорировать, а функцию связывать обычным порядком, если она вызвана по другому имени.

В стандарт MPI входит так называемый ***интерфейс профилирования***. Он состоит в том, что каждая из функций MPI имеет свое каноническое имя в качестве слабого синонима, и имя, полученное приписыванием перед каноническим именем буквы “P”, в качестве основного имени. Так, функция MPI\_Send «в действительности» называется PMPI\_Send, а “MPI\_Send” — лишь слабый синоним ее основного имени.

Пусть теперь мы хотим вставить «обертку» между прикладной программой и библиотекой MPI, не транслируя заново ни программу, ни библиотеку. В программе все функции MPI вызваны по «слабому» варианту, без “P”. Мы же написали, например, для MPI\_Send «обертку» такого вида:

```
int MPI_Send( ..... )
{
    .....
    rc = PMPI_Send( ..... );
    .....
    return rc;
}
```

Если в командной строке линкера указать объектный модуль нашей обертки до библиотеки MPI, то наш вариант MPI\_Send скомпилируется вместо библиотечного. Он вызовет PMPI\_Send уже из библиотеки. Доставая из библиотеки объектный модуль по имени PMPI\_Send, линкер с негодованием обнаружит, что его синонимом является MPI\_Send, а такое имя уже встречалось, как имя совсем другой функции (нашей «обертки»). Казалось бы, налицо повторное определение имени, надо ругаться и завершать работу. Однако, библиотечный синоним MPI\_Send, к счастью, «слабый», и линкер просто проигнорирует его, скомпилировав функцию по альтернативному имени. Задача решена — мы «вклинились» со своим кодом между программой и библиотекой, не имея исходных текстов ни того, ни другого.

Именно на использовании интерфейса профилирования построена библиотека PACX — MPI [29] — набор программ построения «связок» нескольких библиотек MPI в одну без какого-либо вмешательства в связываемые библиотеки. PACX — MPI удобен для применения в GRID-системах. С его помощью, например, легко связать в единое вычислительное поле два суперкомпьютера, один из которых оснащен сетью Myrinet, а второй — сетью Infiniband, причем связь между этими двумя компьютерами осуществляется по спутниковому Internet-каналу. При работе PACX — MPI в рамках этого единого вычислительного поля передача данных внутри каждой из его частей будет происходить с использованием штатно установленной на этой части реализации MPI, и лишь при посылке сообщений из одной части в другую будет использоваться tcp/ip. Сам же PACX — MPI — всего лишь набор оберток.

MPICH — самая популярная свободно распространяемая реализация MPI — включает в себя очень удобный дополнительный сервис по использованию интерфейса профилирования. В составе MPICH поставляется программа `wrappergen` (генератор оберток) — специализированный транслятор со специально разработанного языка, на котором пишутся сценарии порождения полного комплекта оберток для данной реализации MPI. Программа на входном языке `wrappergen` пишется примерно в таких терминах: «Для всех функций, кроме `MPI_Send` и `MPI_Recv`, включить в обертку только измерение времени на входе и выходе, а для указанных функций — следующий код: .....». Что такое «для всех функций», транслятор решит самостоятельно, проанализировав файл `“mpi.h”`. Выходом `wrappergen` является текстовый файл с текстами оберток всех функций данной реализации MPI на языке C.

### 7.3. Метакомпьютинг невыделенных мощностей. Пример технологии

Метакомпьютинг невыделенных мощностей добавляет к проблематике метакомпьютинга выделенных мощностей еще одну, сложнейшую во всех отношениях, проблему — проблему использования *вычислительной мощности низкого качества*.

В самом деле, нелегко организовать единую среду выполнения параллельных программ даже из небольшого числа территориально удаленных и независимо администрируемых суперкомпьютеров. Однако, будучи все же организована, такая среда обладает предсказуемыми во всех отношениях характеристиками производительности. Быстродействие процессоров известно заранее, и, что не менее важно, заранее известно, что все это быстродействие, а также вся «ширина» коммуникационного канала будут во время счета задачи находиться целиком в распоряжении этой задачи. Ведь узлы метакомпьютера — это узлы кластеров выделенных рабочих станций, которые занимаются только порученным им расчетом, и больше ничем. Именно это позволяет спланировать параллельную реализацию алгоритма таким образом, чтобы процессоры работали одновременно, давая скоростной выигрыш.

В случае метакомпьютинга невыделенных мощностей ситуация принципиально иная. Поскольку отбор мощности ведется с

большого количества разнородных машин, каждая из которых, вообще говоря, занята другой работой, сравнительная производительность виртуальных процессоров меняется в широких пределах. Неизбежен дисбаланс нагрузки и, как следствие, утрата эффекта ускорения.

В качестве примера рассмотрим такую примитивную систему отбора мощности, как метакластер МВС-900. Пусть метакластер установлен на офисной сети, про которую достоверно известно, что в среднем, за длительные (порядка минут) промежутки времени каждый из компьютеров преимущественно простаивает. Казалось бы, случай идеальный — можно отбирать неиспользуемую мощность. Запустим на таком метакластере нашу модельную программу прогрева кирпича. Время итерации в нашем расчете — порядка миллисекунды. На каждой итерации происходит общая синхронизация, т. е. ожидание опоздавшего. Каждая итерация происходит, в итоге, с той скоростью, которую показывает самый медленный из процессоров. В случае одинаковых выделенных процессоров время расчета итерации для всех процессоров одинаково. В случае невыделенных процессоров на каждом из процессоров время от времени проявляется какая-то вычислительная активность, не связанная с нашим расчетом, и в это время отбираемое нами быстродействие данного процессора оказывается гораздо ниже, чем обычно. Учитывая, что время итерации — порядка миллисекунды, достаточно всего лишь миллисекундного падения отбираемого быстродействия, чтобы некоторая итерация нашего модельного расчета резко замедлилась. Но ведь в расчете участвует не один процессор, и на каждом из них возможны такого рода «досадные неприятности». Вполне уместно поставить вопрос несколько иначе: а много ли наберется таких миллисекунд, в течение каждой из которых ни один из процессоров, «живущих своей жизнью», ни разу не замедлился? В итоге ситуация довольно парадоксальная: вычислительная мощность есть — все процессоры в среднем почти свободны — но взять ее в параллельном режиме нельзя, поскольку она *низкого качества*. В данном случае это означает всего лишь, что отбираемое быстродействие отдельного процессора непредсказуемо на коротких отрезках времени.

В общем, к понятию низкого качества вычислительной мощности относится и слабость коммуникационной сети, и ее неоднородность, словом — все то, что усложняет организацию совместной работы процессоров над одной задачей. Особенно тяжелый



случай «низкокачественности» — ненадежность виртуального узла. В системах отбора мощности с невыделенных процессоров совершенно нормальной является ситуация, когда некоторый виртуальный процессор просто перестает существовать, поскольку соответствующую физическую машину перезагрузили или выключили.

В общем, как легко видеть, справедлива простая закономерность. Чем менее тесно взаимодействие между ветвями параллельной программы, тем больше шансов на успех при попытке выполнить такую программу в среде низкокачественной вычислительной мощности. Например, описанная выше проблема колебания отбираемого быстродействия на миллисекундных интервалах времени вряд ли имеет шанс проявиться, если интервал между сообщениями в задаче — секунды или минуты. Такая задача вполне способна потребить вычислительную мощность низкого качества. Более того, она имеет шанс выполняться на метакомпьютере почти так же хорошо, как и на «настоящем» суперкомпьютере. Впрочем, проблемы ненадежности виртуального узла такой подход все равно не решает.

Для иллюстрации проблемы отбора вычислительной мощности разного качества приведем несколько примеров из реальной жизни.

*Пример 1* (успешное использование метакластера МВС-900 в производственной обстановке). Некоторая крупная проектная организация машиностроительного профиля ощутила потребность в высокопроизводительных вычислениях, точнее, в ускорении расчетов по конкретной прикладной программе, многопроцессорный вариант которой на базе MPI уже существовал, хотя его не на чем было выполнять. Заинтересованная лаборатория обладала примерно десятком очень хороших компьютеров сравнимого быстродействия, объединенных крайне слабой сетью. Эти машины было решено использовать в качестве узлов МВС-900. Для хоста МВС-900 был приобретен отдельный компьютер, находившийся под присмотром сотрудника, назначенного администратором. Территориальная и административная сосредоточенность компьютеров позволяла договариваться о том, чтобы на время длительных счетных прогонов машины не перезагружались, не выключались и не отключались от сети. Этого оказалось достаточно для уверенной полупромышленной эксплуатации. Было получено кратное ускорение счета для реальных производственных задач в реальной производственной обстановке. При

этом узлы ничего не стоили хозяевам оборудования не только в смысле цены самих компьютеров, но и в смысле затрат труда на эксплуатацию — как и планировалось разработчиками МВС-900, виртуальный узел оказался не нуждающимся в обслуживании и очень живучим. В качестве «побочного продукта» был подготовлен грамотный администратор МВС-1000, и организация смогла конкретизировать свои потребности в высокопроизводительных вычислениях. Это — пример успешного «полупрозрачного» отбора вычислительной мощности с невыделенных узлов.

*Пример 2* (неудача проекта «Дубна-грид»). На базе идей, положенных в основу МВС-900, была сделана попытка организации Grid-структуры обработки данных, получаемых в экспериментах на ускорителях элементарных частиц. Вычислительная структура строилась в интересах Объединенного института ядерных исследований (ОИЯИ) в г. Дубна. На нескольких сотнях компьютеров образовательной сети города, установленных в школах и Университете, были развернуты бездисковые (загружаемые по сети) виртуальные узлы, сконфигурированные в соответствии со стандартами ЦЕРН для машин, объединяемых в Grid. Особо отметим, что расчеты, о которых идет речь, не являлись параллельными. Речь шла о вариантном счете, т. е. о пропуске очень большого числа отдельных задач, заключающихся в весьма длительной обработке довольно небольшого объема исходных данных в каждом расчете. Образовательная сеть г. Дубна прекрасно оснащена, и проблем с ее производительностью не прогнозировалось. Казалось бы, идеально подходящий для отбора вычислительной мощности случай.

После не очень продолжительной экспериментальной фазы проект пришлось свернуть. Фатальной оказалась проблема надежности узлов. Территориальной и административной рассредоточенности, в сочетании с полной незаинтересованностью хозяев физических машин в успехе проекта, оказалось достаточно, чтобы машины слишком часто перезагружались и выключались, а результаты запущенных на них расчетов гибли.

Этот пример исключительно показателен, и вот почему.

С одной стороны, сама природа решаемых задач блестяще подходила для отбора вычислительной мощности как угодно низкого качества. В самом деле, задачи полностью независимы, могут выполняться в любом ритме и в любом порядке, в принципе — с любой вероятностью успеха. В случае неудачи (отказа виртуального узла) расчет необходимо всего лишь повторить еще раз.

С другой стороны, используемая в готовом виде дисциплина обработки потока задач, принятая в ЦЕРН, не содержит подсистемы ведения статистики такого рода, не приспособлена для ведения «реестра» успешных и неуспешных запусков, с повторением запуска до тех пор, пока он не увенчается успехом. Это и обусловило, в конечном счете, неудачу проекта. Задача, принципиально подходящая для использования вычислительных структур такого рода, оказалась совершенно не оформлена надлежащим образом на практике.

Это позволяет нам сделать очевидный, казалось бы, но тем не менее очень важный вывод:

*системы отбора вычислительной мощности с невыделенных компьютеров являются совершенно отдельным классом вычислительного оборудования, отличным от известных нам до сих пор четырех классов, и нуждаются в адекватной программистской и пользовательской абстракции на уровне моделей и технологий.*

Попытка механического переноса технологий из мира «настоящего» оборудования, вообще говоря, не ведет к успеху.

Интересным и, по многим отзывам, плодотворной попыткой построить такую модель является разработанная в НИВЦ МГУ система X-com [4]. Это очень прагматичная система, специально спроектированная в расчете на преодоление всех тех трудностей, о которых мы говорили в настоящем разделе. Система прекрасно документирована, и разбирать ее подробно мы здесь не будем. Остановимся лишь на основных принципах организации метакомпьютера и программистской модели.

**Метакомпьютер X-com** представляет собой набор компьютеров под управлением Linux или Windows, подключенных к Интернету. Для связи между ними применяется протокол HTTP.

Компьютеры эти не являются выделенными, т. е. каждый из них сконфигурирован и используется для какой-то своей работы, отличной от выполнения приложений X-com. Приложения X-com выполняются в фоновом режиме, не нарушая этой основной работы. Компьютеры не предполагаются ни однородными, ни связанными достаточно «хорошей» сетью, ни территориально сосредоточенными, ни согласованно администрируемыми. Состав метакомпьютера не предполагается постоянным — составляющие его компьютеры могут появляться и исчезать в процессе выполнения приложения, т. е. выключаться, перезагружаться, или же,

наоборот, включаться и присоединяться к работе метакомпьютера. Словом, мы имеем дело с весьма общим случаем системы, отбираемая с которой вычислительная мощность крайне низкого качества.

*Приложение X-com* устроено совершенно иначе, чем параллельные программы, с которыми мы имели дело до сих пор. Необходимость в этом диктуется двумя соображениями. Во-первых, такое приложение неизбежно подразумевает гораздо более редкие, чем в традиционной параллельной программе, межпроцессорные обмены данными, с характерным временем независимой работы процессора от обмена до обмена порядка, как минимум, минут. Во-вторых, любой процессор, выполняющий свой, независимый от других, фрагмент работы, постоянно рискует «не дожить» до следующего акта взаимодействия, и тогда порученный этому процессору фрагмент работы придется повторно поручить другому процессору, повторно передавая ему необходимые данные. В этих условиях гораздо естественнее не выписывать запросы на обмен данными в виде операторов программы, как мы привыкли, пользуясь традиционными технологиями, а просто оформить независимые фрагменты работы в виде отдельных исполняемых программ. Обмен данными между этими программами можно организовать через файлы, расположенные в общедоступном месте. Если выполняющий одну из этих программ процессор внезапно выключился, то при повторном запуске этой программы на другом процессоре она просто прочитает те же самые файлы с исходными данными, и ничего не пропадет.

При такой организации приложения в его состав обязательно должна входить специальная программа — арбитр, которая обеспечивает необходимую очередность запуска расчетных фрагментов. Такую программу — арбитр пользователь должен написать самостоятельно, поскольку только он знает, какие из расчетных фрагментов зависят друг от друга по исходным данным, а какие можно выполнять параллельно. Также арбитру придется поддерживать хранилище файлов, через которое обмениваются данными расчетные фрагменты, и следить за успешностью выполнения этих фрагментов, повторяя, при необходимости, запуск фрагмента на другом узле.

Именно на этих основных принципах и построена модель программирования X-com. Приложение организуется в виде набора *вычислительных блоков и серверной части*.

**Вычислительный блок** представляет собой исполняемый файл программы, набор файлов исходных данных для этой программы, и набор файлов результатов счета, которые программа должна породить. Программа в составе вычислительного блока — обычная, последовательная программа для однопроцессорного компьютера, ничего подобного обращениям к функциям MPI в ней нет. В процессе выполнения вычислительный блок никак не взаимодействует с себе подобными. Выполнение приложения состоит в выполнении некоторого количества вычислительных блоков, возможно, в некотором определенном порядке, поскольку результаты выполнения одних вычислительных блоков могут, вообще говоря, быть исходными данными для других. Например, вполне возможен такой сценарий:

- выполнить независимо исполняемый файл А с вариантами исходных данных А1, А2, ..., А15, получив результаты В1, В2, ..., В15, соответственно;

- затем выполнить исполняемый файл Б, с исходными данными В1, В2, ..., В15, и, если он вернет код ответа, равный нулю, то закончить расчет, иначе повторить все с самого начала.

В данном случае первые 15 вычислительных блоков не зависят друг от друга, и могут выполняться на разных компьютерах одновременно, а 16-й, включающий выполнение исполняемого файла Б, может быть запущен только после успешного завершения каждого из первых 15.

**Серверная часть приложения** используется для координации запуска вычислительных блоков и для мониторинга успешности их завершения. Она оформляется как отдельный исполняемый файл, возможно, выполняющийся на отдельном компьютере, который называется **сервером** данного приложения (остальные компьютеры, на которых выполняются вычислительные блоки, называются **узлами**).

На сервере, т. е. на компьютере, выполняющем серверную часть приложения, хранятся все исполняемые файлы вычислительных блоков, а также все файлы с исходными данными и результатами. Если метакомпьютер гетерогенный, исполняемые файлы заготавливаются для всех возможных вариантов системы команд и/или ОС узлов.

Узлы периодически обращаются к серверу за очередной порцией работы (смысл слова «периодически» будет прояснен ниже). В ответ на такое обращение сервер отправляет на узел исполняемый файл и исходные данные некоторого вычислительного

блока. Узел выполняет предложенный ему вычислительный блок и отправляет на сервер результаты. Серверная часть приложения отвечает за соблюдение необходимого по смыслу порядка выполнения вычислительных блоков, а также за успешность такого выполнения, выдавая вычислительные блоки на выполнение в определенном порядке.

Например, пусть в процессе работы по упомянутому выше сценарию сложилась следующая ситуация.

Первые 15 вычислительных блоков распределены по 15 различным узлам, и 14 из них успешно прислали результаты счета, а один — не прислал. При этом оцениваемое время выполнения такого вычислительного блока равно трем минутам, а прошел уже час. Серверная часть данного приложения должна обнаружить эту ситуацию и «сообразить», что соответствующий «замолчавший» узел, видимо, отключился, и в ответ на очередной запрос работы повторно предложить выполнить расчет, результат которого не прислан, вместо того чтобы механически заказывать расчет блока номер 16.

Таким образом, серверная часть приложения — это совершенно отдельная программа, которую пользователь должен написать, наряду с программами, реализующими вычислительные блоки.

На узлах работают специальные клиентские приложения, которые периодически обращаются к серверу за очередным вычислительным блоком. Конкретный смысл слова «периодически» в предыдущей фразе может быть различным. Возможны варианты от однократного запроса вычислительного блока, подлежащего выполнению, до организации с помощью демона *stop* ежеминутного мониторинга загруженности процессора узла и запуска очередного блока, только если процессор достаточно слабо загружен. Если процессор, напротив, становится загруженным слишком сильно, клиентская часть может уничтожить процесс выполнения вычислительного блока, чтобы освободить ресурсы машины для ее основной деятельности, не связанной с участием в метакомпьютере. В любом случае администратор машины, являющейся узлом, полностью контролирует характер ее участия в работе метакомпьютера.

Серверную часть приложения будем далее называть для краткости просто сервером, имея в виду не компьютер, а именно исполняемую программу.

Для каждого приложения требуется, вообще говоря, изготовить свой сервер. Как и в большинстве технологий программи-

рования, сервер складывается из функций, написанных пользователем специально для данного приложения, и стандартной части, оформленной в виде библиотеки объектных модулей. Оттранслировав функции собственного изготовления и скомпоновав их с библиотекой, получаем исполняемый файл сервера для данного приложения. В отличие от многих привычных технологий программирования, логика сборки сервера из пользовательской и общей, стандартной, части в данном случае «вывернута наизнанку». Мы привыкли к тому, что общая логика программы, начиная с функции `main`, пишется пользователем, и в этой написанной пользователем программе присутствуют обращения к библиотечным функциям. В случае сервера X-com ситуация противоположная. Общая логика работы сервера, включая функцию `main`, написана раз и навсегда, т. е. входит в библиотеку. От пользователя требуется написать несколько небольших функций, которые сервер вызывает в процессе своей работы, таких, например, как «запросить очередную порцию исходных данных». Когда некоторый узел обратится к серверу за очередной порцией работы, в серверной части приложения будет вызвана эта функция.

В описании системы явно сказано, что критически важным условием применимости технологии является возможность разбиения задачи на вычислительные блоки с характерным временем выполнения порядка минут. В наших терминах именно возможность такого разбиения и определяет возможность решения задачи в условиях предоставления ей вычислительной мощности низкого качества.

### Глава 8

## «ЭПОХА КЛАСТЕРОВ» ЗАКАНЧИВАЕТСЯ

Содержание предыдущих глав целиком посвящено старым, устоявшимся технологиям. По меркам суперкомпьютерной отрасли — не просто «старым», а «древним» — кроме, конечно, сравнительно «молодых» технологий метакомпьютинга. Уже почти 10 лет весь мир строит кластеры выделенных рабочих станций. Технологии их использования отработаны, как хороший сборочный конвейер. Тем временем, мир компьютеров массового выпуска, поставляющий компоненты для кластеров, буквально бурлит.

Хорошо известно, что ощущение «конца истории» в любой отрасли, техническая база которой бурно развивается, практически всегда означает, что внезапные радикальные изменения не за горами. В чем же они будут заключаться?

Вспомним, что нам известно о закономерностях изменения облика суперкомпьютеров (см. гл. 1):

1) облик суперкомпьютера на каждом этапе технологического развития определяется Правилom экономии сил разработчика. Суперкомпьютеры просто вынуждены быть устроены так, как диктует логика возможно более полного заимствования компонентов из мира компьютеров массового выпуска, и изменяться вслед за изменениями в этой логике;

2) две суперкомпьютерные революции дают нам два примера конкретных движущих сил, порождающих принципиально новые технические решения. Так, наиболее масштабные изменения происходят по причине роста числа транзисторов на кристалле микросхемы (первая суперкомпьютерная революция). Не столь масштабные, но тоже очень важные изменения происходят, когда в мире компьютеров массового выпуска резко меняются ком-



муникационные, сетевые технологии (вторая суперкомпьютерная революция).

В данной главе будет показано, что в настоящий момент **оба** этих фактора действуют одновременно и взаимосвязанно, порождая «кумулятивный эффект». Это с неизбежностью должно привести (и уже приводит) к **третьей суперкомпьютерной революции**, которая обещает изменить облик суперкомпьютера так сильно, как он еще не менялся за всю историю современной информатики.

Сначала оценим масштаб роста числа транзисторов на кристалле, который произошел за время с первой суперкомпьютерной революции до наших дней. Забегая вперед, отметим, что те изменения в коммуникационных технологиях, которые нам еще предстоит обсудить, сами являются прямым следствием роста числа транзисторов на кристалле.

В конце 60-х годов XX в., когда С. Крей еще только придумывал Cray-1 — одну из последних супермашин «домногопроцессорной» эпохи, в СССР был построен суперкомпьютер БЭСМ-6. В его процессоре было около 15 тыс. транзисторов и еще больше диодов. Примерно 20 лет спустя «домногопроцессорная эпоха» кончилась, поскольку число транзисторов на кристалле, т.е. в процессоре, перевалило за миллион. Прошло еще 20 лет. Сегодня в современном сложном и быстром процессоре транзисторов уже миллиард. Транзисторов на кристалле стало больше, чем в начале первой суперкомпьютерной революции было в микропроцессорах всей вычислительной системы. Очевидно, все предпосылки для очередного технологического переворота — третьей суперкомпьютерной революции — налицо.

Приступая к рассмотрению такой рискованной вещи, как прогноз, пусть и краткосрочный, направления развития отрасли, следует еще раз уточнить термины. Само понятие суперкомпьютерных революций в том виде, в каком оно формулируется в этой книге, вовсе не претендует на полноту охвата **всех** идей, подходов и технологий, существующих в суперкомпьютерной отрасли. Например, выше мы говорили о второй суперкомпьютерной революции как о переходе к господству в отрасли кластерных технологий. Это, однако, вовсе не отменяет того факта, что на протяжении всего «кластерного» десятилетия такими фирмами, как Cray и Silicon Graphics, интенсивно и успешно развивались технологии построения суперкомпьютеров на оригинальной схмотехнической (и даже элементной) базе. Это, в свою очередь,

не отменяет того факта, что в течение упомянутого промежутка времени именно кластеры в преобладающей степени определяли лицо отрасли. При этом степень количественного преобладания кластерной технологии над технологиями Cray и Silicon Graphics в полной мере выяснилась «задним числом», на опыте уже прошедшего «кластерного» десятилетия.

Примерно так же следует относиться к предлагаемым здесь вниманию читателя рассуждениям о третьей суперкомпьютерной революции. Содержание второй части книги ни в коей мере не претендует на исчерпывающую систематизацию (и даже достаточно полный обзор) **всего** того, что будет происходить в отрасли в ближайшие годы. Речь идет лишь о том, что направления развития суперкомпьютерных архитектур, о которых пойдет речь ниже, представляются автору назревшими, подготовленными объективным ходом всей предшествующей истории отрасли. Быть может, им надлежит сыграть ту же роль, какую сыграли десять лет назад кластерные технологии. Может статься и так, что автор «проглядел» другие тенденции развития, за которыми, в действительности, будущее. Возможно (и даже весьма вероятно), что отрасль пойдет по пути, прогнозируемому автором, но не только по нему. Конкретный, точный ответ на эти вопросы мы узнаем лет через 10, на пороге очередной — четвертой — суперкомпьютерной революции, о которой пока можно сказать только то, что она обязательно будет. Пока же попробуем обрисовать контуры третьей, не забывая, что это — лишь прогноз одного из направлений развития.

В этой главе мы расскажем о двух взаимосвязанных, но все же достаточно автономных направлениях третьей суперкомпьютерной революции.

*Первое направление* — коренное изменение коммуникационных технологий. Рост числа транзисторов на кристалле сделал возможным «стирание граней» между способами интеграции функциональных устройств в пределах материнской платы и способами интеграции материнских плат в локальную сеть. Последствия этих изменений в коммуникационных технологиях сами по себе достаточно революционны для того, чтобы их стоило рассмотреть внимательно и подробно. Однако, как мы увидим по ходу рассказа, третья суперкомпьютерная революция к этим изменениям вовсе не сводится.

*Второе направление* третьей суперкомпьютерной революции состоит в изменениях в организации самих вычислителей. Фоннейма-

новский процессор практически перестал быть адекватной вычислительным приложениям формой организации того количества транзисторов, которое может быть размещено на кристалле сегодня.

Между первым и вторым направлениями очередной суперкомпьютерной революции имеется как глубинная взаимосвязь, так и существенные различия.

С одной стороны, первое направление является предпосылкой, необходимым условием второго. Какими бы конкретно ни были новые вычислительные структуры, идущие на смену традиционным процессорам, построение из них систем массового параллелизма вряд ли возможно без новых коммуникационных технологий, так что «вычислительное» направление без «коммуникационного» во многом теряет смысл.

С другой стороны, первое, коммуникационное, направление обладает самостоятельной ценностью, даже если разработка перспективных вычислительных структур не предполагается, а речь идет об использовании обычных процессоров. При этом общие контуры предстоящих изменений в коммуникационных технологиях уже более или менее очевидны. Вопрос о том, как именно будет выглядеть новая коммуникационная среда, не стоит. Уже сегодня это более или менее ясно.

С третьей стороны, о втором, вычислительном, направлении ничего подобного сказать нельзя. Здесь не ясно практически ничего. Существует целый ряд подходов к построению альтернативных вычислительных архитектур, и огромное количество нерешенных вопросов. Ясно лишь, что будущее — за новыми архитектурами, но совершенно не известно, за какими именно.

## **8.1. Магистраль обмена данными с процессором: шина или сеть?**

До недавнего времени коммуникационные технологии, применявшиеся для связи между функциональными устройствами в пределах материнской платы (внутренние), принципиально отличались от коммуникационных технологий, применявшихся для связи между материнскими платами узлов в пределах вычислительного кластера (внешних).

Внутренние коммуникации осуществлялись с помощью системы *общих шин*, в то время как внешние коммуникации использовали оборудование локальных сетей.

Общая шина — это коммуникационная магистраль, способная обеспечить очень высокие скорости и очень малые задержки при передаче данных. Платой за высокие показатели эффективности является большое число проводов (десятки и сотни), согласованно меняющих свое состояние на очень высоких частотах, причем по довольно сложным правилам. Это подразумевает, в свою очередь, предельно жесткие требования к качеству физической реализации: длинам соединений, количеству разъемов, качеству раскладки проводников по слоям платы и т. п.

В результате вывод внутренней коммуникационной магистрали за пределы материнской платы в целях построения единой коммуникационной магистрали, охватывающей несколько материнских плат, становится либо невозможным, либо запредельно дорогостоящим.

Именно по этой причине для связи материнских плат между собой применяется оборудование локальных сетей, построенное по другим, нежели общая шина, принципам. В сетевом канале число проводов сведено к минимуму, правила, по которым эти провода меняют свое состояние, предельно упрощены, требования к надежности снижены за счет контрольного суммирования передаваемых данных и повторения передачи сбойных пакетов. Это позволяет резко снизить требования к качеству физической реализации и, как следствие, использовать кабельные соединения, т. е. выводить канал за пределы материнской платы. Платой за такую адаптацию формы представления передаваемых данных к коммуникационным нуждам являются громадные, по сравнению с общими шинами, задержки (латентность) передачи данных. Преобразование передаваемых данных в форму сетевых пакетов происходит во многом программно, а в остальном — с помощью сравнительно медленного оборудования сетевой карты, и все это требует в десятки, в сотни раз большего времени, чем акт срабатывания общей шины.

Собственно, в этом различии между внутренней и внешней коммуникационными магистралями и кроется причина как отсутствия общей памяти в большинстве современных суперкомпьютеров, так и исключительно высокой цены тех из них, в которых общая память все же предусмотрена. Ведь именно высокая латентность, органически присущая сетям, мешает организации на их базе общей памяти в пределах суперкомпьютера.

По мере роста рабочих частот и разрядности процессоров требования к качеству физической реализации, накладываемые

общими шинами, становились все более и более обременительными даже для проектирования и изготовления материнских плат в условиях массового производства. У разработчиков материнских плат появилось желание отказаться от общих шин, перевести внутренние коммуникации на сетевую технологию. Для того чтобы не потерять при этом производительности, необходимо было научиться выполнять сборку и разборку сетевых пакетов аппаратно и очень быстро, т. е. снабдить каждую крупную микросхему, начиная с процессора, «встроенной сетевой картой» и «встроенным сетевым процессором».

По мере роста числа транзисторов на кристалле такая затрата части кристалла на коммуникационные нужды стала не просто возможной, а даже не очень обременительной. Сегодня переход на сетевые технологии интеграции материнских плат (Hypertransport [8,42], PCI Express [7,41]) состоялся. Технологии коммуникаций внутри материнской платы стали очень похожими на технологии внешних коммуникаций. В частности, технологии внутренних коммуникаций резко упростились в части физической реализации, и стали допускать кабельные соединения на метровых расстояниях.

Сформулируем несколько тезисов о том, что это в действительности означает для разработчиков суперкомпьютеров.

1. Отныне объединяемые в кластер узлы могут быть связаны не намного менее тесно, чем функциональные устройства в пределах материнской платы. В частности, процессоры кластера могут напрямую адресовать «чужую» память и «чужие» внешние устройства. Различие во времени доступа к своей и чужой памяти полностью не исчезает, но сокращается на порядки по сравнению с традиционными MPP-системами.

2. Следствием такой унификации внутреннего и внешнего коммуникационного оборудования становится размывание границы узла в кластере. Кластер, т. е. многопроцессорный вычислитель, построенный из стандартных системных блоков массового выпуска, приобретает все положительные свойства NUMA-сервера. При этом и объем коммуникационного оборудования, и цена такого решения не больше, а меньше, чем в случае традиционного кластера.

3. В результате пропадает существовавшая на протяжении 20 лет необходимость отказываться от общей памяти для снижения стоимости суперкомпьютера — реализация общей памяти теперь практически ничего не стоит, и причины для использования

технологий параллельного программирования, специально рассчитанных на ее отсутствие, пропадают.

4. Замена преобладающего в категории доступных суперкомпьютеров класса коммуникационного оборудования (NUMA вместо сетей двусторонних обменов) — изменение само по себе вполне революционное. Оно не может не оказать очень сильного влияния на всю технологическую цепочку применения суперкомпьютеров, начиная с разработки параллельных алгоритмов и заканчивая разработкой трансляторов и библиотек. Изменение это в целом благотворное — как мы видели выше, технологии параллельного программирования для NUMA-систем гораздо более просты и удобны. Однако сам этот технологический переворот ни в коей мере не является давно обещанной читателю третьей суперкомпьютерной революцией. Он является ее важнейшей, но тем не менее довольно малой частью, в чем мы попробуем убедиться в подразд. 8.2.

Попыток перейти на сетевую технологию построения материнских плат было три, и в полной мере увенчалась успехом лишь третья из них.

*Первая попытка* — разработка стандарта Infiniband — несколько опередила свое время. Будучи разработан «на бумаге», стандарт этот не был принят промышленностью в качестве реальной технологии изготовления материнских плат, и в итоге эволюционировал в высокопроизводительную «кластерную» локальную сеть с тем же названием.

*Вторая попытка* — технология Hypertransport — была гораздо более успешной. Разработав эту технологию, фирма AMD применила ее на практике. По причинам, совершенно непонятным многим специалистам, авторы стандарта непросительно долго уклонялись от вывода канала Hypertransport непосредственно на разъем платы расширения. К моменту, когда это все же было сделано, т. е. был специфицирован формат разъема НТХ, время было упущено. В качестве нового периферийного разъема, принятого промышленностью в виде стандарта для подключения плат расширения по «сетевой» технологии, утвердился разъем конкурирующей технологии — PCI Express.

*Третья (удачная) попытка* полноценного перехода на сетевую технологию внутренних коммуникаций — разработанная фирмой Intel технология PCI Express. В настоящее время это индустриальный стандарт «сетевой» технологии построения материнских плат, включая разъемы плат расширения. Периферийные разъ-

емы материнских плат, построенных по технологии Hypertransport, обычно выполняются в виде «моста» между Hypertransport и PCI Express, представляющего собой специальную микросхему или часть чипсета. Это дает возможность устанавливать в такие компьютеры платы расширения формата PCI Express.

Переход производителей материнских плат массового выпуска на технологию PCI Express — уже совершившийся факт. Разработка технологий прямой коммутации линий PCI Express, минуя промежуточные звенья вроде сетевых плат Myrinet или Infiniband — вопрос времени, причем, по оценкам автора, времени совсем небольшого, максимум — одного-двух лет. Это не означает, что для такого перехода разработчикам суперкомпьютеров ничего не потребуется делать. Предстоит разработка специального оборудования и программного обеспечения. Однако разработка такого оборудования на сегодня — задача чисто инженерная. Характер и объем требуемых разработок будет рассмотрен далее. Эти разработки уже ведутся несколькими фирмами в разных странах, в том числе в России, и приводят к системам очень сходной архитектуры.

Отметим также, что речь не идет об «отмирании» локальных сетей как таковых. Локальные сети общего назначения, построенные по самым разным технологиям, самой разной производительности и стоимости, никуда не денутся, но перестанут быть преобладающим способом построения суперкомпьютеров по кластерной технологии.

## **8.2. Кризис фоннеймановской модели вычислительных приложений**

В подразделе 8.1 было показано, насколько революционное влияние оказало увеличение числа транзисторов на кристалле микропроцессора на коммуникационную среду многопроцессорных установок. Вернемся к обсуждению влияния этого фактора на вычислительное ядро процессора. Отчасти мы уже затрагивали этот вопрос в начале гл. 3, обсуждая происхождение вычислительной производительности и единицы ее измерения. Все сказанное там не может не привести нас к довольно неутешительному выводу. Архитектура фон Неймана при нынешнем количестве транзисторов на кристалле перестала быть вполне адекватной вычислительным приложениям. Такое количество транзи-

сторов, вернее логических элементов, для эффективного использования именно в вычислительных приложениях нуждается, как минимум, в дополнении и расширении фоннеймановских принципов, а может быть, и в принципиально иных формах организации.

С практической точки зрения нам важно установить взаимосвязь этого общего утверждения с правилом экономии сил на разработку суперкомпьютера (см. гл. 1). В самом деле, если возможность серьезного повышения производительности процессора любым способом, например путем перехода на альтернативные архитектуры, существует, то почему мы не видим ее материализованной в виде очередной модели Pentium или Opteron? Можно ли говорить о каких-то новых разработках вычислительных архитектур в рамках суперкомпьютерной отрасли, если гораздо более мощная отрасль комплектующих массового выпуска этим путем не идет?

Оба вопроса, в действительности, являются провокационными, т. е. поставленными не вполне корректно.

Начнем с того, что мы констатировали недостаточную адекватность фоннеймановской архитектуры именно **вычислительным** приложениям. Хорошо известно, что приложения вычислительного характера составляют малую долю приложений компьютеров массового выпуска, на потребности которых и ориентируются в основном изготовители микропроцессоров. Доля же именно суперкомпьютерных установок в потреблении выпускаемых сегодня стандартных микропроцессоров просто ничтожна — доли процента. Разработчики суперкомпьютеров, таким образом, решают просто другие задачи, чем разработчики стандартных микропроцессоров, и нет ничего невозможного в том, что для этого в сегодняшних условиях найдется другой, не применяемый разработчиками стандартных микропроцессоров инструмент.

Более того, само утверждение о том, что разработчики комплектующих для техники массового выпуска не ищут новых архитектур, не является верным. За 2006 — 2007 гг. на массовом рынке появились, как минимум, два изделия, построенных на не фоннеймановских процессорах:

— игровая приставка Sony PlayStation III. Ее «сердце» — микропроцессор IBM Cell [11], не являющийся ни фоннеймановским процессором, ни даже традиционным многоядерным процессором с симметричным доступом ядер к общей памяти;



— видеоадаптеры NVIDIA, построенные по технологии CUDA GPU [43]. Помимо применения по прямому назначению — в качестве видеоадаптера, эти платы могут использоваться как векторные вычислительные установки общего назначения, т. е. имеют в своем составе мощный не фоннеймановский вычислитель.

Кроме того, существует целый ряд разработок новых процессоров, выпускаемых малыми сериями, на которые у фирм-производителей мощных серверов имеются вполне конкретные планы.

Таким образом, мы обрисовали целое направление развития вычислительных архитектур, которое никак нельзя обойти своим вниманием, а именно — серийно выпускаемые промышленностью микропроцессоры нетрадиционной архитектуры, специально предназначенные для вычислительных приложений.

Направление это — важнейшее, но не единственное. Более того, возможно, в среднесрочной перспективе даже и не основное. Стратегически гораздо более интересным и важным представляется разработка реконфигурируемых вычислительных структур, специализированных под задачу или класс задач. Растущая неадекватность фоннеймановской архитектуры размерам кристалла, в сочетании с упомянутыми в подразд. 8.1 продвижениями на коммуникационном направлении, дают этому, довольно старому, подходу новый шанс (см. гл. 9).

Таким образом, предстоящий обзор нетрадиционных вычислительных архитектур будет охватывать два направления — обзор сравнительно новых, не существовавших ранее, но сегодня уже серийно выпускаемых изделий, и разработку собственного вычислительного оборудования с использованием технологий программируемой логики.

Наметив, таким образом, план изучения третьей суперкомпьютерной революции, перейдем к его осуществлению. На очереди более подробное знакомство с PCI Express.

# ПРОБЛЕМЫ ОБЪЕДИНЕНИЯ МАГИСТРАЛЕЙ PCI EXPRESS

В гл. 8 мы постарались показать, что переход производителей материнских плат на технологию PCI Express создает предпосылки для построения совершенно новой системы коммуникаций в кластере, обеспечивающей, в частности, общую память. Пока эти соображения носят довольно общий характер, и совершенно не очевидно, что преодоление неизбежных на этом пути технических трудностей не обесценит всю затею. Правила экономии сил на разработку суперкомпьютера никто не отменял. В этой главе будут перечислены конкретные технические проблемы и оценена сложность их преодоления. В том, что проблемы есть, сомневаться не приходится: например, еще в 2006 г. сама идея возможности кабельного удлинения канала PCI Express скептически воспринималась многими специалистами-практиками. Весьма популярным было следующее утверждение: «PCI Express — технология коммуникаций внутри материнской платы. Для объединения материнских плат между собой существует Infiniband». И пусть эта конкретная трудность оказалась, в итоге, надуманной, хотелось бы понимать, какие еще проблемы нас подстерегают.

С тех пор, как примерно 10 лет назад создатели суперкомпьютеров в большинстве своем сознательно отказались от собственных разработок оборудования, перейдя на кластерную технологию, в их среде утвердился весьма разумный обычай. Стало принято начинать всякое обсуждение планов создания собственного, самобытного оборудования с ответа на простой вопрос: «Если это и в самом деле так хорошо, то почему так не делают все остальные»? Невозможностью внятно ответить на этот вполне законный вопрос всякое такое обсуждение, как правило, и заканчивалось.

Любопытно отметить, что первый вариант текста данной главы был посвящен попытке дать развернутый, аргументированный ответ именно на этот «сложный» вопрос. Несколько недель спустя главу пришлось переписать. Об ответе на «сложный»

вопрос позаботилась сама жизнь. Хронология этого ответа целиком укладывается в рамки 2007 г. и выглядит довольно впечатляюще:

- в начале года официально опубликована спецификация кабельного варианта PCI Express [63], подтверждающая работоспособность по медному кабелю длиной до 10 м;

- к концу лета не очень широко известная в мире американская фирма OneStopSystems объявила о производстве коммуникационного оборудования для малоразмерных кластеров на базе прямой коммутации PCI Express [44];

- ближе к концу года с аналогичным предложением вышла на рынок фирма Dolphin Interconnect, знаменитая в прошлом как разработчик кластерной коммуникационной технологии SCI [13];

- наконец, в начале декабря фирма Samtec, признанный в мире «законодатель мод» в выпуске кабелей и разъемов, приступила к серийному производству кабельных удлинителей PCI Express, причем не для новых, так и не появившихся в широкой продаже, разъемов, предусмотренных кабельной спецификацией, а для стандартных ножевых разъемов, используемых в серийно выпускаемых материнских платах [64].

Завершающее событие в этом списке совершенно недвусмысленно свидетельствует о том, что «сложный» вопрос больше не стоит. Точнее, правильная формулировка «сложного» вопроса теперь звучит так: «Если так делают уже буквально все, чего же мы ждем»?

Технологии прямой коммутации каналов PCI Express, очевидно, оказались в «точке роста». В этом случае грань между использованием «самодельного» и покупного оборудования стирается — опытный образец собственной разработки зачастую оказывается законченным одновременно с появлением на рынке аналогичных изделий заводского выпуска. Из этого ни в коей мере не следует, что от разработки опытных образцов надо отказаться и просто подождать, пока все новые технические решения будут найдены другими. Все в точности наоборот. Разработчик, уклоняющийся от участия в общей гонке, рискует отстать настолько, что перестанет понимать, куда и зачем все бегут. Чтобы «пойти в магазин и просто купить, не разрабатывая» новое техническое решение, в котором кто-то удачно и недорого реализовал Ваши мысли, весьма желательно эти мысли иметь.

Словом, в сложившейся ситуации необходимо точно и конкретно разобраться, что именно происходит, какие решения нам

предлагает промышленность, как именно их можно использовать при построении суперкомпьютера с коммуникационной системой нового типа.

Чтобы понять масштаб проблемы, необходимо более подробно, чем при планировании монтажа и наладки традиционного кластера, углубиться во внутреннюю «кухню» PCI Express.

## 9.1. PCI Express на логическом уровне

В этом подразделе мы узнаем (или вспомним) о том, как выглядит коммуникационная магистраль PCI Express на уровне программы, в терминах процессорных команд.

На этом уровне PCI Express представляет собой частный случай, точнее, вариант реализации, более общей схемы — системы адресации PCI. Технология PCI была создана в начале 90-х годов XX в. как шинная, а не сетевая, но общие понятия системы адресации при переходе на сетевой принцип реализации решено было сохранить. Далее будем говорить о системе адресации PCI, не повторяя всякий раз, что речь идет не о старой, традиционной, шине PCI, а об ее современном, сетевом воплощении PCI Express [41].

Современные процессоры имеют обычно два адресных пространства: пространство адресов памяти и пространство адресов портов ввода-вывода (см. гл. 2).

Пространство адресов памяти складывается из всех возможных значений адресов, которые можно упоминать в командах процессора, подразумевающих обращение к памяти.

Пространство адресов ввода-вывода образуют все возможные значения номеров портов ввода-вывода, которые можно упоминать в командах ввода-вывода, если, конечно, таковые предусмотрены системой команд процессора.

Оба адресных пространства обычно бывают не сплошными, содержат «дырки», т. е. за конкретным адресом памяти или номером порта, вообще говоря, может и не быть адресуемого объекта.

Функциональным устройствам, установленным на материнской плате при ее изготовлении, соответствуют определенные области адресов, вообще говоря, в обоих адресных пространствах. Например, встроенный видеоадаптер «виден» процессору как целых три области: набор портов ввода-вывода с определенными

номерами, набор портов ввода-вывода, «замаскированных» под ячейки памяти с определенными номерами, и набор областей собственно памяти — например, видеопамять буфера развертки изображения. Аналогичным образом «видны» сетевые адаптеры, а также адаптеры подчиненных низкоскоростных шин, таких, как USB или Firewire.

Установка в периферийный разъем карты адаптера внешнего устройства приводит к тому, что в обоих, вообще говоря, адресных пространствах появляются новые адресуемые области. Например, это происходит при установке в соответствующий разъем съемного видеоадаптера или сетевой карты.

Наличие адресов позволяет устройствам (всем устройствам, не только процессору) обращаться друг к другу с коммуникационными запросами, такими, как «чтение» или «запись». Далее будем для простоты рассматривать только запрос «запись», поскольку он не подразумевает выдачу ответа. Также не будем далее рассматривать работу адресного пространства портов ввода-вывода. Во всех современных руководствах разработчикам новых устройств настоятельно не рекомендуется пользоваться им вообще, поддержка соответствующих возможностей рассматривается как дань традиции.

Представим все возможные в стандартной материнской плате виды запроса «запись».

1. Запись из процессора в системную память. Это происходит, если, например, процессор выполняет команду записи из регистра общего назначения по адресу памяти, являющемуся адресом в обычной оперативной памяти, а не в видеоадаптере или аналогичном устройстве. Такой вид взаимодействия в работе коммуникационной магистрали PCI вообще не отражается. С точки зрения коммуникационной магистрали, процессор и системная память — единое устройство, и обмен данными между ними — «внутренне дело» этого устройства.

2. Запись из процессора во внешнее устройство. Это происходит, если процессор выполняет команду записи из регистра общего назначения по адресу памяти, являющемуся, например, адресом ячейки буфера регенерации в видеоадаптере, или адресом управляющего регистра, скажем, сетевой карты, «замаскированного» под ячейку памяти. При этом со стороны процессора в коммуникационную магистраль PCI поступает запрос на запись, в котором содержится адрес записи и записываемое значение. Коммуникационная магистраль обязана доставить этот запрос

по указанному адресу. В шинной технологии понятие «доставить» подразумевает электрическое замыкание соответствующих проводов, в сетевой — реальную доставку пакета через сеть коммутации, с проверкой контрольной суммы и повторной передачей, если были сбои.

3. Запись из внешнего устройства в системную память. Это происходит, например, когда узел сети Infiniband получает от другого узла запрос на запись данных в системную память в режиме RDMA. В этом случае в коммуникационную магистраль PCI Express со стороны сетевого адаптера Infiniband поступает запрос на запись, в котором содержится адрес записи и массив записываемых значений. Коммуникационная магистраль обязана доставить этот запрос процессору, поскольку, как мы уже говорили, с ее точки зрения процессор и системная память — это одно большое устройство.

4. Запись из внешнего устройства во внешнее устройство. Это происходит, например, при записи по Infiniband в режиме RDMA, если в качестве адреса записи указан адрес в буфере регенерации видеоадаптера. Процедура аналогична рассмотренным выше.

Рассмотрим простой контрольный пример на понимание. Чем отличается в приведенных только что терминах односторонняя запись в «чужую» память с помощью локальной сети Infiniband от работы системы общей памяти, которую мы проектируем?

В случае односторонней записи из процессора А в процессор Б последовательность событий следующая.

Процессор А выполняет серию команд записи в память по адресам, являющимися адресами «замаскированных под память» управляющих регистров своего сетевого адаптера. Среди записываемых значений — адрес в собственной памяти, из которого надо взять отправляемое процессору Б значение, и длина этого значения в байтах.

В результате выполнения последней из этих команд записи сетевой адаптер процессора А посылает в коммуникационную магистраль материнской платы А запрос на чтение отправляемого значения по указанному в одном из регистров адресу.

Дождавшись ответа от процессора (в случае PCI Express ответ — это сетевой пакет специального вида), сетевой адаптер А формирует сетевой пакет формата Infiniband, типа «односторонняя запись», и посылает его по сети Infiniband сетевому адаптеру Б.

Получив и разобрав пакет, сетевой адаптер Б выполняет в коммуникационной магистрали материнской платы Б запрос на

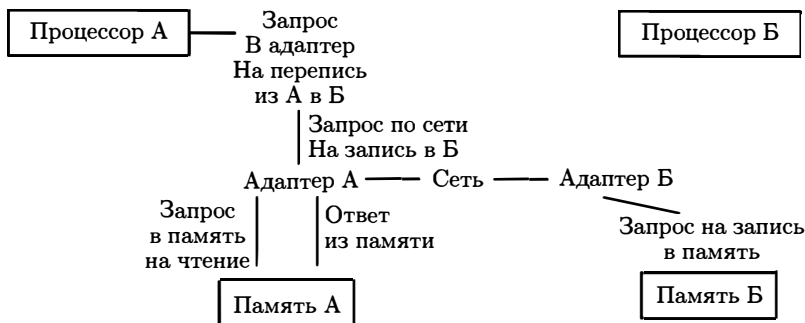


Рис. 9.1. Односторонняя запись в сети Infiniband

запись в процессор, в результате чего принятое из процессора А значение попадает, наконец, в системную память процессора Б (рис. 9.1).

В случае работы системы общей памяти, которую мы пытаемся спроектировать, коммуникационные магистрали процессоров А и Б должны быть связаны в единую сеть PCI Express. Следовательно, процессору А достаточно выполнить одну-единственную команду записи из регистра общего назначения в память по адресу, соответствующему ячейке памяти процессора Б. Все остальное коммуникационная магистраль сделает сама (рис. 9.2).

Здесь мы и сталкиваемся с основной проблемой, ради решения которой нам придется, в конечном итоге, разрабатывать собственное оборудование. Проблема состоит в отсутствии единого адресного пространства в объединенной коммуникационной магистрали материнских плат А и Б.

Мы пока не рассматривали вопрос о том, откуда именно в коммуникационной магистрали материнской платы берутся кон-

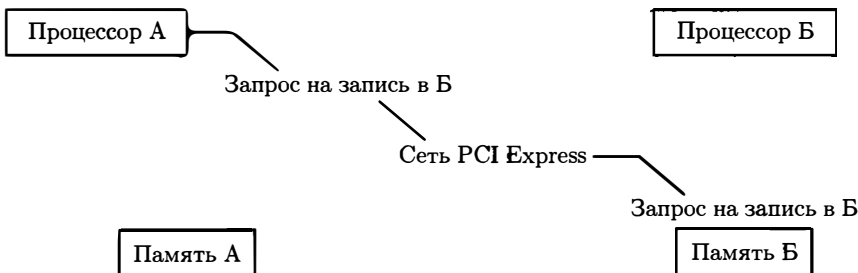


Рис. 9.2. Запись в системе общей памяти

кретные значения адресов для диапазонов памяти разных устройств. Но, откуда бы они ни брались, ясно, что в результате выполнения этой явно или неявно выполняемой процедуры адреса в материнских платах А и Б могут перекрываться. Например, почти наверняка перекрываются адреса системной памяти, т. е. в терминах коммуникационной магистрали адресные диапазоны устройств «процессор А» и «процессор Б». Как же коммуникационная магистраль сможет работать, если адреса не уникальны?

Для начала попытаемся решить эту проблему для себя, «на бумаге». Какую схему адресации чужой памяти нам бы хотелось иметь? Очевидно, нас вполне устраивала бы система на базе систематического сдвига адресов.

Пусть, например, с точки зрения коммуникационной магистрали А системная память процессора А начинается с нулевого адреса. Коммуникационная магистраль Б может сказать о своей системной памяти то же самое. Далее договоримся, например,

Адресное пространство памяти с точки зрения процессора А	Адресное пространство памяти с точки зрения процессора Б
0x00000000	0x00000000
Системная память процессора А, 2 Гб	Системная память процессора Б, 2 Гб
0x80000000	0x80000000
Не используемые адреса, адреса внешних устройств процессора А	Не используемые адреса, адреса внешних устройств процессора Б
0x100000000	0x100000000
Системная память процессора Б, доступная процессору А	Системная память процессора А, доступная процессору Б

Рис. 9.3. Общее адресное пространство на базе сдвига адресов (предполагается 64-разрядная адресация. Процессор Б видит системную память процессора А сдвинутой в адрес 0x100000000, и наоборот)



что процессор А должен «видеть» память процессора Б начинающейся с четвертого гигабайта, а не с нуля. Процессор Б пусть поступит симметрично (рис. 9.3).

Чтобы это реализовать, коммуникационные магистрали А и Б должны быть связаны не непосредственно, например кабелем от разъема PCI Express одной платы до аналогичного разъема другой, а через специальное устройство, преобразующее адреса в проходящих через него пакетах. Тогда доступ в «чужую» память выглядит так.

Процессор А, желая записать значение в память процессора Б, выполняет команду записи в память с адресом, указывающим куда-то за четвертый гигабайт. С точки зрения коммуникационной магистрали А, этот адрес соответствует нашему устройству-переходнику, и она доставляет ему соответствующий пакет. Получив этот пакет, устройство-переходник уменьшает указанный в пакете адрес записи на 4 Гб, и выдает его в коммуникационную магистраль Б. Коммуникационная магистраль Б доставляет этот пакет по модифицированному переходником адресу, который теперь уже является адресом системной памяти, что и требовалось.

Таким образом, необходимое нам устройство — переходник состоит из двух функционально одинаковых половинок, каждая из которых обращена в сторону одной из соединяемых коммуникационных магистралей. Каждая половинка является устройством на своей коммуникационной магистрали. Устройство реализует один или несколько диапазонов памяти и набор управляющих регистров, записанные значения в которых управляют конкретным способом преобразования адресов в пакетах, проходящих через устройство с соответствующей стороны. Преобразование, таким образом, не обязано быть симметричным: процессор А может видеть память процессора Б начинающейся с четвертого гигабайта, в то время как процессор Б видит память процессора А, начинающейся с пятого гигабайта. Важно лишь, что преобразование задается таблично, и должно быть явно настроено программно на стороне процессора — инициатора запросов.

Устройство, обладающее описанными только что свойствами, не впервые потребовалось разработчикам коммуникационных магистралей. Оно было придумано и применялось еще на досетевом этапе, в сложных шинных системах, и имеет название: *непрозрачный мост*[41]. Слово «непрозрачный» означает, что

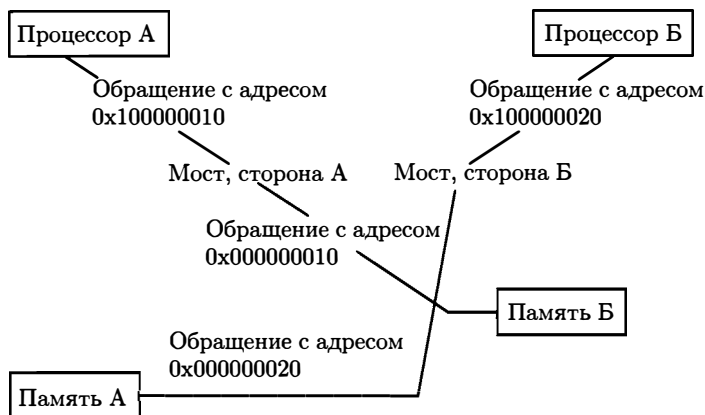


Рис. 9.4. Непрозрачный мост (для простоты показано симметричное преобразование адресов. Конкретный вид преобразования задается программной настройкой моста)

конкретный вид преобразования адресов должен быть явно настроен, поскольку, какова бы ни была процедура назначения адресов в объединяемых коммуникационных магистралях, никаких мостов и объединений в этой процедуре не предусмотрено, и, следовательно, настройка такого моста не может быть выполнена автоматически, при включении питания соответствующих компьютеров (рис. 9.4).

## 9.2. Несколько слов о реализации

Рассмотрев в общих чертах систему адресации, принятую в PCI Express, остановимся кратко на конкретном устройстве этой сети, т. е. на топологии, системе коммутации и назначении адресов.

PCI Express, как и все современные сети, является сетью коммутируемой. Ничего похожего на режим «коллизий», свойственный ранним версиям Ethernet, в ней, конечно же, нет. Коммуникационная магистраль представляет собой набор узлов, соединенных дуплексными каналами «точка — точка». Топология соединений — дерево. Корень дерева — особый узел, называемый root complex (процессор+системная память), листья дерева — устройства [41]. Вершины дерева, не являющиеся листьями, представляют собой коммутаторы, т. е. устройства, которые способны принять сетевой пакет по некоторому входу, проанализи-

ровать содержащийся в пакете адрес назначения и, в зависимости от того, чему равен адрес, выдать пакет в тот или иной выход.

Настройка коммутаторов происходит аппаратно при включении компьютера. В этот момент каждое из устройств — листьев дерева посылает в направлении корня специальные сетевые пакеты, в которых содержится информация о необходимых адресных диапазонах. Например, видеоадаптер мог бы послать такой пакет: «под буфер регенерации мне необходимо 8 мегабайт адресного пространства памяти, а под регистры управления — еще 512 байт». В результате совместной работы корневого узла и коммутаторов в обратном направлении поступают ответные сообщения, в которых содержатся конкретные адреса. Устройствам предлагается занять именно их. Например, ответ видеоадаптеру мог бы иметь вид: «Твои 8 Мбайт отныне начинаются с адреса такого-то, а твои 512 байт — с такого...». Настроившись таким образом, коммутаторы работают прозрачно, т. е. без всякого программного вмешательства в процесс коммутации пакетов. Если узел — лист дерева знает адрес получателя для пакета, который он хочет отправить, ему достаточно сформировать пакет и выдать его в сеть. Пакет найдет адресата.

Естественно, упомянутые нами выше устройства типа «непрозрачный мост» не пропускают через себя никаких управляющих пакетов, циркулирующих в магистральной части в процессе назначения адресов. Каждая из «половинок» непрозрачного моста на это время «притворяется» листом дерева, делает вид, что позади нее ничего нет.

В материнской плате вся необходимая система коммутации реализована в микросхемах чипсета и «южного моста». Пусть в нашем распоряжении имеется материнская плата с несколькими разъемами PCI Express. При установке в эти разъемы соответствующих плат расширения — например, сетевых или графических карт — эти устройства оказываются листьями дерева, включенными в систему коммутации. Эти устройства могут общаться друг с другом, с системной памятью и с прочими устройствами материнской платы. Если бы чипсет умел переводить разъем на материнской плате в режим непрозрачного моста, материнские платы можно было бы соединять между собой просто кабелями, а надлежащая настройка непрозрачных мостов позволила бы нам реализовать систему общей памяти способом, описанным выше. К сожалению, современные чипсеты этого, как правило, не умеют. Экстраполируя тот стремительный темп

роста кабельных технологий PCI Express, о котором мы говорили в начале главы, разумно предположить, что соответствующие чипсеты вот-вот станут нормой, но, пока этого не случилось, попробуем поискать выход из положения имеющимися на сегодня средствами (см. подразд. 9.3).

*Заключение* — несколько слов о масштабировании канала PCI Express по пропускной способности. Как известно, канал PCI Express бывает разной ширины — об этом легко догадаться хотя бы по внешнему виду разъемов на материнской плате. О ширине канала PCI Express мы пока еще не говорили. Понятие это, пожалуй, самое простое во всей иерархии соответствующих протоколов. Любой пакет PCI Express состоит из байтов. Передача байта всегда происходит побитно, в последовательном режиме. Если канал имеет ширину, равную одной линии, байты каждого пакета передаются по этой линии друг за другом. Если линий две, то четные байты можно передавать по одной линии, а следующие за ними нечетные — одновременно по другой. Если линий 4, 8 или более (стандарт допускает до 32), дисциплина передачи обобщается аналогичным образом (рис. 9.5).

Еще раз напомним, что линия канала PCI Express называется на английском языке не «line», а «lane», т. е., в дословном переводе, не «линия», а «дорожка». Таким образом, правильно говорить: «Четырех-лЕйновый разъем», а не «четырех-лАйновый». Впрочем, правильнее говорить: «Разъем шириной четыре».

Использование более широких каналов увеличивает скорость передачи данных и уменьшает латентность. Основные «размерные параметры» здесь таковы.

На каждый успешно переданный пакет приходится, в среднем, немного больше 20 байт накладного расхода [41], учитывая заголовок пакета как таковой и всевозможные «добавки» на нижнем уровне организации протокола. Время передачи байта по линии — 4 нс. Таким образом, скорости передачи данных примерно в 1Гб/с, принятые в современных кластерах на базе Infiniband, требуют каналов шириной 4, и уже в самом ближайшем

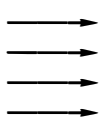
- 
- Линия 0: байты с номерами 0, 4, 8, 12, .....
  - Линия 1: байты с номерами 1, 5, 9, 13, .....
  - Линия 2: байты с номерами 2, 6, 10, 14, .....
  - Линия 3: байты с номерами 3, 7, 11, 15, .....

Рис 9.5. Канал PCI Express из четырех линий (линии работают одновременно)

будущем этот «минимум приличия» вырастет до 8. В 2007 г. был утвержден стандарт передачи на удвоенной частоте, но этот режим вряд ли будет хорошо работать по медному кабелю, так что потребуется переход на оптику.

### 9.3. Недостающие компоненты

Теперь мы наконец знаем о PCI Express вполне достаточно, чтобы оценить масштаб планируемой разработки оборудования.

Итак, для построения NUMA-кластера на основе технологии PCI Express из материнских плат крупносерийного выпуска не хватает всего трех вещей:

- кабеля;
- коммутатора;
- непрозрачного моста.

Кабельный вариант PCI Express был стандартизован, включая формат разъема, в начале 2007 г. [63]. На момент написания настоящего текста материнских плат и устройств с разъемами такого формата не появилось, по крайней мере, в широкой продаже, что удивляет. Это, однако, ни в коей мере не должно расстраивать разработчиков соответствующих решений для суперкомпьютеров. Как мы уже знаем, кабельные удлинители PCI Express, рассчитанные на стандартные разъемы выпускаемых серийно материнских плат, уже выпускаются серийно одним из гигантов мировой кабельной промышленности [64]. В течение всего 2007 г. небольшие фирмы по всему миру вполне успешно разрабатывали собственные кабельные решения, рассчитанные на стандартные разъемы. Как правило, при этом использовались специальные переходные адаптеры и кабели Infiniband [44].

Проблему коммутатора и непрозрачного моста также можно считать решенной. Коммутаторы PCI Express в виде отдельных микросхем выпускаются крупными сериями, в частности, таким крупнейшим производителем микросхем для построения периферийных устройств, как фирма PLX [45].

Большинство модификаций коммутаторов производства PLX допускают перевод одного из каналов в режим непрозрачного моста. Объединяя такие «элементарные коммутаторы» в каскад и настраивая непрозрачные мосты должным образом, можно строить масштабируемые коммутаторы PCI Express необходимого размера [45].

Конструктивно построение таких коммутаторов, по крайней мере их опытно-промышленных образцов, облегчается следующим фактором. Поскольку микросхемы сами по себе не являются конечным продуктом на рынке компьютеров и устройств для них, любой серьезный производитель сложных микросхем всеми силами старается пропагандировать свою продукцию среди производителей компьютеров и внешних устройств.

С этой целью выпускаются так называемые **RDK** (**R**apid **D**evelopment **K**it, Набор для быстрой (буквально — стремительной) разработки), т.е. готовые платы на базе соответствующей микросхемы, оснащенные более или менее стандартными разъемами. Цель выпуска таких плат состоит в том, чтобы дать инженерам — потенциальным покупателям соответствующих микросхем — возможность собрать действующую модель будущего изделия, минуя длительный и дорогостоящий процесс проектирования собственных печатных плат.

Фирма PLX — ведущий производитель микросхем коммутации PCI Express — не является в этом смысле исключением. Для каждой из микросхем коммутации выпускается конструктивно законченное изделие — коммутатор PCI Express в виде платы со стандартными ножевыми разъемами. Такие платы можно устанавливать в материнскую плату в качестве устройства, вставлять друг в друга, а также соединять кабелями, например, производства Samtec, как друг с другом, так и с материнскими платами. Аналогичные платы RDK выпускаются и другими производителями микросхем коммутации PCI Express. Стоимость RDK обычно выше стоимости аналогичного готового изделия серийного выпуска, но в комплект поставки входит программное обеспечение, а также набор технических руководств по проектированию собственных плат на базе использованной в RDK микросхемы.

Суммируя изложенное в этой главе, легко увидеть, что наш план построения новой коммуникационной среды находится в полном соответствии с правилом экономии сил на разработку суперкомпьютера. Буквально все необходимые детали имеются в наличии, их надо только грамотно соединить. Обещанная «разработка оборудования» свелась фактически к так хорошо освоенной нами за 10 лет системной интеграции.

Впрочем, как будет показано далее, от самостоятельной разработки оборудования на уровне логического проектирования нам все равно не уйти. Причем вновь разрабатываемое оборудо-

вание будет гораздо более сложным. В самом деле, ведь речь пойдет уже не о коммуникациях между вычислительными узлами кластера, а о внутреннем строении самого вычислительного узла.

## 9.4. Эскиз базового программного обеспечения

О том, как могло бы выглядеть программное обеспечение прикладного уровня для эскизно спроектированной нами NUMA-системы, мы много и подробно говорили выше. Возможна масса вариантов — например, переработка свободно распространяемой реализации UPC на уровне исходных текстов, реализация собственной библиотеки шаблонов в стиле Co-array Fortran, и даже реализация на базе NUMA сверхбыстрого варианта tcp/ip, с целью использования Cluster Open MP. Словом, работы здесь непочатый край. Но как мог бы выглядеть базовый уровень программного обеспечения для разработанных нами возможностей, например, на уровне системных запросов Linux?

В качестве простейшего варианта можно предложить следующий.

В каждом узле выделяются два диапазона адресов памяти — *локальная порция* и *адресное окно* (или много адресных окон, по числу узлов в кластере). Локальная порция — это часть собственной системной памяти, предоставляемая узлом в «общий пул», т. е. доступная для обращений из других узлов. Адресное окно — это диапазон адресов в коммуникационной магистрали PCI Express, обращение к которым приводит к адресации «чужих» локальных порций. Можно реализовать единое адресное окно, разделенное на блоки равного размера, каждый из которых соответствует локальной порции отдельного узла, или же много — по числу узлов — отдельных адресных окон. Все это достигается соответствующей настройкой системы модификации адресов в непрозрачных мостах. Для того чтобы пользовательская программа могла адресовать адресные окна и локальную порцию, придется написать несложный драйвер. Настройка пользовательской приписки страниц на адресные окна и локальную порцию происходит при обращении к этому драйверу с системным запросом mmap. Конечно, все сказанное здесь — не более чем довольно грубый эскиз, но автору настоящего текста приходилось проходить весь этот путь от начала до конца, по-

лучив в итоге пригодный для испытаний макет базового программного обеспечения.

*Заключение.* Реализованная предложенным способом общая память, к сожалению, не будет кэш-когерентной. В самом деле, несмотря на то, что средства обеспечения кэш-когерентности в протоколе PCI Express предусмотрены, обращения процессора к собственной локальной порции другим процессорам отследить не удастся: ведь они, как мы уже отмечали, происходят не через коммуникационную магистраль, а напрямую.

Краткий обзор коммуникационного аспекта третьей суперкомпьютерной революции нами завершен. Автор надеется, что общие принципы перехода на сетевую технологию интеграции материнских плат и общие тенденции реакции компьютерной промышленности на этот переход делают этот обзор более полезным, чем просто перечень остроумно подогнанных друг к другу «фокусов» с конкретным протоколом PCI Express. В оставшейся части книги хотелось бы рассмотреть наиболее захватывающий и наименее проработанный аспект третьей суперкомпьютерной революции — переход на не фоннеймановскую архитектуру вычислителя.



# АЛЬТЕРНАТИВНЫЕ АРХИТЕКТУРЫ ВЫЧИСЛИТЕЛЯ

В гл. 9 нам чудом удалось ослабить мертвую хватку правила экономии сил, убедив самих себя в том, что совсем незначительные вложения сил и средств в собственные разработки коммуникационного оборудования дадут нам очень и очень значительные преимущества в характеристиках готового изделия. И все же настоящий поединок с правилом экономии сил нам предстоит только сейчас. В самом деле, сборка коммутатора PCI Express из готовых плат и кабелей или даже перепроектирование таких плат по готовому образцу — ничто по сравнению с изготовлением — или хотя бы даже просто использованием — вычислителя альтернативной архитектуры. Пусть чудо совершилось, и на нашем столе стоит системный блок с материнской платой, в которой вместо процессора Pentium или Power PC — систолический массив. Что мы будем делать с этим компьютером? Где Linux или Windows, где трансляторы, как подключиться к сети? «Масштаб бедствия» очевиден и не поддается измерению. Хотелось бы сразу сделать важнейшее замечание и заодно «перекинуть мост» к предыдущей главе.

Представляется совершенно очевидным, что ни о какой **замене** фоннеймановского процессора вычислителем альтернативной архитектуры речь в среднесрочной перспективе идти не может. Говорить можно лишь о **добавлении** к системе, построенной на базе связанных коммуникационной средой фоннеймановских процессоров, некоторого количества **сопроцессоров**, имеющих альтернативную архитектуру. Причин тому две. Первая — совершенно очевидная — была упомянута только что. Объем сервисных, в самом широком смысле этого слова, функций традиционного процессора настолько громаден, что любая попытка «быстренько повторить» все это на принципиально новой базе представляется технологической авантюрой. Но это — не главное. Главное — в том, что прикладные программисты привыкли к фоннеймановскому программированию почти 60 лет, и не знают ничего другого. Любая попытка решения реальной

вычислительной задачи на альтернативной архитектуре может закончиться частичным или полным провалом. Архитектура может оказаться негодной — пусть даже для одной — единственной задачи. С фоннеймановской архитектурой такого произойти не может — за 60 лет мы научились делать на ней все, и даже немножко больше. Значит, необходимо предусмотреть возможность отступления на хорошо подготовленные позиции. **Любой вновь создаваемый суперкомпьютер общего назначения должен быть хорошо — по современным ему меркам — оснащен традиционной вычислительной мощностью**, хотя, по возможности, не только ей. У программиста должна быть возможность, пользуясь как всем системным сервисом, наработанным за десятилетия, так и собственным алгоритмическим багажом, гибко делить работу между традиционными процессорами и альтернативными вычислителями, т. е. суперкомпьютер должен быть **комбинированным**. (З а м е ч а н и е. В современной англоязычной литературе на эту тему применяется слово «hybrid» — «гибридный». Автору этот термин не представляется удачным, по крайней мере, в русскоязычном варианте, поскольку еще сравнительно недавно так называли вычислители, построенные с использованием блоков аналогового моделирования).

Как мы знаем на примере рассмотрения проблематики метакомпьютинга, для гибкого разделения работы мелкими порциями нужна очень хорошая коммуникационная система. Мало толку от специализированного вычислителя, если чтение исходных данных и преобразование их в пригодную для обработки на нем форму занимает у универсального процессора вдесятеро больше времени, чем само вычисление. Именно возможность «погрузить» вновь создаваемые сопроцессоры, наравне с функциональными устройствами стандартных материнских плат, во вновь созданную коммуникационную среду, быструю и низколатентную, и делает разговор о сопроцессорах столь актуальным именно сегодня.

Как отмечалось в гл. 8, к проблематике сопроцессоров с альтернативной архитектурой можно подходить с двух сторон: рассматривать решения, которые промышленность компьютерных комплектующих серийного выпуска уже предлагает специально для приложений вычислительного характера, или готова вот-вот предложить; пытаться строить проблемно-ориентированные вычислительные структуры на базе программируемой логики, при-

влекая к этому увлекательному процессу прикладных программистов. Начнем с первого подхода. Проанализируем кратко основные известные на сегодняшний день универсальные не фоннеймановские архитектуры вычислителей, а также доступные для приобретения изделия на их основе.

## **10.1. Универсальные не фоннеймановские архитектуры**

На протяжении 2007 г. это направление переживало своего рода бум. Пожалуй, впервые в истории высокопроизводительных вычислений было начато строительство крупнейшего (десятки тысяч процессорных ядер) суперкомпьютера с использованием не фоннеймановских ускорителей. Речь идет о суперкомпьютере RoadRunner, который IBM строит по заказу исследовательского центра в Лос-Аламосе, США [11]. Судя по впечатлениям участников самых различных выставок и конференций, состоявшихся в 2007 г., специализированные ускорители стали «вдруг» очень популярными среди разработчиков новых суперкомпьютеров. Рассмотрение этого направления весьма актуально с практической точки зрения, поскольку речь идет об изделиях действительно массового выпуска, вполне доступных тем, кто строит малые и средние параллельные вычислительные системы.

### **10.1.1. Векторный процессор**

**Векторный процессор** — самая старая, известная и простая в применении альтернативная архитектура. Расширениями такого рода снабжались традиционные процессоры от Cray-1 до современных версий Pentium и PowerPC. Расширение состоит в добавлении к традиционной системе команд набора дополнительных — векторных — команд, таких, например, как команда «сложить покомпонентно два вектора чисел». Чем длиннее обрабатываемые векторы, тем больше скоростной выигрыш. «Векторные вставки» современных процессоров общего назначения оперируют векторами чисел с плавающей точкой длиной 4 [7, 11]. Cray-1 «умел» складывать одной командой десятки пар чисел [3, 4]. Математики давно привыкли к этой возможности и интенсивно ее используют, иногда даже не подозревая об этом, по-

сколько современные трансляторы успешно «векторизуют» программы без посторонней помощи.

За счет чего достигается выигрыш быстрогодействия при использовании векторного процессора? Станный вопрос, подумает читатель, конечно, за счет одновременного выполнения многих арифметических операций многими арифметическими устройствами. И будет неправ, точнее, не вполне прав.

Конечно, параллельное выполнение многих арифметических операций над многими парами чисел дает пропорциональный скоростной выигрыш, но источников ускорения счета при применении векторного процессора существует, как минимум, еще два. Первый — в экономии работы по вычислению адресов обрабатываемых чисел, пропорциональной длине вектора. Как отмечалось в гл. 3, современные процессоры зачастую тратят время не столько на вычисления, полезные с точки зрения прикладного программиста, сколько на вычисление адресов тех величин, с которыми требуется эти вычисления выполнить. Векторный процессор вычисляет адреса операндов своих команд не для каждой пары операндов, а один раз для вектора максимально допустимой длины. Это — важный источник экономии. По крайней мере, сама запись программы в векторной форме, явная или неявная, уже способствует тому, чтобы потеря такого рода было как можно меньше. Но гораздо важнее второй источник экономии. Чтобы параллельно выполнять много арифметических операций над многими парами чисел, эти числа надо уметь параллельно доставать из памяти и параллельно же в нее записывать, т.е. векторный вычислитель должен быть оснащен «векторной памятью». В современных процессорах общего назначения «векторной памяти» нет, и именно этим объясняется весьма скромная длина вектора в их «векторных вставках». В Cray-1 возможность оперировать векторами гораздо большей длины обеспечивалась наличием специальных векторных регистров, т.е. отдельной векторной памяти небольшого объема. Ключом к успеху векторизации приложения считалось максимальное использование так называемого зацепления регистров, т.е. как можно более длительная обработка данных, уже попавших в векторные регистры, без обращения к системной памяти [3,4].

Следует отметить, что реализация памяти, способной одновременно выдавать или принимать большое число значений из разных адресов, очень дорога и сложна на схемном уровне. Задача несколько упрощается и становится разрешимой, если реа-

лизуется возможность одновременно выдавать или принимать значения, расположенные в памяти подряд. Для векторного вычислителя требуется именно это. Такой частный случай памяти, способной одновременно выдать или принять много значений, называется *расслоенной памятью*. Она состоит из нескольких отдельных устройств памяти, число которых обычно равно степени двойки. Эти устройства памяти, называемые слоями, объединены таким образом, что в адресном пространстве объединенной памяти ячейки слоев чередуются циклически. Ясно, что при этом любые  $N$  значений, расположенных в объединенной памяти подряд, попадают в разные слои, если значение  $N$  не превосходит числа слоев. Именно расслоенная память и применяется в векторных вычислителях в качестве «векторной памяти». Идеальный векторный вычислитель должен иметь память, сравнимую по размеру с системной памятью универсального компьютера, полностью расслоенную, с числом слоев, равным числу имеющихся в вычислителе арифметических устройств (рис. 10.1).

Векторный вычислитель не обязательно реализуется как традиционный процессор, к которому добавлены векторные команды. Известный по ряду классических старых разработок способ реализации — SIMD-машина. В таком процессоре все команды — векторные, т. е. процессор представляет собой одно устройство управления и много арифметических устройств, одновременно выполняющих одну и ту же команду с разными данными.

В качестве примера выпускаемого серийно и вполне доступного на рынке векторного вычислителя с архитектурой SIMD и глубоко расслоенной памятью рассмотрим видеоадаптер компании NVIDIA, построенный по технологии CUDA GPU [43]. Эта техника стала впервые доступна пользователям в конце 2006 г.

Почему мы называли видеоадаптер векторным вычислителем, какая здесь связь?

Видеоадаптеры современных компьютеров массового выпуска сосредотачивают в себе громадную вычислительную мощность, необходимую для построения реалистичных изображений по

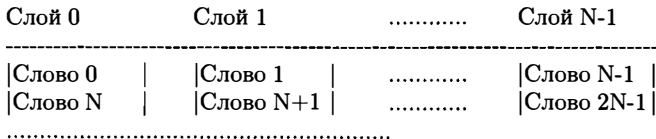


Рис. 10.1. Память из  $N$  слоев (каждый слой — отдельное устройство памяти)

описанию трехмерной сцены. Такой расчет представляет собой типичное вычислительно — емкое приложение. Поскольку приложение это постоянно выполняется в процессе работы компьютера, и всегда хочется, чтобы оно выполнялось побыстрее, его стараются реализовать не программно, на универсальном процессоре, а аппаратно, на специальном «графическом процессоре» в составе видеоадаптера. Графический процессор работает гораздо быстрее универсального, но система команд его настолько специализирована под графику, что практически ни для чего другого не подходит.

Желание все же использовать вычислительную мощность графического процессора не только для формирования картинки на экране монитора, но и для вычислительной работы общего вида, появилось у математиков не вчера. Периодически делались попытки, иногда — успешные, свести не имеющую отношения к графике вычислительную работу к последовательности таких действий, которые встречаются в графике, чтобы поручить эту работу графическому процессору.

В конце концов, разработчики видеоадаптеров пошли на встречу таким попыткам, избавив энтузиастов графических вычислений от утомительной задачи «добывания творога из вареников в промышленных масштабах». Было решено сделать графический процессор гораздо более универсальным, обобщить его архитектуру, чтобы она, став «более программируемой», годилась не только для графики. В результате получился GPGPU — графический процессор с архитектурой общего вида, пригодной и для универсального программирования. Архитектура получила название CUDA (Compute Unified Device Architecture, Унифицированная вычислительная архитектура устройства).

С точки зрения универсального процессора материнской платы, CUDA GPU — это сопроцессор. В коммуникационную магистраль материнской платы он включается через «графический» разъем PCI Express шириной 16, что дает пиковую пропускную способность около 4 Гб/с, на прием и на передачу одновременно. Подключение векторного ускорителя к универсальному процессору по более узкому каналу вряд ли имело бы большой смысл.

Сопроцессор имеет собственную память размером в сотни мегабайт и программируемый универсальным процессором канал DMA для передачи программ и данных между памятью универсального процессора и памятью сопроцессора.

В составе сопроцессора имеется 4, 12 или 16, в зависимости от модели, SIMD-процессоров. Каждый SIMD-процессор имеет 8 арифметических устройств, каждое — со своим набором скалярных регистров общего назначения, и 16 кб векторной памяти, расслоенной на 16 слоев. Арифметические устройства — 32-рядные, используют числа с плавающей точкой и стандартные режимы округления. Память, доступная для обмена данными с универсальным процессором, адресуется всеми SIMD-процессорами, но не является расслоенной, и доступ в нее примерно в 100 раз более медленный, чем в расслоенную память SIMD-процессора.

Для программирования разработан специальный транслятор языка С, и довольно нетривиальная программистская модель, не вполне непосредственно отображаемая на аппаратуру. Она прекрасно документирована [43], и рассматривать ее здесь мы не будем.

Важнейшим выводом из этого предельно краткого обзора является то, что теперь мы можем считать доказанной «конструктивную теорему существования». Задача построить универсальный вычислитель, совсем не похожий по архитектуре на процессоры общего назначения, была не только поставлена промышленностью компьютерных изделий массового выпуска, но и успешно решена. Главный аргумент против перехода на новые архитектуры, состоящий в том, что «если бы более быстрый процессор был возможен, он назывался бы Pentium», опровергнут экспериментально. Альтернативные универсальные архитектуры возможны, реально существуют, и их следует изучать.

### 10.1.2. Процессор IBM Cell

Микропроцессор IBM Cell интересен для нас тем, что является изделием заведомо массового выпуска: на его базе построена игровая приставка Sony PlayStation III. Архитектура была разработана совместно специалистами Sony, Toshiba и IBM [65].

Микропроцессор представляет собой гетерогенный многопроцессорный вычислитель — своего рода «суперкомпьютер на кристалле». В его состав входит универсальный процессор PowerPC и восемь процессорных элементов. Все эти девять процессорных ядер объединены высокоскоростной общей шиной, к которой также подключены два контроллера внешней памяти с пиковой скоростью передачи данных 12,8 Гб/с каждый, и два

контроллера ввода-вывода со скоростью 38,4 Гб/с каждый. Рабочая частота — около 4 ГГц.

Каждый процессорный элемент — это векторный процессор с набором из 128 регистров разрядности 128, с четырьмя арифметическими устройствами с плавающей точкой одинарной точности, и с четырьмя арифметическими устройствами целочисленной арифметики. Кроме того, в состав процессорного элемента входит 256 кб локальной памяти. На общую шину кристалла процессорный элемент выходит через контроллер DMA, способный обмениваться данными между локальной памятью и общей шиной.

Локальная память процессорного элемента не кэшируется, но сама работает со скоростью кэш. Кэш-памяти как таковой в процессорном элементе просто нет.

Внешняя память, подключаемая к кристаллу, адресуется универсальным процессором, но не процессорными элементами. Процессорные элементы могут лишь выполнять односторонние обмены между своей локальной памятью и внешней памятью, подключенной к кристаллу.

Посмотрев на эту архитектуру еще раз, нельзя не поразиться ее простоте. Это — классическая MPP-система, с управляющей машиной и восемью вычислительными узлами, которые связаны сетью односторонних обменов — только все это находится внутри кристалла. Правда, «вычислительные узлы» оснащены векторными расширениями и объемом регистровой памяти, достаточным для «зацепления операций» в стиле Cray-1 [3,4].

### 10.1.3. Мультитредовая архитектура

Исследования в области мультитредовой архитектуры имеют давнюю историю, и на их результаты возлагаются серьезные надежды. Пожалуй, на сегодняшний день это направление можно считать главным в разработке универсальных вычислительных архитектур, отличных от традиционных. Подробное изложение проблематики мультитредовых (multithreaded) архитектур нашей целью не является. Ограничимся обзором основных идей.

Как известно, процесс выполнения команд в процессоре включает в себя работу различных функциональных устройств. При вычислении индексов используются целочисленные арифметические устройства, при выполнении арифметических операций с плавающей точкой — устройства арифметики с плавающей точ-



кой. Также происходят обращения к памяти, в среднем — в десятки и сотни раз более медленные, чем обработка полученных из памяти значений. В работающих последовательно процессорах прошлого почти каждое такое функциональное устройство в составе процессора практически все время простаивало. Сначала вычислялся индекс, затем следовало обращение в память, затем — арифметическая операция с плавающей точкой, и т. д. Для возможно более полной загрузки функциональных устройств со временем стали применяться специальные приемы, в первую очередь — конвейерное выполнение команд с разбивкой их на стадии, а также кэш-память. Это позволило значительно повысить среднюю загрузку функциональных устройств процессора, и, как следствие — его быстродействие. Впрочем, значительный дисбаланс между временем доступа в память и временем обработки данных сохранился, несмотря на интенсивное использование кэш-памяти.

Мультитрединг (multithreading) представляет собой попытку гораздо более радикального решения этой проблемы, чем это пытались делать ранее. Представим себе, что процессор выполняет много процессов в псевдопараллельном режиме, но переключение процессора между такими процессами выполняется аппаратно и очень быстро, в идеале — на каждом такте рабочей частоты процессора, в том числе — «внутри» отдельных процессорных команд. Тогда, если среди ожидающих очереди на выполнение процессов есть хотя бы один, которому уже пора выполнять арифметическую операцию с плавающей точкой — занятость соответствующего устройства на следующем такте обеспечена. Если некоторый процесс надолго «застрял» на обращении к памяти — ничего страшного, те функциональные устройства, которые он собирается использовать, но пока не может, будут использованы другими процессами. Подобрав относительную численность функциональных устройств в соответствии с их относительным быстродействием, можно обеспечить полную загрузку всего оборудования процессора без «узких мест».

Работоспособность этой схемы, представленной в предельно общем виде, зависит от двух факторов:

- реализация в процессоре, на аппаратном уровне, очень быстрого и «умного» планировщика процессов;
- возможность разбиения написанной прикладным программистом программы на очень большое количество параллельных

процессов, взаимодействующих в модели SMP. Речь идет не о десятках, а, скорее, о десятках тысяч процессов.

Процессоры, ориентированные на эту технологию, разрабатываются и выпускаются, по крайней мере, опытными партиями, многими серьезными фирмами, такими, например, как Sun и Cray [20,46]. Тем не менее, работы по практическому применению мультитредовой архитектуры развивались не очень быстро. Однако, в последнее время процесс, похоже, сдвинулся с мертвой точки. Многие элементы мультитредовой архитектуры использованы в кратко рассмотренной выше архитектуре CUDA. В частности, исключительно высокую (200 тактов) латентность обращения к общей памяти вычислителя удается скрыть именно рассмотренным только что способом, с использованием встроенного мультитредового планировщика, при разбиении приложения на многие сотни легковесных процессов. Реализация мультитредовой архитектуры в составе изделия действительно массового выпуска наверняка позволит в короткий срок вывести эту архитектуру на уровень гораздо более широкого практического применения, чем раньше.

В русскоязычной терминологии иногда можно встретить термин «многопоточковая архитектура». Этот термин не представляется вполне удачным для обозначения архитектуры, кратко рассмотренной нами только что. Как мы видели, исходный англоязычный термин непосредственно происходит от понятия «thread», т. е. легковесный процесс, в том самом смысле, в котором мы уже упоминали его в подразд. 5.6. «Потоками» легковесные процессы обычно не называют, но термин «многолегковеснопроцессная архитектура» вряд ли имеет право на существование. Поэтому чаще всего говорят «мультитредовая архитектура».

## **10.2. Проблемно-ориентированные вычислители**

В подразд. 10.1 было приведено несколько примеров более или менее универсальных вычислительных архитектур, призванных так или иначе преодолеть недостаточную адекватность традиционных процессоров приложениям вычислительного характера. Однако к этой проблеме можно подойти и с другой стороны. Новый, не похожий на стандартный процессор, способ организации транзисторов в вычислительную систему вовсе не

обязан быть универсальным. Теоретически вполне можно представить себе специализированный вычислитель, реализующий тот или иной типовой алгоритм непосредственно на аппаратном уровне.

Ответ на совершенно естественный вопрос о том, сколько это будет стоить, как именно делаться, и сколько времени займет, пока отложим — впрочем, совсем ненадолго. Сначала постараемся понять, за счет чего в принципе такой вычислитель мог бы работать быстрее, чем программа, реализующая тот же алгоритм на универсальном процессоре. Для этого вспомним один из обсуждавшихся выше источник ускорения векторного процессора по сравнению с универсальным.

Векторный процессор экономит время, в частности, за счет того, что не вычисляет адреса для каждого из элементов обрабатываемых векторов. Тот факт, что обрабатываемые значения, с которыми требуется выполнить некоторые однотипные действия, расположены в памяти подряд, непосредственно учтен в схеме самого процессора, и программе для векторного процессора уже нет нужды «учитывать» это, выполняя цикл прохода по элементам массивов. Соответствующие циклы «переместились» из программы в структуру схемы. При этом схема не выполняет их быстрее, чем программа на универсальном процессоре — она просто устроена так, что выполнять в этом месте алгоритма стало нечего. Таким образом, векторный процессор уже является специализированным вычислителем. Он хорошо выполняет те приложения, которые могут быть записаны в векторной форме, и довольно бесполезен для приложений, которые так записать нельзя в принципе.

Однако специализация вычислителя под всевозможные приложения, допускающие векторизацию — это специализация довольно слабая. Из программы, написанной для традиционного процессора, при переходе на векторный процессор «перемещается в структуру схемы» лишь малая часть кода, только циклы прохода по векторам.

Следуя далее этим путем, т. е. перекладывая в структуру схемы все большую и большую часть программного кода, можно получить схему, которая вообще не нуждается в программе, и тратит время только на полезные, с точки зрения прикладного программиста, вычисления. Если эта схема еще и обладает внутренним параллелизмом, она имеет шанс работать в сотни раз быстрее программы для универсального процессора при тех же

рабочих частотах элементной базы. Понятия пикового и реально достижимого быстродействия для такой схемы совпадают.

Правда, выполнять она будет всегда ровно один алгоритм, тот, который в нее «зашит». «Перезашить» на другую программу ее нельзя, поскольку программы в ней нет. Для другого алгоритма придется построить другую схему.

В этом подходе можно увидеть некоторую аналогию со сравнением скорости выполнения программы, написанной на интерпретируемом и компилируемом языке. В самом деле, программа на классическом Бейсике может выполняться в двух вариантах: в интерпретируемом и в скомпилированном. Во втором случае компиляция выполняется заранее, что экономит время выполнения в разы. В описываемом нами подходе предлагается выполнить «более глубокую компиляцию»: не в команды универсального процессора, а в структуру схемы. Иными словами, убрать еще один уровень интерпретации.

Подход этот только кажется «нетрадиционным». В действительности, он старше, чем само понятие процессора.

На заре вычислительной техники почти все те установки, которые сегодня принято называть «прообразами современных компьютеров», были, в действительности, машинами одной задачи. Например, машиной для обработки данных переписи населения, или чем-то подобным. Совершившийся же в середине XX в. переход от таких специализированных машин к универсальным процессорам представлял собой совершенно нетривиальное новшество, которое нелегко было придумать, т. е. «подлинно нетрадиционным» является универсальный процессор.

Уже на современном этапе компьютерной истории периодически возникали такие задачи по обработке данных, которые оправдывали построение для них специальных вычислительных устройств.

Геофизикам 70-х годов XX в. наверняка хорошо знакомы «Фурье-процессоры». Гораздо более близкий большинству читателей пример — графические процессоры, о которых мы уже говорили выше. Задолго до того, как рост размера кристалла СБИС позволил превратить их в универсальные, программируемые вычислительные структуры, вроде CUDA GPU, уже существовали специализированные, непригодные для вычислений общего вида, «графические машины», оформленные как видеоадаптеры, или же интегрированные в особо мощные рабочие станции. Еще 20 — 30 лет назад целиком на специализированных под графику вычислителях строились

летные тренажеры — с генерацией в режиме реального времени необходимых изображений тогдашние универсальные компьютеры и близко не могли справиться.

Таким образом, мы располагаем, как минимум, одним примером реального вычислительного приложения, которое предлагаемым путем удалось ускорить во много раз. Это приложение — построение реалистичного изображения по описанию трехмерной сцены.

Теперь самое время вспомнить о вопросе цены и времени реализации, который мы уже упоминали выше, но отложили «на потом».

Видеоадаптеры выпускаются тиражами, как минимум, в сотни тысяч — можно постараться, вложить в проектирование много времени и сил. Летные тренажеры прошлого строились годами, и даже их физическое изготовление, когда все спроектировано, занимало не месяц и не два. При этом работали десятки, а то и сотни, человек. Мы же ищем замену технологии рутинной разработки программ, простой, привычной процедуре «написание — трансляция — запуск — поиск ошибок — исправление». То, что один человек делает в течение часов и дней, предлагается заменить тем, на что десятки и сотни людей тратят месяцы и годы. Не слишком ли, мягко говоря, смелое предложение?

Вопрос, как всегда, в соотношении затрат и результатов. Ожидаемый результат мы примерно себе представили. Осталось оценить затраты. Для этого нам надо вспомнить, хотя бы в общих чертах, как вообще проектируются вычислительные устройства. Откуда, например, берется схема нового процессора, в каком виде она пишется или рисуется, как ее отлаживают и исправляют. Дело в том, что соответствующие технологии в последние десятилетия вовсе не стояли на месте. С появлением примерно 25 лет назад технологий программируемой логики процесс создания цифровых электронных устройств ускорился на порядки, и сегодня уже стал сопоставимым по трудоемкости с обычным программированием. Процесс же физической реализации уже спроектированного устройства, по крайней мере в экспериментальном варианте, вообще занимает секунды и практически ничего не стоит. Схемотехникам-профессионалам это все досконально известно, но данная книга адресована в первую очередь вовсе не схемотехникам-профессионалам. Чтобы во всем этом разобраться, придется обрисовать в самых общих чертах процесс создания цифрового электронного устройства в его современном виде.

# КАК СОЗДАЮТСЯ ЦИФРОВЫЕ ЭЛЕКТРОННЫЕ УСТРОЙСТВА

При создании цифрового электронного устройства, как и при разработке программного приложения, используется вполне определенная устоявшаяся за многие годы система уровней иерархии представления схемы [47].

На самом нижнем уровне иерархии находится *транзистор*. Это аналоговый электронный прибор, функционирование которого описывается рядом плавных, совсем «не цифровых», зависимостей. Количество транзисторов в цифровых схемах таково, что понимание этих схем человеком непосредственно в терминах вольт-амперных характеристик транзисторов невозможно. Никому еще не удавалось понять, как работает процессор, проследившая непосредственно взаимные зависимости токов и напряжений составляющих его транзисторов, подобно тому, как радиолюбитель «разбирает схему» транзисторного приемника с двухкаскадным усилителем сигнала. Тем более, в этих терминах процессор или аналогичное по сложности цифровое устройство невозможно спроектировать.

По этой причине вводится следующий уровень иерархии — уровень логических элементов, или *вентилей*. Это тот уровень представления, на котором происходит переход от аналоговой модели к цифровой. Вводится понятие логического уровня сигнала, «логического нуля» и «логической единицы», и говорится, что в правильно спроектированной схеме сигнал в определенные моменты времени не будет принимать промежуточных значений. А что такое «правильно спроектированная схема»? Это схема, построенная из вентилях, т.е. совсем небольших электронных приборов, каждый из которых реализует некоторую функцию алгебры логики. Например, вентиль «И», вентиль «ИЛИ», вентиль «НЕ». Каждый вид вентиля строится из небольшого числа транзисторов (или диодов) определенным способом, и при дальнейшем рассмотрении схемы о «транзисторном происхождении» вентилях забывают. Схема рисуется как состоящая из вентилях, т.е. выполняется *логическое проектирование* схемы.

Из основных положений алгебры логики известно, что для построения любой функции алгебры логики путем суперпозиции логических элементов надо обладать **полным базисом**, т.е. некоторым набором видов таких элементов. В частности, набор из трех видов элементов: «И», «ИЛИ» и «НЕ», образует полный базис. Способы построения произвольной функции алгебры логики из таких элементов по ее таблице истинности просты и хорошо известны: например, канонический способ построения дизъюнктивной нормальной формы. Таким образом, располагая базисным набором вентилях, мы легко можем построить любую **комбинационную схему**, т.е. схему, выходы которой реализуют некоторую функцию алгебры логики от своих входов [47]. Некоторые примеры построения таких комбинационных схем, реально применяемых в сложных цифровых устройствах, приводятся в прил. 2.

Однако применяемые на практике сложные цифровые схемы редко бывают комбинационными. Значения выходов комбинационной схемы зависят от значений ее входов в данный момент времени, т.е. комбинационная схема «ничего не помнит», по определению не содержит в себе элементов памяти. Любая память, даже объемом в один-единственный бит, комбинационной схемой не является.

Для завершения построения «картины мира» логической схемотехники нам требуется ввести еще два связанных между собой понятия — понятие элемента памяти и понятие дискретного времени.

Простейшим элементом памяти, т.е. цифровой схемой, значения выходов которой зависят от значений ее входов в прошлом, является **RS-триггер**. Такой триггер получается, если соединить два вентиля «И» и два вентиля «НЕ» (рис. 11.1).

Входы этого триггера — инверсные сигналы установки (S) и сброса (R). Если оба они равны 1, т.е. не активны, то триггер сохраняет значение, которое имел ранее. Оно представлено в прямом виде на первом выходе и в инверсном — на втором. Если вход R равен 0, т.е. активен, а вход S — не активен, то триггер запоминает нуль. Если активен вход S, а вход R — нет, то триггер запоминает единицу. Наконец, если активны оба входа, то после их деактивации триггер будет вести себя, как аналоговая схема, т.е. в терминах логики (а не физики транзисторов, проводов и диодов, из которых выполнены вентили) о его последующем состоянии ничего сказать нельзя. Правильная схема, использующая

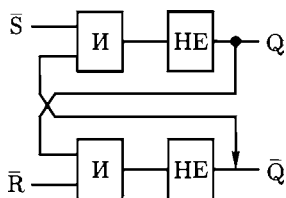


Рис. 11.1. RS-триггер (вертикальная черта над названием сигнала обозначает его инверсию: надчеркнутое «S» читается как «не S». Входы вентиля слева, выходы — справа)

такой триггер, должна быть построена так, чтобы эта ситуация никогда не возникала.

Этот простейший пример показывает, что схемы с последствием, т. е. со свойствами элементов памяти, в принципе также строятся из вентилях, как и комбинационные схемы. Однако возникает вопрос о том, что такое «прошлое» и «настоящее», как учесть время переходных процессов в сложных схемах. Ведь во время перехода вентилях из одного состояния в другое они уж точно не подчиняются законам логики. В это время совершенно неизбежны и промежуточные (не логические) уровни сигналов, и затухающие колебательные процессы, и много других неприятностей.

Для решения этих вопросов вводится понятие дискретного времени. В состав схемы включается специальное устройство — **генератор синхроимпульсов** определенной частоты. Система элементов памяти разного типа, построенных обычно на основе RS-триггера, но с добавлением дополнительных вентилях, строится таким образом, чтобы все передачи сигналов с выходов одних элементов памяти на входы других происходили, например, только в момент появления очередного синхроимпульса, как говорят, «по переднему фронту». Рабочая частота выбирается таким образом, чтобы за время от одного переднего фронта синхроимпульса до другого все переходные процессы в транзисторах и диодах, из которых сделаны вентилях, успели завершиться. Таким образом, время измеряется в синхроимпульсах. Прошлое — это время, после которого прошел хотя бы один синхроимпульс. Состояние схемы в промежутках между синхроимпульсами при логическом проектировании не рассматривается. Все это вместе позволяет говорить действительно о логическом проектировании, без привлечения каких-либо понятий физической реализации, вроде той же вольт-амперной характеристики транзистора.

В соответствии с вполне очевидными принципами структурного проектирования вводится следующий, еще более высокий,



уровень иерархии, а именно уровень *типовых логических узлов*. Самым очевидным примером такого узла является *регистр* — набор триггеров, управляемых и тактируемых совместно. Функционально регистр представляет собой отдельную ячейку памяти определенной разрядности. О проектировании в терминах типовых логических узлов иногда говорят как о проектировании «в регистрах».

Пока мы не видим ничего такого, что делало бы процесс проектирования цифровых электронных схем заметно более сложным, чем, например, системное программирование. Почему же в таком случае программирование издавна считается в чем-то более простой, а главное — осуществимой гораздо более быстро работой?

Попробуем представить себе, как разработка цифровой схемы происходит на практике. Для начала забудем об изделиях современной микроэлектроники, сам физический процесс изготовления которых подразумевает «монтаж» чуть ли не отдельных молекул. Поставим себя мысленно на место проектировщика конца 70-х годов XX в., когда типовые логические узлы были отдельными микросхемами, т.е. регистр можно было взять в руки, установить на плату и припаять обычным низковольтным паяльником. Приступая к разработке, например, процессора, схемотехник сначала рисовал его схему «в регистрах». В этот момент схема, скорее всего, содержала в себе примерно столько же чисто логических ошибок, сколько имеется в тексте только что написанной, но еще не отлаженной программы. Схему, как и программу, требуется отладить. Для этого схему собирали на макетной плате, т.е. на плате, в которой имеется очень много монтажных позиций для ножек микросхем, никак не связанных друг с другом. Установив на макетную плату все необходимые микросхемы, разработчик выполнял необходимые соединения обычным проводом с помощью паяльника. Схема готова и, конечно же, не работает.

Если сравнивать этот процесс с процессом разработки программы, в первую очередь бросается в глаза физическая трудоемкость изготовления опытного образца. Программу, в отличие от схемы, не надо паять. Если она существует в виде файла на диске, ее уже можно считать «физически реализованной». В действительности, главное отличие (и главная проблема) вовсе не в этом. В конце концов, в современных условиях «монтаж» внутренней структуры микросхемы все равно делает некоторый

автомат, и, скорее всего, он делает это довольно быстро, и без вмешательства человека. Но вернемся к нашему опытному образцу.

Главное его отличие от еще не отлаженной программы состоит в том, что он, вообще говоря, вовсе не ведет себя как цифровая, т. е. описываемая законами логики, схема. В самом деле из того, что два транзистора вместе ведут себя как клапан, а четыре клапана вместе — как триггер, вовсе не следует то, что аналогичные рассуждения верны для соединенных проводами тысяч и тысяч транзисторов. В такой большой системе присутствуют «просадки» сигналов и возрастание времени переходных процессов на длинных комбинационных цепочках, тепловые шумы, помехи, «наводки» и прочие эффекты, имеющие совершенно не логическое происхождение. Соединенные вместе в большом количестве, клапаны постоянно рискуют перестать быть клапанами, т. е. начать вести себя не логически, а электрически некорректно. Конечно, наш воображаемый разработчик стремился все это учесть, «разбавив» схему должным числом резисторов и емкостей. Но ведь и в расстановке этих совсем не логических элементов он, скорее всего, где-то ошибся — схема-то еще не отлажена!

Итак, разработчик установил, что схема ведет себя неправильно, и даже понял, как именно. Где ошибка — «в логике» или «в электричестве»? Понять это по внешним проявлениям ошибки зачастую практически невозможно. Искать любую ошибку приходится сразу по двум направлениям. В этом и заключается отличие нашего воображаемого схемотехника от программиста. Он и хотел бы мыслить исключительно на уровне логики, как того требует сложность задачи — да не может. Ему постоянно приходится быть «немножко радиомастером», т. е. держать в голове и логические, и электрические свойства своего изделия.

Именно эта принципиальная невозможность гарантированно разделить уровни иерархии, абстрагироваться от нижележащих и сосредоточиться исключительно на своем, и делала работу схемотехников такой трудоемкой, так отличной от работы программиста. В самом деле вряд ли можно даже представить себе, сколько времени у программиста заняла бы, например, реализация надежно работающего Интернет-приложения на заведомо неисправном процессоре.

Теперь попробуем мысленно вернуться в наши дни. Место макетной (или не макетной, а печатной, готовой к серийному

выпуску) платы заняла миниатюрная пластина кремния, припаять к которой дополнительный клапан довольно затруднительно. Запускать же процесс изготовления новой пластины при всякой необходимости разорвать одно соединение и сделать два новых — баснословно дорого. При этом «электрические» проблемы никуда не исчезли — они просто переместились на микроуровень, т. е. их стало гораздо труднее обнаруживать. Как следует поступить, чтобы разработка новых микросхем вообще стала возможной?

Выход был найден в 1983 г. после появления микросхем программируемой логики [48, 49]. Такая микросхема представляет собой, фактически, миниатюрную «макетную плату», на которой установлено громадное — порядка миллионов — количество клапанов. Клапаны сами по себе никак не связаны друг с другом, но их можно связать практически любым необходимым способом, загрузив карту необходимых связей в специальную *конфигурационную память* микросхемы. Переворот в логическом проектировании произошел тогда, когда эту память удалось сделать оперативной, т. е. не «прожигаемой» однократно, а перезаписываемой неограниченное число раз. Это сделало процесс физической реализации как угодно сложной схемы на кристалле программируемой логики столь же простым, быстрым и дешевым, как запуск исполняемого файла программы на универсальном процессоре.

Появление микросхем программируемой логики значительно облегчило разработку сложных схем, в том числе предназначенных для последующей реализации в обычных, не программируемых, кристаллах. Однако при этом частично «повисал в воздухе» вопрос электрической корректности — ведь схема, электрически корректная в конкретном кристалле программируемой логики, запросто могла оказаться «плохой», будучи перенесена на «настоящую» пластину кремния для серийного выпуска. Вопрос был успешно решен развитием систем автоматизированного проектирования. Были разработаны и реализованы в рамках соответствующих САПР методы достоверного расчета такой «раскладки» логической схемы на конкретном кристалле, не важно, программируемом или «настоящем», чтобы все клапаны всегда оставались клапанами, не превращаясь в причудливые аналоговые структуры из составляющих их транзисторов. Это позволило возложить обеспечение электрически корректной реализации схемы, представленной «в регистрах», на САПР, и дать, наконец,

схемотехнику мыслить, т. е. и создавать, и отлаживать схемы в терминах логики без всякой примеси «радиолюбительства». Тем самым последнее принципиальное отличие схемотехника от программиста было ликвидировано. И тот и другой отныне мыслят логическими понятиями и практически не ограничены в числе попыток при отладке своих изделий. Длительность и стоимость каждой из таких попыток при этом также оказываются схожими.

Теперь мы видим, что поставленная нами задача разработки проблемно-ориентированных вычислительных устройств — «машин одной задачи» — является вполне осуществимой в принципе. Однако, как не уставали повторять схемотехники времен паяльников и макетных плат, «работать должно не в принципе, а в корпусе». Постановка этой задачи в практической плоскости вскрыла такой пласт системных, методических проблем фундаментального характера, что по сравнению с ним состоявшийся 20 лет назад переход от традиционных процессоров к многопроцессорным вычислительным системам кажется мелким, незначительным эпизодом. Прежде чем переходить к рассмотрению этих проблем, поговорим несколько более подробно о том, как конкретно выглядят сегодня технологии применения изделий программируемой логики.

# ТЕХНОЛОГИИ ПРОГРАММИРУЕМОЙ ЛОГИКИ СЕГОДНЯ

В гл. 11 была обоснована принципиальная возможность разработки проблемно-ориентированных вычислителей примерно теми же, по порядку величины, силами и средствами, какими сегодня разрабатываются программы для традиционных процессоров. Но из принципиальной осуществимости изготовления микропроцессора, и даже из факта серийного выпуска микропроцессоров промышленностью, еще не следует, что у Вас на столе стоит системный блок с компьютером, оснащенным ОС Windows. Как конкретно все это выглядит, как используется изготовителями суперкомпьютеров, что можно купить в магазине и попытаться использовать самостоятельно? Ответам именно на эти вопросы посвящена данная глава.

Надо сказать, что малая осведомленность современного программистского сообщества, особенно молодежи, в рассматриваемых вопросах, несколько удивляет. Масса молодых людей считает себя «разбирающимися в компьютерах», тратя свободное время на изучение того, какой драйвер к какому адаптеру жесткого диска лучше подходит, и как избежать перегрева процессора, «разогнав» его рабочую частоту на целых 10 %, но при этом направив на материнскую плату вентилятор от старого пылесоса. Четверти этой интеллектуальной энергии, будь она израсходована «в мирных целях», хватило бы на разработку процессора с собственной системой команд, и — кто знает — может быть лучшей в мире ОС для этого процессора. Конечно, готовая пиратская версия «стрелялки» на таком процессоре работать не будет — возможно, дело именно в этом, не знаю. Знаю лишь совершенно точно, что когда довелось впервые рассказывать о технологиях программируемой логики студентам-программистам, их немало позабавила идея «скачать из Интернета пару-тройку свободно распространяемых процессоров для домашнего применения». К моменту чтения той лекции эта, по мнению слушателей, забавная идея давно уже стала реальностью. Чтобы в этом убедиться, до-

статочно открыть страницу «процессоры» на сайте [opencores.org](http://opencores.org). Впрочем, вернемся к рассматриваемой теме.

Со времени появления микросхем программируемой логики, выпускаемых массовыми тиражами, понимание специалистами возможностей их применения прошло три этапа.

Сначала казалось, что эти микросхемы являются исключительно «современным вариантом макетной платы», т. е. средством сократить и удешевить процесс разработки логики новых микросхем в условиях, когда ручной монтаж опытных образцов создаваемых схем стал невозможен или запредельно дорог.

Вскоре применение микросхем программируемой логики в этом качестве дало такой толчок разработке всевозможных специализированных электронных устройств, что все больше стало появляться устройств заказных, не ориентированных на действительно крупносерийный выпуск. При разработке таких схем часто оказывалось, что изготовление их в виде «настоящих» пластин кремния после завершения разработки просто не оправдывает себя экономически. Например, в свое время вычислительные модули отечественного суперкомпьютера МВС-100 удалось сделать очень простыми и компактными, реализовав всю логику сопряжения двух процессоров и двух блоков оперативной памяти на единственной микросхеме программируемой логики. Программируемость полученного в итоге «чипсета» никак не использовалась в работе модуля, и даже представляла собой некоторую проблему, которую разработчики собирались ликвидировать, изготовив заказную микросхему на «обычной» пластине кремния. Однако выяснилось, что изготовление таких «обычных» микросхем становилось экономически оправданным лишь при тиражности изделия хотя бы в несколько десятков тысяч экземпляров. При этом весь прогнозируемый объем выпуска модулей в рамках суперкомпьютерной программы МВС-100 оценивался максимум в 100—200 штук. В итоге необходимое количество модулей пришлось изготовить на базе микросхем программируемой логики. Не потому, что их предполагалось постоянно перепрограммировать в процессе работы, а просто потому, что объем выпуска оказался слишком мал, чтобы ради него запускать конвейер на заводе по производству «настоящих» микросхем.

На этом этапе стало ясно, что технологии программируемой логики незаменимы в качестве средства изготовления (а не толь-

ко проектирования) штучных и мелкосерийных цифровых электронных устройств.

Третий этап понимания возможностей этой технологии связан с попытками разработки специализированных вычислителей. Такие попытки предпринимались довольно давно, и часто оказывались удачными [4, 50, 69]. Правда, речь шла обычно не о суперкомпьютерах общего назначения для вычислительных центров коллективного пользования, а о специализированных «машинах одной задачи», например, для обработки данных геофизической разведки полезных ископаемых. Методы такой специализированной обработки постоянно совершенствуются, и возможность периодически перепрограммировать «машину одной задачи», чтобы сделать ее «машиной еще одной задачи», стала рассматриваться как ценная сама по себе.

В России наибольший опыт таких работ имеет коллектив Таганрогского НИИ МВС [50,69]. Специалисты этого коллектива являются признанными лидерами как в разработке самих реконфигурируемых вычислителей (плат, системных блоков), так и в логическом проектировании схем, реализующих сложные прикладные алгоритмы. В настоящее время коллектив НИИ МВС интенсивно работает совместно со специалистами НИВЦ МГУ им. М. В. Ломоносова над совершенствованием технологий логического проектирования, с целью приблизить эти технологии к прикладному программисту, сократить объем необходимого участия схемотехника в разработке логической схемы сопроцессора.

В течение длительного времени специализированные вычислители на базе программируемой логики и универсальные компьютеры на стандартных процессорах существовали «в параллельных мирах». По ряду технических причин, которые мы рассмотрим ниже, только в самое последнее время программируемую логику начали вводить в состав суперкомпьютеров общего назначения, в качестве реконфигурируемых сопроцессоров, «помогающих» обычным процессорам. В настоящее время такие суперкомпьютеры выпускают Cray и Silicon Graphics [20,51]. Предполагается, что пользователь должен самостоятельно, возможно с помощью сторонних фирм, спроектировать схему конкретного сопроцессора для взаимодействия со своей и только своей программой, которая выполняется на обычном процессоре. Теперь уже программируемость схемы сопроцессора — это не «досадная необходимость», как в предыдущем примере, а непременное условие его (сопроцессора) использования.

Технологии построения коммуникационных систем архитектуры NUMA также первоначально применялись почти исключительно в изделиях Cray и Silicon Graphics. Тем не менее, как было показано ранее, сегодня такие системы могут строиться с использованием недорогих, доступных кластерных технологий. Далее будет показано, что аналогичная судьба ожидает, и даже не завтра, а уже сегодня, реконфигурируемые сопроцессоры. К сожалению, нам также предстоит убедиться в том, что сложность применения таких систем, будь они изготовлены фирмой Cray на уникальном заводе, или же областным университетом в факультетской лаборатории кластерных технологий, неизмеримо выше, чем сложность их технической реализации «в металле». Впрочем, вернемся к нашим вентильным матрицам.

Микросхемы программируемой логики выпускаются массовыми тиражами уже далеко не первый год. Ведущая роль в выпуске этих изделий принадлежит трем фирмам — Xilinx [48], Altera [52] и Lattice [53]. Для конечного пользователя доступны инструментальные комплекты на базе таких микросхем, оформленные в виде плат расширения, которые устанавливают в стандартную материнскую плату. Дешевые варианты инструментальных плат могут не устанавливаться в материнскую плату, а подключаться стандартными кабелями через медленные каналы, такие, как USB или com-порт. Инструментальные платы выпускаются как разработчиками микросхем программируемой логики, так и многочисленными сторонними фирмами. Стоимость этой продукции варьируется в очень широком диапазоне — от 3—4 тыс. р. до тысяч долларов. Платы эти выпускаются массовыми тиражами и доступны для приобретения. Следует отметить, что рынок изделий программируемой логики во всем мире растет стремительно. Рост продаж микросхем за год на 150—200 % на этом рынке считается нормальным, скромным результатом.

Типичная инструментальная плата представляет собой «заготовку» для изготовления специализированной вычислительной системы. На такой плате, помимо собственно микросхемы программируемой логики, устанавливается некоторый объем оперативной памяти, и набор выходов на периферийные каналы различного физического и конструктивного формата. Например, разъем сетевого кабеля Ethernet с соответствующим приемопередатчиком физического уровня, разъем кабеля com-порта и т. п. Естественно, устанавливается один или несколько генераторов тактирующих импульсов, иногда — программируемой частоты.



Из общих рассмотрений в гл. 11 на первый взгляд следует, что в оперативной памяти микросхема программируемой логики не нуждается — ведь мы показали, как построить элемент памяти из вентиляй. В действительности ситуация здесь та же, что и при использовании «настоящей» логики на серийно выпускаемой пластине кремния. Память может быть изготовлена из вентиляй, но это очень дорого, и на практике так не поступают. При изготовлении компьютера традиционной архитектуры из вентиляй выполняют лишь внутренние запоминающие элементы процессора. Оперативная память изготавливается из других, отличных от вентиляй, элементов, в основе которых не транзисторы, а конденсаторы. Так же поступают и при построении специализированного вычислителя на программируемой логике, если в его составе предусмотрена оперативная память достаточно большого объема.

Важно понимать, что наличие на плате того или иного периферийного разъема ни в коей мере не означает наличия в ее схеме готового устройства, подключаемого через этот разъем. Например, из наличия на плате разъема Ethernet ни в коей мере не следует, что где-то внутри платы имеется встроенный сетевой адаптер. Разъем комплектуется ровно тем объемом электроники, который необходим, чтобы перевести соответствующий сигнал в форму логического сигнала, пригодного для подачи на входы вентиляй. Все остальное оборудование, например внутренняя логика сетевого адаптера, логика шины PCI, к которой этот адаптер подключен, и даже логика процессора с той или иной системой команд, который управляет этой шиной, должно быть «изготовлено» разработчиком из вентиляй, входящих в состав кристалла программируемой логики. Разработчик вовсе не обязан использовать разъем строго «по назначению». Например, в экспериментальном коммутаторе каналов PCI Express с логикой непрозрачного моста, в разработке которого принимал участие автор данного текста, разъемы формата SATA использовались для подключения коммутатора к кабельной линии связи с узлами, а разъемы Infiniband — для соединения модулей коммутатора между собой. Естественно, электрический «формат» сигнала должен соответствовать тому виду электроники, которым реально укомплектован разъем. В данном случае это соответствие соблюдалось.

Инструментальные комплекты обязательно предусматривают тот или иной способ «холодного пуска», т. е. возможность вы-

полнить по инициативе управляющего универсального компьютера загрузку разработанной схемы в конфигурационную память кристалла программируемой логики и ее запуск. Все дальнейшее схема должна делать сама. Если это схема процессора, вся «механика» уже его начального пуска должна быть предусмотрена разработчиком процессора в его схеме.

Микросхемы программируемой логики различаются не только числом доступных логических элементов, но и видом самих этих элементов, а также разной степенью поддержки высокоскоростных каналов обмена с внешним миром. Микросхемы в точности того типа, который был описан в предыдущей главе, т.е. построенные на базе вентилях «И», «ИЛИ», «НЕ», и оснащенные лишь сигнальными входами и выходами для подключения непосредственно к вентилям, исторически были первыми. Выпускаются они и сейчас. Сегодня это самые дешевые и скромные по своим возможностям микросхемы. Они называются **CPLD** (**C**omplex **P**rogrammable **L**ogic **D**evice — Сложное устройство программируемой логики) [49]. Слово «сложное» в названии призвано подчеркнуть отличие от первых, совсем простых вариантов вентилях матриц.

Основным типом микросхем для разработки действительно сложных устройств на сегодня являются микросхемы **FPGA** (**F**ield **P**rogrammable **G**ate **A**rray — Программируемый «в поле» массив вентилях) [48, 49]. Под программированием «в поле» имеется в виду возможность программирования вне завода, т.е. не при изготовлении, а упоминание вентилях в действительности лишь дань традиции. Эти микросхемы построены на низкоуровневом базисе, отличном от вентилях, но эквивалентном ему. Точнее, на базе так называемых функциональных генераторов, т.е. элементов, реализующих таблично заданные функции алгебры логики от четырех (реже — шести) аргументов. Для проектировщика это различие в базисах большого значения не имеет, в чем мы убедимся далее, при обзоре состава САПР.

Важных отличий **FPGA** от более простых и дешевых кристаллов всего два.

Во-первых, кристаллы **FPGA** почти всегда содержат в себе наряду с универсальными логическими элементами некоторое количество типовых логических узлов, реализованных непосредственно, на которые тратить универсальные логические элементы уже не требуется. Например, **FPGA** фирмы Xilinx имеют в своем составе готовые модули оперативной памяти с двумя не-

зависимыми портами и программируемой разрядностью слова, емкостью 18 кбит, и в количестве до сотен штук на кристалл. Старшие модели также оснащаются готовыми процессорами с системой команд Power PC до четырех штук на кристалл, на изготовление которых также не надо тратить универсальные логические элементы. Другим примером типового логического узла может служить умножитель чисел с фиксированной точкой, которых в старших моделях FPGA фирмы Xilinx также насчитываются сотни.

Во-вторых, кристаллы FPGA различаются по степени оснащенности высокоскоростными линиями ввода-вывода, в частности, пригодными для реализации таких каналов, как PCI Express, Infiniband, Myrinet и Ethernet с частотой 1 Гбит/с и более. В старших моделях FPGA фирмы Xilinx число таких линий достигает 24. В некоторых из них также имеется готовый адаптер PCI Express в виде типового логического узла [48, 49].

Инструментальные платы на базе кристаллов FPGA различаются, в первую очередь, своей оснащенностью периферийными разъемами. Например, на основе одной и той же микросхемы, оснащенной 24 высокоскоростными линиями передачи данных, могут быть изготовлены две различные инструментальные платы. На одной все эти линии будут выведены на внешние разъемы, а на другой — не будут, поскольку эта плата ориентирована на другой круг приложений, для которого используемый кристалл также годится. Конечно, при выборе платы для того или иного круга приложений имеет значение и оснащенность платы оперативной памятью.

Вместе с инструментальной платой конечно поставляется версия САПР, как правило, разработки той же фирмы, что выпустила микросхему.

Все САПР, используемые для проектирования на основе программируемой логики, обеспечивают электрическую корректность создаваемых схем, позволяя разработчику сосредоточиться исключительно на логическом уровне представления создаваемых изделий. Системы эти подразделяются на два больших класса.

Системы первого класса позволяют разработчику проектировать схемы из типовых логических узлов, «в регистрах», т.е. «рисовать» схему примерно так, как это делали разработчики прошлого. Конечно, вместо рисования карандашом на листе ватмана происходит построение схемы с помощью САПР в инте-

рактивном режиме, но сам вид схемы на экране, да и способ ее восприятия при разработке и отладке мало отличается от того, как это все выглядело во времена макетных плат и ручного монтажа. У автора нет опыта использования систем такого рода, но среди многих профессиональных схемотехников они пользуются большой популярностью.

Системы второго класса подразумевают написание алгоритма, описывающего функционирование схемы, на специальном языке программирования. При такой работе деятельность разработчика уже внешне совсем не отличается от работы программиста. Трансляция текста на языке схемотехнического проектирования в схему «в регистрах» происходит автоматически, и разработчик вообще не имеет дело с графическим представлением схемы, подобно тому, как программист на C++ не интересуется процессорными командами, в которые его программа превращается при трансляции.

В обеих технологиях на практике интенсивно используются готовые фрагменты схем — так называемые схемотехнические ядра, *IP cores* (*Intellectual Property cores* — Ядра интеллектуальной собственности). Они выполняют при проектировании роль «стандартных функций». Например, в качестве такого ядра может быть оформлен контроллер памяти, системная шина процессора, или сам процессор. Создание схемотехнических ядер, способных работать не только в руках у автора, работа очень трудоемкая. Сложные схемотехнические ядра коммерческого качества тщательно шифруются и стоят очень дорого. Например, стоимость однократной поставки качественно спроектированного ядра адаптера PCI Express может превосходить 1 млн р. Следует отметить, что постепенно в мировом сообществе разработчиков формируется инфраструктура обмена свободно распространяемыми исходными текстами схемотехнических ядер [54], подобно тому как это давно и успешно происходит в мире традиционного программирования.

В обеих упомянутых выше технологиях разработки изготовленная разработчиком схема транслируется в некоторое низкоровневое представление «в регистрах», не предназначенное для непосредственного восприятия человеком — аналог «объектного модуля» в традиционном программировании. Базис такого представления, т. е. конкретный набор логических элементов и типовых узлов, скрыт от разработчика и не совпадает, вообще говоря, с базисом, физически реализованным в микросхеме программи-

руемой логики. Таким же способом оформлены готовые схмотехнические ядра. Готовые логические узлы большой сложности, присутствующие в тех или иных конкретных микросхемах, для разработчика также представлены в виде схмотехнических ядер. Внутри такого «ядра» вместо «начинки» из универсальных логических элементов присутствует просто ссылка на логический узел определенного типа, характерный для конкретной модели FPGA.

САПР объединяет «объектные модули», т. е. ядра в единую схему, и транслирует ее в конкретный, аппаратно реализованный, базис используемой микросхемы, гарантируя при этом электрическую корректность реализации. В результате получается загружаемый файл конфигурационной памяти микросхемы — аналог исполняемого файла программы в традиционном программировании. Этот загружаемый файл может быть загружен в микросхему с помощью оборудования, предусмотренного для этой цели на инструментальной плате.

Таким образом, реконфигурируемый сопроцессор для использования в составе узла вычислительного кластера — это просто готовая инструментальная плата, которую достаточно приобрести и установить в материнскую плату вычислительного узла в качестве платы расширения. Проблем всего две — выбрать подходящую плату и научиться программировать ее должным образом. Обзору этих проблем посвящена гл. 13.

# ПРОБЛЕМЫ РЕАЛИЗАЦИИ РЕКОНФИГУРИРУЕМОГО СОПРОЦЕССОРА

### 13.1. Проблемы физической реализации и смежные вопросы

Как известно, попытки реализации реконфигурируемых вычислителей на практике начались далеко не вчера. Тем не менее, в течение многих лет эти технологии оставались «широко известными в узких кругах», т. е. весьма успешно применялись для решения узкого круга специфических задач, но слабо «пробивали себе дорогу» в мир вычислений общего назначения. Почему?

При попытке ответа на этот вопрос принято сетовать на «трудности программирования на уровне аппаратуры». Трудности эти, без всякого преувеличения, громадны. Однако главная проблема — проблема изоляции логического уровня проектирования от физического уровня — успешно решена. Следовательно, остались проблемы логические, т. е. целиком расположенные «в голове». Почему же только в последнее время разработчики систем программирования обратили внимание на этот круг задач, озаботились, наконец, созданием подходящих для прикладного программиста языков и трансляторов?

Представляется, что важнейшую роль здесь сыграла проблема коммуникаций. В начале гл. 10 мы уже упоминали эту проблему. Реконфигурируемый вычислитель удастся использовать «в связке» с универсальным процессором тем успешнее, чем больше производительность соединяющего их канала связи. Отмечалось также, что такой вычислитель альтернативной архитектуры, как CUDA GPU, вряд ли имел бы смысл, если бы он не был связан с универсальным процессором каналом PCI Express шириной 16. То же самое верно и для реконфигурируемых вычислителей. Используемая в этом качестве готовая плата программируемой логики обязательно должна быть платой формата PCI Express с достаточно широким каналом. Пока таких каналов в составе материнских плат, из которых строятся вычислительные кластеры, не было, не было и особых резонов форсировать решение логических проблем.

Вторая проблема состоит в том, что микросхемы FPGA в настоящее время, к сожалению, имеют гораздо более низкие рабо-

чие частоты, чем современные им традиционные микросхемы. Разница немного превышает десятикратную, причем в последние годы сокращается, хотя довольно медленно. Например, выпускаемые во времена Pentium с рабочей частотой 1 ГГц FPGA фирмы Xilinx имели в своем составе типовой логический блок процессора PowerPC с частотой 300 МГц, а процессоры, реализованные из универсальных логических элементов в этих микросхемах, имели частоту 100 — 120 МГц [48, 49]. Следовательно, для реализации традиционных вычислителей, например, фоннеймановского процессора, работающего быстрее, чем Pentium, FPGA не годятся. Реализуя в FPGA проблемно-ориентированные вычислители, разработчикам приходится ускоряться за счет именно архитектурных факторов гораздо более чем в десять раз, чтобы работа имела смысл. Опыт многочисленных разработок специализированных вычислительных устройств показывает, что это вполне возможно [50, 69]. Отсюда также следует, что при реализации внутреннего параллелизма обработки данных в реконфигурируемых вычислителях мы всегда будем испытывать дефицит количества одновременно работающих функциональных устройств, т. е. дефицит логических элементов. Конечно, главный наш козырь — архитектурное ускорение за счет специализации структуры схемы, но если мы специализируем ее, например, в сторону векторной обработки, нам обязательно захочется, чтобы длина вектора была как можно больше — ведь наш «конкурент», универсальный процессор, имеет в этом соревновании десятикратную «фору». Значит, имеет смысл ориентироваться на кристаллы как можно большего объема. С использованием CPLD объемом 100 тысяч вентилях вряд ли удастся создать что-либо путное. Требуются FPGA с эквивалентной вентиляльной емкостью 1 млн и более. Для того чтобы как-то измерить проблему, приведем простой пример. Для успешной попытки обогнать Pentium с частотой 3 ГГц на знаменитом тесте Linpack путем реализации этого алгоритма в кристалле программируемой логики, при рабочей частоте получившейся схемы 100 МГц, потребовался кристалл с эквивалентной вентиляльной емкостью 3 млн, при этом ускорение, по сравнению с Pentium, было примерно десятикратным [55].

Этот пример довольно интересен по двум причинам: во-первых, пример успешного «обгона» универсального процессора с помощью микросхемы программируемой логики, не современной универсальному процессору, а даже на несколько лет более

старой; во-вторых, пример наглядно опровергает расхожее мнение о том, что «программируемая логика, быть может, и хороша для задач управления, но для вычислений с плавающей точкой универсальным процессорам нет равных, по причине их гораздо более высокой рабочей частоты». Автору приходилось не раз слышать эту точку зрения. Теперь нам известно, что она устарела в той же мере, что и деление команд универсального процессора на «быстрые» и «медленные». В самом деле, нам не раз уже приходилось отмечать, начиная с главы, посвященной происхождению вычислительного быстродействия, что для современных универсальных процессоров любая вычислительная задача является, по структуре затрат времени, той самой «задачей управления». Специализируя наш вычислитель структурно, мы как раз эту «задачу управления» и решаем.

Идеальное, с точки зрения применения в вычислительных кластерах, конструктивное исполнение реконфигурируемого сопроцессора предложено фирмой XtremeData совместно с Altera [56, 52]. В изделиях этих фирм микросхема FPGA устанавливается не на плату расширения, а на цоколь от микропроцессора Pentium или Opteron. Такая микросборка устанавливается в процессорное гнездо двухпроцессорной материнской платы вместо одного из процессоров. Многочисленные выгоды такого решения достаточно очевидны, и в особых комментариях не нуждаются.

Теперь, когда все проблемы физической реализации рассмотрены и признаны практически разрешимыми, перейдем к рассмотрению завершающей, самой сложной, но и самой интересной проблемы. Речь идет о проблеме программирования как таковой, проблеме написания текста программы, который должен превратиться в схему, загружаемую в сопроцессор.

## **13.2. Проблемы логического проектирования**

На первый взгляд кажется, что проблемы логического проектирования как таковой нет. В самом деле, современные САПР предусматривают разработку схем в виде текстов программы на языке схемотехнического проектирования. Да, этот язык — не С и не Фортран, но имеет ли это сколько-нибудь серьезное значение? Даже не очень искушенные в системных тонкостях программисты вряд ли будут сильно расстроены, если для много-



кратного повышения быстродействия им потребуется «всего лишь» выучить новый язык. Да и транслятор с Фортрана или С, скорее всего, появится скоро, великое ли дело — написать еще один транслятор, если пользователи попросят?

Как бы правдоподобно ни звучали приведенные только что соображения, они не просто не полны или поверхностны. Они неверны в принципе, не имеют с практическим положением дел ничего общего. Для иллюстрации сути проблемы начнем с простейшего примера. Ниже приводится тривиальный фрагмент текста на языке схемотехнического проектирования VHDL [57].

```
a <= '1';
b <= '0';
If a = '1' then
  If b = '0' then
    c <= '0';
  else
    c <= '1';
  end if;
end if;
```

Зададим любому программисту, не знакомому с языками схемотехнического проектирования, несколько простых вопросов по поводу этого фрагмента:

- чему будет равен, после исполнения этого фрагмента, триггер «с»;
- в каком порядке будут выполняться проверки в двух вложенных условных операторах;
- что вообще можно сказать о порядке выполнения операторов в этом фрагменте программы?

Правильные ответы, очевидные даже начинающему схемотехнику, таковы:

- о значении триггера «с» после выполнения этого фрагмента программы ничего сказать нельзя, в приведенном тексте информации недостаточно;
- проверки в двух вложенных условных операторах будут выполняться в естественном порядке вложенности, **НО**
- все операторы в приведенном фрагменте, включая все проверки и все присваивания, будут выполняться **одновременно!**

Не являются ли противоречащими друг другу, заведомо несовместимыми, ответы на второй и третий вопросы? В действительности, они не более противоречивы и несовместимы,

чем правая и левая части такого оператора присваивания на языке C:

$$N = N + 1;$$

Покажите эту запись человеку, знающему алгебру, но совсем не знакомому с программированием, и не удивляйтесь, если он попытается немедленно «вызвать санитаров».

Чтобы не быть голословными, попробуем разрешить кажущиеся противоречия.

В случае оператора присваивания формула объяснения очевидна любому программисту — первокурснику, и будет звучать так: «Данная запись является не констатацией факта равенства, а предписанием сделать левую часть равной тому, чему была равна правая в некоторый предыдущий момент времени, в котором разворачивается выполнение программы».

В случае фрагмента текста на VHDL формула объяснения несколько сложнее: «Данная запись является предписанием выполнить некоторые присваивания, причем присваивание триггеру «с» должно произойти лишь в случае успешного выполнения двух вложенных проверок. Вторая проверка подлежит исполнению лишь в случае, если первая будет успешной, и в этом смысле проверки «выполняются в порядке вложенности». Однако, схема, выполняющая все проверки и присваивания, срабатывает за один такт генератора синхроимпульсов, и результат ее срабатывания станет «виден» только на следующем такте. Поэтому и о результате проверок, и о последующем значении триггера «с» ничего сказать нельзя, а утверждение о том, что все операторы выполняются одновременно, является верным».

Словом, если Вы, например, механик, а Ваш родной язык — русский, Вам вряд ли достаточно изучить английский язык, чтобы поддержать беседу на профессиональную тему с английским врачом или юристом. Неплохо бы знать еще медицину или право. Дело не столько в языке, сколько в понятиях, для выражения которых он предназначен.

В данном случае понятийная база отличается от понятийной базы программиста очень сильно. Например, представление о выполнении действий в определенной последовательности, задаваемой программистом — краеугольный камень фоннеймановской модели, на которую ориентированы все традиционные языки программирования. Эту последовательность принято задавать порядком записи операторов в программе. Если какие-то

операторы выполняются параллельно, а не последовательно, это обычно записывается специальным образом. «Продвинутые» программисты знают, что в языках, не являющихся алгоритмическими, вроде упоминавшейся выше Нормы [28], порядок операторов в программе может не иметь значения, поскольку операторы в таких языках не являются выполняемыми в привычном смысле этого слова. Но VHDL — язык алгоритмический, его операторы являются выполняемыми! Просто они выполняются в совершенно другой среде, не описываемой фоннеймановской моделью, грубо говоря, не во времени, а в пространстве. В частности, каждый оператор VHDL — это кусочек схемы, т. е. набор соединенных между собой вентилях. Представление о том, что кусочки эти будут срабатывать в той последовательности, в которой они написаны, так же противоестественно для схемотехника, как представление о том, что схема, начерченная на листе ватмана, будет работать слева направо и сверху вниз по чертежу.

Тем читателям, которые хотят немного больше и конкретнее ознакомиться с программистской моделью VHDL, рекомендует-ся прочитать прил. 3.

Из приведенных примеров должна быть, в общих чертах, ясна безнадежность ожидания того, что появится, наконец, транслятор, преобразующий в схему текст программы на С.

Такой транслятор, конечно, возможен, и даже не очень сложен в реализации. Его работа могла бы состоять в следующем:

- построить схему фоннеймановского процессора с некоторой разумной системой команд;
- выполнить трансляцию текста на С в программу, состоящую из команд этого процессора;
- выполнить эту программу на этом процессоре.

Ничего, кроме десятикратного, как минимум, замедления программы по сравнению с ее выполнением на стандартном процессоре, мы при такой «трансляции в схему» не получим. Сама запись программы на языке С, т. е. в терминах, свойственных фоннеймановскому процессору, делает почти невозможной ту самую архитектурную специализацию, реализацию алгоритма в не фоннеймановских терминах, ради которой все и затевалось.

С этой проблемой мы уже сталкивались в первой части книги, обсуждая задачу распараллеливания последовательной программы. Все сказанное при рассмотрении технологий параллельного программирования подвело нас тогда к довольно очевидному

выводу. Задача распараллеливания, вообще говоря, — задача искусственного интеллекта. Последовательная программа потому и является последовательной, что смысл ее выражен в терминах последовательного выполнения операторов. Чтобы выразить смысл в терминах параллельного выполнения некоторых действий, требуется принципиально другой язык. Не язык, в котором иначе записываются присваивания и проверки условий, а язык, основанный на других базовых понятиях. Например, на понятиях рандеву, или памяти с разным временем доступа независимо выполняющихся процессов к разным ее областям.

В данном случае отличия системы базовых понятий от фоннеймановской модели, привычной программистам, гораздо более сильные, чем в случае ставших уже традиционными систем массового параллелизма. Система массового параллелизма построена из традиционных процессоров, и язык (в широком смысле этого слова) ее программирования является обычно расширением традиционного языка фоннеймановского типа. В данном случае ни о каком расширении речь не идет, набор базовых понятий просто другой. Значит, отличия необходимого нам языка от традиционных языков также должны быть гораздо более сильными. Помимо совершенно непривычной модели течения времени, в котором разворачивается выполнение программы, резко отличается от фоннеймановского случая модель системы памяти. Однородной адресуемой памяти в мире вентильных матриц просто нет, точнее, ее не должно быть у нас, если мы хотим перегнуть основанные на такой памяти традиционные процессоры.

Языки схемотехнического проектирования VHDL [57] и Verilog [58], казалось бы, обладают всеми свойствами, которые нам необходимы. Однако они совершенно не годятся для их применения прикладными программистами. Уровень этих языков слишком низок, но не в том смысле, в котором уровень языка ассемблера ниже уровня языка Фортран, а в смысле необходимого уровня понимания того, как устроена аппаратура. Операторы языка вполне высокоуровневые, синтаксис их, в случае языка Verilog, даже похож на синтаксис языка С. Сложность в том, что те базовые понятия и действия, которые этими операторами выражаются, для прикладного программиста совершенно непостижимы.

Ситуация усугубляется тем, что языки схемотехнического проектирования не обладают свойством, которое мы рискуем назвать «логической надежностью».

Любой текст на С или Фортране, который успешно скомпилируется и соберется в исполняемый модуль, заведомо представляет собой запись программы, которая может выполняться. Возможно, она неправильна или бессмысленна, но программой, т. е. совершенно конкретной последовательностью команд процессора, она является. В этом смысле языки программирования фоннеймановского типа логически надежны. О языке VHDL, например, этого сказать нельзя. На этом языке вполне можно написать текст, удовлетворяющий всем синтаксическим и семантическим правилам языка, но настолько бессмысленный, что построить по нему схему просто невозможно. Трансляторы в составе даже очень качественных версий САПР такие тексты зачастую ставят в тупик. Диагностика невнятна или отсутствует, что получилось на выходе — лучше не спрашивать. Профессионалы, разрабатывающие схемы на VHDL, умудряются годами не попадать в такие неприятные ситуации по одной простой причине. Они понимают, какую схему хотят построить, и им просто в голову не приходит написать бессмысленный текст. Прикладному программисту — типичному пользователю многопроцессорной системы — такое придет в голову обязательно.

Словом, у нас есть язык, **на который** мы могли бы транслировать программы (VHDL или Verilog), но нет языка, **на котором** программисты могли бы эти программы **писать**. Корень проблемы — в совершенно непривычном наборе базовых понятий. Ситуация усугубляется тем, что привычный прикладным программистам набор базовых понятий — всего один, и программисты всего мира привыкали к нему более 60 лет, одновременно отвыкая от мысли, что нечто другое вообще имеет право на существование. Рассматривая технологии параллельного программирования и смежные вопросы, мы уже имели возможность убедиться в трудностях, возникших при необходимости всего лишь расширить этот привычный набор. Теперь же нам требуется создать нечто новое «на ровном месте».

Речь идет, таким образом, о размывании грани между программистами и схемотехниками. Гроть эта всегда проходила по фоннеймановскому процессору, точнее, сам фоннеймановский процессор и был этой гранью. Если нам пришлось «замахнуться» на фоннеймановский процессор, мы с неизбежностью «замахиваемся» и на существовавшее более 60 лет разделение труда, на право программиста не знать, «что там внутри». Раньше казалось, что грань эта незыблема, поскольку схемотехник «знает

электронику», а программист — нет. Современные САПР освободили схемотехника от необходимости «знать электронику», и тут же выяснилось, что логика тоже бывает разная, «схемотехническая» и «программистская».

В этой связи интересно было бы вспомнить, как обстояло дело на заре информатики. Когда разделение труда в его современном виде еще не сложилось, а фоннеймановский процессор был всего лишь одним из многих остроумных способов организации нескольких тысяч радиоламп в систему, на которой можно было бы выполнять расчеты.

Так как же работали в то время, когда еще не было не только трансляторов, но и самих прикладных программистов? Очень просто. Место отсутствующего автомата — системы программирования — занимал человек. Поначалу ему приходилось напрягать все свои творческие способности, чтобы выполнить необходимую работу. Постепенно он справлялся все лучше и лучше, работа становилась все более рутинной, пока не появлялась возможность спроектировать необходимый автомат. Тогда соответствующий человек превращался, образно говоря, из землекопа в конструктор экскаваторов, т. е. из кодировщика — составителя программ в разработчика систем программирования.

Конкретно это выглядело так.

Математики того времени немного (по современным меркам) знали численные методы, но совсем не знали алгоритмики, т. е. фоннеймановский процессор был для них примерно тем же, чем для сегодняшних математиков является вентильная матрица. Специалисты по вычислительной технике зачастую не знали на должном уровне математики. Для общения друг с другом им приходилось придумывать полуформальные **языки записи заданий на программирование**. Процесс был итерационным и, видимо, довольно мучительным. В итоге в каждом более или менее крупном коллективе образовывался свой язык, на котором записывался алгоритм расчета. Важно, что язык этот, не будучи строго формальным, все же был равно понятен обеим договаривающимся сторонам. Именно из таких полуформальных языков записи заданий и родились, в итоге, первые языки программирования. Родились именно тогда, когда люди их, фактически, изобрели в процессе работы, накопили необходимый понятийный базис, и готовы были эти новые языки понять и принять.

Нечто подобное нам необходимо выполнить сегодня. Напрасно было бы надеяться, что можно спроектировать и реализовать

язык, транслируемый на VHDL, после чего опубликовать его описание и ждать, что математики им воспользуются. Воспользоваться им они не смогут, а язык будет плохим. Более того, реализовать требуемый язык гораздо проще, чем его придумать. Придумать же его можно только в тесном диалоге с математиками. Подобно составителям программ прошлого, нам, разработчикам завтрашних суперкомпьютеров, наверняка придется на какое-то время стать постоянно действующими системными архитекторами, т. е. научиться быть посредниками между математиком и вентильной матрицей. Только научившись выполнять эту работу вручную или почти вручную, мы поймем, как ее автоматизировать.

Работа системного архитектора — дело творческое и почетное, если новый процессор создается годами, а эксплуатируется 10—20 лет, как это было когда-то. Сегодня нам необходимо сжать процесс проектирования типовых вычислительных ядер до нескольких дней или недель, чтобы такая деятельность имела смысл. Это кажется почти невозможным, но иного пути нет. Как было показано ранее, все технические предпосылки для движения в этом направлении уже налицо, следовательно, задача скоро будет решена.

Не удивительно, что процесс уже начался. Языки проектирования вычислительных схем, ориентированные не на схемотехника, а на прикладного программиста, в последние два-три года начали появляться. В гл. 14 будут кратко рассмотрены некоторые из них. К сожалению, опыт применения систем такого рода у автора на сегодняшний день отсутствует.

# ЯЗЫКИ ПРОГРАММИРОВАНИЯ, ТРАНСЛИРУЕМЫЕ В СХЕМУ

### 14.1. Handel-C

Среди языков, предназначенных для разработки схем непосредственно прикладным программистом, язык **Handel-C** выделяется своей простотой. В то же время программистской модели этого языка не откажешь в адекватности основным свойствам оборудования. Будет совсем не удивительно, если этот язык сыграет ту же роль среди языков прикладного схемотехнического проектирования, какую в свое время сыграл среди языков программирования язык Фортран. **Handel-C** разработан компанией Celoxica [59].

Многие языки такого рода претендуют на то, чтобы быть «диалектами С», но, пожалуй, именно Handel-C подошел вплотную к воплощению этой идеи на практике. Это позволит нам не тратить время на обзор синтаксиса языка, а сосредоточиться сразу на программистской модели.

Программа на Handel-C оперирует объектами трех видов:

- переменными и массивами;
- каналами;
- блоками динамически адресуемой памяти.

**Переменные и массивы** — это, по существу, одно и то же. При обращении к элементу массива значения всех индексов обязаны быть константами, значения которых известны во время трансляции. Обратиться к  $i$ -му элементу массива, где  $i$  — переменная, нельзя. Это делает элементы массива мало отличающимися по использованию в программе от одиночных переменных.

**Каналы** — это точки коммуникации между параллельно выполняющимися ветвями программы, а также между программой и внешним миром. Коммуникации выполняются в стиле классического рандеву. Коммуникации между параллельно выполняющимися ветвями программы возможны не только через каналы, но и через доступ этих ветвей к одним и тем же переменным. Это позволяет считать все, связанное с каналами, достаточно автономной частью языка, которую мы в нашем кратком обзоре опустим.



**Блоки динамически адресуемой памяти** выполняют в программе ту же роль, что традиционные массивы в программе на языке фоннеймановского типа. В отличие от массивов они позволяют обратиться к *i*-му, а не только 5-му или 7-му элементу блока.

Все перечисленные программные объекты имеют произвольную двоичную разрядность, обычно задаваемую явно при объявлении, например:

```
int 6 a;
```

Здесь объявлена целочисленная, 6-разрядная переменная «a».

```
ram int 18 b[32];
```

Здесь объявлена динамически адресуемая область памяти, допускающая чтение и запись, состоящая из 32 18-разрядных целых чисел.

Единой адресуемой памяти в схеме, вообще говоря, нет. Соответственно, у объектов, которыми оперирует программа, нет адресов, а в языке нет понятия указателя.

В отличие от языков схмотехнического проектирования, в Handel-C порядок выполнения программы соответствует порядку записи операторов программы, как в традиционных языках программирования. Более того, по умолчанию операторы, из которых состоит выполняемая часть программы, выполняются строго последовательно, друг за другом. Как мы уже знаем, программы, записанные в такой форме, гораздо лучше выполнять на универсальном процессоре. По этой причине, реальные программы на Handel-C предусматривают параллельное выполнение операторов программы везде, где это только возможно. Чтобы несколько операторов, записанных в программе друг за другом, выполнялись параллельно, их следует оформить как блок, перед которым пишется специальное ключевое слово «par», например:

```
par
{
    x = 1;
    {
        y = 2;
        z = 3;
    }
}
```

Здесь написано, что присваивание единицы переменной *x* выполняется одновременно с последовательностью присваиваний двойки переменной *y* и тройки переменной *z*. Возможность указывать в качестве оператора внутри параллельно выполняемого блока не только одиночные операции, но и блоки, позволяет записывать параллельное выполнение сложных программных ветвей, возможно, с вложенным параллелизмом. Операторы внутри параллельно выполняемого блока подчиняются дисциплине «fork-join», хорошо знакомой нам на примере OpenMP.

Число параллельно выполняемых ветвей ограничено только объемом используемой микросхемы FPGA. Исполнение программы не предусматривает какого-либо уровня промежуточной интерпретации, каждый оператор превращается, вообще говоря, в отдельную часть схемы. Для выполняемых параллельно ветвей эти части схемы заведомо разные, и на реализацию конструкций с очень высокой степенью параллелизма в кристалле FPGA может просто не хватить логических элементов. В этом случае программисту придется самостоятельно «портить» программу на уровне исходного текста, например, делить обработку длинных векторов на операции с векторами меньшей длины, выполняемые последовательно.

Модель выполнения Handel-C предусматривает глобальное дискретное время, т. е. понятие выполнения оператора на том или ином такте. В терминах реализации это означает, что транслятор строит синхронные, управляемые единым тактовым генератором схемы.

В языке довольно много строгих ограничений на одновременный доступ к переменным из параллельных ветвей программы, которые формулируются в терминах тактов. При этом правило подсчета тактов, необходимых для выполнения тех или иных программных конструкций, просто настолько, что вряд ли станет проблемой даже для «самых прикладных» из программистов. Звучит оно так: «Присваивание занимает один такт, никакие другие действия тактов не требуют». Такая формулировка понятия дискретного времени означает, в частности, что все вычисления в программе выполняются схемами, срабатывающими за 1 такт. По мере усложнения этих схем максимальная допустимая для данной программы рабочая частота может сильно падать.

Общий смысл ограничений на одновременный доступ к переменным из параллельных ветвей программы сводится к тому,

что в одном такте можно присвоить значение данной переменной только один раз. То же касается элемента массива. Обратиться к одной и той же динамически адресуемой области памяти также можно лишь один раз за такт. В описании языка подробно разбираются на простых примерах типичные следствия из этих ограничений, не всегда сразу очевидные прикладному программисту. В целом язык производит очень благоприятное впечатление именно за счет сочетания простоты базовых идей и конструкций в терминах программистской «картины мира» с весьма высокой степенью адекватности языка специфике вентильной матрицы.

Помимо упомянутых достоинств, у языка есть серьезный недостаток. В нем практически невозможно записать конвейерный параллелизм на микроуровне, например, работу арифметического устройства, которое выдает по одному результату за такт, но с задержкой на пять тактов относительно подачи слагаемых. В то же время именно такой режим использования ресурсов реконфигурируемого вычислителя, в частности памяти, является основным в реальных схемах, достигающих высоких показателей реального быстродействия.

## 14.2. Mitrion-C

Наличие буквы «С» в названии языка **Mitrion-C** — пожалуй, единственное, что объединяет его с Handel-C. Причем объединяет, на взгляд автора, совершенно незаслуженно. «Диалектом традиционного С» Mitrion-C не является, поскольку вообще не является алгоритмическим языком. Mitrion-C — это функциональный язык, реализующий модель вычислений по готовности данных. Язык разработан компанией Mitronics [60].

Модель программирования функциональных языков имеет более чем 30-летнюю историю, т. е. издавна существует и используется, наряду с фоннеймановской моделью. Само по себе понятие функционального языка никак не связано ни с параллельным программированием, ни с проблемой трансляции прикладных программ непосредственно в схему. Напомним кратко наиболее важные свойства этой программистской модели [63].

Программа на функциональном языке представляет собой функцию, значение которой требуется вычислить. В качестве аргументов этой функции могут использоваться значения других

функций и т.д. Функции не имеют побочных эффектов, т.е. единственным, с точки зрения вызывающей программы, результатом вызова функции является вычисление значения функции.

Отсутствие побочных эффектов позволяет рассматривать структуру вложенности вызовов функций как естественным образом заданную информацию о зависимостях по данным, имеющихся в данной вычислительной процедуре. В самом деле, если написано что-то вроде:

$$F(a(x, y), b(v, w(z)))$$

то уже из самой этой записи следует, что:

- значение функции  $F$  нельзя вычислять, пока не будут вычислены значения функций  $a$  и  $b$ , и надо вычислять, как только эти значения будут готовы;

- значения функций  $a$  и  $b$  можно вычислять независимо, т.е. в любом порядке, или же параллельно;

- при этом значение функции  $b$  нельзя вычислять до того, как вычислено значение функции  $w$ , и надо вычислять, как только это значение будет готово.

Наличие в тексте программы такой информации о зависимостях по данным делает параллельную реализацию программы совершенно тривиальной, механической работой. В том числе параллельную реализацию в схеме, как в случае Mitrion-C.

В Mitrion-C не только функции как таковые, но и все сложные операторы (условные, операторы цикла) являются функциями без побочных эффектов, возвращающими значение. Это позволяет говорить о языке как о функциональном, и избавляет программиста от необходимости мыслить в терминах глобального времени, избегая одновременного присваивания значений одним и тем же переменным. Вся информация о возможных конфликтах такого рода содержится в тексте программы, и может быть учтена транслятором автоматически.

Таким образом, найден способ записи программы, адекватный целевому оборудованию, но начисто лишенный необходимости мыслить в низкоуровневых терминах вентилей, триггеров и тактовых генераторов.

Само по себе это является серьезным достижением разработчиков языка. Им удалось найти заметно более высокоуровневый, по сравнению с Handel-C, формализм для записи программы, ничуть не менее пригодный для трансляции в схему с соблюде-

нием максимально возможного уровня параллелизма. И все же примененный разработчиками этого языка подход не настолько хорош, как кажется на первый взгляд. Чтобы убедиться в этом, вспомним еще раз, какую проблему призван решить язык прикладного схемотехнического проектирования.

Как отмечалось в начале главы, проблема вовсе не в том, что языка, пригодного для трансляции в схему, не существует, а в том, что набор понятий существующих языков, таких, как, например, VHDL, слишком отличается от набора понятий, которыми мыслят прикладные программисты. Но функциональный язык в этом отношении вряд ли намного лучше, чем VHDL — ведь VHDL, по крайней мере, язык алгоритмический и императивный, хотя и не фоннеймановского типа. Функциональные языки, конечно, существуют и успешно используются программистами не одно десятилетие, но среди именно этих программистов, к сожалению, довольно трудно обнаружить специалистов по численным методам и уравнениям в частных производных. Мир функционального программирования — довольно замкнутое сообщество системных программистов, почти так же далекое от мира прикладного программирования вычислительных задач, как и сообщество схемотехников. Невооруженным взглядом видно, что разработчики Mitrion-C принадлежат именно к этому сообществу. По крайней мере, они постарались максимально отразить в языке все традиционные для своего «клуба», но совершенно непривычные для остальных конструкции языка — списки, кортежи и т. п. Почти наверняка язык стал от этого более красивым и стройным. Более понятным и приемлемым для прикладного программиста вычислительных задач он от этого точно не стал.

Модель программирования, присущая функциональным языкам, конечно является более высокоуровневой, чем модель программирования VHDL, но из этого вовсе не следует, что прикладным программистам будет легче ее освоить. В этом смысле подход разработчиков Handel-C, честно попытавшихся построить императивный, алгоритмический язык, представляется автору более перспективным.

## ЗАКЛЮЧЕНИЕ

Любая отрасль инженерной деятельности проходит в своем развитии этап бурной молодости, о которой потом слагают легенды, а затем — длительный период зрелости, т. е. устойчивого, постепенного развития. Результаты работы созданной отрасли обычно видны на втором этапе. В самом деле, как-то странно даже называть одним и тем же словом самолет братьев Райт и современный самолет, дорожный паровой тягач середины XIX в. и автомобиль XXI в. Объем нетривиальных технических решений, аккумулированный в изделиях «зрелого» этапа развития, громаден, не идет ни в какое сравнение с техническим уровнем собранных «на коленке» первых опытных образцов. Почему же тогда сюжетом легенд становится именно время молодости отрасли? Не в последнюю очередь — потому, что опытный образец, первое в своем роде, принципиально новое изделие, приходится творить **от начала до конца**. Одно дело — придумать, сделать и поднять в воздух самолет, другое — спроектировать закрылок или гидропривод шасси. Второе требует в современных условиях огромного объема конкретных знаний и навыков, но первое — всегда почетнее, а главное — **интереснее**. Почти всегда та область, о которой уже начали рассказывать легенды, период своей молодости прошла. Вдохновленным этими легендами молодым людям, думающим о профессии инженера — исследователя, приходится ясно понимать: в эту отрасль, коль скоро легенды уже появились, идти, может быть, поздно.

Автор этой книги знает лишь одну отрасль, являющуюся исключением из этого правила. Отрасль высокопроизводительных вычислений с завидным постоянством «впадает в детство», снова и снова проходит легендарный этап развития, с периодом всего в несколько лет. Активно работающий инженер имеет совершенно реальный шанс «придумать, построить и поднять в воздух самолет» не раз и не два за время своей профессиональной карьеры, всякий раз начиная все заново, не размениваясь на мелочи, решая принципиальные вопросы в новой постановке,

которой еще вчера не было и не могло быть. Автору очень хотелось бы верить, что все рассказанное в данной книге подведет читателя именно к этому оптимистическому выводу. Как бы ни мало в численном отношении требовалось современной промышленности инженеров широкого профиля, более склонных проектировать самолет в целом, нежели самый лучший в мире закрылок, именно эти специалисты почему-то всегда оказываются в дефиците. Хотелось бы верить, что данная книга поможет решению и этой острейшей проблемы из области параллельной обработки данных.

В заключение хотелось бы поблагодарить многих и многих своих коллег за неоценимую помощь в работе над этой книгой. Задача эта, как и почти все, что связано с параллельной обработкой данных, сложнее, чем кажется на первый взгляд.

Данный текст является, в значительной степени, обобщением всего того, что мне довелось понять на собственном опыте в течение последних 15 с лишним лет. Совершенно очевидно, что осмысление, пусть в чем-то и поверхностное, такого объема сведений может быть только результатом труда большого коллектива, всех тех, с кем на протяжении этого периода приходилось работать вместе. При этом многие мысли, когда-то подсказанные более опытными или более талантливыми коллегами, со временем становятся настолько очевидными, что их постепенно начинаешь принимать за собственные, чего, конечно же, делать не следует. Впрочем, раздел «Благодарности» и есть то самое место в книге, в котором принято исправлять такого рода упущения.

Тексты модельных программ решения двумерной задачи Дирихле, использованные в этой книге, были любезно предоставлены несколько лет назад А. Е. Луцким. Если бы его пришлось благодарить за это столько раз, сколько они были в течение этого времени мной использованы, как в лекциях для студентов, так и во всевозможных пояснениях, список только этих благодарностей занял бы не одну сотню страниц.

Не могу не выразить искреннейшей благодарности М. Ю. Храмову за многочисленные обсуждения самых разных вопросов теории и практики параллельной обработки данных, и не только. Несмотря на весьма бурный, мягко говоря, характер почти всех таких обсуждений, мне не раз приходилось убеждаться впоследствии, что во многих своих кажущихся парадоксальными утверждениях и суждениях мой оппонент оказывался совершенно прав.

На ранних этапах проектирования нового суперкомпьютера неизменную готовность помочь в сокращении «пропасти» между вычислительной математикой и схмотехникой проявлял А. Б. Карагичев. Порой ему доводилось успешно решать предложенные мной задачи, за которые, будь они поставлены кем-то передо мной, я бы точно не взялся.

Невозможно переоценить вклад в создание этой книги, внесенный пользователями параллельных суперкомпьютеров МВС-100 и МВС-1000. Более чем за 15 лет существования Испытательной лаборатории проекта МВС в ИПМ им. М. В. Келдыша РАН мне приходилось общаться с десятками прикладных программистов, приходивших в нашу лабораторию, чтобы понять, как работать на суперкомпьютере. Кому-то это удалось почти сразу, кому-то — так и не удалось. Кто-то проявлял подлинные чудеса терпения и настойчивости, заставляя нас — наладчиков и «системщиков», добиться того, чтобы наши «в принципе работающие» машины стали на практике полезным для математиков рабочим инструментом. Всем им без исключения огромное спасибо.

Не только самих суперкомпьютеров серий МВС-100 и МВС-1000, но и этой книги, написанной в значительной степени «по следам» их изготовления и эксплуатации, не появилось бы без многолетнего плодотворного сотрудничества с коллегами из НИИ «Квант». В. Н. Грознову, Г. С. Елизарову, В. В. Каратанову, В. В. Корнееву, Г. Б. Кулькову, А. В. Патрикееву, А. Г. Титову, С. В. Яблонскому, В. В. Ялину, многим другим и, конечно же, Научному руководителю НИИ «Квант», академику РАН В. К. Левину большое спасибо.

Во многом решающее значение на начальном этапе работы над коммутатором PCI Express имели беседы с А. В. Луценко, который независимо спроектировал и испытал очень похожее техническое решение. Положительные результаты его испытаний позволили нам гораздо быстрее и увереннее продвигаться на ранних стадиях этой работы.

Многое из написанного во второй части этой книги удалось гораздо лучше понять благодаря сотрудничеству с коллегами из НПО «Роста», в первую очередь — М. А. Маркиным, К. К. Хачатуряном и В. Г. Яковлевым. Очень полезными были беседы с И. И. Левиным из НИИ МВС во время наших встреч на конференциях в Пушкино.

Первым читателем черновика этой рукописи любезно согласился быть Арк. В. Климов. Вдумчивость и подробность его



работы с текстом, который на том этапе, честно говоря, такого отношения вовсе не заслуживал, поразила меня и продолжает поражать до сих пор. Спасибо огромное.

Важную помощь в прояснении некоторых парадоксальных свойств языка VHDL оказал И. А. Адамович.

Очень многое довелось по-новому понять, да и просто впервые сделать, благодаря сотрудничеству с кафедрой САУ, возглавляемой проф. Е. Н. Черемисиной, кафедрой РИВС, возглавляемой проф. В. В. Кореньковым, и Отделом компьютерных технологий, возглавляемым Ю. А. Крюковым, Университета «Дубна». Многому научили и студенты университета — даже те, с кем впервые приходилось знакомиться непосредственно на экзамене.

Наконец, особая и отдельная благодарность тем моим коллегам по ИПМ, которых я еще не имел чести упомянуть. По меркам этого довольно большого, но удивительно стабильного коллектива я проработал в нем не так уж и много — меньше 30 лет. Тем не менее за такое время «обрастаешь» многочисленными контактами, и риск не упомянуть кого-то, с кем пришлось в прошлом много и успешно работать, возрастает. Впрочем, хуже всего было бы не упомянуть никого.

Неизменный интерес к работе, готовность выслушать и высказать много ценных замечаний, постоянно проявляли Анд. В. Климов и В. А. Крюков. Вряд ли удалось бы даже просто выделить то время, которое потребовалось на написание этого текста, без неизменной поддержки со стороны моих непосредственных начальников — члена-корреспондента РАН А. В. Забродина и Главного инженера ИПМ им. М. В. Келдыша Ю. П. Смольянова. Без их постоянного внимания и поддержки во многих главах и подразделах этой книги просто не о чем было бы писать. К сожалению, за время подготовки рукописи к изданию Алексей Валериевич Забродин ушел от нас. Вечная память.

За предоставленную возможность тратить рабочее время на литературные упражнения мне, несомненно, следует в первую очередь благодарить моих ближайших сослуживцев по Испытательной лаборатории — М. В. Жердеву, Г. П. Савельева, В. Л. Орлова и С. А. Дбар, а также — А. В. Баранова. Конечно же, благодарен я им всем не только за это, вклад их в создание этой книги неизмеримо больше и шире.

Я безмерно признателен Д. Е. Фролову, а также моей сестре, А. О. Лацис, за помощь в литературном редактировании этой книги.

Наконец, слово особой благодарности хотелось бы произнести в адрес моих нынешних коллег по разработке нового суперкомпьютера — С. С. Андреева, С. А. Дбар и Е. А. Плоткиной. По «гамбургскому счету» они являются соавторами многих разделов второй части книги. Также хотелось поблагодарить Г. В. Карапетяна за помощь в оформлении документации одного из макетов, «по следам» разработки которого были написаны некоторые главы части II.

## ПРИЛОЖЕНИЯ

### Приложение 1. Параллельная реализация метода верхней релаксации с использованием MPI

Текст параллельной реализации метода верхней релаксации, основанной на конвейерном выполнении итераций.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
    static MPI_Request rq;
    static MPI_Status st;
/****/
#define MX 640
#define MY 480
#define NITER 10000
#define STEPITER 100
#define W 0.5
    static float f[MX][MY];
    static int rank, size, mx;
    static FILE *fp;
/****/
int main( int argc, char **argv )
{
    int i, j, n, m;
    double howlong;
/****/
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    mx = ((MX-2)+size-1)/size;
    if ( rank == (size-1) )
    {
        mx = (MX-2)-(size-1)*mx;
    }
    mx += 2;
    if ( !rank )
    {
        printf( "Solving heat conduction task on %d
                by %d grid by %d processors\n",
                MX-2, MY-2, size );
```

```

        fflush( stdout );
    }
/* Initial conditions: */
    for ( i = 0; i < mx; i++ )
    {
        for ( j = 0; j < MY; j++ )
        {
            f[i][j] = 0.0;
            if ( (i == 0) && (rank == 0) )
                f[i][j] = 1.0;
            if (j == 0) f[i][j] = 1.0;
            if ( (i == (mx-1)) &&
                (rank == (size-1)) )
                f[i][j] = 0.5;
            if (j == (MY-1)) f[i][j] = 0.5;
        }
    }
/* Iteration loop: */
    howlong = MPI_Wtime();
    for ( n = 0; n < NITER; n++ )
    {
        if ( !rank )
        {
            if ( !(n%STEPITER) )
                printf( "Iteration %d\n", n );
        }
/* Step of calculation starts here: */
        if ( rank > 0 )
            MPI_Recv( &f[0][1], MY-2,
                      MPI_FLOAT, rank-1, 100,
                      MPI_COMM_WORLD, &st );
        for ( i = 1; i < (mx-1); i++ )
        {
            for ( j = 1; j < (MY-1); j++ )
            {
                f[i][j] = W * ( f[i][j+1] + f[i][j-1] +
                                f[i-1][j] + f[i+1][j] )
                            *0.25
                            + (1-W) * f[i][j];
            }
            if ( ( i == 1 ) && ( rank > 0 ) )
                MPI_Isend( &f[1][1], MY-2,
                          MPI_FLOAT, rank-1, 100,
                          MPI_COMM_WORLD, &rq );
        }
        if ( rank > 0 ) MPI_Wait( &rq, &st );
        if ( rank < (size-1) )

```

```

    {
        MPI_Send( &f[mx-2][1], MY-2,
                  MPI_FLOAT, rank+1, 100,
                  MPI_COMM_WORLD );
        MPI_Recv( &f[mx-1][1], MY-2,
                  MPI_FLOAT, rank+1, 100,
                  MPI_COMM_WORLD,
&st );
    }
}

if ( !rank ) printf( "Elapsed time: %f sec\n",
                    (float)(MPI_Wtime()-howlong) );
/* Calculation is done, F array is a result: */
if ( rank == 0 )
{
    fp = fopen( "progreval.dat", "w" );
    fclose( fp );
}
else
{
    MPI_Recv( &n, 1, MPI_INT, rank-1,
              MPI_ANY_TAG,
              MPI_COMM_WORLD, &st );
}
fp = fopen( "progreval.dat", "a" );
for ( i = 1; i < (mx-1); i++ )
    fwrite( f[i]+1, MY-2, sizeof(f[0][0]), fp );
fclose( fp );
if ( rank < (size-1) )
    MPI_Send( &n, 1, MPI_INT, rank+1, 100,
              MPI_COMM_WORLD );
MPI_Finalize();
return 0;
}

```

Этот вариант программы работает точно так, как было объяснено в гл. 6, т.е. каждый процесс на очередном шаге расчета выполняет свою итерацию над своей порцией данных.

Для демонстрации понятия конвейерного параллелизма этого достаточно, но на практике метод организации конвейера вряд ли применим. Дело в том, что в реальных вычислительных приложениях обычно обрабатывается не одна, а несколько четырехугольных сеток (областей), причем края их являются «граничными условиями» друг для друга. Не вдаваясь в тонкости так называемого «распараллеливания границ», отметим, что при таком расчете устраивать конвейер по итерациям уже нельзя. Переходить к следующей итерации в области

можно только тогда, когда очередная итерация выполнена во всех областях, прилегающих к ней. Надо найти способ параллельной обработки каждой области всеми процессами в ходе выполнения одной итерации, как в методе Якоби.

Это осуществляется следующим образом. Каждая локальная порция области, обрабатываемая одним процессом, делится дополнительно на  $L$  вертикальных столбцов. Обработка локальной порции происходит по столбцам. Обмен данными с верхним и нижним соседом происходит для каждого столбца в отдельности таким же способом, как в приведенном выше варианте программы. Таким образом, внутри итерации организуется конвейер по столбцам, аналогичный конвейеру по итерациям в предыдущем варианте программы.

Эффективность такой реализации зависит от значения параметра  $L$ . В самом деле, если значение  $L$  мало, конвейер будет долго «разгоняться». Если же значение  $L$  велико, придется выполнять слишком много обменов короткими сообщениями. И то и другое ведет к росту накладных расходов. При автоматической реализации этого метода в системе программирования DVM [27] подходящее значение  $L$  выбирается автоматически, по результатам небольшого числа пробных итераций. При реализации с использованием MPI потребуется ручная «подгонка».

Описанный вариант параллельной программы приводится ниже.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
    static MPI_Request rq;
    static MPI_Status st;

/**/
#define MX 640
#define MY 480
#define NITER 1000
#define STEPITER 100
#define W 0.5
#define L 8
    static float f[MX][MY];
    static int rank, size, mx;
    static FILE *fp;

/**/
int main( int argc, char **argv )
{
    int i, j, n, m, is, ie, step;
    double howlong;

/**/
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

```

MPI_Comm_size( MPI_COMM_WORLD, &size );
mx = ((MX-2)+size-1)/size;
if ( rank == (size-1) )
{
    mx = (MX-2)-(size-1)*mx;
}
mx += 2;
if ( !rank )
{
    printf( "Solving heat conduction task on %d
            by %d grid by %d processors\n",
            MX-2, MY-2, size );
    fflush( stdout );
}
/* Initial conditions: */
for ( i = 0; i < mx; i++ )
{
    for ( j = 0; j < MY; j++ )
    {
        f[i][j] = 0.0;
        if ( (i == 0) && (rank == 0) ) f[i][j] = 1.0;
        if ( j == 0 ) f[i][j] = 1.0;
        if ( (i == (mx-1)) &&
              (rank == (size-1)) ) f[i][j] = 0.5;
        if ( j == (MY-1) ) f[i][j] = 0.5;
    }
}
step = MY/L;
/* Iteration loop: */
howlong = MPI_Wtime();
for ( n = 0; n < NITER; n++ )
{
    if ( !rank )
    {
        if ( !(n%STEPITER) )
            printf( "Iteration %d\n", n );
    }
}
/* Step of calculation starts here: */
for ( is = 1; is < (MY-1); is += step )
{
    ie = is + step;
    if ( ie > (MY-1) ) ie = MY-1;
    if ( rank > 0 )
        MPI_Recv( &f[0][is], ie-is,
                  MPI_FLOAT, rank-1, 100,
                  MPI_COMM_WORLD, &st );
    for ( i = 1; i < (mx-1); i++ )

```

```

{
    for ( j = is; j < ie; j++ )
    {
        f[i][j] = W * ( f[i][j+1] + f[i][j-1] +
                        f[i-1][j] + f[i+1][j] ) * 0.
                        + (1-W) * f[i][j];
    }
    if ( ( i == 1 ) && ( rank > 0 ) )
        MPI_Isend( &f[1][is], ie-is, MPI_FLOAT,
                    rank-1, 100, MPI_COMM_WORLD, &rq );
}
if ( rank > 0 ) MPI_Wait( &rq, &st );
if ( rank < (size-1) )
{
    MPI_Send( &f[mx-2][is], ie-is,
               MPI_FLOAT, rank+1, 100,
               MPI_COMM_WORLD );
    MPI_Recv( &f[mx-1][is], ie-is,
               MPI_FLOAT, rank+1, 100,
               MPI_COMM_WORLD, &st );
}
}
if ( !rank ) printf( "Elapsed time: %f sec\n",
                     (float)(MPI_Wtime()-howlong) );
/* Calculation is done, F array is a result: */
if ( rank == 0 )
{
    fp = fopen( "progrex.dat", "w" );
    fclose( fp );
}
else
{
    MPI_Recv( &n, 1, MPI_INT, rank-1,
               MPI_ANY_TAG,
               MPI_COMM_WORLD, &st );
}
fp = fopen( "progrex.dat", "a" );
for ( i = 1; i < (mx-1); i++ )
    fwrite( f[i]+1, MY-2, sizeof(f[0][0]), fp );
fclose( fp );
if ( rank < (size-1) )
    MPI_Send( &n, 1, MPI_INT, rank+1, 100,
               MPI_COMM_WORLD );
MPI_Finalize();
return 0;
}

```



## Приложение 2. Некоторые примеры реально используемых при логическом проектировании комбинационных схем

В качестве примера рассмотрим построение в базисе «И», «ИЛИ», «НЕ» двух реально используемых во многих сложных цифровых устройствах логических блоков — *дешифратора* и *мультиплексора* [47].

*Дешифратором из  $n$  в два в степени  $n$*  называется схема с числом входов, равным  $n$ , и числом выходов, равных двум в степени  $n$ .

Эта схема реализует очень простое преобразование входных сигналов в выходные, называемое дешифрацией входного кода.

Пусть как входы, так и выходы схемы пронумерованы от нуля. Если рассматривать  $n$  входов как  $n$ -значное число в двоичной системе счисления, считая  $i$ -й вход соответствующим разрядом этого числа, то на выходной стороне дешифратора равен единице тот и только тот выход, номер которого равен числу на входе. Остальные выходы должны быть равны нулю.

Рассмотрим, например, дешифратор из двух в четыре. Как следует из названия и определения, у него два входа и четыре выхода. Два входа можно рассматривать как двузначное двоичное число, принимающее значения в диапазоне от 0 до 3. Выходы же можно занумеровать от 0 до 3. Если на входе имеем число 2, то дешифратор должен сделать выходы с номерами 0, 1 и 3 равными нулю, а выход номер 2 — равным единице.

Для начала нам потребуется научиться рисовать схемы из вентилях. Вентили имеют стандартные обозначения, которые приводятся ниже. В правом столбце таблицы приводятся обозначения, принятые в отечественной схемотехнике во времена СССР, если они отличаются от используемых в настоящее время (рис. П2.1).

При изображении схемы в графическом виде следует также учитывать, что входы, если это возможно, должны располагаться слева, а выходы — всегда и только справа.

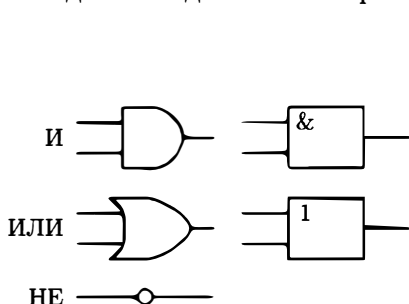


Рис. П2.1. Графические обозначения вентилях

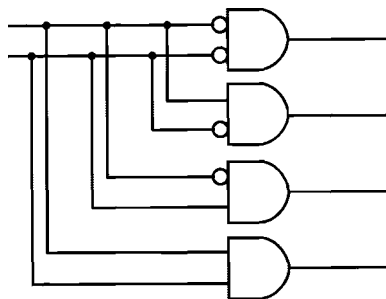


Рис. П2.2. Дешифратор из двух в четыре

Легко убедиться, что приведенная на рис. П2.2 комбинационная схема действительно реализует дешифратор из двух в четыре.

Дешифраторы могут применяться, например, в схеме процессора для выработки сигналов, разрешающих срабатывание различных блоков схемы в зависимости от кода операции в машинной команде.

**Мультиплексором  $k$  линий, шириной  $n$  каждая, в одну линию шириной  $n$**  называется устройство, которое подает на свой выход одну из  $k$  входных линий, если соответствующий сигнал разрешения равен 1. Под линией здесь понимается, вообще говоря, не единственный провод, а жгут проводов определенной ширины. Рассмотрим мультиплексор двух входных линий, шириной 3 каждая, в одну (рис. П2.3). Поскольку каждая входная линия состоит из трех проводов и снабжена дополнительным проводом для передачи сигнала разрешения, у нашего мультиплексора будет 8 входов. Выходов будет 3.

По предложенной схеме видно, что сигнал разрешения, равный 0, делает выходы вентилях «И» на соответствующей входной линии равными нулю. Поскольку такие выходы «собираются» вместе через вентили «ИЛИ», на выходе схемы остаются в точности те значения, которые были на единственной линии, сигнал разрешения которой равен 1.

Если ни один из сигналов разрешения не равен 1, то на выходе мультиплексора будут нули. Если же в некоторый момент времени единице

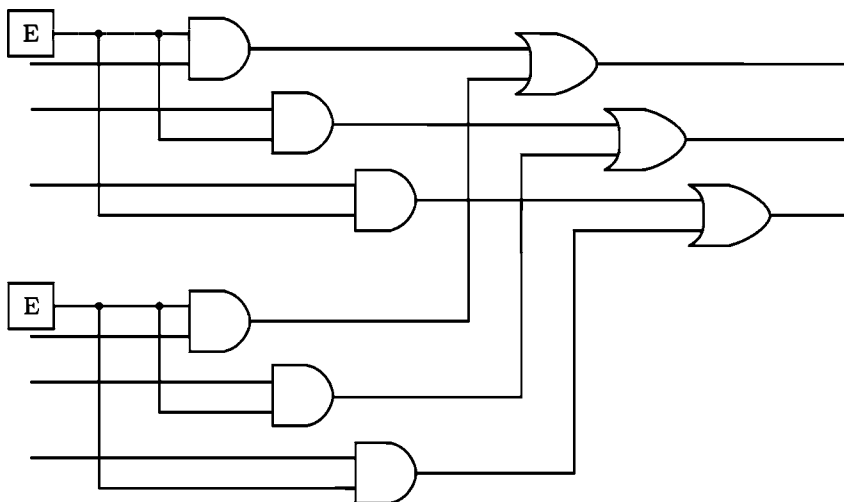


Рис. П2.3. Мультиплексор двух трехпроводных линий в одну (буквой «Е» обозначены линии разрешения. Под каждой из них показана трехпроводная линия, управляемая этой линией разрешения)

равен более чем один сигнал разрешения, на выходной линии окажется результат поразрядного логического «ИЛИ» всех линий, разрешенных в данный момент. Легко видеть, что если сигналы разрешения берутся, например, с некоторого дешифратора номера линии, то такой неприятности никогда не произойдет.

Оба представленных здесь логических узла допускают альтернативные варианты реализации. Так, любая комбинационная схема вообще может быть построена путем табличного задания функции алгебры логики, которую она реализует, в устройстве памяти с асинхронным чтением.

Устройства памяти с асинхронным чтением используются в современных FPGA как базисный элемент для построения комбинационных схем, т. е. вместо вентилях. Функционально такое устройство представляет собой произвольно адресуемый по индексу одномерный массив слов памяти определенной разрядности. Рассмотрим в качестве примера работу 4-словного устройства памяти с разрядностью слова в 1 бит.

Такое устройство имеет два входа, называемых линиями адреса, и выход, называемый линией чтения данных. Внутри устройства имеется массив из четырех ячеек, занумерованных от 0 до 3. Каждая ячейка хранит значение, в данном случае, состоящее из одного бита. Вопрос о том, как эти значения попадают в ячейки массива, для краткости рассматривать не будем. Просто допустим, что у нас есть возможность заранее записать в этот массив необходимые нам значения.

Пару значений на линиях адреса можно рассматривать, как двузначное двоичное число. Работа устройства памяти состоит в том, что значение ячейки с номером, равным этому числу, просто подается на выход. Например, если на входе имеется число 3 (оба входа равны 1), то на выходе немедленно оказывается значение, хранящееся в третьей ячейке массива.

Для того чтобы построить из таких устройств дешифратор из двух в четыре, нам потребуется четыре таких устройства, линии входов которых объединены, т. е. на входы всех устройств всегда подается одно и то же. Линии выходов четырех устройств сделаем выходами дешифратора. Для того чтобы такое образование стало дешифратором, нам достаточно записать в память каждого из устройств таблицу истинности соответствующего выхода.

Например, мы знаем, что при нулевых значениях входов дешифратора нулевой выход должен принять значение 1, а остальные выходы — значение 0. Следовательно, в нулевой бит нулевого устройства памяти мы запишем единицу, а в нулевые биты остальных устройств памяти — нули. Соответственно, в первые биты устройств памяти с номерами 0, 2 и 3 запишем нули, а в первый бит первого устройства памяти — еди-

ницу. Этим мы добьемся того, что при появлении на входах кода «01», т.е. двузначного двоичного числа «1», единице будет равен первый выход, а остальные — нулю, что и требовалось. С двумя оставшимися битами всех устройств памяти поступим аналогично. Тот факт, что мы строим именно дешифратор, для общего хода рассуждений значения не имеет. Ровно тем же способом мы могли бы построить любую комбинационную схему с двумя входами и четырьмя выходами, если нам известна ее таблица истинности. Тот факт, что в современных FPGA на аппаратном уровне используется именно такой, а не вентильный, базис, конечно, не означает, что структурное логическое проектирование для них следует вести именно в этом базисе. Все базисы эквивалентны, и САПР легко переводит один в другой при необходимости. Так что разработчик схемы «в регистрах» вполне может считать, что в его распоряжении привычные «И», «ИЛИ», «НЕ» (рис. П2.4).

Отметим, что наш мультиплексор, если бы мы захотели реализовать его таким способом, потребовал бы трех устройств памяти из 256 бит каждое. В самом деле, выходов у него три, а число всевозможных комбинаций значений на восьми входах, для каждой из которых надо запомнить требуемое выходное значение, равно 256 (2 в степени, равной числу входов).

Мы построим альтернативный вариант реализации мультиплексора другим способом, с использованием еще одного базисного элемента —

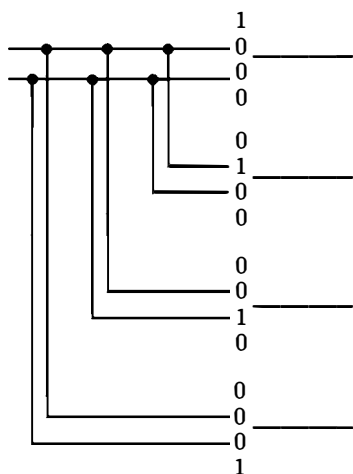


Рис. П2.4. Дешифратор из двух в четыре на базе таблиц истинности, хранимых в асинхронной памяти

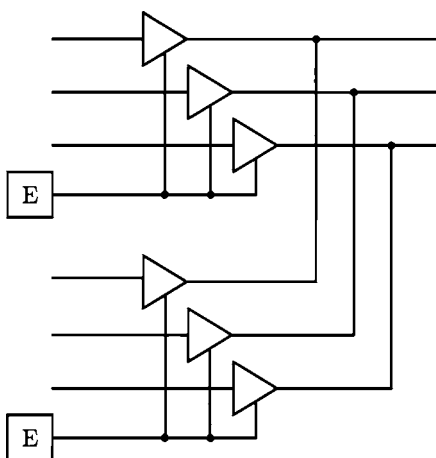


Рис. П2.5. Мультиплексор на тристабильных формирователях, или шина

**тристабильного формирователя.** Этот элемент не является логическим, т. е. построить отдельно взятый тристабильный формирователь из вентилях в принципе нельзя. В то же время комбинационные схемы, выполненные на таких элементах, конечно, можно построить и без их помощи.

Тристабильный формирователь представляет собой электронный ключ. У него два логических входа — управляемый сигнал и сигнал разрешения (при графическом изображении его рисуют входящим в пиктограмму формирователя снизу). Выход тристабильного формирователя логическим не является, поскольку принимает одно из трех (а не двух) значений: 0, 1 или «разрыв линии», или «третье состояние». Если сигнал разрешения равен 1, то выход принимает логическое значение, равное значению входа управляемого сигнала, т. е. элемент ведет себя так, как если бы управляемый сигнал и выход были просто соединены проводом. Если же сигнал разрешения равен 0, то выход переходит в «третье состояние», т. е. ведет себя на электрическом уровне так, как будто бы провод, соединяющий управляемый сигнал с выходом, был перерезан. Это свойство тристабильного формирователя позволяет объединять выходы нескольких тристабильных формирователей, т. е. подавать несколько таких выходов на один и тот же вход некоторого логического элемента. Конечно, схема управления сигналами разрешения таких объединенных по выходам формирователей должна быть построена так, чтобы никакие два из них не были одновременно равны 1. В противном случае схема некорректна электрически, и может, вообще говоря, просто сгореть.

Схема мультиплексора на тристабильных формирователях приведена на рис. П2.5.

Отметим, что реализованные таким способом устройства с функцией мультиплексора в схемотехнике принято называть *шинами* [47]. Классические системные шины компьютеров недавнего прошлого обеспечивают подключение многих устройств к единой магистрали передачи данных именно таким способом.

Тристабильные формирователи включаются в состав базисных элементов современных FPGA. При необходимости САПР может использовать их напрямую, но может и обходиться без них, заменяя схемы на тристабильных формирователях эквивалентными схемами на базе вентилях (например, заменяя шину мультиплексором).

Если ни один из входов шины не разрешен, то на линиях шины, в отличие от случая мультиплексора, будет не ноль, а значение, не определенное электрически. Существует способ, который, как и понятие тристабильного формирователя, объясняется в электрических терминах, «доопределить» это значение некоторой константой. О таких значениях говорят как о «слабом нуле» и «слабой единице» соответственно.

Для выражения описанной здесь техники на уровне логического проектирования, например, при записи функционирования схемы на языке VHDL, необходимо расширить набор значений логического сигнала. Если для проектирования в терминах вентилей или устройств памяти с асинхронным чтением нам достаточно было двузначной логики, со значениями «0» и «1», то теперь нам известно, как минимум, еще три необходимых значения: «третье состояние», «слабый 0» и «слабая 1».

### Приложение 3. Основные понятия программистской модели VHDL

Программистская модель языка существует вне зависимости от того, как именно построено разъяснение языка будущим пользователям. Однако изложение программистской модели в явном виде обычно подразумевает, что рассказ о языке ориентирован на «традиционных» программистов. Но зачем вообще это делать, почему о языке VHDL приходится рассказывать именно программистам?

Из материала, изложенного в подразделе, посвященном проблемам логического проектирования специализированных вычислителей, следует простой и, в общем, довольно безрадостный вывод. Такое проектирование прекрасно подготовлено развитием как аппаратных, так и программных средств с технической стороны, но должно выполняться специалистами, которых в настоящее время не готовит ни одно учебное заведение России. Речь идет о *системных архитекторах специализированных вычислителей*, т. е. о тех людях, которым в среднесрочной перспективе придется работать в тесном контакте с прикладными программистами, заменяя своим трудом не созданные пока необходимые технологии автоматической разработки таких вычислителей.

В самом деле, спроектировать схему по готовым функциональным спецификациям могут многие. Сложнее придумать эти спецификации таким образом, чтобы схема действительно соответствовала «духу и букве» вычислительного алгоритма, действительно содержала в себе внутренне присущий алгоритму параллелизм, выражала в своей структуре свойственную алгоритму специализированную структуру памяти. Более того, из всех имеющихся алгоритмов и численных методов решения поставленной задачи требуется выбрать именно тот, который лучше всего «ложится» именно в схему, а не на универсальный процессор.

Словом, системный архитектор специализированных вычислителей должен сочетать в себе профессиональное владение схемотехникой и

знание численных методов, достаточное, как минимум, для уверенного общения с математиками-прикладниками. Система профессионального компьютерного образования 60 лет уверенно шла к тому, чтобы максимально разделить по разным областям профессиональной подготовки эти две области знаний. В результате сегодня представить себе схемотехника, способного правильно подобрать разностную схему и итерационный алгоритм, так же невозможно, как специалиста по численным методам, способного хотя бы просто представить себе, в чем вообще заключается отличие процессора от вентиляционной матрицы.

Можно, конечно, надеяться, что необходимые специалисты будут подготовлены, когда наступит такая потребность. Проблем здесь всего две.

Во-первых, эта потребность не наступит, пока некоторая «критическая масса» системных архитекторов специализированных вычислителей не приступит к работе и не выдаст первые обнадеживающие результаты.

Во-вторых, чтобы подготовить специалистов такого профиля в меру возникающей потребности, надо иметь соответствующий учебник.

С другой стороны, ситуация, когда фундаментальный прорыв в области вычислительных технологий может быть выполнен целиком «в голове», без каких-либо ссылок на отставание в технологической базе и слабое финансирование строительства новых заводов и технопарков, уникальна. Такое в технике бывает не часто, окно возможностей скоро закроется. Это будет означать, что данную область знаний создадут без нас.

Словом, проблема воспитания первого поколения системных архитекторов указанного профиля сама по себе является важнейшей научно-методической задачей. Для начала хотелось бы понять, из кого можно было бы наиболее успешно готовить таких специалистов.

Теоретически существует три категории специалистов:

- схемотехники;
- системные программисты;
- прикладные программисты, специалисты в области численных методов математического моделирования.

При рассмотрении проблемы в практической плоскости нетрудно прийти к выводу, что наибольшую надежду на успех дадут системные программисты, профессионально наиболее подготовленные к построению новых формализмов, к формулированию новых логических сущностей. При освоении не существующей пока профессии системного архитектора специализированных вычислителей им придется «всего лишь» проявить эту свою способность в новой, непривычной предметной области. Конечно, довольно большое количество схемотехников, самостоятельно освоивших программирование, также является важнейшим

источником «кандидатов в системные архитекторы». Однако специалисты этого профиля обычно знают о прикладной математике и численных методах еще намного меньше (если такое вообще возможно), чем «системщики», что представляет собой серьезную трудность. Так что задача подготовки будущих системных архитекторов именно из системных программистов остается, как минимум, актуальной.

Ясно, что первым и очень важным шагом в такой подготовке системных архитекторов является изучение языка схемотехнического проектирования, например, VHDL [57, 58]. Ведь именно через программистскую модель языка схемотехнического проектирования системный архитектор получает базовое представление о материале, с которым ему придется работать. В специальной литературе на сегодня накоплен довольно богатый опыт разъяснения основ VHDL схемотехникам, привыкшим к структурному проектированию «в регистрах». Главная методическая трудность здесь состоит в освоении самого алгоритмического подхода, в привыкании к тому, что схема может быть описана не в терминах структуры, а в терминах алгоритма. Казалось бы, при разъяснении основ VHDL программисту эта трудность точно не возникает, но возникают другие сложности.

Всего несколько лет назад автору пришлось побывать в роли системного программиста, осваивающего VHDL. Хотелось бы поделиться опытом того, в чем состоят типичные ошибки и трудности, что остается не прочитанным или не понятым с первого раза, когда к изучению VHDL приступает именно системный программист с характерной именно для этой профессии «картиной мира».

Учебников по основам VHDL достаточно высокого качества существует много [57, 58], и предлагаемый ниже текст не претендует на то, чтобы быть заменой или кратким конспектом одного из них. Скорее речь может идти о комментариях к типичному учебнику по основам VHDL, написанных с точки зрения системного программиста и для системных программистов.

Язык VHDL — алгоритмический и императивный. Подобно языкам фоннеймановского типа, он служит для записи совершенно определенных последовательностей действий, которые должна выполнить описываемая текстом схема. Действия, записываемые в программе, состоят в присваивании значений именованным объектам, очень похожим на переменные фоннеймановской модели.

Модель памяти VHDL базируется на двух понятиях — *сигнала* и *переменной*. При поверхностном знакомстве с языком может сложиться впечатление, что именно переменные — те самые объекты, в терминах которых в основном и следует записывать алгоритм. Такое представление является глубочайшей ошибкой и верным путем к написанию текстов, не соответствующих вообще никакой схеме. Переменные в



реальных текстах на VHDL встречаются крайне редко, и пока не будем о них говорить. Сосредоточимся на сигналах.

Сигнал в VHDL — это абстракция провода, на котором имеется напряжение логического уровня, логический ноль или логическая единица. Таким образом, сигнал — это именованный объект, имеющий двоичное значение, подобно переменной в фоннеймановских языках программирования. В полной аналогии с фоннеймановским программированием сигналы должны быть объявлены в начале программы, до их использования, а использование их состоит, в частности, в присваивании им новых значений. В некоторых языках схемотехнического проектирования предусматриваются специальные виды программных объектов для обозначения, с одной стороны, соединений выходов одних элементов с входами других, т.е. проводов в буквальном смысле этого слова, и, с другой стороны, «проводов», обладающих свойством хранить ранее присвоенное значение и изменять его при новом присваивании, т.е. триггеров. VHDL к таким языкам не относится. В нем для обозначения «проводов» обоих видов применяется один и тот же вид объектов — сигналы, а вопрос о том, стоит ли за тем или иным сигналом триггер, или же это просто провод, решается транслятором автоматически, по тексту программы. С точки зрения человека, обладающего опытом фоннеймановского программирования, такое обобщение очень удобно. Можно смело считать, что сигнал аналогичен переменной в традиционном программировании, и не вдаваться в дальнейшие детали.

Сигнал может быть одиночным, т.е. иметь битовое значение, и векторным, т.е. иметь значение вектора битов фиксированной длины. Второй случай, как легко себе представить, соответствует не одному проводу, а жгуту проводов, или же не одному триггеру, а регистру определенной разрядности.

Примеры объявления сигналов:

```
signal a: bit;  
signal b: bit_vector(31 downto 0);
```

В первом случае объявлен одноразрядный сигнал под именем “a”, во втором — 32-разрядный сигнал с именем “b”, разряды в котором занумерованы по убыванию от 31 до нуля.

Примеры присваивания:

```
a <= '0';  
b(18) <= a;  
b(31 downto 24) <= "01010101";
```

Помимо векторов существуют также массивы сигналов (одиночных или векторных). Очень важно понимать, что индексация как векторов, так и массивов может быть только статической. Такая очевидная и привычная для программиста возможность, как, например, «присвоить

единицу тому биту сигнала *b*, номер которого хранится в данный момент в сигнале *c*», в языках схемотехнического проектирования, вообще говоря, не предусмотрена. В модели вентилей и регистров, на которые ориентированы эти языки, эта возможность не является базовой подобно тому, как в С не является базовой возможность присвоить значение полю записи, хранящейся в СУБД. Схемы, реализующие динамическую индексацию, в терминах VHDL можно записать как довольно сложные фрагменты программ, но базовыми конструкциями языка они не являются. Из этого утверждения бывают исключения, которые кратко будут изложены далее.

Наличие в операторе объявления сигнала слова «*bit*» наверняка уже удивило некоторых внимательных читателей. В самом деле, какой еще тип, кроме типа «*бит*», можно придумать для сигнала в цифровой схеме? В действительности существуют и очень часто используются другие типы, например тип сигнала «*std\_logic*», способный принимать несколько других значений, помимо нуля и единицы. Речь идет, конечно, не о возможности проектирования схем для троичных и квантовых компьютеров, как быть может подумали некоторые из читателей, а всего лишь о возможности выражения в схемах функциональности таких не вполне логических элементов, как тристабильные формирователи. Например, среди допустимых значений сигнала типа «*std\_logic*» имеется значение «третье состояние», которое соответствует просто «обрыву линии», выполненному путем отключения тристабильного формирователя на выходе этой линии. Обнаружив в программе присваивание сигналу такого значения, транслятор включит в формируемую схему необходимый тристабильный формирователь. Располагая же сигналом типа «*bit*», автор программы просто не смог бы донести до транслятора необходимую информацию о составе требуемой схемы. Использование сигналов типа «*std\_logic*» для записи схем, полностью выразимых в терминах логических элементов, никак не усложняет итоговую схему по сравнению с использованием сигналов типа «*bit*».

Модель выполнения программы основывается, в первую очередь, на фундаментальном понятии единственности источника сигнала. Понятие это является абстракцией того факта, что ни в какой схеме, вообще говоря, не должно быть нескольких выходов, подключенных к одному входу. Выход полностью определяет состояние входа, к которому он подключен, и «вдвоем» этого делать нельзя. О таких хорошо известных старым схемотехникам исключениях из этого правила, как знаменитое «монтажное ИЛИ», здесь говорить не будем.

С точки зрения модели выполнения программа распадается на некоторое количество *процессов* и *часть программы вне процессов*.

Часть программы вне процессов соответствует комбинационной части схемы, т. е. жестко выполненным при построении схемы соеди-

нениям выходов одних элементов со входами других. Например, если в этой части программы написано:

```
b <= c and d;
```

то транслятор сгенерирует вентиль «И», входами которого являются сигналы «с» и «d», а выход представляет собой сигнал «b». Сигналам, которым присваиваются значения в этой части программы, можно присваивать значения только один раз. В самом деле, выход данного вентиля либо называется «b», либо не называется, никакого третьего варианта здесь быть не может.

Понятие процесса в VHDL объединяет два формально сходных, но совершенно разных по существу схемных понятия, записываемых с использованием одной и той же языковой конструкции. По устоявшейся традиции, о происхождении которой кратко будет сказано ниже, в учебниках эти два разных понятия принято всячески смешивать, а различие между ними затушевывать, в то время как для реального понимания языка следовало бы поступать наоборот.

Формально процесс представляет собой бесконечный цикл, в котором выполняется тело процесса. Очередной виток этого цикла запускается всякий раз, когда один из сигналов, перечисленных в заголовке процесса, меняет свое значение. Каждому сигналу, которому не присвоено значение вне процессов, можно присваивать значения в теле процесса, но только одного, и только один раз за одно срабатывание (виток цикла).

Легко видеть, что в это общее формальное определение действительно укладываются две совершенно разные по смыслу схемные конструкции.

Первая из них — это не более чем вычурная, не очень естественная абстракция комбинационной логики, той, которую можно было бы записать и вне процесса.

Например:

```
process SAMPLE_PROC ( b, c, d ) is
begin
    if c = '1' then
        a <= b;
    else
        a <= d;
    end if;
end process SAMPLE_PROC;
```

Вместо этого процесса можно было бы написать просто:

```
a <= ( b and c ) or ( d and (not c) );
```

Понятие «бесконечного цикла», который «срабатывает при каждом изменении одного из сигналов, перечисленных в заголовке», в данном

случае представляется довольно искусственным, только затуманивающим существо дела. В результирующей схеме не появится никакой дискретно работающей логики, про которую можно было бы сказать, что за данную секунду она сработала, скажем, 100 миллионов раз.

Вторая схемная конструкция, которую без использования понятия процесса записать крайне трудно, если вообще возможно — это часть схемы, управляемая некоторым тактовым генератором.

Например:

```
process SAMPLE_PROC ( Clk ) is
begin
    if Clk'event and Clk = '1' then
        if c = '1' then
            a <= b;
        else
            a <= d;
        end if;
    end process SAMPLE_PROC;
```

Признаком того, что сигнал Clk является не просто логическим сигналом, но выходом тактового генератора, а тело процесса срабатывает дискретно, является третья строка примера. Синтаксически это просто условный оператор, на самом же деле — неделимая языковая конструкция, «видя» которую, транслятор понимает, что речь идет именно о триггерной, а не комбинационной логике. Если бы программист добавил, следуя формальным правилам языка, в заголовок этого процесса дополнительные сигналы по аналогии с первым примером, транслятор бы их при синтезе схемы просто проигнорировал.

Попытка авторов языка объединить в рамках единой языковой конструкции, формально определяемой и разъясняемой единообразно, два таких разных содержательных понятия предметной области, как комбинационная и триггерная логика, конечно, язык вовсе не украшает, но имеет свое объяснение. Единое понятие процесса, не разделяемое на два рассмотренных выше случая, является довольно естественным, если говорить о программном моделировании работы схемы средствами универсального процессора. Тем не менее, при написании на VHDL текстов, предназначенных для синтеза, а не только для моделирования, два рассмотренных случая применения языковой конструкции «процесс» совершенно необходимо четко различать. Один из способов такого различения, применяемый автором, состоит в том, чтобы просто никогда не пользоваться «комбинационными процессами». Далее при рассмотрении понятия процесса в VHDL будем говорить только о триггерной логике.

Вернемся к нашему второму примеру.

Строки с четвертой по седьмую описывают те действия, которые должны происходить по каждому переднему фронту сигнала Clk, т. е.

при его переходе из 0 в 1. Эти действия состоят в присваивании сигналу «а» значения либо сигнала «b», либо сигнала «d», в зависимости от того, чему равно значение сигнала «с».

Присваивание сигналу «а» предусмотрено в этом процессе, следовательно, ни вне процессов, ни в каком-либо еще процессе этой программы, выполнять присваивание сигналу «а» уже нельзя. Очевидно, в итоговой схеме сигнал «а» станет триггером, все тактирование и управление которым должно быть сосредоточено в одной части схемы, тактируемой совершенно конкретным генератором синхроимпульсов. Разные же процессы, вообще говоря, тактируются и управляются по-разному.

Обратим внимание также на то, что на каждом витке цикла может сработать не более одного оператора присваивания нового значения сигналу «а». Например, запись:

```
if c = '1' then
  a <= b;
else
  a <= d;
end if;
if e = '0' then
  a <= f;
end if;
```

абсолютно недопустима. Это связано с тем, что время в схеме течет дискретно, за пределами актов срабатывания тела процесса никаких действий быть не может, а сам акт срабатывания представляет собой момент, не имеющий длительности. Из этого же, в свою очередь, следует, что все тело процесса, включая все проверки и все присваивания, выполняется одновременно! Но нельзя же, в самом деле, одновременно присвоить триггеру «а» и значение «f», и значение «b». Следовательно, в описываемой модели выполнения данная запись является недопустимой.

Сформулируем еще раз полностью принцип единственности источника значений сигнала:

*каждый сигнал может получить значение либо ровно один раз вне процессов, либо в теле ровно одного процесса, не более одного раза за каждое срабатывание тела процесса.*

Этот принцип является удивительно удачной, простой, но точной абстракцией свойств аппаратуры, построенной из логических элементов.

Присваивание значений сигналам в рамках процесса является отложенным на один виток выполнения цикла. Это — абстракция того факта, что тактируемому триггеру нужно время, равное одному такту, чтобы запомнить поданное на него значение. Таким образом, запись:

```

c <= '1';
if c = '0' then
    .....

```

вовсе не является бессмысленной. Она означает: «Сделать триггер «с», начиная со следующего срабатывания этого процесса, равным единице. Если же сейчас он равен нулю, то....». В дискретном времени, собственно, иначе и быть не может. Вслушаемся мысленно в типичное объяснение смысла оператора присваивания фоннеймановских языков: «Сделать левую часть равной тому, чему сейчас равна правая». Здесь почти явно присутствует отсылка к понятию времени. Если же время дискретно, то, естественно, предписание «сделать» может относиться только к следующему моменту, ведь настоящий уже наступил.

В заключение попробуем прояснить вопрос, не раз ставивший в тупик новичков при освоении VHDL, причем вовсе не только программистов. Вопрос также касается понятия времени в модели выполнения VHDL.

Пусть время дискретно и, следовательно, все тело цикла — процесса выполняется одновременно. Но ведь тело процесса содержит условную логику. Значит, проверки, заданные в условных операторах, выполняются сначала, а соответствующие присваивания (или другие, вложенные проверки) — потом. Так все-таки, «потом» или «одновременно»?

Все тело цикла выполняется одновременно, включая условную логику, проверки любой сложности и любого уровня вложенности. При этом условная логика срабатывает мгновенно, но совокупный результат этих проверок таков, как будто бы они выполнялись в порядке вложенности. Таким образом, порядок выполнения программы внутри одного такта — это не порядок срабатывания схемы, а, в некотором смысле, «порядок построения» схемы, которая, будучи построена, срабатывает мгновенно, как единое целое. В действительности следует считать, что для каждого присваивания в теле процесса строится одна большая комбинационная, т.е. мгновенно срабатывающая, схема, единица на выходе которой разрешает выполнение соответствующего присваивания. Порядок же записи условной логики является для транслятора руководством по построению этой схемы.

Опыт автора показывает, что именно эта особенность модели выполнения, принятой в VHDL, дается осваивающим его специалистам особенно тяжело.

Из уже рассказанного должен быть ясен ответ на вопрос, который не очень внимательный читатель, возможно, готов задать, а именно — на вопрос о «go to». Этого вопроса в VHDL нет, как и самого «go to». И не потому, что авторы языка как-то особенно ревностно относятся к принципам структурного программирования, а просто потому, что в этой модели выполнения нет понятия «текущего, выполняемого в дан-

ный момент места программы». Все места программы выполняются одновременно во все моменты.

В языке предусмотрены понятия оператора цикла и вызываемых функций, синтаксически похожие на аналогичные понятия фоннеймановских языков. В фоннеймановских языках оба эти понятия определяются в терминах модели последовательного выполнения операторов программы, которые, как мы только что видели, к модели выполнения VHDL не применимы. Циклы и функции VHDL являются просто макросами. Цикл «разворачивается» во время трансляции, при этом каждый виток цикла превращается в отдельный фрагмент схемы, выполняющийся в рамках процесса, в котором записан цикл. Вызов функции превращается при трансляции в подстановку на место вызова того фрагмента схемы, который получился при трансляции тела функции при заданных значениях аргументов.

Таким образом, многое из того, что в фоннеймановской программе происходит во времени, в VHDL просто «разворачивается» в пространстве, т.е. приводит к генерации соответствующих фрагментов схемы по так или иначе параметризованным «заготовкам».

Функциональным аналогом понятия фоннеймановской подпрограммы является в VHDL понятие компонента. Вызову подпрограммы соответствует конкретизация компонента (port map).

По смыслу компонент — это отдельно записанная и, возможно, отдельно транслируемая схема, связанная с внешним миром совершенно конкретным набором входных и выходных сигналов — «внешним разъемом». Конкретизация компонента — это «впаивание» экземпляра такой схемы с подключением к ее «внешнему разъему» собственных сигналов. Записывается такое «впаивание» вне процессов, поскольку действие это чисто комбинационное. Те или иные сигналы разрабатываемой схемы либо присоединены к «разъему» некоторого экземпляра указанного компонента, либо нет.

Наш обзор модели программирования VHDL практически завершен. В него совершенно сознательно не включены сведения по синтаксису языка, который, кстати, простым и лаконичным назвать трудно. О синтаксисе, как и о прочих подробностях, среди которых немало достаточно важных, можно прочитать в соответствующих учебниках [57, 58]. В заключение хотелось бы сделать еще несколько замечаний.

Помимо сигналов, которые, как мы убедились, выполняют в VHDL ту же роль, что переменные в фоннеймановском программировании, в VHDL есть дополнительные объекты, называемые переменными. Они имеют те же типы, что и сигналы, тоже объявляются и присваиваются. Присваивание переменным записывается не так, как присваивание сигналам, «:=», а не «<=». Переменные VHDL — это довольно странные программные объекты. На них не распространяется модель выполнения

VHDL — они «умудряются» подчиняться фоннеймановской модели выполнения, т.е. присваивания им «выполняются» последовательно, по ходу текста программы, без задержки на такт, и могут быть многократными.

Как мы убедились выше, оборудованию, в которое транслируются тексты на VHDL, эта модель выполнения категорически не соответствует. Следовательно, аппарат переменных может быть (и действительно является) лишь вспомогательным средством оформления текста программы. Всякая попытка воспользоваться им так, как если бы триггеры действительно умели переключаться за один такт много раз, т.е. попытка записать с помощью переменных действия, не выражимые в принципе в терминах сигналов, ни к чему хорошему не приведет. Получится либо текст, который невозможно оттранслировать, либо текст, парадоксально не соответствующий замыслу программиста. Чтобы пользоваться этим аппаратом по назначению, не запутывая себя неосуществимой возможностью «на минуточку сменить модель выполнения», следует придерживаться при использовании переменных некоторых простых правил. Первое правило очевидно: переменными, по возможности, вообще не пользоваться. Второе правило: рассматривать их не как программные, а как алгебраические переменные, т.е. использовать исключительно в следующем стиле: «Обозначим выражение «a and b or not c» в данном месте программы как переменную X, и далее по тексту программы, впредь до переобозначения символа X, будем для краткости вместо «a and b or not c» везде писать X».

Упомянутая нами выше аналогия между понятием компонента и понятием подпрограммы (или функции) фоннеймановских языков распространяется также на библиотеки стандартных компонентов. Стандартные, оттранслированные отдельно, распространяемые не в исходном, а в «объектном» виде готовые фрагменты схем на уровне языка оформляются как компоненты. Для типовых логических узлов, включаемых в некоторые FPGA в готовом виде, наряду с логическими элементами также существуют специальные библиотечные компоненты. При включении в программу «обычного» библиотечного компонента в соответствующем месте схемы строится схема из логических элементов, которая содержится в «объектном модуле» компонента. Если же включается (конкретизируется) специальный библиотечный компонент, то вставляется готовый логический узел.

Ясно, что наборы готовых логических узлов в разных FPGA бывают разными, как и компоненты, используемые для ссылки на эти узлы. Чтобы обеспечить, по возможности, переносимость текстов на VHDL с одних типов FPGA на другие, и при этом не потерять возможность использования сходных по функциям готовых логических узлов, разработчики трансляторов VHDL иногда применяют следующий «фокус».



Придумывается канонический для данного логического узла фрагмент текста на VHDL, который описывает функцию этого логического узла. Например, регистра сдвига, или блока двухпортовой памяти. Транслятор снабжается «распознавателем» в тексте исходной программы «похожих» фрагментов. Если такой фрагмент, похожий, с точки зрения транслятора, удастся обнаружить в тексте программы, то транслятор поступает с ним, как с обращением к специальному библиотечному компоненту, т.е. не порождает обычным порядком схему из вентилях, а вставляет готовый узел. При этом в описании этой версии транслятора написано что-то вроде: «Если Вы хотите, чтобы Ваша схема использовала готовые блоки типа X, но не была привязана к данному типу FPGA, то вместо конкретизации специального библиотечного компонента BLOCK\_X напишите такой текст: .....».

В таких «условных», не транслируемых в логические элементы, фрагментах текста иногда разрешается делать то, чего в «обычной» части текста программы делать нельзя или крайне нежелательно. Например, в «условном» описании готового блока памяти можно адресовать элемент массива по динамическому индексу — ведь для готового блока памяти, в отличие от набора логических элементов, такая возможность является базовой.

Основная сложность использования такого рода «фокусов» в том, что они незначительно упрощают процесс разработки конкретных текстов, при этом значительно запутывая понимание системы в целом. В самом деле, как понять, достаточно ли похож данный фрагмент текста на типовой, транслируемый в готовый блок? Информация об этом приеме трансляции приведена здесь не в качестве примера привлекательной возможности, которой следует всячески пользоваться, а, в основном, для сведения при чтении документации. Автор настоящего текста в своих схемах такой возможностью старается не пользоваться. Для изоляции и минимизации частей схемы, которые потребуются переписать при смене типа FPGA, гораздо лучше пользоваться общепринятыми приемами структурного проектирования.

#### **Приложение 4. Краткий обзор возможностей САПР XPS**

В прил. 3 было рассказано об основных «подводных камнях», ожидающих программиста на пути освоения языка VHDL. Но, как мы знаем, для практической работы помимо языка очень важно владеть операционной обстановкой в целом. Программист, знающий язык C, но не владеющий в достаточной мере системой Windows как пользователь и совсем не знающий конкретных кнопок интегрированной системы разработки программ, вроде Microsoft Visual Studio, будет довольно

беспомощен в общении с компьютером, оснащенным именно этой системой программирования, по крайней мере, первое время. Операционная обстановка САПР логического проектирования сильно отличается от операционной обстановки системы разработки традиционных программ, причем отличается качественно, на уровне базовых понятий. Ниже приводится беглый обзор «картины мира» той САПР, с которой пришлось в свое время начинать работу автору этой книги [48].

Как и в Приложении, посвященном обзору модели программирования VHDL, не будем стремиться подменить штатную документацию, а дадим лишь общие комментарии, которые должны помочь начинающему пользователю лучше разобраться в структуре этой документации.

XPS — Xilinx Project Studio — САПР, базирующаяся на понятии *системы на кристалле* (*SoC — System on a Chip*), т.е. однокристалльного компьютера.

Фундаментальная проблема системы разработки как программ, так и схем — в необходимости обеспечить «самый верхний уровень». В традиционном программировании программа на С начинается с функции «main». Она может вызывать другие функции, те — тоже, но кто вызовет саму функцию main? Откуда брать исходные данные, куда писать результаты? В компьютерах общего назначения в роли такого «верхнего уровня» выступает операционная система.

При разработке цифровых схем проблема обеспечения «самого верхнего уровня» стоит не менее остро. Кто подаст на схему входные сигналы, как увидеть, что на выходе? В технологии систем на кристалле роль «самого верхнего уровня» для создаваемых устройств выполняет традиционный компьютер, реализованный внутри кристалла. Создаваемая схема является его внешним устройством. Бортовой компьютер кристалла оснащается операционной системой, возможно, совсем примитивной, но все же допускающей запуск программ, написанных на С. Эта программа и должна, как минимум, выполнить для создаваемого устройства роль «самого верхнего уровня», т.е. произвести начальный пуск устройства, возможно снабдить его исходными данными и/или прочитать результаты, после чего, например, напечатать их на экране. В более простых САПР разработчик цифровой схемы должен был еще до начала разработки озадачиться физическим созданием «отладочного станка» для такой схемы. В случае системы на кристалле роль «отладочного станка» берет на себя тестовая программа, выполняющаяся на бортовом процессоре.

Конечно, роль бортового процессора не обязана ограничиваться реализацией «отладочного станка». Она может меняться в широких пределах, от практического отсутствия какой-либо роли для случаев, когда «самый верхний уровень» уже реально существует, до полноценного взаимодействия бортового процессора со своим специализиро-

ванным устройством не только при отладке, но и при работе созданной схемы. Однако в любом случае проектируемая схема создается не «в безвоздушном пространстве», а в качестве «начинки» внешнего устройства, входящего в состав системы на кристалле. Сама же система включает в себя помимо разработанной схемы инфраструктуру компьютера (процессор, память, систему периферийных шин), ОС этого компьютера и программу, выполняемую в рамках этой ОС.

Таким образом, работа над проектом XPS начинается с создания компьютера. В качестве процессора в старших моделях FPGA Xilinx выступает готовый блок процессора PowerPC в составе FPGA. В моделях FPGA, в которых такой блок не предусмотрен, используется реализованный в логических элементах процессор MicroBlaze. Это 32-разрядный процессор, специально придуманный для использования в этом качестве [48].

Системы на базе обоих процессоров оснащаются одинаковой системой периферийных общих шин, спроектированных в IBM. Эти шины реализованы полностью в логических элементах, никакие специальные готовые блоки при их изготовлении не используются [48].

В современных FPGA Xilinx имеется довольно большой объем (до мегабайта) оперативной памяти, реализованной в качестве готовых логических блоков в составе кристалла. Из этой памяти можно делать как оперативную память создаваемого компьютера, так и процессорную кэш-память. Скорость ее работы — как у регистров, изготовленных из логических элементов. При наличии на используемой инструментальной плате внешней оперативной памяти ее также можно использовать в качестве оперативной памяти создаваемого компьютера. В этом случае контроллер такой памяти изготавливается из логических элементов.

Проектирование пользователем конкретной конфигурации компьютера из упомянутого выше «материала» происходит в интерактивном режиме, без привлечения технологий схемотехнической разработки. На этом этапе не требуется ничего похожего на рисование схемы «в регистрах» или написание текстов на VHDL. Пользователь просто заполняет предлагаемые ему формы, в которых требуется указать состав и некоторые параметры компьютера, например, тактовую частоту, наличие и размер кэш-памяти, состав, размер и распределение по шинам областей оперативной памяти. Тем же способом и на этом же этапе к компьютеру добавляются готовые внешние устройства, например контроллер com-порта или устройства светодиодной индикации.

При необходимости выполнить эту работу в традиционной САПР более низкого уровня пользователю пришлось бы писать довольно большой текст на VHDL, связывающий между собой уточнения компонентов всех использованных функциональных устройств: процессора, шин, контроллеров. Учитывая, что у процессора, например, число

внешних сигналов превышает сотню, вероятность ошибки при такой работе весьма велика. В данном же случае САПР порождает некоторое количество конфигурационных файлов и скриптов, гарантирующих впоследствии порождение правильного общего текста системы на VHDL.

Следующим шагом после проектирования компьютера является проектирование заготовки устройства, в рамках которого будет выполняться разработка собственной схемы. Этот шаг выполняется способом, очень похожим на способ проектирования компьютера. В интерактивном режиме без написания текста на VHDL или рисования схемы «в регистрах» задаются внешние интерфейсы устройства: логика подключения к выбранной шине из числа имеющихся в компьютере, набор адресуемых регистров и областей памяти. Например: «Устройство должно подключаться к шине первого уровня, содержать 8 64-разрядных регистров и 2 диапазона адресов по 1МБ каждый». Затем разработанное устройство тем же способом включается в состав компьютера.

Завершающий шаг подготовительного этапа разработки, не подразумевающего явного схемотехнического проектирования — разработка стартовой исполняемой программы для бортового процессора. Если в составе оперативной памяти созданного компьютера имеются области, реализованные на блоках памяти из состава FPGA, то исполняемый файл такой программы встраивается в файл конфигурационной памяти для загрузки в кристалл, и программа стартует автоматически, сразу после начальной раскрутки загружаемой схемы. САПР может автоматически построить тривиальную стартовую программу, например тест памяти, выдающий результаты работы путем печати в com-порт. Конечно, текст стартовой программы впоследствии можно будет редактировать по мере необходимости.

Только после всей этой масштабной подготовительной работы, которая в более низкоуровневой САПР сама по себе заняла бы не один день, разработчику предлагается непосредственно приступить к логическому проектированию. Предполагается, что проектирование будет вестись путем написания текста на VHDL или Verilog. Текст не разрабатывается с нуля, а получается путем редактирования автоматически порожденных на этапе создания заготовки устройства текстов.

При создании заготовки устройства САПР порождает в виде исходного текста на выбранном языке компонент устройства с указанным шинным интерфейсом и «почти пустой» внутренней «начинкой». Точнее, порожденный текст представляет собой схему устройства, в котором все адресуемые объекты представляют собой запоминающие элементы. Например, если пользователь собирается разрабатывать видеоадаптер, то он, очевидно, включил в состав устройства диапазон памя-

ти под буфер развертки, и набор управляющих регистров. Ни логики развертки, ни логики управления при записи тех или иных значений в регистры в заготовке устройства, конечно же, нет и не может быть, это все лишь предстоит создать. Но сами регистры, адресуемые на шине, уже есть и представляют собой просто запоминающие регистры. Есть и диапазон адресов памяти, в котором действительно реализован некоторый объем обычной памяти.

Реализация этих тривиальных запоминающих устройств, их взаимодействия с периферийной шиной процессора, содержится в порожденном САПР файле на VHDL (или Verilog). Таким образом, общая схема системы на кристалле, включающая в себя заготовку устройства, уже готова к трансляции и загрузке в кристалл. Разработка содержательной части схемы состоит в изменении и дописывании этого исходного файла заготовки устройства. Порожденный файл очень подробно прокомментирован и имеет нарочито простую структуру, т.е. специально предназначен для выполнения роли именно заготовки, подлежащей изменению, возможно, «по аналогии» с уже имеющимися типовыми конструкциями.

Таким образом, разработчикам САПР XPS удалось действительно очень многое сделать для того, чтобы сократить традиционно трудоемкую, но совершенно необходимую работу по подготовке к разработке схемы как таковой [48].

## СПИСОК ЛИТЕРАТУРЫ

1. *Burks A. W.* Preliminary discussion of the logical design of an electronic computing instrument / A. W. Burks, H. H. Goldstine, J. von Neumann. // The report of the Princeton Institute of Advanced Researches, 1946.
2. *Лацис А. О.* Как построить и использовать суперкомпьютер. — М. : Бестселлер, 2003. — 240 с.
3. *Воеводин Вл. В.* Параллельные вычисления. — СПб. : БХВ-Петербург, 2002. — 608 с.
4. <http://parallel.ru>. Сетевой ресурс.
5. *Лавров С. С.* Введение в программирование. — М. : Наука, 1973. — 352 с.
6. *Фельдман С.* Системное программирование на персональном компьютере. — М.: ИДДК, 2005.
7. <http://www.intel.com>. Сетевой ресурс.
8. <http://www.amd.com>. Сетевой ресурс.
9. <http://www.top500.org>. Сетевой ресурс.
10. <http://www.nas.nasa.gov/Resources/Software/npb.html>. Сетевой ресурс.
11. <http://www.ibm.com>. Сетевой ресурс.
12. <http://www.myri.com>. Сетевой ресурс.
13. <http://www.dolphinics.com>. Сетевой ресурс.
14. <http://www.mellanox.com>. Сетевой ресурс.
15. <http://www.kiam.ru>. Сетевой ресурс.
16. *Лацис А. О.* Из чего состоит барьер вхождения в мир параллельных технологий // В сб. «Системный анализ и информационные технологии. Междунар. ун-т природы, общества и человека «Дубна». — Дубна, 2004.
17. *Лацис А. О.* МВС-900: вариант МВС-1000 на базе локальной сети Windows-машин // Препринт ИПМ им. М. В. Келдыша РАН, 2003. — № 13.
18. <http://www.vmware.com>. Сетевой ресурс.
19. <http://www.emsl.pnl.gov/docs/global>. Сетевой ресурс.
20. <http://www.cray.com>. Сетевой ресурс.
21. <http://upc.lbl.gov>. Сетевой ресурс.

22. <http://www.co-array.org>. Сетевой ресурс.
23. <http://www.netlib.org/scalapack>. Сетевой ресурс.
24. <http://www.mcs.anl.gov/petsc>. Сетевой ресурс.
25. <http://www.llnl.gov/computing/tutorials/pthreads/>. Сетевой ресурс.
26. <http://hpff.rice.edu>. Сетевой ресурс.
27. <http://www.keldysh.ru/dvm>. Сетевой ресурс.
28. <http://www.keldysh.ru/pages/norma>. Сетевой ресурс.
29. <http://www.hlr.de/organization/amt/projects/pacx-mpi>. Сетевой ресурс.
30. <http://www.gnu.org>. Сетевой ресурс.
31. <http://www.pvfs.org>. Сетевой ресурс.
32. Hoare C. A. R. Communicating Sequential Processes // Prentice Hall, 1985.
33. <http://www.csm.ornl.gov/pvm/>. Сетевой ресурс.
34. <http://www-unix.mcs.anl.gov/mpi/mpich1/>. Сетевой ресурс.
35. <http://www.lam-mpi.org>. Сетевой ресурс.
36. [www.open-mpi.org](http://www.open-mpi.org). Сетевой ресурс.
37. <http://www.scai.fraunhofer.de/EP-CACHE/adaptor/>. Сетевой ресурс.
38. The Quadrics Network (QsNet): High-Performance Clustering Technology // [F. Petrini, Wu-chun Feng, A. Hoisie at al] Proceedings of the 9<sup>th</sup> IEEE Hot Interconnects, Palo Alto. — California, 2001.
39. <http://www.openmp.org>. Сетевой ресурс.
40. <http://www.globus.org>. Сетевой ресурс.
41. PCI Express System Architecture / R. Budruk, D. Anderson, T. Shanley. — Addison Wesley, 2003.
42. Trodden J. Hypertransport System Architecture / J. Trodden, D. Anderson. — Addison Wesley, 2003.
43. <http://developer.nvidia.com/cuda>. Сетевой ресурс.
44. <http://www.onestopsystems.com/>. Сетевой ресурс.
45. <http://www.plxtech.com>. Сетевой ресурс.
46. <http://www.sun.com>. Сетевой ресурс.
47. Гивоне Д. Микропроцессоры и микрокомпьютеры. Вводный курс / Д. Гивоне, Р. Россер. — М.: Мир, 1983.
48. <http://www.xilinx.com>. Сетевой ресурс.
49. Parnell K. Programmable Logic Design Quick Start Handbook / K. Parnell, N. Mehta. — Xilinx, Inc, 2004.
50. <http://www.mvs.tsure.ru>. Сетевой ресурс.
51. <http://www.sgi.com>. Сетевой ресурс.
52. <http://www.altera.com>. Сетевой ресурс.
53. <http://www.latticesemi.com>. Сетевой ресурс.
54. <http://www.opencores.org>. Сетевой ресурс.
55. Turkington K. FPGA Acceleration of the LINPACK Benchmark Using Handel-C and the Celoxica Floating Point Library. International

Conference on Field Programmable Logic and Applications / K. Turkington, K. Masselos, G. Constantinides, P. Leong. — Madrid, 2006.

56. <http://www.xtremedatainc.com>. Сетевой ресурс.

57. Бибило П. Н. Основы языка VHDL. — М.: Изд-во «Солон-Р», 2002. — 224 с.

58. Поляков А. К. Языки VHDL и VERILOG в проектировании цифровой аппаратуры. — М.: Изд-во «СОЛОН-Пресс», 2003.

59. <http://www.celoxica.com>. Сетевой ресурс.

60. <http://www.mitronics.com>. Сетевой ресурс.

61. Хендерсон П. Функциональное программирование. Применение и реализация. — М.: «Мир», 1983.

62. Немнюгин С. А. Параллельное программирование для многопроцессорных вычислительных систем / С. А. Немнюгин, О. Л. Стесик. — СПб.: БХВ-Петербург, 2002.

63. <http://www.pcisig.com/specifications/pciexpress>. Сетевой ресурс.

64. [www.samtec.com](http://www.samtec.com). Сетевой ресурс.

65. Introduction To the Cell Multiprocessor / [J. A. Kahle, M. N. Day, H. P. Hofstee at all]. // IBM Journal of research and development. — Vol. 49. — № 4/5, 2005.

66. Корнеев В. В. Вычислительные системы. — М.: Гелиос АРВ, 2004. — 512 с.

67. Корнеев В. В. Современные микропроцессоры / В. В. Корнеев, А. В. Киселев. — СПб.: БХВ-Петербург, 2003. — 440 с.

68. Андреев А. Н. Кластеры и суперкомпьютеры — близнецы или братья? Открытые системы / А. Н. Андреев, Вл. В. Воеводин, С. А. Жуматий. — М., 2000. — № 5—6 — С. 9—14.

69. Каляев А. В. Модульно-наращиваемые многопроцессорные системы со структурно-процедурной организацией вычислений / А. В. Каляев, И. И. Левин. — М.: Янус-К, 2003. — 380 с.



# ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- Адрес виртуальный** 49  
— физический 49  
**Амдала закон** 65  
**Архитектура** 35  
— мультитредовая 258
- Базис полный** 265  
**Барьерная синхронизация** 119, 143  
**Блочно-циклическое распределение** 156
- Вариантный счет** 33, 220  
**Вентиль** 264  
**Ветвь параллельной программы** 29  
**Виртуальность** 35  
**Вычислительная мощность низкого качества** 217  
**Высокопроизводительные вычисления** 33
- Гаусса метод** 62  
**Генератор синхроимпульсов** 266  
**Гибридный параллелизм** 138  
**Граничные условия** 24
- Двусторонний обмен данными** 18, 92  
**Демон** 58  
**Дешифратор** 307  
**Директива параллельного выполнения** 131  
**Драйвер** 58
- Защита памяти** 48  
**Зерно параллелизма** 74
- Императивность правил поведения** 202
- Кластер выделенных рабочих станций** 15  
— невыделенных рабочих станций 16
- Когерентность кэш-памяти** 54  
**Коллективная операция** 109  
**Комбинационная схема** 265  
**Коммуникатор** 97, 120  
**Коммуникационная библиотека** 31  
**Конвейерная обработка данных** 91  
**Коэффициент распараллеливания** 64  
**Кэш-память** 53
- Латентность** 69  
**Логическое проектирование** 264
- Масштабирование приложений** 66  
**MBC-1000** 199  
**MBC-900** 205  
**Метакомпьютинг** 210  
**Метакластер** 205  
**Микросхема программируемой логики** 269  
**Многорежимность процессора** 50  
**Модель параллельного программирования** 86

- Мультикомпьютер 17
- Мультиплексор 308
- Мультипроцессор 17
- Начальный загрузчик 45**
- Непрозрачный мост 243
- Непроцедурный язык 139
- Норма-система 139
- Односторонний обмен данными 141**
- Основной постулат параллельного программирования 87
- Память ассоциативная 55**
  - виртуальная 52
  - разделяемая 17
  - распределенная 17
  - расслоенная 255
- Парадокс неприятия новых технологий 186**
- Параллельная область 180**
- Параллельная программа 29**
- Полнота бисекционная 71**
- Порт внешнего устройства 43**
- Правило экономии сил 12**
- Прерывание внешнее 46**
  - внутреннее 46
  - программное 51
- Привилегированная команда 50**
- Программа-диспетчер 45**
- Прозрачность 35**
- Производный тип 122**
- Пропускная способность 69**
- Производительность канала 69**
  - пиковая 60
- Процесс легковесный 178**
  - полновесный 178
- Разветвление — слияние 180**
- Раздельное администрирование 207**
- Рандеву 32**
- Распараллеливание инкрементальное 136, 163**
  - пошаговое 134, 136
- Распределенный массив 127**
- Регистр данных 42**
  - состояния 42
- САПР XPS 323**
- Семафор взаимного исключения 158**
- Сервис 58**
- Система запуска параллельных программ 31, 200**
  - команд 36
  - очередей и контроля доступа 31, 200
  - параллельного программирования 30
  - сетевая файловая 84
- Системный архитектор 3, 13**
  - запрос 51
- Суперкомпьютер 8, 14**
- Суперкомпьютерная революция, первая 14**
  - — вторая 15
  - — третья 16, 227
- Технология параллельного программирования 86**
- Тред 178**
- Триггер 265**
- Тристабильный формирователь 311**
- Тэг 97**
- Флинна классификация 22**
- Флопс 60**
- Фоннеймановская машина 7, 12**
- Цена обмена 69**
- Шина 229, 311**
- Ядро ОС 58**
- Якоби метод 23**

Cell-процессор 257  
CPLD 276  
CUDA GPU-технология 255  
Co-array Fortran-язык 175  
  
DMA 55  
DVM-система 138  
  
Extrinsic-функция 137  
  
FPGA 276  
  
Infiniband 81  
  
Gigabit Ethernet 80  
Giganet 81  
Global Arrays-библиотека 149  
Grid-технологии 32, 216  
  
Handel-C 290  
HPC 33  
HPF 130  
— HPF Adaptor-система 141  
HTC 33  
Hypertransport 232  
  
Itanium-2 76  
  
LAM 125  
Linpack 62  
  
Message rate 71  
Mitrion-C 293  
MPI 29, 31, 94  
MPI-2 123, 150  
MPICH 125  
MPP-система 17  
MIMD 22  
MISD 22  
Myrinet 80

NASA тесты 63  
NUMA-система 18  
  
Open MP-технология 179  
— Cluster Open MP 191  
Open PBS 201  
Opteron 61, 78  
  
PACX-MPI 216  
PCI Express 81, 232, 236  
Pentium 76  
PETSc 130  
PGAS 152  
PowerPC 76  
PIO 55  
PVFS 85  
PVM 94  
  
Quadrics 81, 141  
  
RDMA 55  
  
SAN/NAS 84  
ScaLAPACK 126  
Shmem-библиотека 141  
SIMD 22  
SISD 22  
SMP-система 17  
SSI-система 177  
  
Top500 68  
  
UPC-язык 153  
  
Verilog 286  
VHD 283, 286, 312  
VmWare 58, 205  
  
X-com-система 221

# ОГЛАВЛЕНИЕ

Предисловие .....	3
Введение .....	6

## ЧАСТЬ I. ОСВОЕННЫЕ ТЕХНОЛОГИИ

<b>Глава 1. Таксономия суперкомпьютеров и применяемых в связи с ними программистских технологий.....</b>	<b>11</b>
1.1. Архитектура фон Неймана .....	12
1.2. Этапы развития суперкомпьютерных технологий .....	13
1.3. Способы объединения многих процессоров в единую систему .....	17
1.4. Альтернативные архитектуры. Классификация Флинна .....	22
1.5. Простейшая модельная программа.....	23
1.6. Общий состав программного обеспечения вычислительного кластера .....	31
1.7. Программистская модель передачи данных.....	32
1.8. Технологии параллельной обработки данных без использования суперкомпьютеров.....	32
<b>Глава 2. Некоторые основные понятия архитектуры процессоров и ОС.....</b>	<b>34</b>
2.1. Некоторые основные определения .....	35
2.2. Пример системы команд: обработка данных в памяти .....	36
2.3. Основные сложности и пути их преодоления.....	40
2.4. Кэш-память и прямой доступ к памяти (DMA) .....	52
2.5. Еще несколько определений.....	57
<b>Глава 3. Критерии эффективности процессора и коммуникационной сети .....</b>	<b>60</b>
3.1. Из чего складывается вычислительная производительность .....	60
3.2. Критерии эффективности коммуникационной сети .....	68
<b>Глава 4. Обзор процессорных и сетевых решений, применяемых в современных кластерах.....</b>	<b>76</b>
4.1. Выбор процессора.....	76
4.2. Выбор коммуникационной сети .....	80
<b>Глава 5. Модели и технологии параллельного программирования.....</b>	<b>86</b>
5.1. Способы параллельной обработки данных.....	88
5.2. Аспекты программистской модели .....	91
5.3. Модель: явный двусторонний обмен сообщениями .....	92

5.4. Модель: односторонний обмен сообщениями .....	141
5.5. Модель: PGAS .....	152
5.6. Модель: SMP .....	176
5.7. Парадокс неприятия новых технологий .....	186
5.8. Сравнительный анализ моделей и технологий параллельного программирования .....	188
<b>Глава 6. Суперкомпьютер МВС-1000 и метакластер МВС-900 ....</b>	<b>199</b>
6.1. Основные принципы организации МВС-1000.....	199
6.2. Как работает и зачем нужен метакластер МВС-900.....	204
<b>Глава 7. Технологии параллельного программирования и метакомпьютинг .....</b>	<b>210</b>
7.1. Две постановки задачи .....	210
7.2. Метакомпьютинг выделенных мощностей. Пример технологии .....	212
7.3. Метакомпьютинг невыделенных мощностей.	
Пример технологии .....	217

## ЧАСТЬ II. НОВЫЕ ТЕХНОЛОГИИ

<b>Глава 8. «Эпоха кластеров» заканчивается .....</b>	<b>226</b>
8.1. Магистраль обмена данными с процессором: шина или сеть?.....	229
8.2. Кризис фоннеймановской модели вычислительных приложений.....	233
<b>Глава 9. Проблемы объединения магистралей PCI Express .....</b>	<b>236</b>
9.1. PCI Express на логическом уровне.....	238
9.2. Несколько слов о реализации.....	244
9.3. Недостающие компоненты .....	247
9.4. Эскиз базового программного обеспечения .....	249
<b>Глава 10. Альтернативные архитектуры вычислителя .....</b>	<b>251</b>
10.1. Универсальные не фоннеймановские архитектуры.....	253
10.2. Проблемно-ориентированные вычислители.....	260
<b>Глава 11. Как создаются цифровые электронные устройства.....</b>	<b>264</b>
<b>Глава 12. Технологии программируемой логики сегодня .....</b>	<b>271</b>
<b>Глава 13. Проблемы реализации реконфигурируемого сопроцессора .....</b>	<b>280</b>
13.1. Проблемы физической реализации и смежные вопросы.....	280
13.2. Проблемы логического проектирования .....	282
<b>Глава 14. Языки программирования, транслируемые в схему.....</b>	<b>290</b>
14.1. Handel-C .....	290
14.2. Mitrion-C .....	293
Заключение.....	296
Приложения .....	301
Список литературы .....	328
Предметный указатель .....	331

*Учебное издание*  
**Лацис Алексей Оттович**  
**Параллельная обработка данных**

*Учебное пособие*  
Редактор *А. Н. Мирошников*  
Технический редактор *Е. Ф. Коржуева*  
Компьютерная верстка: *Н. В. Протасова*  
Корректор *В. А. Жилкина*

Изд. № 101113156. Подписано в печать 30.11.2009. Формат 60×90/16.  
Гарнитура «Таймс». Печать офсетная. Бумага офс. № 1. Усл. печ. л. 21,0.  
Тираж 1500 экз. Заказ № 29493.

Издательский центр «Академия». [www.academia-moscow.ru](http://www.academia-moscow.ru)  
Санитарно-эпидемиологическое заключение № 77.99.02.953.Д.004796.07.04 от 20.07.2004.  
117342, Москва, ул. Бултерова, 17-Б, к. 360. Тел./факс: (495)334-8337, 330-1092.

Отпечатано в соответствии с качеством предоставленных издательством  
электронных носителей в ОАО «Саратовский полиграфкомбинат».  
410004, г. Саратов, ул. Чернышевского, 59. [www.sarpk.ru](http://www.sarpk.ru)