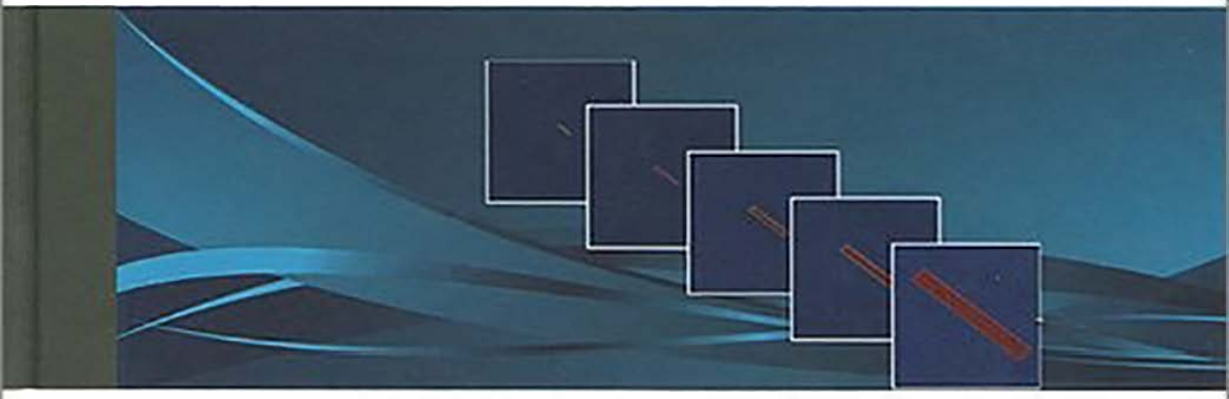
Серия
Суперкомпьютерное
Образование

Дж. Кепнер

Параллельное программирование в среде MATLAB для многоядерных и многоуровневых вычислительных машин



Суперкомпьютерный
консорциум
университетов России

МГУ

УДК 007 (075)
ББК 32.973.2
К 33

Научный редактор *Д. В. Дубров*

Кепнер, Дж.

К 33 Параллельное программирование в среде MATLAB для многоядерных и многоузловых вычислительных машин: Учебное пособие / Под ред. Д. В. Дуброва; Предисл.: В. А. Садовничий. – М.: Издательство Московского университета, 2013. – 296 с., илл. – (Серия «Суперкомпьютерное образование»)

ISBN 978-5-211-06428-7

В настоящее время язык MATLAB является основным языком технических вычислений. Насчитывается около одного миллиона его пользователей по всему миру, многие из которых видят преимущество в повышенной мощности, которую предлагают широкодоступные многоядерные процессоры и многоузловые вычислительные кластеры. Среда MATLAB также идеально подходит для изучения параллельных вычислений, позволяя пользователю обращать основное внимание на параллельные алгоритмы, нежели на детали реализации. Лаконичность языка MATLAB дает возможность представить множество специфических примеров как иллюстрацию основных концепций параллельного программирования.

Данная книга посвящена практической реализации методов вычислений и аспектам высокой производительности научных вычислений с упором на востребованное программное обеспечение, вычислительные среды и инструменты вычислений. В ней представлены вопросы разработки технологий программного обеспечения, такие как текущий статус, приложения и алгоритмы, программное обеспечение для решения математических задач, программные инструменты, языки и компиляторы, вычислительные среды и визуализация.

К целевой аудитории книги принадлежит любой человек, перед которым стоит задача адаптировать последовательную программу MATLAB к параллельной среде. Кроме того, ее можно использовать в качестве основного учебника на занятиях по параллельным вычислениям, либо в качестве пособия для дополнительного чтения в классах изучения численных методов или алгоритмов информатики.

УДК 007 (075)
ББК 32.973.2

Parallel MATLAB for Multicore and Multinode Computers copyright © 2009 Society for Industrial and Applied Mathematics. Published by Moscow University Press with permission. Russian edition copyright © 2012 by Moscow University Press.

© Общество промышленной и прикладной математики, 2003
© Издательство Московского университета, 2012

ISBN 978-5-211-06428-7

Уважаемый читатель!

Вы держите в руках одну из книг серии «Суперкомпьютерное образование», выпущенную в рамках реализации проекта комиссии Президента РФ по модернизации и технологическому развитию экономики России «Создание системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий и специализированного программного обеспечения». Инициатором издания выступил Суперкомпьютерный консорциум университетов России.

Серия включает более 20 учебников и учебных пособий, подготовленных ведущими отечественными специалистами в области суперкомпьютерных технологий. В книгах представлен ценный опыт преподавания суперкомпьютерных технологий в таких авторитетных вузах России, как МГУ, ННГУ, ТГУ, ЮУрГУ, СПбГУ ИТМО и многих других. При подготовке изданий были учтены рекомендации, сформулированные в Своде знаний и умений в области суперкомпьютерных технологий, подготовленном группой экспертов Суперкомпьютерного консорциума, а также международный опыт.

Современный уровень развития вычислительной техники и методов математического моделирования дает уникальную возможность для перевода промышленного производства и научных исследований на качественно новый этап. Эффективность такого перехода напрямую зависит от наличия достаточного числа высококвалифицированных специалистов. Данная серия книг предназначена для широкого круга студентов, аспирантов и специалистов, желающих изучить и практически использовать параллельные компьютерные системы для решения трудоемких вычислительных задач.

Издание серии «Суперкомпьютерное образование» наглядно демонстрирует тот вклад, который внесли участники Суперкомпьютерного консорциума университетов России в создание национальной системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий, а также их четкое понимание ответственности за подготовку высококвалифицированных специалистов и формирование прочного научного фундамента, столь необходимого для эффективного использования суперкомпьютерных технологий на практике.

Ректор Московского университета,
Президент Суперкомпьютерного консорциума
университетов России,
академик РАН ***В.А. Садовничий***

Часть 1. Фундаментальные принципы ... 22
Глава 1. Основные понятия: система записи и интерфейсы ... 22
1.1. Система записи алгоритмов ... 22
1.1.1. Запись распределенного массива данных ... 23
1.1.2. Доступ к распределенным данным ... 24
1.2. Интерфейсы параллельных функций ... 25
1.2.1. Программирование на основе карты размещения ... 27
1.2.2. Параллельное исполнение ... 28
Глава 2. Введение в pMATLAB ... 30
2.1. Программа: Mandelbrot (мелкозернистый тривиальный массовый параллелизм) ... 30
2.1.1. Начало работы ... 31
2.1.2. Разработка параллельной программы ... 32
2.1.3. Код ... 34
2.1.4. Отладка ... 38
2.1.5. Тестирование ... 39
2.2. Программа ZoomImage (крупнозернистый тривиальный массовый параллелизм) ... 41
2.2.1. Начало работы ... 41
2.2.2. Разработка параллельной программы ... 42
2.2.3. Код ... 46
2.2.4. Отладка ... 48
2.2.5. Тестирование ... 48
2.3. Программа ParallelIO ... 50
2.3.1. Начало работы ... 50
2.3.2. Разработка параллельной программы ... 51
2.3.3. Код ... 52
2.3.4. Отладка ... 56
2.3.5. Тестирование ... 57
2.4. Почему это работает ... 58
Глава 3. Взаимодействие с распределенными массивами ... 59
3.1. Начало работы ... 60
3.2. Разработка параллельной программы ... 61
3.3. Код ... 66
3.4. Интерактивная отладка и тестирование ... 69
3.4.1. Корректность последовательного кода ... 70
3.4.2. Корректность параллельного кода ... 70
3.4.3. Корректность локальной коммуникации ... 71

3.4.4. Корректность удаленной коммуникации ... 72
3.4.5. Оценка производительности ... 73
3.5. Тема повышенной сложности: введение в параллельно-конвейерные приложения ... 74
Часть 2. Методики повышенной сложности ... 77
Глава 4. Модели параллельного программирования ... 77
4.1. Проектирование: оценка необходимости распараллеливания ... 78
4.1.1. Профилирование производительности и памяти ... 78
4.1.2. Шаблоны параллельного программирования ... 82
4.1.3. Параллельная реализация программы Blurimage ... 86
4.2. Кодирование ... 90
4.2.1. Начальник/подчиненный ... 90
4.2.2. Обмен сообщениями ... 92
4.2.3. Распределенные массивы ... 94
4.3. Отладка ... 95
4.4. Тестирование ... 96
4.5. Общий анализ производительности ... 98
4.6. Выводы ... 98
Глава 5. Разработка сложных программ с распределенными массивами ... 101
5.1. Введение ... 101
5.2. Сравнительная характеристика чистых и фрагментированных распределенных массивов ... 102
5.3. Интерфейс и реализация архитектуры распределенных массивов MATLAB ... 103
5.4. Карты и распределения ... 104
5.5. Функции поддержки параллелизма ... 108
5.5.1. Циклические распределения и балансирование параллельной нагрузки ... 108
5.6. Выбор в пользу параллелизма или локальности ... 113
5.7. Простота реализации ... 115
5.8. Особенности реализации ... 116
5.8.1. Выполнение программы ... 119
5.9. Перераспределение массивов ... 120
5.10. Слой обмена сообщениями ... 122
5.11. Принципы увеличения производительности ... 126
5.11.1. Производительность, производительность, производительность ... 127
5.11.2. Оценка производительности ... 128
5.11.3. Минимизация накладных расходов ... 128

5.11.4. Массовая параллельность подразумевает линейное ускорение вычислений ... 128
5.11.5. Алгоритм и распределение ортогональны друг к другу ... 128
5.11.6. Не навреди! ... 128
5.11.7. Следите за SLOC ... 129
5.11.8. Бесплатный сыр бывает только в мышеловке ... 129
5.11.9. Четыре простых шага ... 129
5.12. Результаты, предоставленные пользователями ... 131
Глава 6. Метрики производительности и архитектура программного обеспечения ... 135
6.1. Введение ... 135
6.2. Характеристики параллельного приложения ... 136
6.2.1. Характеристики примеров программ ... 136
6.2.2. Метрики производительности в последовательном режиме ... 139
6.2.3. Степени параллелизма ... 141
6.2.4. Метрики параллельной производительности (коммуникация отсутствует) ... 143
6.2.5. Метрики параллельной производительности (коммуникация присутствует) ... 144
6.2.6. Закон Амдала ... 146
6.2.7. Характеристика ускорения ... 147
6.2.8. Пространственная и временная локальность ... 150
6.3. Стандартная параллельная вычислительная машина ... 152
6.3.1. Сетевая модель ... 154
6.3.2. Иерархия Кека ... 158
6.4. Модели параллельного программирования ... 160
6.5. Системные метрики ... 161
6.5.1. Производительность ... 161
6.5.2. Форм-фактор ... 162
6.5.3. Эффективность ... 164
6.5.4. Затраты на программное обеспечение ... 166
6.5.5. Продуктивность программного обеспечения ... 168
Часть 3. Анализ практических примеров ... 173
Глава 7. Анализ параллельных приложений ... 173
7.1. Исторический обзор ... 173
7.2. Характеристика пространства приложений ... 175
7.2.1. Иерархия физической памяти ... 176
7.2.2. Пространственная/временная локальность ... 177

7.2.3. Иерархия логической памяти 178
7.3. Набор HPC Challenge: охват пространства приложений 179
7.3.1. Stream 180
7.3.2. FFT 180
7.3.3. RandomAccess 181
7.3.4. HPL 182
7.4. Собственная производительность алгоритма 182
7.4.1. Структура данных 183
7.4.2. Сложность вычислений 183
7.4.3. Степени параллелизма 184
7.4.4. Сложность коммуникации 185
7.5. Производительность оборудования 185
7.5.1. Пространственная и временная локальности 186
7.5.2. Эффективность производительности 186
7.5.3. Оценка производительности 187
7.5.4. Результаты анализа 189
7.5.5. Значение производительности 190
7.6. Производительность программ 191
7.6.1. Stream 191
7.6.2 RandomAccess 192
7.6.3. FFT 193
7.6.4. HPL 194
7.7. Производительность по отношению к трудозатратам 194
Глава 8. Программа Stream 201
8.1. Начало работы 201
8.2. Параллельная реализация 203
8.3. Код 205
8.4. Отладка 208
8.5. Тестирование 209
Глава 9. Программа RandomAccess 213
9.1. Начало работы 214
9.2. Параллельная реализация 215
9.2.1. Рассеивающий алгоритм 217
9.2.2. Древовидный алгоритм 219
9.3. Код 222
9.3.1. Рассеивающий код 225
9.3.2. Древовидный код 226
9.3.3. Обзор кода 228

9.4. Отладка 229
9.5. Тестирование 230
9.5.1. Производительность на многоядерной системе 231
9.5.2. Производительность на многоузловой системе 233
Глава 10. Программа FFT 236
10.1. Начало работы 237
10.2. Параллельная реализация 238
10.3. Код 241
10.4. Отладка 244
10.5. Тестирование 245
10.5.1 Производительность на многоядерной системе 246
10.5.2. Производительность на многоузловой системе 247
Глава 11. Программа High Performance LiNpack 250
11.1. Начало работы 251
11.2. Параллельная реализация 252
11.2.1. Параллельное LU-разложение 255
11.2.2. Анализ критический путей 257
11.3. Код 260
11.4 Отладка 264
11.5. Тестирование 265
11.5.1. Производительность на многоядерной системе 265
11.5.2. Производительность на многоузловой системе 267
Приложение: Система обозначений для иерархических параллельных алгоритмов для многоядерных систем 269
A.1. Введение 269
A.2. Параллелизм по данным 270
A.2.1. Последовательный алгоритм 270
A.2.2. Параллельный алгоритм 271
A.2.3. Блочный параллельный алгоритм 275
A.2.4 Иерархический параллельный алгоритм 276
A.2.5. Иерархический блочный параллельный алгоритм 277
A.3. Конвейерный параллелизм 279
A.3.1. Неявный конвейерный параллелизм 280
A.3.2. Конвейерный параллелизм по задачам 281
A.3.3. Мелкозернистый конвейерный параллелизм по задачам 284
A.3.4. Обобщенный иерархический блочный параллельный алгоритм 286
Указатель 290

Перечень рисунков

Рисунок 1. Зависимость производительности от трудозатрат 17
Рисунок 2. Развитие языка параллельного программирования MATLAB 21
Рисунок 1.1. Параллельные карты размещения 25
Рисунок 1.2. Структура карты размещения 27
Рисунок 1.3. Параллельное исполнение 29
Рисунок 2.1. Результат программы Mandelbrot 32
Рисунок 2.2. Реализация параллельной программы Mandelbrot 35
Рисунок 2.3. Результат программы ZoomImage 42
Рисунок 2.4. Параллельная реализация программы ZoomImage 45
Рисунок 2.5. Реализация ParallelIO 53
Рисунок 3.1. Результат программы Beamformer 60
Рисунок 3.2. Физика программы Beamformer 61
Рисунок 3.3. Параллельная реализация программы Beamformer 66
Рисунок 3.4. Параллельно-конвейерная реализация программы Beamformer 75
Рисунок 4.1. Результат программы Blurimage 79
Рисунок 4.2. Результаты профилировщика для программы Blurimage 81
Рисунок 4.3. Стандартные шаблоны параллельного взаимодействия 84
Рисунок 4.4. Распределение массивов по картам распределения 86
Рисунок 4.5. Виды параллельных реализаций программы Blurimage 87
Рисунок 4.6. Четырехступенчатый процесс отладки 97
Рисунок 5.1. Сравнительная характеристика чистых и фрагментированных распределенных массивов 103
Рисунок 5.2. Блочно-циклические распределения 107
Рисунок 5.3. Параллелизм и локальность 114
Рисунок 5.4. Многослойная архитектура параллельной библиотеки 118
Рисунок 5.5. Инфраструктура выполнения 120
Рисунок 5.6. Коммуникация на основе файлового ввода/вывода в библиотеке MatlabMPI 123
Рисунок 5.7. Результаты тестирования скорости сети 125
Рисунок 5.8. Сравнительная характеристика MatlabMPI и MPI 126
Рисунок 5.9. Коммуникационные накладные расходы массивов 127
Рисунок 6.1. Типы параллелизма 142
Рисунок 6.2. Закон Амдала 146
Рисунок 6.3. Кривые ускорения. Графики различных типов ускорения 148
Рисунок 6.4. Диапазоны ускорения 149
Рисунок 6.5. Сравнение слабого и сильного масштабирования 150
Рисунок 6.6. Модель пространственной/временной локальности 151
Рисунок 6.7. Пространственные/временные показатели примеров программ 152
Рисунок 6.8. Параллельная архитектура и иерархия памяти 154
Рисунок 6.9. Топологии сетей 155
Рисунок 6.10. Производительность сети 157
Рисунок 6.11. Переразмещение данных с поворотом угла 158
Рисунок 6.12. Двухуровневая иерархия 159
Рисунок 6.13. Пример иерархии Кека 160
Рисунок 6.14. Пример вычислительной стойки 163

Рисунок 6.15. Ускорение в зависимости от трудозатрат ... 169
Рисунок 6.16. Параллельная продуктивность в зависимости от трудозатрат ... 170
Рисунок 7.1. Эволюция вычислений на супер-ЭВМ ... 174
Рисунок 7.2. Иерархия памяти и набор тестов производительности HPC Challenge ... 177
Рисунок 7.3. Сравнение пространственной и временной локальности ... 178
Рисунок 7.4. Коммуникация в иерархии ... 179
Рисунок 7.5. Шаблоны коммуникации пакета HPC Challenge ... 181
Рисунок 7.6. Оценки производительности HPC Challenge ... 190
Рисунок 7.7. Ускорение относительно размера кода ... 198
Рисунок 7.8. Продуктивность HPC Challenge ... 199
Рисунок 8.1. Параллельная реализация программы Stream ... 204
Рисунок 8.2. Производительность программы Stream относительно размера задачи ... 212
Рисунок 9.1. Шаблон коммуникации рассеивающего алгоритма программы RandomAccess ... 219
Рисунок 9.2. Шаблон коммуникации древовидного алгоритма RandomAccess ... 222
Рисунок 9.3. Относительная производительность параллельной программы RandomAccess ... 234
Рисунок 10.1. Параллельная реализация программы FFT ... 241
Рисунок 10.2. Относительная производительность параллельной программы FFT ... 249
Рисунок 11.1. Параллельная реализация алгоритма LU-разложения ... 258
Рисунок 11.2. Критический путь параллельного LU-алгоритма ... 260
Рисунок 11.3. Относительная производительность параллельной программы HPL ... 268
Рисунок А.1. Схема Кека для одного процессора ... 272
Рисунок А.2. Параллельные карты распределения ... 274
Рисунок А.3. Схема Кека для параллельных процессоров ... 275
Рисунок А.4. Схема Кека для иерархического параллельного процессора ... 278
Рисунок А.5. Две задачи, соединенные каналом ... 282
Рисунок А.6. Дублированная задача, связанная с недублированной задачей ... 284

Перечень таблиц

Таблица 2.1. Параллельное распределение в программе Mandelbrot 34
Таблица 2.2. Шаги отладки 38
Таблица 2.3. Производительность программы Mandelbrot на многоядерных и многоузловых системах 40
Таблица 2.4. Параллельное распределение ZoomImage 44
Таблица 2.5. Производительность программы ZoomImage на многоядерных и многоузловых системах 49
Таблица 2.6. Производительность программы ParallelIO на многоядерных и многоузловых системах 57
Таблица 3.1. Параллельное распределение в программе Beamformer 65
Таблица 3.2. Производительность программы Beamformer на многоядерных и многоузловых системах 74
Таблица 3.3. Параллельно-конвейерное распределение в программе Beamformer 75
Таблица 4.1. Классификация параллельных приложений 83
Таблица 4.2. Производительность программы BlurImage на многоядерных и многоузловых системах 97
Таблица 5.1. Цели реализации распределенных массивов 105
Таблица 5.2. Функции поддержки параллельных распределенных массивов 109
Таблица 5.3. Типы распределения распределенных массивов 110
Таблица 5.4. Неравномерность нагрузки в программе Mandelbrot 112
Таблица 5.5. Уровни реализации 117
Таблица 5.6. Функции MPI 122
Таблица 5.7. Результаты работы некоторых пользовательских приложений 132
Таблица 6.1. Характеристики параллельных алгоритмов 139
Таблица 6.2. Оценка эффективности реализации программного обеспечения 165
Таблица 6.3. Оценка скорости написания кода программного обеспечения 168
Таблица 6.4. Приблизительная относительная продуктивность 171
Таблица 7.1. Характеристики параллельных алгоритмов пакета HPC Challenge 184
Таблица 7.2. Параметры иерархии памяти 187
Таблица 7.3. Результаты реализации пакета HPC Challenge 195
Таблица 8.1. Производительность операции triadd в тесте Stream на многоядерных и многоузловых системах 211
Таблица 9.1. Производительность программы RandomAccess на многоядерной системе 231
Таблица 9.2. Производительность программы RandomAccess на многоузловой системе 233
Таблица 10.1. Производительность программы FFT на многоядерной системе 246
Таблица 10.2. Производительность программы FFT на многоузловой системе 248
Таблица 11.1. Производительность программы HPL на многоядерной системе 266
Таблица 11.2. Производительность программы HPL на многоузловой системе 267

Перечень алгоритмов

Алгоритм 1.1. Последовательное добавление единицы 23
Алгоритм 1.2. Параллельное добавление единицы 25
Алгоритм 2.1. Последовательный алгоритм Мандельброта 32
Алгоритм 2.2. Параллельный алгоритм Mandelbrot 34
Алгоритм 2.3 Последовательный алгоритм ZoomImage 43
Алгоритм 2.4. Параллельный алгоритм ZoomImage 44
Алгоритм 2.5. Последовательный алгоритм IO 51
Алгоритм 2.6. Параллельный алгоритм ParallelIO. 51
Алгоритм 3.1. Последовательный алгоритм Beamformer 63
Алгоритм 3.2. Параллельный алгоритм Beamformer 64
Алгоритм 3.3. Параллельно-конвейерный алгоритм Beamformer 75
Алгоритм 4.1. Последовательный алгоритм Blurimage 80
Алгоритм 4.2. Реализация Blurimage на основе модели «начальник/подчиненный» 87
Алгоритм 4.3. Реализация Blurimage на основе обмена сообщениями. 89
Алгоритм 4.4. Реализация Blurimage на основе распределенных массивов 89
Алгоритм 5.1. Тестирование скорости сети 124
Алгоритм 8.1. Последовательная версия программы Stream. 203
Алгоритм 8.2. Параллельная версия программы Stream 205
Алгоритм 9.1. Последовательная реализация RandomAccess 216
Алгоритм 9.2. Рассеивающий параллельный алгоритм для программы RandomAccess 217
Алгоритм 9.3. Древовидный параллельный алгоритм для программы RandomAccess 220
Алгоритм 10.1. Параллельный алгоритм FFT 239
Алгоритм 11.1. Последовательный алгоритм LU-разложения 253
Алгоритм 11.2. Параллельный алгоритм LU-разложения 255
Алгоритм А.1. Последовательный алгоритм 2D-фильтра 271
Алгоритм А.2. Параллельный алгоритм 2D-фильтра 273
Алгоритм А.3. Блочный параллельный алгоритм 2D-фильтра 276
Алгоритм А.4. Иерархический параллельный алгоритм 2D-фильтра 277
Алгоритм А.5. Иерархический блочный параллельный алгоритм 2D-фильтра 279
Алгоритм А.6. Неявный конвейерный алгоритм 2D-фильтра 280
Алгоритм А.7. Параллельный по задачам алгоритм 2D-фильтра 283
Алгоритм А.8. Мелкозернистый параллельный по задачам алгоритм 2D-фильтра 285
Алгоритм А.9. Обобщенный иерархический блочный параллельный алгоритм 2D-фильтра 287
Алгоритм А.10. Иерархические вспомогательные функции 288

Предисловие

В настоящее время язык MATLAB является основным языком технических вычислений. Насчитывается около одного миллиона его пользователей по всему миру, многие из них видят преимущество в повышенной мощности, которую предлагают широко доступные многоядерные процессоры и многоузловые вычислительные кластеры. Среда MATLAB также идеально подходит для изучения параллельных вычислений, позволяя пользователю обратить основное внимание на параллельные алгоритмы, нежели на детали реализации. Лаконичность языка MATLAB дает возможность представить множество специфических примеров как иллюстрацию основных концепций параллельного программирования.

Настоящая книга посвящена параллельному программированию на языке MATLAB и, надеемся, является первой из множества книг на эту тему, так как в настоящее время существует богатый выбор параллельных MATLAB-библиотек для пользователей [Choy 2004]. Мне посчастливилось принимать участие в разработке двух таких библиотек [Kepner 2004, Kepner 2003]. Наиболее широко используются следующие из них: pMATLAB (разработана Линкольнской лабораторией Массачусетского технологического института), Parallel Computing Toolbox (разработана The MathWorks, Inc.) и StarP (разработана Массачусетским технологическим институтом, MCSB и Interactive Supercomputing, Inc.). Все эти библиотеки обеспечивают непосредственную поддержку параллельных вычислений с использованием распределенных массивов и других моделей параллельного программирования. Особые примеры, рассматриваемые в данной книге, написаны с использованием библиотеки pMATLAB, имеющейся в свободном доступе. Преимуществом использования pMATLAB является тот факт, что она может выполняться независимо или поверх вышеупомянутых параллельных библиотек MATLAB. Таким образом, все программы, приведенные в книге в качестве примеров, могут быть выполнены в любой параллельной среде MATLAB. К счастью, концепции, изложенные в нашей книге, независимы от базовой реализации и действительны для любого синтаксиса параллельного программирования, предпочтительного для пользователя. Программное обеспечение, а также инструкции по установке вы можете найти на веб-сайте книги:

http://www.siam:org/KernerBook

Выразительные возможности MATLAB позволяют нам сконцентрироваться на методах создания эффективных параллельных программ (в отличие от синтаксической механики написания параллельных программ). Основное внимание мы уделим технике конструирования, кодирования, отладки и тестирования, необходимой для быстрого производства эффективных параллельных программ за считанные часы, а не недели и месяцы (см. рисунок 1). Эти методы разрабатывались в течение многих лет посредством индивидуального взаимодействия с сотнями пользователей параллельного пакета MATLAB.

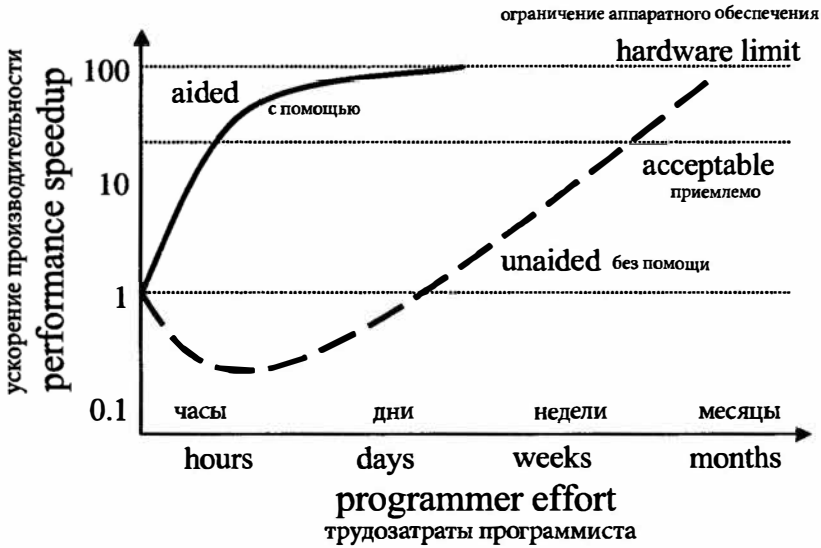


Рисунок 1. Зависимость производительности от трудозатрат
Отображение производительности в зависимости от трудозатрат пользователя, вложенных в стандартную параллельную программу MATLAB. Нижняя кривая показывает трудозатраты самостоятельного пользователя. Верхняя кривая показывает трудозатраты пользователя, обеспеченного квалифицированной помощью

Предполагается, что наш читатель уже знаком с пакетом MATLAB (введение в среду MATLAB представлено в [Higham & Higham 2005] и [Moler 2004]). К целевой аудитории книги принадлежит любой человек, перед которым стоит задача адаптировать последовательную программу MATLAB к параллельной среде. Кроме того, эту книгу можно использовать в качестве основного учебника на занятиях по параллельным вычислениям, либо в качестве дополнительного чтения в классах изучения численных методов вычислений или алгоритмов информатики. Все идеи представлены здесь с практической точки зрения с рассмотрением множества примеров. По возможности, примеры были взяты из широко известных и подтвержденных документально параллельных тестовых кодов, которые уже признаны характерными для многих приложений (хотя связь с тем или иным конкретным приложением может потребовать изучения ссылочной информации). Исторически сложилось, что большинство примеров берется из области технических вычислений, но для использования примеров понимание численных методов не требуется. Они являются удобными и документально подтвержденными примерами различных типов параллельного программирования.

Повествование в настоящей книге организовано вокруг двух центральных концепций: основной процесс программирования (т. е. конструирование, кодирование, отладка и тестирование) и основные модели параллельного программирования (т. е. распределенные массивы, начальник/подчиненный и обмен сообщениями) [Lusk 2004]. Модель программирования на основе распределенных массивов рассматривается как базовая модель программирования на протяжении всего повествования. Распределенные массивы просты для понимания, предполагают меньший объем используемо-

го кода и прекрасно сочетаются с природой MATLAB. Распределенные массивы позволяют продемонстрировать сложные шаблоны взаимодействия более просто, нежели методом обмена сообщениями. Кроме того, они эффективно действуют как на многоядерных, так и на многоузловых параллельных машинах. Распределенные массивы являются естественной иллюстрацией основных концепций параллельного конструирования – параллелизма и локальности. Использованием распределенных массивов можно ограничиться в 90% параллельных приложений, однако, в некоторых ситуациях оптимальными являются другие модели программирования, что также упоминается при необходимости. Конечной целью данной книги является научить читателя «думать [распределенными] матрицами, а не сообщениями» [Moler 2005].

Книга построена следующим образом: сначала представляются конкретные примеры, а затем идет подробное обсуждение наиболее общих принципов параллельного программирования, которые иллюстрируют данные примеры. Книга нацелена на применение данных идей для решения особой задачи написания параллельных программ в MATLAB.

Чтобы учесть все возможные пожелания читателей, книга разделена на три части. Часть I (Главы 1–3) представляет основной концептуальный обзор ключевых принципов программирования, иллюстрированных конкретными примерами. Часть II (Главы 4–6) основное внимание уделяет анализу методов эффективного параллельного программирования. Часть III состоит из изучения особых случаев. Части I, II и III можно рассматривать независимо друг от друга и использовать в качестве модулей в более крупном курсе. Наконец, учитывая жесткие временные рамки, в которых находятся профессиональные пользователи, каждая глава является законченной по смыслу, и из главы в главу по необходимости переносятся определения всех ключевым терминам. Каждая глава начинается с краткого резюме представленной в ней информации. В конце главы даются ссылки на дополнительные источники. Такая организация позволяет пользователю выбрать и использовать только необходимую ему главу.

Глава 1 начинается с представления системы записи и базовых интерфейсов параллельного программирования, используемых на протяжении всей книги. Глава 2 дает краткое введение в принципы создания и выполнения простых параллельных программ. Эта глава дает читателю понять простоту использования параллельного пакета MATLAB. В главе 3 рассматриваются более сложные примеры, и выявляется взаимодействие с распределенными массивами. Глава 4 делает широкий обзор области параллельного программирования с раскрытием ключевых принципов, которые подробнее рассматриваются далее по ходу книги. В главе 4 используется более сложный пример, чем представленные в главе 3, а также освещаются граничные условия различных моделей параллельного программирования. Кроме того, в главе 4 поднимается ряд ключевых вопросов, связанных с параллельным программированием:

Конструирование: когда нужно использовать параллельную обработку, параллелизм и локальность, как рассчитать параллельную производительность, какую модель параллельного программирования использовать распределенные массивы, клиент/сервер или обмен сообщениями.

Кодирование: как и где использовать параллельный код, как написать масштабируемый код (например, масштабированный файл I/O), эффективный стиль кодирования.

Отладка: какие методы применять для перехода от последовательного к параллельному выполнению, на предмет каких ошибок нужно проверять каждый этап программирования.

Тестирование: как измерить достигнутую производительность и сравнить с расчетной.

Глава 5 представляет теорию, алгоритмическую запись и «взгляд изнутри» на программирования на основе распределенных массивов. Глава 6 представляет метрики оценки производительности и кодирования параллельной программы. Глава 7 выборочно дает представление о методах анализа параллельных приложений, особо подчеркивая, каким образом приводимые в книге примеры связаны со многими доменами приложений. В главах 8-10 приводятся примеры хорошо изученных тестовых программ из набора HPC Challenge benchmark (см. http://www.hpcchallenge.org) для иллюстрации решений задач, поставленных параллельным конструированием, кодированием, отладкой и тестированием.

На протяжении всего повествования в книге мы ссылаемся на множество классических примеров параллельного программирования, которые уже хорошо изучены и представлены в литературе. Надеюсь, что с помощью данных примеров читатель сможет оценить простоту и изящество параллельного программирования в MATLAB настолько, насколько это сделал я.

Благодарности

Я бы хотел выразить благодарность многим людям, которым обязан воплощением в жизнь языка параллельного программирования на MATLAB и этой книги. Невозможно перечислить их всех, и я бы хотел заранее извиниться перед теми, кого случайно не упомяну.

Разработка языка параллельного программирования MATLAB стала длительным мероприятием, в котором принимало участие много наших коллег, внесших важный вклад в конечный результат. Этой книгой обозначена важная веха в развитии данной области: широкая доступность и приемлемость языка MATLAB. На техническом уровне это означает широкую доступность и приемлемость слияния модели параллельного программирования на основе распределенных массивов с языками высокого уровня, каким является MATLAB.

Мне помогали многие люди, оказавшие непосредственное влияние на содержание данной книги. Первое представление о численных вычислениях я получил в Колледже Помона (en:Pomona College) на семинаре для студентов первого курса, проводимом профессором Вейном Стайнмецом, а также при дальнейшей поддержке моих научных руководителей – профессоров Каталин Митеску и Роберта Чембера.

Летом 1990 года я работал в компании Aerospace Corporation, где доктор Джон Хэквелл познакомил меня с высокоуровневыми языками программирования на основе массивов. Затем, осенью 1990 года последовал мой первый опыт параллельных вычислений под руководством профессора Александра Чорина в Калифорнийском университете Беркли [Kerner 1990]. С тех пор я стремился расширять свой опыт. Первая возможность сделать это представилась мне в 1997 г., когда я работал с доктором Майей Гохаль и г-ном Роном Минником [Kerner et al. 1998]. Эта работа была поддержана в университете Принстона руководителем моей докторской диссертации профессором Дэвидом Спергелем.

В 1998 году я приступил к работе в Линкольнской лаборатории Массачусетского технологического института, где под руководством г-на Роберта Бонда действовала группа разработчиков по созданию одной из первых библиотек параллельных массивов на языке C++ [Rutledge & Kerner 1999]. Весной 2001 года при поддержке профессора Стана Ахальта я написал библиотеку MatlabMPI за одну неделю [Kerner 2001]. Ее первым пользователем был доктор Гил Рац. Несмотря на трудность использования, MATLABMPI нашла своих сторонников в штате Линкольн и Огайо (благодаря усилиям профессора Ашока Кришнамурти и доктора Джона Нербасса).

Летом 2001 года я впервые познакомился с работой StarP профессоров Алана Эдельмана и Джона Гилберта и их студентов, доктора Перри Хазбандса, г-на Рона Чоя и доктора Вирала Шоу [Husbands 1999]. На этом этапе я понял, что принципы распределенных массивов, разработанные в библиотеках C++, могут быть адаптированы в языке MATLAB с базовой библиотекой MatlabMPI. По этому принципу г-жа Надя Блисс создала библиотеку pMATLAB [Kerner & Bliss 2003], которая при объединении с интерактивной концепцией LLGrid, разработанной совместно с доктором Альбертом Рейтером и г-ном Эндрю МакКаби, стала очень популярной в Линкольне [Reither et al. 2004]. Эта ранняя версия языка MATLAB была поддержана изнутри г-ном Дэвидом Мартинесом и извне г-ном Джоном Грошем.

В течение следующих нескольких лет у меня была возможность тесно сотрудничать с доктором Кливом Молером и многими другими специалистами в MathWorks по мере того, как они выносили параллельный язык MATALB за пределы лаборатории и превращая его в готовый продукт.

В то же время я участвовал в разработке набора новых параллельных тестовых программ, предназначенных для тестирования новых параллельных архитектур и параллельных языков. Набор тестов HPL Challenge benchmark был разработан совместно с профессором Джеком Донгарра, доктором Пиотром Лужеком и доктором Бобом Лукасом [Luszczek at al. 2006]. Г-н Эндрю Фанк, г-н Райан Хейни, г-н Хан Ким, доктор Джулия Маллин, г-н Чарли Рейдер, а также многие из вышеупомянутых лиц внесли большой вклад в язык MATLAB реализацию этих тестов. Эта работа с параллельным языком MATLAB нашла внутреннюю поддержку в лице доктора Кена Сена и поддержку извне от г-на Роберта Грейбила.

Визуально представление этого технологического потока, приведшего к созданию pMATLAB, показано на рисунке 2.

Кроме того, я выражаю благодарность людям, которые помогали с разработкой технологии, и тем, кто помогал мне создавать эту книгу. Среди них мой редактор в SIAM г-жа Элизабет Гринспэн, выпускающий редактор в Линкольне г-жа Дороти Райан и студенты весеннего курса 2008 года по прикладным параллельным вычислениям Массачусетского технологического института. Наконец, я бы хотел поблагодарить нескольких неизвестных рецензентов, чьи комментарии способствовали улучшению этой книги.

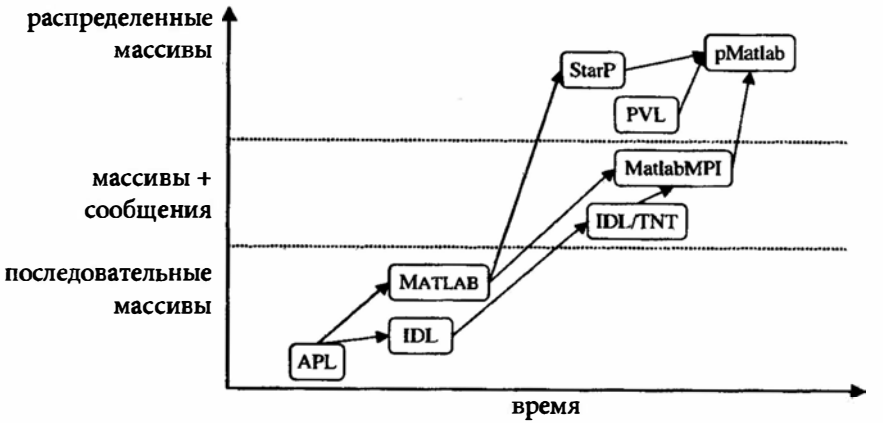


Рисунок 2. Развитие языка параллельного программирования MATLAB
Взгляд на эволюцию некоторых технологий, которые привели к разработке pMATLAB, начиная с первого языка на основе интерактивных массивов APL и заканчивая современной библиотекой pMATLAB

Часть I

Фундаментальные принципы

Глава 1
Основные понятия: система записи и интерфейсы

Резюме

В данной главе представлена система математических обозначений для описания параллельных алгоритмов, а также рассматривается соответствующий комплекс параллельных функций из библиотеки pMATLAB. Представлен метод параллельного программирования на основе отображений и приводится пример использования параллельной записи и данных функций.

В данной книге рассматривается вопрос параллельного программирования, который требует лаконичного описания параллельных алгоритмов. Для этих целей используется математический язык. В настоящей главе приводятся краткие сведения о базовой математической записи, используемой для описания параллельных алгоритмов, встречающихся в учебнике. Система записи (по необходимости) будет дополняться по ходу написания книги (для получения более полной информации о данной системе записи см. Приложение).

Для реализации данных алгоритмов в языке MATLAB необходим комплекс параллельных функций, соответствующих параллельным алгоритмам. Хотя и предполагается, что читатель знаком с пакетом MATLAB, однако, для написания параллельных программ на MATLAB используются новые функции. Книга написана с учетом ограниченного подмножества интерфейса библиотеки pMATLAB. В настоящей главе дается краткое описание некоторых ключевых функций, используемых для написания параллельных алгоритмов на протяжении всей книги. Данные функции (при необходимости) могут быть расширены в последующих главах.

1.1. Система записи алгоритмов

Числа являются основными единицами, которые будут обрабатываться алгоритмами, представленными в этой книге. Числа могут иметь различные типы: булевы, целые, вещественные, комплексные...

Для записи данных типов чисел используются следующие обозначения:

Тип	Наименование	Пример
Логический	B	да (1), нет (0)
Целый	Z	– 5,0,1,3
Вещественный	R	5,6,2, – 3.8,0.37
Комплексный	C	$5+6{,}2\sqrt{-1}$

Язык MATLAB оптимизирован для выполнения операций с массивом данных, и в основе его мощности лежит способность работать с наборами чисел в различных размерностях. В частности, он обеспечивает удобство работы со скалярными, векторными, матричными и тензорными величинами. Это будет обозначаться следующим образом:

Разм.	Наименование	Запись	Пример
0	Скалярная	Нижний регистр	a: R
1	Векторная	Нижний регистр жирный шрифт	a:RN
2	Матричная	Прописные, жирный шрифт	A: RN×N
3	Тензорная	Прописные, начертание, жирный шрифт	A: RN×N×N

Для обозначения субмассивов данных будет использоваться стандартная запись языка MATLAB. Например,

$$a = \mathbf{a}(i),\ \mathbf{a} = \mathbf{A}(i, :),\ \mathbf{A} = \mathbf{A}(i,:,:).$$

Аналогичным образом будет использоваться стандартная запись MATLAB для поэлементного сложения (+), вычитания (–), умножения (.*) и деления (./). С использованием данной записи простой алгоритм для прибавления единицы к матрице может быть записан как Алгоритм 1.1.

Алгоритм 1.1. Последовательное добавление единицы

```
Y : R^N×N = AddOneSerial(X : R^N×N)
1 Y=X+1
```

Запись распределенного массива данных

Для описания параллельных алгоритмов необходимо ввести дополнительные обозначения. В частности, количество копий или экземпляров выполняемых программ обозначается *Np*. Когда алгоритм выполняется в параллельном режиме, это означает, что один и тот же алгоритм (или код) выполняется каждым экземпляром программы. Такая вычислительная модель получила название Single-Program Multiple-Data (SPMD) (одна программа, несколько потоков данных, или архитектура SPMD) [Flynn 1972, Darema 1984]. Чтобы различать экземпляры программы, каждой из них присваивается уникальный идентификатор процессора – P_{ID} – который изменяется от 0 до

Np –1. Такие номера P_{ID} возможно ввести в большом количестве вычислительных систем. Например, предположим, что Np экземпляров программы выполняется на многоузловом параллельном компьютере, содержащем Np узлов таким образом, что каждый узел выполняет один P_{ID}. Данный тип исполнения будет обозначаться Np *1. Другим распространенным примером является выполнение всех экземпляров P_{ID} на одном узле. Данный тип исполнения будет обозначаться 1*Np. Если узел является многоядерным, существует возможность для ускорения программы. Если узел имеет только одно ядро, ускорение программы выполнить практически невозможно, однако данный вариант является оптимальным в случае параллельной отладки или тестирования программ. *Примечание:* Np означает количество созданных экземпляров P_{ID} независимо от того, сколько имеется физических узлов или ядер. И наконец, (Np /4)*4 может использоваться для обозначения смешанных ситуаций, где Np /4 означает узлы, каждый из которых выполняет четыре экземпляра P_{ID}. По ходу книги мы рассмотрим многоядерное выполнение (1*Np) и многоузловое выполнение (Np *1). К счастью, модель распределенных массивов данных сможет работать настолько эффективно, насколько возможно при имеющемся базовом аппаратном обеспечении, в обоих случаях, и одна и та же модель программирования может использоваться как для многоядерной, так и для многоузловой системы. Это является одним из основных преимуществ программирования на основе распределенных массивов.

При программировании на основе распределенных массивов необходимо отображать массивы данных на экземпляры P_{ID}. Такой процесс распределения описывает индексы распределенного массива, которыми «обладает» каждый P_{ID}. «Р-запись» используется в качестве удобного краткого условного обозначения для описания данного процесса распределения [Choy & Edelman 1999]. Матрица, распределенная так, что каждый P_{ID} имеет блок строк, обозначается следующим образом:

$$\mathbf{A} : \mathrm{R}^{P(N)\times N}.$$

Аналогично, матрица, которая распределена таким образом, что каждый P_{ID} имеет блок столбцов, обозначается как:

$$\mathbf{A} : \mathrm{R}^{N\times P(N)}.$$

Декомпозиция по строкам и столбцам одновременно может быть записана так:

$$\mathbf{A} : \mathrm{R}^{P(N)\times N} \text{ или } \mathbf{A} : \mathrm{R}^{P(N\times N)}.$$

В данном случае также необходимо уточнить «сетку» из P_{ID} (например, Np/4×4). Данные распределения показаны на рисунке 1.1.

1.1.2. Доступ к распределенным данным

Доступ к определенному элементу данных в распределенном массиве задается обычным обозначением субмассива. Если $\mathbf{A} : \mathrm{R}^{P(N)\times N}$, то $\mathbf{A}(i, j)$ заставит P_{ID}, который обладает (i, j)-м элементом **A**, отправить данное значение всем другим экземплярам P_{ID}. Точно так же, при заданных двух матрицах с различными распределениями $\mathbf{A} : \mathrm{R}^{P(N)\times N}$ и $\mathbf{B} : \mathrm{R}^{N\times P(N)}$ выражение

$$\mathbf{B} = \mathbf{A}$$

вызовет перераспределение данных из **A** в новое распределение **B**.

Доступ только к локальной части распределенного массива обозначается с помощью дополнения *.loc*. Для **A** : $R^{P(N)\times N)}$ локальной частью является **A**.*loc* : $R^{(N/N_P)\times N}$. Данное обозначение очень полезно при определении полностью локальных операций для каждого P_{ID}, которые не требуют коммуникации. При использовании данного обозначения простой параллельный алгоритм для прибавления единицы к матрице может записываться как Алгоритм 1.2.

Алгоритм 1.2. Параллельное добавление единицы

Y: $R^{P(N)\times N}$ = AddOneParallel(**X**: $R^{P(N)\times N}$)

1 **Y**.*loc* = **X**.*loc* + 1

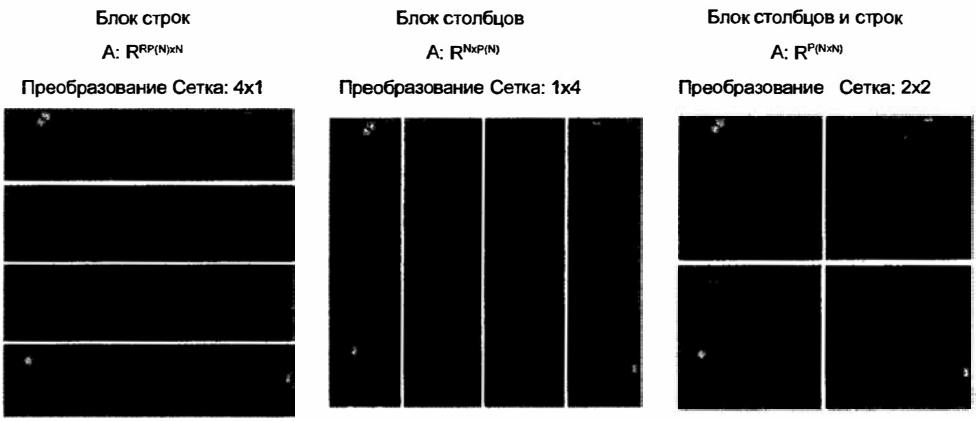


Рисунок 1.1. Параллельные карты размещения
Набор распределений, поддерживаемых большинством сред программирования на основе распределенных массивов

1.2. Интерфейсы параллельных функций

В предыдущем разделе описывались некоторые из параллельных алгоритмических операций, необходимых для написания параллельных программ. Для фактической их реализации нужны соответствующие параллельные функции. Данная книга написана с использованием подмножества функций библиотеки pMATLAB. Перечень функций pMATLAB по ходу книги будет расширяться. Параллельные функции можно разбить на несколько групп.

Параллельные параметры времени выполнения. Данные функции предоставляют информацию обо всей параллельной среде, выполняющей программу, в целом.

Функция *Np* – возвращает общее количество экземпляров MATLAB, выполняющих программу в настоящий момент (например, *Np*). Запись:

```
Np;
```

Функция P_{ID} – возвращает идентификатор каждого экземпляра MATLAB, выполняющего программу в данный момент. Принимает значение 0 на первом экземпляре MATLAB и увеличивается до *Np*–1 на последнем экземпляре MATLAB. Запись:

```
pid;
```

Конструирование распределенного массива. Данные функции позволяют использовать и строить распределенные массивы. При корректном применении ни одна из данных функций не должна исполнять коммуникацию между экземплярами MATLAB.

`map ( [Np 1], {} ,0:Np-1)` Конструирование одномерного размещения для распределения массива вдоль 1-го измерения. Использование:

```
Amap=map([Np 1],{},0:Np-1));
```

`map ( [1 Np], {},0 :Np-1)` Конструирование одномерного размещения для распределения массива во 2-м измерении. Использование:

```
Amap=map([Np 1],{},0:Np-1));
```

`zeros (N1, ... .map)` Перегруженная функция zeros, которая обеспечивает конструирование распределенного массива с размещением, задаваемым при помощи map. Использование:

```
A = zeros(200,100,Amap);
```

`local (A)` Возвращает локальную часть распределенного массива в качестве обычного численного массива MATLAB. Использование:

```
Aloc = local(A);
```

`global_ind (A, dim)` Возвращает список глобальных индексов, которые являются локальными для P_{ID} в указанном измерении dim. Использование:

```
myl = global_ind(A,1);
```

`put_local(A,Aloc)` Копирует обычный численный массив MATLAB в локальную часть распределенного массива. Использование:

```
A = put_local(A,Aloc);
```

Переразмещение распределенного массива. Данные функции позволяют изменять размещение распределенного массива. Эти функции почти всегда выполняют коммуникацию между различными P_{ID}. = Присваивание данных одного распределенного массива другому. Использование:

```
B(:,:) =A;
```

`agg(A)` Копирование всего содержимого распределенного массива в обычный численный массив MATLAB на «ведущем» P_{ID} (т. е. $P_{ID} = 0$). Использование:

```
Aagg = agg(A);
```

Коммуникация точка-точка. В основе реализации всех вышеупомянутых функций лежат функции, обеспечивающие коммуникацию точка-точка между экземплярами MATLAB. Эти функции следует использовать только при необходимости, так как это приводит к сложностям при отладке конечной программы.

`SendMsg (dest , tag, var1, var2, ... )` Отправляет указанные переменные var1, var2,... в экземпляр MATLAB, назначенный при помощи P_{ID} = dest. Сообщение имеет тэг (tag), который обычно является числовым значением от 1 до 256. Использование:

```
SendMsg(2,9,pi,i);
```

`RecvMsg(source,tag)` Получает переменные, отправленные экземпляром MATLAB с P_{ID}==source с соответствующим тэгом. Использование:

```
[pi i] = RecvMsg(1,9);
```

1.2.1. Программирование на основе карты размещения

Указанный выше относительно короткий перечень функций является исчерпывающим для выполнения широкого разнообразия параллельных программ. Задачей является реализация параллельных программ, которые просты в кодировании, отладке, тестировании и обеспечивают высокую производительность. В книге описывается программирование «на основе карты размещения» [Lebak et al. 2005] с применением метода separatio N-of-concerns (разделение ответственности), целью которого является создание алгоритма и его отображения для системы координат параллельных процессоров.

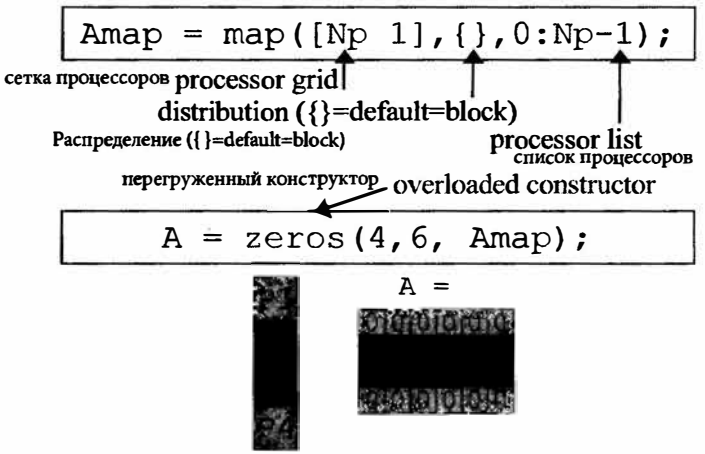


Рисунок 1.2. Структура карты размещения

Карта размещения числового массива создает назначение индексов массива процессорам. Оно состоит из сетки процессора, описания распределения и списка идентификаторов-процессоров. Перегруженный конструктор использует карту для создания распределенного массива.

Карты параллельного размещения содержат информацию только о том, каким образом массив разбивается на группы P_{ID}, и добавление карты не должно влиять на функциональную корректность программы. Карта (см. рис. 1.2) содержит описание того, каким образом индексы массива распределяются по группам P_{ID}. Кроме того, требуются соответствующие конструкторы массивов, которые используют карты для создания распределенных массивов. Пример данного вида программирования показан на примере простой программы AddOne в листинге кода 1.1 (см. также примеры Examples/ AddOne/pAddOne.m).

Листинг кода 1.1. Параллельная программа добавления единицы на MATLAB

```
1   N = 100;                              % Set matrix size.
2   XYmap = map ( [Np=1],{},0:Np-1);      % Create parallel map.
3   X = zeros(N,N,XYmap);                 % Create distributed x.
4   Y = zeros(N,N,XYmap);                 % Create distributed Y.
5   Xloc    local(X);                     % Get local part of X.
6   Yloc = local(Y);                      %       Get local part of Y.
7   Yloc = Xloc + 1;                      %       Add one to local part.
8   Y = put_local(Y,Yloc);                % Put back into Y.
```

В вышеуказанной программе строка 2 создает карту размещения для декомпозиции массива в первом измерении. Строки 3 и 4 используют эту карту для создания соответствующих распределенных массивов. Строки 5 и 6 получают локальные части данных распределенных массивов. Строка 7 выполняет вычисление на локальных частях. Строка 8 помещает результат местного вычисления обратно в распределенный массив. Данный стиль программирования будет использоваться в большинстве примеров, встречающихся в нашей книге. Основным преимуществом данного стиля программирования является то, что он поддается четко определенному процессу кодирования–отладки–тестирования, позволяющему писать высокопроизводительные параллельные программы в сжатые сроки.

1.2.2. Параллельное исполнение

Итак, система математической записи и параллельных функций для указания и выполнения параллельных программ на MATLAB уже определена, и настало время запустить и выполнить программу. Служебная программа pRUN, сопровождающая программы, приведенные в данной книге, используется для исполнения сценария MATLAB на множественных экземплярах MATLAB (см. инструкции по установке на веб-сайте книги: http://www.siam.org/KepnerBook). Например, перейдите в директорию Examples/AddOne (например, выполнив команду, cd Examples/AddOne), запустите MATLAB и пропишите

```
eval(pRUN('pMandelbrot' ,2,{}»))
```

При этом выполнится программа pAddOne посредством запуска двух экземпляров MATLAB на локальном узле. Данное выполнение будет обозначаться 1*2 согласно вышеуказанной записи. На рисунке 1.3 показано, что происходит во время исполнения. Когда выполняется параллельная программа, она создает *Np* (или *Np*) копий одной и той же программы. Каждая копия имеет уникальный P_{ID} (или `Pid`). Распределенные массивы неявно используют P_{ID}, чтобы определить, какие части каким процессорам принадлежат. В итоге, по завершении программы, выполняется закрытие и выход из всех экземпляров MATLAB.

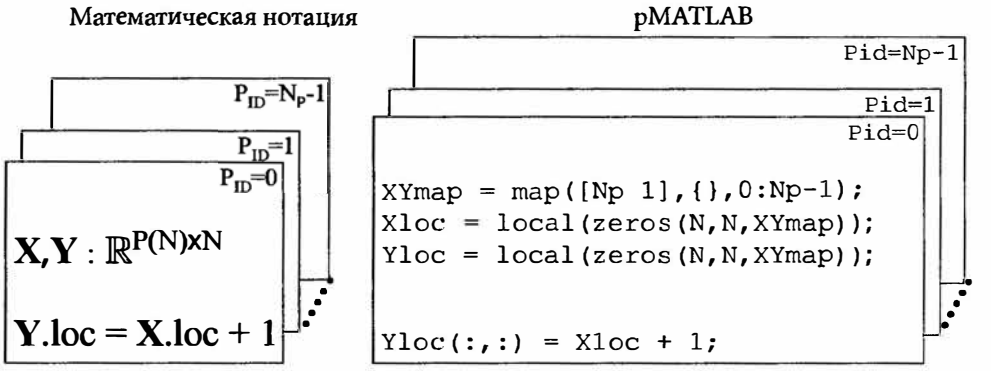


Рисунок 1.3. Параллельное исполнение

При параллельном выполнении программы в математической записи или кода MATLAB создается *Np* (или Np) копий одной и той же программы. Каждая копия имеет уникальный P_{ID} (или Pid)

Список литературы

1. *[Choy & Edelman 1999] Ron Choy and Alan Edelman, PARALLEL MATLAB: Doing it right, Proceedings of the IEEE (Feb. 2005), Vol. 93, No. 2, pp. 331–341.*

2. *[Darema 1984] Francis Darema, IBM Memo, 1984.*

3. *[Flynn 1972] Michael Flynn, Some computer organizations and their effectiveness, IEEE Trans. Comput., Vol. C–21, p. 948, 1972.*

4. *[Lebak et al. 2005] James Lebak, Jeremy Kepner, Henry Hoffmann, and Edward Rutledge, PARALLEL VSIPL++: An open standard software library for highperformance PARALLEL signal processing, Proceedings of the IEEE (Feb. 2005), Vol. 93, No. 2, pp. 313–330.*

Глава 2
Введение в pMATLAB

Резюме

В настоящей главе рассмотрено несколько примеров параллельных программ на языке MATLAB. В каждом из этих примеров реализуется метод программирования на основе карт размещения и соответствующий ему процесс разработки–написания программы–отладки–тестирования. В качестве примера мелкозернистого приложения с тривиальным массовым параллелизмом используется вычисление множества Мандельброта. В качестве примера крупнозернистой программы с массовым параллелизмом используется приложение ZoomImage. На примере программы ParallelIO показано, как нужно читать и записывать файлы на многопроцессорных системах.

Добро пожаловать в параллельное программирование на языке MATLAB. В ходе ознакомления с данной главой читатель сможет убедиться в преимуществах производительности нескольких параллельных программ на MATLAB. В данной главе дается краткое изложение нескольких программ, позволяющее получить представление об основных принципах написания и выполнения множества параллельных программ. Каждый пример является самодостаточным и может рассматриваться независимо от других. [Примечание: для практического выполнения данных примеров необходимо установить соответствующие программы и базовое параллельное программное обеспечение. Инструкции по установке вы найдете на веб-сайте книги (http://www.siam.org/KepnerBook).] Последующие разделы имеют такую же структуру. Каждый раздел открывается подразделом «Начало работы», в котором программа выполняется на различных процессорах, при этом изучается ее результат и регистрируется время выполнения. Во втором подразделе представлен математический аспект задачи, а также базовая параллельная реализация. В третьем подразделе описывается, каким образом была написана параллельная программа на MATLAB. В четвертом подразделе рассматривается процесс отладки программы. Завершается раздел изучением временных характеристик и оценкой общей производительности. В конце настоящей главы следует обсуждение работы программ и то, как была достигнута высокая производительность.

2.1. Программа: Mandelbrot (мелкозернистый тривиальный массовый параллелизм)

Наиболее распространенным способом применения параллельных вычислений является многократное выполнение одной и той же программы с незначительно отличающимися друг от друга входными параметрами. Такие программы обычно относят к «программам с тривиальным массовым параллелизмом», потому что они могут выполняться независимо друг от друга. При выполнении данных программ не требуется коммуникации [Moler 1985]. Конечно, в таких программах нет ничего тривиального. Данный термин означает, что их достаточно просто заставить эффективно функционировать на параллельном компьютере. В общем случае выбор среди различных

видов параллельных реализаций в пользу «тривиального массового параллелизма» почти всегда является наилучшим.

Первая программа применяет подход «тривиального массового параллелизма» для вычисления множества Мандельброта [Mandelbrot 1982], одного из наиболее популярных приложений для реализации в параллельных вычислениях, так как с его помощью достаточно просто получить хорошую производительность. В то же время эта задача является представителем широкого класса задач: простых, но в то же время требующих большого объема вычислений. В конечном итоге в качестве результата работы программы получаем характерное изображение множества Мандельброта в виде «Мечехвоста», которое давно украшает обложки многих научных изданий и является популярной экранной заставкой.

2.1.1. Начало работы

Для выполнения программы запустите MATLAB и перейдите в директорию Examples/Mandelbrot (т. е. введите `cd Examples/Mandelbrot`). В командной строке наберите

```
pMandelbrot
```

В этом случае программа выполняется на вашем локальном процессоре и выводит изображение (см. рис. 2.1) и подобные временные характеристики:

```
Compute Time / время вычисления (sec/c)=29.396
Launch+Comm Time/время запуска+взаимодействия(sec/c)=0.003034
```

Запишите эти временные характеристики для последующего анализа. Повторите процедуру для каждой из следующих команд:

```
eval(pRUN('pMandelbrot' ,1,{}))
eval(pRUN('pMandelbrot' ,2,{}))
eval(pRUN('pMandelbrot' ,4,{}))
```

Данные команды выполняют программу Mandelbrot с использованием одного, двух и четырех P_{ID}. При выполнении на параллельном компьютере выдаются значения временных параметров, похожие на указанные в таблице 2.3.

Поздравляем! Вы успешно выполнили свою первую параллельную программу MATLAB. Полученные результаты будут изучены более подробно в ходе дальнейшего обсуждения программы и получения более полного представления о том, что означают данные временные характеристики.

Важное примечание: последний аргумент команды `pRUN` определяет, на каком параллельном компьютере будет выполняться программа. Может возникнуть необходимость в получении надлежащих установочных параметров у вашего системного администратора; для получения более подробной информации обращайтесь на веб-сайт книги. Установка последнего аргумента значением {} приводит к выполнению множества копий программы на вашем локальном компьютере и получению функциональных характеристик, отличных от указанных в таблице 2.3.

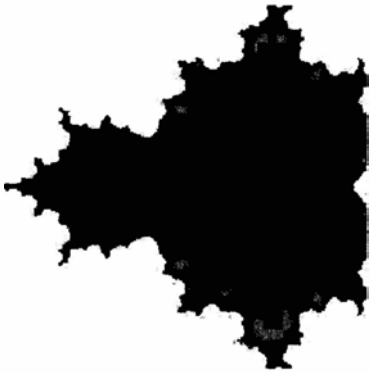


Рисунок 2.1. Результат программы Mandelbrot. Пример результата программы pMandelbrot

2.1.2. Разработка параллельной программы

Разработка параллельной программы начинается с указания того, что должна делать программа. В данном случае – вычислять множество Мандельброта. Множество Мандельброта вычисляется путем рекурсивного применения формулы

$$f(\mathbf{Z}) = \mathbf{Z}.*\mathbf{Z} + \mathbf{C} \ ,$$

где Z и C – комплексные числа. Изображение множества Мандельброта формируется путем перебора параметра C в своей области на комплексной плоскости:

$$-1.6 < Re(\mathbf{C}) < 0.4, -1 < Im(\mathbf{C}) < 1,$$

где `Re()` и `Im()` – мнимая и действительная части соответственно. Базовый алгоритм для вычисления множества Мандельброта показан в Алгоритме 2.1 (более подробная информация в подписи к алгоритму).

Алгоритм 2.1. Последовательный алгоритм Мандельброта

В качестве исходных данных алгоритм принимает размер вычисляемой матрицы N, максимальное число выполняемых итераций N_{Iter} и нижний порог ε. Комплексные матрицы **Z** и **C** инициализируются в строках 1 и 2. В строках 5 и 6 вычисляется следующая итерация множества Мандельброта, и значения масштабируются в выходную матрицу **W**. Значения, превышающие порог, определяются в строке 7.

$\mathbf{W} : \mathbb{R}_{+}^{N \times N} = \textsc{MandelbrotSerial}(N, N_{step}, \epsilon)$
1 $\mathbf{Z}, \mathbf{C} : \mathbb{C}^{N \times N} \quad i_\epsilon = 1 : N^2$
2 $\mathbf{C}(i,j) = (jN/2 - 1.6) \quad (iN/2 - 1)\sqrt{-1}, \qquad i,j = 1 : N$
3 **for** $i_{step} = 1 : N_{step}$
4 **do**
5 $\mathbf{Z}(i_\epsilon) = \mathbf{Z}(i_\epsilon) \,.*\, \mathbf{Z}(i_\epsilon) \quad \mathbf{C}(i_\epsilon)$
6 $\mathbf{W}(i_\epsilon) = \exp(-|\mathbf{Z}(i_\epsilon)|)$
7 $i_\epsilon = \arg(\mathbf{W} > \epsilon)$

В данный момент точные детали вычислений нам не важны. Наиболее значимая информация заключается в следующих двух свойствах.

- Вычисление множества Мандельброта представляет собой многократное повторение одного и того же вычисления для конкретного параметра **C***(i,j)*.
- Результат для одного значения **C***(i,j)* не зависит от других и может быть вычислен независимо (т. е. параллельно).

Первое свойство означает, что задача в достаточной степени содержит в себе параллелизм (т. е. есть вычисления, которые фактически могут быть выполнены одновременно). В общем случае параллелизм в программе обнаружить несложно. Второе свойство означает, что данный конкретный параллелизм имеет высокую локальность данных (т. е. данные для выполнения параллельного вычисления являются локальными для этого P_{ID}). Локальность найти в программе сложнее, однако она является ключевым фактором для получения хорошей параллельной производительности. Эти два свойства – общие для многих типов вычислений и выступают в качестве основных требований для параллельных программ с массовым параллелизмом.

В первую очередь при разработке параллельной программы необходимо решить, каким образом нужно распределить параметры между разными P_{ID}. Данный процесс будет именоваться *размещением* алгоритма на параллельном компьютере. Стандартный способ распределения для мелкозернистых задач с тривиальным массовым параллелизмом заключается в передаче каждой части области **C** отдельной группе P_{ID}. Например, если имеется четыре P_{ID}, область определения можно было бы распределить (как показано в таблице 2.1). Такое распределение разбивает область в первом измерении и дает разные вертикальные полосы каждому P_{ID}. В результате распределения получаем параллельный алгоритм, показанный с использованием P-записи в Алгоритме 2.2.

Подход к параллельному программированию на основе карт размещения является основой параллельного программирования распределенных массивов. Он был неоднократно реализован (см. Главу 5 и указанные ссылки). Позже упрощенная форма подхода на основе карт была успешно адаптирована для работы с определенными классами крупномасштабных приложений распределенного программирования [Dean & Ghemawat 2004].

Таблица 2.1. Параллельное распределение в программе Mandelbrot

Диапазон изменения C распределен для $Np = -4$ таким образом, что каждый P_{ID} будет выполнять вычисления для отдельной вертикальной полосы

P_{ID}	Действительный диапазон	Мнимый диапазон
0	$-1.6 < Re(\mathbf{C}) < -1.1$	$-1 < Im(\mathbf{C}) < +1$
1	$-1.1 < Re(\mathbf{C}) < -0.6$	$-1 < Im(\mathbf{C}) < +1$
2	$-0.6 < Re(\mathbf{C}) < -0.1$	$-1 < Im(\mathbf{C}) < +1$
3	$-0.1 < Re(\mathbf{C}) < +0.4$	$-1 < Im(\mathbf{C}) < +1$

Алгоритм 2.2. Параллельный алгоритм Mandelbrot

W: R +P(N)XN = MANDELBROTPARALLEL(N,N_{step},e,Np)
1 **Z,C:**CP(N)XN i_e=1, ... ,$N2/Np$
2 **C**(i,j) = ($jN/2 - 1.6$) + ($iN/2 - 1$)$\sqrt{-1}$, $i,j = 1 : N$
3 **for** i_{step} = 1 : Nstep
4 **do**
5 **Z**.$loc(i_e)$ = **Z**.$loc(i_e)$. * **Z**.$loc(i_e)$ + **C**.$loc(i_e)$
6 **W**.$loc(i_e)$ = exp($-|$**Z**.$loc(i_e)|$)
7 i_e = arg(**W**.loc > e)

Во-вторых, при разработке параллельной программы необходимо определить, каким образом доставить каждому P_{ID} информацию, необходимую ему для вычислений. В данном примере используется распределенный массив C, что позволяет каждому P_{ID} определить часть области, над которой он должен работать. Затем каждый P_{ID} выполняет вычисления на своей локальной части матрицы W. Ведущий процессор P_{ID} может получить локальные матрицы от других P_{ID}, а также собрать и отобразить конечную матрицу. Визуальное представление такой реализации параллельной программы показано на рисуике 2.2.

2.1.3. Код

Следующим шагом станет написание параллельной программы Mandelbrot. При стандартном (и рекомендуемом) подходе к написанию следует начать с корректной последовательной программы Mandelbrot. Отладка параллельных программ может представлять некоторую сложность, и лучше всего начать с программы, которая уже работает и может быть проверена на корректность по мере добавления параллельных конструкций. Как уже было сказано в предыдущем подразделе, для параллельного кода необходимы три дополнительные части фуикциональных возможностей.

- Каждый P_{ID} создает распределенные массивы W, C и Z с использованием карты размещения.
- Каждый P_{ID} определяет часть области, которую он будет вычислять, посредством обращения к локальной части матрицы C.

- Ведущий экземпляр P_{ID} получает локальные части **W** и формирует конечный результат.

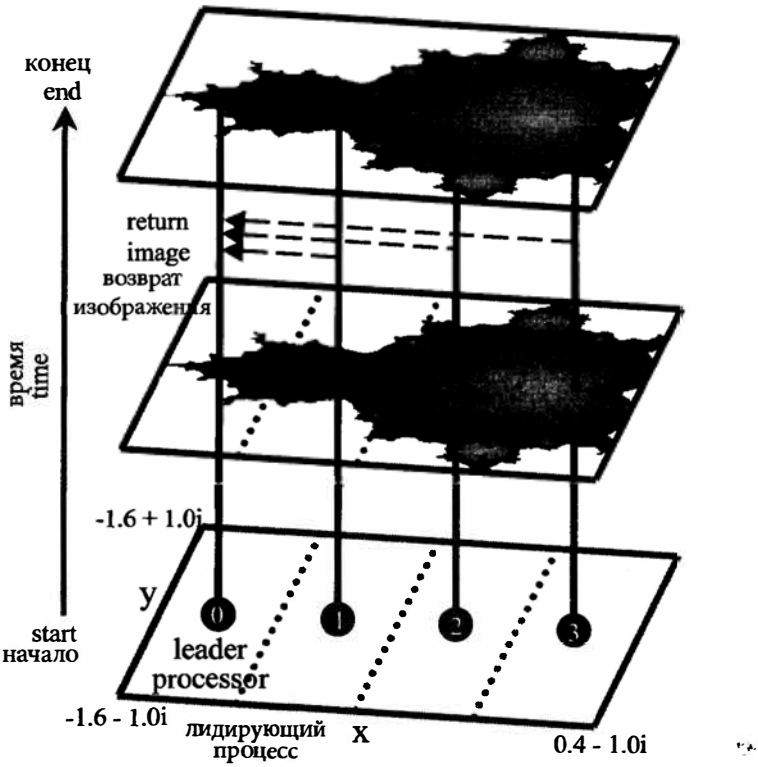


Рисунок 2.2. Реализация параллельной программы Mandelbrot
Каждый экземпляр P_{ID} определяет, какую часть области ему необходимо вычислить, вычисляет локальную часть матрицы и посылает ее ведущему коду P_{ID} (пунктирные линии). Затем ведущий P_{ID} формирует и отображает конечную матрицу

Ядро параллельного кода показано в Листинге кода 2.1 (полный листинг расположен в Examples/Mandelbrot/pMandelbrot.m). Следует понимать, что один и тот же код будет выполняться каждым экземпляром MATLAB. Единственным различием между экземплярами является наличие у каждого из них уникального P_{ID} (т. е. `Pid`), который используется для определения того, какие части распределенного массива принадлежат каждому из экземпляров MATLAB.

Листинг кода 2.1. Параллельная программа MATLAB для вычисления множества Мандельброта (программа Mandelbrot)

```
1 % Set number of iterations, mesh size and threshold.
2 N=2400; Niter=20; epsilon = 0.001;
3 PARALLEL = 1;                          % Set control flag.
4 Wmap = 1;                              % Create serial map.
5 if (PARALLEL)
6 Wmap = map ( [Np 1], {}, 0: Np -1);    % Create parallel map.
7 end
8 W= zeros (N,N, Wmap);                  % Distributed array
9 Wloc = local (W);                      % Get local part.
10 myI = global_ind(W,1);                % Get local i indices.
11 myJ = global_ind(W,2);                % Get local j indices.
12 [ReC ImC] = meshgrid(myJ./(N/2) -1.6, myI./(N/2) -1);
13 Cloc = complex(ReC, ImC);             % Initialize C.
14 Zloc = Cloc;                          % Initialize Z.
15 ieps = 1: numel (Wloc);               % Initialize indices.
16 tic;
% Start Clock.
17 for i=1 : Niter;                      % Compute Mandelbrot set.
18 Zloc(ipes) = Zloc(ipes) .* Zloc(ipes) + Clock(ipes);
19 Wloc(ipes) = exp(-abs(Zloc(ipes)));
20 ieps=ieps( find(Wloc(ipes) > epsilon) );
21 end
22 W = put_local (W, Wloc);              %Put back into W;
23 Tcompute = toc;                       % Stop Clock.
24 tic;                                  % Start Clock.
25 W1 = agg(W);                          %Gather back to leader.
26 Tcomm = toc;                          %Stop Clock.
```

Перевод

1 Установить количество итераций, шаг ячейки и пороговое значение;
3 Установить контрольный флаг;
4 Создать последовательную карту размещения;
6 Создать параллельную карту размещения;
8 Распределенный массив;
9 Получить локальную часть;
10 Получить локальные индексы i;
11 Получить локальные индексы j;
13 Инициализация C;
14 Инициализация Z;
15 Инициализация индексов;
16 Запустить часы;
17 Вычислить множество Мандельброта;
22 Поместить обратно в W;
23 Остановить часы;

24 Запустить часы;
25 Собрать и направить ведущему;
26 Остановить часы.

Строки 3–7 создают параллельное размещение `Wmap`. Флаг PARALLEL является дополнительным инструментом отладки, позволяющим включать и выключать распределенные массивы. Примечание: данный метод может применяться только в том случае, если все массивы объявлены как минимум с двумя измерениями (т. е. используйте `zeros(N,N)` вместо `zeros(N)` для создания квадратных матриц).

Строки 8–9 создают распределенный массив **W**, при этом каждый экземпляр MATLAB извлекает свою локальную часть `Wloc`.

Строки 10–12 содержат команду для каждого экземпляра MATLAB вычислить индексы `myI` и `myJ` распределенного массива **W**, соответствующие его локальной части. Эти индексы позже используются для создания начальных параметров `Cloc`, которые использует каждый экземпляр MATLAB для выполнения своей части вычислений.

Строки 13–21 являются ядром алгоритма множества Мандельброта и аналогичны строкам последовательного кода. В данном коде каждый экземпляр MATLAB будет вычислять множество Мандельброта с использованием локальных матриц `Cloc`, `Zloc` и помещать результат в `Wloc`.

В **строках 22 и 25** каждый P_{ID} помещает свою локальную часть результата `Wloc` назад в распределенный массив **W**. Затем результат собирается обратно в ведущий процессор MATLAB (т. е. $P_{ID}=0$).

Вышеуказанный код является ярким примером подхода к программированию на основе карт размещения. После того, как создана карта `Wmap`, она используется для создания распределенной матрицы **W** для хранения результатов вычисления множества Мандельброта. Теперь каждый экземпляр MATLAB может использовать распределенный массив для определения индексов, которыми он владеет, с использованием функции `global_ind`. Эти индексы впоследствии используются для вычисления локальной части областей `Cloc`, которые затем используются для вычисления локального множества Мандельброта. Локальные результаты `Wloc` копируются обратно в распределенный массив **W** с использованием функции `put_local`. Наконец, матрица **W** направляется «ведущему» при помощи функции `agg`.

На данном примере кода можно проследить некоторые свойства подхода к программированию на основе карт размещения.

- Незначительное влияние на код. Программа преобразована из последовательной в параллельную путем добавления около 10 дополнительных строк кода.
- Последовательный код сохраняется. При установке `PARALLEL=0` данный код может быть выполнен последовательно без поддержки параллельных библиотек, так как **W** будет лишь регулярным массивом MATLAB.
- Небольшой объем использования параллельной библиотеки. Потребовалось только пять функций параллельной библиотеки: `map`, `local`, `global_ind`, `put_local`, и `agg`.
- Масштабируемость. Код может выполняться на задаче любого размера и на любом количестве процессоров, так как размер локальных переменных не

учитывается ни в одной части вычислений, а `Wloc`, `Cloc` и `Zloc` вычисляются из `W`.

- Ограниченная коммуникация. Поскольку используются локальные переменные `Wloc`, `Cloc`, и `Zloc`, использование коммуникация строго контролируется.
- Независимость от карт размещения. Программа будет работать с большим числом разнообразных карт. Значение `Wmap` может быть заменено на `map ([1NP],{},0:NP-1)`, при этом код будет выполняться корректно.
- Гарантия производительности. Использование в основном алгоритме лишь локальных переменных `Wloc`, `Cloc`, и `Zloc`, которые являются обычными числовыми массивами MATLAB, гарантирует отсутствие скрытого ухудшения производительности при параллельном выполнении данных строк кода.

2.1.4. Отладка

Как упоминалось ранее, параллельная отладка представляет собой непростую задачу. Основным преимуществом подхода к программированию на основе карт размещения является то, что оно поддается строго определенному процессу написания-отладки-тестирования. Параллельный код вводит в программу новую функциональность, отладка которой производится независимо от последовательной функциональности. Наилучшим способом отладки такой программы является ее повторное выполнение в несколько этапов. Соответствующие шаги указаны в таблице 2.2 (более подробное обоснование смысла, скрытого за этими шагами, будет представлено в книге позже).

Шаг 1. Выполнение программы последовательно на локальном компьютере с распределенным массивом **W**, превращенным в обычный числовой массив. Таким образом обеспечивается сохранение корректности последовательного кода, а операции `global_ind` и `local` работают на обычных числовых массивах.

Таблица 2.2. Шаги отладки

Стандартные последовательно усложняющиеся шаги при отладке параллельной программы. При каждом шаге тестируются определенные аспекты кода

Шаг	Режим	Процессоры	Задача	Цель отладки
1	`PARALLEL=0`	1 (лок.)	Мал.	Корректный последов. код
2	`PARALLEL=1`	1 (лок.)	Мал.	Корректный парал. код
3	`PARALLEL=1`	2+ (лок.)	Мал.	Корректная лок. коммуникация
4	`PARALLEL=1`	2+ (удал.)	Мал.	Корректная удал. коммуникация
5	`PARALLEL=1`	2+ (удал.)	Бол.	Измерение производительности

Шаг 2. Выполнение программы последовательно на локальном компьютере с распределенным массивом **W**. Данный шаг позволяет убедиться, что операции над **W** все еще работают, когда он является уже распределенным.

Шаг 3. Выполнение программы на локальном компьютере в параллельном виде. На этом шаге определяется, является ли декомпозиция области корректной конкретно для данного примера - правильно ли произведено вычисление `Cloc` на основании `myI` и `myJ`. Он также определяет ошибки коммуникации, выполняемой при помощи опера-

ции agg. Из всех шагов этот, вероятнее всего, является самым важным и выявляет больше всего ошибок.

Шаг 4. Выполнение программы на удаленном компьютере в параллельном виде. На данном шаге определяются ошибки удаленной коммуникации, выполняемой при помощи операции agg. В основном на этом шаге определяются ошибки конфигурации при установке и настройке параллельного программного обеспечения MATLAB.

Шаг 5. Выполнение программы параллельно на удаленном компьютере, но с интересуемым размером задачи. До этого момента тестирование должно было осуществляться на задаче малого размера. Данный шаг позволяет определить производительность на реальной задаче.

Описанные шаги являются существенным отражением этапов, указанных в разделе «Начало работы». На каждом из шагов тестируются определенные аспекты параллельного кода. Кроме того, в наибольшем объеме тестирование производится локально на малых задачах, которые выполняются быстрее всего. Очень часто программисты пытаются выполнить отладку на больших задачах (при которых процесс выявления ошибок занимает больше времени), хотя эти ошибки можно было бы обнаружить, затратив намного меньше времени, посредством выполнения отладки на малых задачах.

2.1.5. Тестирование

Тестирование программы Mandelbrot – при условии, что она уже отлажена и выдает правильные ответы – в основном заключается в определении времени, которое требуется для ее выполнения, а также выяснения того, насколько ее производительность соответствует ожидаемой. Почти все временные показатели, необходимые для проведения такого анализа, были получены в разделе «Начало работы». В таблице 2.3 указано относительное время вычисления (приведенное к случаю $Np = 1$), время запуска + коммуникации и коэффициент ускорения вычисления. Эти данные были собраны на восьми-процессорной многоядерной системе и 16-узловой многоузловой системе. В многоядерных системах все процессоры используют общую память. В многоузловых системах каждый узел имеет отдельную память, и процессоры взаимодействуют по сети.

Целью параллельной программы является уменьшение времени, затрачиваемого на вычисления. В таблице 2.3 показано сокращение времени вычислений при увеличении NP как в многоядерной, так и в многоузловой системе.

Таблица 2.3. Производительность программы Mandelbrot на многоядерных и многоузловых системах

Относительное время вычислений, абсолютное время запуска + коммуникации, коэффициент ускорения вычислений на многоядерных и многоузловых вычислительных системах для различных значений *Np*. Данные о производительности приведены для фиксированного размера задачи (N и N_{iter} – постоянные величины). Все значения нормализованы по столбцу 2, в котором $Np = 1$ с выключенными распределенными массивами (PARALLEL=0). Столбцы 3–7 содержат значения с включенными распределенными массивами (PARALLEL=1). Запись 1*8 означает, что $Np = 8$ и все коды P_{ID} были выполнены на одном процессорном узле с общей памятью. Запись 16*1 означает, что $Np = 16$ и каждый P_{ID} был выполнен на своем индивидуальном процессорном узле с использованием отдельной памяти

Многоядерные процессоры:	1	1*1	1*2	1*4	1*8	
Относительное время вычислений	1	1.0	0.47	0.16	0.07	
Время запуска+коммуникации (с)	0.0	0.0	2.0	5.7	5.5	
Коэффициент ускорения вычислений	1	1.0	2.1	8.2	14	
Многоузловые процессоры:	1	1*1	2*1	4*1	8*1	16*1
Относительное время вычислений	1	1.0	0.47	0.16	0.06	0.02
Время запуска+коммуникации (с)	0.0	0.0	8.4	14.2	13.8	11.3
Коэффициент ускорения вычислений	1	1.0	2.1	6.3	18	42

Однако в параллельной программе обычно увеличивается время, затрачиваемое на запуск и коммуникацию между всеми экземплярами MATLAB. В таблице 2.3 прослеживается увеличение времени запуска + коммуникации с увеличением *Np*, а также сравнительно больший временной показатель запуска + коммуникации в многоузловой системе, так как запуск и коммуникация осуществляются через сеть.

Конечный этап анализа включает вычисление коэффициента ускорения, достигнутого благодаря параллельной программе. Формула коэффициента ускорения S:

$$S(N_P) = \frac{T(N_P = 1)}{T(N_P)},$$

где $T(N_p)$ – время вычисления с использованием *Np* процессоров. Применяя данную формулу к временным характеристикам вычислений, получаем значения коэффициента ускорения, указанные в таблице 2.3. Коэффициент ускорения вычислений демонстрирует производительность частей кода, которые были распараллелены. Этим подчеркивается одно очевидное, но очень важное свойство: параллельные вычисления воздействуют только на параллельно выполняемые части кода. Тестирование, и это очень важный аспект, необходимо начать с измерения коэффициента ускорения части программы, которая должна показать высокую производительность, потому что если эта часть работает неудовлетворительно, параллельная программа не сможет в дальнейшем компенсировать увеличенное время запуска – накладные расходы коммуникации.

Данная программа имеет высокую производительность и высокий коэффициент ускорения на вычислительной части кода. Примечание: проницательный читатель может посчитать, что коэффициент ускорения вычисления слишком высок (см. раздел 5.5). Однако, поскольку накладные расходы увеличиваются с ростом *Np*, общая производительность программы ограничена. Это побочный эффект того, что программа выполняется, по большей части, в течение лишь нескольких секунд. В большинстве реальных приложений стоит цель взять программу, выполняющуюся достаточно продолжительное время, и заставить ее работать быстрее. По мере того, как программа вычисления множества Мандельброта станет выполняться за более длительный период (т. е. при выборе больших значений N и/или N_{iter}), время вычислений станет намного превышать время запуска + коммуникации, и коэффициент ускорения вычислений будет отражать общую производительность параллельного приложения.

2.2. Программа ZoomImage (крупнозернистый тривиальный массовый параллелизм)

Пример с вычислением множества Мандельброта иллюстрирует концептуальное представление о параллельной программе с массовым параллелизмом, которая выполняет множество независимых мелкозернистых вычислений посредством распределения элементов матрицы между различными процессорами. В настоящем разделе на примере программы ZoomImage наглядно демонстрируются особенности крупнозернистой параллельной программы с массовым параллелизмом.

В примере ZoomImage берется начальное изображение (в данном случае это серый прямоугольник на черном фоне), и вычисляется множество новых изображений, каждое с несколько отличным коэффициентом увеличения. Результатом вычислений является последовательность изображений, которая создает ощущение стремительно приближающегося начального изображения.

2.2.1. Начало работы

Чтобы начать выполнение программы, запустите MATLAB и перейдите в директорию Examples/ZoomImage (т. е. введите `cd/Examples/ZoomImage`). В командной строке наберите

```
pZoomImage
```

В этом случае программа будет выполняться на вашем локальном процессоре и получит последовательность изображений (см. рис. 2.3), а также подобные временные характеристики:

```
Compute Time/время вычисления (sec/c)                    =22.7613
Launch+Comm Time/Время на запуск + коммуникацию (sec/c)  =0.003693
Performance/производительность (Gigaflops/гигафлопс)     =0.84701
```

Запишите эти временные характеристики для последующего анализа. Повторите процедуру для каждой из следующих команд:

```
eval(pRUN('pZoomImage',1,{}))
eval(pRUN('pZoomImage',2,{}))
eval(pRUN('pZoomImage',4,{}))
```

Данные команды выполняют программу ZoomImage на одном, двух и четырех процессорах. При выполнении на параллельном компьютере выдаются значения временных параметров, близкие к указанным в таблице 2.5.

На этом раздел «Начало работы» завершается. Полученные результаты будут изучены более подробно в ходе дальнейшего рассмотрения программы и получения более полного представления о том, что означают приведенные временные характеристики.

Важное примечание: Последний аргумент команды `pRUN` определяет, на каком параллельном компьютере будет выполняться программа. Может возникнуть необходимость получения надлежащих установочных параметров у вашего системного администратора; для получения более подробной информации обращайтесь на веб-сайт книги (http://www.siam.org/KepnerBook). Установка последнего аргумента значением {} приводит к выполнению множества копий программы на вашем локальном компьютере и получению производительности, отличной от указанной в таблице 2.5.

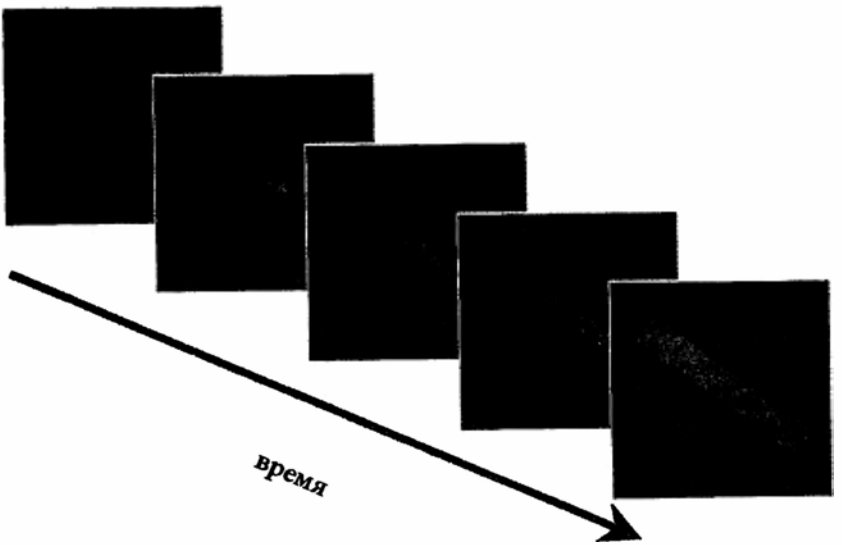


Рисунок 2.3. Результат программы ZoomImage.
Пример выходных данных программы `pZoomImage`

2.2.2. Разработка параллельной программы

Основное назначение данной программы заключается в вычислении измененного в масштабе изображения ***Zs*** размером N×N при заданном исходном изображении ***Zo*** и коэффициенте масштабирования. Существует много способов выполнить данную операцию. Простой метод заключается в размытии исходного изображения с после-

дующей его интерполяцией. Размытие осуществляется при помощи операции свертки исходного изображения с ядром **Ks** – матрицей размеров $N_K \times N_K$; (т. е. $Z_S^{blur} = K_S \circ Z_0$), где свертка (конволюция) определяется как

$$\mathbf{Z}_s^{blur}(i,j) = \sum_{i'=1}^{N_K}\sum_{j'=1}^{N_K} \mathbf{K}_s(i',j')\mathbf{Z}_0(i-i',j-j')$$

где $N_K = \lceil 5s\sigma \rceil$ и

$$\mathbf{K}_s(i,j) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left[\frac{(i-N_K/2)^2 + (j-N_K/2)^2}{\sigma^2}\right]$$

– это стандартная матрица с Гауссовым ядром. Этап интерполяции осуществляется с использованием стандартной билинейной интерполяции

$$\begin{aligned}\mathbf{Z}_s(i,j) = {} & a_{00}\mathbf{Z}_s^{blur}(i_s,j_s) + a_{10}\mathbf{Z}_s^{blur}(i_s+1,j_s) \\ & + a_{01}\mathbf{Z}_s^{blur}(i_s,j_s+1) + a_{11}\mathbf{Z}_s^{blur}(i_s+1,j_s+1)\end{aligned}$$

где $i_s = [i/s], j_s = [j/s]$ и

$$\begin{aligned} a_{00} &= (i-i_s)(1-j+j_s), \quad & a_{10} &= (i-i_s)(1-j+j_s) \\ a_{01} &= (1-i+i_s)(j-j_s), \quad & a_{11} &= (i-i_s)(j-j_s)\end{aligned}$$

Теперь рассмотрим применение вышеуказанных математических приемов для получения последовательности масштабированных изображений с использованием в качестве входных данных *Ns*-элементный вектор коэффициентов масштабирования **s**. Основной алгоритм для вычисления множества масштабированных изображений показан в Алгоритме 2.3 (подробная информация приведена в сопроводительной подписи).

Алгоритм 2.3 Последовательный алгоритм ZoomImage

Алгоритм в качестве исходных данных принимает исходное изображение Zo размеров $N \times N$, *Ns*-элементный вектор *s* коэффициентов масштабирования и ширину ядра размытия. На выходе получаем массив размеров $N \times N \times Ns$ из масштабированных изображений *Z*. Строка 1 организует цикл по каждому коэффициенту масштабирования в **s**. В строке 3 выбирается коэффициент масштабирования. Строка 4 производит свертку исходного изображения, а строка 5 осуществляет интерполяцию.

$\mathcal{Z} : \mathbb{R}_+^{N\times N\times N_s} = \textsc{ZoomFrames}(\mathbf{Z}_0 : \mathbb{R}_+^{N\times N}, \mathbf{s} : \mathbb{R}_+^{N_s}, \sigma)$

1 **for** $i_s = 1 : N_s$

2 **do**

3 $s = \mathbf{s}(i_s)$

4 $\mathbf{Z}_s^{blur} = \mathbf{K}_s \circ \mathbf{Z}_0$

5 $\mathcal{Z}(:,:,i_s) = \textsc{Interpolate}(\mathbf{Z}_s^{blur}, s)$

Для наших целей точные детали вычислений не важны. Наиболее значимая информация заключается в следующем:

- Вычисление множества масштабированных изображений представляет собой многократное повторение одного и того же вычисления для различных коэффициентов масштабирования s при одинаковом входном параметре Zo.
- Результат для одного значения $Z(:,:, i_s)$ не зависит от других и может быть вычислен независимо (т. е. параллельно).

Первое свойство означает, что задача в достаточной степени содержит в себе параллелизм (т. е. существуют N_s вычислений, которые фактически могут быть выполнены одновременно). Второе свойство означает, что вычисление каждого $Z(:, :, i_s)$ имеет высокую локальность данных (т. е. данные для вычисления каждой матрицы $Z(:, :, i_s)$ являются локальными для этого P_{ID}). Локальность найти в программе сложнее, однако, она является ключевым фактором для получения хорошей параллельной производительности. При стандартном подходе к задаче с крупнозернистым массовым параллелизмом каждому P_{ID} дается отдельный набор масштабирующих коэффициентов s для вычислений, а также владение соответствующей частью трехмерного массива Z. Например, при наличии четырех процессоров коэффициенты масштабирования от 1 до Ns могут быть распределены (как указано в таблице 2.4). В результате получаем параллельный алгоритм, показанный с использованием P-записи в Алгоритме 2.4.

Таблица 2.4. Параллельное распределение ZoomImage

Для вектора с Ns коэффициентами масштабирования S, назначенными четырем процессорам таким образом, что каждый P_{ID} вычисляет отдельный набор коэффициентов масштабирования

P_{ID}	Коэффициент масштабирования
0	$S\left(1:\left\lvert\frac{1}{4}\right\rvert Ns\right)$
1	$S\left(1:\left\lvert\frac{1}{4}\right\rvert Ns+1:\left\lvert\frac{1}{2}\right\rvert Ns\right)$
2	$S\left(\left\lvert\frac{1}{2}\right\rvert Ns+1:\left\lvert\frac{3}{4}\right\rvert Ns\right)$
3	$S\left(\left\lvert\frac{3}{4}\right\rvert Ns+1:Ns\right)$

Алгоритм 2.4. Параллельный алгоритм ZoomImage

Третье измерение выходных данных распределено между различными P_{ID} при объявлении (т. е. $Z : R_4^{N\times N\times P(Ns)}$). Параллельный алгоритм просто вызывает последовательный алгоритм, таким образом, каждый P_{ID} вычисляет отдельный набор коэффициентов масштабирования. Такое распределение индексов для кодов P_{ID} достигает-

ся посредством операции **arg(Z(1,1:)).*loc***, которая возвращает индексы локальной части **Z**.

$$\mathcal{Z} : \mathbb{R}_+^{N\times N\times P(N_s)} = \textsc{ZoomImageParallel}(\mathbf{Z}_0 : \mathbb{R}_+^{N\times N}, \mathbf{s} : \mathbb{R}_+^{N_s}, \sigma)$$
$$1 \quad \mathcal{Z}.loc = \textsc{ZoomFrames}(\mathbf{Z}_0, \mathbf{s}(\ \textsc{find}(\mathcal{Z}(1,1,:)).loc\ ), \sigma)\,.$$

Далее, при разработке параллельной программы необходимо определить, каким образом доставить экземпляру P_{ID} коэффициенты масштабирования, необходимые ему для вычислений. В этой реализации представления все коэффициенты масштабирования и исходное изображение используются в вычислениях всеми экземплярами P_{ID}. В общем случае данный подход предпочтительнее вычисления *s* и **Zo** одним P_{ID} и отправки их другим экземплярам P_{ID}. Конкретные коэффициенты масштабирования выбираются путем выключения распределенного третьего измерения массива **Z**. Каждый P_{ID} затем вычисляет свои изображения ***Z.loc***. Весь массив *Z* затем может быть собран обратно в ведущий P_{ID}. Визуальное представление такой параллельной реализации для программы ZoomImage показано на рисунке 2.4.

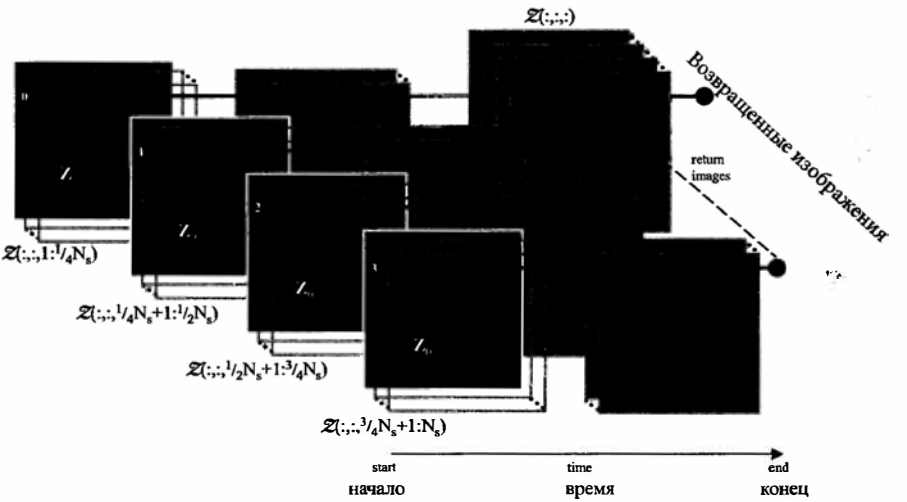


Рисунок 2.4. Параллельная реализация программы ZoomImage
Каждый P_{ID} выполняет операции на своей локальной части распределенного массива *Z*. После завершения операций локальные части посылаются ведущему P_{ID} (пунктирные линии). Затем ведущий P_{ID} собирает и отображает конечную последовательность изображений

2.2.3. Код

Следующим шагом является написание параллельной программы ZoomImage. Как уже было сказано в предыдущем подразделе, для реализации параллельного кода необходимо добавить к последовательной программе три дополнительные функциональные части.

- Каждый экземпляр MATLAB создает распределенный массив *Z*.
- Каждый экземпляр MATLAB определяет свои коэффициенты масштабирования *s* из значений своих локальных индексов распределенного третьего измерения *Z*.
- Ведущий экземпляр MATLAB собирает все части *Z*.

Ядро параллельного кода показано ниже в Листинге кода 2.2 (полный листинг расположен в директории `Examples/ZoomImage/pZoomImage.m`). Следует понимать, что один и тот же код будет выполняться каждым экземпляром MATLAB. Единственным различием между экземплярами является наличие у каждого из них уникального P_{ID} (т. е. `Pid`), который используется для определения того, какие части *Z* принадлежат каждому из экземпляров MATLAB.

Листинг кода 2.2. Параллельная MATLAB-программа ZoomImage

```
1 % Set image size, number frames, start and stop scale.
2 N = 128; Ns = 16; Sstart = 32; Send = 1;
3 sigma = 0.5;                              % Width of blur kernel.
4 PARALLEL = 1;                             % Set control flag.
5 Zmap = 1;                                 % Create serial map.
6 if (PARALLEL)
7 Zmap = map([1 1 Np],{},0:Np-1);           % Create parallel map.
8 end
9 Z = zeros(N,N,Ns.,Zmap);                  % Distributed array.
10 s = linspace(Sstart,Send,Ns);            % Compute scale factors.
11 disp('Zooming frames ...' );
12 tic;  % Start Clock.
13 ZO = referenceFrame(N,0.1,0.8);          % Create reference frame.
14 % Compute local frames.
15 Zloc = zoomFrames(ZO,S( global_ind(Z,3) ),sigma);
16 Tcompute = toc;                          % Stop Clock.
17 Z = put_local(Z,Zloc);                   % Insert into Z.
18 tic;                                     % Start Clock.
19 Zagg = agg(Z);                           % Aggregate on leader.
20 Tcomm = toc;                             % Start Clock.
```

Перевод

1 Установить размер изображения, количество кадров, начальный и конечный масштабы;

3 Ширина ядра размытия;

4 Установить контрольный флаг;

5 Создать последовательную карту размещения;

7 Создать параллельную карту размещения;

9 Распределенный массив;

10 Вычислить коэффициенты масштабирования;
12 Запустить часы;
13 Создать начальный кадр;
16 Остановить часы;
17 Вставить в Z;
18 Запустить часы;
19 Выполнить сбор данных на ведущем;
20 Запустить часы.

В **строках 4–8** создается параллельная карта размещения `Zmap`. Флаг `PARALLEL` является дополнительным инструментом отладки, позволяющим включать и выключать распределенные массивы, в результате чего *Z* становится обычным массивом.

Строка 9 создает распределенный массив **Z**.

В **строке 15** используется команда `global_ind`, предписывающая каждому экземпляру MATLAB вычислить индексы третьего измерения распределенного массива *Z*, соответствующие собственным локальным частям. Эти индексы затем используются для выбора коэффициентов масштабирования *S*, применяющихся при вычислении масштабированных изображений.

В **строке 17** каждый P_{ID} помещает свою локальную часть Zloc результата обратно в распределенный массив Z.

В **строке 19** результат собирается обратно в ведущий MATLAB (т. е. $P_{ID}=0$) с использованием функции `agg`.

Вышеуказанный код ZoomImage является ярким примером подхода к программированию на основе карт размещения. После того, как создана карта Zmap, она используется для создания распределенного массива *Z* с целью хранения результатов. Теперь каждый экземпляр MATLAB может использовать распределенный массив для определения индексов, которыми он владеет, с использованием функции `global_ind`. Эти индексы впоследствии используются для выбора коэффициентов масштабирования *S*, применяющихся при вычислении масштабированных изображений. Локальные результаты копируются обратно в распределенный массив с использованием функции `put_local`. Наконец, массив направляется «ведущему» с использованием функции `agg`.

На данном примере кода ZoomImage можно проследить некоторые свойства подхода к программированию на основе карт размещения.

- Незначительное влияние на код. Программа преобразована из последовательной в параллельную путем добавления около 10 дополнительных строк кода.
- Последовательный код сохраняется. При установке PARALLEL=0, *Z* становится обычным массивом, который может быть обработан последовательно без поддержки параллельных библиотек.
- Небольшой объем использования параллельной библиотеки. Потребовалось только четыре функции параллельной библиотеки: `map`, `global_ind`, `put_local` и `agg`.
- Масштабируемость. Код может выполняться на задачах различного размера, так как все массивы параметризованы размером *N*. Код также может выпол-

няться на любом количестве процессоров меньше *Ns*, так как все размеры локальных массивов получены из одного распределенного массива *Z*.

- Ограниченная коммуникация. Так как входные и выходные аргументы для функции `zoomFrames` являются локальными переменными, внутри этой функции не может произойти случайной коммуникации.
- Независимость от карт размещения. Программа будет работать с большим числом разнообразных карт при условии, что карта работает только на третьем измерении. Например, циклическое распределение в данном измерении тоже может использоваться.
- Гарантия производительности. Использование функцией `zoomFrames` локальных переменных гарантирует отсутствие каких-либо скрытых ухудшений производительности при параллельном выполнении данных строк кода.

2.2.4. Отладка

Как упоминалось ранее, параллельная отладка представляет собой непростую задачу. Основным преимуществом подхода на основе карт размещения является то, что оно поддается строго определенному процессу написания–отладки–тестирования. Соответствующие шаги указаны в таблице 2.2.

Шаг 1. Выполнение программы последовательно на локальном компьютере с массивом *Z*, в виде обычного массива. Этот шаг позволяет убедиться в том, что после добавления функций `global_ind` и `agg`, базовый последовательный код остается корректным.

Шаг 2. Выполняет программу последовательно на локальном компьютере с *Z* в качестве распределенного массива и позволяет проверить правильность работы функции `zoomFrames` с входными данными, полученными от распределенного массива.

Шаг 3. Выполнение программы параллельно на локальном компьютере. Данный шаг выясняет, является ли параллельная декомпозиция *Z* корректной и предоставляются ли правильные индексы для выбора коэффициентов масштабирования из *S*. Он также определяет ошибки взаимодействия, происходящие от операции `agg`. Из всех шагов этот, вероятнее всего, является самым важным и выявляет больше всего ошибок.

Шаг 4. Выполнение программы параллельно на удаленном компьютере. На данном шаге определяются ошибки в функции `agg`, связанные с использованием удаленной коммуникации; данные ошибки обычно возникают вследствие неверной установки программного обеспечения или ошибки конфигурации.

Шаг 5. Выполнение программы параллельно на удаленном компьютере, но с интересуемым размером задачи. До этого момента тестирование должно было осуществляться на задаче малого размера. Этот шаг позволяет определить производительность на реальной задаче.

Данные шаги являются существенным отражением этапов, указанных в разделе «Начало работы» для ZoomImage. На каждом из шагов тестируются определенные аспекты параллельного кода ZoomImage. Кроме того, в наибольшем объеме, тестирование производится локально на малых задачах, которые выполняются быстрее всего.

2.2.5. Тестирование

Тестирование параллельной программы ZoomImage – при условии, что она уже отлажена и выдает правильные ответы – в основном заключается в определении коли-

чества времени, которое требуется для ее выполнения и выяснения того, насколько ее производительность соответствует ожидаемой. Почти все временные показатели, необходимые для проведения такого анализа, были представлены в разделе «Начало работы» (см. таблицу 2.5). Производительности многоядерных и многоузловых систем аналогичны друг другу, и в обоих случаях наблюдается некоторое ускорение.

Таблица 2.5. Производительность программы ZoomImage на многоядерных и многоузловых системах

Относительное время вычислений, абсолютное время на запуск + коммуникацию, коэффициент ускорения вычислений на многоядерных и многоузловых вычислительных системах для различных значений *Np*. Данные о производительности приведены для фиксированного размера задачи (*N* и *Ns* константы). Все значения нормализованы по столбцу 2, в котором N = 1 с выключенными распределенными массивами (PARALLEL = 0). Столбцы 3–7 содержат значения с включенными распределенными массивами (PARALLEL = 1). Запись 1*8 означает что *Np* = 8, и все P_{ID} были выполнены на одном процессорном узле с общей памятью. Запись 16*1 означает что *Np* = 16, и каждый P_{ID} был выполнен на своем индивидуальном процессорном узле с использованием отдельной памяти

Многоядерные процессоры:	1	1*1	1*2	1*4	1*8	
Относительное время вычислений	1	1.0	0.86	0.63	0.34	
Время запуска + взаимодействия (с)	0.0	0.0	0.1	0.3	0.2	
Коэффициент ускорения вычислений	1	1.0	1.2	1.6	2.9	
Многоузловые процессоры:	1	1*1	2*1	4*1	8*1	16*1
Относительное время вычислений	1	1.0	0.85	0.56	0.32	0.17
Время запуска + взаимодействия (с)	0.0	0.0	0.16	0.17	3.4	6.5
Коэффициент ускорения вычислений	1	1.0	1.2	1.8	3.1	5.8

Целью параллельной программы является уменьшение времени, затрачиваемого программой на вычисления. Таблица 2.5 демонстрирует сокращение времени вычислений при увеличении *Np* как в многоядерной, так и в многоузловой системе. Однако в параллельной программе обычно увеличивается время, затрачиваемое на запуск и коммуникацию всех экземпляров MATLAB. В таблице 2.5 прослеживается увеличение времени запуска + коммуникации с увеличением *Np*, а также сравнительно больший временной показатель запуска + коммуникация в многоузловой системе, так как запуск и коммуникация осуществляются через сеть.

Программа имеет высокий коэффициент ускорения на вычислительной части кода. *[Примечание: проницательный читатель может посчитать, что коэффициент ускорения вычисления не такой высокий, как можно было надеяться (см. раздел 5.5)]* Поскольку накладные расходы увеличиваются с ростом *Np*, общая производительность программы ограничена. По большей части, это, побочный эффект того, что программа выполняется лишь в течение нескольких секунд. В большинстве реальных приложений стоит цель взять программу, выполняющуюся достаточно продолжительное время, и заставить ее работать быстрее. По мере того, как программа ZoomImage

станет выполняться за более длительный период времени (т. е. при выборе больших значений *N* и/или *Ns*), время вычислений станет намного превышать время запуска + коммуникации, и коэффициент ускорения вычислений будет отражать общую производительность параллельного приложения.

2.3. Программа ParallelIO

Предыдущие примеры служат иллюстрацией концептуального представления о параллельных программах с массовым параллелизмом, выполняющих большое количество независимых вычислений на различных процессорах. Важным элементом, отсутствующим в данных программах, является файловый ввод/вывод. Большинство приложений используют некоторые формы файлового ввода/вывода. В данном разделе на примере программы ParallelIO мы продемонстрируем, каким образом можно реализовать файловый ввод/вывод независимо от *Np*.

Программа ParallelIO создает матрицу, записывает ее на диск и затем считывает ее в оперативную память. На выходе получается количество времени, затрачиваемое на выполнение данной операции. Трудность в ParallelIO представляет создание программы, которая записывает файлы таким образом, что они могут быть считаны даже при изменении *Np*.

2.3.1. Начало работы

Чтобы начать выполнение программы, запустите MATLAB и перейдите в директорию Examples/IO (т. е. введите `cd/Examples/IO`). В командной строке наберите

```
pIO
```

В этом случае программа выполняется на вашем локальном процессоре и выводит подобные временные характеристики:

```
Write Time / время записи (sec/c)=          21.5991
Read Time / время считывания (sec/c)=       2.1681
```

Запишите данные временные характеристики для последующего анализа. Повторите процедуру для каждой из следующих команд:

```
eval(pRUN('pIO',1,{}))
eval(pRUN('pIO',2,{}))
eval(pRUN('pIO',4,{}))
```

Данные команды выполняют программу IO на одном, двух и четырех процессорах. При выполнении на параллельном компьютере выдаются значения временных параметров, близкие к указанным в таблице 2.6.

Полученные результаты будут изучены позже в разделе «Тестирование».

Важное примечание: последний аргумент команды `pRUN` определяет, на каком параллельном компьютере будет выполняться программа. Может возникнуть необхо-

димость в получении надлежащих установочных параметров у вашего системного администратора; для получения более подробной информации обращайтесь на веб-сайт книги (http://www.siam.org/KepnerBook). Установка последнего аргумента значением {} приводит к выполнению множества копий программы на вашем локальном компьютере и получению производительности, отличной от указанной в таблице 2.6.

2.3.2. Разработка параллельной программы

Основное назначение данной программы заключается в создании матрицы **X** размером $N \times M$ из случайных чисел, записи их в файлы и считывании их снова в оперативную память. Базовый алгоритм для данной операции показан в Алгоритме 2.5 (подробное описание в подписи).

Алгоритм 2.5. Последовательный алгоритм IO

Алгоритм записывает массив размера $N \times M$ в M файлов данных и затем считывает их обратно в массив. Это делается с использованием функций WriteMatrixSerial и ReadMatrixSerial, которые также показаны.

```
SerialIO(X : RN×M, file)
1 WriteMatrixSerial(X, file)
2 Y = ReadMatrixSerial(file)

WriteMatrixSerial(X: RN×M, file)
1 for j = 1: M
2    do
3            WRITE(X(:,j), file.j.dat)

Y : RN×M = ReadMatrixSerial(file)
1 for j = 1 : M
2    do
3            Y(:,j) = READ (file.j.dat)
```

Параллельная реализация программы IO выполнена с использованием распределенного массива. Каждый P_{ID} записывает только свою локальную часть массива на диск. Например, если $N = 1000$, $M = 32$ и $Np = 4$, каждый P_{ID} запишет восемь файлов на диск, и каждый файл будет содержать 1000–элементный вектор. Результаты данного отображения в параллельном алгоритме показаны с использованием P-записи в Алгоритме 2.6. Одним из преимуществ данного вида модели является запись файлов таким образом, что они могут считываться в оперативную память, даже если количество процессоров будет меняться. Это достигается путем создания M уникальных файлов вместо создания Np уникальных файлов. Визуальное представление модели алгоритма ParallelIO показано на рисунке 2.5.

Алгоритм 2.6. Параллельный алгоритм ParallelIO

Алгоритм записывает распределенный массив размера $N \times M$ в M файлов данных и затем считывает их обратно в распределенный массив. Это делается с использованием

функций WriteMatrixPARALLEL и ReadMatrixPARALLEL. В каждой из данных функций строка 1 получает глобальные индексы локальной части распределенного массива и записывает их в вектор $\mathbf{Lj}_{my}$. Строка 2 выполняет цикл FOR от 1 до количества элементов в $\mathbf{Lj}_{my}$, которое обозначается $\| \mathbf{Lj}_{my} \|$. Наконец строка 4 производит фактическую операцию с файлом с использованием глобального индекса для определения уникального глобального имени файла.

```
PARALLELIO(X : RN×P(M), file)
1 WRITEMATRIXPARALLEL(X, file)
2 Y = READMATRIXPARALLEL(file)
WRITEMATRIXPARALLEL(X : RN×P(M) , file)
1 jmy = FIND(X(1, :)).loc
2 for j = 1 : ||jmy||
3    do
4            WRITE(X.loc(:, j), file.jmy(j).dat)
Y : RN×P(M) = READMATRIXPARALLEL(file)
1 jmy = FIND(Y(1, :)).loc
2 for j = 1 : ||jmy||
3    do
4            Y'loc(:, j) = READ(file.jmy(j).dat)
```

2.3.3. Код

Следующим шагом является написание параллельной программы ParallelIO. Как уже было сказано в предыдущем подразделе, для параллельного кода необходимо добавить три дополнительные функциональные части.

- Каждый экземпляр MATLAB создает распределенный массив **X** с использованием карты размещения.
- Каждый экземпляр MATLAB определяет векторы, которые он запишет из распределенного массива **X**, и записывает эти векторы как уникальные файлы.
- Каждый экземпляр MATLAB определяет векторы, которые он считает с распределенного массива **Y**, и считывает соответствующие файлы.

Ядро параллельного кода показано ниже (полный листинг расположен в `Examples/ID/Pid.m`). Следует понимать, что один и тот же код будет выполняться каждым экземпляром MATLAB. Единственным различием между экземплярами является наличие у каждого из них уникального P_{ID} (т. е. `Pid`), который используется для определения того, какие части распределенного массива принадлежат каждому из экземпляров MATLAB (т. е. ***X.loc*** и ***Y.loc***)

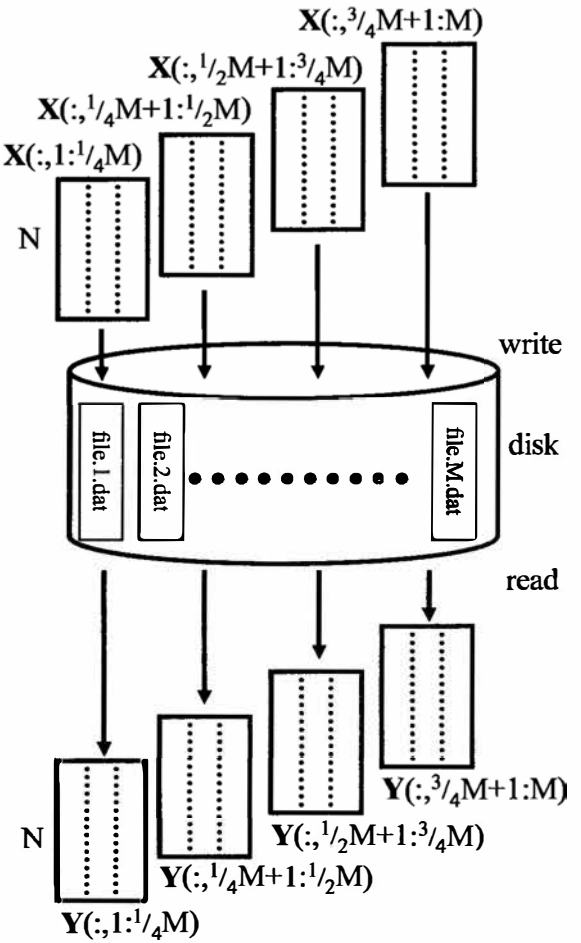


Рисунок 2.5. Реализация ParallelIO

Каждый P_{ID} записывает локальную часть распределенного массива **X** в набор файлов (один на вектор) и затем считывает данные в распределенный массив **Y**

Листинг кода 2.3. Параллельная MATLAB-программа ParallelIO

```
% Set the problem dimensions and filename.
N = 1024; M = 32; FILE './pIOvector';
PARALLEL = 1;     % Set control flag.
Xmap = 1; Ymap = 1;       % Create serial maps.
if (PARALLEL)      % Create parallel maps.
      Xmap = map([1 Np] ,{},0:Np-1); Ymap = Xmap;
end
```

```
8 Xrand = rand(N,M,Xmap); % Create distributed array.
9 Yrand = zeros(N,M,Ymap); % Create distributed array.
10 WriteMatrixParallel(Xrand,FILE);        % Write files.
11 Yrand = ReadMatrixParallel(Yrand,FILE); % Read files.
```

```
1 function WriteParallelMatrix(X,filebase)
2 myJ = global_ind(X,2);    % Get local indices.
3 Xlocal = local(X);        % Get local matrix.
4 for j=1:length(myJ)       % Loop over local indices.
5    Xj = Xlocal(: ,j);     % Get a vector.
6 filej = [filebase '. ' num2str(myJ(j)) ' .mat']; % Filename.
7 save(filej,'Xj'); % save vector to a file.
8 end
```

```
1 function Y = ReadMatrixParallel(X,file)
2 Y = X;     % Create output matrix.
3 myJ = global_ind(Y,2);    % Get local indices.
4 Ylocal = local(Y);        % Create a local matrix.
5 for j=1:length(myJ)       % Loop over local indices.
6    filej = [file'.' num2str(myJ(j)) ' .mat']; % Filename.
7    temp = load(filej, 'Xj');       % Read data.
8    Ylocal(: ,j) = temp.Xj;         % Insert column.
9 end
10 Y = put_local(Y,Ylocal);  Copy back to Y.
```

Перевод

1 Установить размеры задачи и имя файла.
3 Установить контрольный флаг.
4 Создать последовательную карту размещения.
5 Создать параллельную карту размещения.
8 Создать распределенный массив.
9 Создать распределенный массив.
10 Записать файлы.
11 Считать файлы.

2 Получить локальные индексы.
3 Получить локальную матрицу.
4 Цикл на локальных индексах.
5 Получить вектор.
6 Имя файла.
7 Сохранить вектор в файл.

2 Создать выходную матрицу.
3 Получить локальные индексы.
4 Создать локальную матрицу.
5 Цикл на локальных индексах.

6 Имя файла.
7 Прочитать данные.
8 Вставить столбец.
10 Копировать обратно в Y.

В основной программе

Строки 3–7 создают параллельные карты размещения `Xmap` и `Ymap`. Флаг `PARALLEL` является дополнительным инструментом отладки, позволяющим включать и выключать распределенные массивы **X** и **Y**.

Строки 8–9 создают распределенные матрицы **X** и **Y**.

Строки 10–11 записывают и считывают распределенную матрицу.

В программе WriteMatrixPARALLEL

Строки 2–3 предписывают каждому экземпляру MATLAB получить индексы `myJ` распределенной матрицы, соответствующие его локальной части. Эти индексы затем используются для создания имен файлов векторов, которые записываются.

Строки 4–8 записывают векторы из `Xloc` в отдельные файлы.

В программе ReadMatrixPARALLEL

Строки 3–4 предписывают каждому экземпляру MATLAB получить индексы `myJ` распределенной матрицы, соответствующие его локальной части. Эти индексы затем используются для выбора имен файлов векторов, которые считываются.

Строки 5–9 считывают локальные векторы матрицы из отдельных файлов в `Yloc`.

Строка 10 создает распределенный массив **Y** из локальных матриц `Yloc`.

Вышеуказанный код является ярким примером подхода к программированию на основе карт размещения. После того, как созданы карты `Xmap` и `Ymap`, они используются для создания распределенных матриц **X** и **Y**. Теперь каждый экземпляр MATLAB может использовать распределенный массив для определения индексов, которыми он владеет, с использованием функции `global_ind`. Эти индексы впоследствии используются для создания имен файлов, ассоциирующихся с локальными векторами, которые будет записывать и считывать каждый P_{ID}.

На данном примере кода можно проследить некоторые свойства метода программирования на основе карт размещения.

- Незначительное влияние на код. Программа преобразована из последовательной в параллельную путем добавления около 10 дополнительных строк кода.
- Последовательный код сохраняется. При установке PARALLEL=0 распределенные массивы **X** и **Y** становятся обычными массивами, которые могут быть обработаны последовательно без поддержки параллельных библиотек.
- Небольшой объем использования параллельной библиотеки. Потребовалось только четыре функции параллельной библиотеки: `map`, `put_local`, `local`, и `global_ind`.
- Масштабируемость. Код может выполняться на задаче любого размера и на любом количестве процессоров, если N > Np. Более важной особенностью является то, что количество процессоров, используемых для записи данных, может отличаться от количества процессоров, необходимых для считывания данных.

- Ограниченная коммуникация. Так как локальные переменные используются внутри функций `WritePARALLELMatrix` и `ReadPARALLELMatrix`, тем самым осуществляется строгий контроль над тем, когда осуществляется коммуникация.
- Независимость от карт размещения. В данном случае программа более зависима от карт, чем должна быть в идеале. Карта размещения и параллельные функции записи и считывания связаны и будут работать только для распределенных матриц, разбитых по второму измерению. Однако программа будет работать для любого распределения во втором измерении (т. е. блочное, циклическое, блочно-циклическое).
- Гарантия производительности. Так как команды `save` и `load` работают с локальными переменными, которые являются обычными числовыми массивами MATLAB, гарантируется отсутствие каких-либо скрытых ухудшений производительности при параллельном выполнении данных строк кода.

2.3.4. Отладка

Как упоминалось ранее, параллельная отладка представляет собой непростую задачу. Основным преимуществом подхода на основе карт размещения является то, что оно поддается четко определенному процессу написания программы–отладки–тестирования. Соответствующие шаги указаны в таблице 2.2.

Шаг 1. Выполнение программы последовательно на локальном компьютере с массивами **X** и **Y**, сконструированными как обычные массивы. Таким образом проверяется корректность базового последовательного кода после применения функций `global_ind` и `local`.

Шаг 2. Выполние программы последовательно на локальном компьютере с **X** и **Y** в качестве распределенных массивов и проверка правильности работы функций `WritePARALLELMatrix` и `ReadPARALLELMatrix` с распределенными массивами в качестве входных данных.

Шаг 3. Выполнение программы параллельно на локальном компьютере. Данный шаг определяет, является ли параллельная декомпозиция **X** и **Y** корректной и вычисляются ли правильные индексы для создания имен файлов для записи и считывания. Из всех шагов этот, вероятнее всего, является самым важным и обнаружит больше всего ошибок.

Шаг 4. Выполнение программы параллельно на удаленном компьютере. На данном шаге выявляются ошибки функций `save` и `load`, представленных удаленными файловыми операциями, в частности, если директория, которую нужно записать, невидима для всех процессоров.

Шаг 5. Выполнение программы параллельно на удаленном компьютере, но с задачей интересующего размера. До этого момента тестирование должно было осуществляться на задаче малого размера. Данный шаг позволяет определить производительность на реальной задаче.

Данные шаги являются существенным отражением этапов, указанных в разделе «Начало работы» ParallelIO. На каждом из шагов тестируются определенные аспекты параллельного кода. Кроме того, везде, где это возможно, тестирование производится локально на малых задачах, которые выполняются быстрее всего. Очень часто

программисты пытаются выполнить отладку на больших задачах, при которой процесс выявления ошибок занимает больше времени, хотя эти ошибки можно было бы обнаружить, затратив меньшее количество времени, посредством выполнения отладки на малых задачах.

2.3.5. Тестирование

Тестирование параллельной программы – при условии, что она уже отлажена и выдает правильные ответы – в основном заключается в определении количества времени, которое требуется для ее выполнения и выяснения, насколько ее производительность соответствует ожидаемой. Почти все временные показатели, необходимые для проведения такого анализа, были представлены в разделе «Начало работы» (см. таблицу 2.6).

Во-первых, хотелось бы заметить, что время выполнения разбито на две фазы: запись и считывание. Пока каждая их фаз выполняется исключительно в параллельном режиме, функциональность будет ограничиваться масштабируемостью файловой системы, на которую идет запись и с которой идет считывание. В данном случае, поскольку задача небольшая, файловая система способна объединять команды ввода/вывода из всех *Np*. Таким образом, коэффициенты ускорения записи и считывания возрастают с увеличением *Np* как в многоядерных, так и в многоузловых системах.

Таблица 2.6. Производительность программы ParallelIO на многоядерных и многоузловых системах

Относительное время записи и считывания на многоядерных и многоузловых системах для различных значений *Np*. Данные производительности приведены для фиксированного размера задачи (N и M – константы). Все значения нормализованы по столбцу 2, в котором $N_p = 1$ с выключенными распределенными массивами (PARALLEL = 0). Столбцы 3–7 содержат значения с включенными распределенными массивами (PARALLEL = 1). Запись 1*8 означает что *Np* = 8, и все экземпляры P_{ID} будут выполнены на одном процессорном узле при совместно используемой памяти. Запись 16*1 означает что *Np* = 16, и каждый P_{ID} был выполнен на своем индивидуальном процессорном узле с использованием отдельной памяти

Многоядерные процессоры	1	1*1	1*2	1*4	1*8	
Относительное время записи	1	1.0	0.45	0.17	0.09	
Относительное время считывания	1	1.0	0.67	0.20	0.15	
Коэффициент ускорения записи	1	1.0	2.2	5.9	10.7	
Коэффициент ускорения считывания	1	1.0	1.5	4.9	6.5	
Многоузловые процессоры	1	1*1	2*1	4*1	8*1	16*1
Относительное время записи	1	1.0	0.48	0.24	0.12	0.07
Относительное время считывания	1	1.0	0.46	0.23	0.11	0.06
Коэффициент ускорения записи	1	1.0	2.1	4.2	8.6	13.5
Коэффициент ускорения считывания	1	1.0	2.1	4.3	8.8	17.7

2.4. Почему это работает

В данной главе были кратко описаны некоторые примеры программ, а также проанализирована процедура их разработки, написания, отладки и тестирования. Теоретически, данные примеры дают хорошее представление о том, как выполнять параллельные программы MATLAB, некоторые знания о параллельном написании данных программ и способе оценки их надлежащей производительности. Высокий уровень производительности данных программ достигается за счет нескольких факторов.

- Данные программы выбраны из приложений, которые известны как имеющие хороший потенциал производительности (т. е. массово-параллельные)
- Реализации выполнены с целью минимизировать дополнительный расход времени путем максимизации локальности.
- Код был также написан с целью минимизации дополнительного расхода времени и количества новых строк кода.
- Отладка осуществляется инкрементно.
- Измерен коэффициент ускорения выполнения операций и проведено сравнение с ожидаемым результатом.

В остальных главах книги эти и другие вопросы будут рассмотрены более подробно с целью представить ключевые концепции, необходимые для быстрого и эффективного получения высококачественных параллельных программ, подобных описанным в настоящей главе.

Список литературы

1. *[Dean & Ghemawat 2004] Jeffrey Dean & Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.*

2. *[Mandelbrot 1982] Benoit Mandelbrot, The Fractal Geometry of Nature, 1982, New York: W.H. Freeman.*

3. *[Moler 1985] Cleve Moler is believed to have originated this term while at Intel Corporation in 1985.*

Глава 3
Взаимодействие с распределенными массивами

Резюме

Представлен пример многоэтапной программы с вводом/выводом с целью демонстрации способа разработки таких программ с использованием распределенных массивов. Программа создает имитированные данные от многоканального датчика и затем обрабатывает набор этих данных. На приведенном примере также показано, как нужно работать с распределенными массивами во время процесса отладки и тестирования. В частности, рассматриваются виды информации, которые можно извлечь из распределенных массивов при выполнении программы.

В настоящей главе изучается многоэтапное приложение с вводом/выводом. Оно строится на концепциях, изложенных в предыдущей главе, с целью создания более сложной программы с использованием распределенных массивов. Основной задачей является выполнение программы в четырехшаговом процессе отладки и интерактивное исследование распределенных массивов для выяснения того, что можно увидеть на каждом шаге. Данный пример иллюстрирует способ взаимодействия с распределенными массивами, а также освещает некоторые вопросы, связанные с многоэтапными параллельными программами. [Примечание: для практического выполнения примеров необходимо установить соответствующие программы и базовое параллельное программное обеспечение. Инструкции по установке можно найти на веб-сайте книги http://www.siam.org/KepnerBook).

Последующие разделы имеют следующую структуру. Каждый раздел открывается подразделом «Начало работы», в котором программа выполняется на различном количестве процессоров, и собираются временные показатели. В следующем подразделе представлены математический аспект задачи и основная реализация параллельной программы. В третьем подразделе разъясняется, каким образом была написана программа. В четвертом разделе программа выполняется снова. Здесь также представлены инструкции по исследованию и изучению распределенных массивов в интерактивной сессии MATLAB. В заключение данной главы следует краткое рассмотрение расширенных возможностей параллельно-конвейерных программ.

Программа Beamformer является иллюстрацией концепции многоэтапной параллельной программы, которая состоит из серии мелкозернистых тривиально массовых параллельных операций. Во-первых, программа Beamformer имитирует волновую картину, полученную массивом датчиков от двух точечных источников. Затем программа обрабатывает имитированные входные данные путем формирования лучей, фокусирующих энергию на определенных участках; эти данные записываются в файлы. В конце все данные собираются вместе, суммируются, и затем отображается результат.

3.1. Начало работы

Для выполнения программы запустите MATLAB и перейдите в директорию Examples/Beamformer (т. е. наберите `cd Examples/Beamformer`). В командной строке введите

```
pBeamformer
```

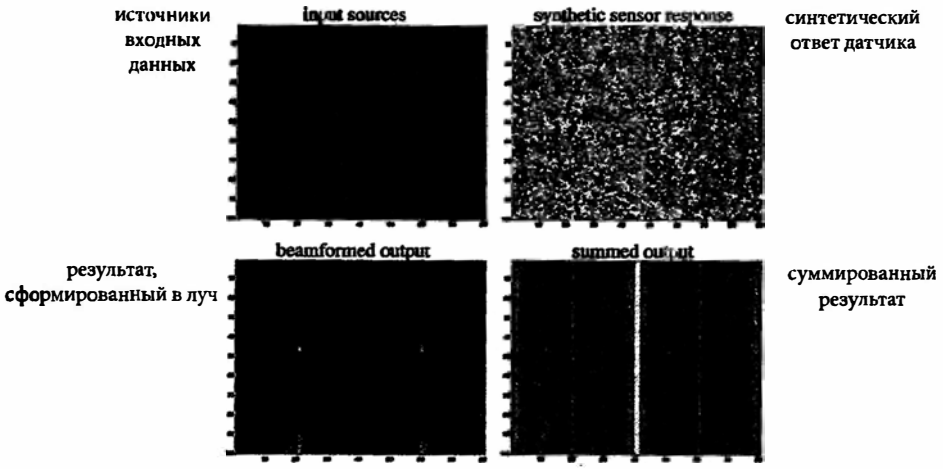


Рисунок 3.1. Результат программы Beamformer
Пример выходных данных программы pBeamformer

Это запустит программу на локальном процессоре и выдаст изображение (см. рис. 3.1), а также подобные временные показатели:

```
Compute Time/время вычисления (sec/c)                    =22.5518
Launch+Comm Time/время запуска+взаимодействия (sec/c)    =0.00576
```

Запишите эти временные характеристики для последующего анализа. Повторите процедуру для каждой из следующих команд:

```
eval(pRUN('pBeamformer' ,1,{}))
eval(pRUN('pBeamformer',2,{}))
eval(pRUN('pBeamformer' ,4,{}))
```

Данные команды выполняют программу Beamformer на одном, двух и четырех процессорах. При выполнении на параллельном компьютере выдаются значения временных параметров, аналогичные указанным в таблице 3.2.

Мы вернемся к этим результатам позже в ходе дальнейшего обсуждения программы и после получения более полного представления о том, что означают данные временные характеристики.

Важное примечание: последний аргумент команды `pRUN` определяет, на каком параллельном компьютере будет выполняться программа. Может возникнуть необходимость в получении надлежащих установочных параметров у вашего системного администратора; для получения более подробной информации обращайтесь на веб-сайт книги (http://www.siam.org/KepnerBook). Установка последнего аргумента значением {} приводит к выполнению множества копий программы на вашей локальной машине и получению производительности, отличной от указанной в таблице 3.2.

Рисунок 3.2. Физика программы Beamformer

Программа Beamformer имитирует ответ, полученный на линейных массивах датчиков, для двух источников волн

3.2. Разработка параллельной программы

Рассмотрим два точечных источника, испускающих волновую энергию (см. рис. 3.2). Энергия может быть акустической (например, звуковые волны в воздухе или воде) или электромагнитной (например, радиоволны). Первым шагом в программе Beamformer является вычисление суммарной энергии этих волн, полученных на линейном массиве датчиков, содержащем *Ns* элементов датчиков с набором частот *Nf* в различные моменты времени (снимки состояния) *Nt*. Расположение источников по отношению к элементам массива датчиков находится во входном тензоре

$$x_0 : \mathrm{R}^{N_t \times N_b \times N_f},$$

где *Nt* – количество «лучей» или углов, которые принимают получатели. Если источник 1 расположен под углом 45°, это может быть указано как

$$x_0(:,[N_b/4,:)=1.$$

Таким же образом, если источник 2 расположен под углом 90°, это может быть указано как

$$x_0(:,[N_b/2,:)=1.$$

Конечно, другие значения также могут меняться, если необходимо изменить положение источников между снимками состояния или частотная характеристика различных источников не одинакова. Для удобства в данном примере используются одинаковые источники.

Реакция датчиков на источники хранится в комплекснозначном тензоре $x_1 : \mathbf{C}^{Nt \times Ne \times Nf}$. Вычисление данной реакции осуществляется путем умножения входных данных на матрицу

$$x_1(i_t,:i_f) = v(:,:,i_f) * x_0(i_t,:,i_f)^T,$$

где $v : \mathbf{C}^{Ns \times Nb \times Nf}$ – частотно-зависимый тензор «вектора управления», который трансформирует волны от источников в реакцию на приемных элементах. Имитированная реакция завершается простым добавлением случайных помех к X_1.

Следующим этапом обработки является попытка восстановления позиций источников из зашумленных данных методом «формирования луча» (beamforming или пространственной фильтрации) и суммирования по различным частотам. Формирование луча выполняется по формуле

$$x_2(i_t,:i_f) = |x_1(i_t,:,i_f) * v(:,:,i_f)|^2,$$

где $x_2 : \mathbf{R}_+^{Nt \times Nb \times Nf}$. Финальный шаг суммирования выполняется следующим образом

$$x_3(i_t,i_b) = \sum_{if=1}^{Nf} x_2(i_t,i_b,i_f),$$

где $x_3 : \mathbf{R}_+^{Nt \times Nb}$ – матрица, показывающая угол каждого источника при каждом снимке. В ультразвуковых приложениях такая матрица носит название «график временной зависимости углов».

Базовый последовательный алгоритм формирования луча показан в Алгоритме 3.1 (подробное описание в подписи к алгоритму).

Точные детали вычислений нам не важны. Наиболее значимая информация заключается в следующих положениях.

- Вычисления производятся над трехмерными массивами с четырьмя основными измерениями: *Nt*, *Nb*, *Ns* и *Nf*.
- Алгоритм состоит из четырех этапов; каждый этап состоит из цикла `for` над тензором.

- Первые три цикла `for` совершаются над одинаковыми индексами цикла, из этого следует, что они могут использовать одну и ту же параллельную карту распределения.
- Заключительный цикл включает в себя суммирование над различными наборами индексов цикла; из этого следует, что для завершения операции необходимо будет обменяться этими данными.

Алгоритм 3.1. Последовательный алгоритм Beamformer

Алгоритм в качестве входных данных берет тензор источников X_0, тензор векторов управления $\mathcal{V}$, дисперсию шума σ и заголовок файла для сохранения результатов. Результатом является матрица X_3, которая восстанавливает угловое расположение источников как функцию времени. Алгоритм состоит из четырех различных циклов `for`. Первый создает имитированный отклик источников и комплексные случайные помехи ($n^{C}_{\mu=0,\sigma}$). Второй цикл осуществляет пространственную фильтрацию. Третий – записывает сформированные данные в набор файлов. Финальный цикл суммирует сформированные данные по частотам.

```
X3 : R+Nt×Nb= BeamFormerSerial(
            X0 : RNt×Nb×Nf, V : CNs ×Nb ×NJ,σ , file)
1 X1: CNtxNsxNf     X2: R +NtxNbxNf
2 for it = 1 : Nt, if = 1 : N f
3   do -> Stage 1: create sensor data.
4    X1(it,:,if)=V(:,:,if) * X0(it,:,if)T+n C=o,σ
5 for it = 1 : Nt, if = 1 : N f
6   do -> Stage 2: beamform sensor data.
7    X2(it,:,if) = |X1(it,:,if) * V(:,:,if)|2
8 for if = 1 : Nf
9   do -> Stage 3: write data to files.
10   WRITE(X2(:,:, if), file.if.dat)
11 for it = 1 : Nt, ib = 1 : Nb,
12  do -> Stage 4: sum along frequency dimension.
13   X3(it,ib) = ∑ ifNf = 1X2(it,ib,if)
```

Перевод
3 Этап 1: создать показания датчика.
6 Этап 2: выполнить пространственную фильтрацию показаний датчика.
9 Этап 3: записать данные в файлы.
12 Этап 4: суммировать вдоль измерения частоты.

Первый пункт означает, что задача содержит в себе достаточную степень параллелизма (т. е. есть вычисления, которые фактически могут быть выполнены в одно и то же время). В общем случае параллелизм в программе обнаружить не сложно. Второй пункт означает, что данный конкретный параллелизм имеет высокую степень локальности данных (т. е. данные для выполнения каждого параллельного процесса являются локальными для этого процессора). Локальность найти в программе сложнее, од-

нако она является ключевым фактором для получения хорошей параллельной производительности. Третий и четвертый пункты показывают, что локальность для первых трех этапов одинаковая, а для четвертого этапа она отличается. Эти свойства являются общими для многих типов вычислений и типичными для многоэтапных программ. Как правило, каждый этап программы распараллеливается независимо, и затем по необходимости осуществляется коммуникация между этапами или внутри них. В данном примере используется мелкозернистый подход с перебором параметра внутри каждого цикла, таким образом, каждый экземпляр P_{ID} отвечает за отдельную часть каждого массива.

В первую очередь при разработке параллельных программ необходимо решить, каким образом нужно распределять массивы между разными P_{ID}. Стандартный способ распределения для мелкозернистых тривиально массовых параллельных задач заключается в передаче каждой части области массивов отдельному набору P_{ID}. Например, если $Np = 4$, область можно было бы отобразить, как показано в таблице 3.1. Это отображение разбивает область в третьем измерении и передает различные части массива каждому экземпляру P_{ID}. В результате распределения получаем параллельный алгоритм, показанный с использованием P-записи в Алгоритме 3.2.

Во-вторых, при разработке параллельной программы необходимо определить, каким образом можно доставить каждому P_{ID} информацию, необходимую ему для вычислений. В данном примере используются распределенные массивы, что позволяет каждому экземпляру P_{ID}, задействованному в вычислениях, определить часть области, над которой он должен работать. Затем каждый P_{ID} выполняет вычисления на своей части результирующей матрицы и посылает данные назад ведущему P_{ID} (как правило, $P_{ID} = 0$). Ведущий P_{ID} получает все матрицы от других экземпляров P_{ID}, а затем суммирует и отображает результирующую матрицу. Визуальное представление такого метода разработки показано на рисунке 3.3.

Алгоритм 3.2. Параллельный алгоритм Beamformer

Параллельный алгоритм схож с последовательным алгоритмом с внесением некоторых изменений. Основное изменение заключается в том, что $X_{0,1,2}$ являются теперь распределенными массивами, декомпозированными в своем третьем измерении (строка 1). Соответствующие циклы `for` также меняются, таким образом, что индекс изменяется в пределах от $i_f = 1$ до $i_f = [\![i_f^{my}]\!]$, что является размером локальной части массива в третьем измерении (строки 3, 6 и 9). В конце финальный этап вычислений разбивается на два цикла `for`. Первый цикл `for` суммирует пространственно отфильтрованные данные по частотам на локальном P_{ID} (строки 12–14). Эти данные собираются вместе (строка 15) с использованием распределенного массива $\mathbf{x}_3$, и суммирование завершается (строки 16–18).

$\mathbf{X}_3 : \mathbb{R}_+^{N_t \times N_b} = \textsc{BeamformerParallel}($
$\mathcal{X}_0 : \mathbb{R}^{N_t \times N_b \times P(N_f)}, \mathcal{V} : \mathbb{C}^{N_s \times N_b \times P(N_f)}, \sigma, file)$

1 $\mathcal{X}_1 : \mathbb{C}^{N_t \times N_s \times P(N_f)} \quad \mathcal{X}_2 : \mathbb{R}_+^{N_t \times N_b \times P(N_f)} \quad \tilde{\mathcal{X}}_3 : \mathbb{R}_+^{N_t \times N_b \times P(N_P)}$
2 $i_f^{my} = \arg(\mathcal{X}_1(1,1,:)).loc$
3 **for** $i_t = 1 : N_t,\ i_f = 1 : \|i_f^{my}\|$
4 **do** ▷ Stage 1: create sensor data.
5 $\mathcal{X}_1.loc(i_t,:,i_f) = \mathcal{V}(:,:,i_f^{my}(i_f)) * \mathcal{X}_0.loc(i_t,:,i_f)^T + \mathbf{n}_{\mu=0,\sigma}^{\mathbf{C}}$
6 **for** $i_t = 1 : N_t,\ i_f = 1 : \|i_f^{my}\|$
7 **do** ▷ Stage 2: beamform sensor data.
8 $\mathcal{X}_2.loc(i_t,:,i_f) = |\mathcal{X}_1.loc(i_t,:,i_f) * \mathcal{V}(:,:,i_f^{my}(i_f))|^2$
9 **for** $i_f = 1 : \|i_f^{my}\|$
10 **do** ▷ Stage 3: write data to files.
11 $\textsc{Write}(\mathcal{X}_2(:,:,i_f), file.i_f^{my}(i_f).dat)$
12 **for** $i_t = 1 : N_t,\ i_b = 1 : N_b$
13 **do** ▷ Stage 4a: sum along local frequency dimension.
14 $\tilde{\mathcal{X}}_3.loc(i_t,i_b,1) = \sum_{i_f=1}^{\|i_f^{my}\|} \mathcal{X}_2(i_t,i_b,i_f)$
15 $\textsc{Agg}(\tilde{\mathcal{X}}_3)$
16 **for** $i_t = 1 : N_t,\ i_b = 1 : N_b$
17 **do** ▷ Stage 4b: sum aggregated data.
18 $\mathbf{X}_3(i_t,i_b) = \sum_{i_p=1}^{N_P} \tilde{\mathcal{X}}_3(i_t,i_b,i_P)$

Перевод
4 Этап 1: создать показатели датчика.
7 Этап 2: пространственно отфильтровать показатели датчика.
10 Этап 3: записать данные в файлы.
13 Этап 4а: суммировать вдоль локального измерения частоты.
17 Этап 4б: суммировать собранные данные.

Таблица 3.1. Параллельное распределение в программе Beamformer

Набор массивов распределяется на четыре экземпляра P_{ID} таким образом, что каждый экземпляр P_{ID} вычисляет отдельную часть массива

P_{ID}	Массив	Снимки	Лучи/датчик	Частоты
0	$x_{0,1,2}$ (	1: N_t,	1: $N_{b/s}$,	$1:\lceil 1/4 N_f \rceil$
1	$x_{0,1,2}$ (	1: N_t,	1: $N_{b/s}$,	$\left\lceil \frac{1}{4} N_f \right\rceil + 1 : \left\lceil \frac{1}{2} N_f \right\rceil$)
2	$x_{0,1,2}$ (	1: N_t,	1: $N_{b/s}$,	$\left\lceil \frac{1}{2} N_f \right\rceil + 1 : \left\lceil \frac{3}{4} N_f \right\rceil$)
3	$x_{0,1,2}$ (	1: N_t,	1: $N_{b/s}$,	$\left\lceil \frac{3}{4} N_f \right\rceil + 1 : N_f$)

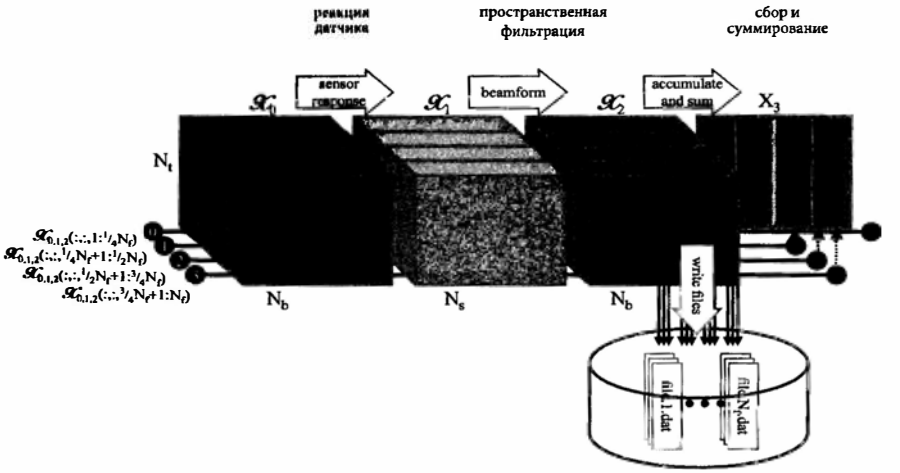


Рисунок 3.3. Параллельная реализация программы Beamformer
Каждый этап алгоритма взаимодействует прежде всего с двумя массивами: одним для входа и одним для выхода. Все распределенные массивы $x_{0,1,2}$ декомпозированы в измерении *Nf*, таким образом, не требуется никакого взаимодействия на стадиях ответа датчика, пространственной фильтрации и записи данных. Конечное состояние (аккумулирование и суммирование) требует, чтобы все результаты были отправлены обратно на $P_{ID} = 0$

3.3. Код

Следующим шагом станет написание параллельной программы. При стандартном (и рекомендуемом) подходе к написанию следует начать с корректной последовательной программы. Отладка параллельных программ может представлять некоторую сложность, и лучше всего начать с программы, которая уже работает и может быть проверена на корректность по мере добавления параллельных конструкций. Как уже было сказано в предыдущем разделе, для параллельного кода необходимы три дополнительные части функциональных возможностей.

- Каждый экземпляр MATLAB создает множество распределенных массивов с использованием карты распределения.
- Каждый экземпляр MATLAB определяет свою часть области из распределенного массива.
- Ведущий экземпляр MATLAB получает части и формирует конечный результат.

Листинг кода 3.1. Параллельная MATLAB–программа Beamformer

```
% Number of times, sensors, beams and frequencies.
Nt = 50; Ns = 90; Nb = 40; Nf = 100;
PARALLEL = 1;      % Set control flag.
Xmap = 1;          % Create serial map.
if (PARALLEL)
```

```
7  end
8  XO = zeros(Nt,Nb,Nf,Xmap); % Source array.
9  X1= complex(zeros(Nt,Ns,Nf,Xmap)); % Sensor input.
10 X2 = zeros(Nt,Nb,Nf,Xmap);      % Beamformed output.
11 X3 = zeros(Nt,Nb,Np,Xmap);      % Intermediate sum.
12 myI_f = global_ind(X1,3);       % Get local indices.
13 % Get local parts of arrays and insert two targets in XO.
14 XOloc = local(XO); X1loc = local(X1); X2loc = local(X2);
15 XOloc(: ,round(0.25*Nb),:)=1; XOloc(: ,round(0.5*Nb), :)=1;

16 for i_t=1: Nt    % Time snapshots loop.
17   for i_f=1: numel (my I_f )            % Local frequencies loop.
18          X1loc(i_t,: ,i_f) = ...        % Shift beams to sensors.
19          (squeeze(myV(:,: ,i_f))*squeeze(XOloc(i_t,: ,i_f)). ') ...
20          + sqrt(Ns)*complex(rand(Ns,1),rand(Ns,1));
21   end
22 end

23 for i_t=1: Nt    % Time snapshots loop.
24   for i_f=1 :numel(myI_f)               % Local frequencies loop.
25          X2loc(i_t,: ,i_f) = ...        % Sensors back to beams.
26          abs(squeeze(X1loc(i_t,:,i_f))*squeeze(myV(:, :,i_f))).^2;
27   end
28 end

29 for i_f=1:numel(myI_f)  % Loop over frequencies.
30   X_i_f = squeeze(X2loc(:,: ,i_f)); % Get a matrix of data.
31   filename=['dat/Beam_freq.' num2str(myI_f(i_f)) '.mat'];
32   save(filename,'X_i_f'); % save to a file.
33 end

34 X3 put_local(X3,sum(X2loc,3)); % Sum local into X3.
35 X3 squeeze(sum(agg(X3),3)); % Agg and complete sum.
```

Перевод

1 Количество снимков, датчиков, лучей и частот.

3 Установить контрольный флаг.

4 Создать последовательную карту размещения.

6 Создать параллельную карту размещения.

8 Массив источников.

9 Входные данные датчиков.

10 Пространственно отфильтрованный выход.

11 Промежуточные суммы.

12 Получить локальные индексы.

13 Получить локальные части массивов и вставить две цели в XO.

16 Цикл снимков состояния в различные моменты времени.

17 Цикл локальных частот.

18 Переместить лучи к датчикам.

23 Цикл снимков состояния в различные моменты времени.
24 Цикл локальных частот.
25 Датчики обратно к лучам.

29 Цикл по частотам.
30 Получить матрицу данных.
32 Сохранить в файл.

34 Суммировать локальные данные в X3.
35 Собрать и завершить суммирование.

Ключевые фрагменты параллельного кода показаны выше (полный листинг кода находится в директории `Examples/Beamformer/pBeamformer.m`). Следует иметь в виду, что один и тот же код будет выполняться каждым экземпляром MATLAB. Единственным различием между экземплярами является наличие у каждого из них уникального P_{ID} (т. е. `Pid`), который используется для определения того, какие части распределенного массива принадлежат каждому из экземпляров MATLAB.

Строки 3–7 создают параллельную карту размещения `Xmap`, которая используется для создания распределенных массивов. Флаг `PARALLEL` является дополнительным инструментом отладки, позволяющим включать и выключать распределенные массивы.

Строки 8–11 создают распределенные массивы `X1`, `X2`, `X3` и `X4`. Эти строки кода могут повторно использовать `Xmap`, потому что все распределенные массивы трехмерные и должны распределяться по третьему измерению.

Строка 12 предписывает каждому экземпляру MATLAB получить индексы распределенного массива, соответствующие его локальной части. Эти индексы используются для выбора частот и векторов управления, применяемых при вычислениях.

Строка 14 предписывает каждому экземпляру MATLAB получить свою локальную часть распределенных массивов.

Строки 16–22 создают набор данных имитированного отклика датчика. Цикл по частоте выполняется по локальным индексам каждого экземпляра MATLAB.

Строки 23–28 пространственно фильтруют данные имитированного датчика. Цикл по частоте выполняется по локальным индексам каждого экземпляра MATLAB.

Строки 29–33 записывают данные в файл, при этом каждый экземпляр MATLAB записывает один файл для каждой частоты, которая является локальной для этого экземпляра.

Строки 34–35 аккумулируют и суммируют данные.

Вышеуказанный код является ярким примером подхода к программированию на основе карт размещения. После того, как создана карта, она используется для создания распределенных массивов, предназначенных для хранения результатов. Теперь каждый экземпляр MATLAB может обращаться к распределенному массиву, используя функцию `global_ind` для определения индексов, которыми он владеет. Эти ин-

дексы впоследствии используются для выбора частот и векторов управления, применяемых при вычислении результатов. На первых нескольких этапах коммуникация не требуется, таким образом, используются только локальные части распределенных массивов. Локальные результаты копируются обратно в распределенный массив с использованием функции `put_local`. Наконец, массив собирается и суммируется «ведущим» с использованием функции `agg`.

На данном примере кода можно проследить некоторые свойства подхода к программированию на основе карт размещения.

- Незначительное влияние на код. Программа преобразована из последовательной в параллельную путем добавления около 10 дополнительных строк кода.
- Последовательный код сохраняется. При установке `PARALLEL=0` данный код может быть выполнен последовательно без поддержки параллельных библиотек.
- Небольшой объем использования параллельной библиотеки. Потребовалось только четыре функции параллельной библиотеки: `map`, `global_ind`, `local`, `put_local`, и `agg`, а также перегруженные функции-конструкторы `zeros()`, `complex()` и `rand()`. [Примечание: `rand` может выдавать значения в различном порядке в зависимости от параллельной карты размещения].
- Масштабируемость. Код может выполняться на задаче любого размера или любом числе *Np*.
- Ограниченная коммуникация. Так как используются локальные переменные, моменты осуществления коммуникации строго контролируются.
- Независимость от карт размещения. Программа будет работать с большим числом разнообразных карт.
- Гарантия производительности. Использование в основном алгоритме лишь локальных переменных, которые являются обычными числовыми массивами MATLAB, гарантирует отсутствие каких-либо скрытых ухудшений производительности при параллельном выполнении данных строк кода.

3.4. Интерактивная отладка и тестирование

В данном разделе программа Beamformer выполняется за пять определенных шагов. Шаг 1 выполняет программу последовательно на локальном компьютере с выключенными распределенными массивами. Этим проверяется правильность лежащего в основе последовательного кода. Шаг 2 выполняет программу последовательно на локальной машине с включенными распределенными массивами. На данном шаге проверяется правильность параллельного кода. Шаг 3 выполняет программу параллельно на локальном компьютере. На данном шаге выявляются ошибки коммуникации. Из всех шагов, этот, вероятнее всего, является самым важным и выявляет больше всего ошибок. Шаг 4 выполняет программу параллельно на удаленном компьютере. На данном этапе выявляются ошибки, внесенные при удаленном взаимодействии. Шаг 5 выполняет программу параллельно на удаленном компьютере, но на задаче интересующего нас размера. До этого момента тестирование должно было осуществляться на задаче малого размера. Данный шаг позволяет определить производительность на реальной задаче.

В дополнение к выполнению программы будут также предоставлены описания того, как можно взаимодействовать с некоторыми переменными. Чтобы выполнить программу, запустите MATLAB и перейдите в директорию Examples/Beamformer (т. е. наберите `cd Examples/Beamformer`).

3.4.1. Корректность последовательного кода

Чтобы перевести программу `Beamformer` в последовательный режим, отредактируйте файл `pBeamfomer.m` и установите `PARALLEL=0`, затем раскомментируйте последнюю строку файла:

```
whos X0 X1 X2 X3 X0loc X1loc      X2loc x3 myI_f        my V
```

Затем в командной строке MATLAB введите

```
pBeamfomer
```

При этом получите следующий вывод:

Имя	Размер	Байт	Класс	Атрибуты
X0	50x40x100	1600000	double	
X0loc	50x40x100	1600000	double	
X1	50x90x100	7200000	double	complex
X1loc	50x90x100	7200000	double	complex
X2	50x40x100	1600000	double	
X2loc	50x40x100	1600000	double	
X3	50x40	16000	double	
x3	50x40	16000	double	
myI_f	1x100	800	double	
my V	90x40x100	5760000	double	complex

Установка `PARALLEL=0` выключает параллельную библиотеку, превращая карты размещения в последовательные. Использование последовательной карты в качестве аргумента для конструктора MATLAB приводит к возвращению конструктором регулярного массива MATLAB. Фактически, это то же самое, что и выполнение программы без распределенных массивов. Это можно подтвердить, проверив, являются ли размеры и классы пар распределенных/локальных переменных `X0/X0loc`, `X1/X1loc`, `X2/X2loc` и `X3/x3` одинаковыми. Если они отличаются, это признак того, что массивы не объявлены правильно. Выполнение программы в таком режиме проверяет корректность последовательной программы.

3.4.2. Корректность параллельного кода

Чтобы вернуть программу Beamformer в параллельный режим, отредактируйте файл `pBeamfomer.m` и установите `PARALLEL=1`. Затем в командной строке MATLAB введите

```
pBeamfomer
```

Это аналогично записи

```
eval(pRUN('pBeamformer' ,1,{}))
```

При этом получите следующий вывод:

Имя	Размер	Байт	Класс	Атрибуты
XO	50x40xl00	1600928	dmat	
XOloc	50x40xl00	1600000	double	
Xl	50x90xl00	7200928	dmat	
Xiloc	50x90xl00	7200000	double	complex
X2	50x40xl00	1600928	dmat	
X2loc	50x40xl00	1600000	double	
X3	50x40	16920	dmat	
x3	50x40	16000	double	
myI_f	lxl00	800	double	
my V	90x40xl00	5760000	double	complex

Установка PARALLEL=1 включает параллельную библиотеку, создавая распределенные карты размещения. Использование распределенной карты в качестве аргумента для конструктора MATLAB обусловливает возвращение конструктором распределенного массива MATLAB. Это можно подтвердить, проверив размеры и классы пар распределенных/локальных переменных XO/XOloc, Xl/Xlloc, X2/X2loc и X3/x3. На данном этапе размеры массивов должны быть одинаковыми, но классы распределенных массивов должны иметь тип dmat. Если они не одинаковы, это признак того, что массивы не объявлены правильно. На данном этапе выполнение программы проверяет правильность объявления распределенных массивов и правильность работы перегруженных функций (в данном случае zeros, rand() и complex).

3.4.3. Корректность локальной коммуникации

Для выполнения программы с использованием четырех экземпляров MATLAB на локальной компьютерной системе в командной строке MATLAB введите

```
eval(pRUN('pBeamformer' ,4,{}))
```

При этом получится следующий вывод:

Имя	Размер	Байт	Класс	Атрибуты
XO	50x40xl00	400952	drnat	
XOloc	50x40x25	400000	double	
Xl	50x90xl00	1800952	drnat	
Xlloc	50x90x25	1800000	double	complex
X2	50x40xl00	400952	dmat	
X2loc	50x40x25	400000	double	
X3	50x40x4	16952	drnat	
x3	50x40	16000	double	
myl_f	1x25	200	double	
my V	90x40x25	1440000	double	complex

Установка последнего аргумента в команде `dRUN()` на 4 запускает программу на четырех экземплярах MATLAB на локальном процессоре. Это можно подтвердить, проверив размеры и классы пар распределенных/локальных переменных `XO/XOloc`, `Xl/Xlloc`, `X2/X2loc` и `X3/x3`. На данном этапе размеры массивов должны отличаться, а классы распределенных массивов должны иметь тип `dmat`. В частности, третье измерение локального массива должно быть равно 1/4 размера третьего измерения соответствующего распределенного массива. Если они отличаются в другой пропорции, это признак того, что массивы не объявлены правильно. Кроме того, выводы от других экземпляров MATLAB можно просмотреть, введя в командной строке MATLAB следующее:

```
!type MatMPI\*.out
```

Это аналогично записи

```
!more MatMPI\*.out
```

Размеры и типы классов распределенных и локальных массивов должны следовать тому же принципу.

На данном этапе программа выполняется параллельно на локальной машине. На данном шаге выявляются ошибки, связанные со взаимодействием. Из всех шагов этот, вероятнее всего, является самым важным и обнаруживает больше всего ошибок.

3.4.4. Корректность удаленной коммуникации

Для выполнения программы с использованием четырех экземпляров MATLAB на удаленной компьютерной системе в командной строке MATLAB введите

```
eval(pRun('pBlurimage',4,ParallelMachine)).
```

Последний аргумент команды `dRUN()` определяет, на каком параллельном вычислительном компьютере будет выполняться программа. (Может возникнуть необходимость в получении надлежащих установочных параметров у вашего системного администратора; для получения более подробной информации обращайтесь на веб-сайт книги (http://www.siam.org/KepnerBook)). Вывод, полученный на данном этапе,

должен быть идентичен выводу, полученному на предыдущем. Если выводы не одинаковы, это признак того, что способ выполнения программного обеспечения на удаленной машине отличается от его локального выполнения.

3.4.5. Оценка производительности

Шаг 5 выполняет программу параллельно на удаленном компьютере, но на задаче интересующего нас размера. До этого момента тестирование должно было осуществляться на задаче малого размера. Данный шаг позволяет определить производительность на реальной задаче. Чтобы получить временные характеристики требуется выполнить программу для диапазона значений *Np* (типичными значениями являются степени двойки).

Тестирование программы `Beamformer` – при условии, что она уже отлажена и выдает правильные ответы – в основном заключается в определении времени, которое требуется для ее выполнения, и выяснения, насколько ее производительность соответствует ожидаемой. Почти все временные показатели, необходимые для проведения такого анализа, были собраны в разделе «Начало работы».

Первое испытание проводится для сравнения производительности программ с использованием *Np* = 1 с выключенными (`PARALLEL=0`) распределенными массивами и включенными (`PARALLEL=1`) распределенными массивами. Эти данные содержатся в столбцах 2 и 3 таблицы 3.2. В данном случае различия значений производительности пренебрежимо малы; они показывают, что распределенные массивы не вызывают ухудшения производительности. Если использование распределенных массивов с *Np* = 1 значительно замедляет работу, необходимо уделить внимание этой проблеме. Очень трудно получить какое-либо преимущество при *Np* > 1, если при *Np* = 1 выполнение программы происходит медленнее, чем при исходном последовательном коде. Работа только на локальных частях распределенных массивов почти гарантирует, что производительность будет такой же, как при работе без использования распределенных массивов.

В таблице 3.2 показано относительное время вычислений (нормализовано по отношению к случаю *Np* = 1), время запуска + коммуникации и коэффициент ускорения вычислений. Эти данные были собраны с восьмипроцессорной многоядерной системы и 16-узловой системы. В многоядерных системах все процессоры используют одну память. В многоузловых системах каждый узел имеет свою память, и процессоры осуществляют коммуникацию через сеть.

В таблице 3.2 показано сокращение времени вычислений при увеличении *Np* как в многоядерной, так и в многоузловой системе. Однако в параллельной программе обычно появляется дополнительное время, затрачиваемое на запуск и коммуникацию между элементами MATLAB. В таблице 3.2 прослеживается увеличение времени запуска + коммуникации с увеличением *Np*, а также сравнительно больший временной показатель запуска + коммуникации в многоузловой системе, так как работа осуществляется через сеть.

Программа имеет высокий коэффициент ускорения на вычислительной части кода. Однако, так как дополнительный расход времени увеличивается с ростом *Np*, общая производительность программы ограничена. Это по большей части побочный эффект того, что программа выполняется в течение лишь нескольких секунд. В боль-

достаточно продолжительное время, и заставить ее работать быстрее. По мере того как программа Beamformer будет выполняться за более длительный период времени (т. е. при выборе больших значений *Nt*, *Ns*, *Nb* и *Nf*), время вычислений станет намного превышать время запуска + коммуникации, и коэффициент ускорения вычислений будет отражением общей производительности параллельного приложения.

Таблица 3.2. Производительность программы Beamformer на многоядерных и многоузловых системах

Относительное время вычислений, абсолютное время запуска + коммуникации, коэффициент ускорения вычислений на многоядерных и многоузловых вычислительных системах для различных значений *Np*. Данные о производительности приведены для фиксированного размера задачи (*Nt*, *Ns*, *Nb* и *Nf* константы). Все значения нормализованы по столбцу 2, в котором *N = 1* с выключенными распределенными массивами (PARALLEL=0). Столбцы 3–7 содержат значения с включенными распределенными массивами (PARALLEL=1). Запись 1*8 означает что *Np* = 8, и все экземпляры P_{ID} были выполнены на одном процессорном узле при совместно используемой памяти. Запись 16*1 означает что *Np* = 16, и каждый экземпляр P_{ID} был выполнен на своем индивидуальном процессорном узле с использованием отдельной памяти

Многоядерные процессоры:	1	1*1	1*2	1*4	1*8	
Относительное время вычислений	1	1.0	0.48	0.22	0.12	
Время запуска+взаимодействия (с)	0.0	0.0	0.5	0.5	1.2	
Коэффициент ускорения вычислений	1	1.0	2.1	4.4	8.0	
Многоузловые процессоры:	1	1*1	2*1	4*1	8*1	16*1
Относительное время вычислений	1	1.0	0.51	0.26	0.12	0.06
Время запуска+взаимодействия (с)	0.0	0.0	8.4	10.9	10.7	14.3
Коэффициент ускорения вычислений	1	1.0	2.0	3.9	8.1	15.7

3.5. Тема повышенной сложности: введение в параллельно-конвейерные приложения

В предыдущих разделах распределенные массивы использовались для того, чтобы каждый экземпляр P_{ID} мог выполнять операции на малой части целого массива. Данный тип параллелизма часто называется «параллелизм по данным», так как каждый P_{ID} выполняет одну и ту же строку кода, но на различных частях данных. Чтобы параллелизм данных работал, каждый экземпляр P_{ID} должен выполнять одну и ту же программу. Данная модель параллельного программирования носит название Single-Program Multiple-Data (SPMD) (одна программа, несколько потоков данных, или архитектура SPMD).

Приложение Beamformer также демонстрирует другой тип параллелизма, так как он характеризуется многошаговостью и возможностью распределить различные этапы по разным наборам P_{ID}. Такой тип параллелизма носит название «параллелизм по задачам». Чтобы использовать этот вид параллелизма на практике, необходимо внести некоторые изменения в приложение. В частности, вместо простой обработки одиночного массива входных данных он требует обработки длинной последовательности

независимых массивов данных, что встречается довольно часто. В данном контексте каждый этап можно сделать параллельно-конвейерным с обработкой массива данных и последующей отправкой его другим наборам P_{ID} (см. Алгоритм 3.3). Распределение для i-ого набора P_{ID} обозначается как $P_{\{i\}}()$. Конкретный пример приведен на рисунке 3.4, а распределение набора P_{ID} показано в таблице 3.3.

Таблица 3.3. Параллельно-конвейерное распределение в программе Beamformer

Обозначение	Множество P_{ID}	Карта
$P_{\{1\}()}$	1, 2, 3, 4	map ([1 1 4] , , 1:4)
$P_{\{2\}()}$	5, 6, 7, 8	map ([1 1 4] , , 5:8)
$P_{\{3\}()}$	9, 10, 11, 12	map ([1 1 4] , , 9:12)
$P_{\{4\}()}$	0	map ([1 1 4] , , 0)

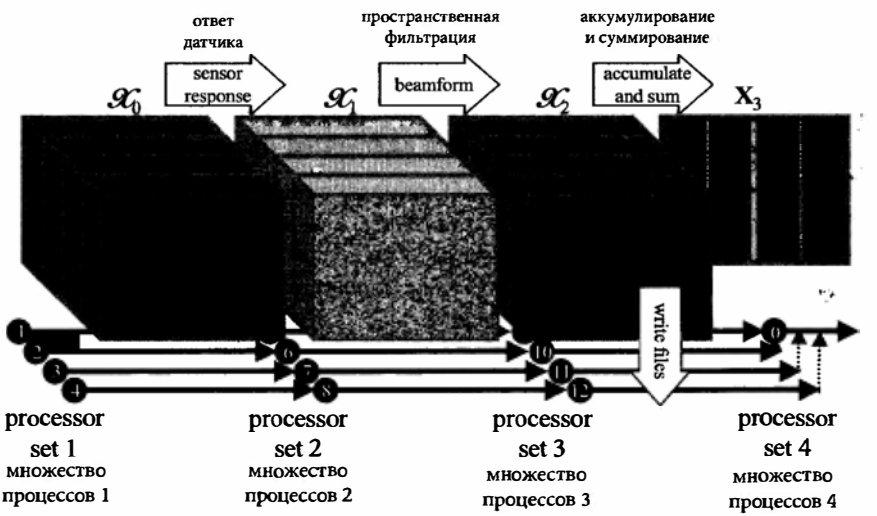


Рисунок 3.4. Параллельно-конвейерная реализация программы Beamformer
Каждый шаг алгоритма размещается на определенном множестве P_{ID}

Алгоритм 3.3. Параллельно-конвейерный алгоритм Beamformer

Схема данного алгоритма иллюстрирует изменения в параллельно-конвейерном алгоритме по сравнению с параллельным алгоритмом Beamformer. Каждый распределенный массив объявлен дважды (строки 1–3) с набором P_{ID} {1}, {2}, {3}, или {4}. Присваивание между двумя распределенными массивами вызывает коммуникацию между ними (строки 8 и 11).

$\mathbf{X}_3 : \mathbb{R}_+^{P_{\{4\}}(N_t \times N_b)} = \textsc{BeamformerPipeline}($
$\mathcal{X}_0 : \mathbb{R}^{N_t \times N_b \times P_{\{1\}}(N_f)}, \mathcal{V} : \mathbb{C}^{N_s \times N_b \times P_{\{1\}}(N_f)}, \sigma, file)$

1 $\tilde{\mathcal{X}}_1 : \mathbb{C}^{N_t \times N_s \times P_{\{1\}}(N_f)}$ $\quad \mathcal{X}_1 : \mathbb{C}^{N_t \times N_s \times P_{\{2\}}(N_f)}$
2 $\tilde{\mathcal{X}}_2 : \mathbb{R}_+^{N_t \times N_b \times P_{\{2\}}(N_f)}$ $\quad \mathcal{X}_2 : \mathbb{R}_+^{N_t \times N_b \times P_{\{3\}}(N_f)}$
3 $\tilde{\mathcal{X}}_3 : \mathbb{R}_+^{N_t \times N_b \times P_{\{3\}}(N_P)}$
4 **for** $i = 1 : \infty$
5 **do** ▷ Loop over sequences of input data.
6 **for** i_t, i_f
7 **do** ▷ Stage 1: create sensor data.
8 $\mathcal{X}_1 = \tilde{\mathcal{X}}_1$
9 **for** i_t, i_f
10 **do** ▷ Stage 2: beamform sensor data.
11 $\mathcal{X}_2 = \tilde{\mathcal{X}}_2$
12 **for** i_f
13 **do** ▷ Stage 3: write data to files.
14 **for** i_t, i_b
15 **do** ▷ Stage 4a: sum along local frequency dimension.
16 $\textsc{Agg}(\tilde{\mathcal{X}}_3)$
17 **for** i_t, i_b
18 **do** ▷ Stage 4b: sum aggregated data.

Перевод

5 Цикл по последовательностям входных данных.

7 Этап 1: создать показания датчиков.

10 Этап 2: пространственно отфильтровать показатели датчиков.

13 Этап 3: записать данные в файлы.

15 Этап 4а: суммировать вдоль локального измерения частоты.

18 Этап 4б: суммировать собранные данные.

Часть II

Методики повышенной сложности

Глава 4
Модели параллельного программирования

Резюме

На примере нетривиальной массовой параллельной программы (Blurimage) проиллюстрированы ключевые компромиссы проектирования, кодирования, отладки и тестирования, связанные с параллельным программированием. При рассмотрении программы Blurimage выявляется большой ряд вопросов в связи с выполнением данных этапов. Кроме того, в главе будет также представлено краткое введение в основные модели параллельного программирования: распределенные массивы, обмен сообщениями и схема начальник/подчиненный (manager/worker).

Данная глава нацелена на получение читателем более широкого представления о компромиссах проектирования, кодирования, отладки и тестирования кода, связанных с параллельным программированием. Основная задача главы – выявление широкого круга проблем, связанных с применением этих принципов. Кроме того, здесь также представлено краткое введение в основные модели параллельного программирования: семантику распределенных массивов, обмен сообщениями и схему начальник/подчиненный (manager/worker). Некоторые принципы, рассмотренные в данной главе, включают в себя следующие аспекты.

Проектирование: оценка необходимости распараллеливания, классы параллельных программ, модели параллельного программирования, машинные модели.

Кодирование: различные стили параллельного программирования, влияние кода на производительность и масштабируемость.

Отладка: особенности процесса параллельной отладки, ошибки, которые можно обнаружить на каждом шагу.

Тестирование: измерение и прогнозирование производительности (относительно параметров компьютера).

Программа, используемая в качестве основного примера в настоящей главе, является более сложной, чем программы, представленные в главе 2. Она предназначена для выявления некоторых ограничений, присущих различным моделям параллельного программирования.

4.1. Проектирование: оценка необходимости распараллеливания

Единственным наиболее важным решением при параллельном программировании является оценка необходимости применения параллельных вычислений. Существует две и только две причины использовать параллельный компьютер.

- Время: вычисление на одном процессоре занимает слишком много времени.
- Память: чтобы выполнить вычисление, требуется больше памяти, чем может предоставить один процессор.

Очень часто присутствуют обе причины. Выполнение программ, требующих наличия большого объема памяти, часто занимает большое количество времени. Однако каким образом программист MATLAB определяет, имеются ли данные условия? Понятие о количестве затрачиваемого времени очень субъективно. Как правило, это означает, что вычисление занимает часы или дни, но иногда и десять минут считается долгим сроком. «Память» понятие менее субъективное. Вычислительная машина обладает определенным объемом памяти, и если программа пытается задействовать объем памяти больше имеющегося, в этом случае либо выдается ошибка «недостаточно памяти», либо программа начинает существенно замедлять свою работу, так как операционная система пытается использовать локальные носители данных в качестве памяти (это явление носит название «свопинг» (swapping)). К счастью, MATLAB обладает некоторыми встроенными инструментами, которые помогают определить не только количество времени и памяти, необходимое для выполнения программы, но и ее конкретные части, требующие наибольшего количества времени и памяти. Функции данных инструментов будут наглядно показаны для начала в простой программе обработки изображений (Blurimage).

4.1.1. Профилирование производительности и памяти

Рассмотрим последовательное приложение, выполнение которого требует много времени или занимает объем памяти, превышающий объем, имеющийся у одного процессора. Каким образом программист определяет, где и как применить параллельную обработку для наиболее успешного решения данных вопросов? Первым шагом является профилирование приложения и выявление структур данных, которые занимают наибольший объем памяти, и операций, на которые затрачивается больше всего времени. В настоящей главе в качестве иллюстрации этого процесса используется пример программы Blurimage.

Для начала запустите MATLAB и наберите `cd Examples/Blurimage`. Для перевода программы `Beamformer` в последовательный режим отредактируйте файл `pBlurImage.m` и установите `PARALLEL=0`. В командной строке MATLAB введите `profview` для вызова профилировщика. В поле «Run this command» (выполнить эту команду) наберите `pBlurimage`. При этом выполняется профилирование MATLAB на этой программе. Через некоторое время выдается изображение (см. рис. 4.1). В окне профилировщика щелкните кнопкой мыши на строке `dBeamformer`, после чего откроются данные профиля, подобные изображенным на рис. 4.2.

Рисунок 4.1. Результат программы Blurimage

Профилировщик предоставляет подробную информацию о том, как долго выполнялась каждая строка программы. Для получения информации о занимаемой памяти наберите `whos`, в результате чего получится следующее:

```
...
N_x         1x1          8          double
N_y         1x1          8          double
Nk_x        1x1          8          double
Nk_y        1x1          8          double
Z           1024x1024    8388608    double
Zloc        1024x1024    8388608    double
Zmap        1x1          8          double
kernel      32x32        8192       double
my I        1x1024       8192       double
myJ         1x1024       8192       double
n_blur      1x1          8          double
...
```

Как видно из рисунка 4.2, одна функция (`conv2`) требует больше всего времени, а отдельная переменная (Z) занимает больше всего памяти, что соответствует ожиданиям. Приложение Blurimage представляет собой простую программу фильтрации изображений, которая многократно применяет к изображению свертку с помощью фильтра (или ядра). Это одна из наиболее общих операций по обработке изображений, и математически она представлена формулой двойного суммирования

$$Z_S^{blur}(i,j) = \sum_{i'=1}^{N_K} \sum_{j'=1}^{N_K} K_S(i',j')Z_0(i-i',j-j'),$$

где i и j - это пиксельные индексы, Z и Z^{blur} - входное и выходное изображения, K – ядро фильтра размеров N_K x N_K. Вышеуказанное прямое суммирование включает в себя $2N_K^2$ операций на пиксель (2 в связи с тем, что есть операция умножения и сложения). Последовательный алгоритм для данной программы приведен в Алгоритме 4.1.

Алгоритм 4.1. Последовательный алгоритм Blurimage

Алгоритм в качестве входных данных принимает исходное изображение Z размером $N_x \times N_y$, ядро K размером $N_K \times N_K$ и N_{blur} – количество операций размытия изображения. Результатом (выходными данными) является размытое изображение Z размером $N_x \times N_y$.

$$\mathbf{Z} : \mathbb{R}_+^{N_x \times N_y} = \textsc{Blurimage}(\mathbf{Z} : \mathbb{R}_+^{N_x \times N_y}, \mathbf{K} : \mathbb{R}_+^{N_K \times N_K}, N_{blur})$$

1 **for** $i_{blur} = 1 : N_{blur}$
2 **do**
3 $\mathbf{Z} = \mathbf{K} \circ \mathbf{Z}$

Начать профилирование	Выполнить этот код (Run this code)	dBlurimage		Время профилирования: 26 с	
Строки, на которые потрачено больше всего времени					
№ строки	Код	Вызовы	Общее время	% времени	График временной зависимости
51	Zloc(1:end-Nk_x+1,1:end-Nk_y+1...	2	20.811 s	80.9%	
75	figure	1	2.561 s	10.0%	
76	pcolor(transpose(local(z)));...	1	1.641 s	6.4%	
31	zloc = (myI,'*ones(1,length(...	1	0.317 s	1.2%	
34	z = darray(zloc,z);	1	0.087 s	0.3%	
Все другие строки			0.309 с	1.2%	
Итоговые данные			25.726 с	100%	

Начать профилирование	Выполнить этот код (Run this code)	dBlurimage	Время профипирования: 26 с
< 0.01	1 **48** for i_blur = 1:n_blur % Loop over each blur.		
0.02	2 **49** %loc = local (z); % Get local data.		
	50 % Perform covolution.		
20.81	2 **51** Zloc (1; end - Nk_x+1, 1: end-Nk_y+1) = conv2(zloc,icernel,'valid');		
< 0.01	2 **52** Z = darray (zloc,Z0; % Put local back in global.		
< 0.01	2 **53** Z = synch(Z); % Copy overlaping boundaries.		
< 0.01	2 **54** if (CHECK)		
	55 im(1:end-Nk_x+1, 1:end-Nk_y+1) = conv2 (im,kernel,'valid');		
	56 end		
	57		
< 0.01	2 **58** end		

Перевод
48 Цикл по каждому размытию.
59 Получить локальные данные.
50 Выполнить свертку.
52 Поместить локальное обратно в глобальное.
53 Копировать перекрывающиеся границы.

Рисунок 4.2. Результаты профилировщика для программы Blurimage
Профилировщик записывает общее время и количество вызовов каждой функции или строки кода. Данный профиль указывает, что на функцию `conv2` затрачивается наибольшее количество времени

Свертка обходит каждую точку (или пиксель) изображения, умножает все другие точки вокруг пикселя на значения в фильтре и затем суммирует их вместе для получения нового значения пикселя. Данный тип программы хорошо подходит для распараллеливания, так как все время затрачивается на одну функцию, и всю память занимает единая структура данных. Во многих отношениях программа Blurimage подобна программе Mandelbrot, представленной в главе 2, однако, между ними есть одно небольшое, но весьма существенное отличие. В программе Mandelbrot каждую точку изображения можно вычислить независимо от других. В Blurimage каждая точка зависит от окружающих ее значений. Это тонкое различие будет иметь значительное влияние на метод распараллеливания программы. В сущности, многие из существующих программ характеризуются таким типом нелокальной зависимости.

Итак, каким образом лучше использовать параллельный компьютер в данной ситуации? Этот вопрос заставляет нас обратиться к другому очень важному принципу: лучшая параллельная программа – это хорошая последовательная программа. Параллельные вычисления привносят неизбежные сложности в любую программу. Поэтому сначала лучше попытаться решить вопросы производительности с помощью других средств, если это возможно. Таким образом, первым шагом является оптимизация

- Время
  - Могут ли структуры данных размещаться заблаговременно, а не динамически?
  - Можно ли использовать векторизованные выражения вместо циклов `for`?
  - Существуют ли очень длинные циклы `for`, которые можно заменить на более короткие циклы `for`?
  - Можно ли написать и связать подпрограмму на языке C, C++ или Fortran для более быстрого выполнения ключевых функций?

- Память
  - Присутствует ли излишнее дублирование переменных?
  - Можно ли удалить переменные после их использования?
  - Можно ли использовать массивы вместо структур или массивов ячеек?
  - Можно ли использовать меньший тип данных (например, int8 вместо double)?
  - Может ли задача решаться на одной части в каждый момент времени, а промежуточные результаты храниться на диске?

Это лишь некоторые рекомендации по оптимизации. За более подробной информацией по оптимизации программ MATLAB обращайтесь к [McGarrity 2007]. Настоящая книга написана о параллельных вычислениях, поэтому допустим, что начальной точкой является хорошо оптимизированная последовательная программа, способная преодолеть ограничения по времени и памяти только посредством использования параллельных вычислений. В примере с программой Blurimage большая часть вычислений используется единственной функцией MATLAB (`conv2`), а наибольшая часть памяти расходуется на единый массив данных (`image`), оставляя лишь небольшую возможность дополнительной оптимизации последовательной программы.

4.1.2. Шаблоны параллельного программирования

Если принято решение, что использование параллельных вычислений является оптимальным способом преодоления ограничений времени и/или памяти, возникает следующий вопрос – каким образом распараллеливать программу. При выборе параллельной реализации лучше следовать классическому инженерному принципу: лучшим способом решения задачи является заблаговременное знание ответа. Точнее говоря, существует множество классических задач параллельных вычислений с известными решениями (некоторые из них были представлены в главе 2). В связи с этим рекомендуется сначала определить, какая из этих классических задач наиболее похожа на имеющуюся задачу, и следовать уже известному методу решения. Такой подход иногда называется программированием на основе шаблонов (см. [Mattson et al. 2004]). В нашей книге приводится много подобных примеров, а в таблице 4.1 дается краткая классификация и систематизация. Однако выбор необходимого шаблона является только первым шагом в процессе проектирования. Цель данной книги – представить суть всего процесса создания приложения с высокой производительностью.

Скорее всего, данная таблица не очень полезна на данном этапе, так как читатель еще не знаком с этими программами. Не стоит беспокоиться – по ходу книги программы будут описаны более подробно. Некоторые из них были уже представлены (см. примеры в главе 2) и являются одними из самых важных. Если ваше приложение представлено одним из примеров в главе 2, вам уже известно, что необходимо

превратить в параллельную программу. Если после прочтения книги вы обнаружили, что ваша задача не подходит ни под один из примеров, рекомендуется обсудить ее со специалистом в области параллельного программирования. Наконец, может потребоваться выработать ваше собственное решение задачи, и в этом случае вы читаете правильную книгу. В таблице 4.1 вводятся два новых термина, которые еще не рассматривались. Первый – это «модель коммуникации», необходимая программе. Второй – «модель параллельного программирования».

Таблица 4.1. Классификация параллельных приложений

Перечень параллельных программ, описываемых в данной книге (более подробное описание в главе 7). Имя – это общее наименование приложения или теста производительности. Описание представляет собой краткое описание того, что пытается сделать программа. Коммуникация – тип коммуникации, необходимый для осуществления программы параллельно. Модель программирования – «лучшая» модель программирования для данной задачи

Имя (глава)	Описание	Коммуникация	Модель программирования
AddOne (1)	Перебор значений параметра	распределение/сбор (scatter/gather)	Массив или начальник/подчиненный
Mandelbrot (2)	Перебор значений параметра	распределение/сбор	Массив или начальник/подчиненный
ParallelIO (2)	Парал. файловый ввод/вывод	нет	Любой
Zoom1mage (2)	Масштвбирование изображения	распределение/сбор	Массив или начальник/подчиненный
Beamformer (3)	Обработка поквзаний датчика	пареллально-конвейерный	
Blurimage (4)	Фильтрация большого изображения	Ближайшая соседняя точка (nearest neighbour)	
Пакет тестов производительности: HPC Challenge			
STREAM (8)	Складывает большие векторы	Нет	Любой
RandomAccess (9)	Обновл. большие векторы	Каждый с каждым (All-to-all)	Массив и сообщения
1D FFT (10)	БПФ большого вектора	Каждый с каждым	
HPL(11)	Разложение LU – матрицы	Каждый с каждым	Массив и сообщения

Шаблоны параллельной коммуникации

Классификация по шаблону коммуникации является стандартным способом классификации программ, применяемым экспертами в параллельном программировании. Четыре наиболее распространенных шаблона представлены на рисунке 4.3. Первый шаблон: «распределение/сбор» (scatter/gather) был использован во многих примерах в главе 2. В этом шаблоне один экземпляр P_{ID} распределяет данные для других экземпляров и ждет от них ответов. Этот шаблон может реализовываться любой из моделей параллельного программирования. В большинстве случаев модель начальник/подчиненный ограничивается только этим шаблоном взаимодействия. Следующий шаблон – параллельно-конвейерный, где каждый экземпляр P_{ID} получает данные, обрабатывает их и затем посылает их другому P_{ID}. Такой параллельно-конвейерный шаблон обычно используется во многих приложениях обработки показаний датчиков. Шаблон ближайшей соседней точки организует экземпляры P_{ID} в виртуальную сетку;

каждый P_{ID} взаимодействует только со своими ближайшими соседями. Blurimage попадает под этот шаблон. Наконец, шаблон «каждый каждому» предполагает, что любой P_{ID} может пытаться взаимодействовать с любым другим P_{ID}.

Большая часть примеров из главы 2 попадает под шаблон «распределение/сбор» (scatter/gather). Возможно, это самый распространенный шаблон. Преимуществом его применения является сравнительная простота реализации с использованием любых моделей программирования. В общем случае модель программирования на основе распределенных массивов будет использоваться для реализации данных типов приложений, так как это самый простой вариант (т. е. требуется наименьшее количество изменений в последовательном коде).

Blurimage имеет шаблон взаимодействия с соседями, который легче всего реализовать с использованием модели программирования на основе распределенных массивов. Три модели программирования, рассматриваемые в данной книге, – это распределенные массивы, обмен сообщениями и схема начальник/подчиненный. В определенных программах с шаблоном «каждый каждому» рекомендуется использовать обмен сообщениями, либо комбинированный подход на основе распределенных массивов и обмена сообщениями. Подробнее этот вопрос рассматривается в главах 9 и 11.

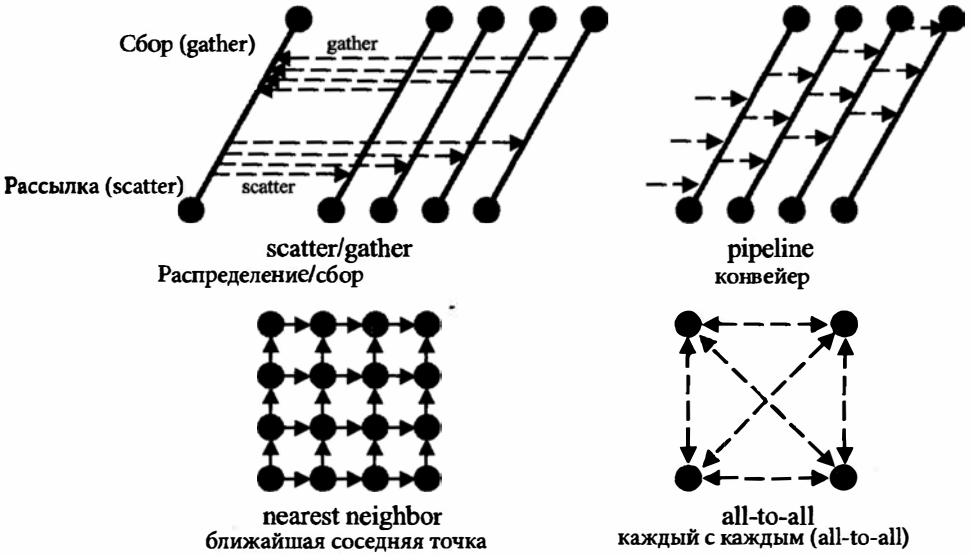


Рисунок 4.3. Стандартные шаблоны параллельного взаимодействия
В шаблоне распределение/сбор один экземпляр P_{ID} разбивает данные для других экземпляров P_{ID} и ждет от них ответов. При конвейеризации каждый экземпляр P_{ID} получает данные, обрабатывает их и затем посылает другому P_{ID}. Шаблон ближайшей соседней точки организует экземпляры P_{ID} в виртуальную сетку; каждый P_{ID} взаимодействует только со своими ближайшими соседями. Шаблон «каждый с каждым» предполагает, что любой экземпляр P_{ID} может пытаться взаимодействовать с любым другим P_{ID}

Модели параллельного программирования

Модель «начальник/подчиненный» является самой простой моделью параллельного программирования. Она используется, когда задачу можно разбить на множество совершенно независимых задач, которые подчиненный (клиенты) могут обработать без взаимодействия друг с другом. Основное ограничение данной модели начальник/подчиненный заключается в том, что каждый подчиненный взаимодействует только с ведущим (сервером), и ему не требуется знать, что делают другие исполнители. Это ограничение очень мощное и значительно упрощает работу. Кроме того, оно очень надежное в использовании, и многие сетевые программы следуют такой модели.

Модель обмена сообщениями во многих аспектах является противоположностью модели «начальник/подчиненный». Модель обмена сообщениями подразумевает, что любой экземпляр P_{ID} может посылать и получать сообщения от другого P_{ID}. Инфраструктура модели довольно проста. Эта инфраструктура в кругах специалистов по параллельным вычислениям, как правило, реализуется посредством интерфейса передачи сообщений (MPI) [Gropp at al. 1999]. Модель обмена сообщениями требует, чтобы каждый процессор обладал уникальным идентификатором (P_{ID}) и знал, сколько других процессоров (Np) работает вместе над задачей (по терминологии MPI (P_{ID}) и (Np) носят название «ранг» процессора и «размер» группы «мир» (world) MPI). Любая параллельная программа может быть реализована с использованием модели обмена сообщениями. Основной недостаток этой модели заключается в том, что программист должен управлять каждым индивидуальным сообщением в системе; таким образом, модель требует написания большого дополнительного кода, и ее отладка может быть очень затруднительна. Тем не менее существуют определенные параллельные программы, которые можно реализовать только при помощи модели обмена сообщениями.

Модель на основе распределенных массивов представляет собой компромисс между двумя вышеуказанными моделями. Распределенные массивы накладывают дополнительные ограничения на программу, что позволяет писать сложные программы относительно просто. Во многих отношениях эта модель параллельного программирования является самой естественной для MATLAB, так как она реализуется при помощи массивов, являющихся здесь основным типом данных. В нескольких словах, модель на основе распределенных массивов создает распределенные массивы, в которых каждый экземпляр P_{ID} хранит часть целого массива или владеет ей. Дополнительная информация хранится в массиве, таким образом, каждый P_{ID} знает, какими частями массива обладают другие экземпляры P_{ID}. Порядок разбиения массивов между экземплярами P_{ID} определятся картой размещения [Lebak et al. 2005]. Например, на рисунке 4.4 показана матрица, разбитая на строки; столбцы; строки и столбцы; столбцы с некоторым наложением. Знание способов размещения на основе карт может оказаться полезным, даже если модель распределенных массивов не используется. Концепция разбивания массивов различными способами является одной из ключевых идей параллельных вычислений.

Вычисления на распределенных матрицах обычно выполняются согласно правилу «вычисляет владелец», которое означает, что каждый экземпляр P_{ID} отвечает за выполнение вычислений на данных, которые он хранит локально. Карты размещения могут стать довольно сложными и фактически выражать произвольное преобразова-

ние. Одним из примеров сложности является случай, когда граница массива требуется более, чем одному P_{ID}. В примере с программой Blurimage имеется данное условие, так как свертка нуждается в данных вокруг каждой точки.

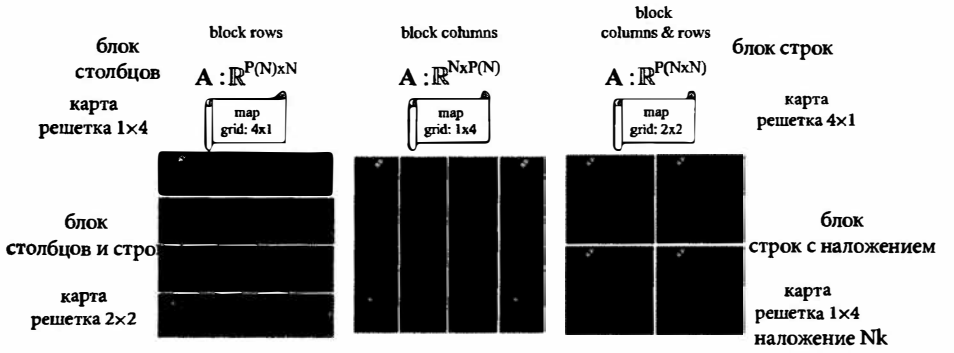


Рисунок 4.4. Распределение массивов по картам распределения
Различные способы параллельного распределения двумерных массивов. Массивы могут быть разбиты в любом измерении. Блочное распределение означает, что каждый P_{ID} содержит непрерывную часть массива. Наложение позволяет границам массива храниться в двух соседних P_{ID}

4.1.3. Параллельная реализация программы Blurimage

Проще всего понять принцип функционирования этих моделей на конкретных примерах. В данном разделе мы обсудим пример реализации программы Blurimage с использованием всех трех моделей программирования. В этом случае читатель сможет получить некоторое понимание, например, компромиссов, использованных при реализации каждой модели. Различные виды реализации параллельной программы Blurimage показаны на рисунке 4.5

В реализации по схеме «начальник/подчиненный» начальник посылает часть изображения вместе с границей каждому P_{ID}. Исполнители осуществляют свертку и посылают обратно ответ. Этот процесс разбиения и сборки изображения повторяется для каждой итерации свертки. При реализации данной модели существуют две основные трудности. Во-первых, если причиной выполнения программы в параллельном режиме является чрезмерный размер изображения (что случается чаще всего), модель «начальник/подчиненный» не сработает, потому что у ведущего будут проблемы с хранением всего изображения. Во-вторых, многократные разбиения и сборка могут занять большое количество времени, и необходимость в них пропадает. При каждой итерации требуется произвести обмен $N_x N_y + N_K N_y N_p$ пикселей или $8+8 N_K N_p / N_x$ байт на каждый пиксель (при условии 8-байтных пикселей). Термин «отношение вычисления к коммуникации» используется для измерения важной характеристики размера вычислительной интенсивности параллельной программы. Чем выше данное отношение, тем быстрее программа будет выполняться на параллельном компьютере. В данном случае имеется следующее отношение в операциях на байт:

$$\text{вычисление/коммуникация} = 2N_K^2 /(8+8N_K Np / Nx) \approx \frac{1}{4} N_K^2 ;$$

для $N_K = 32$ получается отношение 256.

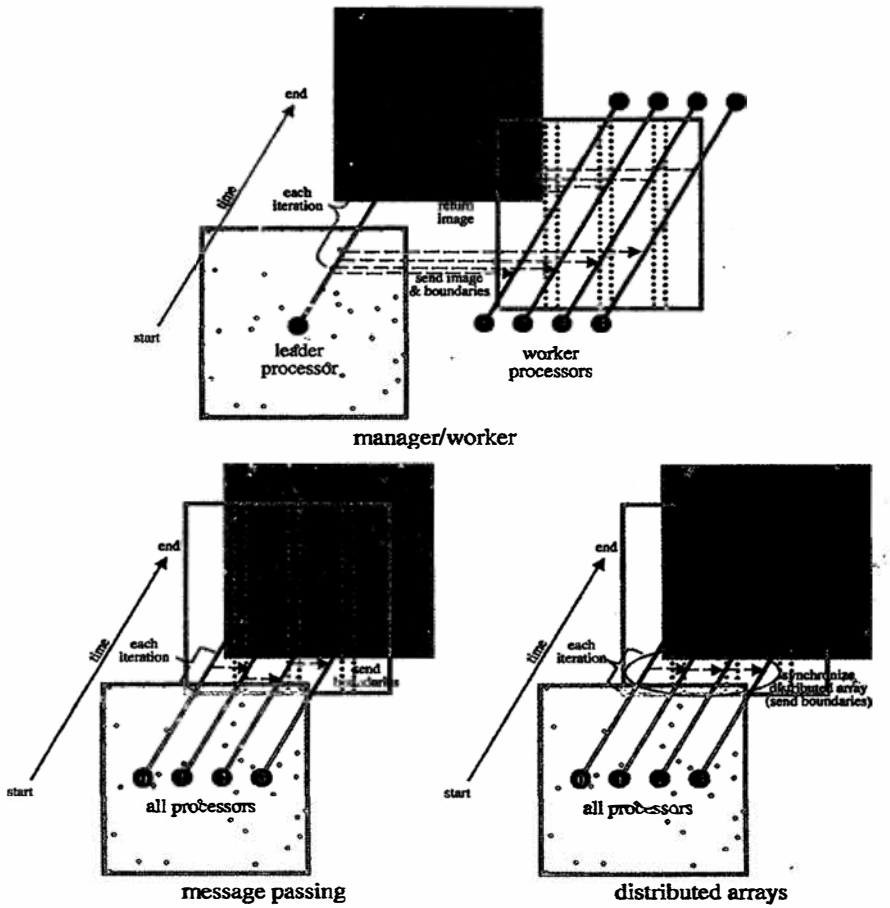


Рисунок 4.5. Виды параллельных реализаций программы Blurimage
Реализации параллельной программы по схеме «начальник/подчиненный» на основе обмена сообщениями и распределенных массивов для программы Blurimage. Пунктирными линиями обозначены взаимодействия между экземплярами P_{ID}

Алгоритм 4.2. Реализация Blurimage на основе модели «начальник/подчиненный»

Алгоритм состоит из двух программ: программы начальник и программы подчиненный. Программа начальник выполняется на одной машине, посылает входные данные каждому подчиненному (строка 5) и получает результат, когда подчиненный

выполнил свертку (строка 7). Программа исполнителя выполняется на Np процессорах, получает входные данные от начальника (строка 3), выполняет свертку (строка 4) и посылает результат (без границы) обратно (строка 5).

$\mathbf{Z} : \mathbb{R}_+^{N_x \times N_y} = \textsc{BlurimageManager}($
$\qquad \mathbf{Z} : \mathbb{R}_+^{N_x \times N_y}, \mathbf{K} : \mathbb{R}_+^{N_K \times N_K}, N_{blur})$

1 $i_{start} = \lfloor N_x(0 : N_P - 1)/N_P \rfloor + 1$
2 $i_{end} = \lfloor N_x(1 : N_P)/N_P \rfloor + 1$
3 **for** $i_{blur} = 1 : N_{blur}$
4 $\quad$ **do for** $P_{ID} = 1 : N_P$
5 $\quad\quad$ **do** $\textsc{Send}(P_{ID}, \mathbf{Z}(i_{start}(P_{ID}) : \min(i_{end}(P_{ID}) + N_K, N_X), :)$
6 $\quad$ **for** $P_{ID} = 1 : N_P$
7 $\quad\quad$ **do** $\mathbf{Z}(i_{start}(P_{ID}) : i_{end}(P_{ID}), :) = \textsc{Receive}(P_{ID})$

$\textsc{BlurimageWorker}(\mathbf{K} : \mathbb{R}_+^{N_K \times N_K}, N_{blur})$

1 **for** $i_{blur} = 1 : N_{blur}$
2 $\quad$ **do**
3 $\quad\quad \mathbf{Z} = \textsc{Receive}(manager)$
4 $\quad\quad \mathbf{Z} = \mathbf{K} \circ \mathbf{Z}$
5 $\quad\quad \textsc{Send}(manager, \mathbf{Z}(1 : end - N_K, :))$

Реализация на основе обмена сообщениями решает оба этих вопроса посредством того, что каждый P_{ID} инициализирует и хранит только часть изображения, над которым работает, а также дополнительные граничные точки, необходимые ему для выполнения вычисления. Единственной необходимой коммуникацией между итерациями является обновление граничных точек новыми значениями, вычисленными смежным P_{ID}. В результате при каждой итерации посылается только $N_K N_y N_p$ пикселей (или $8 N_K N_p / N_x$ байтов на пиксель). Результирующим отношением вычисление– коммуникация в операциях на байт является

$$\text{вычисление/коммуникация} = 2N_K^2 / (8N_K N_P / N_x) = \frac{1}{4} N_K N_x / N_P.$$

При $N_K = 32$ и $N_x/N_P = 1024$ получается отношение 8192, что в 32 раза выше, чем отношение для алгоритма начальник/подчиненный. Это означает, что алгоритм обмена сообщениями выполняет приблизительно в 32 раза меньший объем коммуникации, чем алгоритм начальник/подчиненный.

Данный этап коммуникации включает в себя определение того, какие P_{ID} являются смежными, упаковку данных, отправку данных и их получение. На одном уровне эта операция являются относительно простой, так как P_{ID} фактически выполняют почти одно и то же. К сожалению, то, что выполняет каждый P_{ID} в отдельности, немного отличается от того, что выполняет любой другой P_{ID}. Два P_{ID} не будут посылать или получать одни и те же данные от одних и тех же P_{ID} или для них. Написание и отладка этих типов коммуникации требуют учета многих деталей в большом объеме и являются уязвимыми для ошибок.

Алгоритм 4.3. Реализация Blurimage на основе обмена сообщениями.

Алгоритм состоит из выполнения одной программы на всех экземплярах P_{ID}. Каждый P_{ID} работает только с $\tilde{Z}$, являющейся его ($Nx/Np+N_K \times Ny$)-й частью целого изображения. При каждой итерации каждый экземпляр P_{ID} явно посылает (строка 4) и получает (строка 6) границу для/от своих соседних P_{ID} и затем выполняет свертку (строка 7).

$$\tilde{\mathbf{Z}} : \mathbb{R}_+^{N_x/N_P+N_K \times N_y} = \textsc{BlurimageMessagePassing}(\tilde{\mathbf{Z}} : \mathbb{R}_+^{N_x/N_P+N_k \times N_y}, \mathbf{K} : \mathbb{R}_+^{N_K \times N_K}, N_{blur})$$

1 **for** $i_{blur} = 1 : N_{blur}$
2 **do**
3 **if** $(P_{ID} > 1)$
4 **do** $\textsc{Send}(P_{ID} - 1, \tilde{\mathbf{Z}}(1 : N_K, :))$
5 **if** $(P_{ID} < N_P)$
6 **do** $\tilde{\mathbf{Z}}(N_x/N_P : N_x/N_P + N_K, :) = \textsc{Receive}(P_{ID} + 1)$
7 $\tilde{\mathbf{Z}} = \mathbf{K} \circ \tilde{\mathbf{Z}}$

Алгоритм 4.4. Реализация Blurimage на основе распределенных массивов

Алгоритм похож на последовательный, но имеет следующие отличиях: Во-первых, распределенный массив **Z** объявлен с N_K накладывающейся граничной информацией с использованием записи $P_{ouNk}()$. Во-вторых, каждый P_{ID} выполняет свертку на своей локальной части изображения (строка 3). Наконец, граничные условия синхронизируются (строка 4), что вызывает обмен перекрывающихся границ между соседними P_{ID}.

$$\mathbf{Z} : \mathbb{R}_+^{P_{ov(N_K)}(N_x) \times N_y} = \textsc{BlurimageDistributedArray}(\mathbf{Z} : \mathbb{R}_+^{P_{ov(N_K)}(N_x) \times N_y}, \mathbf{K} : \mathbb{R}_+^{N_K \times N_K}, N_{blur})$$

1 **for** $i_{blur} = 1 : N_{blur}$
2 **do**
3 $\mathbf{Z}.loc = \mathbf{K} \circ \mathbf{Z}.loc$
4 $\textsc{Synch}(\mathbf{Z})$

Реализация на основе распределенных массивов задействует такой же шаблон коммуникации, что и реализация на основе обмена сообщениями (и имеет такое же соотношение вычисление–коммуникация), однако в первом случае используется метод более простой с точки зрения кодирования. Единое распределенное изображение создается по всем экземплярам P_{ID} с указанным перекрытием между P_{ID}. Каждый P_{ID} выполняет операцию свертки на своих локальных данных, и обновления граничных условий осуществляется посредством единственной распределенной команды синхронизации, которая определяет, что ближайшим соседям необходимо отправить только границы, а также управляет всеми необходимыми сообщениями и индексированием массивов.

4.2. Кодирование

В предыдущем разделе была представлена концепция шаблона коммуникации. От этого шаблона зависит, какую модель программирования необходимо использовать. В разделе, посвященном тестированию, станет очевидно, что модель программирования может иметь большое влияние на производительность (как правило, по причине того, что различные модели программирования в большей или меньшей степени поддерживают различные шаблоны коммуникации). Кроме того, модель параллельного программирования может оказывать большое влияние на сложность написания программы. Действительно, иногда целесообразнее использовать более простую модель (например, «начальник/подчиненный») с меньшей производительностью, чем более сложную (например, обмен сообщениями), которая имеет более высокую производительность.

В данном разделе рассматривается процесс распараллеливания с использованием этих моделей программирования и также связанные с ними компромиссы. Из-за немалого размера кодов моделей «начальник/подчиненный» и «обмен сообщениями» будет показана и выполнена только версия с распределенными массивами.

При преобразовании последовательной программы в параллельную необходимо помнить о нескольких важных целях этого процесса. Первой целью является достижение оптимальной производительности путем представления параллелизма в задаче с тем, чтобы каждый процессор был полностью задействован при минимизации накладных расходов коммуникации. Второй целью, которая обычно не согласуется с первой, является минимизация сложности программы. Сложность программы можно оценить по следующим факторам.

- Рост размера кода. Насколько больше параллельный код по сравнению с последовательным?
- Распределение изменений кода. Локализованы ли изменения, или они вносятся по всему коду?
- Масштабирование. Параметризованы ли функции при помощи *Np* или размера задачи? Минимизировано ли явное использование параметров, таких как, например, *Np* или P_{ID}?
- Тестирование. Насколько больше тестов необходимо выполнить для проверки правильности выполнения программы?
- Отладка. Является ли параллельный код правильным последовательным кодом? Поддерживает ли код прямой процесс перехода от последовательного кода к параллельному?

4.2.1. Начальник/подчиненный

Данный подход к реализации параллельных программ на MATLAB является одним из самых старых. В схемах «начальник/подчиненный» [Choy & Edelman 1999, Morrow et al. 1998, Dean et al. 2005] применяется MATLAB как пользовательский интерфейс в качестве начальник набору подчиненный. Подчиненный могут выполнять или не выполнять MATLAB. Например, StarP хранит распределенные массивы на параллельном сервере, который вызывает необходимые подпрограммы из параллель-

ных библиотек, таких как ScaLAPACK и FFTW. Такие методы зачастую обеспечивают лучшую производительность, когда данные передаются на сервер. Однако эти методы ограничены теми функциями, которые были непосредственно связаны с параллельной библиотекой. pMATLAB может работать в данном режиме, хотя в случае pMATLAB исполнительным сервером является MATLAB, выполняющийся на каждом P_{ID}, и пользователь ответственен за разбиение вычислений на массово-параллельные задачи, которые можно независимо распределить по исполнителям. В настоящем разделе дается краткое описание реализации схемы «начальник/подчиненный» программы Blurimage. Ключевая концептуальная разница между моделью «начальник/подчиненный» и другими моделями программирования заключается в том, что существуют две (возможно и три) разные роли, исполняемые компьютерами. Существует пользовательская среда MATLAB, которая запускает программу, разбивает работу на задачи и рассылает их подчиненному. Затем начальник отправляет задачи многочисленным исполнительным экземплярам P_{ID}, которые в свою очередь направляют свои результаты обратно начальнику для передачи среде MATLAB.

Код «начальник/подчиненный» требует, чтобы к базовому последовательному алгоритму были добавлены четыре дополнительных элемента функциональности.

- Начальник вычисляет часть изображений, которую должен получить каждый исполнитель.
- Начальник запускает каждого исполнителя со своей частью изображения.
- Подчиненные посылают свои части изображения обратно ведущему процессору и затем выходят из процедуры.
- Начальник получает части изображения и формирует конечное изображение.

В коде начальника управляющий P_{ID} вычисляет границы изображения, за которые несет ответственность каждый подчиненный, и помещает их в массивы индексов `i_start` и `i_stop`. Эти индексы затем используются для копирования частей изображения (включая перекрывающиеся края) в массив `image_overlap_cell`, который имеется в одном экземпляре на каждый P_{ID}. Следующая команда dfeval разбивает изображение по исполнителям. Подчиненный выполняет свертку, и ведущий собирает результаты обратно. Наконец, возвращенный массив ячеек конвертируется обратно в изображение при помощи команды `cell2mat`. Добавление функциональности коду имеет ряд последствий.

Рост размера кода. Переход от последовательной программы к программе «начальник/подчиненный», как правило, приводит к незначительному увеличению размеров кода.

Распределение изменений кода. Является наиболее значительным последствием использования программы «начальник/подчиненный», так как программу потребуется разбить на две части, каждую строку кода необходимо назначить одной из двух программ.

Масштабирование. Как правило, программы «начальник/подчиненный» разделяют задачи на основании определенного предустановленного размера малой задачи и затем варьируют размер малой задачи на основании общего размера основной задачи. К сожалению, это часто приводит к появлению множества мелких задач, которые не могут эффективно задействовать модель «начальник/подчиненный». Для решения

данной проблемы код «начальник/подчиненный» может быть параметризован количеством исполнителей, что приведет к созданию более эффективного кода. К сожалению, такая параметризация исключает одно из главных преимуществ кода «начальник/подчиненный»: отсутствие явной зависимости от *Np*.

Тестирование. Количество необходимых тестов сокращается, так как проведение тестирования последовательной программы более не несет никакого смысла.

Отладка. Вероятно, самый большой недостаток данного метода. В модели «начальник/подчиненный» последовательная программа утрачена, и зачастую невозможно выполнить программу чисто в последовательной форме. Таким образом, отсутствует чистый процесс систематического отделения и тестирования конкретных аспектов параллельной функциональности. Отсутствие процесса значительно увеличивает время создания кода программы, так как для него более сложно выделить и выявить ошибки.

С точки зрения написания кода модель «начальник/подчиненный» хорошо справляется с задачей обеспечения небольших размеров и простоты программы. Потеря возможности выполнить программу только в последовательном виде влечет за собой трудности тестирования. С точки зрения производительности программа «начальник/подчиненный» требует, чтобы все части изображения были отправлены обратно ведущему и повторно переданы исполнителям при каждой итерации, что ведет к большим накладным расходам. Кроме того, по причине того, что ведущий процессор должен хранить целое изображение, возможности этой программы ограничиваются обработкой изображения, которое помещается в его памяти.

4.2.2. Обмен сообщениями

В данном разделе дается краткое описание реализации программы Blurimage на основе модели «обмена сообщениями». Данный метод [Kepner 2004, Zollweg 2006, Hudak 2008] требует отправки пользователем сообщений внутри кода. Эти методы часто реализуют вариант стандарта MPI [Gropp et al. 1999]. Обмен сообщениями позволяет любому P_{ID} напрямую взаимодействовать в любым другим P_{ID} и обеспечивает минимально необходимую функциональность для реализации любой параллельной программы.

Как и модель на основе распределенных массивов, программа «обмена сообщениями» выполняет одну и ту же программу на каждом P_{ID}, но работает с несколько отличными данными. Это относится все к той же модели реализации «Одна программа, несколько потоков данных» (SPMD, произносится «спим-ди») [Flynn 1972]. Ключевая концептуальная разница между моделью на основе распределенных массивов и моделью обмена сообщениями заключается в наличии явного управления обменом сообщениями между вычислительными машинами. В программе «обмена сообщениями» существуют два ключевых вида данных, которые программа использует для управления действиями: количество процессоров, *(Np)*, и идентификатор процессора, (P_{ID}). Используя эту информацию, каждый P_{ID} определяет часть задачи, над которой он работает, и с какими другими экземплярами P_{ID} ему необходимо взаимодействовать.

Код «обмена сообщениями» требует дополнения исходного последовательного кода четырьмя функциональными элементами.

- Каждый экземпляр P_{ID} определяет размер своего локального изображения.
- Каждый экземпляр P_{ID} определяет идентификаторы соседних экземпляров P_{ID}.

- Каждый экземпляр P_{ID} посылает граничные данные соседним P_{ID}.
- Каждый экземпляр P_{ID} получает границу от соседних P_{ID}.

Данный код состоит из одной программы, работающей на всех экземплярах P_{ID}. Каждый P_{ID} работает только на своей локальной части изображения. Точный размер этого локального изображения должен быть явно рассчитан. На каждой итерации каждый P_{ID} явно посылает и получает граничные данные для/от своих соседних экземпляров P_{ID} и затем выполняет свертку. Добавление этой функциональности коду имеет ряд последствий.

Рост размера кода. Переход от последовательной программы к программе обмена сообщениями, как правило, приводит к значительному увеличению размера кода. Последовательные программы, преобразованные в параллельные с использованием обмена сообщениями, как правило, увеличиваются на 25–50%; для сравнения, методы «начальник/подчиненный» и «распределенные массивы» увеличивают размер кода приблизительно на 5% [Funk & Kepner 2005].

Распределение изменений кода. Программы обмена сообщениями, как правило, требуют внесения изменений в программу на нижнем уровне, и изменения кода часто носят нелокализованный характер.

Масштабирование. Программы с обменом сообщениями обычно требуют наименьшего объема коммуникации среди всех моделей программирования и часто демонстрируют наилучшую масштабируемость производительности. К сожалению, программы обмена сообщениями по своей природе зависят от Np и P_{ID}. Эти понятия используются на протяжении всей программы. При надлежащей реализации это можно решить параметризацией. Однако при этом потребуется много усилий для проверки краевых условий при масштабировании Np или размера входных данных.

Тестирование. Количество необходимых тестов сокращается, так как проведение тестирования последовательной программы более не имеет никакого смысла.

Отладка. Пользователи, знакомые с сообщениями, находят данный метод эффективным. Однако рядовому пользователю необходимо будет изучить довольно много, так как метод явного обмена сообщениями находится на нижнем уровне абстракций и требует от пользователя работы напрямую с взаимоблокировками, синхронизацией и другими понятиями параллельного программирования нижнего уровня. Как и в модели «начальник/подчиненный», последовательная программа теряется, и невозможно выполнить программу в чисто последовательном виде. Таким образом, отсутствует ясный процесс систематического отделения и тестирования конкретных аспектов параллельной функциональности. Отсутствие процесса значительно увеличивает время создания кода, так как в нем сложнее выявить и устранить ошибки.

Несмотря на эти трудности, возможности обмена сообщениями являются необходимыми как для метода «начальник/подчиненный», так и для метода «распределенных массивов». Более того, обмен сообщениями зачастую служит самым эффективным методом реализации программы, и определенные типы программ со сложными моделями коммуникации могут быть реализованы только при помощи прямого обмена сообщениями. Таким образом, любое законченное параллельное решение должно обеспечивать механизм доступа к нижележащему слою обмена сообщениями.

В данном случае программа «обмена сообщениями» и программа распределенных массивов выполняют одинаковую коммуникацию, и использование метода «обме-

на сообщениями» в отношении производительности не дает никаких преимуществ. Основным недостатком такой программы является значительная усложненность. Код программы больше, изменения последовательной программы не локализованы, и ее сложно тестировать.

4.2.3. Распределенные массивы

Реализация программы на основе распределенных массивов уже рассматривалась и была проиллюстрирована на рисунке 4.5. Реализация с помощью распределенных массивов аналогична подходу программы Mandelbrot и других программ, описанных в главе 2. Код, использующий распределенные массивы, требует добавления четырех дополнительных функциональных элементов.

- Создание карт размещения и распределенного изображения с накладывающимися краями.
- Выполнения свертки на локальной части изображения.
- Синхронизация границ.
- Собирание ведущим процессором частей изображения для создания конечного изображения.

Ядро кода распределенных массивов показано ниже (полный листинг находится в `Examples/Blurimage/pBlurimage.m`).

В данном случае модель на основе распределенных массивов обеспечивает лучшее одновременно из миров модели «начальник/подчиненный» и модели «обмена сообщениями». Она обеспечивает небольшой простой код, который легко тестируется как программа, работающая по схеме «начальник/подчиненный». Кроме того, память и коммуникации масштабируемы, как в программе обмена сообщениями.

Листинг кода. Параллельная MATLAB-программа Blurimage

```
% Set image size (scaled by Np), filter size and blur.
Nx = 1024*Np; Ny = 1024; Nk = 32; Nblur = 2;
PARALLEL = 1;                          % Set control flag.
Zmap = 1;                              % Create serial map.
if (PARALLEL)                          % Parallel map with overlap.
Zmap = map([Np 1] ,{},0:Np-1, [Nk 0]);
end
Z = zeros(Nx,Ny,Zmap) + 1.e-4;         % 2D distributed array.
[ii jj] = find(sprand(Nx,Ny,1e-4)); % Create noN-zeros.
for i=1:numel(ii)
Z(ii(i),jj(i))=1;                      % Insert noN-zeros.
end
my I = global_ind(Z,1); % Get local i indices
myJ = global_ind(Z,2); % Get local j indices.
kernel = ones(Nk,Nk); % Create kernel.
Z = synch(Z); % Synchronize boundary conditions.
for iblur = 1:Nblur                    % Loop over each blur.
Zloc = local(Z);                       % Get local data.
```

```
19 Zloc(1:end-Nk+1,1:end-Nk+1) = conv2(Zloc,kernel, 'valid');
20 Z = put_local(Z,Zloc); % Put local back in global.
21 Z = synch(Z);                          % Copy overlapping boundaries.
22 end
```

Перевод
1 Установить размер изображения (кратный Np), размер фильтра и размытие.
3 Установить контрольный флаг.
4 Создать последовательную карту.
5 Параллельная карта с наложением.
8 Двумерный распределенный массив.
9 Создать ненулевые значения.
11 Вставить ненулевые значения.
13 Получить локальные i–индексы.
14 Получить локальные j–индексы .
15 Создать ядро.
16 Синхронизировать граничные условия.
17 Цикл по каждому размытию.
18 Получить локальные данные.
20 Поместить локальное обратно в глобальное.
21 Копировать границы наложения.

4.3. Отладка

Цель отладки параллельной программы заключается в достижении высокого уровня производительности при выполнении интересующей нас задачи. Теоретически это можно сделать по окончании написания программы. Практически появилось много дополнительных функций, добавленных к программе, и каждая из них должна отлаживаться отдельно. Кроме того, параллельная программа обычно нацелена на задачу, требующую длительного выполнения, а процесс отладки происходит гораздо быстрее, если программа быстро завершает свою работу. Процесс достижения компромисса между этими конкурирующими целями приведен в главе 2. Этот процесс изображен на рисунке 4.6 и состоит из четырех основных шагов. Очень важно выполнять эти шаги итерационно. При обнаружении ошибки рекомендуется вернуться в начало и попытаться отладить ошибку на более раннем этапе.

Первый шаг – необходимо взять последовательный код и добавить базовую функциональность распределенных массивов, используя карту размещения. Этот код выполняется на одном процессоре локально:

```
eval(pRun('pBlurimage',1,{})).
```

Цель заключается в определении того, является ли код распределенного массива синтаксически корректным. Как правило, ошибки, встречающиеся на этом шаге, имеют отношение к преобразованию массивов в распределенные массивы.

Второй шаг – тестирование параметризации кода в отношении *Np*. Этот код выполняется с использованием множественных экземпляров MATLAB на локальном компьютере:

```
eval(pRun('pBlurimage',2,ParallelMachine)).
```

Цель заключается в определении того, является ли декомпозиция синтаксически корректной. Как правило, ошибки, встречающиеся на данном шаге, имеют отношение к арифметике декомпозиции.

Третий шаг – обычно не требует дополнительного составления кода и заключается в выполнении на удаленных компьютерах:

```
eval(pRun('pBlurimage',2,ParallelMachine)).
```

На этом шаге, главным образом, тестируются нижележащие механизмы распределения данных и удаленного выполнения программы. Как правило, ошибки, встречающиеся на данном шаге, имеют отношение к конфигурации установки параллельного программного обеспечения или файловой системе. Например, функция, необходимая для выполнения программы, существует на локальной машине, но недоступна на удаленном компьютере.

Четвертый шаг – как правило, также не требует дополнительного кода, заключается в выполнении на удаленных компьютерах при установке постоянно увеличивающихся значений *Np* и увеличении задачи до целевых размеров.

```
eval(pRun('pBlurimage',4,ParallelMachine)).
```

Цель заключается в начале процесса оценки производительности программы и определении того, достигнута ли желаемая производительность.

4.4. Тестирование

В предыдущем разделе мы запускали версию программы Blurimage с распределенными массивами для задач различных размеров. В дополнение к визуальному выводу, каждый запуск также сопровождался отображением производительности программы в гигафлопсах (один миллиард операций с плавающей точкой в секунду). Следующий вопрос – являются эти значения хорошим или плохим показателем? Другими словами, является ли pBlurimage хорошо написанной параллельной программой?

Для ответа на этот вопрос требуется собрать данные о производительности и провести более глубокий анализ. Делением производительности на производительность, достигнутую на одном процессоре, получаем достигнутое ускорение вычислений параллельной программы. В таблице 4.2 показано относительное время вычислений (нормализованное к случаю $Np = 1$), время запуска и относительная производительность. Эти данные были собраны на 8-процессорной многоядерной системе и 16-узловой многоузловой системе. В многоядерной системе все процессоры используют

общую память. В многоузловой системе каждый узел имеет отдельную память, и процессоры взаимодействуют через сеть.

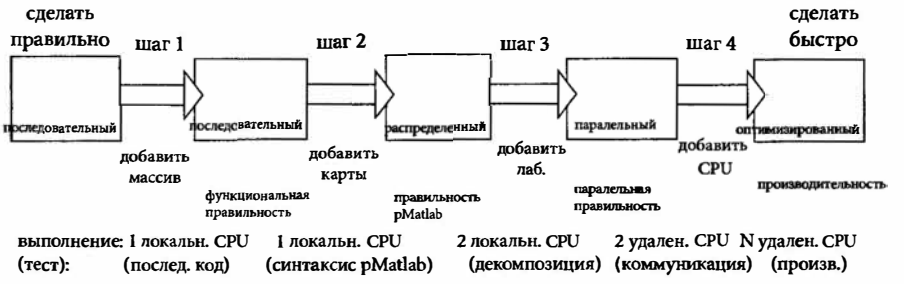


Рисунок 4.6. Четырехступенчатый процесс отладки

На каждом шаге добавляется минимальный объем функциональности, и выполняются специальные тесты для тестирования этой функциональности

Таблица 4.2. Производительность программы BlurImage на многоядерных и многоузловых системах

Относительное время вычислений, абсолютное время запуска+коммуникации, коэффициент ускорения вычислений на многоядерных и многоузловых вычислительных системах для различных значений *Np*. Данные о производительности приведены для масштабированного размера задачи (*Nx* α *Np*). Все значения нормализованы по столбцу 2, в котором *Np* = *1* с выключенными распределенными массивами (`PARALLEL=0`). Столбцы 3–7 содержат значения с включенными распределенными массивами (`PARALLEL=1`). Запись 1*8 означает, что *Np* = 8 и все экземпляры P_{ID} были выполнены на одном вычислительном узле при совместно используемой памяти. Запись 16*1 означает, что *Np* = 16 и каждый P_{ID} был выполнен на своем индивидуальном вычислительном узле с использованием отдельной памяти.

Многоядерные процессоры:	1	1*1	1*2	1*4	1*8	
Относительное время вычислений	1	1.0	0.85	0.87	0.86	
Время запуска (с)	0.0	0.0	0.7	1.2	1.8	
Относительная производительность	1	1.0	2.3	4.6	9.3	
Многоузловые процессоры:	1	1*1	2*1	4*1	8*1	16*1
Относительное время вычислений	1	1.0	1.0	1.0	1.0	1.0
Время запуска (с)	0.0	0.0	8.0	8.0	7.8	7.8
Относительная производительность	1	1.0	2.0	4.0	7.9	15.9

Сравнение производительности выполнения программы с использованием *Np* = 1 при выключенных (`PARALLEL=0`) и включенных (`PARALLEL=1`) распределенных массивах показывает, что распределенные массивы не вызывают ухудшения производительности. Если использование распределенных массивов с *Np* = 1 приводит к значительному замедлению работы, необходимо уделить внимание этой проблеме. Очень трудно получить какую-либо эффективность при *Np* > 1, если при *Np* = 1 программа работает медленнее, чем лежащий в ее основе последовательный код. Работа

лишь на локальных частях распределенных массивов почти гарантирует ту же производительность, что и без использования распределенных массивов.

Таблица 3.2 демонстрирует сокращение времени вычислений при увеличении *Np* как в многоядерной, так и в многоузловой системе. Однако в параллельной программе обычно увеличивается время, затрачиваемое на запуск и коммуникацию между каждым из экземпляров MATLAB. В таблице 4.2 прослеживается увеличение времени запуска + коммуникации с увеличением *Np*, а также сравнительно больший временной показатель запуска + коммуникации в многоузловой системе, так как запуск и коммуникация в этом случае осуществляются через сеть.

Выполнение Blurimage осуществлялось на кратном размере задачи (т. е. $Nx \propto Np$). Время исполнения почти не изменяется с увеличением *Np*. Производительность возрастает, так как общий объем работы также возрастает с увеличением *Np*.

4.5. Общий анализ производительности

Напоминаем, что в алгоритме Blurimage требуется $O(N_K^2)$ операций на вычисление каждого пикселя. При реализации модели «начальник/подчиненный» прекрасно осуществляется распределение вычислений таким образом, что на каждый P_{ID} приходится равное количество пикселей изображения. Таким образом, производительность вычислений должна быть кратна значению *Np*. Однако при реализации метода «начальник/подчиненный» целое изображение необходимо посылать туда и обратно от лидера к исполнителям на каждом шаге. Другими словами, каждый вычисляемый пиксель посылается дважды. Не учитывая постоянные величины, при этом получаем отношение вычисление–коммуникация $O(N^2_K)$ операций на каждый отправленный пиксель, которое для $N_K=16$ равно 256. Данное отношение сравнимо с получаемым на типичных параллельных машинах. Таким образом, эффективность на уровне 0,5 может являться максимально возможной при использовании метода «начальник/подчиненный».

Учитывая, что схема «начальник/подчиненный» ограничивает производительность приложения, с какой эффективностью будут работать модели обмена сообщениями и распределенных массивов? Вычисления в обеих моделях остаются такими же, как в модели «начальник/подчиненный»: $O(N^2_K)$ на пиксель. Обе реализации – обмен сообщениями и распределенные массивы – используют модель коммуникации «ближайшей соседней точки». В данной модели только небольшая часть пикселей изображения должна быть передана: $O(N_K/N_I)$ для изображения размером $N_I \times N_I$. Результатом модели «ближайшей соседней точки» является отношение вычисление–коммуникация, равное $O(N_K/N_I)$ операций на отправленный пиксель, которое для N_I =1024 предполагает отношение около 16 000. Эта цифра намного превышает полученную при использовании модели «начальник/подчиненный». Таким образом, обе реализации, а именно обмен сообщениями и распределенные массивы, должны иметь высокую производительность с увеличением *Np*, как и было показано в таблице 4.2.

4.6. Выводы

Цель данной главы заключалась в кратком представлении большинства концепций, упоминающихся в книге.

Проектирование: оценка необходимости распараллеливания, классы параллельных программ, модели параллельного программирования, машинные модели.

Кодирование: различные стили параллельного программирования, влияние кода на производительность и масштабируемость.

Отладка: особенности процесса параллельной отладки, ошибки, которые можно обнаружить на каждом шаге.

Тестирование: метрика производительности и прогнозирование производительности (относительно параметров машины).

К счастью, эти концепции охватывают большую часть необходимой информации. Далее по ходу книги будут представлены некоторые новые концепции. В основном, оставшаяся часть книги посвящена более подробному изучению уже упомянутых концепций и обучению читателя необходимым навыкам их использования в собственных программах.

Список литературы

1. *[Choy & Edelman 1999] Ron Choy and Alan Edelman, PARALLEL MATLAB: Doing it right. Proceedings of the IEEE (Feb. 2005), Vol. 93, No. 2, pp. 331–341.*

2. *[Dean et al. 2005] Loren Dean, Silvina Grad-Freilich, J. Kepner, and A. Reuther, Distributed and PARALLEL Computing with MATLAB. Tutorial presented at Supercomputing 2005, Nov. 12, 2005, Seattle, WA.*

3. *[Flynn 1972] Michael Flynn, Some computer organizations and their effectiveness, IEEE Trans. Comput., Vol. C–21, p. 948, 1972.*

4. *[Funk & Kepner 2005] Andrew Funk, Jeremy Kepner, Loren Hochstein, and Victor Basili, A Relative Development Time Productivity Metric for HPC Systems, HPEC Workshop, Sep. 20–22, 2005, Lexington, MA.*

5. *[Gropp et al. 1999] William Gropp, Ewing Lusk, and Anthony Skjellum, Using MP!: Portable PARALLEL Programing with the Message Passing Interface, Cambridge, MA: MIT Press, 1999.*

6. *[Hudak 2008] Dave Hudak, Blue Collar MPI, http://www.bluecollarcomputing.org [Kepner 2004] Jeremy Kepner and Stan Ahalt, MATLABMPI, Journal of PARALLEL and Distributed Computing, Vol. 64, No. 8, pp. 997–1005, 2004.*

7. *[Lebak et al. 2005] James Lebak, Jeremy Kepner, Henry Hoffmann, and Edward Rutledge, PARALLEL VSIPL++: An open standard software library for highperformance PARALLEL signal processing, 2005, Proceedings of the IEEE: Special Issue on Program Generalization, Optimization, and Platform Adaptation, Jose Moura (editor).*

8. *[Mattson et al. 2004] Timothy G. Mattson, Beverly A. Sanders, and Berna L.*

9. *Massingill, Patterns for PARALLEL Programming 2004, Pearson Education, Boston, MA.*

10. *[McGarrity 2007] Stuart McGarrity, Programming Patterns: Maximizing Code Performance by Optimizing Memory Access, The Math Works News & Notes – June 2007.*

11. *[Morrow et al. 1998] G. Morrow and R. A. van de Geijn, PARALLEL Linear Algebra Server for MATLAB-like Environments. In Proceedings of Supercomputing 1998, Orlando, FL, November 1998.*

12. *[Zollweg 2006] John Zollweg, Cornell Multitask Toolbox for MATLAB, 2006, http://www.tc.cornell.edu/services/support/forms/cmtm.p2.htm*

Глава 5
Разработка сложных программ с распределенными массивами

Резюме

Для эффективного использования распределенных массивов необходимо понимание базовой технологии, которая применяется для их реализации. Распределенные массивы MATLAB в основном реализованы в библиотеке с использованием методов перегрузки операторов объектно–ориентированного программирования. Основную структуру данных в библиотеке представляет распределенный числовой массив, распределение которого по множеству процессоров определяется структурой карты размещения. Операции коммуникации между распределенными массивами находятся на нижнем уровне абстракции, не видимом пользователю. Типы распределения данных, поддерживаемые картами, как правило, принадлежат классу N-мерных блочно-циклических распределений с перекрытием. Блочные распределения используются чаще всего. Циклические распределения зачастую могут быть полезными при решении вопросов балансировки нагрузки. Библиотека распределенных массивов базируется на библиотеке обмена сообщениями точка–точка, например, на интерфейсе передачи сообщений (MPI). Библиотека сообщений устанавливает основной лимит на производительность коммуникации в библиотеке распределенных массивов.

5.1. Введение

Сущность библиотеки распределенных массивов заключается в способности поддерживать параллельное перераспределение в распределенных массивах

$$\mathbf{B} = \mathbf{A}.$$

В простом случае, когда **A** и **B** распределены в соответствии с картами размещения идентично (например, $\mathrm{A}, \mathrm{B} : \mathrm{R}^{P(N)\times N}$), операция относительно проста. Каждый P_{ID} просто копирует свою локальную часть **A** в свою локальную часть **B**; иными словами

$$\mathbf{B}.loc = \mathbf{A}.\ loc.$$

[Помните, что запись $P()$ обозначает разбиение определенного измерения на однотипные блоки в Np процессорах]. В более сложном случае, например, $\mathbf{B} : \mathrm{R}^{P(N)\times N}$ и $\mathrm{B} : \mathrm{R}^{P(N)\times N}$, лежащая в основе библиотека распределенных массивов должна определить, какой идентификатор P_{ID} «владеет» какими элементами данных, и отправить соответствующие элементы данных соответствующим P_{ID}. В главах 2 и 3 были наглядно продемонстрированы более сложные примеры, которые включали наложение границ и конвейерную обработку данных наборов P_{ID}. Типы распределения данных, поддерживаемые картами размещения, как правило, принадлежат классу N-мерных блочно-циклических распределений с наложением. В итоге именно библиотека распределен-

ных массивов должна определять это таким образом, чтобы обеспечить простоту и эффективность использования.

5.2. Сравнительная характеристика чистых и фрагментированных распределенных массивов

Программирование на основе распределенных массивов в MATLAB имеет несколько предшественников, среди которых наиболее заметные – pMATLAB [Bliss & Kepner 2007], StarP [Choy & Edelman 1999], Falcon [Falcon] и иерархические расширения массивов [Kim & Kepner 2006, Bikshandi et al. 2006]. Эти методы предоставляют механизм создания массивов, которые распределяются среди множества процессоров. Распределенные массивы давно применяются в других языках, например, Fortran [Koelbel 1994, Numrich & Reid 1998] и C [El-Ghazawi et al. 2005], а также во многих библиотеках C++, таких как POOMA [Cummings et al. 1998], GA Toolkit [Nieplocha et al. 2002], PVL [Rutledge & Kepner 1999] и PARALLEL VSIPL++ [Lebak et al. 2005]. Метод распределенных массивов позволяет пользователю видеть распределенный объект как единое целое. Этот метод позволяет выполнять операции на массиве в целом или на локальных частях массива. Кроме того, эти библиотеки совместимы с библиотеками коммуникации, такими как MPI, и способны быть реализованы на основе гибридной общей/распределенной памяти.

Распределенные массивы можно использовать как в моделях чистых распределенных массивов, так и в моделях программирования на основе фрагментированных распределенных массивов (см. рисунок 5.1). Модель на основе чистых распределенных массивов представляет полный глобальный вид распределенного массива. В частности, распределенные массивы, будучи созданными с помощью соответствующего объекта карты, рассматриваются одинаковым образом вместе с нераспределенными массивами. При использовании модели программирования пользователь никогда не обращается к локальной части массива, и все операции (такие как умножения матриц, БПФ, свертки и т. д.) совершаются на глобальной структуре. Преимуществами использования чистых распределенных массивов являются простота программирования и высочайший уровень абстрагирования. К недостаткам относится необходимость реализации параллельных версий последовательных операций и вытекающие из этого накладные расходы производительности библиотеки.

Модель на основе фрагментированных распределенных массивов сохраняет высокий уровень абстрагирования, но также позволяет осуществлять доступ к локальным частям массива. В частности, распределенные массивы создаются так же, как и чистые распределенные массивы; однако, операции можно выполнять только на локальной части массива. Позже глобальную структуру можно обновить с учетом результатов локальных вычислений. Данный подход обеспечивает большую гибкость. Кроме того, он не требует функционального покрытия или реализации параллельных версий всех существующих последовательных функций. Более того, программы на основе фрагментированных распределенных массивов часто достигают более высокой производительности за счет устранения накладных расходов локальной коммуникации библиотеки.

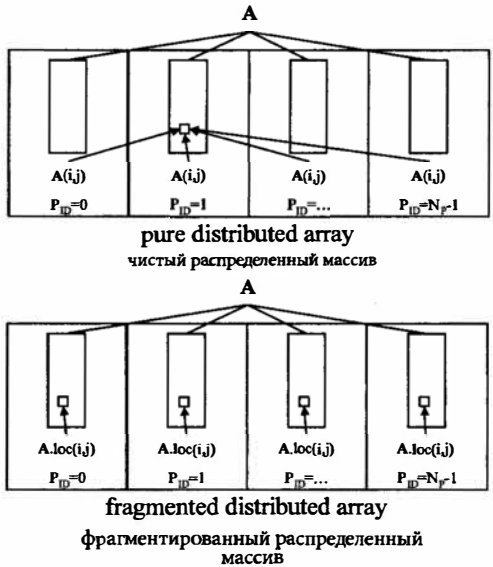


Рисунок 5.1. Сравнительная характеристика чистых и фрагментированных распределенных массивов

Верхняя половина рисунка иллюстрирует чистые распределенные массивы. Матрица A распределяется среди N_p процессоров. К элементу *i, j* обращаются все процессоры. В чистых распределенных массивах индекс *i, j* является глобальным индексом, который обращается к одному и тому же элементу на всех процессорах (на $P_{ID} = 0$ этот элемент является локальным; на всех других P_{ID} (он удаленный). Нижняя половина рисунка иллюстрирует фрагментированные распределенные массивы. Здесь каждый P_{ID} обращается к элементу *i, j*, локальному для данного P_{ID}; таким образом, каждый P_{ID} обращается к разным элементам глобальной матрицы

Библиотеки распределенных массивов, используемые в нашей книге, поддерживают обе модели – на основе чистых и фрагментированных распределенных массивов – и позволяют комбинировать метод распределенных массивов с методом прямого обмена сообщениями для оптимизации производительности. Обе модели программирования как с чистыми, так и с фрагментированными распределенными массивами, используют одну и ту же структуру распределенного массива данных, позволяющую преобразования между моделями. В то время как библиотека использует обмен сообщениями в библиотечных процедурах, обычный пользователь не обязан явно включать сообщения в код программы.

5.3. Интерфейс и реализация архитектуры распределенных массивов MATLAB

Основной трудностью при реализации библиотеки параллельных вычислений является уравновешивание противоречивых целей, к которым относится простота использова-

ния, высокая производительность и простота реализации. В отношении распределенных массивов в MATLAB, каждую их этих целей можно определить и измерить (таблица 5.1); в главе 6 данные характеристики будут представлены более подробно. Характеристики производительности являются типичными из числа тех, которые используются в сообществе специалистов по высокопроизводительным вычислениям, и, прежде всего, рассматривают накладные расходы на память и вычисления программ, написанных с использованием MATLAB, относительно последовательных программ, написанных в MATLAB, и параллельных программ, написанных на языке C с использованием MPI. Характеристики для простоты использования и реализации взяты у разработчиков программного обеспечения (см. [Johnson 2004, Johnson 2005] и [Kepner 2004] и их работы) и рассматривают размер кода, объем работы программиста и необходимую квалификацию. Эти метрики содержат субъективный элемент; тем не менее, они являются полезными инструментами для оценки прогресса в отношении целей программирования. Далее в разделе пойдет речь об определенных вариантах выбора при реализации библиотеки распределенных массивов для достижения указанных целей.

5.4. Карты и распределения

Первым шагом в написании параллельной программы является получение функционально корректной последовательной программы. Для перехода от последовательной программы к параллельной пользователь должен дополнить код новыми конструкциями. В программировании на основе распределенных массивов к данному процессу применен подход «separation-of-concerns» (разделение ответственности), который делает функциональность программы ортогональной по отношению к отображению ее на параллельную архитектуру. Последовательная программа трансформируется в параллельную путем добавления карт размещения к массивам, чтобы можно было создать распределенные массивы. Карты содержат информацию только о том, каким образом массив разбивается на множество P_{ID}, и добавление карты не должно менять функциональной корректности программы. Карта состоит из решетки, задающей принцип разделения каждого измерения; распределения, которое выбирает разбиение по блочному, циклическому или блочно-циклическому принципу; списка идентификаторов P_{ID}, определяющих, какие P_{ID} фактически хранят данные.

Концепция использования карт размещения для описания распределения массивов имеет давнюю историю. Идеи карт для параллельного размещения в MATLAB взяты главным образом у специалистов в области языка High Performanve Fortran (HPF) [Loveman 1993, Zosel 1993], библиотеки пространственно–временной адаптивной обработки Линкольнской лаборатории Массачусетского технологического института (STAPL) [DeLuca et al. 1997] (MIT Lincoln Laboratory Space–Time Adaptive Processing Library) и параллельной векторной библиотеки (PARALLEL Vector Library PVL) [Lebak et al. 2005]. Распределенные массивы MATLAB явно поддерживают параллелизм по данным; однако неявный параллелизм по задачам может быть реализован посредством тщательного распределения массивов данных.

Структура карты распределенного массива определяется тремя компонентами: (1) описание решетки, (2) описание распределения и (3) список P_{ID}. Описание решетки совместно со списком P_{ID} служат для описания места распределения объекта данных, в то

время как распределение описывает, каким образом распределен объект. В зависимости от нижележащей реализации распределенные массивы MATLAB могут поддерживать любую комбинацию блочно-циклических распределений. Прикладной программный интерфейс (API), определяющий эти распределения, показан на рисунке 5.2.

Таблица 5.1. Цели реализации распределенных массивов

Характеристики определяются для каждой из целей создания библиотеки распределенных массивов высокого уровня: простота использования, производительность и простота реализации. Эти характеристики приводят к использованию специальных подходов для представления целей в виде измеряемых величин

Цель	Простота использования
Метрики	– время, требующееся пользователю для создания эффективного параллельного кода из последовательного кода; – часть последовательного кода, которую нужно модифицировать; – специальные знания, необходимые для достижения хорошей производительности
Метод	– отделение кодирования функциональности от отображения на параллельную ерхитектуру; – абстрагирование передачи сообщений на уровень, не видимый пользователю; – обеспечение простоты предстааления простых (массово) параллельных программ; – предоставление простого механизма для глобального включения и выключения конструктивных элементов pMATLAB; – обеспечение обратной совместимости с последовательным MATLAB; – предоставление четко определенного и повторимого процесса для перехода от последовательного кода программы к пареллельному
Цель	**Высокая производительность**
Метрики	– накладные расходы по памяти и времени исполнения по сравнению с последовательным MATLAB, нижележащей библиотекой коммуникации и тестами производительности, написанными на C+MPI;
Метод	– использование последовательных процедур MATLAB, где это возможно (даже если это означает небольшое увеличение пользовательского кода); – минимизация использования перегруженных функций, производительность которых зависит от того, как распределены массивы по картам размещения; – предоставление простого механизма для использования коммуникации нижнего уровня в случае необходимости

Цель	Простота реализации
Метрики	– время для реализации эффективной параллельной библиотеки; – размер кода библиотеки; – количество объектов; – количество перегруженных функций; – полнота покрытия тестов на производительность и функциональность
Метод	– применение многослойной реализации, которая отделяет математический аспект от коммуникации; – использование понятной концепции разделенного глобального адресного пространства (Partitioned global address space – PGAS) и конструкций перераспределенных данных; – минимизация использования перегруженных функций; – развитие реализации на «чистом» MATLAB для минимизации размера кода и получения максимальной переносимости

Блочное распределение является типичным распределением, используемым по умолчанию, которое может определяться простой передачей конструктору карты измерения, подлежащего распределению. Циклические и блочно-циклические распределения требуют от пользователя предоставления большего количества информации. Распределения могут быть описаны для каждого измерения, и каждое измерение потенциально может иметь отличную схему распределения. Кроме того, если указано только одно распределение, а решетка определяет, что имеет место распределение более одного измерения, данное распределение применяется также к каждому измерению.

Некоторым приложениям, в особенности приложениям обработки изображений, требуется перекрытие данных или копирование строк или столбцов данных по соседним экземплярам P_{ID}. Эта возможность также поддерживается через интерфейс карты. Если требуется перекрытие, это указывается дополнительными аргументами. На рисунке 5.2 четвертый аргумент означает, что имеется 0 перекрытий строк и 1 столбец перекрытия между столбцами. Перекрытие может быть определено для любого измерения, и оно не обязано быть одинаковым по остальным измерениям.

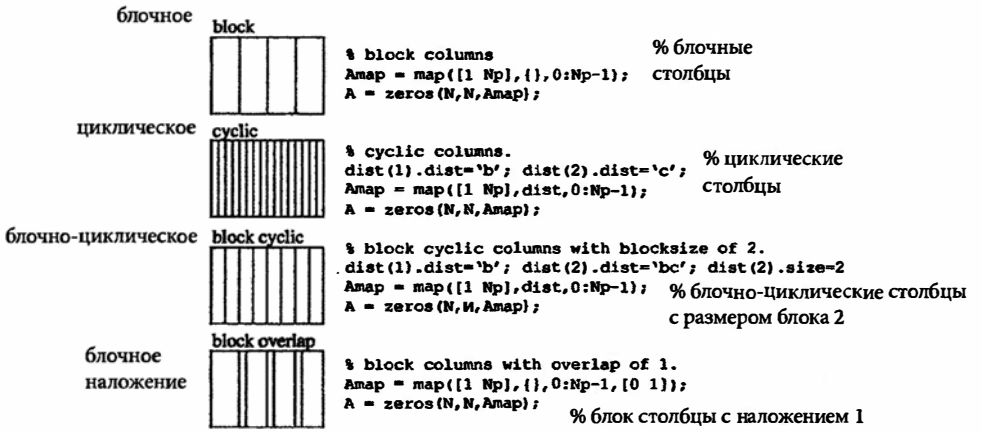


Рисунок 5.2. Блочно-циклические распределения

Блочное распределение разделяет объект равномерно между доступными P_{ID}. Циклическое распределение помещает по одному элементу на каждый имеющийся P_{ID} и затем повторяется. Блочно-циклическое распределение помещает определенное количество элементов на каждый имеющийся P_{ID} и затем повторяется. Перекрытие позволяет граничным элементам присутствовать на соседних экземплярах P_{ID}

Карты размещения представляют собой новую конструкцию и потенциально ухудшают простоту программирования, но вместе с тем они имеют значительное преимущество над методами «начальник/подчиненный» и «обмен сообщениями». В частности, карты поддаются масштабированию, позволяют создавать оптимальное распределение для различных алгоритмов и поддерживают конвейерную обработку данных.

Карты распределения являются масштабируемыми как в отношении размера данных, так и в отношении *Np*. Применение карт позволяет пользователю отделять задачу по распределению приложения от задачи написания приложения. Различные наборы карт не требуют внесения изменений в код приложения. В частности, распределение данных и *Np* может быть изменено без внесения изменений в алгоритм. Отделение распределения программы от функционального программирования является важным методом реализации в параллельной библиотеке MATLAB.

Карты облегчают процесс определения различных распределений для поддержки различных алгоритмов. Для множества конкретных вычислений существуют оптимальные или предлагаемые распределения. Например, операции умножения матриц – наиболее эффективны на процессорных решетках, которые являются транспонированием друг друга. Операции быстрого преобразования Фурье (БПФ) по столбцам и строкам становятся массово-параллельными, если измерение, по которому разбивается массив, соответствует измерению, по которому выполняется БПФ.

Возможно, наиболее сильный аспект концепции карт размещения лежит в ее применимости к широкому кругу ситуаций. Карты позволяют объявлять простые параллельные программы простым способом, в то же время поддерживая довольно сложные приложения. Например, как говорилось в главе 4, карты позволяют внедрять конвейерную обработку в вычисления, таким образом поддерживая неявный параллелизм по

задачам. Параллельно-конвейерный метод широко применяется для скрытия задержек при коммуникации между этапами. Для внедрения конвейерного подхода можно произвести следующие небольшие изменения в картах размещения (см. Алгоритм 4.3), при этом первый набор P_{ID} ($x_1 : C^{Nr \times Ns \times P_{(1)}(Np)}$) выполняет первую часть вычислений, а второй набор P_{ID} ($x_2 : C^{Nr \times Ns \times P_{(2)}(Np)}$) выполняет вторую часть и т. д. Когда P_{ID} встречается с такой картой, сначала выполняется проверка на предмет, имеет ли он данные, которые будет обрабатывать. Если P_{ID} не обладает данными, он переходит к следующей строке. В случае вышеописанных преобразований первый набор P_{ID} будет просто выполнять первый шаг вычислений, отправлять данные второму набору P_{ID}, пропускать остальные этапы и переходить к обработке следующего набора данных. Таким же образом второй набор P_{ID} пропустит первый шаг вычислений, получит данные от первого набора экземпляров P_{ID} и выполнит второй шаг вычислений.

5.5. Функции поддержки параллелизма

Каждая реализация распределенных массивов должна предоставлять набор функций для управления и работы с распределенными массивами, не имеющими последовательных эквивалентов. Набор функций поддержки параллельных распределенных массивов приводится в таблице 5.2. Эти функции позволяют пользователю собирать данные на одном или нескольких P_{ID}, определять, какие глобальные индексы являются локальными для каких P_{ID}, и получать данные из локальной части распределенного массива или помещать их в нее. Этот набор функций относительно небольшой и требует внесения минимального количества изменений в исходный пользовательский код. Эти функции, в дополнение к определениям карт размещения, представляют основу синтаксиса распределенных массивов пакета MATLAB. С целью поддержки описанного процесса разработки, все эти функции были перегружены для работы также на последовательных массивах MATLAB так, чтобы код продолжал работать даже при отключенных картах размещения.

Существует масса различных возможных карт распределения, которые могут поддерживаться библиотекой распределенных массивов. Ограничение библиотеки до N-размерных блочно-циклических распределений с перекрытиями тем не менее оставляет большое количество возможностей. Виды этих распределений, а также их запись и синтаксис представлены в таблице 5.3

5.5.1. Циклические распределения и балансирование параллельной нагрузки

Из множества видов распределений, описанных в предыдущем разделе, наиболее широко используемыми являются одномерные блочные распределения. Удивительно, но на втором месте по популярности среди пользователей идет одномерное циклическое распределение.

Циклические распределения являются одним из наиболее полезных видов распределений, при помощи которых достигается рост производительности массово-параллельных вычислений. В частности, циклические распределения полезны при решении широко распространенной проблемы балансирования параллельной нагрузки, которая относится к вопросу равномерности распределения работы между различными P_{ID}. Например, если некоторое вычисление распределятся по Np процессо-

рам, пусть $T(P_{ID})$ – время, которое требуется каждому процессору для завершения своей части работы. В идеале время должно быть примерно одинаковым для всех процессоров, и оно же является временем выполнения всего вычисления. Однако общее время на *Np* процессорах фактически составляет

$$T(N_P) = \max_{P_{ID}} T(P_{ID}).$$

Таблица 5.2. Функции поддержки параллельных распределенных массивов

Набор функций без последовательных эквивалентов, который должен быть предоставлен библиотекой распределенных массивов

Математич. запись	Функция	Описание
P_{ID}	Pid	Идентификатор каждого экземпляра программы, выполняющего программу в настоящий момент
N_P	Np	Количествово экземпляров MATLAB, выполняющих программу в настоящий момент
P_0		Создает карту
B: $R^{P(N)xN}$	map	Создает распределенный массив
A*loc*	zeros	Возвращает локальную часть распределенного массива
arg(A).loc	local	Возвращает глобальные индексы, локельные по отношению к текущему P_{ID}
A = B	global_ind	
A = A	=	Перераспределяет данные от одного распределенного массива другому
Synch(A)	agg	Собирает части распределенного массива в один или более P_{ID}
	synch	Синхронизирует данные в распределенном массиве с перекрытием

Таким образом, если одному из процессоров требуется намного больше времени, чем другим, это напрямую повлияет на общее время выполнения вычисления. Когда такое происходит, нагрузка по всем процессорам распределяется неравномерно. Такая неравномерность в общем виде определяется так:

$$\text{неравномерность нагрузки} = \max_{P_{ID}} T(P_{ID}) / \min_{P_{ID}} T(P_{ID}).$$

В идеальном случае данное значение очень близко к 1. Чем больше неравномерность нагрузки, тем большее количество процессоров простаивает.

Одной из самых распространенных причин неравномерности нагрузки является ситуация, когда время выполнения программы зависит от значения входного параметра. Например, предположим, что вычисление должно повториться по каждому из *N* значений в векторе входных параметров **x**. Пусть $T(\mathbf{x}(i))$ – время выполнения вычисления с *i*-м входным параметром **X** (*i*). При наличии корреляций в этих временных показателях с индексом параметра *i* наблюдается неравномерность нагрузки. Например, предположим, что существует небольшое линейное увеличение времени выпол-

$$T(\mathrm{x}(i+1)) = T(\mathrm{x}(i)) + e$$

или

$$T(\mathrm{x}(i)) = T(\mathrm{x}(1)) + ie.$$

Если входной параметр распределен между процессорами при помощи стандартного блочного распределения (например, $\mathrm{x}: \mathrm{R}^{P(N)}$), очевидно, что процессору $P_{ID} = Np$ потребуется больше времени для выполнения своей работы, чем процессору $P_{ID} = 1$.

Результирующая неравномерность нагрузки составит (в предположении, что $T(\mathrm{x}(1) = 1)$

$$\text{неравномерность нагрузки} = \frac{\sum_{i=N-N/Np}^{i=N} 1 + i\varepsilon}{\sum_{i=1}^{i=N/Np} 1 + i\varepsilon} \approx \frac{1 + \varepsilon(N - N/2Np)}{1 + \varepsilon N/2Np} \approx 1 + \varepsilon N$$

Для типичных значений N = 1000, e = 0,01 и $Np = 50$ результирующая неравномерность нагрузки составит ~ 10.

Таблица 5.3. Типы распределения распределенных массивов

Обозначения и синтаксис для описания различных типов распределения распределенных массивов

Математич. Запись	Функция	Преобразование
$P(N_1) \times N_2$	`map([Np 1],{},0:Np-1)`	1-мерное блочное в 1-м измерении
$N_1 \times P(N_2)$	`map([1 Np],{},0:Np-1)`	1-мерное блочное в 2-м измерении
$P(N_1) \times P(N_2)$	`map([Np/2 2],{},0:Np-1)`	2-мерное блочное в 1- и 2-м измерениях
$P(N_1 \times N_2)$	`map([Np/2 2],,0:Np-1)`	2-мерное блочное в 1- и 2-м измерениях Явное блочное
$P_b(N)$	`map([Np 1],{},0:Np-1)`	Явное циклическое
$P_c(N)$	`dist.dist='c'` `map([Np 1],dist,0:Np-1)`	Блочно-циклическое [B =1 тождественно P_c]
$P_{bc(B)}(N)$	`dist.dist='bc'; dist.size=B;` `map([Np 1],dist,0:Np-1)`	
$P_{ov(B)}(N)$	`map([Np 1],,0:Np-1,[0 B])`	Перекрытие с размером B
$P_S(N)$	`map([Np/2 2],{},S)`	Блочное по множеству номеров P_{ID} S

Простым методом выравнивания данной неравномерности нагрузки является разрушение корреляции между временем выполнения и индексом параметра при по-

мощи циклического распределения (например, x : R $^{P_c(N)}$). В данном случае неравномерность нагрузки составит

$$\text{неравномерность нагрузки} \approx 1 + eNp,$$

что для типичных значений превращается в неравномерность нагрузки ~ 1,05.

Пример: вычисление в программе Mandelbrot с использованием циклического распределения

Чтобы получить представление о неравномерности нагрузки на практике, выполните повторно программу `Mandelbrot` (см. главу 2) на четырех процессорах. Запустите MATLAB и перейдите в директорию `Examples/Mandelbrot` (т. е. наберите: `cd Examples/Mandelbrot`). В командную строку введите

```
eval(pRUN ('pMandelbrot',4,{}))
```

При этом программа выполняется на вашем локальном процессоре и выдает временные показатели, аналогичные нижеприведенным

```
Compute Time /время вычисления (sec/c)                    = 6.0421
Launch/Comm Time/ время запуска + коммуникации  (sec/c)   = 14.2262
```

Кроме того, вывод от других экземпляров MATLAB можно посмотреть, если ввести в командной строке MATLAB следующее

```
!type MatMPI\*.out
```

В системах UNIX это эквивалентно написанию

```
!more MatMPI/*.out
```

Выполнение на параллельной вычислительной машине выдаст временные параметры, сходные с параметрами в первой строке таблицы 5.4. Эти результаты показывают, что экземпляры P_{ID} 1 и 2 работают почти в два раза дольше, чем P_{ID} 0 и 3. Общим ускорением вычисления будет минимальное ускорение, достигнутое по всем P_{ID}, которое приблизительно равно 3.1.

Для завершения сбора временных характеристик отредактируйте файл `Examples/Mandelbrot/pMandelbrot.m` и раскомментируйте строки

```
dist(1) .dist = 'c'; dist(2).dist ='b';
Wmap = map([Np 1J ,dist,0:Np-1);
```

При этом распределение переключится с блочного на циклическое. Теперь снова выполните вычисление, набрав

```
eval(pRUN ('pMandelbrot',4,{}))
```

Выполнение на параллельной машине выдаст временные характеристики, аналогичные второй строке в таблице 5.4. Данные результаты показывают, что всем процессорам требуется приблизительно одинаковое время, и это время находится в промежутке между максимальным и минимальным временем в случае блочного распределения. Таким образом, общее время выполнения будет меньше с использованием циклического распределения, так как оно имеет более равномерную нагрузку.

Важное примечание: последний аргумент команды pRUN определяет, на какой параллельной вычислительной машине будет выполняться программа. Может возникнуть необходимость в получении надлежащих установочных параметров у системного администратора; для получения более подробной информации см. веб-сайт книги (http://www.siam.org/KepnerBook). Установка последнего аргумента значением {} приводит к выполнению множества копий программы на вашей локальной машине и получению значений производительности, отличных от указанных в таблице 5.4.

Повышение вычислительной производительности программы Mandelbrot при использовании циклического распределения типично для многих приложений, для которых характерна неравномерность нагрузки. В случае с программой Mandelbrot алгоритм выполняет повторяющиеся вычисления для ряда входных параметров Z, которые покрывают область комплексной плоскости. Для каждого параметра вычисление повторяется, пока он не достигнет определенного значения. Значения, близкие к центру области, требуют большего количества шагов для достижения данного критерия, чем значения, близкие к краю. В результате блочного распределения большую часть работы выполняют средние P_{ID} (1 и 2), а меньшую часть работы выполняют крайние P_{ID} (0 и 3). При циклическом распределении работа между процессорами распределяется равномерно.

Таблица 5.4. Неравномерность нагрузки в программе Mandelbrot

Относительное время вычислений и коэффициент ускорения вычислений для каждого P_{ID} при $Np = 4$ при выполнении программы Mandelbrot на многоузловой системе. Результаты показаны с использованием блочного распределения и циклического распределения и нормализованы по времени выполнения при $Np = 1$

P_{ID}	0	1	2	3
Блочное				
Относительное время вычислений	0.15	0.31	0.31	0.15
Коэффициент ускорения вычислений	6.33	3.16	3.19	6.33
Циклическое				
Относительное время вычислений	0.24	0.24	0.24	0.24
Коэффициент ускорения вычислений	4.21	4.20	4.23	4.20

5.6. Выбор в пользу параллелизма или локальности

Ключевым начальным шагом при разработке параллельного алгоритма является определение набора операций, которые можно выполнить параллельно. В частности, параллельные операции – это набор операций, не зависящих от порядка, в котором они выполняются. К счастью, выявление параллелизма в программе на MATLAB не вызывает затруднений, так как большинство программ основаны на массивах, которые параллельны по своей природе. Любая операция на одном или нескольких массивах, которая приводит к созданию нового массива, может выполняться параллельно по всем элементам в результирующем массиве. Фактически параллелизм в большинстве программ на MATLAB носит повсеместный характер; таким образом, написание программ, которые выполняются параллельно, не представляет сложности. Настоящую трудность представляет написание параллельных программ, которые работают быстро. Ключевым моментом для написания такого рода программ является выявление параллелизма программы, результатом чего также должна стать высокая степень локальности. Такая программа способна выполнять параллельные операции с минимальным объемом данных, приходящих от других P_{ID} (см. рисунок 5.3).

Модель программирования на основе распределенных массивов является уникальной среди всех моделей параллельного программирования в связи с тем, что здесь с самого начала достигается локальность посредством простого действия по созданию распределенного массива. После этого параллелизм в задаче может быть получен на основе локальности, что и приводит к созданию эффективной параллельной программы. Напротив, другие модели параллельного программирования обычно вынуждают программиста сначала решать проблему параллелизма, и уже потом иметь дело с более сложной задачей локальности. Основной смысл использования функций поддержки параллелизма `local` и `global_ind` заключается в возможности извлечения локальности и параллелизма из распределенных массивов.

Для иллюстрации данных концепций рассмотрим программу `AddOne`, описанную в главе 1. Сущность данной программы заключается в операции

$$\mathbf{Y} = \mathbf{X} + 1,$$

где $X, Y : R^{N \times N}$. Данная программа имеет N^2 операций, которые можно выполнить параллельно. Размер параллелизма часто называют степенью параллелизма. В большинстве случаев параллельное распределение **X** и **Y** отразит данный параллелизм. Если $X, Y : R^{P(N) \times N}$, то результирующий алгоритм обнаружит N степеней параллелизма (при условии $N > Np$). Больше степеней параллелизма можно получить, если оба измерения массива распределены. Например, если $X, Y : R^{P(N) \times P(N)}$ или $X, Y : R^{P(N \times N)}$ (что эквивалентно), получим N^2 степеней параллелизма.

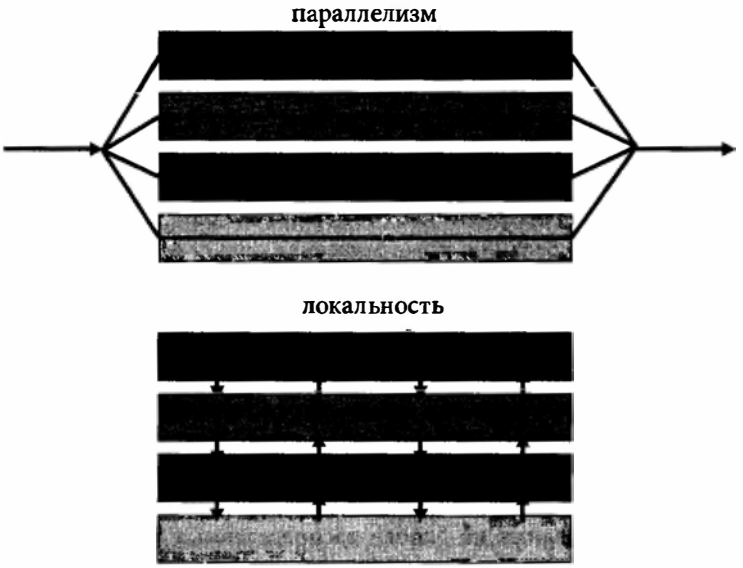


Рисунок 5.3. Параллелизм и локальность

Параллелизм описывает, сколько операций можно выполнить параллельно в любой момент времени. Локальность показывает, какое количество данных необходимо переместить для завершения параллельных операций

Локальность программы можно измерить, используя отношение «вычисление–коммуникация» ***W/D***, где ***W*** – количество совершенных операций, ***D*** – общий объем данных, перемещаемых между P_{ID}. Программа AddOne будет иметь высокую степень локальности, если **X** и **Y** распределены по картам одинаково, так как нет необходимости в передаче данных. Результирующее отношение «вычисление–коммуникация» составит

$$W / D = N^2 / 0 \rightarrow \infty.$$

С другой стороны, если **X** : $R^{P(N)XN}$, а **Y** : $R^{NXP(N)}$, то для завершения практически каждой операции необходимо будет выполнять обмен информацией. Результирующее отношение «вычисление–коммуникация» составит

$$W / D = N^2 /(N^2(N_P - 1)/N_P) = N_P(N_P - 1) \approx 1.$$

Для демонстрации параллелизма версия программы AddOne на MATLAB может быть написана явно следующим образом:

```
x = zeros(N,N); Y
Y = zeros(N,N);
for i=1:N
    for j=1:N
```

```
            Y(i,j) = X(i,j) + 1;
        end
    end
```

Чтобы преобразовать вышеуказанный код в параллельный, необходимо просто применить карту к конструкторам **X** и **Y**:

```
XYmap = map([Np 1] ,{},0:Np-1);
X = zeros(N,N,XYmap);
Y = zeros(N,N,XYmap);
for i=1:N
    for j=1:N
            Y(i,j) X(i,j) + 1;
    end
end
```

Очевидно, что полученная в результате программа имеет хорошую локальность, так как **X** и **Y** используют одну и ту же карту. Однако отношение между параллелизмом и локальностью выражено неявно. Несколько более длинный способ написания этой программы, в явном виде гарантирующий локальность, выглядит так:

```
XYmap = map([Np 1] ,{},0:Np-1);
Xloc = local(zeros(N,N,XYmap»;
Yloc = local(zeros(N,N,XYmap»; for
i=1:size(Xloc,1)
    for j=1:size(Xloc,2)
            Yloc(i,j)= Xloc(i,j) + 1;
    end
end
```

В данном случае связь между параллелизмом и локальностью явная. Как правило, написание программы таким способом обеспечивает наивысшую производительность.

5.7. Простота реализации

Достижение простоты использования и высокой производительности – это цели, хорошо понятные разработчикам параллельных вычислений. К сожалению, реализация этих целей в библиотеке промежуточного уровня требует больших затрат. Типичная библиотека распределенных массивов на C++ может достигать 50 000 строк кода и потребовать нескольких человеко-лет для реализации. Библиотеки распределенных массивов, реализованные для MATLAB, применяют некоторые стратегии для уменьшения этих затрат. Общий смысл данных стратегий заключается в нахождении минимального набора возможностей, которые все еще будут позволять разработчикам писать эффективные программы.

Одним из ключевых моментов при реализации библиотеки распределенных массивов является выбор поддерживаемых распределений данных. С одной (крайней)

точки зрения, можно заявлять, что большинство разработчиков будут довольны одномерными блочными распределениями. С другой (крайней) точки зрения, можно найти приложения, которые требуют действительно произвольных распределений индексов массивов по экземплярам P_{ID}. В настоящее время реализации распределенных массивов в MATLAB выбрали наборы распределений, лежащие между двумя этими крайностями. Как правило, эти распределения поддерживают N-мерные блочно-циклические распределения с перекрытием, так как проблема перераспределения между двумя такими распределениями (см. раздел 3.3) была многократно решена при помощи различных технологий параллельных вычислений.

Перегрузка оператора MATLAB `subsasgn` (индексированное присваивание) поддерживает перераспределение данных между массивами с использованием = операции. Следующий вопрос состоит в том, какие другие функции нужно поддерживать и для каких распределений. В таблице 5.5 перечислены различные уровни поддержки распределенных массивов. Способность работать с локальной частью распределенного массива и его индексами демонстрировалась неоднократно. Значительная трудность заключается в перегрузке всех математических функций в библиотеке, чтобы обеспечить эффективность работы с каждой комбинацией входных распределений. Как уже обсуждалось в разделе 2.2, данное свойство очень трудно реализовать, и в нем нет необходимости, если разработчики готовы работать в менее красивом стиле написания кода программ, используемом при применении фрагментированных распределенных массивов. Таким образом, большинство библиотек распределенных массивов MATLAB предоставляют широкий набор типов распределений данных, но сравнительно скромный набор перегруженных функций, главным образом сконцентрированных на функциях конструирования массива, функциях поддержки индексов массивов и различных поэлементных операций (`+, --, .*, ./`).

Последним решением, которое нужно принять при реализации написания библиотеки распределенных массивов, является выбор языка. В этом отношении MATLAB уникален, так как он является языком высокого уровня на основе массивов с достаточным языковым абстрагированием, позволяющим реализовывать библиотеки распределенных массивов на MATLAB с минимальной зависимостью от других языков. При данном подходе минимизируется размер кода, а переносимость увеличивается до максимума. Например, можно реализовать большинство функциональных возможностей распределенных массивов с помощью кода всего лишь из 3000 строк и представления только двух новых объектов (карты и распределенные массивы). Такой метод гарантирует выполнение кода везде, где бы ни работал MATLAB.

5.8. Особенности реализации

В данном разделе подробно рассматривается реализация распределенных массивов в MATLAB. Основа реализации библиотеки распределенных массивов строится на концепциях параллельной векторной библиотеки (PVL) [Lebak et al. 2005] и StarP [Choy & Edelman 1999] и использовании концепций, примененных в MPI [Kepner & Ahalt 2004] для коммуникационного слоя. На рисунке 5.4 изображена многослойная архитектура параллельной библиотеки. При многослойной архитектуре библиотека распределенных массивов реализует распределенные конструкции, такие как распределенные ма-

трицы и многомерные массивы. Кроме того, библиотека обеспечивает параллельную реализацию заданного набора функций, таких как перераспределение, быстрое преобразование Фурье (БПФ) и перемножение матриц. Однако программисту обычно легче создать параллельную реализацию функции под конкретные размеры задачи и распределение данных, чем разрабатывать общие параллельные реализации функций, которые обеспечат хорошую производительность для всех распределений данных и их размеров. Библиотека распределенных массивов использует подход «параллелизм–через–полиморфизм», как описано в [Choy & Edelman 1999]. Мономорфные языки требуют, чтобы каждая переменная относилась только к одному типу; с другой стороны, в полиморфных языках переменные могут быть разных типов, а полиморфные функции могут выполняться на различных типах переменных [Abadi et al. 1991].

Таблица 5.5. Уровни реализации

Уровни поддержки параллелизма для данных и функций. Примечание: поддержка распределения данных включает поддержку перекрытия в любом распределенном измерении. Уровень `Data4/Op1` успешно реализовывался много раз. Уровень `Data1/Op2` возможен, но пока не был никем продемонстрирован

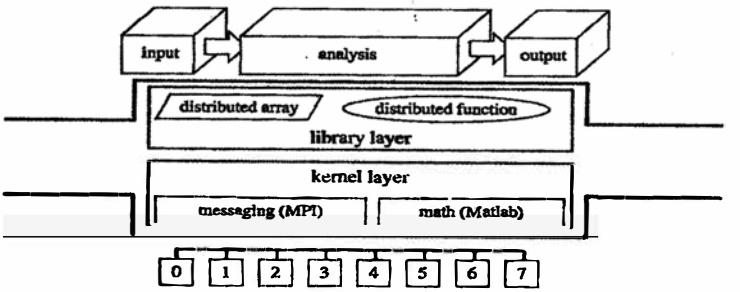
Уровень данных	Описание поддержки
Data0	Распределение данных не поддерживается (непараллельная реализация);
Data1	Возможно блочное распределение данных в одном измерении;
Data2	Возможно блочное распределение данных в двух измерениях;
Data3	Возможно блочное распределение данных в любом и во всех измерениях;
Data4	Возможно блочное или циклическое распределение данных в любом и во всех измерениях;
Уровень операций	**Описание поддержки**
Op0	Не поддерживаются операции распределения (непараллельная реализация);
Op1	Распределенные операции присваивания, приема и отправки, поддержка получения данных и индексов локальных данных от распределенного объекта;
Op2	Поддержка распределенных операций (реализация должна определять конкретные виды операций);

Принцип полиморфизма свойственен языку MATLAB: не нужно определять типы переменных, типы переменных можно изменять во время выполнения программы; и многие функции способны работать со многими типами данных, например, `double`, `single`, `complex` и т. д.

В библиотеке распределенных массивов по сравнению с этим принципом был сделан еще один шаг вперед. Полиморфизм применяется путем внедрения объекта карты. Объекты карт принадлежат к классу языка MATLAB и создаются посредством задания описания решетки, описания распределения и перечня процессоров, как уже было упомянуто в предыдущих разделах. Объект карты после этого можно передавать в конструктор MATLAB, такой как `rand`, `zeros` или `ones`. Конструкторы пере-

гружены, и когда объект передается в конструктор, библиотека создает переменную распределенного типа, т. е. распределенный массив.

Как говорилось выше, подмножество функций, таких как `plus`, `minus`, `fft`, `mtimes` и поэлементные операции, перегружены для работы на объектах распределенных массивов. Если использовать модель программирования на основе чистых распределенных массивов и перегруженную функцию, объект распределенного массива может использоваться так же, как и обычный массив. Функции, которые работают только на локальной части распределенного массива (поэлементные операции) просто совершают требуемые операции на локальном массиве, который относится к типу стандартных числовых массивов MATLAB, указанному при создании массива. Функции, требующие коммуникации, такие как перераспределение (или `subsasgn` в синтаксисе MATLAB), в качестве коммуникационного слоя используют метод обмена сообщениями.



Перевод
Application – приложение;
PARALLEL library – параллельная библиотека;
PARALLEL hardware – аппаратные средства для параллельных вычислений;
User interface – интерфейс пользователя;
Hardware interface – интерфейс аппаратных средств;
INput – входные данные;
Analysis – анализ;
Output – результат;
Distributed array – распределенный массив;
Distributed function – распределенная функция;
Library layer – слой библиотеки;
Kernel layer – слой ядра;
Messaging (MPI) – обмен сообщениями (MPI);
Math (MATLAB) – математика (MATLAB).

Рисунок 5.4. Многослойная архитектура параллельной библиотеки
Библиотека распределенных массивов реализует распределенные конструкции, такие как векторы, матрицы и многомерные массивы, а также параллельные алгоритмы, которые работают на этих конструкциях, такие как перераспределение, БПФ и перемножение матриц. Коммуникация типа точка–точка и последовательные функции реализуются в слое ядра и вызываются слоем библиотеки

Основной целью использования параллельного компьютера является улучшение временных показателей выполнения. Первым шагом к достижению высокой производительности является минимизация накладных расходов при использовании конструкций распределенных массивов по сравнению с их последовательными эквивалентами. К сожалению, в случае идеального «чистого» примера с распределенными массивами, в котором все требуемые функции перегружены, имеется тенденция к замедлению работы. Более того, нецелесообразно (и необязательно) обеспечивать оптимизированную реализацию около 8000 встроенных функций MATLAB для каждой комбинации распределения массивов. Вместо этого, гораздо проще будет переложить на программиста ответственность за используемый стиль составления кода, при котором применяются конструкции фрагментированных распределенных массивов. Этот стиль не такой красивый, но он обеспечивает строгую гарантию производительности. По большому счету, самые эффективные программы на основе распределенных массивов – это программы, использующие только распределенные массивы для осуществления коммуникации между процессорами.

5.8.1. Выполнение программы

Весь код распределенных массивов находится в рамках инфраструктуры общего кода (см. рисунок 5.5), предназначенного для инициализации библиотеки распределенных массивов (`pRUN`), определения количества выполняемых экземпляров MATLAB (*Np*), определения идентификатора локального процессора (P_{ID}) и деинициализации (финализации) библиотеки по завершении вычисления (`pRUN`). Программа распределенных массивов использует модель выполнения SPMD (одна программа, несколько потоков данных) [Flynn 1972]. Пользователь выполняет программу, использующую распределенные массивы, посредством вызова команды `pRUN` для запуска и инициализации экземпляров MATLAB, необходимых для параллельной работы. На рисунке 5.5 показан пример сценария `RUN.m` с использованием функции `pRUN` для запуска четырех копий сценария `dI0.m` (см. главу 2).

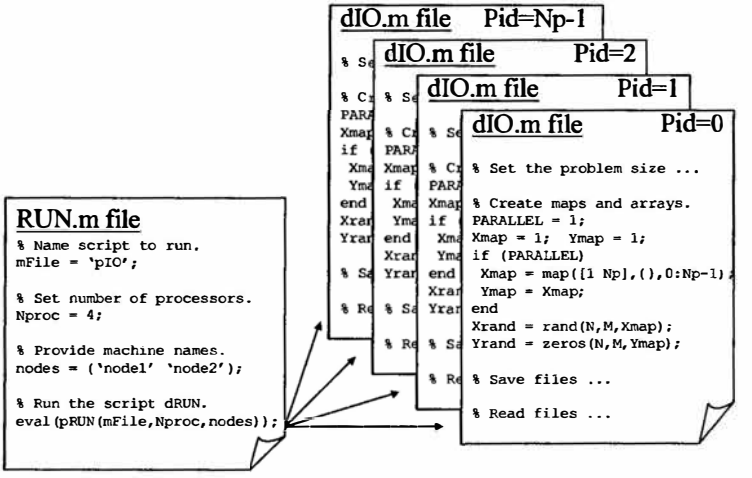


Рисунок 5.5. Инфраструктура выполнения

Программа распределенных массивов (dI0.m) запускается при помощи команды pRUN, показанной в файле RUN.m, которая устанавливает количество процессоров и определяет машины, на которых будет происходить выполнение. Сценарий pRUN.m запускает *Np* экземпляров MATLAB, каждый со своим P_{ID}. Внутри программы, использующей распределенные массивы, инициализируется среда распределенных массивов, после чего можно получить количество процессоров (*Np*) и локальный номер P_{ID}. Программа завершает работу, когда завершается сценарий dI0.m и выполняется автоматический выход из экземпляра MATLAB внутри команды pRUN

5.9. Перераспределение массивов

Пожалуй, основной операцией библиотеки распределенных массивов является перераспределение данных между двумя распределенными массивами. Алгоритмически это можно кратко записать как **B** = **A** с массивами типа **A** : $R^{P(N)\times N}$ и **B** : $R^{P(N)\times N}$. В виде кода это выглядит следующим образом:

```
Amap = map([Np 1J ,{},0:Np-1);
Bmap = map([1 NpJ ,{},0:Np-1);
A = rand(N,N,Amap);
B = zeros(N,N,Bmap);
B(:,:)=A;
```

Эффективным и общим методом перераспределения данных считается PITFALLS (Processor - процессорное, Indexed Tagged Family of Line Segments- теговое индексированное семейство линейных сегментов) [Ramaswamy & Banerjee 1995], которое представляет собой математическую модель распределения данных на основе восприятия каждого измерения массива в качестве отдельного семейства линейных сегментов. Кроме того, в [Ramaswamy & Banerjee 1995] приводится алгоритм для определения того, каким парам P_{ID} необходимо выполнить коммуникацию, если требуется перераспределение, и конкретно какие данные необходимо отправить.

Процедура перераспределения начинается с определения линейного сегмента (LS), который представляет собой набор смежных индексов в одном измерении массива. Для распределенного массива **LS** описывает набор индексов, которые являются локальными для определенного P_{ID}. Линейный сегмент **LS** может быть представлен парой чисел (l, r), где l является начальным индексом, а **r** – конечным. Вычисление перераспределения от одного распределенного массива другому включает в себя многократное нахождение области пересечения двух **LS:** $L_1 = (l_1, r_1)$ и $L_2 = (l_2, r_2)$ по формуле

$$L_1 \cap L_2 = (max(l_1, l_2), min(r_1, r_2)).$$

Если $max(l_1,l_2) > min(r_1,r_2)$, то область пересечения пуста.

Семейство линейных сегментов (FALLS) представляет собой последовательность равноотстоящих линейных сегментов и обозначается кортежем (l, r, S, N,), где N – количество линейных сегментов, а s – шаг между левыми границами двух соседних сегментов. Таким образом, i–ый линейный сегмент задается следующим образом:

$$L^i = (l,r)+i\times s.$$

На основе данного принципа структура PITFALLS определяется следующим кортежем:

$$PITFALLS = (l,r,s,n,d,Np),$$

где

l: начальный индекс,

r: конечный индекс,

s: расстояние между двумя последовательными левыми границами,

n: количество равноотстоящих одинаковых по размеру блоков элементов на каждый процессор,

d: шаг между индексами l, входящими в FALLS последующего процессора,

Np: количество процессоров.

Структура PITFALLS описывает набор равноотстоящих семейств FALLS на количестве процессоров Np. Семейство FALLS на P_{ID} задается так:

$$\text{FALLS}(P_{ID}) = (l + (P_{ID} - 1)\times d, r + (P_{ID} - 1)\times d, s, n).$$

Таким образом, i–ый линейный сегмент на P_{ID} задается так:

$$L^i(P_{ID}) = (\text{l}, \text{r}) + (P_{ID} - 1)\times d + i \times s.$$

Структуру PITFALLS можно без труда вычислить из карты распределенного массива и использовать для быстрого определения того, какие элементы массива принадлежат заданному процессору (и наоборот).

Алгоритм области пересечения PITFALLS используется для определения сообщений, необходимых для перераспределения между двумя распределенными массивами, и используется операцией sabsasgn. Алгоритм может применяться к каждому измерению массива, таким образом обеспечивая эффективное перераспределение массивов с произвольным числом измерений. Алгоритм пересекает структуры PITFALLS для каждого измерения. Если области пересечения во всех измерениях не являются пустыми, для каждого процессора вычисляются размер и содержание сообщений, подлежащих отправке,. В алгоритме области пересечения уменьшается количество необходимых вычислений посредством использования свойства повторяемости блочно-циклических распределений. Кроме того, структура PITFALLS также используется операцией обращения subsref, чтобы определить, какая часть распределенного массива какому процессору принадлежит. Подробное описание алгоритма и его эффективности см. в [Ramaswamy & Banerjee 1995].

5.10. Слой обмена сообщениями

Одним из главных преимуществ использования распределенных массивов является относительная простота описания очень сложных перераспределений данных между массивами. В результате требуется лишь небольшое количество базовых функций коммуникации для построения библиотеки распределенных массивов. На уровне пользователя требуются только две функции: SendMsg и RecvMsg. Эти функции строятся поверх небольшого количества функций MPI. Функции MPI, используемые библиотекой распределенных массивов, перечислены в таблице 5.6.

Таблица 5.6. Функции MPI

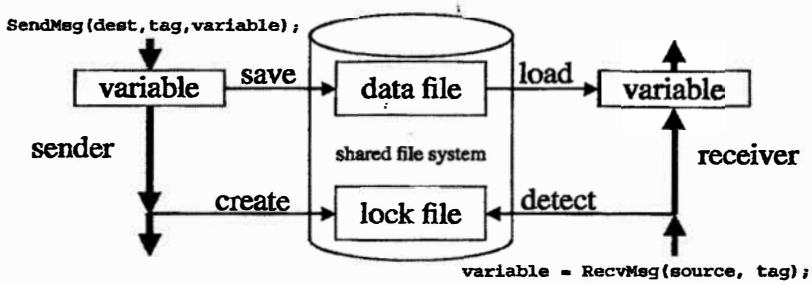
Основной набор функций MPI, необходимый библиотеке распределенного массива

Функция	Описание
MPI_Unit	Инициализация MPI
MPI_Comm_size	Получение кол–ва процессоров при коммуникации
MPI_Comm_rank	Получение ранга текущего процессора внутри коммуникатора
MPI_Send	Отправление сообщения процессору
MPI_Recv	Получение сообщения от процессора
MPI_Finalize	Финализация MPI

Небольшое число необходимых коммуникационных функций также позволяет использовать разнообразные протоколы коммуникации на уровне ниже, чем уровень библиотеки. В качестве экстремального примера можно привести MatlabMPI [Kerner & Ahalt 2004], где фактическая коммуникация осуществляется посредством файлового ввода/вывода (см. рисунок 5.6) через общую файловую систему. Преимущество данного подхода заключается в том, что библиотека MatlabMPI очень мала (~300 строк) и легко переносится. Однако такая переносимость имеет свой недостаток: если

производительность MatlabMPI сравнима с традиционными библиотеками MPI для больших сообщений, то задержка при пересылке маленьких сообщений значительно увеличивается (см. рисунок 5.8). Библиотека распределенных массивов, использующая MatlabMPI, для достижения синхронизации должна добавлять уникальный тэг к каждому посылаемому сообщению. В частности, при каждой процедуре коммуникации увеличивается глобальный тэг сообщения по всем задействованным P_{ID}. Также MatlabMPI предоставляет неограниченный буфер вследствие ее коммуникационной схемы на основе файлового ввода/вывода. Управление тэгами при помощи библиотеки распределенных массивов в сочетании с наличием неограниченных буферов предотвращает возникновение взаимоблокировок.

Производительность библиотеки сообщений в своей основе определяется в соответствии с размером сообщений по отношению к кривой задержке. Данная кривая показывает время, необходимое для передачи сообщения заданного размера от одного процессора к другому. Простой алгоритм измерения данного аспекта показан в Алгоритме 5.1.



Перевод
Sender – отправитель;
Variable – переменная;
Save – сохранить;
Create – создать;
Data file – файл данных;
Shared file system – разделяемая файловая система;
Lock file – файл блокировки;
Load – загрузить;
Detect – обнаружить;
Receiver – получатель.

Рисунок 5.6. Коммуникация на основе файлового ввода/вывода в библиотеке MatlabMPI

Для реализации коммуникации точка–точка MatlabMPI использует файловый ввод/вывод. Отправитель записывает переменные в буферный файл и затем записывает файл блокировки. Получатель ждет, пока не увидит файл блокировки, и затем считывает буферный файл

Алгоритм 5.1. Тестирование скорости сети

Отправляет сообщения разного размера между двумя процессорами и измеряет результаты по времени.

$\mathbf{t}_m : \mathbb{R}^{N_m} = \text{SPEEDTEST}(N_m)$

1 $P_{ID}^{send} = (P_{ID} - 1 \mod N_P) + 1$
2 $P_{ID}^{recv} = (P_{ID} + 1 \mod N_P) + 1$
3 **for** $i = 1 : N_m$,
4 **do**
5 $\mathbf{d}_{send}, \mathbf{d}_{recv} : \mathbb{R}^{2^i}$
6 CLOCKSTART()
7 SEND($\mathbf{d}_{send}, P_{ID}^{send}, i$)
8 $\mathbf{d}_{recv} = \text{RECEIVE}(P_{ID}^{recv}, i)$
9 $\mathbf{t}_m(i) = \text{CLOCKSTOP}()$

Соответствующая программа может быть выполнена после запуска MATLAB и перехода в директорию Examples/Speedtest (т. е. введите `cd Examples/Speedtest`). В командной строке наберите

```
eval(pRUN('pMandelbrot' ,2,{})).
```

Эта команда запускает программу для работы на двух процессорах. Так как программа `pSpeedtest` осуществляет только коммуникацию, она не будет выполняться только на одном процессоре. После завершения программы дополнительно наберите

```
load speedtest.1.mat
loglog(ByteSize, TotalTime)
loglog(ByteSize, Bandwidth).
```

В результате появятся графики, аналогичные изображенным на рисунке 5.7. Программа `pSpeedtest` выполняет несколько попыток серии измерений, и в результате на каждую попытку выдается одна кривая. График зависимости задержки сообщений от их размера показывает, сколько времени занимает пересылка сообщения заданного размера между двумя процессорами. Обратите внимание, что первая точка на графике располагается гораздо выше остальных. Она соответствует задержке вследствие ожидания первым процессором запуска остальных процессоров. Эту задержку иногда называют «сдвиг запуска». Время, соответствующее плоской части кривой задержки сообщений (при небольших сообщениях), означает задержку коммуникационной сети. График зависимости пропускной способности от размера сообщений показывает скорость передачи данных по сети. Пропускная способность, соответствующая плоской части кривой (при больших сообщениях), показывает полосу пропускания коммуникационной сети.

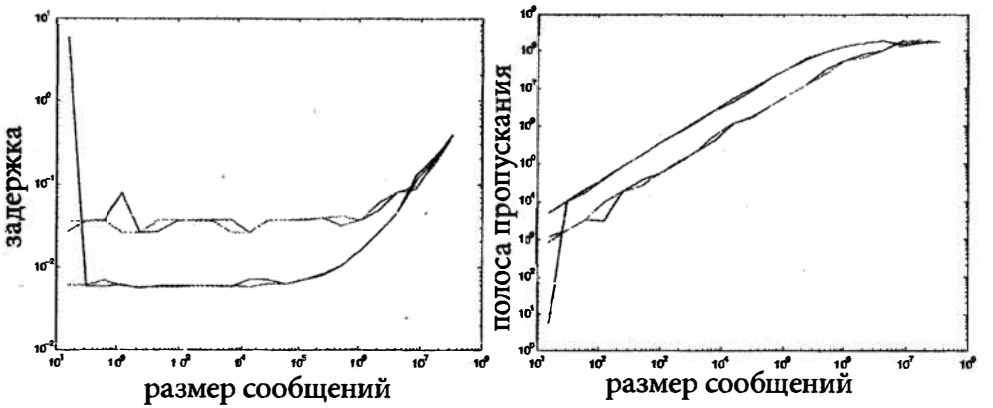
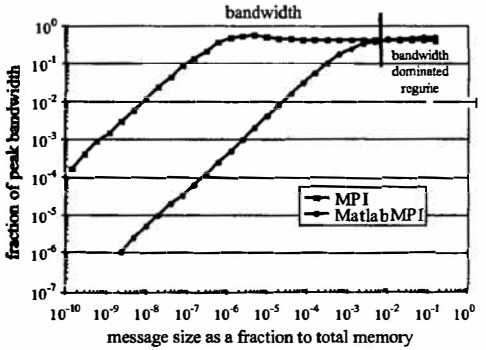
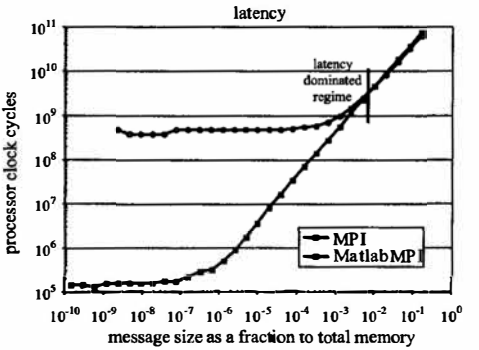


Рисунок 5.7. Результаты тестирования скорости сети

Программа `Speedtest` может выполняться с использованием различных коммуникационных библиотек нижнего уровня для сравнения их относительной производительности. На рисунке 5.8. показаны кривые задержки и пропускной способности для двух различных библиотек обмена сообщениями. Обе библиотеки имеют приблизительно одинаковую пропускную способность и будут иметь приблизительно одинаковую производительность при использовании в приложениях, в которых пересылается большое количество больших сообщений. Однако, очевидно, что одна из них характеризуется меньшей задержкой и хорошо подходит для пересылки сообщений небольшого размера

При создании библиотеки распределенных массивов важно обеспечить отсутствие значительного влияния на производительность накладных расходов, появляющихся вследствие использования библиотеки. В отношении библиотеки это означает, что производительность операций коммуникации с использованием перегруженного оператора `subsasgn` должна быть как можно более близкой эквивалентному коду обмена сообщениями без распределенных массивов. На рисунке 5.9 показана производительность операции перераспределения «каждый к каждому» (all-to-all) с использованием только обмена сообщениями, присваивания распределенных массивов и специальной функции `transpose_grid`. Эти результаты являются типичными для библиотеки распределенных массивов. Перераспределение с использованием **`B(:,:)=A`** обычно происходит в 2 раза медленнее, чем выполнение той же операции посредством явной записи всех операций `SendMsg` и `RecvMsg`. Эти накладные расходы обычно имеют место вследствие нескольких факторов. Во-первых, общее перераспределение должно поддерживать все возможные распределения и выполнять большое количество проверок по определению, куда какие данные необходимо отправить. Во-вторых, распределенные массивы обычно требуют хранения нескольких дополнительных копий переменных. В-третьих, трудно применять специальные методы оптимизации распределений (такие как упорядочение сообщений для минимизации узких мест). При помощи специализированных функций для конкретных распределений можно добиться такой же производительности, как при обмене сообщениями.





Перевод
Fraction of peak bandwidth – доля пиковой полосы пропускания;
Bandwidth – пропускная способность;
Bandwidth Dominated regime – доминирующий режим полосы пропускания;
Message size as a fraction of total memory – размер сообщения в долях от общего объема памяти;
Processor Clock cycles – тактовые циклы процессора;
Latency – задержка;
Latency dominated regime – доминирующий режим задержки.

Рисунок 5.8. Сравнительная характеристика MatlabMPI и MPI

Полоса пропускания и задержка в зависимости от размера сообщений. Пропускная способность представлена как доля пиковой пропускной способности соединения нижнего уровня. Задержка выражена в циклах процессора. Для больших сообщений производительность обеих библиотек сопоставима. Для малых сообщений задержка MatlabBMPI гораздо выше

В отношении приложений минимизация накладных расходов означает применение алгоритма, который использует меньшее количество больших сообщений вместо множества малых. Относительная производительность хорошо реализованной программы может в значительной степени основываться на производительности нижележащей библиотеки обмена сообщениями. Очевидно, тривиально массовые параллельные вычисления, которые характеризуются небольшим объемом коммуникации, смогут обеспечить хорошую производительность. Приложения, использующие групповую коммуникацию распределенных массивов (например, «каждый к каждому» или широковещательная рассылка – `broadcast`), склонны к переходу в режим работы с большими сообщениями, обеспечивая таким образом приемлемую производительность. Приложения, которым необходимо множество случайных сообщений небольшого размера, менее эффективны.

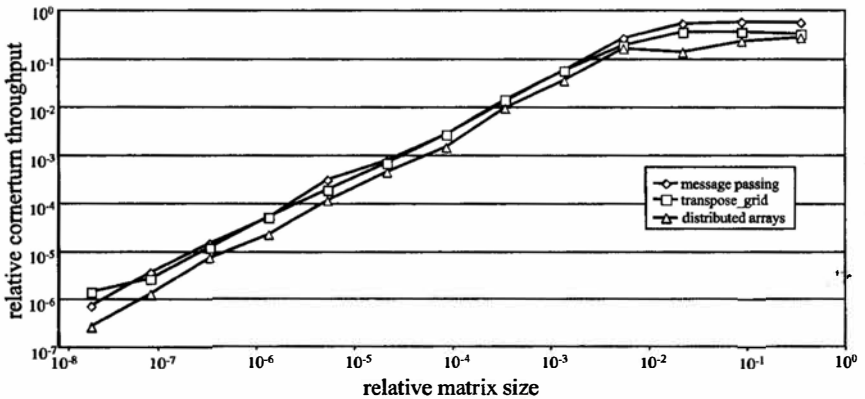
5.11. Принципы увеличения производительности

В предыдущем подразделе были представлены некоторые подробности реального устройства библиотеки распределенных массивов в MATLAB. Основываясь на дан-

ной информации, можно вычленить некоторые принципы увеличения производительности, которые могут стать полезными при использовании распределенных массивов для создания параллельных программ на MATLAB.

5.11.1. Производительность, производительность, производительность

Единственной причиной создания параллельной программы является стремление ускорить ее выполнение. К сожалению, при этом очень легко погрузиться в сложности при попытке заставить параллельное приложение работать и потерять из виду главную цель. На каждом этапе разработки необходимо проверять производительность каждой части. Если отдельные части параллельного приложения строятся без учета их производительности, очень сложно будет «настроить» приложение позже. На больших проектах обычно необходимо присутствие так называемого «performance hawk» (ярого сторонника жесткого курса производительности), который имеет право отклонить внесение каких-либо изменений в программу, которые снижают производительность. Причиной этому является естественный интерес большинства программистов к функциональности, нежели проведению времени за регрессивным тестированием производительности.



Перевод
Relative cornerturn throughput – относительная пропускная способность передачи с переупорядочиванием данных;
Relative matrix size – относительный размер матрицы;
Message passimg – обмен сообщениями;
Distributed arrays – распределенные массивы.

Рисунок 5.9. Коммуникационные накладные расходы массивов
Относительная производительность метода «каждый к каждому» для реализации при помощи чистого обмена сообщениями с помощью операции `A(:,:)=B` и `transpose_grid`. Ось **X** представляет собой размер каждой матрицы по отношению к объему памяти узла. Ось **Y** представляет пропускную способность по отношению к пиковой

5.11.2. Оценка производительности

Очень важно иметь разумное представление о производительности будущей программы. Самым простым способом ее получения является применение одного или двух тестов производительности, которые были бы более простыми, чем «реальное» приложение, но при этом были бы способны передать суть того, что пытается сделать данное приложение. Используя этот тест производительности, обычно гораздо легче исследовать результаты добавления процессоров и изменения карт распределенных массивов. Поняв особенности функционирования тестов производительности, разработчик яснее представляет себе, как должно работать реальное приложение. Использование тестов также помогает выявить, в каком аспекте необходимо приложить дополнительные усилия с целью улучшения производительности.

5.11.3. Минимизация накладных расходов

Использование распределенных массивов облегчает разработку параллельных программ, однако увеличивает накладные расходы последовательных приложений. Важно провести сравнительный анализ производительности программ с одним процессором с использованием распределенных и нераспределенных массивов. Если программа с распределенными массивами работает даже на 10% медленнее, это может значительно отразиться на производительности программы, выполняемой на нескольких процессорах. Этот эффект носит название «закон Амдала» (см. следующую главу).

5.11.4. Массовая параллельность подразумевает линейное ускорение вычислений

Если приложение является массово (тривиально–параллельным (т. е. в нем отсутствует коммуникация), то с увеличением количества процессоров оно должно ускоряться линейно. Например, при работе на 10 процессорах выполнение приложения должно занимать 1/10 часть времени выполнения приложения на одном процессоре. Если данное условие не выполняется, это является признаком того, что что-то не так с приложением (например, неравномерность нагрузки) или нижележащей библиотекой распределенных массивов (например, сдвиг запуска).

5.11.5. Алгоритм и распределение ортогональны друг к другу

Задание алгоритма и задание параллелизма должны быть ортогональны. Карта размещения должна являться единственным хранилищем всей информации, описывающей параллелизм, и должна быть независимой от алгоритма. Таким образом, распределенный массив представляет собой обычный массив с картой размещения. Изменение карты должно изменять только производительность, но не должно влиять на правильность результата.

5.11.6. Не навреди!

Ни один из последовательных кодов не является хорошим параллельным кодом, однако, все хорошие параллельные коды должны хорошо выполняться последовательно. Самое последнее, что хотелось бы делать разработчику, это добавить группу конструкций, которые загрязняют код, так что он перестает быть работоспособным последовательным кодом MATLAB, который может выполняться без поддержки па-

раллелизма (и может быть передан коллегам, у которых может не быть установленных параллельных библиотек).

5.11.7. Следите за SLOC

Библиотека распределенных массивов может реализовываться через относительно небольшое количество программных строк кода (SLOC – Software Lines of Code). 5000 строк - разумное количество. К сожалению, по мере расширения библиотеки становится все труднее поддерживать уровень производительности. Следовательно, существует тонкая грань между возросшей функциональностью и возросшей производительностью.

5.11.8. Бесплатный сыр бывает только в мышеловке

Программирование параллельных библиотек зачастую ассоциируется с «фальшивым золотом». Если сложная задача выглядит так, как будто она имеет простое решение, то, вероятно, это ошибочное представление; иначе, мы бы давно его нашли. Отрезвляющим фактом является то, что более умные люди (чем мы), обладающие большими деньгами, потерпели неудачу в этой области задолго до нас. Просто прочтите работы, изданные в конце 1980-х – начале 1990-х годов, и станет очевидным, что самый обычный практик имел гораздо более глубокое понимание параллелизма, чем большинство из нас.

5.11.9. Четыре простых шага

Библиотеки распределенных массивов, как правило, реализуются в специальном инкрементном процессе разработки кода. Все эти шаги – добавление параллельности в код, управление коммуникацией и отладка – должны напрямую поддерживаться библиотеками. Исходя из нашего опыта работы со многими пользователями параллельных MATLAB-программ, мы получили стандартизированный и повторяемый процесс (см. предыдущие главы) для быстрого перехода от последовательного кода к эффективному параллельному коду. Этот процесс имеет большое значение, так как наблюдается естественная тенденция среди пользователей параллельных программ на MATLAB после добавления параллельных функций незамедлительно решать крупные задачи на большом количестве процессоров.

Шаг 1. Добавление последовательных карт

Четырехшаговый процесс начинается с добавления распределенных матриц в последовательную программу, а затем присвоения всем картам значения 1 и проверки программы с $Np = 1$ на локальной машине.

Возьмите последовательный код и добавьте карты к ключевым массивам; используйте их для создания параллельных циклов `for`; затем отключите все карты. Выполнение этого кода должно привести к точно такому же коду, как и прежде, с теми же ответами и производительностью, как при последовательном выполнении. Как вы можете представить, этот этап выявляет все типы дефектов, которые относительно легко устранить в данном ограниченном контексте.

Для поддержки этого этапа библиотека распределенных массивов обеспечивает наличие целого ряда возможностей. Все конструкторы должны выключить объект

карты. Конструкторы, которые работают без явного объекта карты, не могут быть выключены, поэтому их следует избегать. Все функции должны работать и иметь смысл как на последовательных, так и на распределенных массивах. В противном случае, код откажется работать при попытке выполнять его последовательно. По этой причине большинство библиотек имеют последовательные «заглушки» для всех своих функций. Рекомендуется избегать использования функций, которые не работают в последовательном режиме. В результате получаем код, по-прежнему являющийся хорошим последовательным кодом, которым можно поделиться с другими, кому необязательно требуется его параллельное выполнение.

Шаг 2. Включение параллельных карт

Вторым шагом является включение карт размещения и повторное выполнение программы с $Np = 1$, в ходе чего проверяется корректность работы перегруженных конструкций в библиотеке распределенных массивов. Также немаловажно взглянуть на относительную производительность на первом и втором шаге, так как это поможет определить любые непредвиденные накладные расходы, вызванные использованием конструкций распределенных массивов.

Это служит проверкой того, что результаты и производительность в последовательном режиме по-прежнему работают корректно в присутствии распределенных массивов. На этом этапе выявляются все типы дефектов, которые относительно легко устранить в данном ограниченном контексте. Кроме того, он выявляет все непосредственные накладные расходы по производительности (например, от использования функций распределенных массивов), которые необходимо исправить до попытки перехода к параллелизму.

Для поддержания этого этапа библиотека распределенных массивов использует ту же технологию, что и на первом этапе: карты могут быть выключены и включены без затруднений. Кроме того, все функции поддержки параллелизма должны работать на обычных массивах. В итоге издержки любых перегруженных функций необходимо минимизировать; иначе простое использование распределенных массивов замедляет работу программы, что затрудняет ускорение вычислений программой в параллельном режиме.

Шаг 3. Масштабирование на рабочей станции

Следующий шаг заключается в выполнении программы при помощи множества экземпляров MATLAB на локальной рабочей станции. Как правило, выполнение с $Np = 2, 3$ и 4 работает лучше всего и осуществляет адекватную проверку кода. Часто случай $Np = 3$ бывает особенно полезным, так как размеры массивов (т. е. N), как правило, не кратны 3, и выявляются ошибки, возникшие вследствие допущения: «$N \bmod Np = 0$».

Шаг 3 является наиболее важным шагом процесса отладки, так как именно на нем параллелизм тестируется впервые. На данном шаге проверяется правильность использования измерений, полученных от распределенных массивов. Кроме того, при необходимости осуществления коммуникации, она впервые проверяется именно на этом этапе.

Для поддержки этого этапа необходимо, чтобы библиотека распределенных массивов позволяла запустить множество экземпляров MATLAB на одной машине. Для этого требуется минимизация времени запуска и объема занимаемой памяти для каждого экземпляра MATLAB. Кроме того, взаимодействия должны осуществляться

без взаимоблокировок. Наконец, необходимо иметь понятный механизм наблюдения за выводом всех выполняемых экземпляров MATLAB.

Шаг 4. Масштабирование на параллельной системе

Последний шаг заключается в фактическом выполнении программы на системе параллельных вычислений с $Np > 1$. Большинство разработчиков (во вред себе) перескакивают сразу на этот этап. При первом выполнении на такой системе основной целью является проверка корректного запуска программы и правильности функционирования параллельной коммуникации. Если программа хорошо выполнялась на шаге 3, на шаге 4 в основном происходит тестирование правильности установки программного обеспечения и надлежащего конфигурирования параллельной машины.

До настоящего момента программа должна была выполняться на задачах небольшого размера (т. е. малым числом N), которые выполняются быстро. Зачастую программисты сразу пробуют выполнить программу на крупных задачах, при этом выявление ошибок занимает больше времени. Размер задачи можно увеличивать только после успешного выполнения программы на параллельной машине.

Этот этап является главной целью использования библиотеки распределенных массивов, и обычно в центре внимания разработки параллельной библиотеки находится поддержка этого этапа. Основным свойством, необходимым библиотеке распределенных массивов для поддержки этого этапа, является простота установки и конфигурирования. Кроме того, масштабирование размера задачи означает, что библиотека должна обеспечить его простоту независимо от Np. В этом заключается основное преимущество подхода на основе карт размещения, который позволяет изменять Np и N без изменения функционирования кода.

Наконец, очень важно выполнять отладку задач на самом низком этапе, который выполняется более быстро и позволяет скорее устранить ошибки.

5.12. Результаты, предоставленные пользователями

Истинная мера успешности той или иной технологии – это эффективность ее применения реальными пользователями. В таблице 5.7 приведены несколько проектов, использующих вышеупомянутые технологии параллелизма в MATLAB [Reuther et al. 2004]. Эти проекты поддерживались сотнями пользователей и являются репрезентативными в пользовательской среде. Особый интерес вызывают столбцы, показывающие время распараллеливания и то, какие возможности дает параллелизм. Время распараллеливания показывает, как быстро код на MATLAB может быть преобразован из последовательного в параллельный, а также насколько быстро пользователь может получить параллельный код, работающий на параллельной машине. Приложения, которые становятся доступными благодаря распараллеливанию, включают сценарии, в которых могут рассматриваться наборы данных большего размера, более тщательно изучаться наборы параметров и рассматриваться более сложные имитации. В то время как мы, как правило, обеспечиваем пользователей начальными направляющими рекомендациями по распараллеливанию алгоритмов, пользователи быстро осваиваются в написании новых параллельных кодов.

Таблица 5.7. Результаты работы некоторых пользовательских приложений

В первой и последней колонках приводится краткое описание кода и то, чего удалось достичь благодаря параллельной версии кода. В средней колонке приводится расчетное время написания первоначального последовательного кода и дополнительное время для распараллеливания кода при помощи распределенных массивов и получения эффективности выполнения на параллельной машине

Описание кода	Последов. время (ч) / Парал. время (ч)	Что улучшилось/стало быстрее в результате распараллеливания
Имитации датчиков	2000/8	Более высокоточный радар
Основные принципы LADAR (лазерн. локация)	1300/1	Имитации спекл-структуры изображений
Имитации датчиков	40/0,4	Исследования области параметров
Имитации датчиков	900/0,75	Метод Монте-Карло
Распространение лазерного луча	40/1	Время выполнения
Полиномиальная аппроксимация	700/8	Более быстрый алгоритм обучения
Слежение за движением земной коры	600/3	Более быстрые/большие наборы данных
Автоматическая идентификация цели	650/40	Классы/сценарии целей
Гиперспектральный анализ	960/6	Увеличение наборов данных изображений

Почти все эти приложения работают на задачах массового параллелизма. Так как язык MATLAB является языком, ориентированным на обработку массивов данных, пользователи считают распределенные массивы естественным способом описания задач массового параллелизма. Для этих типов приложений накладные расходы написания кода намного меньше, чем при использовании обмена сообщениями. Кроме того, распределенные массивы естественным образом выполняют декомпозицию задач на наибольшие естественные части, что максимально увеличивает локальную производительность. В противоположность этому, подход «клиент/сервер» имеет тенденцию делить задачи на наименьшие функциональные единицы, вызывая более высокие накладные расходы. Пользователи распределенных массивов считают, что функций, которые предоставляет библиотека распределенных массивов, достаточно для распараллеливания их приложений и получения значительного увеличения производительности. Они пользуются преимуществами как распределенных массивов, которые часто применяются ими как классы–контейнеры, так и распределенных индексов в ситуациях, когда требуются распределенные циклы `for`. Большинство разработчиков использует фрагментированные распределенные массивы, считая, что они просты в использовании и хорошо решают поставленные задачи.

Список литературы

1. *[Abadi et al. 1991] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, Dynamic Typing in a Statically Typed Language, ACM Transactions on Programming Languages and Systems 13(2), Pages: 237–268, 1991.*

2. *[Bikshandi et al. 2006] G. Bikshandi, J. Guo, D. Hoefiinger, G. Almasi, B. Fraguela, M. Garzaran, D. Padua, and C. von Praun, Programming for PARALLELism and Locality with Hierarchically Tiled Arrays. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of PARALLEL Programming (PPoPP06), New York, March 2006.*

3. *[Bliss & Kepner 2007] Nadya Bliss and Jeremy Kepner, pMATLAB PARALLEL MATLAB Library, International Journal of High Performance Computing Applications Special Issue on High Productivity Languages and Models (editors: J. Kepner and H. Zima), SAGE Publications, Volume 21, Number 3, pages 336–359 (August 2007).*

4. *[Choy & Edelman 1999] Ron Choy and Alan Edelman, 2005, PARALLEL MATLAB: Doing it Right. Proceedings of the IEEE 93(2), Pages 331–341.*

5. *[Cummings et al. 1998] J. C. Cummings, J. A. Crotinger, S. W. Haney, W. F. Humphrey, S. R. Karmesin, J. V. Reynders, S. A. Smith, and T. J. Williams, RaPid Application Development and Enhanced Code Interoperability Using the POOMA Framework. In Proceedings of the SIAM Workshop on Object-Oriented Methods and Code Interoperability in Scientific and Engineering Computing (0098), Oct. 1998.*

6. *[DeLuca et al. 1997] Cecelia DeLuca, Curtis Heisey, Robert Bond, and James Daly, A Portable Object-Based PARALLEL Library and Layered Framework for RealTime Radar Signal Processing. In Proc. 1st Conf. International Scientific Computing in Object-Oriented PARALLEL Environments (IS COPE 97), Pages: 241–248.*

7. *[El-Ghazawi et al. 2005] Tariq El-Ghazawi, William Carlson, Thomas Sterling, and Kathy Yelick, UPC: Distributed Shared Memory Programming, John Wiley and Sons, May, 2005.*

8. *[Falcon] Falcon Project: Fast Array Language Computation, http:j /polaris.cs. uiuc. edu/falcou/*

9. *[Flynn 1972] Michael Flynn, Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C–21, p. 948, 1972.*

10. *[Johnson 2004] Phil Johnson (editor), Proceedings of 26th International Conference on Software Engineering (ICSE 2004), Edinburgh, Scotland, UK, May 23–28.*

11. *[Johnson 2005] Phil Johnson (editor), Proceedings of 27th International Conference on Software Engineering (ICSE 2005), St. Louis, MO, May 15–21.*

12. *[Kepner & Ahalt 2004] Jeremy Kepner and Stan Ahalt, MATLABMPI, Journal of PARALLEL and Distributed Computing, 2004, Volume 64, Issue 8, Pages: 997–1005.*

13. *[Kepner 2004] Jeremy Kepner (editor), 2004. Special issue on HPC Productivity, International Journal of High Performance Computing Applications, 18(4).*

14. *[Kim & Kepner 2006] Hahn Kim and Jeremy Kepner, PARALLEL Out-of-Core Programming in MATLAB Using the PGAS Model, presented at The Second Conference on Partitioned Global Address Space Programming Models, October 3–4, 2006, Washington DC.*

15. *[Koelbel 1994] Charles Koelbel, The High Performance Fortran Handbook, MIT Press, 1994.*

16. *[Lebak et al. 2005] James Lebak, Jeremy Kepner, Henry Hoffmann, and Edward Rutledge, 2005, PARALLEL VSIPL++: An Open Standard Software Library for High-Performance PARALLEL Signal Processing, Proceedings of the IEEE 93(2), Pages 313–330.*

17. *[Loveman 1993] D.B. Loveman, High Performance Fortran. PARALLEL and Distributed Technology: Systems and Applications, IEEE 1(1), 1993.*

18. *[Nieplocha et al. 2002] J. Nieplocha, R. Harrison, M. Krishnan, B. Palmer, and V. Tipparaju, Combining Shared and Distributed Memory Models: Evolution and Recent Advancements of the Global Array Toolkit. In Proceedings of POOHL2002 Workshop of ICS-2002, New York, 2002.*

19. *[Numrich & Reid 1998] R. Numrich and J. Reid, Co-array Fortran for PARALLEL pro-gramming. ACM SIGPLAN Fortran Forum, Volume 17 , Issue 2 (August 1998), Pages: 1–31.*

20. *[Ramaswamy & Banerjee 1995] S. Ramaswamy and P. Banerjee, Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In Proceedings of the Fifth Symposium on the Frontiers of Massively PARALLEL Computation (Frontiers '95), McClean, VA, February 6–9.*

21. *[Reuther et al. 2004] Albert Reuther, Tim Currie, Jeremy Kepner, Hahn Kim, Andrew McCabe, Micheal Moore, and Nadya Travinin, LLgrid: Enabling OnDemand Grid Computing with gridMATLAB and pMATLAB. In Proceedings of High Performance Embedded Computing Workshop (HPEC2004), Lexington, MA, September 28-30, 2004.*

22. *[Rutledge & Kepner 1999] Eddie Rutledge and Jeremy Kepner, PVL: An Object Oriented Software Library for PARALLEL Signal Processing, Cluster 2001, October 9, 2001, Newport Beach, CA.*

23. *[Zosel1993] M.E. Zosel, High performance Fortran: An overview. Compcon Spring 93, Digest of Papers, San Francisco, CA, February 22–26, 1993.*

Глава 6
Метрики производительности и архитектура программного обеспечения

Резюме

Метрики производительности являются бесценным инструментом для реализации параллельных программ, которые отличаются высокой эффективностью при минимизации стоимости программного обеспечения. В настоящей главе описываются некоторые метрики производительности, а также показаны способы их применения для принятия правильных решений относительно реализации приложения. В ходе представления этих метрик описывается стандартная система параллельных вычислений и рассматривается влияние производительности на различные модели параллельного программирования.

6.1. Введение

Эффективную параллельную MATLAB-программу написать довольно сложно по причине большого количества дополнительных факторов, которые необходимо учитывать, помимо основной функциональной корректности программы. Данные факторы приведены ниже.

Производительность: задержка и пропускная способность.

Эффективность: обработка, ширина пропускания и память.

Затраты на программное обеспечение: размер и переносимость кода.

Затраты на аппаратные средства: размер и мощность.

Метрики производительности являются важным инструментом, который позволяет разработчику параллельных MATLAB-программ сделать осознанный выбор метода программирования и реализации.

В данной главе основное внимание уделяется метрикам производительности и архитектурам программного обеспечения, предназначенного для реализации параллельных приложений, которые способствуют минимизации затрат на программное обеспечение при том, что они отвечают как можно большему числу других требований. В частности, мы рассмотрим различные архитектуры программного обеспечения, которые можно использовать с целью применения параллельных вычислений, предназначенных для достижения высокой производительности. В связи с этим, доминирующими факторами для архитектуры параллельного программного обеспечения являются следующие.

Тип параллелизма: параллелизм по данным, по задачам, конвейерная обработка и/или карусельная диспетчеризация (циклический алгоритм).

Модель параллельного программирования: обмен сообщениями, «начальник/подчиненный» и глобальные массивы.

Среда программирования: языки и/или библиотеки.

Подходы к решению данных вопросов лучше всего могут быть проиллюстрированы на конкретных примерах. Данная глава написана с опорой на примеры, рассмотренные в

предыдущих главах: AddOne, Mandelbrot, ZoomImage, ParallelIO, Beamformer и Blurimage.

В разделе 6.2 описываются некоторые метрики параллельных приложений и их применение на примере программ, представленных в предыдущих главах. В разделе 6.3 представлено определение типичной параллельной вычислительной машины и ее характеристики с целью получения представления о ее влиянии на параллельное приложение. В разделе 6.4 описано влияние на программное обеспечение различных моделей параллельного программирования. Наконец, раздел 6.5 дает определения метрик производительности системы, эффективности, форм-фактора и затрат на программное обеспечение, которые будут использоваться нами для оценки реализации.

6.2. Характеристики параллельного приложения

Различные программы имеют различные параллельные характеристики. Параллельная реализация, хорошо подходящая одному приложению, может совершенно не подходить другому. Отсюда естественным образом возникает следующий вопрос: какие характеристики параллельной программы оказывают наибольшее влияние на ее параллельную реализацию? Для целей отображения алгоритма на параллельную машину наиболее значимыми характеристиками являются следующие:

Основные структуры данных: массив(– ы) на каждой стадии программы, потребляющие наибольший объем памяти, на них выполняется большинство вычислений.

Сложность вычислений (W): общее количество выполненных операций (или выполненной работы) и их зависимость от алгоритмических параметров

Количество операций на элемент данных ($W_{element}$): количество выполненных операций (или выполненной работы) на каждом элементе данных.

Степени параллелизма (DoP): параллелизм, свойственный алгоритму, и его связь с основными структурами данных.

Сложность коммуникации (D): общее количество элементов данных, перемещаемых при выполнении параллельного алгоритма.

Отношение вычисление–коммуникация (W/D): объем выполненных вычислений на каждый перемещенный элемент данных.

Некоторые из этих характеристик для программ, описанных в главах 1–4, приведены в таблице 6.1.

6.2.1. Характеристики примеров программ

Программа AddOne (см. главу 1) предназначена для добавления 1 к матрице **X** $N \times N$; таким образом, общая выполненная работа составляет $W = N^2$, а работа, выполненная на каждом элементе матрицы, составляет $W_{element} = 1$.

Параллельный алгоритм AddOne разбивает матрицу в первом измерении, таким образом, имеется N степеней параллелизма, из чего следует, что не более $Np = N$ процессоров могли эффективно использоваться с данным алгоритмом. Наконец, этот алгоритм не требует коммуникации, следовательно, общий объем отправленных данных составляет $D = 0$ и $W / D \rightarrow \infty$.

Программа Mandelbrot (см. раздел 2.1) итерационно выполняет поэлементное умножение на матрицу **Z** размером $N \times N$ и экспоненциально масштабирует результат

при помощи функции `exp()`. Для умножения комплексных чисел как правило требуется ~5 операций, а экспоненциальная функция обычно вычисляется в ~15 операций, что в результате составляет ~20 операций, выполненных на каждый элемент на каждом шаге. Таким образом, выполненная работа на одном элементе матрицы составляет $W_{element}$ ~20 N_{step}, и общий объем работы составит W~20 N_{step} N^2. Параллельный алгоритм Mandelbrot разбивает матрицу в первом измерении, таким образом имеется N степеней параллелизма, из чего следует, что не более $Np = N$ процессоров могли эффективно использоваться с данным алгоритмом. Параллельная программа отправляет матрицу обратно ведущему процессору для отображения, и передаются $D = N^2$ элементов данных, в результате получаем отношение вычисление–коммуникация, равное W/D = ~20 N_{step} операций на каждый отправленный элемент данных.

Программа ZoomImage (см. раздел 2.2) выполняет свертку и интерполяцию на каждом изображении размером $N \times N$ в тензоре Z размером $N \times N \times N_s$. Каждая свертка выполняет $2N^2_K$ операций на пиксель. Каждая интерполяция выполняет ~10 операций на пиксель. Суммарная работа на каждый элемент данных составляет $W_{element}$ ~ $2N^2_K$+10. Общий объем работы составляет W~$(2N^2_K+10)N^2Ns$. Алгоритм параллельной программы `ZoomImage` обрабатывает каждое изображение независимо, следовательно, имеется N_s степеней параллелизма, и значит не более $Np = N_s$ процессоров могли эффективно использоваться с данным алгоритмом. Параллельная программа отправляет тензор ведущему процессору для отображения целиком, таким образом передаются $D = N^2N_s$ элементов данных, в результате получаем отношение вычисление–коммуникация, равное W/D ~ $2N^2_K$+10 операций на каждый отправленный элемент данных.

Программа `ParallelIO` (см. раздел 2.3) не выполняет никаких вычислений, а просто записывает и считывает матрицу **X** размером $N \times M$. Таким образом, $W = 0$ и $W_{element} = 0$. Программа `ParallelIO` разбивает матрицу во втором измерении, что дает M степеней параллелизма. Технически, в программе `PARALLELIO` отсутствует коммуникация между процессорами, но она записывает файлы на устройство хранения данных. Если устройство хранения данных доступно посредством сети, то это эквивалентно отправке $D = NM$ элементов данных. При использовании этого значения D очевидно, что отношение вычисление–коммуникация составит $W/D = 0$.

Программа Blurimage (см. главу 4) выполняет свертку на изображении Z размером $N_x \times N_y$. Каждая свертка выполняет $2N^2_K$ операций на пиксель, и в результате работа на каждый элемент данных составляет $W_{element} = 2N^2_K N_{blur}$. Общий объем работы составляет $W = 2N^2_K N_x N_y N_{blur}$. Алгоритм распределенных массивов параллельной программы Blurimage разбивает изображение в первом измерении, но каждый процессор хранит N_K дополнительных столбцов перекрывающихся граничных данных так, чтобы, он мог выполнять свертку. Без этого перекрытия существовало бы N_x степеней параллелизма. Однако если $N_x/Np < N_K$, алгоритм не будет работать, так как перекрытие будет больше, чем часть изображения, хранящегося в одном процессоре. Таким образом, фактические степени параллелизма составляют N_x/N_K, поэтому максимальное количество процессоров, которые можно использовать, составит $Np = N_x/N_K$. Алгоритм распределенных массивов Blurimage пересылает перекрывающиеся граничные данные после каждой итерации, в результате чего отправляются $D = N_p N_k N_y N_{blur}$ элементов данных. Соответствующее отношение вычисление–коммуникация составляет $W/D = 2N_K N_x/N_p$ операций на каждый [illegible] элемент данных.

Программа Beamformer (см. главу 3) состоит из четырех этапов, которые заключаются в создании имитированных данных датчика, пространственной фильтрации данных, записи и затем суммировании по третьему измерению. На этапе 1 комплексная матрица размером $N_s \times N_b$ умножается на каждую из вещественных матриц размером $N_t \times N_b$, записывая результаты в массив $\boldsymbol{x}_0$ размером $N_t \times N_b \times N_f$. Каждое перемножение матриц совершает $W_{element} = 2N_b$ операций на элемент. Общая работа составляет $W = 4N_t N_b N_f N_s$. Параллельный алгоритм Beamformer разбивает изображение в третьем измерении, в результате чего имеется N_f степеней параллелизма. На этом этапе коммуникация не требуется, таким образом $D = 0$ и $W / D \rightarrow \infty$. Характеристики этапа 2 идентичны этапу 1 за исключением того, что выполняется умножение комплексных матриц, за которым следует вычисление абсолютной величины, а второе измерение $\boldsymbol{x}_1$ составляет N_s вместо N_b, в результате чего получаем $W_{element} = 13N_S$. На этапе 3 данные записываются во внешнюю память, и его характеристики идентичны примеру `ParallelIO`. На этапе 4 суммируется вещественный массив $\boldsymbol{x}_3$ размером $N_t \times N_b \times N_f$ по третьему измерению, при этом выполняется $W_{element} = 1$ операций на элемент. Параллельный алгоритм сначала выполняет суммирование локально, а затем собирает результаты на ведущем процессоре, который суммирует Np матриц. В результате получаем $D = N_t N_b N_p$ и $W/D = N_f/N_p$.

Выводы

Характеристики приложений, приведенные в таблице 6.1, представляют собой наиболее заметные особенности параллельной реализации.

Основной массив данных характеризует размер задачи, из чего определяется, помещается ли эта задача на один процессор. Если размер задачи не меняется в зависимости от количества процессоров, это называется «фиксированным» размером задачи, и для некоторого числа Np найдется достаточно памяти, чтобы задача могла разместиться в ней. Если размер задачи меняется в зависимости от количества процессоров, это называется «масштабированным» размером задачи. Например, в программе `dB1urimage` $N_x = 1024Np$, таким образом ... размер задачи линейно пропорционален числу Np. [Примечание: исторически сложилось так, что иногда производительность алгоритма на фиксированных размерах задач называется «сильным» масштабированием, в то время как производительность на масштабированных задачах называется «слабым» масштабированием].

Сложность вычислений (W) показывает, какое общее количество операций необходимо выполнить. Разделив это число на скорость системы (в операциях в секунду), получим оценку общего времени выполнения данных операций.

Таблица 6.1. Характеристики параллельных алгоритмов

Различные вычислительные характеристики примеров программ, приведенных в главах 1–4

Программа	Массив данных	W сложность вычисланий операций	$W_{element}$ операций/ элемент	Степени паралле–лизма	D сложность коммуник. (отправлено элементов)	W/D выч/ коммуник. Опер./ элемент
AddOne	$X : R^{P(N)\times N}$	N^2	1	N	0	∞
Mandelbrot	$Z : C^{P(N)\times N}$	$20N_{step}N^2$	$20N_{step}$	N	N^2	$20N_{step}$
Zoom-1mage	Z:	$2N_K^2+10N^2N_s$	$20N_K^2+10$	N_s	N^2N_s	N_K^2+10
ParallelIO	$X : R^{N\times P(M)}$	0	0	M	$2NM$	0
Blurimage	$Z : R_+^{P_{on(N_K)}(N_x)\times N_y}$	$2N_K^2N_xN_yN_{blur}$	$2N_K^2N_{blur}$	N_x/N_K	$N_PN_kN_yN_{blur}$	$2(N_KN_xN_p)$
Beamformer						
Stage 1	$x_0 : R^{N_t\times N_b\times P(N_t)}$	$4N_tN_bN_rN_s$	$4N_b$	N_r	0	∞
Stage 2	$x_1 : R^{N_t\times N_b\times P(N_t)}$	$13N_tN_bN_rN_s$	$4N_s$	N_r	0	∞
Stage 3	$x_2 : R^{N_t\times N_b\times P(N_t)}$	0	0	N_r	$N_tN_bN_r$	0
Stage 4	$x_3 : R^{N_t\times N_b\times P(N_p)}$	$N_tN_bN_r$	N_r	N_r	$N_tN_bN_P$	N_r/N_p

Значение операций на элемент данных ($W_{element}$) показывает, насколько большой вычислительной мощности ($W_{element} > 100$) или большого количества данных для обработки ($W_{element} < 10$) требует алгоритм. В общем, алгоритмы, требующие большой вычислительной мощности, с большей вероятностью эффективны в исполнении, чем алгоритмы, требующие обработки большого количества данных.

Степени параллелизма показывают, вдоль какого измерения распределяется алгоритм. Он определяет количество процессоров, которые могут эффективно использоваться. Также, для многоэтапных приложений, если параллельное измерение и степени параллелизма не меняются на каждом этапе, дополнительной коммуникации между этапами не требуется.

Сложность коммуникации (D) показывает, сколько всего элементов данных необходимо передать. Разделив это значение на пропускную способность канала коммуникации (в элементах данных в секунду), получим оценку общего времени передачи элементов данных.

Отношение вычисление–коммуникация (W/D) показывает, насколько большой вычислительной мощности ($W/D > 100$) или объема коммуникации ($W/D < 10$) требует алгоритм. В общем, алгоритмы, требующие большой вычислительной мощности, с большей вероятностью эффективны в исполнении, чем алгоритмы, требующие передачи большого количества данных. Для многих алгоритмов $W_{element}$ и W/D сходны.

6.2.2. Метрики производительности в последовательном режиме

Характеристики программы, представленные в таблице 6.1, предоставляют большое количество информации, но ни одна из них не является метрикой производительности. Чтобы получить метрики производительности на основе этих характери-

стик, необходимо ввести в рассмотрение временные рамки, внутри которых выполняются операции или передаются элементы данных.

В описании требований общей производительности системы пропускная способность является ключевым параметром. Рассмотрим некое обобщенное приложение, которое многократно считывает D элементов данных в каждый момент времени и выполняет $W_{element}$ операций на каждом элементе данных. Далее, пусть T_{input} – время, за которое мы хотим, чтобы проходила обработка. Например, T_{input} может быть интервалом времени между последовательным поступлением наборов из D элементов данных. В данном случае необходимо выполнить все операции на одном наборе данных перед тем, как поступит следующий набор, чтобы предотвратить их нагромождение. В более общем смысле, T_{input} может быть просто желаемым временем обработки D элементов данных.

Необходимая пропускная способность BW_{input} (в элементах в секунду) задается следующим выражением

$$BW_{input} = D/T_{input}.$$

В общем случае если приложение является многоэтапным, пропускная способность i-ого этапа определяется как

$$BW^i_{input} = D^i/T_{input}.$$

Простым способом расчета общей необходимой скорости обработки является умножение пропускной способности на количество необходимых операций на каждом элементе данных. Таким образом, целевая норма производительности R^i_{goal} определяется так

$$R^i_{goal} = W^i_{element} BW^i_{input} = W^i/T_{input}.$$

Если желаемая норма производительности меньше той, которую можно получить, используя один процессор, тогда нужно использовать параллельные вычисления.

Изменение количества процессоров. Расчет необходимого количества процессоров для достижения целевой производительности является важной частью разработки хорошей реализации параллельного приложения. Алгоритм, который с успехом применяется для 20 процессоров, может не сработать для 200 процессоров, если степеней параллелизма недостаточно. Основная оценка количества процессоров состоит из следующих этапов:

Анализ сложности вычислений. Расчет общего объема работы, который необходимо выполнить приложению.

Анализ пропускной способности. Преобразование общего объема выполненной работы в скорость, с которой эта работа должна быть выполнена.

Производительность процессора. Расчет производительности алгоритма на одном процессоре.

После анализа вышеуказанных характеристик рассчитывается номинальный размер процессорной части (выраженный в количестве процессоров) делением необходимой пропускной способности на скорость обработки

$$N_p^{est} \approx \sum_i \frac{R_{goal}^i}{e_{comp}^i R_{peak}},$$

где e^i_{comp} – эффективность, при которой процессор с пиковой производительностью R_{peak} (в операциях в секунду) может выполнять работу на этапе i.

6.2.3. Степени параллелизма

Термин «степени параллелизма», упомянутый в предыдущем разделе, относится к объему параллелизма, доступному определенному параллельному алгоритму. Рассмотрим некий общий алгоритм с множеством этапов, в котором на каждом этапе возможности параллелизма могут отличаться. Типы параллелизма, можно условно разделить на три категории (см. рисунок 6.1).

Крупноэернистый (карусельная диспетчеризация): представляет собой высший уровень параллелизма и использует возможность обработки одних входных данных независимо от других. Эта форма параллелизма обычно предполагает наименьший объем коммуникации между процессорами, но требует наибольший объем памяти и *характеризуется самой высокой задержкой.*

Параллелизм по задачам/конвейерный: разбивает этапы обработки на части и составляет из них конвейер. Результат одного этапа подается на вход следующего этапа таким образом, что в любой момент времени на каждом этапе идет работа над разными наборами данных.

Параллелизм по данным: это наиболее мелкозернистый параллелизм, он разбивает каждую порцию входных данных на более мелкие части.

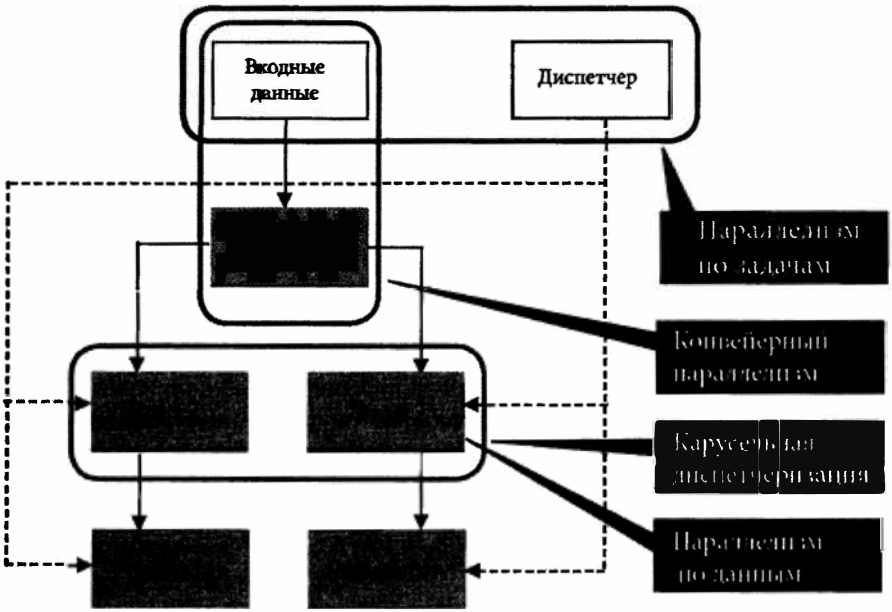


Рисунок 6.1. Типы параллелизма

Внутри некоторого общего приложения можно встретить различные типы параллелизма, которые можно задействовать: параллелизм по задачам (крупнозернистый), конвейерный параллелизм, карусельная диспетчеризация (крупнозернистая) и параллелизм по данным (мелкозернистый)

Параллельное распределение указывает, сколько процессоров используется на каждом этапе. Цель параллельного распределения заключается в том, чтобы наилучшим образом распределить процессоры для достижения желаемой производительности. Распределение приложения может параметризироваться следующими значениями:

N_{coarse} ≡ количество различных задач или конвейеров;
N_{stage} ≡ количество этапов в конвейере;
N^{i}_{stage} ≡ количество процессоров, используемых на i–ом этапе в конвейере.

Эти параметры должны удовлетворять следующему условию: сумма всех процессоров, используемых на каждом этапе в каждом конвейере, должна быть равной общему количеству процессоров Np:

$$N_p = N_{coarse} \sum_{i=1}^{N_{stage}} N^{i}_{stage} .$$

Существует несколько важных частных случаев из приведенного выше описания. Первый случай – чистый крупнозернистый параллелизм, в котором каждому процессору дается весь набор данных для обработки, при этом не задействуется ни мелкозернистый, ни конвейерный параллелизм:

$$N_{coarse} = N_P,\ N_{stage} = 1,\ N^1_{stage} = 1.$$

Следующий случай – чистый конвейерный параллелизм. В данном случае действует ограничение, определяющее, что общее количество процессоров не может превышать количество этапов в конвейере:

$$N_{coarse} = 1,\ N_{stage} = N_P,\ N^1_{stage} = 1.$$

Наконец, случай чистого параллелизма по данным, при котором задействуется только мелкозернистый параллелизм:

$$N_{coarse} = 1,\ N_{stage} = 1,\ N^1_{stage} = N_P.$$

6.2.4. Метрики параллельной производительности (коммуникация отсутствует)

В отсутствие затрат на коммуникацию и ввод/вывод все описанные выше подходы к параллельным вычислениям требуют одинакового количества времени на обработку входного блока данных (которых предполагается N_{frame} экземпляров)

$$T_{comp}(Np) = \frac{W_{tot}}{e_{comp} R_{peak} Np} \infty \frac{W_{tot}}{Np},$$

где W_{tot} – общее количество вычислительных операций или работы, которую необходимо выполнить для обработки набора элементов данных на каждом этапе (см. таблицу 6.1) задается следующим образом

$$W_{tot} \equiv \sum_{i=1}^{Nstage} W^i = W^{1a} + W^{1b} + W^{1cd} + W^{1e} + W^2.$$

Ускорение вычислений будет линейно зависеть от количества процессоров:

$$S_{comp}(Np) \equiv \frac{T_{comp}(1)}{T_{comp} Np} = Np.$$

Таким же образом при нулевой коммуникации задержка (т. е. время, затрачиваемое на получение первого ответа) – это количество этапов, умноженное на самый длинный отрезок времени, необходимый для выполнения обработки на одном из этапов

$$T_{latency}(Np) = N_{stage} \max_i (T^i_{latency}(N^i_{stage})),$$

где

$$T_{latency}(N^i{}_{stage}) = \frac{W^i{}_{stage}}{e_{comp} R_{peak} N^i{}_{stage}}.$$

Причина в том, что ни один этап не может продолжаться, пока не закончился предыдущий этап. Ускорение задержки в этом случае составит

$$S_{latency}(Np) = \frac{T_{latency}(1)}{T_{latency}(Np)}.$$

Общий объем памяти, необходимый при различных подходах, составляет

$$M(Np) = N_{coarse} \sum_{i=1}^{N_{stage}} \frac{M^i{}_{stage}}{N^i{}_{stage}}.$$

Возможно, наиболее важной метрикой является объем памяти, необходимый для отдельного процессора

$$M_{cpu\text{-}max}(Np) = \max_i (M^i{}_{stage}/N^i{}_{stage}).$$

6.2.5. Метрики параллельной производительности (коммуникация присутствует)

Безусловно, допущение, что затраты на коммуникацию и хранение отсутствуют, не соответствует действительности, и единой целью разработки и реализации параллельных приложений главным образом является минимизация этих затрат. Тем не менее вышеописанные модели полезны нам в том отношении, что они определяют верхние границы производительности. Кроме того, определения ускорения и объема занимаемой памяти при добавлении коммуникации остаются неизменными.

В случае мелкозернистого параллелизма требуется коммуникация между этапами, на которых меняется направленность параллелизма (принцип размещения данных). В свою очередь, при изменении направленности требуется выполнить переупорядочивание данных с целью применения параллелизма. Время для сообщения данных от этапа i до этапа $i+1$ задается так

$$T_{comm}^{i \to i+1}(N^i{}_{stage} \to N^{i+1}{}_{stage}) = \frac{D^{i \to i+1}}{BW_{ef} f(N^i{}_{stage} \to N^{i+1}{}_{stage})},$$

где $D^{i \to i+1}$ – объем данных (в байтах), перемещаемых от этапа i до этапа $i+1$, и $BW_{eff}(N^i{}_{stage} \to N^{i+1}{}_{stage})$ – эффективная пропускная способность (в байтах в секунду) между процессорами, которая задается так:

$$BW_{eff}(N^i{}_{stage} \to N^{i+1}{}_{stage}) = e_{comm}^{i \to i+1} BW_{peak}(N^i{}_{stage} \to N^{i+1}{}_{stage}).$$

$BW_{peak}(N^i{}_{stage} \to N^{i+1}{}_{stage})$ – пиковая пропускная способность от процессоров на этапе i к процессорам на этапе $i+1$. Для получения более подробной модели такой производительности требуется модель соединений процессоров, которая приведена в следующем разделе.

Выводы из оценки коммуникации в разных случаях заключаются в следующем. В случае чистого конвейерного параллелизма (при условии, что вычисление и коммуникация могут выполняться одновременно) время вычисления и задержка составляют

$$T_{tot}(Np) = \max_i(\max(T^i{}_{latency}(1), T^{i\to i+1}_{comm}(1\to 1)))/N_{coarse}$$

$$T_{latency}(Np) = 2N_{stage}\max_i(\max(T^i{}_{latency}(1), T^{i\to i+1}_{comm}(1\to 1))).$$

Если вычисление и коммуникация не могут выполняться одновременно, внутренняя функция `max()` заменится сложением. В этом случае основным преимуществом конвейерного параллелизма является то, что он определяется максимальным (а не суммированным) временем по всем этапам. За счет улучшения производительности, однако, увеличивается задержка, которая пропорциональна количеству этапов коммуникации и вычислений. Наблюдается также незначительное влияние на объем используемой памяти.

В случае чистого параллелизма по данным время вычисления и задержка составляют

$$T_{tot}(Np) = \frac{T_{comp}(Np) + T_{comm}(Np)}{N_{coarse}}$$

$$T_{latency}(Np) = T_{comp}(Np) + T_{comm}(Np),$$

где

$$T_{comm}(Np) = \sum_i T^{i\to i+1}_{comm}(Np \to Np).$$

Как правило, в случае чистого параллелизма по данным сложно обеспечить одновременность вычислений и коммуникации. Основное преимущество параллелизма по данным заключается в том, что время вычислений уменьшается при условии, что они не ожидают необходимой коммуникации. Таким же образом в этом случае уменьшается задержка. Данный метод также обеспечивает линейное сокращение объема требуемой памяти.

Для комбинированного случая параллельно-конвейерного подхода мы получаем

$$T_{tot}(Np) = \max_i(\max(T_{comp}(N^i{}_{stage}), T^{i\to i+1}_{comm}(N^i{}_{stage} \to N^{i+1}{}_{stage})))/N_{coarse}$$

$$T_{latency}(Np) = 2N_{stage}\max_i(\max(T_{comp}(N^i{}_{stage}), T^{i\to i+1}_{comm}(N^i{}_{stage} \to N^{i+1}{}_{stage}))).$$

Этот подход обеспечивает общее уменьшение времени вычислений и допускает одновременность вычислений и коммуникации, а также дает сокращение объема требуемой памяти. Как мы увидим далее, этот метод часто применяется в системах параллельной обработки сигналов.

6.2.6. Закон Амдала

Возможно, наиболее важным принципом разработки параллельных реализаций является управление накладными расходами. Предположим, что общий объем работы, которую необходимо выполнить, может быть разбит на две части: одну можно выполнить в параллельном режиме, а другую – только в последовательном (т. е. на одном процессоре)

$$W_{tot} = W_{||} + W_{|}.$$

Время выполнения тогда будет пропорционально следующей величине

$$T_{comp}(Np) \propto W_{||} / Np + W_{|}.$$

и преобразуется в ускорение так

$$S_{comp}(Np) = \frac{W_{tot}}{W_{||} / Np + W_{|}}.$$

Если мы нормализуем его по W_{tot}, оно преобразуется так

$$S_{comp}(Np) = \frac{1}{w_{||} / Np + w_{|}}.$$

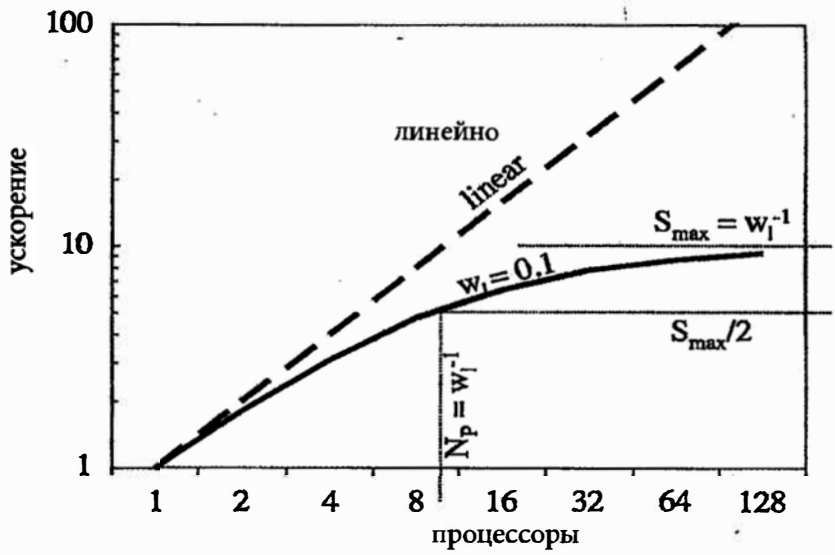


Рисунок 6.2. Закон Амдала

График показывает ускорение программы с долей Амдала $w_{|} = 0.1$. Точечным пунктиром обозначено максимальное ускорение $S_{max} = w^{-1}_{|} = 10$ и точка половинного ускорения $Np = w^{-1}_{|}$, где $w_{||} = W_{||} / W_{tot}$ и $w_{|} = W_{|} / W_{tot}$. Если Np очень большое, то максимальное достижимое ускорение составляет

$$S_{\max} = S_{comp}(Np \to \infty) = w_{\mathrm{I}}^{-1}.$$

Например, если $w_{\mathrm{I}} = 0.1$, то максимальное ускорение составляет 10 (см. рисунок 6.2). Этот основной результат носит название закон Амдала [Amdahl 1967], а значение $w_{\mathrm{I}} = 0.1$ часто называют «долей Амдала» в приложении. Закон Амдала подчеркивает необходимость сделать каждый аспект кода параллельным. Он также верен по отношению к другим накладным расходам (например, коммуникации), из-за которых не выполняется никакой полезной работы. Наконец, закон Амдала также является полезным инструментом для нахождения компромиссов. Половина максимального ускорения достигается при $Np = w_{\mathrm{I}}^{-1}$.

$$S_{comp}(Np = w_{\mathrm{I}}^{-1}) = w_{\mathrm{I}}^{-1} / (w_{\mathrm{II}} + 1) \approx \frac{1}{2} S_{max}.$$

Если $w_{\mathrm{I}} = 0.1$, то использование более 10 процессоров будет иметь минимальную полезность. Аналогично: если для соответствия целевой производительности необходимо ускорить приложение в 100 раз, то реализация должна быть оптимизирована до момента $w_{\mathrm{I}} < 0.0$.

6.2.7. Характеристика ускорения

Ускорение, достигнутое параллельным приложением, является фундаментальной мерой эффективности использования приложением преимуществ системы параллельных вычислений. Ускорение обычно характеризуется как функция количества процессоров $S(Np)$. Форма данной кривой, как правило, позволяет выделить несколько категорий. Обычно ускорение измеряется по показателям степеней двойки, равных количеству процессоров, $Np = 1, 2, 4, 8, \ldots$, и кривая строится логарифмически. На рисунке, 6.3 изображены несколько наиболее распространенных кривых ускорения. Идеальным случаем является линейное ускорение, где $S(Np) > Np$. Более типичным является сублинейное ускорение, где $S(Np) = \alpha Np$ и $0 < \alpha < 1$. К сожалению, даже более распространенной среди начинающих пользователей является кривая ускорения, насыщение которой происходит при довольно небольшом числе процессоров. Наиболее редко встречается сверхлинейное ускорение, где $S(Np) > Np$. Сверхлинейное ускорение обычно случайно достигается квалифицированными специалистами.

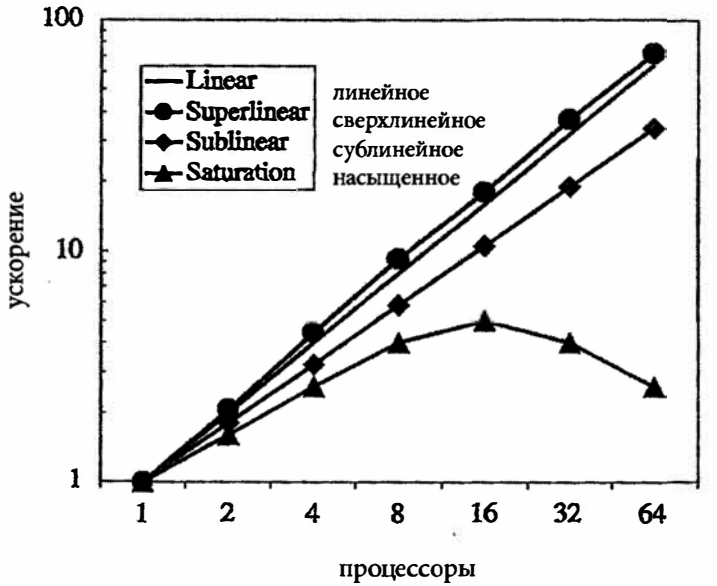


Рисунок 6.3. Кривые ускорения. Графики различных типов ускорения
По данным кривым можно охарактеризовать реализацию параллелизма. Если достигнуто линейное ускорение, то, как правило, нет необходимости в дальнейшей оптимизации. Линейного ускорения достичь легче всего на задачах массового параллелизма с минимальной коммуникацией или при ее отсутствии. При наличии таких задач целью должно быть линейное ускорение, и многим программистам в итоге удается его получить

Часто допускается сублинейное ускорение, которое обычно означает наличие постоянных накладных расходов, количественное отношение которых остается постоянным при увеличении числа процессоров. Например, программа, в которой доля Амдала уменьшается с увеличением количества процессоров $W_1(Np)\ \alpha\ Np^{-1}$, характеризуется сублинейным поведением. При работе массово-параллельного приложения сублинейное поведение может служить признаком наличия ненужных расходов, которых потенциально можно избежать. Если программа не является массово (тривиально) – параллельной и требует наличия постоянной коммуникации, сублинейное ускорение часто бывает лучшим результатом, которого можно достичь.

Насыщение типично для многих параллельных программ на ранних стадиях разработки. Это показатель «узкого места» в коде, где один процессор ждет всех остальных для завершения некоторой задачи. Анализ программы может помочь обнаружить эти «узкие места» и устранить их.

Сверхлинейное ускорение типично для массово-параллельных программ с фиксированным размером задач (т. е. размером задачи, который не увеличивается с ростом числа Np). Последовательная производительность программы улучшается с ростом Np, так как размер задачи на каждом процессоре становится меньше и приводит к увеличению производительности кэш-памяти.

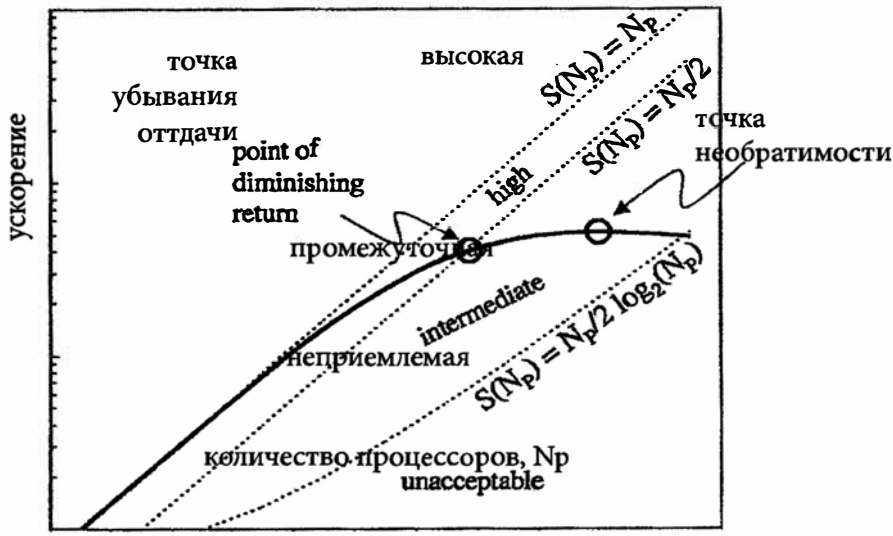


Рисунок 6.4. Диапазоны ускорения

График, демонстрирующий различные диапазоны ускорения

На рисунке 6.3 изображены кривые ускорения, демонстрирующие, как оно может наблюдаться пользователем для определенного приложения. В действительности все приложения имеют одну кривую (см. рисунок 6.4), потому что все параллельные программы характеризуются насыщением для достаточно большого числа *Np* [Kuck 1996]. Цель эффективной реализации заключается в том, чтобы передвинуть точку необратимости к достаточно большому значению *Np*. В данном контексте ускорение приложения можно разбить на три диапазона. В диапазоне высокой производительности видим ускорение от линейного до сублинейного ($Np/2 < S(Np)$). С ростом значения *Np* ускорение начинает падать, и выгода от увеличения числа процессоров уменьшается. Если $S(Np) = Np/2$, программа работает только в половину общей производительности: эта точка называется «точкой убывания отдачи». В промежуточном диапазоне ($Np/2\log_2(Np)) < S(Np) < Np/2$ программа получает ограниченную выгоду от увеличения числа процессоров. В итоге пик ускорения приходится на точку необратимости, в которой увеличение числа процессоров начинает замедлять работу программы. В диапазоне неприемлемой производительности ($S(Np) < Np/2\log_2(Np)$) выгода от выполнения программы в параллельном режиме очень невелика.

До этого момента ускорение рассматривалось только как функция *Np*. Однако существует еще одна переменная – это размер задачи *N*, которую решает программа. Совмещение обоих этих параметров дает поверхность ускорения $S(Np; N)$. На рисунке 6.5 показаны два среза этой поверхности. Во-первых, важно помнить, что только часть целого диапазона *Np* и *N* физически поместится в память компьютера. При измерении ускорения приложения важно определить, как изменяется *N* с ростом *Np*. Существует два стандартных подхода. Кривая ускорения с фиксированным размером задачи удерживает общий размер задачи *N* постоянным при увеличении *Np*. В данном

случае размер задачи на процессор: *N/Np* – уменьшается. В некоторых случаях размер задачи *N* настолько большой, что она не помещается на небольшом количестве процессоров. Исторически сложилось, что кривая ускорения часто называется кривой «сильного масштабирования». На кривой ускорения с масштабированным размером задачи общий размер задачи *N* увеличивается с увеличением *Np*. В этом случае размер задачи на процессор: *N/Np* – остается неизменным. При этом такую кривую ускорения часто называют кривой «слабого масштабирования».

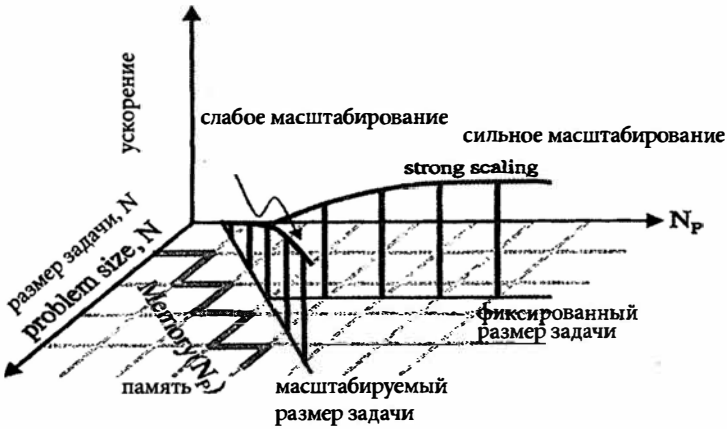
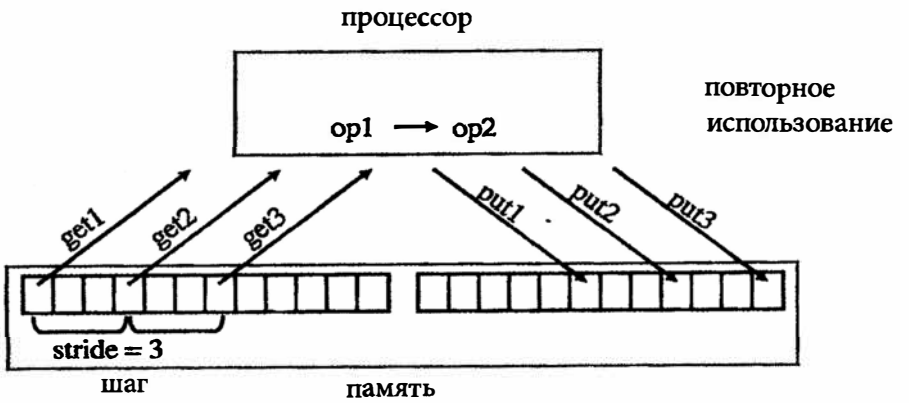


Рисунок 6.5. Сравнение слабого и сильного масштабирования. График ускорения масштабированного (слабого) и фиксированного (сильного) размера задачи

6.2.8. Пространственная и временная локальность

Более абстрактное понятие о программе можно получить, изучив ее с точки зрения выполняющей ее вычислительной машины. Для вычислительной машины программа является лишь потоком обращений к ячейкам памяти и соответствующих операций. Такой взгляд на программу часто называется моделью фон Неймана [Von Neuman 1945]. В этой модели обращения к ячейкам памяти можно анализировать с точки зрения того, как близко они находятся друг к другу (пространственная локальность) и как часто определенные обращения используются повторно (временная локальность) (см. рисунок 6.6). Пространственный промежуток между обращениями к ячейкам памяти называется шагом (stride) памяти. Повторное использование связано с тем, как часто фрагменты памяти используются повторно.



Перевод
get – получить;
put – поместить.

Рисунок 6.6. Модель пространственной/временной локальности

Понятия пространственной/временной локальности очень эффективно дают представление о программе с точки зрения вычислительной машины. Интуитивно понятно, что приложения с высоким уровнем пространственной/временной локальности должны быть «легче» для выполнения машиной, чем приложения с низким уровнем пространственной/временной локальности. Аналогично, программы с высоким уровнем пространственной/временной локальности должны легче реализовываться в параллельном режиме, так как доступы к данным являются регулярными, и каждый элемент данных, который используется, скорее всего, будет использоваться много раз

Пространственная/временная локальность – это не только абстрактное понятие. Ее можно измерить (см. [Weinberg at al. 2005]) путем изучения статистики обращений к ячейкам памяти программы во время выполнения. С помощью этой статистики можно вычислить баллы пространственной локальности по формуле

$$\text{Пространственный показатель} = \sum \frac{Stride(i)}{i},$$

где *Stride (i)* – это доля обращений к ячейке памяти с шагом *i*. Пространственный показатель – это средневзвешенное значение всех этих ссылок. Программа, содержащая обращения только с шагом 1, будет иметь *Stride (1)* = 1 и ноль для других шагов и пространственный показатель, равный 1. Программа, имеющая только большие шаги, будет обладать показателем пространственной локальности, стремящимся к нулю.

Таким же образом, временной показатель вычисляется по формуле

$$\text{Временной показатель} = \sum \frac{Reuse(2^{i+1}) - Reuse(2^{i})}{2^{i}}.$$

где *Reuse*(2^i) – доля обращений к ячейкам памяти, которые посещаются повторно менее чем в - 2i обращениях к ячейкам. Программы, часто посещающие повторно одну и ту же ячейку, будут иметь временной показатель, стремящийся к 1, а программы, которые никогда не используют данные повторно, будут иметь временной показатель, стремящийся к 0. Номинальные показатели пространственной/временной локальности для примеров программ даны на рисунке 6.7. В общем, эти примеры приложений имеют высокий уровень показателей пространственной локальности, так как они обрабатывают данные, используя регулярность обращений. Программы `AddOne` и `ParallelO` имеют низкий уровень показателей временной локальности, так как они выполняют очень мало операций на элемент данных.

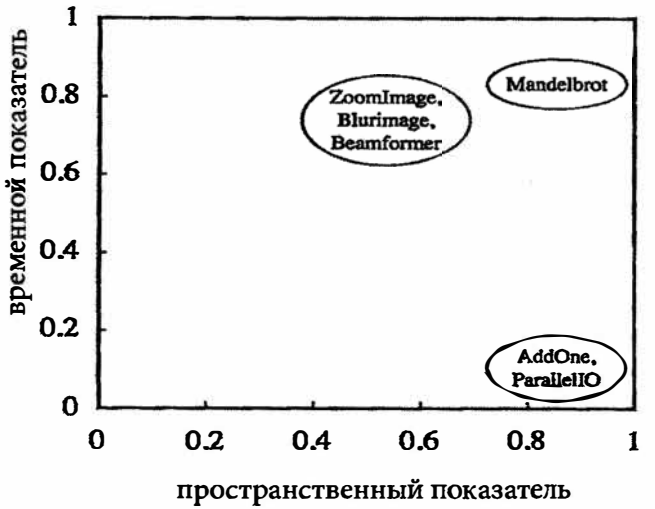
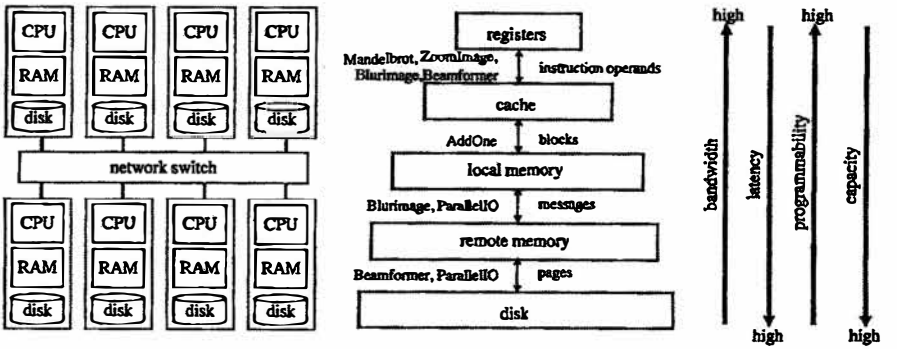


Рисунок 6.7. Пространственные/временные показатели примеров программ
Данные приложения имеют высокий уровень показателя пространственной локальности. AddOne и PARALLELIO имеют низкий уровень показателей временной локальности, так как выполняют очень мало операций

6.3. Стандартная параллельная вычислительная машина

Толкование метрик параллельных приложений, представленных в предыдущих разделах, удобнее проводить в контексте модели аппаратных средств параллельной вычислительной машины. Визуализация процесса выполнения параллельной программы на параллельной вычислительной машине – это один из самых важных и самых сложных аспектов параллельного программирования. Как правило, необязательно понимать подробности устройства оборудования, а достаточно более абстрактной модели. В данном разделе описана традиционная параллельная вычислительная машина, и этого будет достаточно для получения представления о влиянии метрик параллельного приложения на его реализацию. Более детальное описание архитектуры параллельной машины можно найти в [Hennesy & Paterson 2006].

Архитектура классической многоузловой вычислительной машины состоит из некоторого числа узлов, соединенных посредством сети. Каждый узел состоит из одного или более процессоров и памяти. Если узел имеет более одного процессора, это считается многоядерным узлом. Память в узле может подразделяться на различные уровни: кэш-память, основную память и дисковую память (см. рисунок 6.8). В этом отношении существует множество вариантов. Узел может иметь множество процессоров, делящих между собой различные уровни кэш-памяти, память и дисковую память. Кроме того, может существовать внешняя глобальная память или дисковая память, которая является видимой для всех узлов. Важным аспектом подобной традиционной архитектуры является «иерархия памяти» (см. рисунок 6.8).



Перевод
PARALLEL architecture – параллельная архитектура;
CPU – ЦПУ;
RAM – ОЗУ;
Disk – диск;
Network switch – сетевой коммутатор;
Memory hierarchy – иерархия памяти;
Application – приложение;
Unit of Memory – единица памяти;
Registers – регистры;
Instruction operands – операнды команд;
Cache – кэш-память;
Blocks – блоки;
Local memory – локальная память;
Messages – сообщения;
Remote memory – удаленная память;
Pages – страницы;
Implications – влияние;
High – высокий;
Bandwidth – пропускная способность;
Latency – задержка;

Programmability – программируемость;
Capacity – емкость.

Рисунок 6.8. Параллельная архитектура и иерархия памяти
Традиционная параллельная вычислительная машина состоит из узлов с процессорами и памятью, связанных коммуникационной сетью (слева). Эта архитектура приводит к традиционной иерархии памяти (справа)

Понятие иерархии памяти относится к тому факту, что память, находящаяся «ближе» к процессору гораздо быстрее (пропускная способность выше, а задержка ниже). Однако такая производительность достигается ценой емкости. Структура иерархии памяти обычно выглядит так:

- Регистры процессора;
- Кэш-память L1, L2,...;
- Локальное основное запоминающее устройство;
- Удаленное основное запоминающее устройство;
- Локальный диск;
- Удаленный диск.

Обычно на каждом уровне иерархии 10-кратное увеличение производительности приводит к 10-кратному уменьшению емкости. Такая «крутизна» иерархии памяти определяет величину снижения производительности по мере спуска по иерархии. Если производительность снижается значительно, мы говорим, что система имеет высокую крутизну иерархии памяти.

Различные приложения будут создавать нагрузку на различные части иерархии памяти. На рисунке 6.8 каждая часть иерархии отмечена различными приложениями: `AddOne` (локальная память), `Mandelbrot` (процессор), `ZoomImage` (процессор), `Parallel0` (сеть и диск), `Beamformer` (процессор и диск) и `Blurimage` (процессор и сеть).

Понятие иерархии памяти является, возможно, одной из самых важных идей, которую необходимо учитывать при разработке параллельного приложения. В частности, хорошее параллельное приложение эффективно ослабляет влияние на производительность иерархии памяти аппаратных средств. В этом случае от программистов требуется наличие ясного представления о системе, что поможет понять сущность ее влияния на производительность. Основное практическое правило заключается в конструировании реализации, которая минимизирует как общий объем, так и количество перемещений данных вверх и вниз по иерархической лестнице.

6.3.1. Сетевая модель

В рамках иерархии памяти коммуникационная сеть, соединяющая процессоры в параллельной системе, обычно является наиболее важной частью, которую учитывает программист, так как здесь возникают вопросы, характерные только для параллельной реализации (т. е. коммуникации между процессорами). Это справедливо для

сети как внутри узла, так и между узлами, а также для обоих случаев одновременно. Остальные части иерархии памяти также очень важны, но не являются уникальными для параллельной реализации. Например, способ использования программой регистров или локальной памяти является фундаментальным вопросом последовательного программирования. Точно так же способ записи программой на локальный диск является основным вопросом последовательного программирования. В общем случае сначала рекомендуется решить вопросы последовательного программирования, а затем приступать к параллельной реализации.

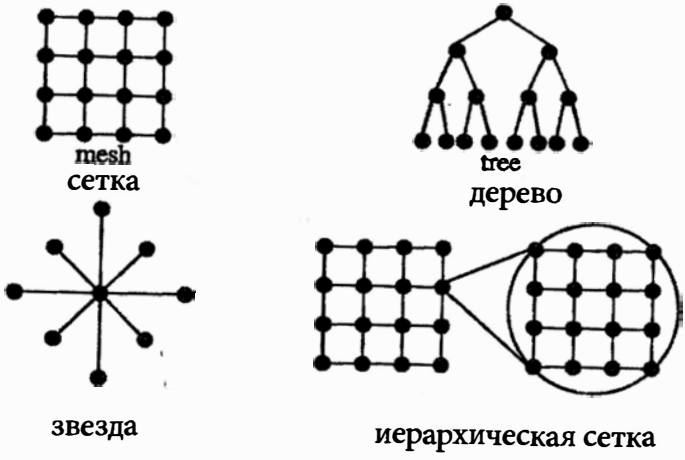


Рисунок 6.9. Топологии сетей

Способы соединения компьютеров друг с другом посредством кабелей и промежуточных точек (например, маршрутизаторы) называются топологиями сети.

Сети могут быть организованы различным образом. Способы организации соединения различных точек сети называются топологиями сети (см. рисунок 6.9). Разработчики параллельных систем обработки прикладывают неимоверные усилия к созданию и построению эффективной топологии сети. С точки зрения программиста топология сети может быть абстрагирована и приближенно приравнена ко времени, которое требуется одному процессору для отправки данных другому процессору. Отправной точкой моделирования коммуникационной сети является производительность соединений «точка–точка». Ее можно эффективно измерить посредством измерения времени, необходимого двум процессорам для отправки сообщений различных размеров друг другу. В результате получаем стандартную кривую, которая характеризуется одной функцией $T_{comm}(m)$, означающей время, затрачиваемое на отправку сообщения размера m. Функция $T_{comm}(m)$ хорошо описывается следующей простой двухпараметрической формулой:

$$T_{comm}(m) = Latency + \frac{m}{Bandweight}.$$

Пропускная способность (обычно измеряется в битах, байтах или словах в секунду) – это максимальная скорость, при которой данные могут перемещаться по сети. Пропускная способность обычно измеряется посредством измерения времени, необходимого для пересылки больших сообщений между двумя процессорами, и деления размера сообщения на общее время:

$$\text{Пропускная способность} = \lim_{m \to \infty} \frac{T(m)}{m}. \tag{6.1}$$

Задержка – это количество времени, затрачиваемое на отправку одного бита от одного процессора другому. Она обычно оценивается измерением времени для очень короткого сообщения и вычисляется по формуле

$$\text{Задержка} = \lim_{m \to \infty} T(m). \tag{6.2}$$

Эти два параметра имеют большое значение, так как некоторые программы требуют отправки нескольких сообщений очень большого размера (и ограничены пропускной способностью), а некоторые программы отправляют множество небольших сообщений (ограничивается задержкой).

На рисунке 6.10 показаны задержка и мгновенная пропускная способность для типичной кластерной сети как функции размера сообщений. Важными свойствами данной кривой, типичными для большинства сетей, являются выравнивание задержки на маленьких сообщениях и выравнивание пропускной способности при пересылке больших сообщений. Безусловно, знать пропускную способность и задержку системы очень полезно, но еще более полезно сравнить данные параметры с вычислительной мощностью процессора. В частности, если разделить скорость процессора (обычно измеряется во Флопс/FLOPS – операциях с плавающей запятой в секунду) на пропускную способность (измеряется в 8-байтовых словах в секунду), получим количество Флопс/FLOPS, которые могут быть выполнены за время, затрачиваемое на пересылку сообщения заданного размера (см. рисунок 6.10). Это «обратное значение пропускной способности» дает ясное представление о минимальном количестве операций, которое необходимо выполнить на значении для погашения издержек передачи этого значения.

В данном случае для небольших сообщений необходимо выполнить около 10 000 операций на каждом 8-байтном элементе. Для больших сообщений необходимо выполнить только 100 операций. На этом примере хорошо проиллюстрирована ключевая парадигма параллельного программирования: лучше отправить несколько сообщений большого размера, чем много сообщений малого размера.

Особенно радует в описании сети и производительности процессора тот факт, что приведенные цифры относительно постоянны для большинства интересующих нас систем. Например, эти значения (10^5 для малых сообщений и 10^2 для больших) типичны для многих систем и почти не изменяются с течением времени.

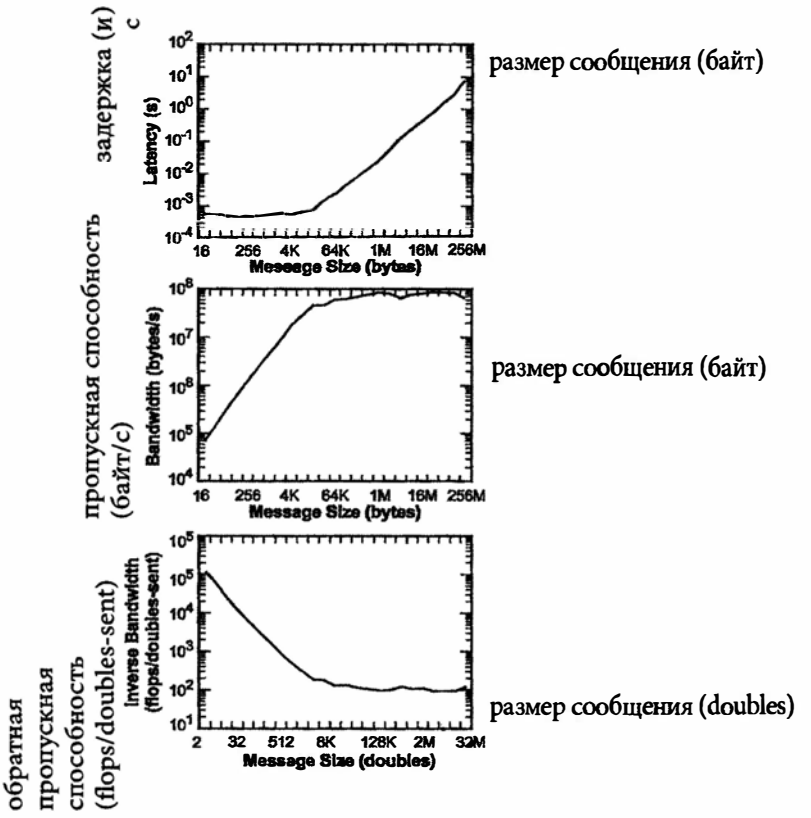


Рисунок 6.10. Производительность сети

Задержка, пропускная способность и «обратная пропускная способность» как функции размера сообщений для стандартной кластерной сети. Обратная пропускная способность выражена в отношении количества операций с вещественными числами в секунду (FLOPS) к количеству отправленных за это время по сети вещественных чисел двойной точности (double)

Абсолютные скорости процессора и сети могут существенно измениться с течением времени, но их соотношения остаются относительно неизменными. Например, в то время как скорости процессора и сети улучшились за последние несколько десятилетий в 10^3 раз, относительная производительность изменилась только на коэффициент 10.

По данным параметрам можно понять, что если параллельная программа использует сообщения большого размера, она должна выполнять >100 операций на каждое число перед передачей ее по сети. Точно так же, если программа отправляет сообщения малого размера, она должна выполнять >10 000 операций на каждое число перед отправкой. Выполнение меньшего количества операций, чем указано выше, приведет к тому, что параллельная программа будет функционировать неэффективно, так как большее количество времени будет потрачено на пересылку сообщений, а не выполнение вычислений.

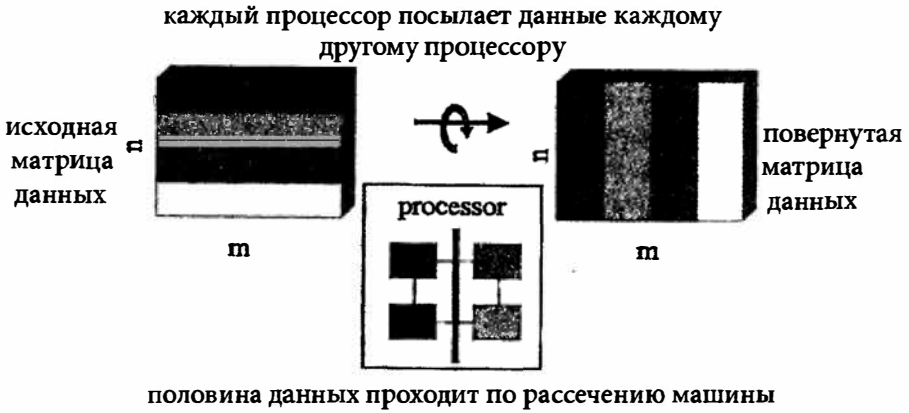


Рисунок 6.11. Переразмещение данных с поворотом угла
Перемещения данных требуют перераспределения данных из одного распределения по рядам в другое распределение по рядам

Используя простую модель «точка–точка», мы можем прогнозировать время для более сложной операции, такой как «переразмещение данных с поворотом угла» (corner–turn) (см. рисунок 6.11). Для этого требуется коммуникация типа «каждый с каждым» (all-to-all), при которой набор процессоров P_1 пересылает сообщения размера m каждому набору процессоров P_2,

$$T_{CornerTurn} = \frac{P_1 P_2 (Latency + m / Bandwidth)}{Q},$$

где B – количество байт на одно сообщение, и Q – количество одновременных параллельных путей от процессоров P_1 к процессорам P_2. Общее количество перемещенных данных за данную операцию равно mP_1P_2.

6.3.2. Иерархия Кека

Иерархия памяти и модель сети, описанные в предыдущем разделе, обычно предоставляют достаточно знаний для программирования большинства параллельных систем. Другими словами, мы рассматриваем вычислительную машину как набор процессоров, соединенных сетью. Парадигма параллельного программирования, возникающая из этой модели, заключается в разработке параллельного алгоритма таким образом, чтобы между процессорами пересылались данные наименьшего объема и наименьшее количество сообщений.

Это простая модель компьютера вполне эффективна для большинства задач, однако, в некоторых случаях необходимо применять более усложненную модель, способную работать с иерархическими коллекциями процессоров, запоминающих устройств и сетей. На рисунке 6.12 представлена такая модель, основанная на терминологии, разработанной Кеком [Kuck 1996]. Данная модель содержит процессоры, запомина-

ющие устройства и сети. На нижнем уровне (уровень 0) мы имеем набор процессоров P_0 со своей собственной локальной памятью M_0. Сообщения могут пересылаться между процессорами посредством сети $N_{0.5}$. Процессоры на уровне 0 также имеют доступ к общей памяти на следующем уровне, обозначаемой SM_1. К данным внутри SM_1 имеют доступ процессоры P_0 посредством сети общей памяти SMN_1. Такую последовательность сетей и запоминающих устройств можно продолжить на любом количестве уровней.

В реальных системах применение сетей и общих запоминающих устройств на каждом уровне маловероятно. Например, рассмотрим систему, состоящую из узлов, соединенных посредством сети (см. рисунок 6.13). Каждый узел содержит процессоры с локальной кэш-памятью и совместно используемой памятью. Такая система будет содержать P_0, M_0, SM_1, SMN_1 и $N_{1.5}$. При этом в данной системе будут отсутствовать $N_{0.5}$, SM_2 и SMN_2. Иерархия Кека также полезна для понимания более сложных многоядерных процессоров. Перед началом программирования параллельной вычислительной машины рекомендуется потратить немного времени и нарисовать схему, подобную изображенной на рисунке 6.13. Таким образом, можно взглянуть на систему изнутри и выделить какие-либо пробелы понимания, требующие дальнейшего изучения.

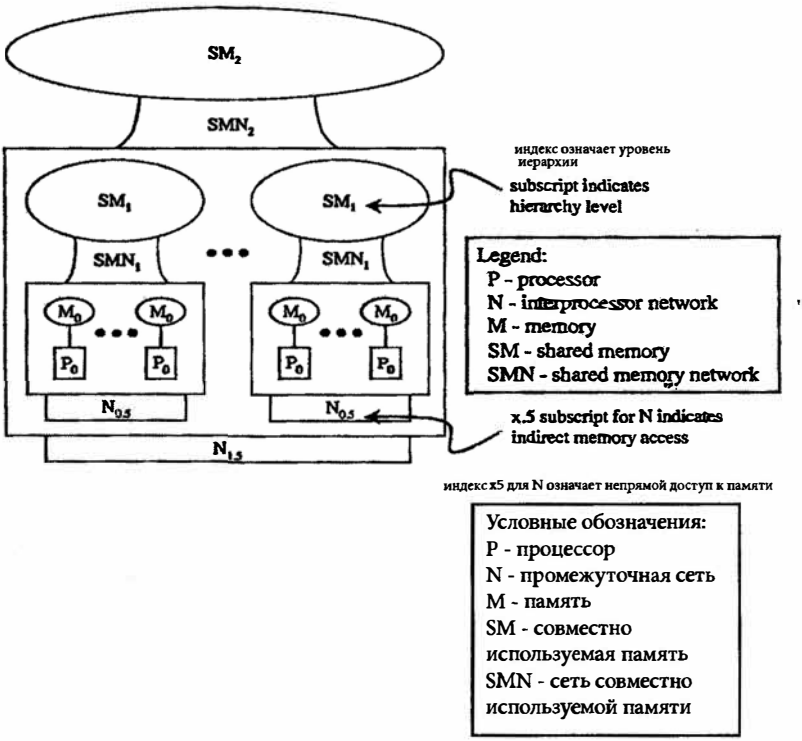


Рисунок 6.12. Двухуровневая иерархия

Терминология иерархии памяти для двухуровневой структуры (по материалам [Kuck 1996]). P_i = процессор, M_i = запоминающее устройство, N_i = сеть, SM_i = общее запоминающее устройство, SMN_i = сеть общего запоминающего устройства

6.4. Модели параллельного программирования

Модель параллельного программирования описывает, каким образом программное обеспечение будет реализовывать приложение на параллельной машине. Эффективная модель ПО позволяет задействовать различные типы параллелизма: параллелизм по данным, параллелизм по задачам, конвейерный параллелизм и карусельную диспетчеризацию. Кроме того, эффективная реализация модели параллельного программирования позволяет менять используемый тип параллелизма по мере построения системы и развития требований к производительности.

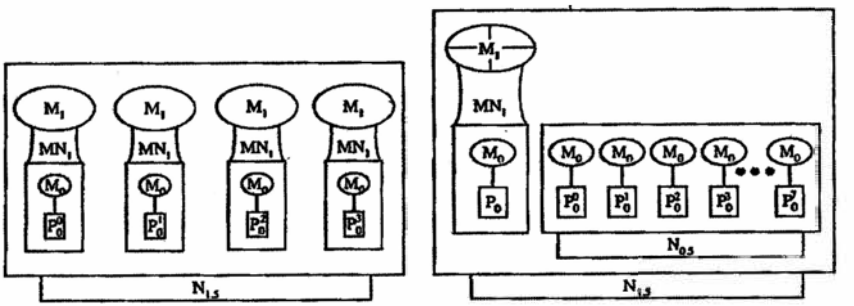


Рисунок 6.13. Пример иерархии Кека

Применение иерархии Кека к типичному кластеру однопроцессорных узлов – на схеме слева. Применение иерархии к более сложному гетерогенному (разнородному) многоядерному процессору, состоящему из универсального процессора и восьми ускорительных элементов – на схеме справа

Существует несколько широко используемых моделей параллельного программирования. Мы рассмотрим три из них подробнее: «начальник/подчиненный», обмен сообщениями и распределенные массивы.

Модель «начальник/подчиненный» является наиболее простой моделью параллельного программирования. Она используется в случаях, когда задачу можно разбить на множество относительно независимых задач, которые обрабатываются исполнителями без явной коммуникации друг с другом. Основное ограничение модели «начальник/подчиненный» заключается в том, что каждый подчиненный осуществляет коммуникацию только через отправку информации менеджеру. Это ограничение является очень сильным и значительно упрощающим. Более того, оно очень надежное в использовании, и множество параллельных программ с небольшим количеством процессоров (т. е. рабочих станций) применяют данную модель. К сожалению, эту модель можно использовать только в приложениях, не требующих коммуникации между процессорами. Кроме того, центральный начальник не допускает масштабирования данного метода на большое количество процессоров.

Модель обмена сообщениями во многих аспектах является противоположностью модели «начальник/подчиненный». Модель обмена сообщениями подразумевает, что любой процессор может посылать и получать сообщения от любого другого процессора. Инфраструктура модели обмена сообщениями довольно проста. Эта инфраструк-

тура среди специалистов по параллельным вычислениям, как правило, используется посредством интерфейса передачи сообщений (MPI) (http://www.mpi–forum.org). Модель передачи сообщений требует, чтобы каждый процессор обладал уникальным идентификатором (P_{ID} или `Pid`) и знал, сколько других процессоров работает вместе над задачей (Np или *Np*). В соответствии с терминологией MPI эти понятия носят название «ранг» процессора и «размер» «мира» MPI. Любая параллельная программа может быть реализована с использованием модели передачи сообщений. Основной недостаток этой модели заключается в том, что программист должен управлять каждым индивидуальным сообщением в системе; таким образом, модель требует написания большого дополнительного кода, и ее отладка может представлять собой чрезвычайную трудность. Тем не менее, существуют определенные параллельные программы, которые можно реализовать только при помощи модели обмена сообщениями.

Модель на основе распределенных массивов представляет собой компромисс между двумя вышеуказанными моделями. Распределенные массивы накладывают дополнительные ограничения на программу, что позволяет относительно упростить написание сложных программ. Во многих отношениях эта модель параллельного программирования является самой естественной для MATLAB, так как она реализуется при помощи массивов, являющихся здесь основным типом данных. В нескольких словах, модель на основе распределенных массивов создает распределенные массивы, в которых каждый процессор владеет частью целого массива, или хранит ее. В массиве также хранится дополнительная информация, при помощи которой каждый процессор знает, какими частями массива обладают другие процессоры. Способ разбиения массивов между процессорами определятся картой размещения [Lebak et al. 2005]. Знание различных способов распределения на основе карт может быть полезным, даже если модель распределенных массивов не используется. Концепция разбиения массивов различными способами является одной из ключевых идей параллельных вычислений. Вычисления на распределенных массивах обычно выполняются согласно правилу «вычисляет владелец», которое означает, что каждый процессор отвечает за выполнение вычислений на данных, которые он хранит локально. Карты размещения могут стать довольно сложными и фактически выражать произвольные распределения.

6.5. Системные метрики

В предыдущих разделах были описаны способы характеристики параллельных приложений по отношению к традиционному параллельному процессору. Фактический выбор типа параллельного размещения представляет собой процесс условной оптимизации. Ограничения обеспечиваются приложением, и целью этого процесса является оптимизация метрики системного уровня. В данном разделе представлено описание некоторых основных системных метрик: производительность, эффективность, форм-фактор и затраты на программное обеспечение.

6.5.1. Производительность

Повышение производительности является основной причиной использования системы параллельных вычислений и соотносится со временем, затрачиваемым на об-

работку одного набора данных из серии наборов данных. Понятие производительности обычно включает в себя задержку и пропускную способность.

Задержка означает время, требующееся первому набору данных на прохождение цепочки обработки. Задержка главным образом ограничена тем, насколько быстро потребителю данных нужна информация. Некоторые типичные задержки для различных систем составляют:

- микросекунды: управление в реальном масштабе времени;
- миллисекунды: оператор в цикле;
- секунды: системы обеспечения качества;
- минуты: системы мониторинга;
- часы: системы архивации.

Процесс оптимизации начинается с выбора целевой задержки для предполагаемого приложения ($T_{latency}^{goal}$) и может быть выражен при помощи T_{input}. Например, установим произвольную целевую задержку

$$T_{latency}^{goal} / T_{input} \approx 10.$$

Пропускная способность представляет собой скорость, с которой элементы выполняемой работы могут быть обработаны. Как правило, данные должны обрабатываться с той же скоростью, с которой они поступают в систему; иначе, они начнут «нагромождаться» на одном из этапов обработки. Ключевым параметром здесь является необходимая пропускная способность по отношению к возможностям одного процессора. Для нашего примера установим произвольную целевую скорость отработки:

$$S_{latency}^{goal} = T_{latency}^{goal} / T_{comp} \approx 100.$$

6.5.2. Форм-фактор

Теоретически для выполнения приложения доступна произвольно большая вычислительная машина с неограниченным числом процессоров. К сожалению, фактически существующие параллельные машины имеют ряд ограничений по следующим аспектам:

Размер. физический объем параллельной машины должен соответствовать помещению, где она устанавливается.

Мощность. общая мощность, потребляемая параллельной машиной и ее системой охлаждения.

Тепловыделение. общее тепло, которое выделяет параллельная машина и которое должна нейтрализовать система охлаждения.

Каналы ввода/вывода. Количество и скорость каналов данных, подводимых к системе и отходящих от нее.

Выбор процессора. Как выбрать процессор для приложения? После того, как принято решение строить систему параллельной обработки, необходимо выбрать физическое оборудование. Чаще всего, этот выбор продиктован вышеуказанными требованиями форм-фактора. Например, имеется помещение для размещения пяти блоков с обеспечением необходимого питания и охлаждения. Предположим, каждый блок имеет 14 посадочных мест. Для каждого блока необходимо предусмотреть плату буферного ввода, одну основную управляющую машину (и резервную) и центральное устройство хранения данных. При этом остается десять слотов, в каждый из которых можно вставить двухпроцессорный узел (см. рисунок 6.14). Получаем

$$N_p^{real} = (5 \text{ блоков})\ (10 \text{ мест/блок})\ (2 \text{ процессора/место}) = 100$$

На данном этапе выбор сделан, и теперь те, кто реализует приложение, должны определить соответствие необходимой функциональности выбранному процессору. Порядок осуществления этой задачи прежде всего включает в себя обеспечение первоначальной программной реализации с некоторой оптимизацией под оборудование. Если при этой реализации невозможно соответствовать требованиям производительности, обычно идут на компромисс и выясняют, обеспечит ли уменьшение параметров алгоритма (например, объема обрабатываемых данных) соответствие целевой производительности. В итоге, может понадобиться совершенно другой алгоритм, который совместно с героическими усилиями программистов поможет выжать всю возможную производительность из системы.

Compute (2)	Compute (2)	Compute (2)	Compute (2)	Compute (2)
Compute (2)	Compute (2)	Compute (2)	Compute (2)	Compute (2)
Compute (2)	Compute (2)	Compute (2)	Compute (2)	Compute (2)
Compute (2)	Compute (2)	Compute (2)	Compute (2)	Compute (2)
Compute (2)	Compute (2)	Compute (2)	Compute (2)	Compute (2)
Storage	Storage	Storage	Storage	Storage
Control Spare	Control Spare	Control Spare	Control Spare	Control Spare
Control	Control	Control	Control	Control
IO	IO	IO	IO	IO

Перевод
Compute – вычислительный узел;
Storage – хранилище;
Control spare – резервное управление;
Control – управление;
IO – ввод/вывод.

Рисунок 6.14. Пример вычислительной стойки

Традиционная стойка параллельной обработки. Каждая стойка состоит из пяти блоков. Каждый блок имеет 14 посадочных мест. Четыре посадочных места необходимо отвести под ввод/вывод, управление (и резерв) и хранение данных. В результате весь блок может вместить 100 процессоров

6.5.3. Эффективность

Эффективность является долей пиковых возможностей системы, достигаемой приложением. Значение T_{comp} определяет коэффициент эффективности на одном процессоре по отношению к теоретической пиковой производительности (например, $e_{comp} \approx 0.2$). Аналогичные коэффициенты эффективности характеризуют пропускную способность (например, $e_{comm} \approx 0.5$) и память (например, $e_{mem} \approx 0.5$). Выделяют два основных следствия из данных коэффициентов эффективности. Если требуемая эффективность намного превышает данные значения, это может означать, что необходимо заменить оборудование (например, на непрограммируемые аппаратные средства, сети с большей пропускной способностью или память большей емкости). Если необходимая эффективность гораздо ниже этих значений, это означает, что можно использовать более гибкие среды более высокого уровня программирования, что значительно сократит сроки и стоимость работы.

Реализация программного обеспечения обычно раскладывается на две части. Первая – способ реализации кода на каждом отдельном процессоре. Вторая – способ реализации коммуникации между процессорами. Типичные категории для сред последовательной реализации приведены ниже.

Машинный ассемблер. Набор команд конкретного процессора. Ассемблер обеспечивает высочайшую производительность, но требует значительных трудозатрат и высокой квалификации, а также не обеспечивает переносимости программного обеспечения.

Процедурные языки с оптимизированными библиотеками. Такие как C и Fortan, которые используются совместно с BLAS (базовый пакет подпрограмм линейной алгебры). В результате также можно получить эффективный код, при этом требуется меньше трудозатрат, и не нужно иметь особую квалификацию, а также код является настолько переносимым, насколько таковой является лежащая в основе реализации библиотека.

Объектно-ориентированные языки с оптимизированными библиотеками. Такие как C++, используемый совместно с оптимизированными библиотеками C++ (например, VSIPL++ [Lebak et al. 2005]). Этот метод дает производительность, сравнимую с процедурными языками, требует такой же квалификации, однако гораздо меньше трудозатрат. При объектно-ориентированном подходе код является более или менее переносимым по сравнению с процедурным подходом, что зависит от доступности компилятора и библиотек.

MATLAB. Производительность обычно значительно ниже, чем у процедурных языков, но и потребность в трудозатратах гораздо меньше. Переносимость ограничена процессорами, поддерживающими MATLAB.

Типичные категории для сред параллельной реализации перечислены ниже.

Прямой доступ к памяти (DMA/ПДП). Особый протокол, зависящий от конкретной сети и процессора, позволяющий одному процессору записывать данные в память другого процессора. Он схож с обменом сообщениями, но реализуется на более низком уровне. DMA обеспечивает высокую производительность, однако требует больших трудозатрат и высокой квалификации, а также не обеспечивает переносимости ПО.

Обмен сообщениями. Такие как интерфейс передачи сообщений (MPI). Это протокол для пересылки сообщений между процессорами. В результате получаем эффективный код, при этом требуется меньше трудозатрат, и не нужно иметь особую квалификацию, а также код является настолько переносимым, насколько таковой является лежащая в основе реализации библиотека.

«начальник/подчиненный». Использование либо высокоуровневого метода, такого как планировщик заданий, либо низкоуровневого метода на основе потоков с использованием OpenMP или pthreads.

Распределенные массивы. Такие как используемые в Unified PARALLEL C (UPC), Co-Array Fortran (CAF), Parallel VSIPL++ и pMATLAB. Этот подход создает распределенные массивы, что позволяет записывать сложные перемещения данных в сжатой форме. Производительность, как правило, сравнима с обменом сообщениями.

Приблизительная оценка эффективности производительности вышеуказанных методов ПО дана в таблице 6.2 (для получения более подробной информации см. [Kerner 2004, Kerner 2006]). Второй столбец (обозначенный e_{comp}) показывает приблизительную относительную эффективность производительности однопроцессорной реализации при использовании подхода, указанного в первом столбце. Вторая строка (обозначенная e_{comm}) дает приблизительную относительную эффективность пропускной способности при использовании метода, указанного в первой строке. Внутренняя часть матрицы из строк и столбцов показывает произведение этих двух видов эффективности и отражает диапазон производительности, на которую воздействует реализация программного обеспечения.

Таблица 6.2. Оценка эффективности реализации программного обеспечения

В первом столбце перечисляются методы написания последовательного кода. Во втором столбце указана приблизительная оценка эффективности (e_{comp}) последовательного метода. Первая строка столбцов 3, 4, 5 и 6 перечисляет различные модели коммуникации для параллельной реализации. Во второй строке этих столбцов приведена приблизительная оценка эффективности коммуникации (e_{comm}) для этих моделей. В остальных ячейках таблицы указана комбинированная эффективность ($e_{comp}e_{comm}$). Пустые ячейки оставлены для последовательного кода и моделей коммуникации, которые вместе используются редко

Serial Code	Serial e_{comp}	DMA	Comm Message Passing	Model Manager/Worker	Distributed Arrays
e_{comm}	–	0.8	0.5	0.4	0.5
Assembly	0.4	0.36			
C/Fortran	0.2	0.16	0.1	0.08	0.1
C++	0.18	0.14	0.09	0.07	0.09
MATLAB	0.04		0.02	0.016	0.02

Перевод
Serial code — последовательный код;
Serial — последовательный;
Comm. model — модель коммуникации;

Message passing – обмен сообщениями;
Manager/worker – «начальник/подчиненный»;
Distributed arrays – распределенные массивы.

Значимость произведения $e_{comp}e_{comm}$ можно проиллюстрировать следующим образом. Общая скорость работы может быть записана как

$$R(Np) = W/T = W/(T_{comp}(Np)+T_{comm}(Np))$$

Замена $T_{comp}(Np) = W/e_{comp}R_{peak}Np$ и $T_{comm} = D/e_{comm}BW_{peak}Np$ дает

$$R(Np) = \frac{e_{comp}e_{comm}R_{peak}Np}{e_{comm} + e_{comm}(D/W)(R_{peak}/BW_{peak})}.$$

где D/W – отношение коммуникация–вычисление, свойственное приложению, и R_{peak}/BW_{peak} – отношение вычисление–коммуникация для вычислительной машины. Оба отношения фиксированы для поставленной задачи и архитектуры. Таким образом, основная «педаль» для повышения общей скорости вычислений, доступная программисту, – это комбинированная эффективность метода написания последовательного кода и модели коммуникации.

6.5.4. Затраты на программное обеспечение

Затраты на программное обеспечение являются основной составляющей стоимости параллельного приложения. Существует много методов реализации системы параллельного программного обеспечения, отличающихся по производительности, трудозатратам и переносимости. Основной подход к моделированию затрат на программное обеспечение предоставляется моделью стоимости и составления кода (Code and Cost Modeling, CoCoMo) [Boehm et al. 1995]:

$$\text{Трудозатраты разработчика [сутки]} \approx \frac{\text{(Общ. SLOC)} * \text{(новая доля кода)} + 0{,}05\ \text{(повторно использованная доля кода)}}{\text{SLOC / сутки}}$$

Из формулы видно, что трудозатраты почти линейны относительно общего количества написанных строк кода ПО (SLOC). SLOC обычно определяется как каждая не содержащая комментариев и непустая строка в программе. Это показывает, что существует три очевидных пути уменьшить трудозатраты, связанные с реализацией программы. Приведем их ниже.

Увеличение повторного использования. Использование кода, который был написан ранее, гораздо дешевле, чем написание кода с нуля. Например, использование некоторого инструментального пакета MATLAB дешевле, чем написание такой же функциональности с нуля.

Повышение уровня абстрагирования. Если одна и та же функциональность может быть описана с использованием меньшего количества строк кода, стоимость такого кода будет ниже. В общем случае языкам высшего уровня, таким как MATLAB, требуется меньше строк кода, чем языкам более низкого уровня, например, C.

Увеличение скорости кодирования. Если среда позволяет написать больше строк кода в заданный отрезок времени, это позволит снизить стоимость кода. Например, полностью интегрированная среда разработки программ (IDE), как правило, позволяет разработчикам писать код быстрее.

Приблизительная оценка влияния на программное обеспечение вышеописанных методов приведена в таблице 6.3 (за подробной информацией обращайтесь к [Kerner 2004, Kerner 2006]). Количество SLOC для заданной программы сильно меняется от разработчика к разработчику. Однако SLOC, нормализованное по отношению к определенной реализации данного разработчика, взятой в качестве образца, является более точным. Таким образом, данные в таблице 6.3 должны рассматриваться как относительные для данного разработчика, реализующего одну и ту же программу различными способами. В первом столбце указан размер кода относительно эквивалентного кода, написанного на процедурном языке (например, C). В следующем столбце указана типичная скорость (SLOC / сутки), при которой пишутся строки в данной среде. Столбец, обозначенный «последовательно», показывает скорость..., деленную на относительный размер кода ... и дает эффективную относительную скорость выполненной работы, нормализованную по отношению к процедурной среде. Строка, обозначенная «код», показывает оценку увеличения размера кода при переходе от последовательного кода к параллельному для различных методов параллельного программирования. Строка, обозначенная «трудозатраты», показывает относительное увеличение трудозатрат, связанных с каждой из строк параллельного кода. Во внутренней части матрицы приведены произведения этих значений, которые дают представление об эффективном уровне трудозатрат для каждой среды последовательного и параллельного программирования. Например в случае объектно-ориентированной среды в среднем каждая строка выполняет работу двух строк процедурного языка. Кроме того, типичный разработчик может писать код в последовательной среде со скоростью 25 строк в сутки. Если код, написанный в этой среде, переводится в параллельный при помощи обмена сообщениями, ожидается, что общий размер кода увеличится с коэффициентом 1,5. Более того, скорость, с которой эти дополнительные строки записываются в код, уменьшится с коэффициентом два, так как написать их сложнее. В результате общая скорость параллельной реализации составит 25 эффективных процедурных строк (т. е. C) в сутки.

На рисунке 6.15 показаны теоретически объединенные данные из таблиц 6.2 и 6.3 для гипотетической 100-процессорной системы и иллюстрируются различные компромиссы между производительностью и трудозатратами при использовании различных моделей программирования.

Таблица 6.3. Оценка скорости написания кода программного обеспечения

В первом столбце указан размер кода относительно эквивалентного кода, написанного на процедурном языке (например, С). В следующем столбце указана типичная скорость (SLOC / сутки), при которой пишутся строки в данной среде. Столбец, обозначенный «последовательно», показывает скорость, деленную на относительный размер кода, и дает эффективную относительную скорость выполненной работы, нормализованную по отношению к процедурной среде. Строка, обозначенная «код», показывает оценку увеличения размера кода при переходе от последовательного кода к параллельному для различных методов параллельного программирования. Строка, обозначенная «трудозатраты», показывает относительное увеличение трудозатрат, связанных с каждой из строк параллельного кода. Внутренняя часть матрицы отображает произведения всех этих параметров и дает эффективную норму трудозатрат для каждой среды последовательного программирования и параллельного программирования, заданных при помощи (SLOC/day)/(RelativeSLOC)/(1+(Expansion Factor - 1)(Effort Factor))

Serial Code	Relative SLOC	SLOC/ Day	Serial	DMA	Comm Message Passing	Model Manager/ Worker	Distrib-uted Arrays
Code	–	–	1	2	1.5	1.05	1.05
Effort	–	–	1	2	2	2	1.5
Assembly	3	5	1.6	0.5			
C/Fortran	1	15	15	5	8	14	14
C++	1/2	25	50	17	25	45	47
MATLAB	1/4	40	160		80	140	148

Перевод
Serial code – последовательный код;
Relative SLOC – относительный SLOC;
SLOC/day – SLOC / сутки;
Comm. model– модель коммуникации;
Serial – последовательная;
Message passing – обмен сообщениями;
Manager/worker – «начальник/подчиненный»;
Distributed arrays – распределенные массивы.

6.5.5. Продуктивность программного обеспечения

В итоге целью любого усилия является продуктивность. Однако определение продуктивности в каждом конкретном случае требует более глубокого анализа. Экономическое определение продуктивности:

$$\Psi = \text{полезность / стоимость},$$

где Ψ = продуктивность

В случае программного обеспечения стоимость определить довольно легко, она пропорциональна трудозатратам разработчика. Если данных по трудозатратам разработчика нет, размер кода может выступить в качестве разумного приближения значения трудозатрат [Boehm et al. 1995]. Полезность – понятие менее очевидное. Экономисты определяют его как относительную выгоду от товара или услуги. Полезность используется для описания сущностей, которые нельзя легко обменять. Например, улучшение программы является выгодой только для этой программы. Это улучшение имеет полезность только для этой программы, но не для другой.

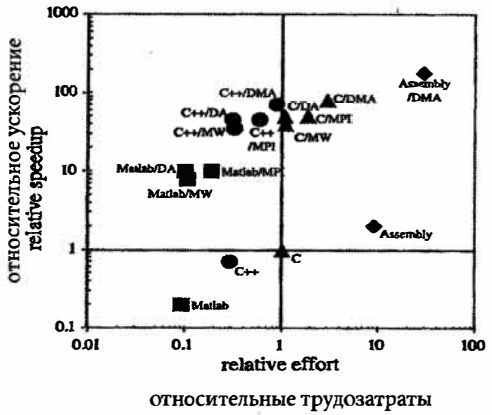


Рисунок 6.15. Ускорение в зависимости от трудозатрат

Оценка относительного ускорения (сравнение с последовательным C) на гипотетической 100-процессорной системе изображено по отношению к оценке относительных трудозатрат (сравнение с последовательным C). Изображены языки: ассемблер, C (процедурный), C++ (объектно-ориентированный) и MATLAB (высокого уровня). Модели взаимодействия включают прямой доступ к памяти (DMA), интерфейс передачи сообщений (MPI), «начальник/подчиненный» (MW) и распределенные массивы (DA)

К счастью, в особом контексте данной книги, параллельном программировании на MATLAB, полезность определить легко. Полезность – это ускорение, достигнутое посредством применения параллельного программирования. Такая параллельная продуктивность составляет

$$\Psi_{\parallel} = \frac{\text{ускорение}}{\text{трудозатраты параллельного программирования}}$$

Если использовать данные определения, самостоятельный разработчик и разработчик, получающий квалифицированную помощь, упомянутые в предисловии, получают кривые продуктивности, показанные на рисунке 6.16. В обоих случаях, каждый разработчик поддерживает постоянную продуктивность, однако разработчик, получающий квалифицированную помощь, в 100 раз продуктивнее, чем действующий самостоятельно.

Ту же идею можно применить к средам параллельного программирования для получения относительной продуктивности для традиционной среды программирования, такой как последовательный язык С. Используя данные рисунка 6.15, можно оценить продуктивность, взяв оценку ускорения (относительно последовательного языка С) на гипотетической 100-процессорной системе и разделив ее на относительные трудозатраты программирования. Результаты приведены в таблице 6.4. Неудивительно, что метод распределенных массивов оказался самым продуктивным в любом из языков. Кроме того, языки высокого уровня, такие как С++ и MATLAB, являются наиболее продуктивными. Любопытно, что различие между MATLAB (меньше трудозатраты) и языком С++ (выше производительность) исчезает при вычислении продуктивности.

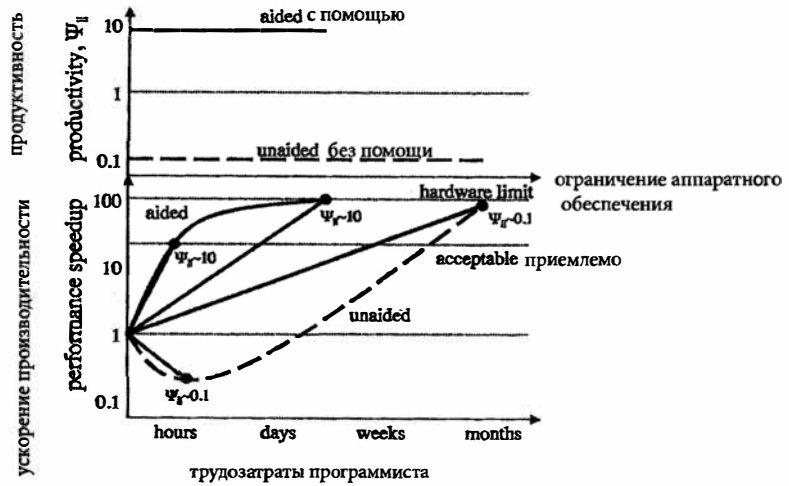


Рисунок 6.16. Параллельная продуктивность в зависимости от трудозатрат
Нижние графики показывают параллельное ускорение относительно трудозатрат разработчика для разработчика, получающего помощь эксперта и работающего без помощи экспертов (самостоятельно). Параллельная продуктивность $\Psi_{\|}$ вычисляется в различных точках. Верхние графики показывают общую параллельную продуктивность в ходе проекта

Таблица 6.4. Приблизительная относительная продуктивность

Оценка относительной продуктивности различных методов параллельного программирования (сравнительно с языком C) на гипотетической 100-процессорной системе. В первом столбце указан подход при составлении последовательного кода. Остальные столбцы показывают комбинированные воздействия различных сред последовательного и параллельного программирования. Как правило, методы распределенных массивов являются наиболее продуктивными.

Serial Code	Serial	DMA	Comm Message Passing	Model Manager/Worker	Distributed Arrays
Assembly	0.2	5			
C/Fortran	1	25	25	35	50
C++	2	75	75	100	150
MATLAB	2		50	75	100

Перевод
Serial code – последовательный код;
Serial – последовательный;
Comm. model– модель коммуникации;
Message passing – обмен сообщениями;
Manager/worker – «начальник/подчиненный»;
Distributed arrays – распределенные массивы.

Список литературы

1. *[Amdahl 1967] C.M. Amdahl, Validity of the single-processor approach to achieving large scale computing capabilities, Proceedings of AFIPS Conference, 1967, pp. 483–485.*
2. *[Boehm et al. 1995] B. Boehm, B. Clark, E. Horowitz, R. Madachy, R. Shelby, and C. Westland, Cost Models for Future Software Life Cycle Processes: COCOMO 2.0, Annals of Software Engineering, 1995.*
3. *[DeLuca et al. 1997] Cecelia DeLuca, Curtis Heisey, Robert Bond, and James Daly, A portable object-based PARALLEL library and layered framework for real-time radar signal processing. In Proc. 1st Conf. International Scientific Computing in Object-Oriented PARALLEL Environments (ISCOPE 97), pp. 241–248.*
4. *[Hennessy & Paterson 2006] John Hennessy and David Paterson, Computer Architecture: A Quantitative Approach, The Morgan Kaufmann Series in Computer Architecture and Design, 2006.*
5. *[Kepner 2004] Jeremy Kepner (editor), Special issue on HPC Productivity, International Journal of High Performance Computing Applications, Vol. 18, No.4, 2004.*

6. *[Kepner 2006] Jeremy Kepner (editor), High Productivity Computing Systems and the Path Towards Usable Petascale Computing: User Productivity Challenges, CTWatch Quarterly, Vol. 2, No. 4A, November 2006.*

7. *[Kuck 1996] David Kuck, High Performance Computing: Challenges For Future Systems, Oxford, UK: Oxford University Press, 1996.*

8. *[Lebak et al. 2005] James Lebak, Jeremy Kepner, Henry Hoffmann, and Edward Rutledge, PARALLEL VSIPL++: An open standard software library for highperformance PARALLEL signal processing. Proceedings of the IEEE 93(2), pp. 313– 330.*

9. *[Love man 1993] D.B. Loveman, High Performance Fortran. PARALLEL and Distributed Technology: Systems and Applications, IEEE 1(1), 1993.*

10. *[Luszczek et al. 2006] Piotr Luszczek, Jack Dongarra, and Jeremy Kepner, Design and Implementation of the HPC Challenge Benchmark Suite, CT Watch, Vol. 2, No. 4A, November 2006.*

11. *[Von Neumann 1945] John Von Neumann, First Draft of a Report on the EDVAC, 1945.*

12. *[Weinberg et al. 2005] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely, Quantifying Locality in the Memory Access Patterns of HPC Applications, Supercomputing 2005.*

13. *[Zose11993] M.E. Zosel, High Performance Fortran: An Overview. Compcon Spring 93, Digest of Papers, San Francisco, CA, February 22–26, 1993.*

Часть III

Анализ практических примеров

Глава 7
Анализ параллельных приложений

Резюме

За последние несколько десятилетий диапазон приложений, в которых применяются параллельные вычисления, заметно расширился. На первый взгляд, способы параллельной реализации этих приложений представляются очень разнообразными. Тщательный анализ показывает, что эти приложения можно разделить на категории по типу параллелизма, применяющегося в них, пространственной и временной локальности их вычислений и нагрузке на иерархию устройств памяти системы параллельных вычислений. Более того, эти категории могут быть представлены относительно простыми вычислениями, отражающими ключевые элементы параллельных вычислений. HPC Challenge benchmark (набор тестов производительности) представляет собой коллекцию этих стандартных вычислений. Сравнение и сопоставление этих тестов позволяет выявить некоторые важные отличия реальных параллельных приложений. Определив, к какой категории принадлежит параллельное вычисление, разработчик может быстрее принимать эффективные решения в отношении разработки параллельной программы.

7.1. Исторический обзор

Исследование мира параллельных приложений – чрезвычайно трудная задача. Описать каждое параллельное приложение невозможно, и как же в этом случае выбрать из них подмножество? Одним из способов является взгляд в прошлое на то, как развивалась и изменялась данная область в последние десятилетия.

Исторически концепция «организации вычислений на супер-ЭВМ» родилась в правительственных исследовательских лабораториях на заре эры вычислений (см. рисунок 7.1). В этих лабораториях целевого назначения основное внимание уделялось разработке приложений в национальных интересах: разработка вооружений, имитация наземных и воздушных судов, взлом кодов. К удивлению, этот «узкий» набор приложений имеет широкий диапазон требований к вычислительным средствам. В готовых программах используются различные математические понятия (непрерывность и дискретность), алгоритмы (физика и теория чисел) и разнообразные языки (Fortran и C).

Первые системы, как правило, состояли из небольшого количества векторных процессоров. Каждый векторный процессор, как правило, загружал вектор и затем вызывал одну команду, которая применялась ко всему вектору сразу.

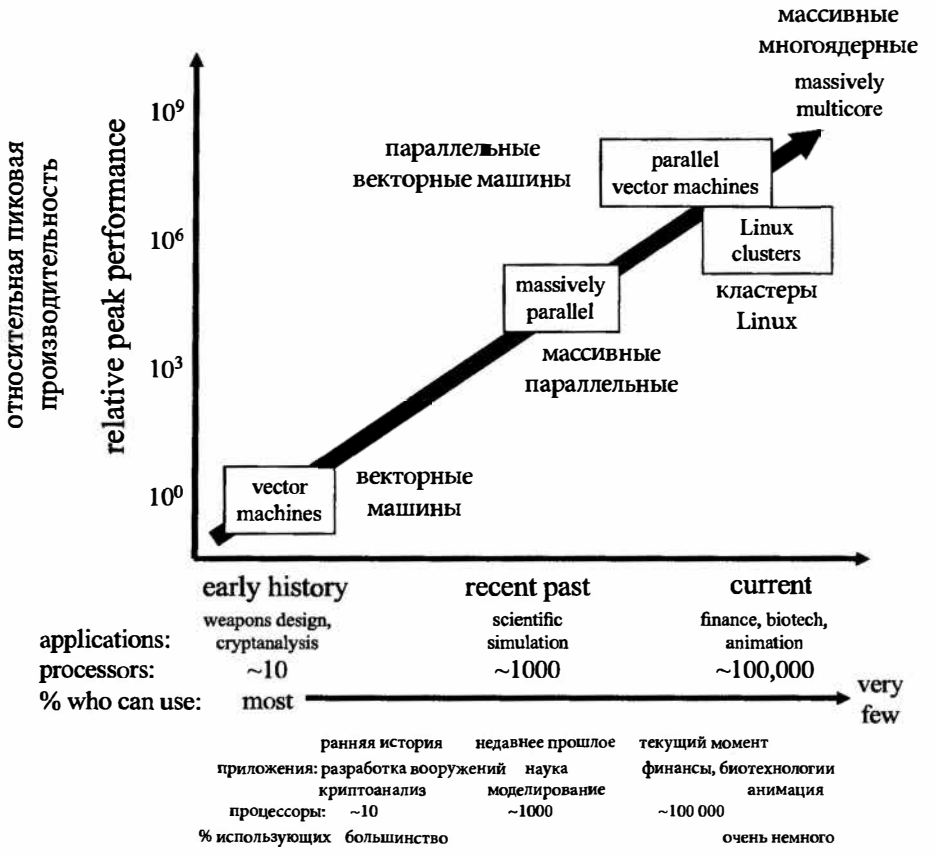


Рисунок 7.1. Эволюция вычислений на супер-ЭВМ

История началась с векторных машин, которые были относительно просты в использовании. В недавнем прошлом появились массивно-параллельные компьютеры. В настоящее время стали распространенными кластеры Linux, более сложные в программировании, а массивно-многоядерные системы представляют собой крупнейшие в мире системы

Такие процессоры часто назывались процессорами «один поток команд, много потоков данных» (SIMD) [Flynn 1972]. Ключом к эффективному использованию этих систем было создание больших циклов `for`, которые распознавались векторизирующим компилятором и отображались на векторные процессоры.

Хотя область использования параллельных вычислений с тех пор значительно расширилась, эти первые приложения сохраняют свою важность и зачастую являются ключевыми факторами роста для больших систем параллельных вычислений во всем мире. Кроме того, эти приложения дали толчок множеству исследований в области улучшения программирования параллельных машин. Развитие технологий распределенных массивов, таких как POOMA [Cummings et al. 1998], GA++ [Nieplocha et al.

2002]' и UPC [El-Ghazawi et al. 2005], взяло начало от этих приложений. В недалеком прошлом концепция параллельных вычислений вышла за рамки национальных исследовательских лабораторий и стала основным элементом научных исследований во многих дисциплинах. Это привело к официальному признанию «вычислительной науки и инженерии» или «науки моделирования» во многих университетах. Некоторые области науки, получившие выгоду от супервычислений, включают в себя моделирование погоды и климата, науку о материалах, химию и астрофизику. Примечательно, что, хотя сфера использования параллельных вычислений расширялась, дабы охватить весь диапазон научных вопросов, требования к ним почти не изменились с самых ранних дней. Самыми большими изменениями, произошедшими в эту эпоху, были: внедрение массивно параллельного аппаратного обеспечения и широкое использование модели программирования MPI (вместе с сопутствующими трудностями программирования).

В настоящее время разнообразие приложений в области параллельных вычислений вышло за пределы исследований в область коммерческой деятельности. Биоинформатика, финансовый инжиниринг, компьютерная анимация и интеллектуальный анализ данных (data mining) – вот неполный перечень «новых» приложений, продвигающих параллельные вычисления. И снова, вместо того чтобы расширять пространство вычислительных требований, эти приложения сузили пространство алгоритмов (прежде всего, массово-параллельного) и оборудования (прежде всего, серийные кластеры и параллельные рабочие станции), однако они существенно расширили пространство программного обеспечения (MATLAB, Java, Python) и увеличили необходимость в средах параллельного программирования, которые можно было бы быстро изучить.

Технологически тренды, ознаменовавшие эволюцию параллельных вычислений, включают следующее:

- увеличение пиковой производительности более чем в миллиард раз;
- смена основной области применения от нескольких алгоритмов, имеющих важный национальный интерес (написанных на языках C и Fortran), к приложениям, охватывающим весь диапазон науки и промышленности (написанным на многих языках);
- увеличение количества процессоров от нескольких единиц до нескольких тысяч и соответствующее увеличение общей трудоемкости программирования;
- последним шагом в расширении параллельных вычислений стало развитие многоядерных процессоров, сделавшее использование параллельных вычислений повсеместным. Сейчас каждая программа представляет собой параллельную программу. К счастью, метод распределенных массивов с акцентом на локальность данных, одинаково хорошо работает как на многоядерных процессорах, так и на больших серийных кластерах.

7.2. Характеристика пространства приложений

Из краткого исторического обзора, представленного в предыдущем разделе, видно, что за последние десятилетия изменилось многое, но в то же время многое осталось неизменным. В частности, с самых ранних дней было известно, что делает параллельное приложение «сложным» или «простым». Точнее говоря, простые прило-

жения чаще всего бывают массово-параллельными с регулярным доступом к памяти и совершают много операций над каждым элементом данных, прочитанным из памяти. Сложные приложения чаще всего требуют межпроцессорной коммуникации нерегулярного доступа к памяти и совершают мало операций над каждым элементом данных, прочитанным из памяти. Таким образом, с точки зрения параллельного программирования, нас меньше интересуют особенности приложения, но больше интересует «простота» или «сложность» программы. Оценка простоты или сложности может осуществляться с трех точек зрения. Первая точка зрения ориентирована на аппаратное обеспечение и рассматривает то, как программа нагружает иерархию устройств памяти оборудования. Вторая точка зрения ориентирована на алгоритм и рассматривает пространственную и временную локальность шаблонов доступа к памяти в алгоритме. Третья точка зрения ориентирована на разработчика и рассматривает иерархию логической памяти вычислительной машины и то, каким образом параллелизм приложения согласуется с этой иерархией.

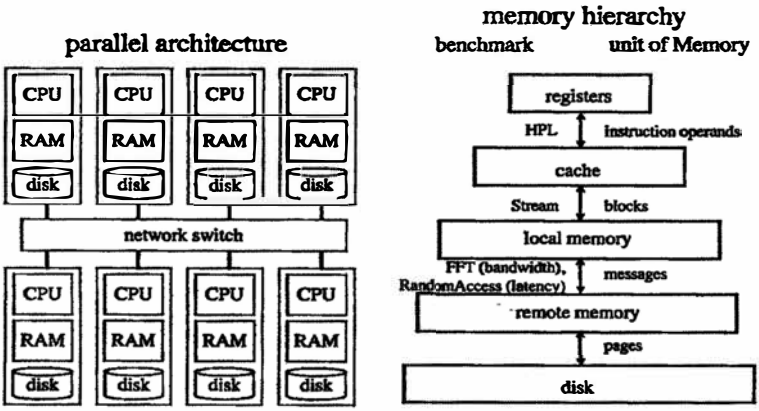
7.2.1. Иерархия физической памяти

Понятие иерархии устройств памяти появилось на заре разработки вычислительной техники. Наиболее важные характеристики включают в себя регистры процессора и кэш-памяти, локальные запоминающие устройства, удаленные запоминающие устройства и устройства хранения данных. Обычно в любой заданный момент времени пока выполняется программа, одна часть иерархии памяти будет загружена больше всего. В связи с этим все программы выполняются при 100% эффективности по отношению к какой-либо части иерархии памяти. В идеале эта часть находится ближе всех к процессору, так как процессор является единственной частью системы, которая может совершать полезную работу над данными. Однако если какая-либо другая часть иерархии памяти выполняется с максимальной скоростью (например, доступ к удаленному запоминающему устройству), то, вероятнее всего, процессор работает с гораздо меньшей эффективностью.

Простым способом охарактеризовать систему параллельных вычислений является написание набора тестов производительности, которые нацелены на определенную часть иерархии памяти. Выполнение этих программ на определенной вычислительной машине позволит измерить производительность этой системы. Сравнение производительности этих тестовых программ с производительностью реального приложения обеспечивает понимание того, какая часть иерархии памяти приложения подвергается воздействию больше всего. Другими словами, имеется возможность определить, на какую тестовую программу или программы приложение похоже больше всего.

На рисунке 7.2 показан набор тестов производительности HPC Challenge benchmarks в соотношении с канонической иерархией памяти. Тест HPL (High Performance LINpack) имеет склонность нагружать процессор (или обращения процессора к регистру). Тест Stream в основном сконцентрирован на пропускной способности к основной памяти. Тест FFT (БПФ – быстрое преобразование Фурье) нагружает пропускную способность сети, а RandomAccess измеряет задержку передачи сети. Измеряя скорость выполнения тестов при различных значениях *Np*, и сравнивая с соответствующими кривыми для реальных параллельных приложений, мы получаем четкое понимание того, насколько наше приложение похоже на HPL, Stream, FFT или

RandomAccess. Это сравнение далее определяет, какая технология параллельного программирования будет для приложения более эффективной.



Перевод
Parallel architecture – параллельная архитектура;
CPU – ЦПУ;
RAM – ОЗУ;
disk – диск;
network switch – сетевой коммутатор;
memory hierarchy – иерархия памяти;
benchmark – тест производительности;
unit of Memory – единица памяти;
registers – регистры;
instruction operands – операнды команд;
cache – кэш-память;
blocks – блоки;
local memory– локальная память;
bandwidth – пропускная способность;
latency – задержка;
messages – сообщения;
remote memory – удаленная память;
pages – страницы;
disk – диск.

Рисунок 7.2. Иерархия памяти и набор тестов производительности HPC Challenge
Набор тестов производительности HPC Challenge разработан таким образом, что каждый тест воздействует на определенную часть иерархической памяти параллельного компьютера

7.2.2. Пространственная/временная локальность

Другой способ характеристики параллельных приложений, который имеет более прямое отношение к их алгоритмам, рассматривает пространственную и временную

локальность приложений. Помните, что пространственная локальность – мера упорядоченности шаблонов доступа к памяти программы, а временная локальность является мерой количества операций, выполняемых над элементом данных, прочитанным из памяти. Посредством профилирования приложения можно фактически вычислить пространственный и временной показатель. На рисунке 7.3 эти показатели продемонстрированы для нескольких реальных приложений [Weinberg et al. 2005].

Эти приложения заполняют некоторую область в пространственно–временных координатах. Область, ограниченная набором тестовых программ HPC Challenge, имеет склонность охватывать все другие приложения. Точное расположение любого заданного приложения характеризуется тем, к какой тестовой программе пакета HPC Challenge оно находится ближе всего. Не удивительно, что научные приложения находятся ближе к FFT и HPL. Аналогично приложения из области анализа данных находятся ближе к Stream и RandomAccess. Более того, пространственная/временная локальность приложения может измениться во время его выполнения и может приближаться к тому или иному тесту в разные моменты времени.

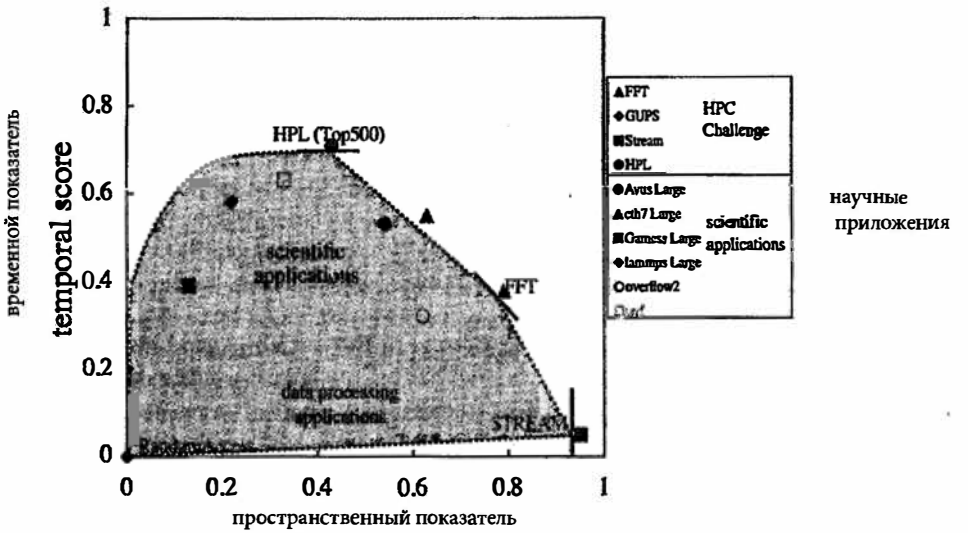


Рисунок 7.3. Сравнение пространственной и временной локальности
Пространственный/временной показатели некоторых приложений и пакета HPC Challenge, измеренные при помощи MetaSim [Weinberg et al. 2005]

7.2.3. Иерархия логической памяти

Способ наиболее эффективного программирования параллельного приложения в значительной степени зависит от того, каким образом необходимо перемещать данные по логической иерархии памяти системы. Для некоторых шаблонов перемещения данных код писать сложнее, чем для других. К счастью, существует лишь несколько основных шаблонов. Используя тесты производительности, которые используют данные шаблоны, можно сделать вывод в отношении простоты или сложности написания программы реального приложения по аналогии с подобными шаблонами перемещения данных.

В частности, эти шаблоны помогут определить, какие модели параллельного программирования наиболее эффективны в использовании для различных приложений.

В наборе тестов HPC Challenge используется широкий диапазон шаблонов перемещения данных (см. рисунок 7.4). Наиболее простым является шаблон массового параллелизма, который не требует перемещения данных между процессорами. Это шаблон теста Stream. Тест FFT представляет более регулярный шаблон взаимодействия «каждый к каждому» (all-to-all). В этом шаблоне каждый процессор отправляет регулярный предопределенный объем данных всем другим процессорам. Шаблон HPL подобен шаблону FFT, но в нем обычно задействуется один или более процессоров, рассылающих широковещательно данные одному или более подмножествам процессоров. Наконец, шаблон коммуникации RandomAccess является самым сложным, так как он отправляет сообщения между процессорами случайным образом.

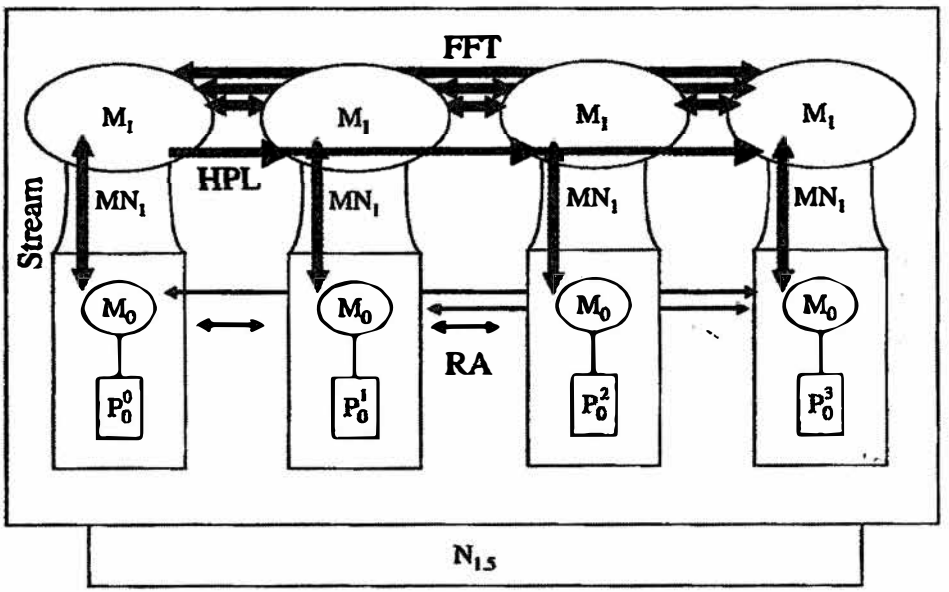


Рисунок 7.4. Коммуникация в иерархии

Шаблоны коммуникации набора тестов производительности HPC Challenge по отношению к иерархии Кека на стандартной вычислительной машине

7.3. Набор HPC Challenge: охват пространства приложений

В предыдущем разделе мы постарались обосновать идею того, что небольшое количество тестов может дать представление о параллельной производительности и характеристиках большого пространства приложений. В частности, был выбран пакет тестов HPC Challenge, так как он нацелен на различные части иерархии физической памяти, охватывает пространственную/временную локальность реальных приложений и задействует разнообразные шаблоны перемещения логических данных. В настоящем разделе представлены высокоуровневые детали этих тестов производительности. Каждый тест будет изучен более подробно в последующих главах.

7.3.1. Stream

Тест производительности Stream является самым простым из набора HPC Challenge. Он выполняет некоторое количество простых операций на N –элементных векторах **a**, **b** и **c**. Наиболее известной является операция Stream triadd:

$$\mathbf{a}(i) = \mathbf{b}(i) + \mathbf{q}\mathbf{c}(i),$$

где q – константа. Целью этого теста является измерение пропускной способности локальной памяти всех процессоров системы. Таким образом, каждый процессор выполняет вышеуказанную операцию на своей локальной части массивов без осуществления коммуникации с любым другим процессором. Типичное распределение данных для этого алгоритма: $\mathbf{a} : R^{P(N)}$ (см. рисунок 7.5).

Stream является представителем многих массово-параллельных приложений, которые осуществляют регулярный доступ к массивам данных. С помощью этого теста можно хорошо представить многие задачи анализа данных (например, обработку изображений). Stream легко реализовать в любой модели параллельного программирования.

7.3.2. FFT

Быстрое преобразование Фурье (БПФ) – важное вычисление во многих приложениях. Оно является основой обработки сигналов, так как позволяет эффективно выполнять преобразование данных о временных рядах x в частотные данные **z**. Преобразование БПФ также важно в численных решениях дифференциальных уравнений в частных производных, где используются псевдоспектральные методы.

Преобразование БПФ является быстрым методом вычисления дискретного преобразования Фурье (ДПФ):

$$z(k) = \sum_{i=1}^{N} x(i) e^{2\pi\sqrt{1-k}},$$

где $\mathbf{x}, \mathbf{z} : C^N$. Часто эта операция обозначается как

$$z = FFT(x).$$

Преобразование БПФ не является операцией тривиального параллелизма. Фактически каждый элемент результатов $Z(i)$ зависит от каждого значения **x**. Обычно параллельная операция преобразования БПФ использует тот факт, что одномерное преобразование БПФ можно вычислить предварительным преобразованием входного вектора в матрицу, выполнением БПФ на строках, умножением на некоторые константы и затем выполнением преобразования БПФ на столбцах. Результирующая матрица затем преобразуется обратно в вектор. Такой алгоритм делает параллелизм в программе достаточно ясным. Каждая операция преобразования БПФ над строкой или столбцом может выполняться независимо от других. Однако данные матрицы необходимо перераспределять между каждым из шагов

$$\mathbf{Z}=\mathbf{X},$$

где $\mathbf{X} : C^{P(M_i)\times M_j}$, $\mathbf{Z} : C^{M_i \times P(M_j)}$ и $N = M_i M_j$. Такое перераспределение требует, чтобы каждый процессор отправлял данные другому процессору и нагружал пропускную спо-

собность сети межпроцессорной коммуникации параллельной машины (см. рисунок 7.5). Это перераспределение является классическим примером типа операции глобальной коммуникации, которую легко представить с использованием распределенных массивов.

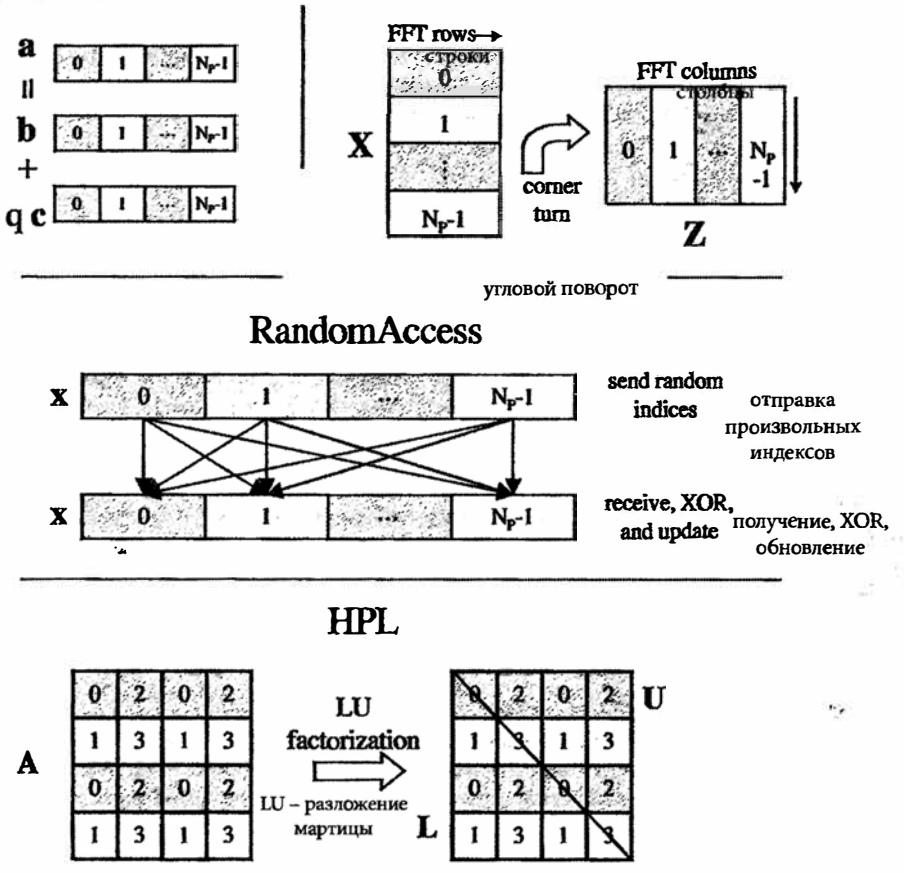


Рисунок 7.5. Шаблоны коммуникации пакета HPC Challenge

7.3.3. RandomAccess

Тест Stream является представителем многих приложений анализа данных, но в то же время существуют другие приложения, имеющие более сложные шаблоны доступа к памяти. Например, простая операция вычисления гистограммы последовательности случайных чисел по существу требует произвольного обращения к памяти. Тест RandomAccess создан для представления этого класса операций, которые можно найти в базах данных, анализе графов, анализе кода и технологий имитации/моделирования на основе частиц методом Монте-Карло. Основная операция теста:

$$x(i) = XOR(x(i), r),$$

где x : Z^N – таблица целых чисел, r : Z– случайное целое число и $i(r)$ = AND(r, N-1)– номер обновляемой ячейки таблицы. Обычно таблица распределена между процессорами **t** : $Z^P(N)$ (см. рисунок 7.5), при этом каждый процессор генерирует последовательность из r. Для выполнения обновления необходимо, чтобы каждый процессор обращался к индексу $i(r)$, определял, какой процессор владеет элементом таблицы $t(i)$, и отправлял соответствующее значение r этому процессору для выполнения обновления. Следствием данного параллельного метода является то, что каждый процессор направляет данные произвольным подмножествам процессоров. Такой шаблон коммуникации влияет на задержку соединений между процессорами. Кроме того, для этого шаблона трудно писать код в большинстве моделей параллельного программирования.

7.3.4. HPL

Тест производительности High Performance LiNpack (HPL) имеет долгую историю и хорошо известен как тест, ассоциирующийся со списком «Top 500». В данный список входит 500 самых быстрых вычислительных машин в мире в соответствии со скоростью выполнения ими теста HPL. В основе теста HPL лежит вычисление вектора $\boldsymbol{x}$: R^M, который является решением системы линейных уравнений:

$$\mathbf{Ax} = \mathbf{b}$$

со случайно генерирующимися входными данными **A** : $R^{M \times M}$ и **b** : R^M. Алгоритм, использующийся для нахождения x, сначала раскладывает матрицу на две матрицы-сомножителя: нижне-треугольную матрицу **L** и верхне-треугольную матрицу **U**. Эти матрицы затем используются для нахождения решения **x** с использованием исключений Гаусса. В вычислениях преобладают многочисленные перемножения матриц на матрицы, находящиеся на шаге LU-разложения матрицы, которые больше всего нагружают процессор в параллельной системе.

Наиболее высокопроизводительные реализации HPL используют сложное двумерное блочно-циклическое распределение матрицы **A** : $R^{P_{bc}(M) \times P_{bc}(M)}$ (см. рисунок 7.5). Это распределение обеспечивает оптимальную равномерность нагрузки и позволяет каждому процессору выполнять приблизительно одинаковый объем работы по мере выполнения вычислений. Побочным эффектом такого распределения является тот факт, что, хотя общий объем перемещаемых данных небольшой (по сравнению с вычислением), шаблоны коммуникации довольно сложные и требуют, чтобы каждый процессор определял подкласс процессоров для коммуникации по мере выполнения вычислений. Данный шаблон является сложным для написания кода в большинстве моделей параллельного программирования, однако здесь на помощь приходят распределенные массивы. Стоит упомянуть, что можно реализовать HPL, используя более простые распределения данных (например, **A** : $R^{M \times P(M)}$), однако эти распределения приведут к получению более низкой параллельной производительности.

7.4. Собственная производительность алгоритма

По структуре набор тестов производительности HPC Challenge представляет собой весьма разнообразные алгоритмы и обеспечивает отличную возможность подчеркнуть, как различные алгоритмы могут обладать различными параллельными

свойствами. Для оценки алгоритмов необходимо определить их следующие фундаментальные характеристики.

Основные структуры данных: массив(-ы) на каждой стадии программы, которые потребляют наибольший объем памяти и на которых выполняется большинство вычислений.

Сложность вычислений (W): общее количество выполняемых операций (или выполняемой работы) и их зависимость от алгоритмических параметров.

Количество операций на элемент данных ($W_{element}$): количество выполняемых операций (или выполняемой работы) на каждом элементе данных.

Степени параллелизма (DoP): параллелизм, свойственный алгоритму, и его связь с основными структурами данных.

Сложность коммуникации (D): общее количество перемещаемых элементов данных, необходимых для выполнения параллельного алгоритма.

Отношение вычисление–коммуникация (W/D): объем выполняемых вычислений на каждый перемещенный элемент данных.

Вышеуказанные значения для каждого теста обобщены в таблице 7.1.

7.4.1. Структура данных

Основные структуры данных для каждого теста были представлены в предыдущем разделе. Тесту Stream требуется три N-элементных вектора. Тест FFT также использует N-элементный комплекснозначный вектор, но преобразует его в матрицу из $M_1 \times M_2$ элементов, где $N = M_1M_2$. RandomAccess использует N-элементный вектор (или таблицу) целых чисел. Наконец, HPL использует матрицу $M \times M$, где $N = M^2$ – общее количество элементов в матрице. Структура всех тестов производительности позволяет выполнять их как масштабированную задачу, таким образом, N выбирается для использования большой доли общей памяти вычислительной машины. В общем, это означает, что $N \alpha Np$. На практике N/Np поддерживается неизменным, а N определяется умножением этого значения на Np.

7.4.2. Сложность вычислений

Часть теста Stream – triadd – выполняет одно умножение и одно сложение на каждый элемент данных ($W_{element} = 2$), в результате чего общая сложность вычислений составляет $W = 2N$. Аналогично RandomAccess совершает только одну операцию XOR на каждом элементе данных $W_{element} = 1$. Точная сложность вычислений БПФ незначительно отличается от реализации к реализации. Стандартное значение для БПФ на комплексных данных составляет $W = 5N\log2(N)$ с соответствующей работой на каждом элементе данных, равной $5\log_2(N)$. Для решения уравнения $\mathbf{Ax} = \mathbf{b}$ существует много алгоритмов; однако тесты производительности HPL требуют использовать LU-разложение матрицы с последующими исключениями Гаусса. В данном случае сложность вычислений составит $W = (2/3)M^3 = (2/3)N^{3/2}$, в результате чего работа на каждом элементе данных составляет $W_{element} = (2/3)\sqrt{N}$. Точная сложность вычислений HPL дана в последующей главе.

Согласно реализации, объем работы на элемент данных заметно варьируется среди тестов производительности HPC Challenge, от $W_{element} = 1$ для RandomAccess до $W_{element} = 2/3\sqrt{N}$ для HPL. По этим данным можно принципиально определить, на-

диционной процессорной системы с традиционной иерархией памяти ожидается, что реализация HPL будет самой эффективной, а тестов Stream и RandomAccess – самой неэффективной.

Таблица 7.1. Характеристики параллельных алгоритмов пакета HPC Challenge

Различные вычислительные характеристики тестов производительности HPC Challenge

Программа	Массив данных	W сложность вычислений (операций)	$W_{element}$ операции/ элемент	Степени параллелизма	D сложность коммуник. (отправлено элементов)	W/D вычисления/ коммуник. Опер./элемент
Stream	$a, b, c : R^{P(N)}$	$2N$	2	N	0	
FFT	$\mathbf{X} : C^{P(M_1)\times M_2}$ $\mathbf{Z} : C^{M_1\times P(M_2)}$	$5N\log2(N)$	$5\log2(N)$	$\min(M_1, M_2)$	N	$5\log2(N)$
RandomAccess	$\mathbf{t} : Z^{P(N)}$	N	1	N	N	1
HPL	$\mathbf{A} : R^{P_{bc}(M)\times P_{bc}(M)}$	$(2/3)M^3$	$(2/3)M$	M	M^2	$(2/3)M$

7.4.3. Степени параллелизма

Структуры данных и сложность вычислений пакета HPC Challenge свойственны самим алгоритмам независимо от того, как они отображены на параллельную машину. Для вычисления степеней параллелизма необходимо учитывать определенное параллельное размещение. В нашем случае будут использоваться параллельные размещения, рассмотренные в предыдущей главе.

В тесте Stream используется одномерное блочное параллельное размещение для его трех векторов **a, b, c** : $R^{P(N)}$. В этом случае каждое выражение $\mathbf{a}(i) = \mathbf{b}(i)+q\mathbf{c}(i)$ может быть выполнено независимо, приводя к N степеням параллелизма.

В тесте RandomAccess также используется одномерное блочное параллельное размещение своего главного вектора **t** : $Z^{P(N)}$. На первый взгляд, так как RandomAccess случайно обновляет **t**, может показаться, что здесь нет степеней параллелизма, потому что есть вероятность, что два процессора могут попытаться обновить одно и то же значение $\mathbf{t}(i)$. Однако, так как операция XOR является коммутативной, порядок выполнения этих операций не имеет значения. В этом случае каждое выражение $\mathbf{t}(i) = \mathrm{XOR}(t(i),r)$ может выполняться независимо, приводя к N степеням параллелизма.

Параллельный алгоритм FFT обычно требует применения двух одномерных блочных распределений к двум матрицам **X** : $C^{P(M_1)\times M_2}$, **Z** : $C^{M_1\times P(M_2)}$. Степени параллелизма в любой заданной точке алгоритма зависят от того, какая матрица используется. На первом шаге алгоритма, где операции совершаются над **X**, имеется M_1 степеней параллелизма. На втором шаге алгоритма, где операции совершаются над Z, имеется M_2 степеней параллелизма. Обычно, Np не меняется от первого шага до второго, таким образом, общее количество степеней параллелизма составляет $\min(M_1,M_2)$.

Параллельный алгоритм HPL использует двумерное блочно-циклическое распределение на своей главной матрице **A** : $R^{P_{bc}(M)\times P_{bc}(M)}$. Теоретически, если размер блока установить равным 1 элементу, то такую матрицу можно разбить между $M^2 = N$ процессорами, что привело бы к N степеням параллелизма. Однако лежащий в основе программы ал-

горитм имеет определенные зависимости, так что эффективное максимальное количество степеней параллелизма, как правило, составляет $M = \sqrt{N}$.

Количество степеней параллелизма в тестах производительности HPC Challenge варьируется в пределах от $\sqrt{N}$ до N. Так как набор HPC Challenge предназначен для выполнения на больших задачах, где $N \alpha Np$, почти всегда будет выполняться неравенство $\sqrt{N} > Np$. Другими словами, точно так же, как и для множества других задач, здесь имеется достаточно степеней параллелизма, и обнаружить их несложно. Действительную трудность представляет создание алгоритма с достаточной степенью локальности, который будет эффективно работать на заданной параллельной машине.

7.4.4. Сложность коммуникации

Количество перемещаемых данных также зависит от используемого параллельного размещения. В случае с тестом Stream – поскольку все три вектора используют одинаковое параллельное размещение – нет необходимости в перемещении данных ($D = 0$), в результате чего отношение вычисление–коммуникация $W / D \to \infty$. Если по какой-либо причине один из массивов имел бы карту размещения, отличающуюся от других, то было бы необходимо передать весь вектор целиком другим процессорам ($D = N$), и в этом случае $W/D = 2$.

В тесте производительности RandomAccess передача данных сходна с тестом Stream в случае различных размещений: $D = N$. Это может показаться удивительным, так как каждый процессор генерирует индексы для обновления. Однако эти индексы охватывают весь диапазон от 1 до N, поэтому вероятность того, что процессор будет генерировать индексы, которые останутся в его локальной памяти, составляет $1/Np$. В результате $W / D \to 1$ при увеличении Np.

Для работы FFT необходимо перераспределение данных между двумя матрицами, из чего следует $D = M_1M_2 = N$. В результате отношение вычисление–коммуникация равно $W/D = 5\log_2(N)$.

Для работы HPL также требуется перемещение всей матрицы в ходе вычислений, таким образом, $D = M^2 = N$ и $W / D = (2/3)M = (2/3)\sqrt{N}$.

Благодаря составу пакета HPC Challenge, отношение вычисление–коммуникация значительно варьируется между различными его тестами, от $W/D = 1$ для RandomAccess до $W_{element} = 2/3\sqrt{N}$ для HPL. Эти значения являются показателем степени локальности вычислений и принципиально определяют, насколько эффективно заданная система параллельных вычислений может выполнять эти тесты производительности. При наличии традиционной системы параллельных вычислений с традиционной иерархией памяти ожидается, что реализации тестов HPL и Stream будут самыми эффективными, а тестов RandomAccess и FFT – самыми неэффективными. Во многих случаях отношение вычисление–коммуникация W/D совпадает с количеством работы на одном элементе данных $W_{element}$, кроме случаев, когда вычисление является массово-параллельным, пример – тест Stream.

7.5. Производительность оборудования

Характеристики алгоритма, такие как $W_{element}$ и W/D, дают общее представление об ожидаемой параллельной производительности приложения. В общем случае ал-

горитм с высокими значениями $W_{element}$ и W/D будет работать лучше, чем алгоритм с низкими значениями.

Чтобы отойти от этих обобщенных утверждений, необходимо рассмотреть шаблоны доступа к памяти и вычисления более подробно при помощи исследования пространственной и временной локальности этих алгоритмов.

7.5.1. Пространственная и временная локальности

Как уже было упомянуто, пространственная локальность является мерой того, насколько близко друг к другу находятся шаблоны доступа к памяти для данного вычисления. Временная локальность является мерой количества операций, выполняемых над каждым элементом данных, к которому осуществляется доступ. Измеренные показатели пространственной и временной локальности для набора тестов производительности HPC Challenge приведены на рисунке 7.3.

Тест Stream – алгоритм–прототип высокой пространственной локальности. Он обходит каждый массив в строго последовательном порядке, так что каждая операция доступа к памяти обращается к элементу, расположенному сразу за использовавшимся предыдущей операцией. Невозможно представить программу с более высоким уровнем пространственной локальности. Тест Stream также является алгоритмом–прототипом низкой временной локальности, так как он выполняет только две операции на каждые два считываемых значения. Точно так же трудно представить себе программу с более низким уровнем временной локальности. Таким образом, тест Stream определяет нижний правый угол на графике пространственного/временного показателя.

Тест производительности RandomAccess является алгоритмом–прототипом низкой пространственной локальности. Он в случайном порядке считывает данные из памяти, таким образом, каждая операция обращения к памяти никак не связана с предыдущей. Тест RandomAccess, так же как и Stream, имеет низкий уровень временной локальности. Тест RandomAccess определяет нижний левый угол на графике пространственного/временного показателя.

Пространственная локальность тестов FFT и HPL располагается между тестами Stream и RandomAccess. Временная локальность HPL находится на очень высоком уровне, так как он совершает $(2/3)\sqrt{N}$ операций на каждый элемент данных – это означает, что на каждую операцию доступа к памяти приходится очень много операций. Временная локальность у FFT выше, чем у теста Stream или RandomAccess, но ниже, чем у HPL, так как он выполняет $5\log2(N)$ операций на каждый элемент данных.

7.5.2. Эффективность производительности

Знание количества операций на каждый элемент данных, отношения вычисление–коммуникация и пространственной и временной локальности алгоритма дает нам большое количество информации о приложениях, которую можно использовать для общей оценки производительности. Для количественной оценки производительности необходимы знания особенностей конкретной параллельной вычислительной

архитектуры. Для этого предположим наличие стандартной параллельной вычислительной машины (см. рисунок 7.2, в которой имеется кэш-память, локальная и удаленная память). В такой стандартной машине каждый уровень иерархии памяти характеризуется задержкой и обратной пропускной способностью по отношению к данному уровню иерархии памяти. Более того, тактовые циклы процессора используются в качестве единиц времени. Таким образом, задержка измеряется числом тактовых циклов процессора, которые необходимы для извлечения одного элемента данных с этого уровня иерархии. Таким же образом, обратная пропускная способность измеряется в тактовых циклах процессора на элемент данных, когда большой объем данных извлекается с данного уровня иерархии памяти. Используя эти единицы, можно абстрактно охарактеризовать параллельную машину, как показано в таблице 7.2. Теперь можно оценить производительность набора HPC Challenge.

Таблица 7.2. Параметры иерархии памяти

Параметры гипотетической иерархии памяти для стандартной параллельной машины

Память Уровень	L задержка (циклы)	пропускная пособность (циклы/элементы)
Процессор – локальная кэш-память (ca)	10^1	10^1
Процессор – локальная память (lm)	10^3	10^2
Процессор – уделенная память (rm)	10^5	10^3
Каждый к каждому (aa)	–	10^3

7.5.3. Оценка производительности

Первый шаг при оценке производительности параллельного приложения заключается в расчете производительности на одном процессоре. Производительность всегда измеряется относительно пиковой производительности на одном процессоре. Исходное значение, равное 1, будем рассматривать в качестве пиковой производительности нашей стандартной машины. Тогда пиковая производительность параллельного процессора равна Np.

Производительность одного процессора можно рассчитать, взяв отношение времени, затрачиваемого на выполнение операций на процессоре без иерархии памяти, ко времени, затрачиваемому на выполнение операций на процессоре с иерархией памяти. В случае Stream она равна

$$\text{Производительность}(Np=1)=\frac{W}{W+2B_{lm}^{-1}N}$$
$$=\frac{1}{1+2B_{lm}^{-1}W_{element}^{-1}}$$
$$\approx 1/B_{lm}^{-1}=10^{-2},$$

где в знаменателе учитывается время, затрачиваемое на перемещение данных из памяти и в память. В данном случае ясно, что определяющим фактором произво-

производительность теста Stream рассчитать легко, так как он является массово-параллельным:

$$\text{Производительность}\,(Np > 1) \approx Np / B_{lm}^{-1}.$$

Таким образом, хотя Stream характеризуется довольно низкой последовательной производительностью, она масштабируется линейно с увеличением Np.

Последовательная производительность RandomAccess может быть рассчитана таким же способом:

$$\text{Производительность}\,(Np = 1) = \frac{1}{1 + L_{lm} W_{element}^{-1}}.$$
$$\approx 1 / L_{lm} = 10^{-3}$$

В данном случае ясно, что задержка локальной памяти является определяющим фактором производительности. Параллельную производительность теста RandomAccess рассчитать более сложно, так как по правилам теста каждый процессор может генерировать наборы индексов для обновления группами по 1024 единицы. Таким образом, количество сообщений, отправленных процессором, будет равно наименьшему значению из 1024 и Np. Прибавив дополнительное время на отправку сообщений, получаем параллельную производительность RandomAccess, равную

$$\text{Производительность}\,(Np > 1) \frac{Np}{1 + L_{lm} W_{element}^{-1} + L_{rm} \min(1024, Np)(W / D)^{-1}}$$
$$= \frac{Np}{1 + L_{lm} + L_{rm} \min(1024, Np)}$$
$$\approx 1 / L_{rm} = 10^{-5}\ for Np < 1024$$

В этом случае очевидно, что задержка удаленной памяти является определяющим фактором производительности.

Последовательная производительность теста FFT задается так:

$$\text{Производительность}\,(Np = 1) = \frac{1}{1 + 6 B_{lm}^{-1} W_{element}^{-1}},$$
$$\approx 5 \log_2(N) / 6 B_{lm}^{-1}$$
$$\approx 0.2\ \ for\ N = 2^{24}$$

где в знаменателе учитывается факт, что в последовательном БПФ данные считываются в и из локальной памяти три раза. В отличие от тестов Stream и RandomAccess, последовательная производительность явно зависит от размера задачи N, так как при увеличении N увеличивается $W_{element}$ в пропорции $\log_2(N)$. Параллельная производительность теста FFT задается так

$$\text{Производительность}\,(Np > 1) = \frac{Np}{1 + 6 B_{lm}^{-1} W_{element}^{-1} + 3 B_{rm}^{-1} (W / D)^{-1}},$$
$$\approx 5 Np \log_2(N) / 3 B_{rm}^{1}$$
$$\approx 0.02 Np[1 + 2 \log_2(Np)]\ \ for N / Np = 2^{24}$$

где N пропорционально Np. Из вышеуказанной формулы можно предположить, что производительность БПФ масштабируется суперлинейно с ростом Np. Однако в этой формуле есть неявное допущение, что коммуникационная сеть между удаленными запоминающими устройствами может работать с числом Np процессоров одновременно, взаимодействуя без какого-либо понижения производительности. Это допущение не соответствует действительности. Более реальное допущение состоит в том, что B_{rm} фиксировано или растет логарифмически с ростом Np. Принимая, что B_{rm} фиксировано, получаем

$$\begin{aligned}\text{Производительность } (Np>1) &= \frac{Np}{1+6B_{lm}^{-1}W_{element}^{-1}+3NpB_{rm}^{-1}(W/D)^{-1}},\\ &\approx 5\log_2(N)/6B_{rm}^{-1}\\ &\approx 0.02[1+2\log_2(Np)] \;\; for \;\; N/Np=2^{24}\end{aligned}$$

что является более реалистичной моделью параллельной производительности теста FFT.

Последовательная производительность HPL составляет

$$\begin{aligned}\text{Производительность } (Np=1) &= \frac{1}{1+2B_{lm}^{-1}W_{element}^{-1}}.\\ &= \frac{1}{1+(4/3)B_{lm}^{-1}/\sqrt{N}}\\ &\approx 1\end{aligned}$$

То есть количество операций на элемент данных достаточно велико, чтобы временем, затрачиваемым на перемещение данных в процессор и из процессора, можно было пренебречь. Похожая ситуация и в случае параллельной реализации:

$$\begin{aligned}\text{Производительность } (Np>1) &= \frac{Np}{1+2B_{lm}^{-1}W_{element}^{-1}+NpB_{rm}^{-1}(W/D)^{-1}}.\\ &= \frac{Np}{1+(4/3)B_{lm}^{-1}/\sqrt{N}+(2/3)B_{rm}^{-1}/\sqrt{N}}\\ &\approx Np\end{aligned}$$

Даже при фиксированной пропускной способности сети

$$\begin{aligned}\text{Производительность } (Np>1) &= \frac{Np}{1+2B_{lm}^{-1}W_{element}^{-1}+NpB_{rm}^{-1}(W/D)^{-1}}\\ &= \frac{Np}{1+(4/3)B_{lm}^{-1}/\sqrt{N+(2/3)NpB_{rm}^{-1}/\sqrt{N}}}\\ &\approx Np\end{aligned}$$

7.5.4. Результаты анализа

Вышеуказанные оценки производительности можно сравнить с многочисленными результатами набора тестов HPC Challenge. Использование системы с характеристиками производительности, близкими к указанным в таблице 7.2, дает эмпирические результаты, отображенные на рисунке 7.6. Эти результаты показывают, что рас-

четы измеренной и предполагаемой производительности согласуются между собой. Такой уровень согласования между теорией и практикой является убедительным доказательством того, что параллельная программа реализована правильно, и достигается максимальная производительность для данной параллельной машины.

7.5.5. Значение производительности

Оценки производительности налагают фундаментальные ограничения параллельных алгоритмов на конкретную систему аппаратного обеспечения. Качество реализации программного обеспечения измеряется относительно этих пределов производительности. Очень полезно знать данные пределы.

Во-первых, оценки производительности необходимы для того, чтобы определить, имеется ли потенциальная выгода от использования параллельной машины, до того, как разработчик начинает проходить через муки написания ПО. Тесты Stream и HPL существенно выигрывают от реализации параллелизма. FFT получает среднюю выгоду. RandomAccess не имеет выгоды в плане производительности при использовании стандартной параллельной вычислительной машины.

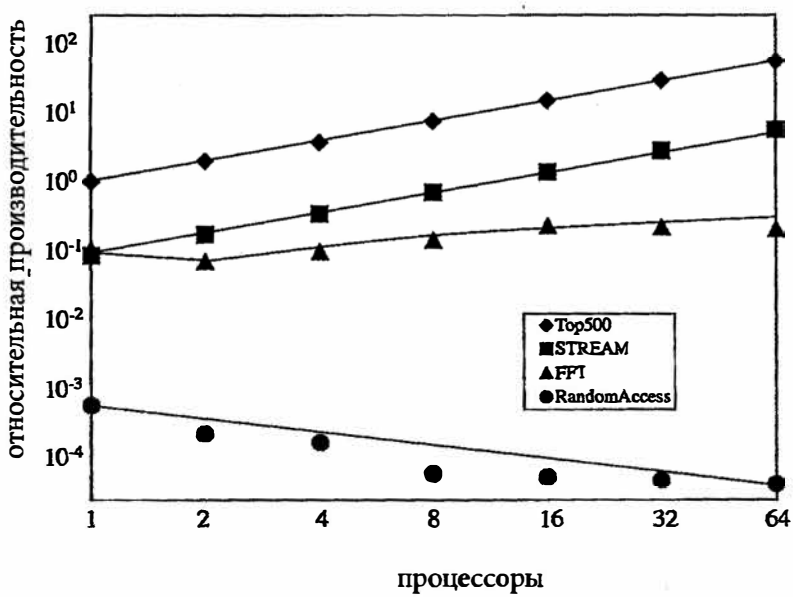


Рисунок 7.6. Оценки производительности HPC Challenge

Производительность HPC Challenge в зависимости от количества процессоров. Точки означают измеренные величины. Сплошные линии означают производительность, предсказанную на основе оценок

Во-вторых, оценки производительности могут быть полезны для выявления части реализации программного обеспечения, которая оказывает наибольшее влияние на производительность. Для теста Stream наиболее важным является перемещение крупных блоков данных в локальную память и из нее при высокой пропускной способно-

сти. Для FFT – передача крупных блоков данных между удаленными устройствами памяти при высокой пропускной способности. Для RandomAccess – передача блоков данных малых размеров с низкой задержкой.

В-третьих, с помощью расчета производительности можно оценить, насколько эффективно выполняется реализация. В частности, если при реализации удается достичь только небольшой части оценочной производительности, велика вероятность, что есть смысл в дальнейшей оптимизации. Напротив, если при реализации достигается большая часть производительности, дальнейшая оптимизация нецелесообразна.

7.6. Производительность программ

Анализ производительности в предыдущих разделах устанавливает верхнюю границу достижимой производительности для определенного алгоритма на определенной архитектуре параллельных процессоров. Целью реализации программного обеспечения является нахождение разумного соотношения между достижением определенного уровня производительности и необходимыми трудозатратами. При параллельной реализации с помощью MATLAB необходимо выбрать из следующих параметров:

Тип параллелизма: параллелизм по данным, параллелизм по задачам, конвейер и/или карусельная диспетчеризация.

Модель параллельного программирования: обмен сообщениями, «начальник/подчиненный» и распределенные массивы.

Среда программирования: языки и/или библиотеки.

Выбор каждого из параметров особым образом влияет на эффективность программного обеспечения (обработку, пропускную способность и память) и его затраты (размер и переносимость кода).

7.6.1. Stream

Анализ производительности теста Stream показывает, что параллельная реализация должна работать эффективно. Основным узким местом производительности Stream является пропускная способность локальной памяти каждого процессора. Таким образом, при реализации этого теста производительности необходимо особо аккуратно использовать пропускную способность локальной памяти. В частности, при реализации следует избегать создания дополнительных временных копий массивов, которые могут еще больше загрузить локальную память.

Stream является достаточно простым тестом производительности, который можно реализовать с использованием различных типов параллелизма. Однако наиболее естественным здесь будет параллелизм по данным, при котором каждый процессор работает над своей локальной частью большого распределенного массива. Основное преимущество этого подхода заключается в том, что параллельная версия может быть реализована с внесением сравнительно небольшого количества изменений в последовательный код. Параллелизм по задачам, конвейерный параллелизм и карусельная диспетчеризация в тесте Stream присутствуют, но их применение приведет к реализации с низким уровнем локальности и, соответственно, производительности. Например, конвейерная реализация теста Stream может выполнять операцию умножения на одном наборе процессоров, а операцию сложения на другом наборе, но при этом тре-

буется передавать массив целиком. На стандартном параллельном процессоре это будет неэффективно.

При условии, что параллелизм по данным является оптимальной параллельной моделью, модель распределенных массивов с использованием библиотеки распределенных массивов закономерно является программной технологией, естественной для использования в этой реализации. Регулирование пропускной способности памяти при использовании этой технологии требует некоторой осторожности. Во-первых, использование явного цикла `for` для выполнения вычислений обычно влечет за собой некоторые накладные расходы в реализации на MATLAB. Во-вторых, использование чистого «векторизованного» кода распределенных массивов может повлечь за собой некоторые накладные расходы от библиотеки распределенных массивов, так как она пытается определить, какие индексы локальны для каких процессоров. Таким образом, оптимальная реализация может быть эффективной с использованием явного локального «векторизованного» подхода:

$$\mathbf{a}.loc = \mathbf{b}.loc + q\mathbf{c}.loc.$$

При использовании этих методов реализация теста Stream на основе распределенных массивов должна быть способна достичь уровня производительности, близкого к тому, какой может предоставить параллельных процессор. Кроме того, параллельный код на MATLAB должен включать относительно небольшое количество изменений по сравнению с последовательным кодом на MATLAB.

7.6.2. RandomAccess

Другой крайностью является тест RandomAccess. Анализ производительности показывает, что параллельная реализация RandomAccess на стандартной параллельной машине имеет низкую производительность. Обычно это означает, что нецелесообразно даже пытаться выполнять параллельную реализацию программы. Однако при некоторых обстоятельствах параллельная программа может быть все же желательна, например, если размер задачи больше, чем может поместиться в памяти одной машины. В данных условиях важно измерить производительность реализации относительно того, что способна предоставить параллельная машина. При очень эффективной реализации теста RandomAccess может не наблюдаться параллельного ускорения.

Основным узким местом производительности RandomAccess является способность отправлять сообщения малых размеров очень быстро большому количеству процессоров. Такая особенность производительности фундаментальным образом определяется задержкой нижележащей параллельной сети. И программная реализация должна уделять особое внимание тому, чтобы не делать ничего, что способно увеличить эту задержку.

RandomAccess главным образом проявляет параллелизм по данным, и его основная структура данных естественным образом отображается на распределенные массивы как способ достижения параллелизма. К сожалению, шаблон коммуникации при использовании чистых распределенных массивов выполняется неэффективно. В частности, если операция

$$\mathbf{t}(i) = \mathrm{XOR}(\mathbf{t}(i), r).$$

выполняется непосредственно на распределенных массивах, это, скорее всего, повлечет за собой дополнительные накладные расходы, поскольку каждый процессор будет пытаться определить, какой процессор владеет каждым индексом *i*. Кроме того, библиотека распределенных массивов не может знать, что операция XOR является коммутативной, и может попытаться «навязать» правильный порядок обновлений. Все это будет иметь эффект роста задержки для операции. Таким образом, для высокопроизводительной реализации следует использовать комбинированный метод – распределенные массивы и обмен сообщениями. Каждый процессор генерирует набор индексов *i*, явно определяет, какие процессоры каким индексом владеют, и затем отправляет индексы этим процессорам для выполнения обновления.

При использовании данных методик комбинированная реализация на основе распределенных массивов и обмена сообщениями теста RandomAccess должна быть способна достигнуть уровня производительности, близкого к тому, который может предоставить параллельный процессор. К сожалению, на большинстве параллельный процессоров эта производительность не будет высокой. Кроме того, так как параллельный код на MATLAB требует использовать обмен сообщениями, этот код будет содержать большое количество изменений по сравнению с последовательным кодом на MATLAB.

7.6.3. FFT

Анализ производительности теста FFT показывает, что на параллельном процессоре можно достичь среднего ускорения. Это приложение попадает в серую зону между отметками «чистая победа» и «оно того не стоит». Применение параллельной реализации, так же как в случае RandomAccess, по большей части может быть вызвано необходимостью выполнять задачи большого размера, которые не помещаются в памяти одного процессора. В этом случае также важно сопоставлять производительность реализации с той, которой можно достичь на параллельном процессоре. Узким местом производительности теста FFT является шаблон коммуникации «каждый к каждому» (all-to-all) , требующийся при перераспределении данных на шаге:

$$\mathbf{Z} = \mathbf{X},$$

где $\mathbf{X} : C^{P(M_1)\times M_2}$ и $\mathbf{Z} : C^{M_1\times P(M_2)}$. Производительность на этом этапе фундаментальным образом определяется пропускной способностью коммуникационных сетей. Параллельная реализация должна заботиться о том, чтобы не ухудшить эту пропускную способность.

FFT проявляет как параллелизм по данным, так и конвейерный параллелизм, которые естественным образом могут быть выражены при помощи распределенных массивов. Перераспределение, упомянутое выше, является классическим примером преимущества кодирования при помощи распределенных массивов. Запись эквивалентных выражений при помощи обмена сообщениями приводит к значительному увеличению кода. Однако перераспределение общего вида, которое должно работать для любой возможной карты размещения, повлечет за собой определенные накладные расходы, которые отсутствуют при использовании метода обмена сообщениями. Кроме того, реализация обмена сообщениями может более аккуратно управлять буферами данных и порядком рассылки сообщений по сети – так, чтобы избежать пере-

насыщения сообщениями определенного процессора. Таким образом, обычно для перераспределения данных, подобного этому, реализуется специальная функция в виде частного случая, которая может использовать преимущества оптимизации, специфичной для данного размещения.

При применении этого метода реализация теста FFT на основе распределенных массивов должна достигнуть уровня производительности, близкого к тому, какой может предоставить параллельный процессор (т. е. средняя величина параллельного ускорения). Кроме того, параллельный код на MATLAB должен содержать относительно небольшое количество изменений по сравнению с последовательным кодом.

7.6.4. HPL

Анализ производительности теста HPL показывает, что параллельная реализация должна работать эффективно. Основным узким местом HPL является скорость, с которой процессор может выполнять операции с плавающей запятой, присущие множеству операций перемножения матриц. Таким образом, реализация этого теста производительности должна обеспечить сохранность этих операций, не разбивая их на множество более мелких операций перемножения матриц.

Тест HPL главным образом проявляет параллелизм по данным, и его основная структура данных естественным образом отображается на распределенные массивы как способ достижения параллелизма. Если используется просто параллельное распределение, такое как $A : C^{P(M)\times M}$, то результирующие операции коммуникации становятся простыми для написания с использованием распределенных массивов. К сожалению, такое распределение приведет к неравномерному распределению работы в ходе выполнения вычислений. Метод, обеспечивающий более равномерное распределение нагрузки, использует двумерное блочно-циклическое распределение $A : C^{P_{bc}(M)\times P_{bc}(M)}$. Шаблоны коммуникации, вытекающие из такого распределения, довольно сложные. Ими непросто управлять при использовании метода чистых распределенных массивов. Обычно для достижения оптимальной производительности используется комбинированный метод на основе распределенных массивов с обменом сообщениями.

При использовании этих методик реализация теста производительности HPL на основе распределенных массивов должна достигнуть уровня производительности, близкого к тому, какой может предоставить параллельный процессор. Однако параллельный код на MATLAB будет сильно отличаться от последовательного.

7.7. Производительность по отношению к трудозатратам

В предыдущем разделе был проведен анализ производительности набора тестов HPC Challenge и сделана серия предположений о производительности и трудозатратах, связанных с реализацией этих тестов. На рисунке 7.6 ранее было показано, что производительность тестовых версий, написанных на языке C с использованием обмена сообщениями, согласуется с моделью производительности. Что касается версий параллельных программ на MATLAB, написанных с использованием распределенных массивов, – более подробно вопрос их реализации будет рассматриваться в последующих главах.

В этом отношении нас интересуют два сравнения. Первое – производительность параллельных программ на MATLAB по отношению к последовательным, так как это

один из важнейших вопросов, интересующих разработчиков. Второе – производительность параллельных MATLAB-программ по отношению к C+MPI как способ измерения качества реализации распределенных массивов и руководство к будущему повышению производительности.

Были реализованы четыре основных теста производительности с использованием параллельного программирования на MATLAB; они были запущены на серийной кластерной системе (точное описание оборудования приведено в конце главы). Обе параллельные реализации тестов производительности – на MATLAB и C+MPI, взятых в качестве образца, – были запущены на 1, 2, 4, 8, 16, 32, 64 и 128 процессорах. Для каждого варианта количества используемых процессоров была выполнена наибольшая по размеру задача, помещающаяся в имеющуюся основную память. По собранным данным можно определить относительную производительность вычислений и накладные расходы памяти параллельных программ на MATLAB по сравнению с C+MPI. Относительные значения памяти, производительности и размеров кодов обобщены в таблице 7.3.

Таблица 7.3. Результаты реализации пакета HPC Challenge

(Верхняя часть) Максимальный размер задачи по отношению к однопроцессорному варианту C+MPI на 128 процессорах. (Средняя часть) Производительность тестов по отношению к однопроцессорному варианту C+MPI на 128 процессорах. (Нижняя часть) Размер кода измерен в строках кода (SLOC). Размер параллельного исходного кода C+MPI набора HPC Challenge, взятого в качестве образца, взят из раздела HPC Challenge FAQ. Размер кода FFT включает в себя код, используемый для создания случайных волн, и не включает код начальных и конечных операций «каждый к каждому». Эти операции приблизительно уравновешивают друг друга

Реализация	Stream	FFT	RandomAccess	HPL(32)
Максимальный размер задачи				
C+MPI/C serial	63.9	72.7	48	32.6
pMatlab/C serial	42.8	21.3	32	9.3
C+MPI/pMatlab	1.5	3.4	1.5	3.5
Относительная производительность				
C+MPI/C serial	62.4	4.6	0.074	28.2
pMatlab/C serial	63.4	4.3	0.0016	6.8
C+MPI/pMatlab	1	1	46	4
Размер кода				
C+MPI	347	787	938	8800
pMatlab	119	78	157	190
C+MPI/pMatlab	3	10	6	40

Как правило, реализации параллельных MATLAB-программ могут выполнять задачи, размер которых составляет 1/2 размера задач, способных решаться реализациями на C+MPI. Причина этого в необходимости создавать временные массивы при использовании выражений высокого уровня. Производительность параллель-

ных MATLAB-программ изменяется, начиная от показателей, сравнимых с кодом на C+MPI (FFT и Stream), и заканчивая более медленной (Top 500) и еще более медленной (RandomAccess). Напротив, код параллельных программ на MATLAB обычно в 3–40 раз меньше эквивалентного кода на C+MPI.

По сравнению с последовательными кодами на MATLAB все параллельные коды допускают выполнение задач, масштабируемых линейно с количеством процессоров. Таким же образом они показывают значительное увеличение производительности (кроме RandomAccess). Производительность RandomAccess ограничивается нижележащим механизмом коммуникации. По сравнению с C+MPI размеры задач для параллельных MATLAB-программ меньше в 2 раза, а производительность параллельных MATLAB–версий как Stream, так и FFT является сопоставимой.

В тесте производительности Stream максимальный размер задачи для параллельного кода на MATLAB в 1,5 раза меньше, чем максимальный размер задачи в случае кода на C+MPI из-за необходимости создавать промежуточные временные массивы. Необходимость во временных массивах является побочным эффектом большинства сред программирования высокого уровня. Производительность параллельного кода на MATLAB такая же, как и в случае C+MPI. Причина заключается в том, что интерпретатор MATLAB распознает оператор масштабирования и прибавления и заменяет его на вызов из соответствующего оптимизированного BLAS (базовый пакет подпрограмм линейной алгебры). Параллельный код на MATLAB в 3 раза меньше, чем код на C+MPI из-за исключения различных циклов `for` и использования встроенных функций MATLAB.

В тесте производительности RandomAccess максимальный размер задачи для параллельного кода на MATLAB в 1,5 раза меньше, чем максимальный размер задачи для кода на C+MPI из-за необходимости создавать промежуточные временные массивы. На одном процессоре производительность параллельного RandomAccess на MATLAB сравнима с производительность кода на C+MPI. Однако на большем количестве процессоров параллельный код на MATLAB в 45 раз медленнее, чем код на C+MPI. Эта разница производительности наблюдается по причине большой задержки нижележащего механизма коммуникации, который можно потенциально улучшить при использовании механизма коммуникации с меньшей задержкой. Параллельный код на MATLAB в 6 раз меньше, чем код на C+MPI.

В тесте производительности FFT максимальный размер задачи для параллельного кода на MATLAB в 3,5 раза меньше, чем максимальный размер задачи для кода на C+MPI, из-за необходимости создавать промежуточные временные массивы. Кроме того, внутри MATLAB используется раздельное представление для комплексных типов данных, в то время как вызываемая нижележащая последовательная библиотека БПФ использует чередующееся представление. В результате данные необходимо преобразовывать между этими представлениями. На одном процессоре производительность FFT на MATLAB в 5 раз ниже, чем производительность кода на C из-за накладных расходов времени, необходимых для выполнения преобразования между форматами хранения комплексных данных. По мере роста размера задачи время FFT начинает все больше зависеть от времени осуществления коммуникации «каждый к каждому», необходимой между этапами вычисления. Так как это в основном сообщения большого размера, производительность параллельного кода на MATLAB становится равной производительности кода на C+MPI при большом количестве процессоров.

Параллельный код на MATLAB в 10 раз меньше, чем код на C+MPI из-за локальных вызовов встроенных функций БПФ и исключения кода обмена сообщениями MPI.

В тесте производительности HPL максимальный размер задачи для параллельного кода на MATLAB в 3,5 раза меньше, чем максимальный размер задачи в случае кода на C+MPI из-за необходимости создавать промежуточные временные массивы. В частности, нижне- и верхне-треугольные матрицы возвращаются как полные матрицы; напротив, в коде на C+MPI эти матрицы могут быть объединены в один массив. Параллельный код на MATLAB обеспечивает 10-кратное ускорение на 32 процессорах, что приблизительно в 4 раза меньше, чем ускорение с использованием кода на C+MPI. Улучшение сети этого оборудования должно привести к значительному улучшению производительности параллельного кода на MATLAB относительно кода на C+MPI. Параллельный код на MATLAB в 40 раз меньше, чем код на C+MPI. 10-кратное сокращение данного кода наблюдается благодаря абстракциям более высокого уровня из параллельного кода на MATLAB, а дополнительное 4-кратное сокращение происходит из-за использования более простого алгоритма.

Один подход к изображению производительности тестов HPC Challenge показан на рисунке 7.7. Ускорение и относительный размер кода для каждой реализации программы были рассчитаны по отношению к последовательной реализации на C/Fortran. На данном графике можно увидеть, что, за исключением RandomAccess, все программы на C+MPI попадают в верхний правый квадрант диаграммы, показывая, что они обеспечивают некоторый уровень параллельного ускорения, требуя создания кода большего размера, нежели последовательный код. Как и ожидалось, реализации последовательных кодов на MATLAB не дают никакого ускорения, однако все они требуют написания кода меньшего размера, чем реализации в последовательном коде на C/Fortran. Параллельные реализации на MATLAB (кроме RandomAccess) попадают в верхний левый квадрант диаграммы, обеспечивая параллельное ускорение при меньшем количестве строк кода.

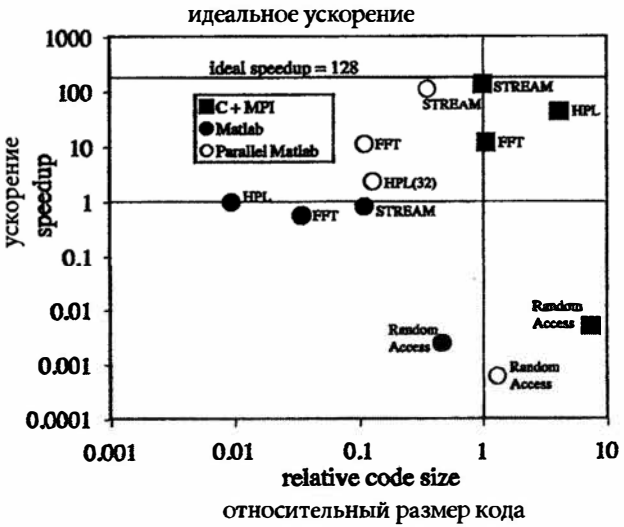


Рисунок 7.7. Ускорение относительно размера кода

Ускорение (относительно последовательного C) по отношению к размеру кода (относительно последовательного C). Верхний правый квадрант – это традиционный режим HPC: для увеличения производительности требуется увеличение размера кода, и большинство кодов на C+MPI попадают сюда. Нижний левый квадрант – стандартный режим последовательных высокоуровневых языков, которые обеспечивают код много меньшего размера, но при этом работают гораздо медленнее. RandomAccess находится в нижнем правом квадранте и представляет алгоритмы, которые просто не соответствуют нижележащему оборудованию и механизму коммуникации. В верхнем левом квадранте расположено большинство параллельных реализаций на MATLAB, имеющих коды меньших размеров и обеспечивающих некоторое ускорение

Такой компромисс между производительностью и трудозатратами подытоживается при изучении относительной продуктивности сред (см. главу 6). На основе данных об ускорении и относительном размере кода на рисунке 7.8 показана продуктивность каждого подхода на каждом тесте производительности.

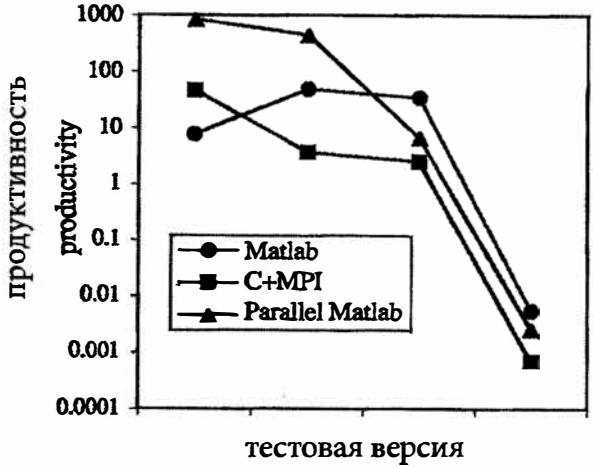


Рисунок 7.8. Продуктивность HPC Challenge

Относительная продуктивность каждого теста производительности в каждой из сред программирования

ıсок литературы

1. *[Bliss & Kepner 2007] N. Bliss and J. Kepner, pMATLAB PARALLEL MATLAB library, Special issue on high productivity programming languages and models, International Journal of High Performance Computing Applications, Vol. 21, No. 3, pp. 336–359, 2007.*

2. *[Cummings et al. 1998] J.C. Cummings, J.A. Crotinger, S.W. Haney, W.F. Humphrey, S.R. Karmesin, J.V. Reynders, S.A. Smith, and T.J. Williams, RaPid application development and enhanced code interoperability using the POOMA framework, Proceedings of the SIAM Workshop on Object-Oriented Methods and Code Interoperability in Scientific and Engineering Computing (0098), 1998.*

3. *[Dongarra and Luszczek 2005] Jack Dongarra and Piotr Luszczek, Introduction to the HPC Challenge Benchmark Suite, Technical Report UT-CS-05-544, University of Tennessee, 2005.*

4. *[Weinberg et al. 2005] J. Weinberg, M.O. McCracken, E. Strohmaier, and A. Snavely, Quantifying locality in the memory access patterns of HPC applications, Conference on High Performance Networking and Computing, Proceedings of the ACMjIEEE Conference on Supercomputing, 2005.*

5. *[El-Ghazawi et al. 2005] Tariq El-Ghazawi, William Carlson, Thomas Sterling, and Kathy Yelick, UPC: Distributed Shared Memory Programming, Hoboken, NJ: John Wiley & Sons, 2005.*

6. *[Flynn 1972] Michael Flynn, Some computer organizations and their effectiveness, IEEE Transactions on Computers, Vol. C–21, No. 9, p. 948, 1972.*

7. *[Nieplocha et al. 2002] J. Nieplocha, R. Harrison, M. Krishnan, B. Palmer, and V. Tipparaju, Combining shared and distributed memory models: Evolution and recent advancements of the Global Array Toolkit, Proceedings of the POHLL–02 Workshop of the International Conference on Supercomputing, New York, 2002.*

8. *[Reuther et al. 2004] Albert Reuther, Tim Currie, Jeremy Kepner, Hahn Kim, Andrew McCabe, Michael Moore, and Nadya Travinin, LLGrid: Enabling oN-demand grid computing with gridMATLAB and pMATLAB, Proceedings of the Eighth Annual High Performance Embedded Computing Workshop (HPEC2004), Lexington, MA, 2004.*

Глава 8
Программа Stream

Резюме

Тест производительности Stream представляет собой простую программу, которая будет нам очень полезна в качестве иллюстрации нескольких важных принципов параллельного программирования. Программа Stream – представитель широкого класса массово параллельных программ. Она демонстрирует важность локальности данных, а также показывает, каким образом извлечь параллелизм из локальности данных. Программа Stream также демонстрирует производительность разных методов доступа с параллелизмом по данным. Наконец, изучение программы Stream дает представление о компромиссах производительности многоядерных процессоров.

Программа Stream была представлена в главе 7 в качестве теста производительности для измерения пропускной способности между процессором и его основной памятью [MCCalpin 2005]. Программа Stream также является хорошим примером из множества массово-параллельных приложений, и на ней можно продемонстрировать некоторые полезные методы написания кода параллельной программы, в частности способ написания программы с параллелизмом по данным, которая извлекает параллелизм из распределенных массивов.

Программа Stream создает несколько больших N-элементных векторов и многократно совершает базовые операции над этими векторами. Так как количество операций, выполняемых над каждым элементом вектора, невелико, производительность Stream ограничена пропускной способностью к основной памяти. Программа Stream является прекрасной иллюстрацией компромиссов производительности многоядерных процессоров, которые имеют общий доступ к памяти.

8.1. Начало работы

Для выполнения программы запустите MATLAB и перейдите в директорию Examples/Sream (т.е. наберите `cd Examples/Sream`). Отредактируйте файл `pStream.m` и установите `PARALLEL=0`. В командной строке введите

```
pStream
```

Так как Stream является тестом производительности, она выдаст большой объем данных о производительности, аналогичных следующим:

Время распределения/назначения (с)	Allocation Time (sec)	= 1.4523
Время запуска (с)	Launch Time (sec)	= 0.094735
A,B,C ошибки	A, B, C errors	= 0 1.819e-12 3.638e-12
Np	Np	= 1
Pid	Pid	= 0
Размер глобального массива (элем.)	Global Array size (elem)	= 33554432
Размер глобального массива (байты)	Global Array size (bytes)	= 268435456
Необходимая глобальная память (байты)	Global memory required (bytes)	= 805306368
Размер локального массива (элем.)	Local Array size (elem)	= 33554432
Размер локального массива (байты)	Local Array size (bytes)	= 268435456
Необходимая локальная память (байты)	Local memory required (bytes)	= 805306368
Количество пробных выполнений	Number of trials	= 4
Локальная пропускная способность Copy (Гб/с)	Local Copy Bandwidth (MB/sec)	= 2338.2472
Локальная пропускная способность Scale (Гб/с)	Local Scale Bandwidth (MB/sec)	= 710.2395
Локальная пропускная способность Add (Гб/с)	Local Add Bandwidth (MB/sec)	= 994.2772
Локальная пропускная способность Triadd (Гб/с)	Local Triadd Bandwidth (MB/sec)	= 798.4208
Глобальная пропускная способность Copy (Гб/с)	Global Copy Bandwidth (MB/sec)	= 2338.2472
Глобальная пропускная способность Scale (Гб/с)	Global Scale Bandwidth (MB/sec)	= 710.2395
Глобальная пропускная способность Add (Гб/с)	Global Add Bandwidth (MB/sec)	= 994. 2772
Глобальная пропускная способность Triadd (Гб/с)	Global Triadd Bandwidth (MB/sec)	= 798.4208

Вышеуказанную информацию можно разбить на четыре категории. Первая – параметры программы, которые описывают размер задачи N, количество необходимых байтов, количество пробных запусков, количество процессоров Np и P_{ID} процессора, выполнившего вывод. Вторая категория включает в себя время инициализации массива и время запусков, которые описывают количество времени, затрачиваемое до начала выполнения программы. Третья категория – это значения локальной и глобальной пропускной способности. Четвертая категория – значения проверки правильности. Эти результаты будут рассмотрены далее в разделе «Тестирование». Сейчас отредактируйте файл `pStream.m`, установите `PARALLEL=1` и выполните Stream повторно с использованием следующих команд:

```
eval(pRun('pStream',1,{}))
eval(pRun('pStream',2,{}))
eval(pRun('pStream',4,{}))
```

Эти команды выполняют программу Stream с использованием одного, двух и четырех процессоров. При выполнении на параллельной вычислительной машине выдаются значения временных параметров, подобные указанным в таблице 8.1. Эти результаты позже будут рассмотрены подробнее в разделе «Тестирование».

Важное примечание. Последний аргумент команды `pRUN` определяет, на какой параллельной вычислительной машине будет выполняться программа. Может возникнуть необходимость в получении надлежащих установочных параметров у вашего системного администратора; для получения более подробной информации обращайтесь на веб-сайт книги (http://www.siam.org/KepnerBook). Установка последнего аргумента значением {} приводит к выполнению множества копий программы на вашей локальной машине и получению производительности, отличной от указанной в таблице 8.1.

8.2. Параллельная реализация

Спецификация теста производительности Stream предполагает наличие трех N-элементных векторов **a**, **b**, **c** : R^N таких, что общая занятая память составляет значительную частью памяти процессора. Заданные последовательности операций выполняются на этих векторах в следующем порядке:

```
Copy: c = a Scale: b =
  q c Add: c = a+ b
Triadd : a = b + q c
```

Перевод
Copy – копирование;
Scale – умножение;
Add – сложение;
Triadd – умножение с последующим сложением;

Вышеуказанные операции повторяются N раз. Исходный алгоритм для Stream показан в Алгоритме 8.1 (подробнее см. в подписи).

Алгоритм 8.1. Последовательная версия программы Stream

Алгоритм создает три N-элементных вектора и выполняет серии операций над этими векторами. Время выполнения каждой из операций замеряется при помощи операций TIC и TOC.

STREAM($\mathbf{a}, \mathbf{b}, \mathbf{c} : \mathbb{R}^N, q : \mathbb{R}, n : \mathbb{Z}$)

```
t_copy, t_scale, t_add, t_copy : R
for i = 1 : n
    do
        TIC
          c = a
        t_copy += TOC
        TIC
          b = q c
        t_scale += TOC
        TIC
          c = a + b
        t_add += TOC
        TIC
          a = b + q c
        t_triadd += TOC
```

Цель программы Stream заключается в измерении пропускной способности памяти в байтах в секунду. Для теста производительности требуется, чтобы векторы пред-

ставляли собой 8-байтовые значения с плавающей запятой с двойной точностью, в результате чего получаются следующие формулы пропускной способности:

Copy Bandwigth:	$16\ n\ N/t_{copy}$
Scale Bandwigth:	$16\ n\ N/t_{scale}$
Add Bandwigth:	$24\ n\ N/t_{add}$
Triadd Bandwigth:	$24\ n\ N/t_{triadd}$

Проверка правильности является важной частью любого теста производительности. Если значениям вектора **a** были присвоены начальные значения a_0, конечные результаты могут быть проверены по формулам

$$a_{n-1} = (2q + q^2)^{n-1}\ a_0$$
$$a_n(:) = (2q + q^2)^n\ a_0$$
$$b_n(:) = q\ a_{n-1}$$
$$b_n(:) = (1 + q)\ a_{n-1}$$

Хотя программа Stream очень простая, в ней содержится суть многих программ. В частности, она работает на последовательностях векторов с определенными начальными значениями. Она повторяет последовательность стандартных векторных операций и выдает ответы, правильность которых может быть проверена.

Параллельная программа Stream может быть реализована довольно легко с использованием распределенных массивов. До тех пор, пока векторы будут использовать одинаковое размещение, результирующая программа не будет требовать межпроцессорной коммуникации. Строгое соблюдение требования отсутствия коммуникации можно обеспечить, используя в программе конструкцию *.loc*. Этот подход рекомендуется вследствие того, что он гарантирует отсутствие какой-либо коммуникации. Если случайно было использовано размещение, требующее коммуникации, результирующая программа либо выдаст ошибку, либо не пройдет проверку. Это размещение приводит к параллельному алгоритму, представленному с использованием P-записи в Алгоритме 8.2. Визуальное представление реализации параллельного алгоритма программы Stream приведено на рисунке 8.1.

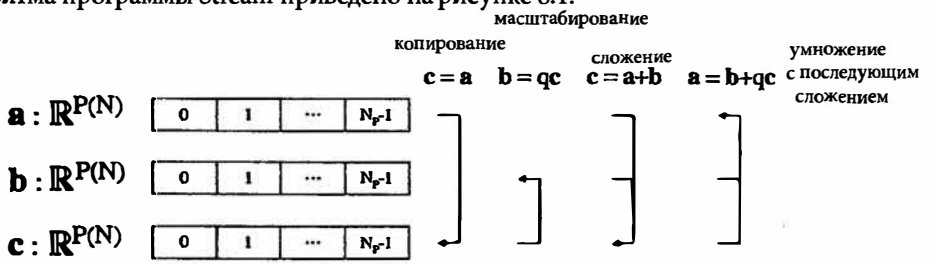


Рисунок 8.1. Параллельная реализация программы Stream

Каждый вектор является распределенным массивом. Если каждый вектор имеет одинаковую параллельную карту размещения, результирующая программа не требует коммуникации

Алгоритм 8.2. Параллельная версия программы Stream

Параллельный алгоритм создает три N-элементных вектора и выполняет серии операций только над локальными частями этих векторов. Для получения общей параллельной пропускной способности результирующие временные показатели можно усреднить.

```
PARALLELSTREAM(a, b, c : R^P(N), q : R, n : Z)
 1  t_copy, t_scale, t_add, t_copy : R
 2  for i = 1 : n
 3      do
 4          TIC
 5            c.loc = a.loc
 6          t_copy += TOC
 7          TIC
 8            b.loc = q c.loc
 9          t_scale += TOC
10          TIC
11            c.loc = a + b.loc
12          t_add += TOC
13          TIC
14            a.loc = b + q c.loc
15          t_triadd += TOC
```

8.3. Код

Следующим шагом является написание параллельной программы Stream. Параллельный код может быть реализован простым преобразованием векторов из обычных массивов в распределенные массивы. Основная часть параллельного кода приведена ниже (полный листинг расположен в директории `Examples/Stream/pStream.m`). Следует понимать, что один и тот же код выполняется каждым экземпляром MATLAB. Единственным различием между экземплярами является наличие у каждого из них уникального P_{ID} (или `Pid`), который используется для определения того, какие части распределенного массива принадлежат каждому из экземпляров MATLAB (т. е. a.loc, b.loc и c.loc).

Листинг кода 8.1. Параллельная MATLAB-программа Stream

```
1  PARALLEL = 1;
2  NTRIALS = 4; lgN = 25; N = 2.^lgN;
3  A0 = 1.0; B0 =2.0; C0 = 0.0; q = 3.14;
4
5  ABCmap = 1; SyncMap = 1;     % Serial maps.
6  if (PARALLEL)                % Parallel maps.
7     ABCmap = map([1 Np], ,0:Np-1);
8     SyncMap = map([Np 1], ,0:Np-1);
9  end
10 tic;                         % Allocate vectors.
11    Aloc = local(zeros(1, N, ABCmap)) + A0;
12    Bloc = local(zeros(1, N, ABCmap)) + B0;
13    Cloc = local(zeros(1, N, ABCmap)) + C0;
14 Talloc = toc;
15 tic;                         % Synch launch.
16    sync = agg(zeros(1, Np, SyncMap));
17 Tlaunch = toc;
18
19 TsumCopy=0.0; TsumScale =0.0; TsumAdd=0.0; TsumTriad=0.0;
20 for i = 1:NTRIALS
21    tic;
22         Cloc(:,:)  = Aloc;          % Copy.
23    TsumCopy = TsumCopy + toc;
24    tic;
25         Bloc(:,:)  = q*Cloc;        % Scale.
26    TsumScale = TsumScale + toc;
27    tic;
28         Cloc(:,:)  = Aloc + Bloc;   % Add.
29    TsumAdd = TsumAdd + toc;
30    tic;
31         Aloc(:,:)  = Bloc + q*Cloc; % Triadd.
32    TsumTriad = TsumTriad + toc;
33 end
```

Перевод

5 % Последовательные карты размещения;

6 % Параллельные карты размещения;

10 % Размещение векторов;

15 % Синхронизация запуска;

22 % Copy – Копирование;

25 % Scale – Масштабирование;

28 % Add – Сложение;

31 % Triadd– Умножение с последующим сложением.

В вышеприведенной программе:

Строки 5–9 создают параллельные карты размещения ABCmap и SyncMap. Флаг PARALLEL является дополнительным инструментом отладки, позволяющим включать и выключать распределенные массивы A, B и C.

Строки 11–13 создают распределенные векторы A, B и C; извлекают их локальные части; затем присваивают им начальные значения A_0, B_0 и C_0. Так как программа не требует коммуникации, распределенные массивы A, B и C никогда фактически не выделяются. Вместо этого создаются только их локальные части Aloc, Bloc и Cloc. Это упрощает программу и уменьшает общий требуемый объем памяти для выполнения программы посредством исключения ненужного дублирования.

Строка 14 сохраняет время, затраченное на выделение этих массивов.

Строка 16 показывает, каким образом можно создать барьер или точку синхронизации по всем P_{ID} программы. В этом случае, создается простой *Np*-элементный распределенный вектор и затем агрегируется по всем процессорам. Агрегирование вектора обеспечивает ожидание всеми $P_{ID} > 0$ в этой точке до того момента, как $P_{ID} = 0$ сможет продолжать работу. Эта точка синхронизации требуется для получения истинного измерения производительности системы. Такие точки часто вставляются в параллельные программы, когда необходимо, чтобы все P_{ID} были приблизительно в одной точке исполнения. Использование команды agg() для синхронизации синхронизирует все $P_{ID} > 0$ с $P_{ID} = 0$. Более агрессивная синхронизация достигается при помощи команды agg_all(), при которой требуется, чтобы все P_{ID} отправляли свои значения каждому другому процессору.

Строки 20–33 выполняют операции copy (копирование), scale (масштабирование), add (сложение) и triadd (умножение с последующим сложением) над векторами Aloc, Bloc и Cloc. Время для каждой из этих операций суммируется по каждому пробному выполнению.

Вышеуказанный код является ярким примером подхода к программированию на основе карт размещения. После создания карт ABCmap и SyncMap они используются для получения локальных распределенных векторов Aloc, Bloc и Cloc.

На данном примере кода можно проследить следующие свойства подхода к программированию на основе карт размещения:

- Незначительное влияние на код. Программа преобразована из последовательной в параллельную путем добавления около 10 дополнительных строк кода.
- Последовательный код сохраняется. Посредством установки PARALLEL=0 исключается использование распределенных массивов, и программа может выполняться последовательно без поддержки параллельных библиотек.
- Небольшой объем использования параллельной библиотеки. Потребовалось только три функции параллельной библиотеки: map, local и agg.
- Масштабируемость. Код может выполняться на задачах различного размера и любом количестве процессоров при условии $N > Np$, пока векторы имеют одинаковые карты размещения.
- Ограниченная коммуникация. Так как используются локальные переменные, использование коммуникации строго контролируется.

- Независимость от карт размещения. Если для всех трех векторов используется одна и та же карта, программа будет работать для любого распределения во втором измерении (т. е. блочное, циклическое, блочно-циклическое). Чтобы программа стала независимой от карт размещения для любой их комбинации, необходимо заменить локальные операции на глобальные. Например, если операция `scale` (масштабирование) была реализована с использованием **`C(:,:) = A`**, она будет выполняться корректно, независимо от карт. Однако если `A` и `C` имеют разные карты, возникнет необходимость в значительном объеме коммуникации.
- Гарантия производительности. Так как команды операций `copy`, `scale`, `add` и `triadd` работают над локальными переменными, которые представляют собой обычные числовые массивы MATLAB, гарантируется отсутствие каких-либо скрытых ухудшений производительности при параллельном выполнении данных строк кода.

8.4. Отладка

Как упоминалось ранее, параллельная отладка представляет собой непростую задачу. Основным преимуществом подхода на основе карт размещения является то, что оно поддается четко определенному процессу кодирования–отладки–тестирования. Соответствующие шаги указаны в таблице 2.2

Шаг 1. Выполнение программы последовательно на локальной машине с векторами массивов `Aloc`, `Bloc` и `Cloc`, созданными как обычные массивы (т. е. `PARALLEL=0`). Так проверяется, что исходный последовательный код остается корректным после применения функции `local`.

Шаг 2. Выполняет программу последовательно на локальной машине с векторами `Aloc`, `Bloc` и `Cloc`, полученными из распределенных массивов (т. е. `PARALLEL=1`), и проверяет правильность работы программы с распределенными массивами в качестве входных данных.

Шаг 3. Выполнение программы параллельно на локальной машине. Данный шаг определяет, являются ли параллельные карты, которые генерируют `Aloc`, `Bloc` и `Cloc`, корректными. Из всех шагов этот, вероятнее всего, является самым важным, так как на нем выявляется больше всего ошибок.

Шаг 4. Выполнение программы параллельно на удаленной машине.

Шаг 5. Выполнение программы параллельно на удаленной машине, но с задачей интересующего размера. До этого момента тестирование должно было осуществляться на задаче малого размера. Данный шаг позволяет определить производительность на реальной задаче.

Эти шаги являются существенным отражением этапов, указанных в разделе «Начало работы», и на каждом из них тестируются определенные аспекты параллельного кода. В дополнение к отладке функциональности программы вышеуказанный процесс также может быть использован для оптимизации производительности кода (см. следующий раздел). Помимо указанных выше шагов, тест производительности Stream использует некоторые дополнительные техники кодирования для помощи в отладке.

Программа Stream отображает параметры программы, которые описывают размер задачи *N*, количество необходимых байтов, количество пробных выполнений, количество процессоров *Np* и P_{ID} процессора, выполнившего вывод. Это осуществляется

на каждом P_{ID} программы. Сравнение результирующих значений путем обращения к различным файлам с выводом от каждого P_{ID} (т. е. `MatMPI/*.out`) является действенным способом выявления ошибок.

В программе Stream также отображается время инициализации и время запуска для каждого P_{ID} программы, которые описывают количество времени, затрачиваемое до начала выполнения программы. Признаком того, что что-то происходит не так на ранних стадиях, является значительное отличие этих значений для разных P_{ID}.

Так как Stream является тестом производительности, производительность выполнения измеряется и выводится в единицах, относящихся к процессору. Тем не менее, это хорошая практика для создания программ общего назначения. Значения производительности, значительно отличающиеся между экземплярами, могут стать показателем наличия более серьезных проблем в программе (например, создание массивов очень разных размеров).

Наконец, программа Stream осуществляет проверку корректности, которая проверяет выдачу программой корректных результатов. В реальных приложениях такую проверку как правило произвести невозможно, однако можно выполнить приближенные проверки, которые бывают очень полезны. Например, часто можно проверить знак или сумму значений в массиве. Повторимся, что отображение этих значений для каждого экземпляра может выявить ошибки, зависящие от P_{ID}.

8.5. Тестирование

Программа Stream является эффективной программой для анализа неявных факторов производительности целого ряда методов написания кода. Тестирование параллельной программы – при условии, что она уже отлажена и выдает корректные ответы – в основном заключается в определении количества времени, которое требуется для ее выполнения и выяснения, насколько ее производительность соответствует ожидаемой. Почти все данные о производительности, необходимые для проведения такого анализа, были получены в разделе «Начало работы» (см. таблицу 8.1).

Первый тест заключается в сравнении производительности при выполнении программы с использованием $Np = 1$ с выключенными (`PARALLEL=0`) и включенными (`PARALLEL=1`) распределенными массивами. Эти данные представлены в столбцах 2 и 3 таблицы 8.1. В данном случае различия в производительности пренебрежимо малы; они показывают, что распределенные массивы не вызывают ухудшение производительности. Если использование распределенных массивов с $Np = 1$ значительно замедляет работу, необходимо устранить эту проблему. Очень трудно получить какое-либо преимущество при $Np > 1$, если при $Np = 1$ операции выполняются медленнее, чем в исходном последовательном коде. Работа только на локальных частях распределенных массивов практически гарантирует, что производительность будет такой же, как при работе без использования распределенных массивов.

При подробном рассмотрении значений в столбцах 2 и 3 можно увидеть, что они делятся на три категории. Строки 2, 3, 7 и 8 содержат время выделения и время запуска. Строки 4 и 5 содержат локальную пропускную способность по данным $P_{ID} = 0$ на различных операциях. Строки 9 и 10 содержат глобальную пропускную способность, которую можно оценить умножением локальной пропускной способности на Np и точно

вычислить посредством усреднения значений по всем экземплярам. В столбцах 2 и 3, где *Np* = 1, локальная и глобальная пропускные способности идентичны. В целом, операция `copy` имеет наивысшую пропускную способность, в то время как `scale`, `add` и `triadd` – очень близки по значению. Основной причиной таких различий является функция деталей архитектуры процессора. В таблице 8.1 пропускная способность согласуется с пиковой пропускной способностью основной памяти для конкретного процессора. Таким образом, можно заключить, что программа работает близко к тому пиковому значению, которое может обеспечить используемое оборудование. Такого рода данные о производительности трудно переоценить при оптимизации программы. С их помощью можно провести количественный анализ необходимости дополнительных усилий на оптимизацию. Если программа выполняется с производительностью менее чем 1% от пикового значения, скорее всего, дополнительное усилие будет оправданным. Если программа работает с производительностью более 10% пикового значения, эти усилия могут не оказаться столь плодотворными. Наконец, в данном случае пиковое значение не является пиковым значением процессора, но пиковым значением пропускной способности памяти. Программа Stream относится к программам, которые не предполагают большого числа операций над каждым элементом данных и ограничиваются пропускной способностью памяти, а не скоростью процессора.

Данные в таблице 8.1 были получены на восьмиядерной системе, в которой все ядра используют память совместно. Столбцы 3–7 показывают эффект увеличения количества экземпляров программы. Во-первых, время размещения остается почти неизменным. Во-вторых, возрастает время запуска, так как запуск четырех экземпляров программы занимает больше времени, чем запуск двух экземпляров. Это может стать значимым фактором при большом числе *Np*, при этом необходимо тщательно подобрать соответствующий механизм запуска. В-третьих, что наиболее важно, локальная пропускная способность по всем операциям несколько падает с увеличением *Np*. Другими словами, глобальная пропускная способность несколько возрастает с увеличением числа *Np*. Даже при наличии дополнительных ядер для обработки увеличение количества экземпляров программы не приведет к ускорению производительности. В данной ситуации единственным способом улучшить производительность является увеличение числа процессоров, имеющих отдельную память.

Увеличение количества экземпляров в рамках обрабатывающего узла с общей памятью (обозначенное здесь 1**Np*) называется «вертикальным масштабированием». Увеличение количества экземпляров за пределами одного обрабатывающего узла с общей памятью (обозначенное здесь *Np**1) называется «горизонтальным масштабированием». Относительная производительность вертикального и горизонтального масштабирования для операции triadd в программе Stream приведена в таблице 8.1. Эти результаты показывают, что программа Stream получает выгоду от вертикального масштабирования, однако при горизонтальном она масштабируется очень хорошо. Это явление весьма характерно для многих прикладных программ, интенсивно использующих память, и имеет явные указания на то, каким образом выполнять программу, и даже на каких типах параллельных машин эти виды программ будут выполняться более эффективно.

Таблица 8.1. Производительность операции triadd в тесте Stream на многоядерных и многоузловых системах

Относительное время размещения, абсолютное время запуска, относительная локальная и глобальная пропускная способность для операции triadd в тесте Stream на многоядерных и многоузловых вычислительных системах для различных значений *Np*. Данные о производительности приведены для фиксированного размера задачи (*N* – константа). Все значения нормализованы по столбцу 2, в котором *Np* = 1 с выключенными распределенными массивами (`PARALLEL=0`). Столбцы 3–7 содержат значения с включенными распределенными массивами (`PARALLEL=1`). Запись 1*8 означает, что *Np* = 8 и все экземпляры P_{ID} были выполнены на одном процессорном узле при совместно используемой памяти. Запись 16*1 означает, что *Np* = 16 и каждый P_{ID} был выполнен на своем индивидуальном процессорном узле с использованием отдельной памяти

Многоядерные процессоры:	1	1*1	1*2	1*4	1*8	
Относительное время размещения	1	1.0	1.0	0.8	0.55	
Время запуска (с)	0.0	0.0	0.5	1.2	1.8	
Относительная локальная пропускная способность	1	1.0	0.89	0.86	0.64	
Относительная глобальная пропускная способность	1	1.0	1.8	3.4	5.1	
Многоузловые процессоры:	1	1*1	2*1	4*1	8*1	16*1
Относительное время размещения	1	1.0	1.0	1.0	1.0	1.0
Врамя запуска (с)	0.0	0.0	1.7	2.2	2.4	2.5
Относительная локальная пропускная способность	1	1.0	1.0	1.0	1.0	1.0
Относительная глобальная пропускная способность	1	1.0	2.0	4.0	8.0	18.0

Другим важным аспектом является тестирование того, как производительность приложения масштабируется с размером задачи *N*. Из таблицы 8.1 понятно, что если размер задачи масштабируется с *Np* при горизонтальном масштабировании, следует ожидать линейного увеличения производительности. Из-за этого увеличения возникает вопрос, каков оптимальный размер задачи для выполнения на одном узле, так как это станет определяющим фактором общей производительности при горизонтальном масштабировании приложения. На рисунке 8.2 показана относительная производительность операции `triadd` в Stream как функция относительного размера задачи. График производительности имеет плоский характер за пределами 1000-кратного диапазона размера задачи и затем возрастает на 50% при движении в сторону небольших размеров задач. Это увеличение обусловлено тем, что задача помещается в кэш-память процессора, пропускная способность которого выше.

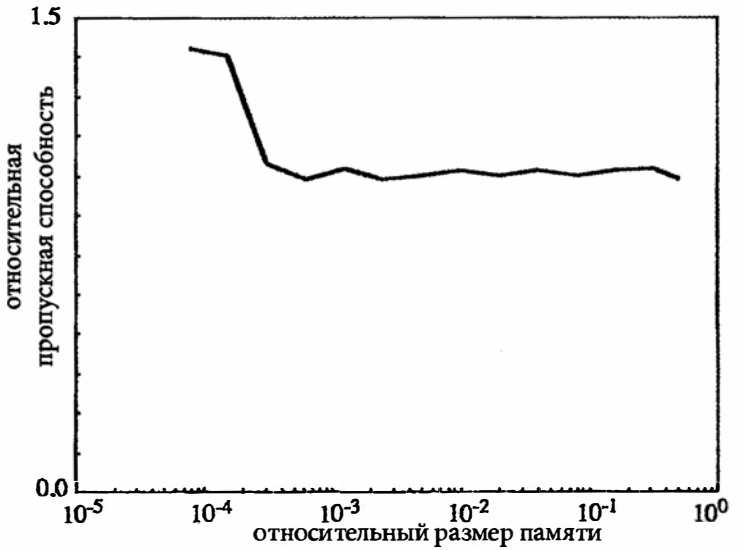


Рисунок 8.2. Производительность программы Stream относительно размера задачи
Относительная пропускная способность операции triadd в Stream как функция относительного размера выполняемой задачи. Производительность остается неизменной на протяжении большого участка диапазона и увеличивается на 50% для задач весьма небольшого размера по причине влияния кэш-памяти

После сбора и изучения всех данных о производительности программы Stream можно сделать следующие выводы о выполнении приложений такого типа на кластерах многоядерных процессорных узлов. Во-первых, выполнение множества экземпляров программ (1*Np) на одном процессорном узле дает определенную выгоду. Во-вторых, производительность одного узла почти не изменяется с увеличением размера задачи N. В-третьих, производительность должна увеличиваться линейно при выполнении программы на множестве узлов (Np*1).

Список литературы

1. *[McCalpin 2005] John McCalpin, 2005, STREAM: Sustainable Memory Bandwidth in High Performance Computers, http://www.cs.virginia.edu/stream/*

Глава 9
Программа RandomAccess

Резюме

Тест производительности RandomAccess является простой программой, предназначенной для создания нагрузки на коммуникационную сеть параллельной вычислительной машины. Это типичная программа для класса задач, выполнение которых неэффективно на стандартной параллельной машине. Программа RandomAccess прекрасно иллюстрирует целый ряд сложных принципов параллельного программирования. Шаблон коммуникации данного теста производительности трудно реализовать при помощи чистых распределенных массивов. Требуется применение комбинированного подхода к параллельному программированию с использованием распределенных массивов и обмена сообщениями. В данной главе рассматриваются две версии этого теста с использованием различных шаблонов коммуникации, которые иллюстрируют, что производительность при использовании меньшего количества больших сообщений выше производительности при использовании множества сообщений малых размеров.

Программа RandomAccess была представлена в главе 7 в качестве теста производительности для измерения задержки между процессорами в сети [Dongarra and Luszczek 2005]. Программа RandomAccess также является хорошим примером сложного параллельного приложения со сложным шаблоном коммуникации и позволяет продемонстрировать некоторые полезные методы составления параллельного кода, в частности способ написания комбинированной программы, которая использует распределенные массивы для организации взаимодействия, а обмен сообщениями для фактической отправки сообщений.

Программа RandomAccess создает большую N-элементную таблицу и многократно выполняет простую операцию обновления в этой таблице. Местоположение обновлений определяется генерированием случайных позиций по всему диапазону индексов таблицы. Таким образом, существует лишь ничтожная вероятность, что два последовательных обновления будут находиться рядом в таблице. Так как количество операций, выполняемых над каждым элементом, невелико и обновления могут производиться одновременно только в небольшом количестве, производительность RandomAccess ограничена задержкой коммуникационной сети параллельной вычислительной системы. Программа RandomAccess является отличной иллюстрацией класса приложений, которые обычно неэффективно работают на стандартных параллельных машинах. Не является редкостью то, что часть большого параллельного приложения имеет целый ряд сходств с RandomAccess. В данной главе описываются как подходы в кодировании, так и алгоритмические способы минимизации влияния на производительность части программы, аналогичной RandomAccess.

9.1. Начало работы

Для выполнения программы запустите MATLAB и перейдите в директорию Examples/RandomAccess (т. е. наберите `cd Examples/RandomAccess`). Отредактируйте файл `pRandomAccess.m` и установите `PARALLEL=0`. В командной строке введите

```
pRandomAccess
```

Так как RandomAccess является тестом производительности, она выдаст большой объем данных о производительности, аналогичных следующим:

```
Allocation Time (sec)                          =1.5989
Distributed table size                         =2~25=33554432 words
Distributed table size (bytes)                 =268435456
Local table size (bytes)                       =268435456
Number of updates                              =8388608
Block size (should be 1024)                    =1024
Launch Time (sec)                              =0.005367
Run time (sec)                                 =3.2708
Giga Updates Per Sec                           =0.0025647
Validating results
Validate time (sec)                            =120.7053
Error rate                                     =0.001209
Validation Passed
```

Перевод
Allocation Time (sec) – время размещения (с);
Distributed table size – размер распределенной таблицы;
Distributed table size (bytes) – размер распределенной таблицы (байты);
Local table size (bytes) – размер локальной таблицы (байты);
Number of updates – количество обновлений;
Block size (should be 1024) – размер блока (должен быть 1024);
Launch Time (sec) – время запуска (с);
Run time (sec) – время выполнения (с);
Giga Updates Per Sec – 109 обновлений в секунду;
Validating results – результаты проверки;
Validate time (sec) – время проверки (с);
Error rate – частота появления ошибок;
Validation Passed – проверка пройдена.

Вышеуказанную информацию можно разбить на четыре категории. Первая – параметры программы, которые описывают размер задачи *N*, количество необходимых байтов, количество выполненных обновлений и размер блока этих обновлений. Вторая категория включает в себя время инициализаций массива и время запусков, которые описывают количество времени, затрачиваемое до начала выполнения программы. Третья категория – это время выполнения и производительность в миллиардах обновлений в секунду (GUPS – Giga Updates Per Second). Четвертая категория – значения проверки правильности. Эти результаты будут рассмотрены далее в разделе «Тестирование». Сейчас отредактируйте файл `pRandomAccess.m`, установите `PARALLEL=1` (для включения параллельных частей кода) и `VALIDATE=0` (для сокращения времени выполнения). Выполните `RandomAccess` снова, используя следующие команды:

```
eval(pRUN('pRandomAccess' ,1,{}))
eval(pRUN('pRandomAccess' ,2,{}))
eval(pRUN('pRandomAccess' ,4,{}))
```

Данные команды выполняют программу RandomAccess с использованием одного, двух и четырех *Np*. При выполнении на параллельной вычислительной машине выдаются значения производительности, аналогичные указанным в таблице 9.1 (см. раздел «Тестирование»). Эти результаты будут рассмотрены позже в разделе «Тестирование».

Важное примечание. Последний аргумент команды `pRUN` определяет, на какой параллельной вычислительной машине будет выполняться программа. Может возникнуть необходимость в получении надлежащих установочных параметров у вашего системного администратора; для получения более подробной информации обращайтесь на веб-сайт книги (http://www.siam.org/KepnerBook). Установка последнего аргумента значением {} приводит к выполнению множества копий программы на вашей локальной машине и получению производительности, отличной от указанной в таблице 9.1.

9.2. Параллельная реализация

Спецификация теста производительности RandomAccess предполагает создание *N*-элементной таблицы беззнаковых 64-битных целых чисел $\mathbf{x} : Z^N$ такой, что объем памяти, занятый **x**, представляет собой значительную часть общей доступной памяти. Обновление совершается над этой таблицей следующим образом:

```
x(i) = XOR(x(i), r),
```

где $\mathbf{r} : Z^{1024}$ – 1024-элементный вектор случайных целых чисел, а $i = \mathrm{AND}(r, N - 1)$ – вектор табличных позиций для обновления. Вышеуказанные операции повторяются $4N/1024$ раз (хотя точные результаты производительности как правило могут быть получены с меньшим количеством итераций). Исходный алгоритм RandomAccess показан в Алгоритме 9.1 (подробнее см. в подписи к Алгоритму)

Алгоритм 9.1. Последовательная реализация RandomAccess

Алгоритм создает N-элементную таблицу и выполняет обновления случайных позиций таблицы. Строки 2–4 присваивают начальные значения и вычисляют количество обновлений и блоков для использования. Строка 7 получает следующий набор из 1024 случайных чисел. Строка 8 вычисляет индекс таблицы, производный от этих случайных значений. Строка 9 выполняет обновление над таблицей **x**.

```
RANDOMACCESS(x : Z^N)
 1  r, i : Z^1024
 2  N_up = 4N        N_b = N_up/1024
 3  r = RANDSTART( 0 : 1023 )
 4  TIC
 5  for b = 1 : N_b
 6        do
 7              r = RAND(r)
 8              i = AND(r, N − 1)
 9              x(i) = XOR(x(i), r)
10  t_up = TOC
```

Цель программы RandomAccess заключается в измерении возможности вычислительной машины выполнять обновления малых размеров в памяти. Производительность измеряется в GUPS и вычисляется посредством

10^9 обновлений в секунду (Giga Updates Per Second GUPS): $N_{up}/t_{up}/10^9$.

Проверка правильности является важной частью любого теста производительности. Если значения таблицы **x** были проинициализированы так, что $\mathbf{x}(i) = i-1$, то сброс в исходное значение генератора случайных чисел и повторное выполнение теста производительности вернет **x** к начальным значениям. При сравнении с корректными начальными значениями можно вычислить количество ошибок, совершенных в процессе обновления:

$$N_{err} = \sum_{i=1}^{N} \mathrm{NOT}(\mathrm{x}(i) == (i\text{-}1)).$$

В отличие от многих тестов производительности, RandomAccess допускает частоту появления ошибок

$$N_{err}/N_{up} < 0.1\% .$$

Ошибки могут возникать по нескольким причинам. В Алгоритме 9.1 основным источником возникновения ошибок будет являться ситуация, когда **i** будет содержать повторяющиеся значения. Когда все обновления выполняются блоком (это обычно делается в целях повышения производительности), только одно из повторных обновлений будет выполнено. Во время проверки на корректность программа должна быть выполнена способом, исключающим ошибки, поэтому строка 10 заменяется следующим циклом:

for $j = 1: 1024$
do $\mathbf{x}(\mathbf{i}(j)) = \mathrm{XOR}(\mathbf{x}(\mathbf{i}(j)), \mathbf{r}(j))$.

Вышеуказанный цикл не генерирует ошибок, однако, как было видно из предыдущего раздела, приводит к более медленному выполнению.

Хотя RandomAccess является очень простой программой, она отражает суть многих программ. В частности, она работает на таблицах с определенными начальными значениями. Она повторяет последовательность стандартных обновлений в непредсказуемых позициях в памяти и выдает ответы, корректность которых может быть проверена.

В параллельной программе RandomAccess каждый P_{ID} генерирует блок из 1024 случайных чисел, в результате получаем 1024 Np обновлений во время каждой итерации программы. Если значения **i**, генерируемые каждым P_{ID}, были ограничены локальным диапазоном **x**, RandomAccess стала бы массово-параллельным тестом производительности. Однако это не соответствует действительности. Каждый P_{ID} генерирует набор $i \in [1, N]$, который охватывает весь диапазон таблицы **x**. Таким образом, каждый P_{ID} должен определять, какой из своих индексов необходимо отправить другим P_{ID}.

9.2.1. Рассеивающий алгоритм

Так как параллельная программа RandomAccess характеризуется таким динамическим поведением, ее трудно реализовать с использованием только распределенных массивов. Комбинированный метод распределенных массивов и обмена сообщениями является одной из эффективных моделей программирования этого теста производительности. В комбинированном подходе используют распределенные массивы для распределения **x** совместно с обменом сообщениями для отправления обновлений разными P_{ID} (см. Алгоритм 9.2). Этот алгоритм по сути «рассеивает» обновления по вычислительной машине.

Алгоритм 9.2. Рассеивающий параллельный алгоритм для программы RandomAccess

Алгоритм использует N-элементную распределенную таблицу. Каждый P_{ID} генерирует набор из 1024 случайных позиций, определяет, какому P_{ID} принадлежит каждая позиция, и затем отправляет обновления правильному P_{ID}. Строка 3 инициализирует последовательность случайных чисел соответствующим значением для каждого P_{ID}. Строка 4 вычисляет идентификаторы экземпляров P_{ID}, которым отправляет и от которых получает данные каждый P_{ID}. Строка 10 устанавливает тэг сообщения, который полезен при отладке потоков сообщений. Строка 11 выполняет цикл на отправляющих P_{ID}. Строки 12 и 13 определяют, какие обновления принадлежат **p**, и отправляют их в сообщении. Строка 14 инициализирует получаемые обновления теми позициями, которые будут локальными для принимающих P_{ID}. Строки 15 и 16 выполняют цикл по P_{ID} для приема и дополняют принимаемые сообщения к значениям, подлежащим обновлению. Строки 19 и 20 выполняют обновления в локальной части **x**.

```
PARALLELRANDOMACCESSSPRAY(x : Z^{P(N)})
 1  r, i : Z^{1024}
 2  N_up = 4N        N_b = N_up/1024
 3  r = RANDSTART( (N_b/N_P)(1024P_ID + 0 : 1023) )
 4  P_ID^send = P_ID^recv = {0, ..., N_P - 1}\P_ID
 5  TIC
 6  for b = 1 : N_b
 7        do
 8           r = RAND(r)
 9           i = AND(r, N - 1)
10           tag = b  mod 32
11           for p ∈ P_ID^send
12                 do
13                    r_send = r( x.loc_p⟨1⟩ ≤ i ≤ x.loc_p⟨end⟩ )
14                    SENDMSG(p, tag, r_send)
15           r_recv = r( x.loc_P_ID⟨1⟩ ≤ i ≤ x.loc_P_ID⟨end⟩ )
16           for p ∈ P_ID^recv
17                 do
18                    r_recv = APPEND(r_recv, RECVMSG(p, tag))
19           i = AND(r_recv, N - 1) - x.loc_P_ID⟨1⟩ - 1
20           x.loc(i) = XOR(x.loc(i), r_recv)
21  t_up = TOC
```

Алгоритм 9.2 показывает, как распределенные массивы и обмен сообщениями могут использоваться совместно. В частности, распределенный массив **x** используется для разбиения данных между P_{ID}. Информация в **x** также используется для выбора значений, подлежащих отправке, путем сравнения позиций в **i** с максимальным и минимальным глобальным индексом на каждом P_{ID}. С точки зрения обозначений строки 13 и 15 используют угловые скобки для извлечения этого глобального индекса. Помните, что ссылка на определенный индекс локальной части выполняется обычным способом: **x**.*loc(i)*. Как правило, глобальный индекс **x**(*i*) определяет элемент массива, который отличается от элемента, определяемого локальным индексом **x**.*loc(i)*. Если необходимо получить локальную часть от определенного P_{ID}, ее можно обозначить как **x**.loc_p. Когда требуется глобальный индекс определенного локального значения, он обозначается как **x**.*loc(i)*. Очевидно, i = **x**(i). В более общем виде это равенство будет выглядеть как

$$x(\mathbf{x}.loc(i)) = \mathbf{x}.loc(i).$$

Минимальные и максимальные глобальные индексы на конкретном P_{ID}, таким образом, задаются при помощи **x**.$loc_p(1)$ и **x**.$loc_p(end)$.

Рассеивающий алгоритм, представленный в Алгоритме 9.2, является традиционным методом реализации параллельной программы RandomAccess, он отправляет N_P^2 сообщений с приблизительно 1024/*Np* обновлениями в каждом сообщении. Как было указано, все P_{ID} сначала отправляют в одинаковом порядке, что, скорее всего, приведет к проявлению узких мест, так как каждый P_{ID} будет сначала пытаться отправить экземпляру P_{ID} = 0, затем экземпляру P_{ID} = 1 и т. д. Несложным улучшением этого является

изменение порядка выбора экземпляров для отправки: P_{ID}^{recv} так, чтобы два P_{ID} не отсылали данные в одно и то же место одновременно. Аналогично может быть установлен порядок экземпляров для приема P_{ID}^{recv} так, чтобы каждый P_{ID} получал сообщения в порядке, в котором они были ему отправлены. Шаблон коммуникации этого модифицированного рассеивающего алгоритма показан на рисунке 9.1.

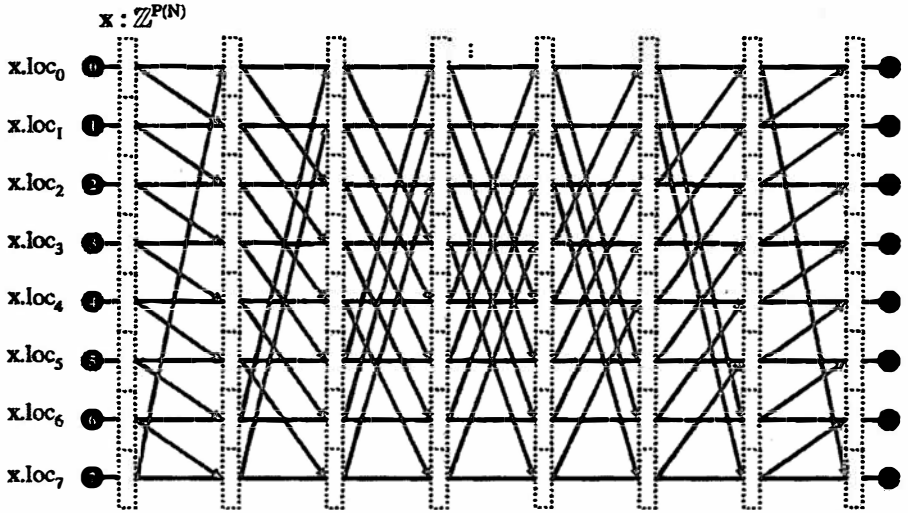


Рисунок 9.1. Шаблон коммуникации рассеивающего алгоритма программы RandomAccess

Каждый P_{ID} отправляет сообщение всем остальным экземплярам P_{ID}, в результате чего получаем N_P^2 сообщений с $1024/N_p$ обновлениями на каждое сообщение. Изображенный шаблон составлен таким образом, что никакие два P_{ID} не могут отправить данные по одному и тому же направлению одновременно

9.2.2. Древовидный алгоритм

Рассеивающий алгоритм RandomAccess является наиболее простым параллельным методом реализации этого теста производительности. Однако он имеет ряд ограничений, которые нуждаются в изменениях. Во-первых, по мере достижения *Np* значения 1024 все увеличивается вероятность того, что у некоторых P_{ID} не будет обновлений для отправки другим P_{ID}. Этот подход нуждается в более динамичном алгоритме, в котором каждый P_{ID} проверяет, пришли ли какие-либо сообщения, таким образом позволяя алгоритму работать для любого значения *Np*. К сожалению, даже при этих изменениях рассеивающий алгоритм работает медленно, так как он отправляет N_p^2 очень малых сообщений на каждую итерацию. Следовательно, все конкурентные версии RandomAccess используют разные алгоритмы, которые отправляют меньшее количество сообщений большего размера.

Древовидный алгоритм RandomAccess для сокращения общего количества отправляемых сообщений превращает каждый P_{ID} в «программный маршрутизатор». При этом методе каждый P_{ID} разбивает свои обновления на группы позиций из верхней и нижней половины таблицы, и либо оставляет себе, либо отправляет их другому

P_{ID}. После первого этапа каждый P_{ID} будет иметь набор обновлений, диапазон которых сократился с N до $N/2$. После двух этапов диапазон значений на каждом P_{ID} уменьшается до $N/4$. Повторение этого этапа в общей сложности $\lg(Np)$ раз сокращает диапазон значений до момента, в котором они все лежат в пределах **x**.*loc* каждого P_{ID}. Древовидный алгоритм изображен в Алгоритме 9.3, а соответствующий шаблон коммуникации показан на рисунке 9.2. Данная версия работает, если Np является точной степенью двойки. Простая модификация алгоритма позволяет ему работать для любого разложимого на множители Np.

Преимущество древовидного алгоритма заключается в том, что только $Np\lg(Np)$ сообщений с 512 обновлениями на каждое сообщение отправляется во время каждой итерации древовидного алгоритма RandomAccess, что в $Np/\lg(Np)$ раз меньше, чем в случае рассеивающего алгоритма. Компромисс заключается в том, что в то время как рассеивающий алгоритм отправляет $8192Np$ байт за каждую итерацию, древовидный алгоритм отправляет $4096Np\lg(Np)$ обновлений за каждую итерацию. Следовательно, притом, что древовидный алгоритм отправляет меньшее количество сообщений, ему нужно отправить $2/\lg(Np)$ больше байт информации. Обычно отправка большего объема информации занимает больше времени. Однако если сообщения имеют малый размер, задержка сети превалирует над временем отправки. В этом случае для отправки 1-байтного сообщения понадобится столько же времени, сколько и на отправку 4096-байтного, что приводит к значительному преимуществу древовидного алгоритма в отношении производительности.

Алгоритм 9.3. Древовидный параллельный алгоритм для программы RandomAccess

Строка 9 инициализирует значения, которые будут использоваться в древовидном алгоритме. Строка 10 начинает цикл `while`, который будет продолжаться до того момента, пока все обновления не будут направлены своим правильным P_{ID}. Строка 12 вычисляет P_{ID} для обмена данными. Строка 13 вычисляет P_{ID}, используемый для разбиения **r** на «верхние» и «нижние» значения в строках 15 и 16. Строки 17–24 отправляют и получают верхние и нижние значения, на основании того, является ли P_{ID} больше или меньше, чем P_{pair}. Строки 26 и 27 выполняют обновление в локальной части **x**.

```
ParallelRandomAccessTree(x : Z^{P(N)})
r_0, r, i : Z^{1024}
N_up = 4N        N_b = N_up/1024
r_0 = randStart( (N_b/N_P)(1024 P_ID + 0 : 1023) )
tic
for b = 1 : N_b
      do
         r_0 = rand(r_0)        r = r_0
         tag = b  mod 32
         P_mid = N_P/2        P_mod = N_P
         while P_mod > 1
               do
                  P_pair = (P_ID + P_mid)  mod P_mod + ⌊P_ID/P_mod⌋ P_mod
                  P_split = P_mid + ⌊P_ID/P_mod⌋ P_mod
                  i = and(r, N − 1)
                  r_hi = r( i ≥ x.loc_{P_split} < ⟨1⟩ )
                  r_lo = r( i < x.loc_{P_split} < ⟨1⟩ )
                  if P_ID < P_pair
                     do
                        SendMsg(P_pair, tag, r_hi)
                        r = append(r_lo, RecvMsg(P_pair, tag))
                  if P_ID > P_pair
                     do
                        SendMsg(P_pair, tag, r_lo)
                        r = append(r_hi, RecvMsg(P_pair, tag))
                  P_mid = P_mid/2        P_mod = P_mod/2
         i = and(r, N − 1) − x.loc_{P_ID} ⟨1⟩ − 1
         x.loc(i) = xor(x.loc(i), r)
t_up = toc
```

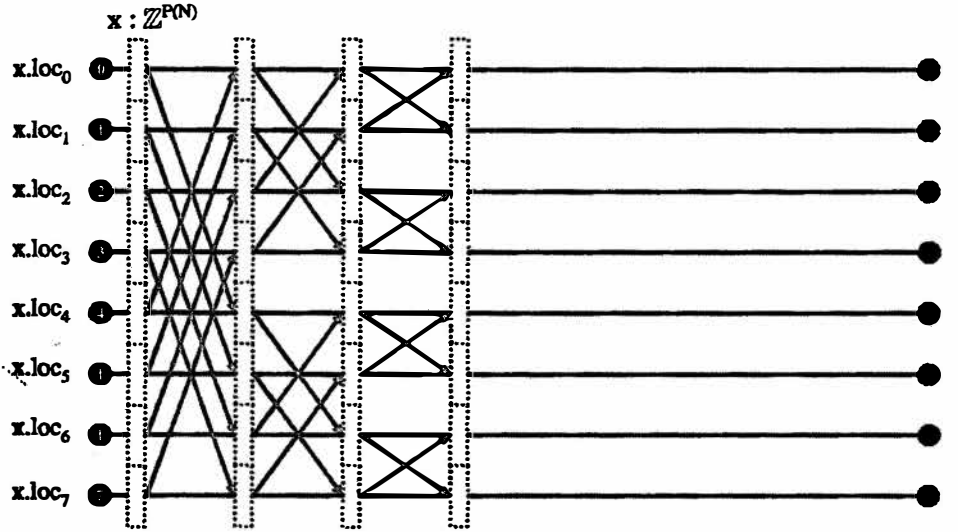


Рисунок 9.2. Шаблон коммуникации древовидного алгоритма RandomAccess
Каждый P_{ID} отправляет свои верхние или нижние половины обновлений одному $P_{ID'}$. Этот процесс повторяется до тех пор, пока обновления не перейдут к соответствующим $P_{ID'}$, в результате чего получаем $Np\lg(Np)$ сообщений с 512 обновлениями на каждое сообщение

9.3. Код

Следующим шагом является написание программы RandomAccess. Реализация состоит из двух основных частей. Первая часть – это управляющая часть кода, которая устанавливает параметры, инициализирует структуры данных и выполняет основной параллельный алгоритм. Вторая часть – это основной параллельный алгоритм, который может являться либо рассевающим алгоритмом, либо древовидным. Управляющая часть параллельного кода показана в листинге кода 9.1 (полный листинг расположен в директории `Examples/RandomAccess/pRandomAccess.m`). Следует понимать, что один и тот же код выполняется каждым экземпляром MATLAB. Единственным различием между экземплярами является наличие у каждого из них уникального P_{ID} (т. е. `Pid`), который используется для определения того, какие части распределенного массива принадлежат каждому из экземпляров MATLAB (т. е. *x.loc*).

Листинг кода 9.1. Параллельная MATLAB-программа RandomAccess

```
PARALLEL = 1;      % Turn distributed arrays on.
VALIDATE = 1;      % Turn verification on.
ErrorRate = 0.01;          % Set allowed error rate.
IgN = 25; N = 2^IgN;       % Size main table X.
```

```
5 IgNb = 10; Nb = 2^IgNb;                    % Size of block.
6 Nup = 4*N; Nblocks = Nup/Nb;               % Number of updates and blocks.
7 mask = uint64(N-1);                        % Mask that selects low bits.
8 Xmap = 1;                                  % Serial map.
9 if PARALLEL
10   Xmap = map([1 NpJ ,{},0:Np-1);          % Map for table.
11 end
12 tic;
13   X = zeros(1,N,Xmap);                    % Allocate main table.
14   Xloc = uint64(global_ind(X,2)-1);       % Init local part.
15 Talloc = toc;
16 myX = global_block_range(X,2);            % Local index range.
17 allX = global_block_ranges(X,2);          % All index ranges.
18 myBLOCK = global_ind(zeros(1,Nblocks,Xmap),2); % Local blocks.
19 ranStarts = RandomAccessStarts((myBLOCK(1)-1)*Nb + …
20   (0:(Nb-1))*length(myBLOCK));            % Init random sequence.
21 tic
22   sync = agg(zeros(1,Np,Xmap));           %Synchronize start
23 Tlaunch = toc;
24 tempVALIDATE = VALIDATE; VALIDATE = 0; % Cache flag.
25 tic;
26   pRandomAccessSpray;                     % Run spray algorithm.
27   %pRandomAccessTree;                     % Run tree algorithm.
28 Trun = toc;
29 VALIDATE = tempVALIDATE;                  % Put flag back.
30 if VALIDATE
31   tic;
32           pRandomAccessSpray;    % Run spray algorithm.
33           %pRandomAccessTree;    % Run tree algorithm.
34           XlocO = uint64(global_ind(X,2)-1); % True values.
35           Nerrors length(find(Xloc = XlocO)); % Compute errors.
36   Tvalidate = toc;
37 end
```

Перевод

1 % Включение распределенных массивов;
2 % Включение проверки;
3 % Установить допустимый уровень ошибок;
4 % Размер основной таблицы X;
5 % Размер блока;
6 % Количество обновлений и блоков;
7 % Маска, которая выбирает нижние биты;
8 % Последовательная карта;
10 % Карта для таблицы;
13 % Разместить основную таблицу;
14 % Иниц. локальную часть;
16 % Диапазон локальных индексов;
17 % Диапазоны всех индексов;

18 % Локальные блоки;
20 % Иниц. случайную последовательность;
22 % Синхронизация начала;
24 % Запомнить флаг;
26 % Выполнить рассеивающий алгоритм;
27 % Выполнить древовидный алгоритм;
29 % Поместить флаг обратно;
32 % Выполнить рассеивающий алгоритм;
33 % Выполнить древовидный алгоритм;
34 % Истинные значения;
35 % Вычислить ошибки.

В вышеприведенной программе

Строка 1 устанавливает флаг PARALLEL, который включает и выключает распределенные массивы.

Строка 2 устанавливает флаг VALIDATE, который включает и выключает проверочную часть программы. Этот флаг присутствует по причине того, что проверка может значительно увеличить время выполнения программы.

Строки 3–7 устанавливают различные параметры программы RandomAccess.

Строки 8–15 создают параллельную карту `Xmap`, размещают распределенную таблицу X и инициализируют локальную часть `Xloc`, так что $X(i)=i-1$. Эта инициализация выполняется с использованием функции `global_ind`, которая возвращает глобальный индекс каждого локального элемента и эквивалентна установке

$$\mathbf{x}.loc(i) = \mathbf{x}.loc(i)-1.$$

Строка 16 использует функцию `global_block_range` для возвращения глобальных индексов первого ($\mathbf{x}.loc(1)$) и последнего ($\mathbf{x}.loc(end)$) элементов в локальной части X.

Строка 17 использует функцию `global_block_ranges` для возвращения массива, содержащего глобальные индексы первого ($\mathbf{x}.loc(1)$) и последнего ($\mathbf{x}.loc(end)$) элементов всех локальных частей массива X для всех P_{ID}. Эти индексы будут использованы для определения, какие обновления подлежат отправке каким P_{ID}.

Строки 18–19 распределяют блоки обновлений между P_{ID}, так что каждый начинает работу с правильной позиции в последовательности случайных чисел.

Строка 22 использует функцию `agg` для синхронизации всех экземпляров P_{ID}, так что они начинают работу приблизительно в одно время. Функция `agg` заставляет каждый P_{ID} отправлять свою часть распределенного массива экземпляру $P_{ID} = 0$. Более агрессивную синхронизацию можно выполнить с использованием функции `agg_all`, которая заставляет каждый P_{ID} отправлять свою часть распределенного массива всем другим P_{ID}.

Строки 25–28 выполняют и рассчитывают время теста производительности RandomAccess, используя либо рассеивающий, либо древовидный алгоритм.

Строки 30–37 повторно выполняют тестовую программу RandomAccess, используя либо рассеивающий, либо древовидный алгоритм с безошибочным механизмом обновления, сравнивают результаты с корректными значениями и подсчитывают количество ошибок.

Вышеуказанный код является хорошим примером того, как среда программирования на основе распределенных массивов может помочь при обмене сообщениями. В частности, такие операции, как `global_block_range` и `global_block_ranges` могут быть очень полезными для определения того, какие данные и куда следует отправить при использовании комбинированной модели программирования.

9.3.1. Рассеивающий код

Реализация рассеивающего алгоритма показана в листинге кода 9.2 (полный листинг расположен в директории `Examples/RandomAccess /pRandomAccessSpray.m`).

Листинг кода 9.2. Рассеивающий алгоритм MATLAB-программы RandomAccess

```
1 ran = ranStarts;                    % Init random sequence.
2 tag = 0;                            % Init message tag.
3 % Set optimal send and receive order.
4 mySendOrder = circshift(
     [0: (Pid-1) (Pid+1): (Np-1)] , [0, -Pid]) ;
5 myRecvOrder = fliplr(
     circshift ( [0: (Pid-1) (Pid+1): (Np-1)], [0, -Pid] )) ;
6 for ib = myBLOCK  % Loop over my blocks.
7    tag = mod (tag+1 , 32) ;        % Increment message tag.
8    ran = RandomAccessRand(ran);  % Next random numbers.
9    Xi = double(bitand(ran,mask))+1;      % Compute global index.
10   for p = mySendOrder                   % Find and send updates.
11           ranSend = ran«Xi>=allX(p+1,2))& (Xi<=allX(p+1,3)));
12           SendMsg(p,tag,ranSend );
13   end
14   ranRecv = ran( (Xi >= myX(1)) & (Xi <= myX(2)) );
15   for p = myRecvOrder                           % Receive updates.
16           ranRecv = [ranRecv RecvMsg(p,tag)];   % Append.
17   end
18   Xi = double(bitand(ranRecv,mask))+1-(myX(1)-1);      % Local Xi.
19   if (not(VALIDATE))                                   % Fast update.
20           Xloc(Xi) = bitxor(Xloc(i),ranRecv);
21   else                                  % Slow error-free update.
22           for j=1:length(ranRecv)
23                   Xloc(Xi(j)) = bitxor(Xloc(Xi(j)), ranRecv(j));
24            end
25   end
26 end
```

Перевод

1 % Инициализировать случайную последовательность;

2 % Инициализировать тэг сообщения;

3 % Установить оптимальный порядок отправки и получения;
6 % Цикл по «моим» блокам;
7 % Увеличить тэг сообщения;
8 % Следующие случайные числа;
9 % Вычислить глобальный индекс;
10 % Найти и отправить обновления;
15 % Получить обновления;
16 % Добавить;
19 % Быстрое обновление;
21 % Медленное обновление без ошибок.

В данной программе

Строки 4 и **5** вычисляют порядок отправки сообщений и порядок получения, так что каждый P_{ID} отправляет данные в различном направлении в соответствии с рисунком 9.1.

Строки 10–13 выполняют цикл по каждому P_{ID} получателя и определяют, какие обновления отправлять этому P_{ID}, используя информацию в `allX`.

Строка 14 определяет обновления, подлежащие сохранению на локальном P_{ID}, используя информацию в `myX`.

Строки 15–17 выполняют цикл по каждому P_{ID} отправителя, получают обновления от этого P_{ID} и добавляют их к `ranRecv`.

Строка 18 вычисляет глобальные индексы полученных обновлений и преобразует их в свои соответствующие локальные индексы в `Xloc` путем вычитания глобального индекса первого элемента `myX(1)`.

Строки 19–26 выполняют обновления к `Xloc`, используя либо схему быстрого обновления, которая может совершать некоторое количество ошибок, либо схему медленного обновления, которая не содержит ошибок.

Вышеприведенный код показывает, как использовать результат команд `global_block_range` и `global_block_ranges` для выбора данных для отправки процессорам. Эта функция является весьма распространенной в программах с обменом сообщениями. В чистых программах с обменом сообщениями вся информация в `allX` и `myX` требует расчетов вручную на основе N и Np. Такой код обычно сложен, и высока вероятность ошибки. Использование функции поддержки распределенных массивов позволяет этой информации рассчитываться автоматически и без ошибок, обеспечивая при этом контроль шаблона сообщений.

9.3.2. Древовидный код

Реализация древовидного кода приведена в листинге кода 9.3 (полный листинг расположен в директории `Examples/RandomAccess/pRandomAccessTree.m`). Древовидный код во многих аспектах является отражением рассеивающего кода. Основные различия заключаются в следующем:

Строка 3 раскладывает Np на список простых сомножителей, которые затем используются в каждой итерации древовидного алгоритма.

Строка 9 инициализирует несколько значений, используемых для определения того, где разбивать данные на «верхние» и «нижние» наборы обновлений.

Строка 10 выполняет цикл, пока не выполнится обход дерева.

Строка 14 вычисляет соответствующий парный P_{ID}, который будет обмениваться информацией.

Строки 17 и **18** разбивают обновления на «верхние» и «нижние» наборы.

Строки 19–25 отправляют верхний/нижний набор обновлений парному P_{ID} и затем получают и добавляют верхний/нижний набор обновлений.

Так же как и рассеивающий код, древовидный код показывает, каким образом использовать результат `global_block_range` и `global_block_ranges` для выбора данных для отправки различным экземплярам P_{ID}.

Листинг кода 9.3. Древовидный алгоритм MATLAB-программы RandomAccess

```
1 ran = ranStarts;                    % Init random sequence.
2 tag = 0;                            % Init message tag.
3 tree = factor(Np);                  % Factor processors into a tree.
4 for ib = myBLOCK                    % Loop over all update blocks.
5    tag = mod(tag+1,32);   % Increment message tag.
6    ran = RandomAccessRand(ran); % Next random numbers.
7    ranRecv = ran;         % Init input.
8    % Initialize values for computing splits.
9    midP = Np ./ tree(1); modP = Np; k = 0;
10   while (modP > 1)
11           % Compute PID to exchange info with.
12           pairPid = mod((Pid+midP),modP)+floor(Pid./modP).*modP;
13           % Compute PID for splitting data.
14           splitPid = midP + floor(Pid./modP) .*modP;
15           Xi = double(bitand(ranRecv,mask))+1;  % Global index.
16           % Find values above and below split.
17           hi = ranRecv(Xi >= allX(splitPid+1,2));
18           lo = ranRecv(Xi < allX(splitPid+1,2));
19           if (Pid < pairPid)            % Exchange hi/lo  data
20                  SendMsg(pairPid,tag,hi);
21                  ranRecv = [lo RecvMsg(pairPid,tag)];
22            elseif (Pid > pairPid)
23                  SendMsg(pairPid,tag,lo);
24                  ranRecv = [hi RecvMsg(pairPid,tag)];
25           end
26           % Update values for computing splits.
27           midP = midP./tree(k+1); modP = modP./tree(k+1); k=k+1;
28   end
29   Xi = double(bitand(ranRecv,mask))+1-(myX(1)-1);      % Local Xi.
30   if (not(VALIDATE))                                   % Fast update.
31           Xloc(i) = bitxor(Xloc(i),ranRecv);
32   else                                  % Slow error free update.
33   for j=1:length(ranRecv)
```

```
34          Xloc(Xi(j)) = bitxor(Xloc(Xi(j)), ranRecv(j));
35    end
36 end
```

Перевод

1 % Инициализировать случайную последовательность;
2 % Инициализировать тэг сообщения;
3 % Разложить процессоры в дерево;
4 % Цикл по всем блокам обновления;
5 % Увеличить тэг сообщения;
6 % Следующие случайные числа;
7 % Инициировать вход;
8 % Инициализировать значения для вычисления разбивок;
11 % Вычислить P_{ID} для обмена информацией с ним;
13 % Вычислить P_{ID} для разбиения данных;
15 % Глобальный индекс;
16 % Найти значения выше и ниже разбивки;
19 % Обмен верхними/нижними данными;
26 % Обновить значения для вычисления разбивок;
30 % Быстрое обновление;
32 % Медленное обновление без ошибок.

9.3.3. Обзор кода

Как рассеивающая, так и древовидная реализация является хорошим примером того, как комбинированную программу на основе распределенных массивов и обмена сообщениями можно упростить с использованием программного подхода на основе карт размещения. После создания карты `Xmap` она используется для размещения распределенного массива X и инициализации его локальной части `Xloc`.

На данном примере кода можно проследить некоторые свойства подхода к программированию на основе карт размещения.

- Влияние на код. Преобразование программы из последовательной в параллельную с использованием комбинированного метода приводит к составлению кода, значительно большего, чем последовательный код. Однако комбинированный код имеет значительно меньшую величину, чем при чистом обмене сообщениями, так как вся информация об индексах массивов предоставляется распределенными массивами.
- Последовательный код сохраняется. При помощи установки PARALLEL = 0 исключается использование распределенных массивов, и программа может быть выполнена последовательно без поддержки параллельных библиотек.
- Средний объем использования параллельной библиотеки. В дополнение к обычным функциям распределенных массивов *Np*, P_{ID} и `agg` программа также использует функции индексов распределенных массивов `global_block_range` и `global_block_range1`, а также основные функции обмена сообщениями `SendMsg` и `RecvMsg`. В общем, используется восемь функций парал-

лельной библиотеки, что больше того, что было использовано в программе на основе чистых распределенных массивов.

- Масштабируемость. Код может выполняться на задачах различного размера, такого, что N является степенью двойки, и любом количестве процессоров Np.
- Ограниченная коммуникация. Так как используются локальные переменные и прямой обмен сообщениями, использование коммуникации строго контролируется. Кроме того, комбинированный подход допускает шаблоны коммуникации, гораздо более сложные, чем можно поддерживать при использовании только распределенных массивов.
- Зависимость от карт размещения. Побочным эффектом использования комбинированного подхода является тот факт, что код будет выполняться только с использованием карт блочного размещения.
- Гарантия последовательной производительности. Так как операции обновления применяются к локальным переменным, которые являются обычными числовыми массивами MATLAB, гарантируется отсутствие каких-либо скрытых ухудшений производительности при выполнении данных строк кода.

9.4. Отладка

Как упоминалось ранее, параллельная отладка представляет собой непростую задачу. Четко определенный процесс кодирования–отладки–тестирования является основным преимуществом подхода на основе карт размещения (см. таблицу 2.2). Метод комбинированного составления кода требует модификации данного процесса.

Шаг 1. Выполнение программы последовательно на локальной машине с таблицей **X**, созданной как обычный массив (т. е. PARALLEL=0). Таким способом проверяется корректность исходного последовательного кода. Для работы этого последовательного кода необходимо предпринять ряд дополнительных шагов по сравнению с кодом, использующим чистые распределенные массивы. Во-первых, все функции индексации массивов должны возвращать правильные результаты для нераспределенных массивов. Во-вторых, код обмена сообщениями написан так, что для $Np = 1$ он выполняться не будет. В листинге кода 9.3 переменные `mySendOrder` и `myRecvOrder` создаются так, что они останутся пустыми, если $Np = 1$, таким образом, соответствующие им циклы For не будут выполняться. Точно так же, в листинге кода 9.3 условные выражения P_{ID}<**pair**P_{ID} и P_{ID}>**pair**P_{ID} предотвращают отправку сообщений, когда P_{ID}=**pair**P_{ID}, что случается при $Np = 1$.

Шаг 2. Выполняет программу последовательно на локальной машине с `X` в качестве распределенного массива (т. е. PARALLEL=1) и проверяет правильную работу программы с распределенными массивами в качестве входных данных. Как и в шаге 1, код должен быть написан так, чтобы при $Np = 1$ обмен сообщениями не осуществлялся.

Шаг 3. Выполнение программы параллельно на локальной машине. Данный шаг определяет, являются ли параллельные карты, которые генерируют `X` и `Xloc`, корректными. Что еще более важно, на этом шаге впервые исследуется код обмена сообщениями. Из всех шагов этот является самым важным, так как обмен сообщениями может легко способствовать появлению множества ошибок. Самой распространенной ошибкой является неправильное вычисление экземпляром корректного P_{ID} для отправки или со-

ответствующего P_{ID} для получения сообщения. Такая ошибка обычно приводит к «зависанию» программы, так как P_{ID} ждет сообщения, которое ему не было отправлено. Другой распространенной ошибкой является неверное содержимое сообщения. Использование функций индексации распределенных массивов применительно к X для определения содержимого сообщений значительно снижает количество ошибок этого класса.

Шаг 4. Выполнение программы параллельно на удаленной машине.

Шаг 5. Выполнение программы параллельно на удаленной машине, но с задачей интересующего размера. До этого момента тестирование должно было осуществляться на задаче малого размера. Данный шаг позволяет определить производительность на реальной задаче.

Указанные шаги являются существенным отражением этапов, описанных в разделе «Начало работы», и на каждом из шагов тестируются определенные аспекты параллельного кода. В дополнение к отладке функциональности программы вышеуказанный процесс также может быть использован для оптимизации производительности кода (см. следующий раздел). Тест производительности RandomAccess использует некоторые дополнительные методы кодирования, помогающие в отладке, помимо указанных выше шагов.

Программа RandomAccess печатает параметры, которые описывают размер задачи *N*, количество необходимых байтов и количество совершенных обновлений. Это осуществляется на каждом P_{ID} программы. Сравнение этих значений путем просмотра файлов с выводом от каждого P_{ID} (т. е. `MatMPI/.*out`) является действенным способом выявления ошибок.

Программа RandomAccess также отображает время размещения и запуска для каждого P_{ID}. Эти временные показатели описывают количество времени, затрачиваемое до начала выполнения программы. Признаком того, что что-то происходит не так на ранних стадиях, является значительное различие между этими значениями для разных P_{ID}.

Так как RandomAccess является тестом производительности, производительность выполнения измеряется и регистрируется в единицах, имеющих отношение к процессору. Значения производительности, значительно отличающиеся между P_{ID}, могут стать показателем более глубоких проблем в программе (например, создание массивов слишком отличных друг от друга размеров).

Наконец, программа RandomAccess имеет проверку корректности, которая проверяет выдачу программой корректных результатов. В реальных приложениях такую проверку, как правило, произвести невозможно, однако, можно выполнить приближенные проверки, которые бывают очень полезны. Например, часто можно проверить знак значений в массиве или сумму значений. Повторимся, что отображение этих значений для каждого P_{ID} может выявить ошибки, зависящие от P_{ID}.

9.5. Тестирование

Как станет очевидно далее, программа RandomAccess является хорошим примером программы, производительность которой не повышается с ростом *Np*. Фактически, программа RandomAccess предназначена для демонстрации ограничений коммуникации в широком диапазоне видов вычислительных архитектур. По этим причинам программу RandomAccess стоит изучать, так как она демонстрирует вычисления, которые

принципиально сложно выполнить эффективно в параллельном режиме. Кроме того, программа RandomAccess является очень хорошей программой для исследования производительности различных алгоритмов передачи сообщений, в частности нахождения компромисса между размером сообщения и количеством сообщений.

9.5.1. Производительность на многоядерной системе

Тестирование параллельной программы – при условии, что она уже отлажена и выдает корректные ответы - в основном заключается в определении количества времени, которое требуется для ее выполнения, и выяснения, насколько ее производительность соответствует ожидаемой. Почти все данные о производительности, необходимые для проведения такого анализа, были получены в разделе «Начало работы» (см. Таблицу 9.1). Кроме того, в таблице 9.1 показана многоядерная производительность на восьмиядерной системе для рассеивающего и древовидного алгоритмов

Таблица 9.1. Производительность программы RandomAccess на многоядерной системе

Относительное время размещения, абсолютное время запуска, относительное время выполнения и относительная производительность для теста производительности RandomAccess с использованием рассеивающего и древовидного алгоритмов на 8-ядерном процессоре. Все значения нормализованы по столбцу 2, в котором $N_p = 1$ с выключенными распределенными массивами (PARALLEL=0). Столбцы 3–6 содержат значения с включенными распределенными массивами (PARALLEL=1). Запись 1*8 означает, что $Np = 8$ и все экземпляры P_{ID} были выполнены на одном процессорном узле при совместно используемой памяти.

Многоядерные процессоры:	1	1*1	1*2	1*4	1*8
Рассеивающий алгоритм					
Относительное время размещения	1	1.0	0.88	0.58	0.41
Время запуска (с)	0	0	1.28	1.34	1.66
Относительное время выполнения	1	1.0	11	24	48
Относительн. произв. (GUPS)	1	1.0	1/11	1/24	1/48
Древовидный алгоритм					
Относительное время размещения	1	1.0	0.57	0.52	0.30
Время запуска (с)	0	0	1.75	1.94	1.65
Относительное время выполнения	1	1.0	10	17	17
Относительн. произв. (GUPS)	1	1.0	1/10	1/17	1/17

Первый тест проводится для сравнения производительности выполнения программы с использованием $Np=1$ с выключенными распределенными массивами (PARALLEL = 0) и с включенными распределенными массивами (PARALLEL = 1). Эти данные представлены в столбцах 2 и 3 таблицы . В данном случае разница производительности вычислений пренебрежимо мала и означает, что распределенные массивы не снижают ее. Если использование распределенных массивов с $Np = 1$ значительно замедляет работу, необходимо решить эту проблему. Очень трудно получить какое-либо преимущество при $Np > 1$, если при $Np = 1$ операции выполняются медленнее, чем в исходном последовательном коде. Работа только на локальных частях рас-

пределенных массивов практически гарантирует, что производительность будет такой же, как при работе без использования распределенных массивов.

В столбцах 3–5 таблицы 9.1 приводится параллельная производительность кода на многоядерной системе. Строки относительного времени размещения показывают, что размещение обычно уменьшается по времени с увеличением *Np*, так как каждому P_{ID} необходимо разместить *N*/*Np* элементов данных. Измеренное уменьшение времени размещения не изменяется линейно с *Np*, так как с размещением связаны некоторые накладные расходы. Обычно этот период времени не занимает большую часть общего времени вычисления. Однако при большом числе *Np* эти расходы могут стать значительными; таким образом, рекомендуется измерять время размещения и убеждаться в его приемлемости.

Строка времени запуска демонстрирует общее увеличение времени запуска с ростом *Np*. В идеале это должна быть постоянная величина, не зависящая от *Np*. Однако в зависимости от используемого механизма запуска этот показатель времени может являться увеличивающейся функцией от *Np*. Как правило время запуска не составляет большую часть от общего времени вычисления, но оно может легко стать значительным при больших значениях *Np*. Например, предположим, что время запуска пропорционально *Np*, и каждый дополнительный P_{ID} добавляет одну секунду. Если *Np* = 100, время запуска составит 100 секунд. Если это доля Амдала программы, то программе понадобится для выполнения не менее N_P^2 секунд для достижения 50-кратного ускорения по сравнению с последовательным кодом.

Наиболее интересные строки – это относительное время выполнения и относительная производительность, выраженная в GUPS. Так как *N* и количество совершенных обновлений одинаковы в этих случаях, эти две строки являются просто обратными друг другу; т. е. относительная производительность обратно пропорциональна относительному времени выполнения. Первое, что нужно отметить, это большое падение производительности при переходе от *Np* = 1 к *Np* = 2. Это типично для теста производительности RandomAccess, когда он выполняется параллельно с использованием системы обмена сообщениями. Помните, что программа RandomAccess отправляет сообщения малого размера, и значение времени на отправку данных сообщений может легко превзойти значение времени вычисления. Таким образом, программа RandomAccess является хорошим примером класса приложений, которые трудно ускорить на параллельной системе и которые нуждаются для этого в специальном аппаратном обеспечении коммуникации.

Хотя тест производительности RandomAccess не масштабируется эффективно, его производительность можно улучшить. В частности, рассеивающий алгоритм демонстрирует производительность, которая линейно уменьшается с *Np*. Древовидный алгоритм, который отправляет меньшее количество сообщений большего размера, не замедляется так значительно. Фактически, для *Np* = 8 древовидный алгоритм обеспечивает почти в 2,8 раза более высокую производительность, чем рассеивающий алгоритм. Вспомните, что предсказанное повышение относительной производительности древовидного алгоритма составило *Np*/lg(*Np*) = 2.7.

9.5.2. Производительность на многоузловой системе

Результаты выполнения рассеивающего и древовидного алгоритмов на многоузловой системе приведены в таблице 9.2.

Относительное время размещения для рассеивающего и древовидного алгоритмов имеет близкие значения и уменьшается с увеличением числа *Np*. Снижение времени размещения более значительно и постоянно, чем в многоядерной системе. В случае многоядерной системы размещение осуществляется из той же памяти, и конфликт этого ресурса может увеличить это время и привести к большему отклонению. Распределение на многоузловой системе полностью независимо, происходит быстрее и более детерминировано, чем на одном узле.

Таблица 9.2. Производительность программы RandomAccess на многоузловой системе

Относительное время размещения, абсолютное время запуска, относительное время выполнения и относительная производительность для теста RandomAccess с использованием рассеивающего и древовидного алгоритмов на 16-узловой системе. Все значения нормализованы по столбцу 2, в котором *Np* = 1 с выключенными распределенными массивами (PARALLEL = 0). Столбцы 3–7 содержат значения с включенными распределенными массивами (PARALLEL = 1). Запись 16*1 означает, что *Np* = 16 и каждый P_{ID} был выполнен на своем индивидуальном процессорном узле с использованием отдельной памяти

Многоузловые процессоры:	1	1*1	2*1	4*1	8*1	16*1
Рассеивающий алгоритм						
Относительное время размещения	1	1.0	0.79	0.43	0.25	0.16
Время запуска (с)	0	0	7.89	11.8	9.9	12.3
Относительное время выполнения	1	1.0	14	16	18	19
Относительн. произв. (GUPS)	1	1.0	1/14	1/16	1/18	1/19
Древовидный алгоритм						
Относительное время размещения	1	1.0	0.79	0.42	0.25	0.17
Время запуска (с)	0	0	7.77	11.8	9.85	9.91
Относительное время выполнения	1	1.0	12	11	8.6	5.5
Относительн. произв. (GUPS)	1	1.0	1/12	1/11	1/8.6	1/5.5

Время запуска на многоузловой системе склонно к увеличению с ростом *Np* и демонстрирует большие отклонения. Более того, время запуска гораздо больше, чем в случае многоядерной системы, так как отдельные системы для запуска своего экземпляра P_{ID} должны связываться по сети.

Относительное время выполнения и относительная производительность для рассеивающего и древовидного алгоритмов не показывают улучшения для *Np* > 1. Как и в многоядерном варианте, это низкое значение является фундаментальным для теста производительности RandomAccess. Однако производительность рассеивающего и древовидного алгоритмов в многоузловой системе несколько лучше, чем в многоядерной системе. Причина этого – состязание за ресурсы системы коммуникации в многоядерной системе, а в многоузловом варианте системы коммуникации суще-

9.3. Производительность можно разбить на три категории. Рассеивающий алгоритм на многоядерной системе демонстрирует стабильное снижение производительности с ростом *Np*, и, вероятнее всего, этот метод не даст какого-либо улучшения производительности. Древовидный алгоритм в многоядерной системе и рассеивающий алгоритм в многоузловой системе демонстрируют неизменную производительность в зависимости от числа *Np*. Эта производительность лучше, чем в случае рассеивающего алгоритма в многоядерной системе, но также маловероятно, что прирост N_p даст какое-либо улучшение производительности кроме того, что на многоузловой системе можно выполнять задачу с большими значениями *N*, чем на одноузловой системе. Наконец, древовидный алгоритм в многоузловой системе демонстрирует увеличение производительности с ростом *Np*, и для достаточного значения *Np* он будет работать быстрее, чем при *Np* = 1. Эти факторы объясняют, почему все конкурирующие реализации теста производительности RandomAccess используют вариации древовидного алгоритма и выполняются на очень больших многоузловых системах.

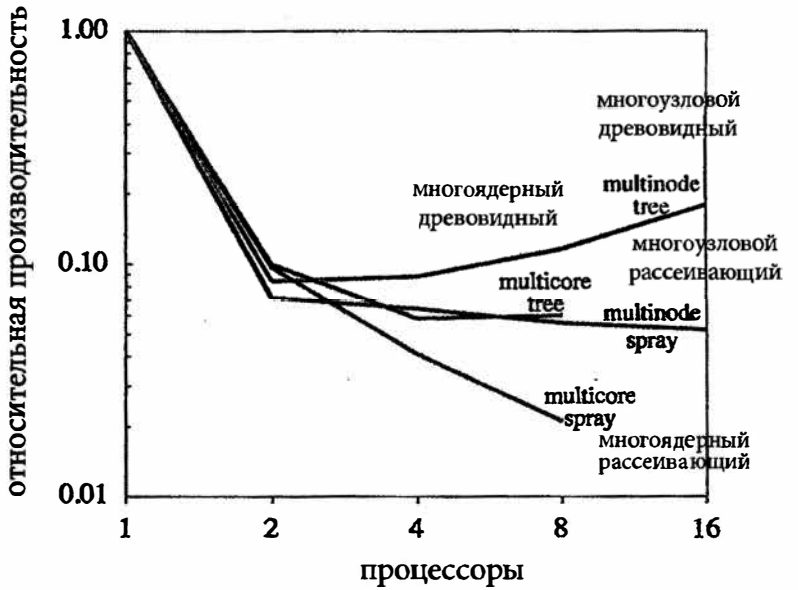


Рисунок 9.3. Относительная производительность параллельной программы RandomAccess

Относительная производительность рассеивающего и древовидного алгоритмов на многоядерных и многоузловых системах.

Собрав и изучив все данные о производительности RandomAccess, можно сделать выводы о выполнении приложений такого типа на кластерах многоядерных процессорных узлов. Во-первых, программа RandomAccess принципиально ограничена задержкой используемого оборудования и при *Np* > 1 будет иметь тенденцию работать медленнее, чем при *Np* = 1. Во-вторых, выполнение множества P_{ID} (1**Np*) на одном процессорном узле может не принести большой выгоды. Таким образом, «вертикальное мас-

штабирование» не может являться эффективным методом. В-третьих, производительность может увеличиваться и в конечном итоге превзойти случай $Np = 1$ при использовании древовидного алгоритма на множестве узлов (Np*1). Таким образом, «горизонтальное масштабирование» оказывается наиболее оптимальным и выгодным методом.

Список литературы

1. *[Dongarra and Luszczek 2005] Jack Dongarra and Piotr Luszczek, Introduction to the HPC Challenge Benchmark Suite, Technical Report UT-CS-05-544, University of Tennessee, 2005.*

Глава 10
Программа FFT

Резюме

Быстрое преобразование Фурье (БПФ) является важным видом вычислений для широкого диапазона приложений – от обработки сигналов до спектральных методов решения дифференциальных уравнений в частных производных. Сложная с вычислительной точки зрения природа БПФ сделала его основой для тестов производительности на десятилетия. Тест производительности FFT представляет собой простую программу, которая создает нагрузку как на пропускную способность локальной памяти, так и на пропускную способность сети параллельной вычислительной машины. Это типичный представитель класса задач, которые могут эффективно выполняться на стандартных параллельных машинах при условии достаточной пропускной способности сети. Шаблон коммуникации этого теста производительности сложен, но хорошо согласуется с методом распределенных массивов и может быть реализован коротко и эффективно.

Программа FFT была представлена в главе 7 в качестве теста производительности для измерения пропускной способности между процессорами в сети [Dongarra and Luszczek 2005]. Программа FFT является хорошим примером параллельного приложения с большим объемом коммуникации со сложными шаблонами и позволяет продемонстрировать ряд полезных методов параллельного программирования, в частности способ написания программы на основе распределенных массивов, который кратко и эффективно реализует эти сложные шаблоны.

Тест производительности FFT создает комплексный вектор $\mathbf{x} : C^N$ и выполняет алгоритм БПФ на этом векторе с целью получения другого комплексного вектора $\mathbf{z} : C^N$. БПФ является эффективным способом реализовать дискретное преобразование Фурье (ДПФ), которое задается формулой:

$$z(k) = \sum_{j=1}^{N} x(j) \exp[-2\pi\sqrt{-1}(k-1)(j-1)/N].$$

ДПФ «тестирует» вектор x на его «близость» к волне «k», умножая $\mathbf{x}$ на волну, заданную выражением $\exp[-2\pi\sqrt{-1}(k-1)(j-1)/N]$. Полученное произведение суммируется, и значение сохраняется в $\mathbf{z}(k)$. Как было указано, ДПФ выполняет $5N^2$ операций. Быстрое преобразование Фурье ускоряет ДПФ, распознавая те произведения, которые могут быть повторно использованы для расчета различных значений $\mathbf{z}$, уменьшая общее количество операций до $5N\lg(N)$.

Весьма значительное сокращение числа операций, обеспечиваемое БПФ, достигается ценой создания сложного шаблона перемещения данных. Последовательный алгоритм БПФ обычно ограничен пропускной способностью памяти. Аналогично параллельный алгоритм БПФ ограничен суммарной пропускной способностью или «разделенной надвое» (bisection) пропускной способностью коммуникационной сети. Программа FFT прекрасно иллюстрирует целый ряд шаблонов сложной коммуникации, которые могут эффективно работать на стандартных параллельных машинах

при условии достаточной пропускной способности сети. Эта глава описывает методы кодирования и оптимизации для эффективной реализации алгоритма БПФ.

10.1. Начало работы

Для выполнения программы запустите MATLAB и перейдите в директорию Examples/FFT (т. е. введите `cd Examples/FFT`). Отредактируйте файл `pFFT.m` и установите PARALLEL=0. В командной строке наберите

```
pFFT
```

Так как FFT является тестом производительности, он выдаст большой объем данных о производительности, аналогичных следующим:

```
Np                                = 1
Pid                               = 0
Distributed vector size           = 33554432 words
Distributed vector size (bytes)   = 536870912
Local vector size (bytes)         = 536870912
Allocation Time (sec)             = 26.3669
Launch Time (sec)                 = 0.025377
Compute time (sec)                = 16.0619
Communication time (sec)          = 0.035599
Run time (sec)                    = 16.0975
Performance (Gigaflops)           = 0.26056
Bandwidth (Gigabytes/sec)         = 45.2432
Max local error                   = 3.5046e-17
Max error                         = 3.5046e-17
Validation Passed
```

Перевод
Размер распределенного вектора;
Размер распределенного вектора (байт);
Размер локального вектора (байт);
Время распределения (с);
Время запуска (с);
Время вычисления (с);
Время коммуникации (с);
Время выполнения (с);
Производительность (гигафлопс);
Пропускная способность (ГБ/с);
Максимальная локальная ошибка;
Максимальная ошибка;
Проверка пройдена.

Эти данные можно разбить на четыре категории. Первая – параметры программы, которые описывают задачу: Np, P_{ID}, размер задачи N и количество необходимых байтов. Вторая категория включает в себя время инициализации массива и время запуска, описывающие количество времени, затрачиваемое до начала выполнения про-

граммы. Третья категория содержит время вычисления, коммуникации и выполнения, а также итоговую производительность, полученную на основе этих показателей времени. Четвертая категория – значения прохождения проверки. Эти результаты будут рассмотрены далее в разделе «Тестирование». Сейчас отредактируйте файл pFFT.m, установите параметр PARALLEL=1 (чтобы включить параллельные части кода) и VALIDATE=0 (для сокращения времени выполнения). Повторно выполните pFFT с использованием следующих команд:

```
eval(pRUN('pFFT' ,1,{}))
eval(pRUN('pFFT' ,2,{}))
eval(pRUN('pFFT' ,4,{}))
```

Данные команды выполняют программу FFT при $Np = 1, 2$ и 4. При выполнении на параллельной вычислительной машине выдаются значения производительности, подобные указанным в таблице 10.1 (см. раздел «Тестирование»).

Важное примечание. Последний аргумент команды pRUN определяет, на какой параллельной вычислительной машине будет выполняться программа. Может возникнуть необходимость в получении надлежащих установочных параметров у вашего системного администратора; для получения более подробной информации обращайтесь на веб-сайт книги (http://www.siam.org/KepnerBook). Установка последнего аргумента значением {} приводит к выполнению множества копий программы на вашей локальной машине и получению производительности, отличной от указанной в таблице 10.1.

10.2. Параллельная реализация

Спецификация параллельного теста производительности FFT характеризуется свойством, которое предполагает вычисление БПФ невысокой размерности с использованием БПФ более высоких размерностей. Например, БПФ одномерного вектора можно вычислить, используя БПФ двумерной матрицы. Аналогично БПФ матрицы состоит из выполнения БПФ на строках, за которым следует выполнение БПФ на столбцах (или наоборот). Таким образом, двумерное БПФ матрицы размером $Np \times M$ может быть вычислено следующим образом:

$$\begin{aligned}
&\mathbf{X} : \mathbb{C}^{N_P \times M} \\
&\textbf{for } i = 1 : N_P \\
&\qquad \textbf{do} \quad \mathbf{X}(i,:) = \mathrm{FFT}(\mathbf{X}(i,:)) \\
&\textbf{for } j = 1 : M \\
&\qquad \textbf{do} \quad \mathbf{X}(:,j) = \mathrm{FFT}(\mathbf{X}(:,j))
\end{aligned}$$

Параллельное двумерное БПФ может быть реализовано довольно просто с использованием распределенных массивов:

$$\begin{aligned}
&\mathbf{X} : \mathbb{C}^{P(N_P) \times M},\ \mathbf{Z} : \mathbb{C}^{N_P \times P(M)} \\
&\mathbf{X}.loc(1,:) = \mathrm{FFT}(\mathbf{X}.loc(1,:)) \\
&\mathbf{Z} = \mathbf{X} \\
&\textbf{for } j = 1 : size(\mathbf{Z}.loc(1,:)) \\
&\qquad \textbf{do} \quad \mathbf{Z}.loc(:,j) = \mathrm{FFT}(\mathbf{Z}.loc(:,j))
\end{aligned}$$

В вышеуказанном фрагменте алгоритма входными данными является матрица **X**, распределенная во втором измерении, а результатом – распределенная матрица *Z*, распределенная в первом измерении. БПФ строк и столбцов являются здесь локальными операциями и могут использовать уже реализованное последовательное одномерное БПФ. Преобразование данных между этими двумя распределениями выполняется через оператор присваивания: **Z** = **X**.

Алгоритм параллельного одномерного БПФ устроен по аналогии с алгоритмом параллельного двумерного БПФ с незначительными изменениями. Во-первых, необходимо изменить порядок входных данных и результата. Во-вторых, результаты БПФ строк должны быть умножены на набор «поворачивающих» множителей до применения БПФ к столбцам. Алгоритм параллельного одномерного БПФ показан в Алгоритме 10.1 (подробнее в подписи). **Примечание.** Ограничением для этого конкретного алгоритма параллельного одномерного БПФ является то, что *N* должно делиться на N_p^2 без остатка.

Алгоритм 10.1. Параллельный алгоритм FFT

Вычисление БПФ *N*-элементного комплекснозначного вектора с использованием параллельного двумерного БПФ. Строка 1 вычисляет вторую размерность соответствующей матрицы. Строка 2 размещает две распределенные матрицы. Строка 3 вычисляет локальные поворачивающие множители на основе P_{ID}. Строка 4 копирует локальную часть вектора в локальную часть распределенной матрицы. Строка 5 перераспределяет матрицу. Строка 6 выполняет БПФ для строк и умножает их на поворачивающие множители. Строка 7 перераспределяет данные. Строки 8 и 9 выполняют БПФ для столбцов. Строка 10 перераспределяет данные, строка 11 копирует локальную часть матрицы в локальную часть вектора.

$\mathbf{z} : \mathbb{C}^{P(N)} = \textsc{ParallelFFT1D}(\mathbf{x} : \mathbb{C}^{P(N)})$

1 $M = N/N_P$
2 $\mathbf{X} : \mathbb{C}^{P(N_P)\times M}$, $\mathbf{Z} : \mathbb{C}^{N_P \times P(M)}$
3 $\mathbf{w} = \exp[-2\pi\sqrt{-1}P_{ID}/N\ (1:M)]$
4 $\mathbf{Z}.loc = \mathbf{x}.loc$
5 $\mathbf{X} = \mathbf{Z}$
6 $\mathbf{X}.loc(1,:) = \mathbf{w}\ .*\ \mathrm{FFT}(\mathbf{X}.loc(1,:))$
7 $\mathbf{Z} = \mathbf{X}$
8 **for** $j = 1 : size(\mathbf{Z}.loc(1,:))$
9 **do** $\mathbf{Z}.loc(:,j) = \mathrm{FFT}(\mathbf{Z}.loc(:,j))$
10 $\mathbf{X} = \mathbf{Z}$
11 $\mathbf{z}.loc = \mathbf{X}.loc$

Параллельный алгоритм FFT является хорошим примером алгоритма со сложными перемещениями данных, которые подходят для использования распределенных массивов. Такой шаблон перемещения данных состоит из пяти этапов, показанных на рисунке 10.1. Первый этап – входные данные преобразовываются в матрицу копированием локальной части распределенного вектора $\boldsymbol{x} : \mathbb{C}^{P(N)}$ в локальную часть распределенной матрицы $\mathbf{Z} : \mathbb{C}^{N_P \times P(M)}$. Второй этап – данные перераспределяются по эк-

земплярам P_{ID} при помощи присвоения **z** распределенной матрице **X** : $C^{P(Np)\times M}$. Теперь каждая строка становится локальной для P_{ID}, и БПФ для строк может выполняться с использованием вызова последовательного БПФ. Третий этап – данные перераспределяются обратно в **z** так, чтобы каждый столбец был локальным для P_{ID}, и можно выполнить преобразование БПФ для столбцов. Четвертый этап – данные перераспределяются в третий раз в **X**. Пятый этап – данные меняют размерность копированием локальной части **X** в локальную часть распределенного вектора **z** : $C^{P(N)}$.

Целью теста производительности FFT является измерение возможности машины выполнить большое БПФ. Производительность измеряется в гигафлопсах и вычисляется по формуле:

$$\text{Gigaflops:} \quad 5N\lg(N)/t_{run}/10^9,$$

где $t_{run} = t_{comp} + t_{comm}$ – это сумма времени вычислений и коммуникации.
t_{comp} – вычисляется измерением времени выполнения строк 6, 8, 9 в Алгоритме 10.1.
t_{comm} вычисляется измерением времени выполнения строк 5, 7, 10 в Алгоритме 10.1.
Параллельная программа FFT требует большого объема коммуникации. Фактически весь вектор перераспределяется три раза по ходу алгоритма. Таким образом, FFT является мерой общей пропускной способности системы, которая измеряется в гигабайтах/с по формуле:

$$\text{Gigabytes/sec:} \quad 48N/t_{comm}/10^9.$$

Проверка правильности является важной частью любого теста производительности. Если значения вектора x инициализируются волной с целочисленной частотой k,

$$x = \exp[-2\pi\sqrt{-1}k / N(1:M)],$$

то БПФ для **x** может приближенно вычисляться в виде: $\mathbf{z}(k) = N$ и нули в остальных элементах. Ожидаемый уровень ошибок будет функцией машинной точности и точности аппроксимации **z**, и может быть вычислен по формуле

$$err = \max(|z - \bar{z}|) / N^{\frac{3}{2}}.$$

Коэффициент $N^{3/2}$ состоит из двух сомножителей: первый коэффициент N – для масштабирования результата к относительной ошибке, так как N – максимальное значение; второй коэффициент $N^{1/2}$ – для учета ошибки аппроксимации к истинному результату, так как БПФ для синусоиды будет иметь очень небольшие пиковые значения вокруг центрального пикового значения (иногда называются побочными максимумами), которое приблизительно равно $N^{1/2}$. Для стандартной машинной точности ошибка $err < 10^{-10}$ приемлема.

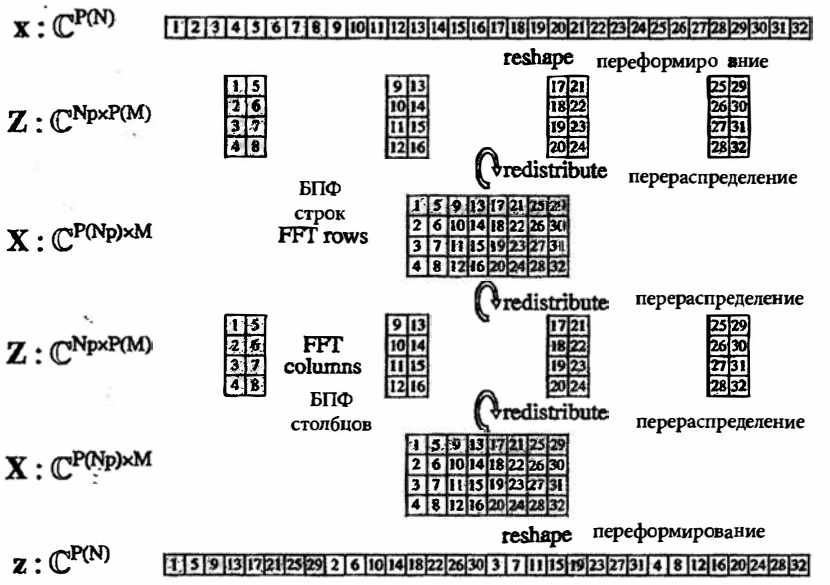


Рисунок 10.1. Параллельная реализация программы FFT

Показано БПФ 32-элементного вектора с $Np = 4$. Значения внутри каждого вектора или матрицы означают индекс этого элемента относительно входного вектора. Одномерное параллельное БПФ выполняет ряд операций переформирования и перераспределения для того, чтобы элементы данных находились в правильном порядке для использования параллельного двумерного БПФ

10.3. Код

Следующим шагом является написание программы FFT. Ключевые строки кода показаны в листинге кода 10.1 (полный листинг расположен в `Examples/FFT/pFFT.m`). Следует понимать, что один и тот же код выполняется каждым экземпляром MATLAB. Единственным различием между экземплярами является наличие у каждого из них уникального P_{ID} (т. е. P_{ID}), который используется для определения того, какие части распределенного массива принадлежат каждому из экземпляров MATLAB (т. е. *x.loc*).

В программе FFT

Строка 1 устанавливает флаг PARALLEL, который включает и выключает распределенные массивы.

Строка 2 устанавливает размер вектора и вычисляет размер соответствующей матрицы. Эта реализация требует, чтобы размер вектора N без остатка делился на N^2_p.

Строки 3–8 создают параллельную карту `Xmap`, размещают распределенный вектор `X` и инициализируют `X` множеством случайных комплексных чисел. Затем выделяется локальная часть. В режиме проверки (не показано) вектор может быть инициализирован известными входными данными.

Строка 9 создает пустой распределенный массив `Xshell`, используя `put_local` для присвоения константы вектору `X`. `Xshell` содержит всю параллельную информацию для *N*-элементного вектора, но не занимает места для хранения элементов.

Строки 10, 13, 16, 20, 23, 26 и **30** запускают и останавливают таймеры для вычислительной и коммуникационной частей программы.

Строка 11 переформирует локальный *M*-элементный вектор в матрицу размеров $Np \times M/Np$.

Строка 12 вставляет матрицу в соответствующую распределенную матрицу $Np \times M$.

Строка 14 перераспределяет данные с использованием функции `transpose_grid` путем транспонирования процессорной сетки в карте распределенной матрицы. Например, матрица с картой `map([1 Np],{}, 0: Np-1)` преобразуется в матрицу с картой `map([Np 1],{}, 0: Np-1)`. Использование этой команды более эффективно, чем использование общего оператора присваивания между распределенными массивами.

Строки 17–19 применяют БПФ к строкам матрицы, умножают их на поворачивающие множители и вставляют результаты обратно в распределенную матрицу.

Строка 21 перераспределяет данные второй раз с использованием функции `transpose_grid`.

Строки 24–25 применяют БПФ к столбцам матрицы и вставляют результаты обратно в распределенную матрицу X.

Строка 27 перераспределяет данные в третий раз с использованием функции `transpose_grid`.

Строка 29 вставляет локальную часть размеров $1 \times M$ распределенной матрицы X в локальную часть размеров $1 \times N$ распределенного вектора `Xshell` для создания результирующего вектора X того же размера и распределения, что и входной вектор.

Листинг кода 10.1. Параллельная MATLAB-программа FFT

```
1 PARALLEL=1;                                  % Turn parallelism on and off.
2 N = 2^24; M = N/Np;                          % Set vector/matrix dimensions.
3 Xmap = 1;                                    % Serial map.
4 if PARALLEL
5 Xmap = map([1 Np] ,{},0:Np-1);               % Parallel map.
6 end
7 X = complex(rand(1,N,Xmap),rand(1,N,Xmap));      % Create X.
8 Xloc = local(X);                             % Get local part.
9 Xshell = put_loeal(X,0);                     % Create an empty X.
10 tic;
11 Xloc = reshape(Xloc,Np,M/Np);               % Reshape local part.
12 X = put_local(zeros(Np,M,Xmap),Xloc);
13 Tcomp = toc; tic;
14 X = transpose_grid(X);                      % Redistribute.
15 Xloc = local(X);
16 Tcomm = toc; tic;
17 Xloc = fft(Xloc,[] ,2);                     % FFT rows.
18 Xloc = omega .* Xloc;                       % Multiply by twiddle factors.
19 X = put_loeal(X,Xloc);
```

```
20 Tcomp = Tcomp + toc; tic;
21 X = transpose_grid(X);                    % Redistribute.
22 Xloe = loeal(X);
23 Tcomm = Tcomm + toc; tic;
24 Xloc = fft(Xloc, [] ,1);                  % FFT columns.
25 X = put_local(X,Xloc);
26 Tcomp = Tcomp + toc; tic;
27 X = transpose_grid(X);                    % Redistribute.
28 Xloc = local(X);
29 X = put_local(Xshell,Xloc);               % Insert into vector.
30 Tcomm = Tcomm + toc;
```

Перевод
1 % Включение и выключение параллелизма;
2 % Установка размеров вектора/матрицы;
3 % Последовательная карта;
5 % Параллельная карта;
7 % Создать X;
8 % Получить локальную часть;
9 % Создать пустой X;
11 % Переформировать локальную часть;
14 % Перераспределение;
17 % БПФ для строк;
18 % Умножить на поворачивающие множители;
21 % Перераспределить;
24 % БПФ для столбцов;
27 % Перераспределить;
29 % Вставить в вектор.

Вышеприведенный код является ярким примером того, как среда программирования на основе распределенных массивов может реализовать сложные шаблоны перемещения данных. В частности, такие операции, как `transpose_grid` совместно с `reshape` и `put_local` допускают перераспределение данных эффективным способом с ясным пониманием того, когда операция является строго локальной и когда имеет место коммуникация.

На данном примере кода можно проследить некоторые свойства подхода к программированию на основе карт размещения.

- Влияние на код. Преобразование программы из последовательной в параллельную приводит к созданию кода несколько большего, чем последовательный. По большей части, увеличение кода вызвано использованием метода на основе двумерного БПФ, который допускает локальное выполнение БПФ над строками и столбцами. Преобразование последовательной программы на основе двумерного БПФ в параллельную требует лишь номинального увеличения кода.

- Последовательный код сохраняется. Посредством установки PARALLEL = 0 исключается использование распределенных массивов, и программа может быть выполнена последовательно без поддержки параллельных библиотек.
- Средний объем использования параллельной библиотеки. В дополнение к обычным функциям распределенных массивов *Np*, `local` и `put_local` программа использует только специальную функцию перераспределения `transpose_grid` с целью увеличения производительности. В общем, используется четыре функции параллельной библиотеки, что типично для программ на основе распределенных массивов.
- Масштабируемость. Код может выполняться на задачах различного размера при условии, что *N* без остатка делится на Np^2. Такое ограничение может быть ослаблено небольшим количеством дополнительного кода для обработки граничных условий.
- Ограниченная коммуникация. Распределенные массивы используются напрямую только при необходимости коммуникации. В остальных случаях все операции выполняются над локальными переменными.
- Зависимость от карт размещения. Этот конкретный алгоритм требует наличия у входных и выходных данных карты *XP()map*.
- Гарантия последовательной производительности. Так как операции БПФ применяются к локальным переменным, которые являются обычными числовыми массивами MATLAB, при выполнении данных строк кода гарантируется отсутствие скрытого ухудшения производительности.

10.4. Отладка

Как упоминалось ранее, отладка параллельной программы представляет собой непростую задачу. Основным преимуществом подхода на основе карт размещения является тот факт, что данный метод поддается четко определенному процессу кодирования–отладки–тестирования (см. таблицу 2.2).

Шаг 1. Выполнение программы в последовательном виде на локальной машине с вектором `X`, заданным как обычный массив (т. е. PARALLEL=0). Этим проверяется сохранение корректности используемого последовательного кода. Для работы такого последовательного кода параллельная библиотека предоставляет последовательные версии функций `local`, `put_local` и `transpose_grid`. Например, выражение `Xloc = local(X)` просто присваивает `X` вектору `Xloc`, если `X` – нераспределенный массив.

Шаг 2. Выполнение программы в последовательном виде на локальной машине с массивом `X` в качестве распределенного массива (т. е. PARALLEL=1) и проверка корректной работы программы с распределенными массивами в качестве входных данных.

Шаг 3. Выполнение программы параллельно на локальной машине. Данный шаг проверяет, являются ли параллельные карты, которые генерируют массивы `X` и `Xloc`, корректными. Что еще более важно, на этом шаге впервые выполняется коммуникация. Из всех шагов этот шаг является самым важным, так как коммуникация может способствовать появлению множества ошибок. Самой распространенной ошибкой является отличие карты массива от предполагаемой. Эта ситуация может быть выяв-

лена путем изучения размеров локальных частей и подтверждения того, что размеры не соответствуют желаемому распределению.

Шаг 4. Выполнение программы в параллельном режиме на удаленной машине.

Шаг 5. Выполнение программы в параллельном режиме на удаленной машине, но с задачей интересующего размера. До этого момента тестирование должно было осуществляться на задаче малого размера. Данный шаг позволяет определить производительность на реальной задаче.

Перечисленные шаги являются отражением сущности этапов, указанных в разделе «Начало работы», и на каждом из шагов тестируются определенные аспекты параллельного кода. В дополнение к отладке функциональности программы, вышеуказанный процесс также может быть использован для оптимизации производительности кода (см. следующий раздел). Помимо перечисленных выше шагов, тест производительности FFT использует некоторые дополнительные техники кодирования для помощи в отладке.

Программа FFT выводит на печать параметры программы, которые описывают размер задачи N, количество необходимых байтов и размер массива Xloc. Этот шаг осуществляется на каждом P_{ID} программы. Сравнение этих значений путем просмотра разных файлов с выводом от каждого P_{ID} (т. е. `MatMPI/*.out`) является действенным способом выявления ошибок. В программе FFT также отображается время размещения и запуска для каждого P_{ID}. Эти временные показатели описывают количество времени, затрачиваемое до начала выполнения программы. Признаком того, что в программе что-то происходит не так на раннем этапе работы, является значительное различие этих значений у разных P_{ID}.

Так как FFT является тестом производительности, производительность ее исполнения измеряется и выводится в единицах, относящихся к процессору. Программа FFT также разбивает эти показатели на время вычисления и время коммуникации. Значения производительности, значительно отличающиеся у разных экземпляров P_{ID}, могут быть показателем более глубоких проблем в программе (например, создание массивов чрезмерно различающихся размеров).

Наконец, программа FFT имеет проверку корректности, которая контролирует получение правильных результатов. В реальных приложениях такую проверку, как правило, произвести невозможно, однако, можно выполнить приблизительную проверку, которая бывает очень полезна. Например, часто можно проверить знак значений или сумму значений в массиве. Повторимся, что отображение этих значений для каждого P_{ID} может выявить ошибки, зависящие от P_{ID}.

10.5. Тестирование

Программа FFT является ярким примером программы, производительность которой увеличивается с ростом числа Np при условии, что пропускная способность коммуникации достаточна. Программа FFT часто используется для измерения пропускной способности системы параллельных вычислений.

10.5.1 Производительность на многоядерной системе

Тестирование параллельной программы – при условии, что она уже отлажена и выдает корректные результаты, – в основном заключается в определении количества времени, которое требуется для ее выполнения, и выяснения того, насколько ее производительность соответствует ожидаемой. Почти все данные о производительности, необходимые для проведения такого анализа, были представлены в разделе «Начало работы». В таблице 10.1 представлена многоядерная производительность на восьмиядерной системе.

Первый тест проводится для сравнения производительности выполнения программы с использованием $Np = 1$ с выключенными (PARALLEL=0) и с включенными распределенными массивами (PARALLEL=1). В столбцах 2 и 3 таблицы 10.1 представлены эти данные. В данном случае разница в производительности пренебрежимо мала и означает, что распределенные массивы не снижают производительности. Если использование распределенных массивов с $Np = 1$ значительно замедляет работу, эту проблему необходимо решать. Очень трудно получить какое-либо преимущество при $Np > 1$, если при $Np = 1$ операции выполняются медленнее, чем в исходном последовательном коде. Работа только на локальных частях распределенных массивов практически гарантирует, что производительность будет такой же, как при работе без использования распределенных массивов.

Таблица 10.1. Производительность программы FFT на многоядерной системе

Абсолютное время запуска, относительное время размещения, вычисления, коммуникации и выполнения, а также относительная производительность для теста FFT с использованием 8-ядерного процессора. Все значения нормализованы по столбцу 2, в котором $N_p = 1$ с выключенными распределенными массивами (PARALLEL = 0). Столбцы 3–6 содержат значения с включенными распределенными массивами (PARALLEL = 1). Запись 1*8 означает, что $Np = 8$ и все экземпляры P_{ID} были выполнены на одном процессорном узле с общей памятью

Многоядерные процессоры:	1	1*1	1*2	1*4	1*8
Время запуска (с)	0	0.0	1.23	1.80	1.57
Относительное время размещения	1	1.0	0.79	0.54	0.42
Относительное время вычисления	1	1.0	0.41	0.25	0.07
Относительное время коммуникации			0.99	0.33	0.30
Относительное время выполнения	1	1.0	1.41	0.58	0.37
Относительная производительность	1	1.0	0.71	1.72	2.70
Относительная пропускная способность			1	3.00	3.30

Столбцы 3–5 таблицы 10.1 демонстрируют параллельную производительность кода на многоядерной системе. Строка времени запуска показывает общее увеличение времени запуска с ростом Np. В идеале это должна быть постоянная величина, не зависящая от Np. Однако в зависимости от используемого механизма запуска этот показатель времени может представлять собой возрастающую функцию от Np. Как правило, время запуска не составляет большую часть от общего времени вычисления, но оно может с легкостью увеличиться при больших значениях Np. Например, предположим, что время запуска пропорционально Np, и каждый дополнительный P_{ID} до-

бавляет одну секунду. Если $Np = 100$, время запуска составит 100 секунд. Если это доля Амдала программы, то программе понадобится для выполнения не менее Np^2секунд для достижения 50-кратного ускорения по сравнению с последовательным кодом.

Строки относительного времени размещения показывают, что оно обычно уменьшается с увеличением Np, так как каждому P_{ID} необходимо разместить N/Np элементов данных. Измеренное уменьшение времени размещения не изменяется линейно с Np, так как с размещением связаны некоторые накладные расходы. Как правило, этот период времени не будет занимать большую часть общего времени вычисления. Однако при большом числе Np эти расходы могут стать существенными, таким образом, рекомендуется измерять время размещения и проверять его приемлемость.

Относительное время вычислений, как правило, уменьшается линейно с ростом Np, так как каждое ядро работает над меньшей проблемой. Относительное время коммуникации вычисляется по отношению ко времени вычисления для случая $Np = 1$. Таким образом, для $Np = 2$ относительное время коммуникации равно времени вычисления, и это приводит к итоговому уменьшению общей производительности по сравнению со случаем $Np = 1$. С увеличением Np время коммуникации уменьшается и наблюдается общее уменьшение относительного времени вычисления.

Строка относительной производительности показывает, что достигнуто ускорение 2,7 при $Np = 8$. Это не является неожиданностью при заданном большом объеме имеющей место коммуникации. В некоторых случаях такое улучшение производительности может быть преимуществом; в других ситуациях из-за такого увеличения программу усложнять не стоит. В целом параллельная программа FFT попадает в середину значений производительности.

Строка относительной пропускной способности демонстрирует, что с ростом Np пропускная способность процессоров возрастает. Фактически,возрастание Np в 4 раза (т. е. переход от $Np = 2$ до $Np = 8$) приводит к увеличению пропускной способности в 3,3 раза. Такое увеличение пропускной способности с ростом Np является единственной причиной, по которой FFT достигает приемлемой производительности в параллельном режиме. Если бы производительность не улучшилась, то общее время коммуникации осталось бы приблизительно таким же, как время вычисления для $Np = 1$.

10.5.2. Производительность на многоузловой системе

Результаты выполнения параллельного теста FFT на многоузловой системе показаны в таблице 10.2. Результаты приведены как для фиксированного размера задачи (N – константа), так и для масштабированного размера задачи $N \alpha Np$. Фиксированный размер задачи выбирался таким же, как в случае многоядерной системы. Масштабированный размер выбирался таким же, как в случае многоядерной системы для $Np = 1$, но увеличенный в 16 раз при $Np = 16$. Увеличение общего размера задачи является одним из преимуществ многоузловой системы над многоядерной системой.

Таблица 10.2. Производительность программы FFT на многоузловой системе

Абсолютное время запуска, относительное время размещения, вычисления, коммуникации и выполнения, а также относительная производительность для теста FFT с использованием 16-узловой системы. Данные о производительности приведены как для фиксированного размера задачи (N – константа), так и для масштабированного размера задачи ($N\alpha Np$). Все значения нормализованы по столбцу 2, в котором $N_p = 1$ с выключенными распределенными массивами (PARALLEL=0). Столбцы 3–7 содержат значения с включенными распределенными массивами (PARALLEL=1). Запись 16*1 означает, что $Np = 16$ и каждый P_{ID} был выполнен на своем индивидуальном процессорном узле с использованием отдельной памяти

Многоузловые процессоры:	1	1*1	2*1	4*1	8*1	16*1
Фиксированный размер задачи						
Время запуска (с)	0	0.0	10.21	9.57	11.36	10.03
Относительное время размещения	1	1.0	0.82	0.61	0.51	0.39
Относительное время вычисления	1	1.0	0.56	0.27	0.088	0.044
Относительное время коммуникации			1.01	0.63	0.41	0.29
Относительное время выполнения	1	1.0	1.57	0.90	0.50	0.33
Относительная производительность	1	1.0	0.64	1.11	2.00	3.03
Относительная пропускная способность			1	1.51	2.42	3.53
Масштабированный размер задачи						
Относительная производительность	1	1.0	0.62	1.08	2.08	3.92
Относительная пропускная способность			1	1.43	2.86	4.83

Время запуска на многоузловой системе склонно к увеличению с ростом Np и показывает большие отклонения. Более того, время запуска намного больше, чем в случае многоядерной системы, так как отдельные системы для запуска экземпляров P_{ID} должны совершать обмен по сети. Относительное время размещения и время вычисления сходны для многоядерной и многоузловой систем.

Время коммуникации для $Np = 8$ выше на могоузловой системе, чем на многоядерной, что приводит к незначительному снижению производительности. Многоузловая система имеет коэффициент ускорения 2,02 против коэффициента 2,7 на многоядерной системе. Однако многоузловая система может быть масштабирована до $Np = 16$, при котором общий коэффициент ускорения возрастает до 3,03.

Многоузловая система также допускает масштабирование размера задачи. Интересно, что производительность растет по мере масштабирования задачи, так что при $Np = 16$ наблюдается коэффициент ускорения 3.92. Причина этому в том, что с ростом размера задачи: $O(N\lg(N))$ количество операций растет быстрее, чем объем данных при коммуникации: $O(N)$. Кроме того, общая пропускная способность возрастает из-за того, что увеличивается размер сообщений и относительно меньше времени затрачивается на задержку коммуникации. Таким образом, при увеличении размера задачи возрастает общая производительность.

Производительность многоядерная (фиксированная), многоузловая (фиксированная) и многоузловая (масштабированная) показана на рисунке 10.2.

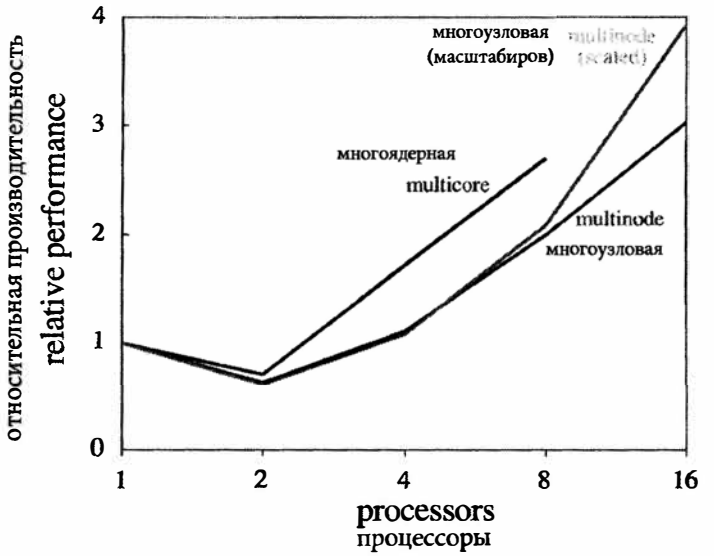


Рисунок 10.2. Относительная производительность параллельной программы FFT
Относительная производительность параллельной программы FFT на многоядерных и многоузловых системах. Показана также производительность на многоузловой системе для масштабированной задачи $N\alpha Np$

Во всех случаях добавление коммуникации приводит к общему уменьшению производительности для случая $Np = 2$. С увеличением Np увеличивается пропускная способность и наблюдается общее улучшение производительности. В случаях многоядерной системы (фиксированная задача) и многоузловой системы (фиксированная задача) наклон графика производительности позволяет предположить, что увеличение числа Np обеспечит дополнительную производительность. Многоузловая (масштабированная) система показывает увеличение наклона графика производительности с ростом Np.

Собрав и изучив все данные о производительности программы FFT, можно сделать следующие выводы о выполнении приложений такого типа на кластерах многоядерных процессорных узлов. Во-первых, программа FFT принципиально ограничена пропускной способностью используемого оборудования, и при $Np > 1$ программа склонна работать медленнее, чем при $Np = 1$. Некоторое увеличение производительности наблюдается на многоядерных и многоузловых системах при фиксированном размере задачи. Однако более значительное увеличение производительности достигается на многоузловой системе при масштабированном размере задачи.

Список литературы

1. *[Dongarra and Luszczek 2005] Jack Dongarra and Piotr Luszczek, Introduction to the HPC Challenge benchmark suite, Technical Report UT-CS-05-544, University of Tennessee, 2005.*

Глава 11
Программа High Performance LiNpack

Резюме

Тест производительности HPL решает плотную линейную систему **Ax=b** и является представителем широкого класса задач вычислительной линейной алгебры. Спецификация программы HPL проста, и последовательные реализации могут быть довольно короткими. Эффективные параллельные алгоритмы HPL требуют сложных двумерных блочно-цикличных распределений данных, что приводит к реализациям с длинным кодом. Более того, шаблон коммуникации HPL сложен для реализации с использованием чистых распределенных массивов, поэтому требуется комбинированный подход, использующий распределенные массивы и обмен сообщениями. Показан более простой одномерный блочный алгоритм, который с легкостью применяется к широкому классу параллельных задач линейной алгебры. Программа HPL прекрасно иллюстрирует такие сложные концепции параллельного программирования, как одновременное выполнение вычислений и коммуникации и анализ критических путей.

Программа High Performance LiNpack (HPL) была представлена в главе 7 в качестве теста для измерения основной производительности процессора [Dongarra and Luszczek 2005]. По существу, она тестирует возможность каждого процессора в параллельной системе выполнять умножение плотных матриц. Большинство процессоров оснащены специализированным аппаратным обеспечением для поддержания вычислений скалярных произведений векторов, которые используются для перемножения плотных матриц. Эффективная реализация HPL может дать результат, близкий к теоретической пиковой производительности системы. Фактически HPL часто используется для определения пиковой производительности системы и является основой для составления списка Top500 (http://www.Top500.org), используемого в качестве рейтинга крупнейших вычислительных систем в мире.

Программа HPL также является ярким примером сложного параллельного приложения со сложным шаблоном коммуникации и позволяет продемонстрировать ряд полезных методик параллельного кодирования, в частности способ написания комбинированной программы, которая использует распределенные массивы для организации коммуникации, а обмен сообщениями для фактической пересылки сообщений.

Программа HPL создает большую случайную матрицу $\mathbf{A} : R^{N \times N}$ и соответствующий случайный вектор $\mathbf{b} : R^N$, и находит вектор $\mathbf{x} : R^N$ как решение системы линейных уравнений:

$$\mathbf{Ax} = \mathbf{b}.$$

Алгоритм HPL имеет ряд шагов. Сначала – **A** раскладывается в сомножители (факторизуется) на нижне-треугольную матрицу **L** и верхне-треугольную матрицу **U**, так что

$$\mathbf{LU} = \mathbf{A}.$$

Эта распространенная алгебраическая операция называется «LU-разложение матрицы», или «LU-декомпозиция», и представляет собой сущность HPL. После получения **L** и **U**, **x** находится при помощи метода исключений Гаусса.

Программа HPL обычно совершает $O(N^3)$ операций и передает $O(N^2)$ элементов данных. Так как количество операций, выполняемых над каждым элементом данных, велико, и одновременно можно выполнить много операций, производительность HPL ограничивается скоростью процессоров. Скорость коммуникационной сети обычно менее важна для конкурентоспособных параллельных реализаций программы HPL. Однако конкурентоспособные реализации HPL требуют осуществления блочно-циклических распределений для $\mathbf{A} : R^{P_{c(n)}(N \times N)}$, $\mathbf{b} : R^{P_{c(n)}(N)}$ и $\mathbf{x} : R^{P_{c(n)}(N)}$. Программирование на основе распределенных массивов естественным образом поддерживает такой тип распределений. Написание кода, необходимое для координирования учета индексов при комбинированной реализации, в данной главе не рассматривается.

Одномерные блочные распределения служат отправной точкой для большинства параллельных алгоритмов линейной алгебры. Как правило сначала реализуются более простые одномерные блочные алгоритмы, которые впоследствии усовершенствуются, если требуется получить дополнительную производительность. В данной главе рассматриваются как методы кодирования, так и алгоритмические методы эффективной реализации одномерных блочных параллельных алгоритмов посредством одновременного исполнения вычислений и коммуникации. Кроме того, представлен метод анализа критических путей для демонстрации способа оценки параллельной производительности сложных алгоритмов.

11.1. Начало работы

Для выполнения программы запустите MATLAB и перейдите в директорию Examples/HPL (т.е. наберите `cd Examples/HPL`). Отредактируйте файл `pHPL.m` и установите флаг `PARALLEL = 0`. В командной строке введите

```
pHPL
```

Так как HPL является тестом производительности, она выдаст большой объем данных о производительности подобного вида:

```
Np                                = 1
Pid                               = 0
Distributed matrix size           = 5000^2 words
Distributed matrix size (bytes)   = 200000000
Local matrix size (bytes)         = 200000000
Allocation Time (sec)             = 1.8742
Launch Time (sec)                 = 0.002242
0
Run time (sec)                    = 58.3743
Performance (Gigaflops)           = 1.4282
Verification Passed
```

Перевод

Размер распределенной матрицы;

Размер распределенной матрицы (байт);

Размер локальной матрицы (байт);
Время размещения (с);
Время запуска (с);
Время выполнения (с);
Производительность (гигафлопс);
Проверка пройдена.

Эти данные можно разбить на пять категорий. Первая – параметры программы, которые включают в себя Np, P_{ID}, размер задачи N, общее количество необходимых байтов и количество байтов, необходимых каждому P_{ID}. Вторая категория включает в себя время размещения и время запуска, описывающие количество времени, затрачиваемое до начала выполнения программы. Третья категория – это номер выполняемой итерации, который отсчитывается от 0 до $Np - 1$. В случае $Np = 1$ печатается только 0. Четвертая категория – это время выполнения и производительность в гигафлопсах. Пятая категория – статус проверки. Эти результаты будут рассмотрены далее в разделе «Тестирование». Сейчас отредактируйте файл `pHPL.m`, установите `PARALLEL=1` (для включения параллельных частей кода) и `VALIDATE=0` (для сокращения времени выполнения). Повторно запустите программу pHPL с использованием следующих команд:

```
eval(pRUN('pHPL' ,1,{}))
eval(pRUN('pHPL' ,2,{}))
eval(pRUN('pHPL' ,4,{}))
```

Данные команды выполняют программу HPL при $Np = 1, 2, 4$. При выполнении на параллельной вычислительной машине выдаются значения производительности, аналогичные указанным в таблице 11.1 (см. раздел «Тестирование»). Эти результаты будут рассмотрены далее в разделе «Тестирование».

Важное примечание. Последний аргумент команды `pRUN` определяет, на какой параллельной вычислительной машине будет выполняться программа. Может возникнуть необходимость в получении надлежащих установочных параметров у вашего системного администратора; для получения более подробной информации обращайтесь на веб-сайт книги (http://www.siam.org/KepnerBook). Установка последнего аргумента значением {} приводит к выполнению множества копий программы на вашей локальной машине и получению значений производительности, отличных от указанных в таблице 11.1.

11.2. Параллельная реализация

Основой теста производительности HPL является выполнение LU-разложения случайной матрицы с двойной точностью. Простой трехстрочный LU-алгоритм выглядит следующим образом [Demmel 2001]:

$$
\begin{aligned}
&\textbf{for } i = 1 : N-1 \\
&\quad \textbf{do} \\
&\qquad \mathbf{A}(i+1:N,i) = \mathbf{A}(i+1:N,i)/\mathbf{A}(i,i) \\
&\qquad \mathbf{A}(i+1:N,i+1:N) = \mathbf{A}(i+1:N,i+ \\
&\qquad\quad -\mathbf{A}(i+1:N,i) * \mathbf{A}(i,i+1:N)
\end{aligned}
$$

Алгоритм выполняется так: для каждого столбца *i* элементы ниже диагонали **A***(i+1: N, i)* делятся на элемент диагонали **A***(i,j)*. Затем находится тензорное произведение этого столбца с соответствующей строкой **A***(i+1: N, i)** **A***(i,i+ 1 : N)*, и вычитается результирующая матрица. Результат вышеуказанного алгоритма заключается в преобразовании **A** так, чтобы ее верхняя часть хранила верхне-треугольную матрицу **U**, а нижняя часть – нижне-треугольную матрицу **LU**, где *i* – единичная матрица.

Исходный алгоритм LU-разложения прост, но склонен к ошибкам вычисления, при значениях **A***(i,j)* близких к нулю. Модификация алгоритма, способная облегчить данную проблему, называется **LU**-разложением матрицы с частичным выбором главного элемента и приведена в Алгоритме 11.1 (см. более подробную информацию в подписи).

Алгоритм 11.2 демонстрирует, каким образом можно совместно использовать распределенные массивы и обмен сообщениями. В частности, распределенный массив **A** используется для разбиения данных по идентификаторам P_{ID}. Информация в массиве **A** также используется для определения глобальных индексов **i**, которыми владеет каждый P_{ID}, а также глобальных индексов **ip** (текущие раскладываемые столбцы). При записи строки 1 и 6 используют угловые скобки для вычленения этих глобальных индексов. Помните, что ссылки на определенный индекс в локальной части даются обычным способом: **A**.*loc(i,j)*. Как правило, глобальный индекс **A***(i,j)* отличается в массиве от локального индекса **A**.*loc(i,j)*. Если необходимо получить локальную часть от определенного P_{ID}, ее можно далее обозначить **A**.loc_p. Когда требуется глобальный индекс определенного локального значения, он обозначается с использованием **A**.*loc(i,j)*. Очевидно, *i* = **A**(*i*,1). В более общем виде следующий единичный элемент будет содержать

$$\mathbf{A}(\mathbf{A}.loc(i,1)) = \mathbf{A}.loc(i,1).$$

Минимальные и максимальные глобальные индексы на конкретном P_{ID}, таким образом, задаются как **A**.loc_p*(1)* и **A**.loc_p*(end)*.

Алгоритм 11.1. Последовательный алгоритм LU-разложения

Алгоритм раскладывает матрицу **A** на нижне-треугольную **L** и верхне-треугольную **U** матрицы-сомножители. В алгоритме используется частичный выбор главного элемента, и индексы главных элементов хранятся в векторе **v**. Строка 3 находит позицию самого большого по абсолютной величине значения в столбце **i**. Строка 4 выполняет обмен текущей строки со строкой, содержащей главный элемент. Строки 5 и 6 обновляют матрицу. Строки 7 и 8 извлекают верхне- и нижне-треугольные матрицы из **A**.

```
L, U : ℝ^{N×N}, v : ℤ^N = SerialLU(A : ℝ^{N×N})
1  for i = 1 : N − 1
2        do
3           v(i) = argmax(|A(i + 1 : N, i)|) + i − 1
4           A = swap(A, i, v(i))
5           A(i + 1 : N, i) = A(i + 1 : N, i)/A(i, i)
6           A(i + 1 : N, i + 1 : N) = A(i + 1 : N, i + 1 : N)
               −A(i + 1 : N, i) * A(i, i + 1 : N)
7  L = lower(A) + I
8  U = upper(A)
```

Цель программы HPL заключается в измерении способности вычислительной машины выполнять операции с плавающей запятой. Производительность измеряется в гигафлопсах и вычисляется по формуле

$$\text{Gigaflops:} \quad ((2/3)N^3+(3/)N^2)/t_{run}/10^9.$$

Проверка корректности является важной частью любого теста производительности. Если массивам **A** и **b** присвоены случайные значения, то можно провести ряд тестов для того, чтобы определить, насколько корректными являются вычисленные ответы. Во-первых, вычисляются невязки:

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x},$$

и берется норма с индексом «бесконечность»: $r_\infty = \| r \|_\infty$. Кроме того, вычисляются следующие дополнительные нормы:

$$r_1 = \frac{r_\infty}{e \| \mathrm{r} \|_1 N}$$

$$r_2 = \frac{r_\infty}{e \| \mathrm{r} \|_1 \| \mathrm{x} \|_1}$$

$$r_3 = \frac{r_\infty}{e \| \mathrm{r} \|_\infty \| \mathrm{x} \|_\infty},$$

где e – машинная точность. Проверка считается пройденной, если выполняется следующее неравенство:

$$\max(r_1, r_2, r_3) < 16.$$

Последовательные алгоритмы HPL и LU просты в определении и являются представителями широкого круга операций линейной алгебры. Параллельные алгоритмы HPL и LU более сложные и дают возможность проследить, каким образом комбинированная модель программирования может быть использована для реализации параллельных операций линейной алгебры.

11.2.1. Параллельное LU-разложение

LU-разложение матрицы заключает в себе свойства многих операций линейной алгебры. В частности, алгоритм работает от первого до последнего столбца, выполняя операции, при которых результаты последующего столбца зависят от результатов предыдущего. Этот процесс создает некоторые трудности при разработке параллельного алгоритма, в частности необходимо найти способ загрузки работой всех экземпляров P_{ID}, в то время как в любой заданный момент времени работа производится только над одним столбцом.

Тест производительности HPL корректно выполняется на задаче масштабированного размера, такой что $N^2 \alpha Np$. Первым шагом при выполнении параллельного LU-разложения является распределение матрицы **A** таким образом, чтобы она помещалась в память системы. Как правило, матрица **A** распределяется так, чтобы каждый *PID* хранил блок столбцов. Это распределение обозначается как **A** : $R^{N \times P(N)}$.

Выполнение параллельного LU-алгоритма продолжается итерациями от $p = 0$ до $p = Np-1$. Когда $p = P_{ID}$, этот P_{ID} выполняет последовательные операции LU-разложения на своей части **A**.*loc* и возвращает локальные части $\mathbf{L}_p$, $\mathbf{U}_p$ и индексы главных элементов $\mathbf{v}_p$, **Lp**, и $\mathbf{v}_p$ затем широковещательно рассылаются всем экземплярам, P_{ID} которых больше p, чтобы они применили их к своим частям **A**.*loc*. Этот алгоритм приведен в Алгоритме 11.2 (более подробная информация в подписи к Алгоритму). Так как коммуникация между P_{ID} изменяется по ходу алгоритма, комбинированный подход на основе распределенных массивов и обмена сообщениями является одним из эффективных моделей программирования данного теста производительности. Комбинированный подход использует распределенные массивы для распределения массива **A** совместно с обменом сообщениями для отправления результатов локальных LU-разложений разным P_{ID}.

Алгоритм 11.2. Параллельный алгоритм LU-разложения

Алгоритм раскладывает распределенную по столбцам матрицу **A** на нижне- (**L**) и верхне- (**U**) треугольные матрицы-сомножители. Строки 1 и 2 вычисляют глобальные индексы столбцов из **A**.*loc*. Строка 3 инициализирует вектор индексов главных элементов v. Строка 4 выполняет цикл по блоку столбцов. Строки 5 и 7 вычисляют глобальные индексы столбцов **A**.loc_p. Строка 8 инициализирует тэги сообщений. Строки 9–16 вычисляют LU-сомножители для **A**.*loc* и отправляют их другим P_{ID}. Строки 17 и 18 получают сообщения от отправляющего P_{ID}. Строка 19 обновляет индексы главных элементов. Строка 20 осуществляет обмены строк в соответствии с главными элементами. Строки 21 и 25 применяют преобразование $\mathbf{L}_p$ к **A**.*loc*. Строки 26–28 вставляют результаты в распределенные массивы **L** и **U**.

$\mathbf{L}, \mathbf{U} : \mathbb{R}^{N \times P(N)}, \mathbf{v} : \mathbb{Z}^N = \textsc{ParallelLU}(\mathbf{A} : \mathbb{R}^{N \times P(N)})$

1 $\mathbf{i} = \mathbf{A}.loc\langle 1, 1 : end\rangle, \qquad N_{loc} = \textsc{size}(\mathbf{i})$
2 $i^1 = \mathbf{i}(1), \qquad i^2 = \mathbf{i}(N_{loc})$
3 $\mathbf{v} = 1 : N$
4 **for** $p = 0 : N_P - 1$
5 **do**
6 $\mathbf{i}_p = \mathbf{A}.loc_p\langle 1, 1 : end\rangle \qquad N^p_{loc} = \textsc{size}(\mathbf{i}_p)$
7 $i^1_p = \mathbf{i}_p(1), \qquad i^2_p = \mathbf{i}_p(N^p_{loc})$
8 $tag_{hi} = 2 * p + 1 \mod 32, \qquad tag_{lo} = 2 * p \mod 32$
9 **if** $p = P_{ID}$
10 **do**
11 $\mathbf{L}_p, \mathbf{U}_p, \mathbf{v}_p = \textsc{SerialLU}(\mathbf{A}.loc(i_1 : N, :))$
12 $\mathbf{v}_p = \mathbf{v}_p + i_1 - 1$
13 $\textsc{SendMsg}((p+1) : (N_P - 1), tag_{hi}, \mathbf{L}_p, \mathbf{v}_p)$
14 $\textsc{SendMsg}(0 : (p-1), tag_{lo}, \mathbf{v}_p)$
15 $\mathbf{L}_p(1 : N_{loc}, :) = \mathbf{L}_p(1 : N_{loc}, :) + \mathbf{U}_p - \mathbf{I}$
16 $\mathbf{A}.loc(i_1 : N, :) = \mathbf{L}_p$
17 **if** $p < P_{ID} \qquad \mathbf{L}_p, \mathbf{v}_p = \textsc{RecvMsg}(p, tag_{hi})$
18 **if** $p > P_{ID} \qquad \mathbf{v}_p = \textsc{RecvMsg}(p, tag_{lo})$
19 $\mathbf{v}(i^1_p : N) = \mathbf{v}(\mathbf{v}_p)$
20 **if** $p \neq P_{ID} \qquad \mathbf{A}.loc(\mathbf{i}_p, :) = \mathbf{A}.loc(\mathbf{v}_p, :)$
21 **if** $p > P_{ID}$
22 **do**
23 $\mathbf{i}_L = (N^p_{loc} + 1) : \textsc{size}(\mathbf{L}_p, 1), \qquad \mathbf{i}_A = (i^2_p + 1) : N$
24 $\mathbf{A}.loc(\mathbf{i}_p, :) = \mathbf{L}_p(1 : N^p_{loc}, :) \setminus \mathbf{A}.loc(\mathbf{i}_p, :)$
25 $\mathbf{A}.loc(\mathbf{i}_A, :) = \mathbf{A}.loc(\mathbf{i}_A, :) - \mathbf{L}_p(\mathbf{i}_L, :) * \mathbf{A}.loc(\mathbf{i}_p, :)$
26 $\mathbf{L}.loc = \textsc{lower}(\mathbf{A}.loc, -i^1)$
27 $\mathbf{L}.loc(\mathbf{i}, :) = \mathbf{L}.loc(\mathbf{i}, :) + \mathbf{I}$
28 $\mathbf{U}.loc = \textsc{upper}(\mathbf{A}.loc, 1 - i^1)$

Алгоритм 11.2 демонстрирует, каким образом можно совместно использовать распределенные массивы и обмен сообщениями. В частности, распределенный массив **A** используется для разбиения данных по P_{ID}. Информация в массиве **A** также используется для определения глобальных индексов **i**, которыми владеет каждый P_{ID}, а также глобальных индексов $\mathbf{i}_p$ (текущие раскладываемые на сомножители столбцы). С точки зрения обозначений, строки 1 и 6 используют угловые скобки для получения этих глобальных индексов. Помните, что использование для ссылок определенных индексов в локальной части выполняется обычным способом: **A**.*loc(i,j)*. Как правило, глобальный индекс **A***(i,j)* отличается в массиве от локального индекса **A**.*loc(i,j)*. Если необходимо получить локальную часть для определенного P_{ID}, это обозначается как **A**.loc_p. Когда требуется глобальный индекс определенного локального значения, он обозначается с использованием **A**.*loc(i,j)*. Очевидно, i = **A**(i,1). В более общем виде будет верным следующее равенство:

$$\mathbf{A}(\mathbf{A}.loc(i,1)) = \mathbf{A}.loc(i,1).$$

Минимальные и максимальные глобальные индексы на конкретном P_{ID}, таким образом, определяются как $\mathbf{A}.loc_p(1)$ и $\mathbf{A}.loc_p(end)$.

Эти подматрицы **A** изображены на рисунке 11.1 для $Np = 4$ и текущего значения $p = 1$. Во-первых, на $P_{ID} = p$ подматрица $\mathbf{A}.loc(i_p{}^1{:}N,:)$ или $\mathbf{A}.loc(i_p{}^1{:}N,\mathbf{i}_p)$ будет разложена на сомножители $\mathbf{L}_p$ и $\mathbf{U}_p$ и связанные с ней индексы главных элементов $\mathbf{v}_p$ затем отправляются всем $P_{ID} > p$, где они применяются к локальным подматрицам $\mathbf{A}.loc(i^1{:}N,:)$ или $\mathbf{A}.loc(i^1{:}N,\mathbf{i})$.

11.2.2. Анализ критический путей

Блочный параллельный LU-алгоритм представляет собой самый простой параллельный LU-алгоритм, однако он может потенциально привести к проблемам из-за узких мест в производительности. В частности, будет наблюдаться неравномерность нагрузки, так как не все экземпляры P_{ID} работают в течение всего времени. Кроме того, существует время, связанное с передачей $\mathbf{L}_p$ всем $P_{ID} > p$. Параллельный LU-алгоритм выполняется в несколько шагов, и проанализировать производительность всех шагов достаточно сложно. Другим методом является изучение «критического пути» алгоритма, состоящего из последовательности шагов, каждый из которых должен быть завершен для того, чтобы можно было перейти к следующему шагу. Критический путь Алгоритма 11.2 состоит из строк 11, 13, 24 и 25, выполняющихся на $P_{ID} = p$. Если эти строки выполнены, алгоритм заканчивается (см. рисунок 11.2). Другие строки тоже важны, но не входят в критический путь, и время выполнения этих строк не вызовет задержку выполнения работы другими P_{ID}.

Для начала анализа приведем время вычисления для идеального LU-разложения матрицы размером $N \times N$ на Np процессорах, которое приблизительно рассчитывается как

$$T_{comp} = \frac{2}{3} N^3 t_{comp} / Np,$$

где t_{comm} – время выполнения одной операции. Допустим, что размер задачи масштабируется числом Np при помощи установки $N = rNp$. Как правило, r будет иметь ограничение $r^2Np = c$, в результате чего имеем $\mathbf{A}.loc : \mathrm{R}^{N \times r}$, которая будет занимать фиксированный объем памяти c для каждого P_{ID}. Тогда время вычисления составит

$$T_{comp} = \frac{2}{3} N_p^2 r^3 t_{comp}.$$

Теперь рассмотрим время выполнения параллельного LU-разложения матрицы с использованием одномерного блочного распределения. Алгоритм состоит из $p = 0,\ldots, Np - 1$ шагов , и на каждом шаге необходимо полностью выполнить три операции, прежде чем можно будет начать следующий шаг. Во-первых, локальное LU-разложение подматрицы размером $(N - pr) \times r$ выполняется на $P_{ID} = p$ за время вычисления равное

$$(N_p - p - \frac{5}{3}) r^3 t_{comp}.$$

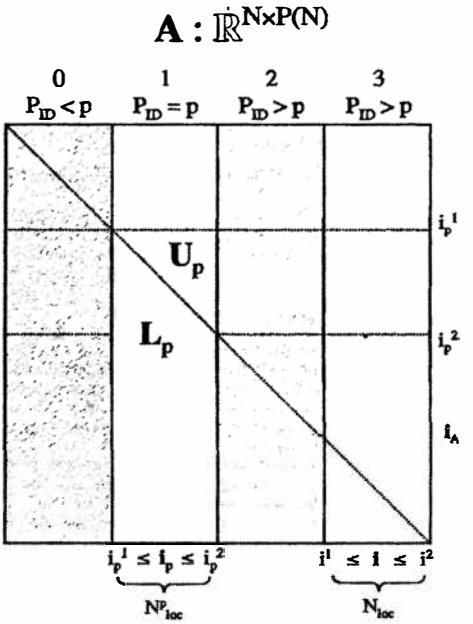


Рисунок 11.1. Параллельная реализация алгоритма LU-разложения
Подматрицы, используемые в параллельном LU-алгоритме для $Np = 4$. Алгоритм выполняется от $p = 0$ до $p = 3$. Подматрицы изображены на этапе, когда $p = 1$. $\mathbf{i}_p$ представляет глобальные индексы для $P_{ID} = p$. i представляет глобальные индексы для $P_{ID} \neq p$ и изображен для $P_{ID} = 3$

Во-вторых, результат этого локального LU-разложения передается широковещательно всем $P_{ID} > p$. Однако только коммуникация от $P_{ID} = p$ до $P_{ID} = p + 1$ лежит на критическом пути и имеет время коммуникации равное

$$(Np - p)r^2 t_{comm},$$

где t_{comm} – среднее время пересылки 8-байтового элемента данных между двумя P_{ID} (подразумеваются большие сообщения). Третьим и последним шагом является применение локального LU-разложения к локальной части матрицы, хранящейся на процессоре, с использованием операции обновления с нижне-треугольной матрицей и умножения матрицы на матрицу:

$$r^3 t_{comp}(1 + Np - p).$$

С этого момента формулы первого и третьего шагов объединяются, так как они отличаются только константой.

Суммируя вышеуказанные шаги от $p = 0$ до $p = Np-1$, получаем общее время, необходимое для вычисления локальных LU-сомножителей, обновления нижне-треугольной матрицы и перемножения матриц:

$$2r^3 t_{comp} \sum_{p=0}^{Np-1} (Np - p - \frac{1}{3})$$
$$= 2r^3 t_{comp} \sum_{p'=1}^{Np} (p' - \frac{1}{3})$$
$$= \frac{1}{3} N_p^2 r^3 t_{comp} (3 + N_p^{-1})$$

Подобным образом получается время коммуникации:

$$r^2 t_{comm} \sum_{p=0}^{Np-1} (N_p - p)$$
$$= r^2 t_{comm} \sum_{p'=1}^{Np} p'$$
$$= \frac{1}{2} r^2 t_{comm} N_p^2 (1 + N_p^{-1}).$$

Совмещая выражения вычисления и коммуникации, получаем:

$$N_p^2 r^3 \left[t_{comp} \frac{1}{3} (3 + N_p^{-1}) + t_{comm} \frac{1}{2r} (1 + N_p^{-1}) \right].$$

Разделив вышеуказанный результат оценки времени на идеальный коэффициент ускорения, получаем относительное время одномерного параллельного алгоритма LU-разложения для идеального случая:

$$\frac{1}{2}(3 + N_p^{-1}) + \frac{3t}{4r} \frac{t_{comm}}{t_{comp}} (1 + N_p^{-1}).$$

Рассмотрим случай, когда $10 < N_p < 1000$: Тогда выражения Np^{-1}становятся малыми. Для многих систем параллельных вычислений отношение коммуникация–вычисление обычно составляет $100 < t_{comm}/t_{comm} < 1000$ и $r = \sqrt{(c/N_P)} > 1000$ для стандартной памяти на каждый узел в количестве $c > 10^9$. Таким образом, второе выражение для коммуникации обычно будет < 0.1, и общая производительность относительно идеального случая составит ≈ 1.5. Во многих случаях такое уменьшение производительности является приемлемой ценой упрощения алгоритма.

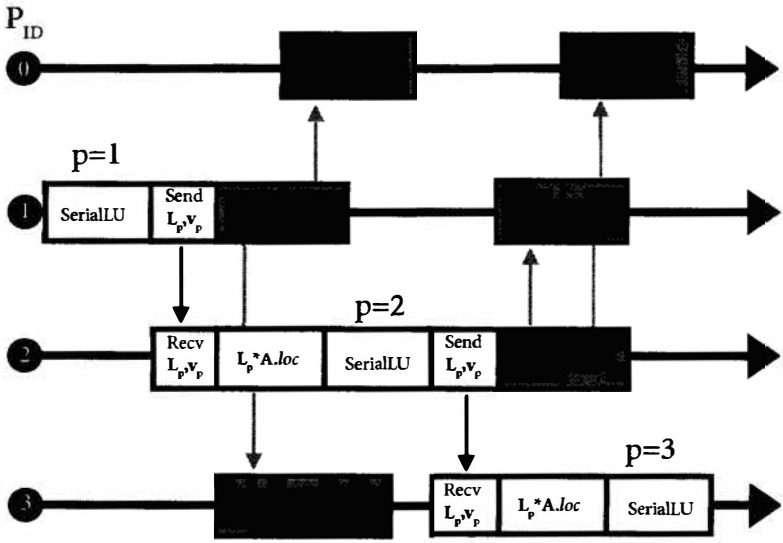


Рисунок 11.2. Критический путь параллельного LU-алгоритма
Параллельный LU-алгоритм для $Np = 4$, начиная от $p = 1$ и доходя до $p = 3$. Шаги на критическом пути показаны на белом фоне. Шаги, находящиеся вне критического пути, показаны на сером фоне

11.3. Код

Следующим шагом является написание программы HPL. Реализация состоит из двух основных частей. Первая часть – это управляющая часть кода, которая устанавливает параметры, инициализирует **A** и **b**, выполняет параллельное LU-разложение и затем использует результат для поиска решения **x**. Вторая часть – это параллельный LU-алгоритм, который показан в листинге кода 11.1 (полный листинг расположен в директории `Examples/HPL/pLUfactor.m`). Следует понимать, что один и тот же код выполняется каждым экземпляром MATLAB. Единственным различием между экземплярами является наличие у каждого из них уникального P_{ID} (т. е. `Pid`), который используется для определения какие части распределенного массива принадлежат каждому из экземпляров MATLAB (т. е. **A**.*loc*).

В параллельной LU-программе

Строка 1 объявляет входные и выходные параметры функции. Эта функция будет работать либо на распределенных, либо на нераспределенных квадратных матрицах. Карты результата **L** и **U** будут такими же, как для входной матрицы **A**. Если **A** является распределенной, поддерживается только одномерное блочное распределение $\mathbf{A} : R^{N \times P(N)}$.

Строки 2–4 извлекают локальную часть из **A** и получают соответствующие размеры.

Строка 5 возвращает глобальные индексы **A**, которыми владеет каждый P_{ID}.

Строка 6 возвращает диапазоны глобальных индексов для всех P_{ID}. Эта информация используется для вычисления глобальных индексов при каждой итерации p.

Строки 9–10 вычисляют глобальные индексы, которыми владеет $P_{ID} = p$.

Строки 11–12 устанавливают тэги сообщений, которые $P_{ID} = p$ будет использовать для отправки данных другим экземплярам P_{ID}.

Строки 13–20 выполняются только на $P_{ID} = p$. Эти строки выполняют LU-разложение для **A**.*loc* и отправляют результирующие $\mathbf{L}_p$ и $\mathbf{v}_p$ в адрес $P_{ID} > p$. Только $\mathbf{v}_p$ отправляется в адрес $P_{ID} < p$.

Строка 22 получает $\mathbf{L}_p$ и $\mathbf{v}_p$. Эта строка выполняется только на $P_{ID} > p$.

Строка 24 получает $\mathbf{v}_p$. Эта строка выполняется только на $P_{ID} < p$.

Строки 26–29 повсюду обновляют информацию о главных элементах и выполняют обмены соответствующих им строк **A**.*loc* на $P_{ID} \neq p$.

Строки 30–35 применяют преобразование с матрицей $\mathbf{L}_p$ к **A**.*loc* для всех $P_{ID} > p$.

Строки 37–41 используют **A**.*loc* для заполнения **L**.*loc* и **U**.*loc*.

Код, приведенный ниже, является прекрасной иллюстрацией того, как среда программирования на основе распределенных массивов может быть скомбинирована с методом обмена сообщениями для реализации программы, в которой различные P_{ID} выполняют различные вычисления. В частности, такие операции, как `global_ind` и `global_block_ranges` могут быть очень полезными для определения того, над какими строками и столбцами A следует работать на каждой итерации p.

Наиболее сложной частью параллельного кода LU-разложения являются пять множеств процессоров: все P_{ID}, $P_{ID} = p$, $P_{ID} > p$, $P_{ID} < p$ и $P_{ID} \neq p$. Каждое множество процессоров будет выполнять индивидуальную функцию по ходу работы алгоритма, которая будет явно выбираться при помощи оператора `if`.

Листинг кода 11.1. Параллельная MATLAB-программа LU-разложения

```
1 function [L,U,v] = pLUfactor(A)
2    N = size(A,2);  % A size.
3    Aloc = local(A);  % Local A.
4    Nloc = size(Aloc,2); % Aloc size.
5    i = global_ind(A,2);  % Local columns.
6    allJ = global_block_ranges(A,2); % All ranges.
7    v=(1:N)';  % Init pivots.
8    for p = 0:Np-1 % Loop over each PID.
9            ip = aIIJ(p+1,3):aIIJ(p+1,2); % p columns.
10           Nlocp = length(ip);  % Aloc size.
11           tagHigh = mod(2*p+1,32);  % PID>p message tags.
12           tagLow = mod(2*p,32);   % PID>p message tags.
13           if PID == P
14                   [Lp Up vp] = lu(Aloc(ip, :),'vector');  % LU factor.
15                   vp = vp+ip(1)-1;       % Update pivots.
16                   pHigh = (p+1):Np-1; pLow = 0:(p-1);   % Higher PIDs.
17                   SendMsg(pHigh,tagHigh,Lp,vp); % Send Lp, vp to PID>p.
18                   SendMsg(pLow,tagLow, vp);      % Send vp to PID<p.
19                   Lp(1:Nlocp,:) =  Lp(1:Nlocp, :)+(Up-eye(Nlocp,Nlocp));
20                   Aloc(ip(1):N,:) = Lp;  % Copy to Aloc.
21           elseif PID = P
22                   [Lp vp] = RecvMsg(p,tagHigh);% Receive Lp and vp.
```

```
23          else if PID < P
24                  vp = RecvMsg(p,tagLow);        % Receive vp.
25          end
26          v(ip(1):N) = v(vp);    % Update pivots.
27          if not(p = PID)
28                  Aloc(ip(1):N,:)        Aloc(vp, :); % Pivot rows.
29          end
30   if PID > p
31          iL = (ip(1)+1):size(Lp,1); % Lower rows of Lp.
32          iA = (ip(end)+1):N;    % Lower rows of Aloc.
33          Aloc(ip,:) = Lp(1:Nlocp, :)\Aloc(ip, :); % Solve.
34          Aloc(iA,:) = Aloc(iA, :)-Lp(iL, :)*Aloc(ip, :); % Update.
35   end
36   end
37   Lloc = tril(Aloc,-i(1));       % Get lower triangle.
38   Lloc(i,:) = Lloc(i,:)+eye(Nloc,Nloc); % Add identity.
39   L = put_local(A,Lloc);         % Put into distributed L.
40   Uloc = triu(Aloc,-(i(1)-1));   % Get upper triangle.
41   U = put_local(A,Uloc);         % Put into distributed U.
42 end
```

Перевод

2 % Размер матрицы A;
3 % Локальная часть A;
4 % Размер Aloc;
5 % Локальные столбцы;
6 % Все диапазоны;
7 % Иниц. № главн. эл.;
8 % Цикл по каждому P_{ID};
9 % Столбцы для p;
10 % Размер Aloc для p;
11% Тэг сообщения для $P_{ID} > p$;
12 % Тэг сообщения для $P_{ID} < p$
14 % LU-разложение ;
15 % Обновить № главн. эл.;
16 % Верхние P_{ID};
17 % Отправка Lp, vp в адрес $P_{ID} > p$;
18 % Отправка vp в адрес $P_{ID} < p$;
20 % Копировать в Aloc;
22 % Получить Lp и vp ;
24 % Получить vp;
26 % Обновить № главн. эл.;
28 % Обмены строк;

31 % Нижние строки Lp;
32 % Нижние строки Aloc;
33 % Решить;
34 % Обновить;
37 % Получить нижне-треугольную матрицу;
38 % Добавить единичную матрицу;
39 % Поместить в распределенную матрицу L;
40 % Получить верхне-треугольную матрицу;
41 % Поместить в распределенную матрицу U.

На данном примере кода можно проследить некоторые свойства подхода к программированию на основе карт размещения.

- Влияние на код. Преобразование программы из последовательной в параллельную с использованием комбинированного подхода приводит к получению размера кода, значительно большего, чем последовательный. Однако комбинированный код значительно меньше, чем при подходе с чистым обменом сообщениями, так как вся информация об индексах массивов предоставляется распределенными массивами.
- Последовательный код сохраняется. Код будет работать без изменений как в последовательном, так и в параллельном режиме.
- Средний объем использования параллельной библиотеки. В дополнение к обычным функциям распределенных массивов *Np* и P_{ID} программа также использует функции индексов распределенных массивов `global_ind` и `global_block_ranges`, а также основные функции обмена сообщениями `SendMsg` и `RecvMsg`. В общем, используется шесть функций параллельной библиотеки, что немного больше того, что было использовано в программе на основе чистых распределенных массивов.
- Масштабируемость. Код может выполняться на задачах любого размера *N* и любом количестве процессоров *Np*.
- Ограниченная коммуникация. Так как используются локальные переменные и прямой обмен сообщениями, использование коммуникации строго контролируется. Кроме того, комбинированный подход допускает шаблоны коммуникации гораздо более сложные, чем стандартно поддерживаемые при использовании только распределенных массивов.
- Зависимость от карт преобразования. Побочным эффектом использования комбинированного подхода является то, что код будет выполняться только с использованием карт блочного распределения.
- Гарантия последовательной производительности. Поскольку последовательные операции линейной алгебры применяются к локальным переменным, которые являются обычными числовыми массивами MATLAB, при выполнении данных строк кода гарантируется отсутствие каких-либо скрытых ухудшений производительности.

11.4. Отладка

Как упоминалось ранее, отладка параллельной программы представляет собой непростую задачу. Четко определенный процесс кодирования–отладки–тестирования является основным преимуществом подхода на основе карт размещения (см. таблицу 2.2). Комбинированный подход к кодированию требует модификации данного процесса.

Шаг 1. Выполнение программы последовательно на локальной машине с матрицей **A**, созданной как обычный массив (т. е. `PARALLEL = 0`). Так проверяется корректность исходного последовательного кода. Для работы этого последовательного кода необходимо предпринять ряд дополнительных шагов помимо тех, которые использовались вместе с кодом на основе чистых распределенных массивов. Во-первых, все функции индексов массивов должны обеспечить правильные результаты на нераспределенных массивах. Во-вторых, весь код сообщений написан так, что для $Np = 1$ он выполняться не будет. В листинге кода 11.1 переменные `pHigh` и `pLow` создаются так, что они будут пусты, если $Np = 1$, таким образом, соответствующий обмен сообщениями выполняться не будет.

Шаг 2. Выполняет программу последовательно на локальной машине с матрицей **A** в качестве распределенного массива (т. е. `PARALLEL = 1`) и проверяет правильность работы программы с распределенными массивами в качестве входных данных. Как и на шаге 1, код должен быть написан так, чтобы при $Np = 1$ обмен сообщениями не происходил.

Шаг 3. Выполняет программу параллельно на локальной машине. Данный шаг проверяет, являются ли параллельные карты, которые генерируют массивы **A** и **A**.*loc*, корректными. Что еще более важно, на этом шаге впервые выполняется код обмена сообщениями. Из всех шагов этот является самым важным, так как обмен сообщениями может способствовать появлению множества ошибок. При параллельном LU-разложении самые распространенные ошибки совершаются в ходе организации выполнения операций на пяти множествах процессоров. Реализации этих множеств процессоров при помощи операторов `if` могут легко привести к появлению неверного кода, выполняемого на неправильном множестве процессоров. К сожалению, единственным способом отладки такого типа кода является анализ вывода от всех P_{ID} для проверки выполнения правильного кода.

Шаг 4. Выполнение программы в параллельном режиме на удаленной машине.

Шаг 5. Выполнение программы в параллельном режиме на удаленной машине, но с задачей интересующего размера. До этого момента тестирование должно было осуществляться на задаче малого размера. Данный шаг позволяет определить производительность на реальной задаче.

Эти шаги являются существенным отражением этапов, указанных в разделе «Начало работы», и на каждом из шагов тестируются определенные аспекты параллельного кода. В дополнение к отладке функциональности программы вышеуказанный процесс также может быть использован для оптимизации производительности кода (см. следующий раздел). Тест производительности HPL использует некоторые дополнительные методики кодирования для отладки, помимо указанных выше шагов.

Программа HPL печатает параметры, описывающие размер задачи N, количество необходимых байтов и количество совершенных обновлений. Это осуществляется на каждом P_{ID} программы. Сравнение этих значений путем просмотра различных фай-

лов с результатами от каждого P_{ID} (т. е. `MatMPI/*.out`) является действенным средством выявления ошибок.

В программе HPL также отображается время размещения и запуска для каждого P_{ID}. Эти временные показатели описывают количество времени, затрачиваемое до начала выполнения программы. Признаком того, что в программе происходит нечто неправильное на ранних стадиях, является значительное отличие этих значений для разных P_{ID}.

Так как HPL является тестом производительности, производительность выполнения измеряется и выводится в единицах, имеющих отношение к процессору. Значения производительности, отличающиеся значительно между экземплярам и P_{ID}, могут стать показателем более глубоких проблем в программе (например, создание массивов слишком отличающихся размеров).

Наконец, программа HPL имеет проверку правильности, которая контролирует выдачу правильных результатов программой. В реальных приложениях такую проверку как правило произвести невозможно, однако можно выполнить приближенные проверки, являющиеся очень полезными. Например, часто можно проверить знак значений в массиве или сумму значений в массиве. Повторимся, что отображение этих значений для каждого P_{ID} может выявить ошибки, зависящие от P_{ID}.

11.5. Тестирование

Как станет видно далее, программа HPL является хорошим примером программы, производительность которой повышается с ростом числа *Np*. Фактически программа HPL предназначена для выявления пиковой вычислительной производительности в широком диапазоне вычислительной архитектуры. По этим причинам программу HPL стоит изучать. HPL представляет собой сложную параллельную программу, которая может эффективно работать в параллельном режиме, так как она характеризуется высоким отношением вычисление–коммуникация. Все конкурентоспособные реализации HPL используют сложные двумерные блочно-циклические распределения. В данной главе для рассмотрения было выбранное более простое одномерное блочное распределение, так как оно дает более простой LU-алгоритм. HPL является идеальной программой для изучения компромисса между производительностью и сложностью кода.

11.5.1. Производительность на многоядерной системе

Тестирование параллельной программы – при условии, что она уже отлажена и выдает корректные результаты, – в основном заключается в определении количества времени, которое требуется для ее выполнения, и выяснения, насколько ее производительность соответствует ожидаемой. Почти все данные о производительности, необходимые для проведения такого анализа, были представлены в разделе «Начало работы». В таблице 11.1 приведена многоядерная производительность программы HPL на восьмиядерной системе.

Таблица 11.1. Производительность программы HPL на многоядерной системе

Относительное время распределения, абсолютное время запуска, относительное время выполнения и относительная производительность для теста HPL с использованием одномерного блочного алгоритма для масштабированного размера задачи ($N^2 \propto N_P$) на 8-ядерном процессоре. Все значения нормализованы по столбцу 2, в котором $N = 1$ с выключенными распределенными массивами (PARALLEL = 0). Столбцы 3–6 содержат значения с включенными распределенными массивами (PARALLEL = 1). Запись 1*8 означает, что $Np = 8$ и все экземпляры P_{ID} были выполнены на одном процессорном узле с общей памятью

Многоядерные процессоры:	1	1*1	1*2	1*4	1*8
Относительное время размещения	1	1.0	1.33	1.92	3.13
Время запуске (с)	0	0	1.13	1.88	1.61
Относительное время выполнения	1	1.0	1.30	1.83	2.89
Относительная производительность (гигафлопс)	1	1.0	2.18	4.41	7.85

Первый тест проводится для сравнения производительности выполнения программы с использованием $Np = 1$ с выключенными распределенными массивами (PARALLEL = 0) и с включенными распределенными массивами (PARALLEL = 1). В столбцах 2 и 3 таблицы 11.1 приведены эти данные. В данном случае разница в производительности пренебрежимо мала и означает, что распределенные массивы не снижают производительности. Если использование распределенных массивов с $Np = 1$ значительно замедляет работу, необходимо решить данную проблему. Очень трудно получить какое-либо преимущество при $Np > 1$, если скорость при $Np = 1$ ниже, чем исходный последовательный код. Работа только на локальных частях распределенных массивов практически гарантирует, что производительность будет равна производительности при работе без использования распределенных массивов.

Столбцы 3–5 таблицы 11.1 содержат параллельную производительность кода на многоядерной системе. Строки относительного времени размещения показывают, что этот параметр увеличивается с ростом числа Np, так как для каждого P_{ID} выделяется один и тот же объем данных вместе с некоторыми дополнительными структурами данных, которые содержатся внутри распределенного массива. Обычно время размещения не является большей частью общего времени вычислений. Однако при больших значениях Np накладные расходы могут стать значительными; таким образом, рекомендуется измерять время распределения и проверять его приемлемость.

Строки времени запуска показывают общее увеличение времени запуска с ростом числа Np. В идеале это должна быть постоянная величина, не зависящая от Np. Однако в зависимости от используемого механизма запуска этот показатель времени может быть увеличивающейся функцией от Np. Как правило время запуска не составляет большую часть от общего времени вычисления, но размер его может легко стать значительным при больших Np. Например, предположим, что время запуска пропорционально Np, и каждый дополнительный P_{ID} добавляет одну секунду. Если $Np = 100$, время запуска составит 100 секунд. Если это является долей Амдала программы, то программе понадобится для выполнения не менее Np^2 секунд для достижения 50-кратного ускорения на последовательном коде.

Наиболее интересные строки – это относительное время выполнения и относительная производительность в гигафлопсах. Так как $N^2 \alpha Np$, относительное время выполнения увеличивается с ростом Np даже при увеличении производительности вместе с Np. Как и ожидалось, почти линейное ускорение наблюдается на программе HPL даже при использовании упрощенного параллельного LU-алгоритма.

11.5.2. Производительность на многоузловой системе

Результаты выполнения LU-алгоритма на многоузловой системе показаны в таблице 11.2.

Таблица 11.2. Производительность программы HPL на многоузловой системе

Относительное время размещения, абсолютное время запуска, относительное время выполнения и относительная производительность для теста HPL с использованием одномерного блочного алгоритма для масштабированного размера задачи ($N^2 \alpha Np$) на 16-узловой системе. Все значения нормализованы по столбцу 2, в котором $Np = 1$ с выключенными распределенными массивами (PARALLEL = 0). Столбцы 3–7 содержат значения с включенными распределенными массивами (PARALLEL = 1). Запись 16*1 означает, что $Np = 16$ и каждый P_{ID} был выполнен на своем индивидуальном процессорном узле с использованием отдельной памяти

Многоузловые процессоры:	1	1*1	2*1	4*1	8*1	16*1
Относительное время размещения	1	1.0	1.40	2.02	3.03	4.1
Время запуска (с)	0	0	10.1	8.98	13.3	14.9
Относительное время выполнения	1	1.0	2.33	3.72	5.77	9.34
Относительная производительность (гигафлопс)	1	1.0	1.20	2.14	3.91	6.83

Относительное время размещения для многоядерной и многоузловой систем практически одинаково. Время запуска на многоузловой системе склонно к увеличению с увеличением Np и демонстрирует большие отклонения. Более того, время запуска гораздо больше, чем в случае многоядерной системы, так как для запуска экземпляров P_{ID} отдельные системы должны связываться по сети.

Относительная производительность увеличивается с ростом числа Np для многоядерной и многоузловой систем. Увеличение производительности для многоядерной системы несколько больше, так как пропускная способность коммуникации многоядерных процессоров несколько выше, чем у многоузловых. Пропускная способность по большей части влияет на производительность параллельного LU-алгоритма для небольших значений Np. Анализ производительности критического пути алгоритма показывает, что с ростом числа Np пропускная способность становится менее значительной, объясняя тем самым увеличивающийся наклон кривой производительности для многоузловой системы (см. рисунок 11.3).

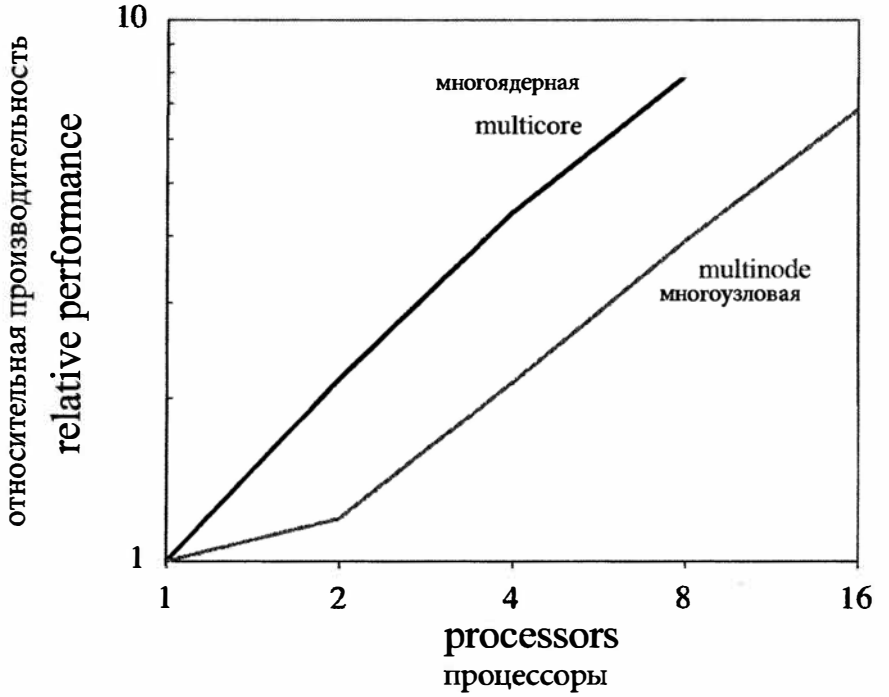


Рисунок 11.3. Относительная производительность параллельной программы HPL
Относительная производительность параллельной программы HPL на многоядерных и многоузловых системах

Список литературы

1. *[Bliss & Kepner 2007] N. Bliss and J. Kepner, pMATLAB parallel MATLAB library, Special Issue on High Productivity Programming Languages and Models, Inter-national Journal of High Performance Computing Applications, Vol. 21, No. 3, pp. 336-359, 2007.*
2. *[Demmel 2001] J. Demmel, Dense linear algebra, CS 267 Applications of Parallel Computers class notes, University of California Berkeley, 2001, available online athttp://www.cs.bcrkclcy.edu/-demmel.*
3. *[Dongarra and Luszczek 2005] J. Dongarra and P. Luszczek, Introduction to the HPC Challenge Benchmark Suite, Technical Report UT-CS-05-544, University of Tennessee, 2005.*

Приложение

Система обозначений для иерархических параллельных алгоритмов для многоядерных систем

Резюме

Увеличение сложности параллельных многоядерных процессоров вызывает необходимость использования соответствующих сложных параллельных алгоритмов. Эти алгоритмы часто используют иерархические шаблоны доступа к данным и шаблоны конвейерного выполнения. Чтобы упростить рассмотрение этих алгоритмов, вводится математическая система обозначений на основе распределенных массивов. Эта запись расширяет диапазон существующей записи распределенных массивов с целью охватить иерархические массивы и допускает конструирование алгоритмов, характеризующихся высокой степенью локальности данных. Кроме того, здесь представлены конвейерные конструкции, состоящие из «задач» и «каналов».

А.1. Введение

Эффективное использование параллельных многоядерных процессоров требует разработки комплексных алгоритмов, которые обеспечивают максимальную локальность данных и минимизируют коммуникацию. Эти алгоритмы часто используют иерархические описания данных и конвейерное исполнение. Описание этих алгоритмов представляет трудность, однако эту задачу можно упростить, используя соответствующие обозначения. Такая система обозначений представлена здесь вниманию читателя.

Используя предложенную систему обозначений, можно описать широкий спектр параллельных алгоритмов и шаблонов доступа к данным. Далее на основе этих алгоритмов можно получить требования к программному обеспечению для поддержания любого конкретного класса алгоритмов. Многие из этих алгоритмов могут быть поддержаны существующими технологиями, такими как PVL, pMatlab, VSIPL++, pMatlabXVM, ROSA и PVTOL. Однако ни одна технология не может поддерживать все алгоритмы, которые можно представить с использованием данной системы обозначений.

Ключевым начальным шагом при разработке параллельного алгоритма является определение множеств операций, которые можно выполнить параллельно. Более точно, параллельные операции представляют собой множества операций, которые не зависят от того, в каком порядке их выполняют. К счастью, выявление параллелизма в программах обработки сигналов является достаточно прямолинейным процессом, так как большинство программ основаны на использовании распределенных массивов, которые по своей природе содержат параллелизм. Большинство операций над од

ним или несколькими массивами, в результате которых создается новый массив, могут выполняться параллельно по всем элементам в выходном массиве. Фактически параллелизм в большинстве программ обработки сигналов носит повсеместный характер, что делает написание программ, выполняемых параллельно, несложным. Что действительно представляет сложность при написании параллельных программ, так это написание программ, которые выполняются быстро. Ключевым шагом к написанию такого рода программ является выявление параллелизма в программе, который также приводит к высокой степени локальности. Такие программы способны выполнять параллельные операции с минимальным количеством перемещений данных.

Модель программирования на основе распределенных массивов является уникальной среди всех моделей программирования благодаря тому, что локальность идентифицируется с самого начала простым действием по созданию распределенного массива. Затем из локальности можно получить параллелизм в задаче, что приводит к созданию эффективной параллельной программы. Напротив, другие модели параллельного программирования обычно вынуждают программиста выявлять параллелизм, а затем иметь дело с более сложной задачей локальности во время реализации.

Введение в систему обозначений представлено в контексте конкретного примера. В раздел А.2 приводятся примеры исходного последовательного алгоритма и множества параллельных алгоритмов. В разделе А.3 даны параллельно-конвейерные конструкции.

А.2. Параллелизм по данным

Параллелизм по данным предполагает создание массива данных и выполнение различными процессорами одного и того же кода на своих «собственных» частях массива. Такая схема параллельного выполнения часто носит название модели Single-Program Multiple-Data (SPMD) (одна программа, несколько потоков данных, или архитектура SPMD). [Flynn 1972]. Такой принцип работы с данными часто называется моделью «вычисляет владелец».

В первом подразделе описывается простой последовательный двумерный фильтрующий алгоритм. Второй подраздел представляет основные обозначения распределенных массивов, которые позволяет этому алгоритму использовать параллелизм по данным. Третий подраздел описывает немного более сложную версию алгоритма, который в дальнейшем разбивает данные на особые блоки на локальных процессорах. Четвертый подраздел идет на шаг дальше и описывает алгоритм с использованием иерархического распределенного массива. Последний подраздел объединяет формирование блоков и иерархию.

А.2.1. Последовательный алгоритм

Пример представляет собой базовый двумерный фильтрующий алгоритм, как правило встречающийся во многих контекстах, связанных с обработкой сигналов. Алгоритм состоит из двух этапов. Первый этап фильтрует строки комплексной матрицы размеров $N \times M$. Второй этап фильтрует столбцы матрицы. Последовательное описание алгоритма приведено в Алгоритме А.1.

Алгоритм А.1. Последовательный алгоритм 2D-фильтра

Этап 1 – это частотный фильтр строк матрицы. Второй этап выполняет БПФ столбцов.

$\mathbf{Y} : \mathbb{C}^{N \times M} = \textsc{FilterSerial}(\mathbf{X} : \mathbb{C}^{N \times M})$

1 $\mathbf{c} : \mathbb{C}^{M}$
2 **for** $i = 1 : N$
3 **do** ▷ Stage 1.
4 $\mathbf{X}(i, :) = \mathrm{IFFT}(\mathrm{FFT}(\mathbf{X}(i, :)) \,.*\, \mathbf{c})$
5 $\mathbf{Y} = \mathbf{X}$ ▷ Copy data.
6 **for** $j = 1 : M$
7 **do** ▷ Stage 2.
8 $\mathbf{Y}(:, j) = \mathrm{FFT}(\mathbf{Y}(j, :))$

Перевод
3 Этап 1;
5 Копировать данные;
7 Этап 2.

Вышеуказанный алгоритм очень схож со многими полезными вычислениями. Как указано, этап 1 идентичен с тестом производительности HPEC (High Performance Embedded Computing – высокопроизводительная встроенная вычислительная обработка) Challenge FDFIR (Frequency Domain Finite Impulse Response – фильтрация с конечной импульсной характеристикой частотного диапазона, см. http://www.ll.mit.edu/HPECchallenge). Исключение IFFT (Inverse Fast Fourier Transform – обратное быстрое преобразование Фурье) из этапа 1 преобразует код в тест производительности HPC (High Performance Computing – высокопроизводительные вычисления) Challenge FFT (см. http://www.hpcchallenge.org). Исключение IFFT и умножения на вектор коэффициентов **c** приведет к простому двумерному БПФ, которое широко используется во многих сферах.

Как было упомянуто ранее, для достижения эффективной производительности важно понимать, где лежит массив в физической памяти (его локальность). С точки зрения памяти, физическое расположение массива может быть изображено при помощи диаграммы Кека (см. рисунок А.1).

А.2.2. Параллельный алгоритм

Для описания параллельных алгоритмов необходимо ввести дополнительные обозначения. В частности, количество процессоров, используемых при вычислении, обозначается N_p. Когда алгоритм выполняется в параллельном режиме, это означает, что один и тот же алгоритм (или код) выполняется на каждом процессоре. Такая вычислительная модель получила название SPMD (одна программа, несколько потоков данных, или архитектура SPMD) [Flynn 1972]. Чтобы различать N_p экземпляров программы, каждой из них присваивается уникальный идентификационный процессорный номер – P_{id}, – который изменяется от 0 до $Np - 1$.

При программировании на основе распределенных массивов необходимо распре-
делять элементы массива на множестве процессоров. Такой процесс распределения
описывает индексы распределенного массива, которым «обладает» каждый процес-
сор. «Р-запись» используется в качестве удобного краткого условного обозначения
для описания данного процесса распределения [Choy & Edelman 1999]. Р-запись бе-
рет обычную матрицу и преобразует ее в параллельную матрицу, применяя карту рас-
пределения. Матрица, распределенная так, что каждый процессор хранит блок строк,
обозначается следующим образом:

$$\mathbf{A} : R^{P(N)\times N}.$$

$$\mathbf{A} : R^{N\times N}$$

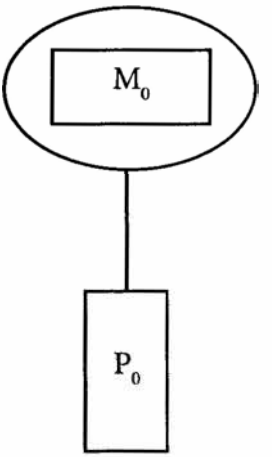


Рисунок А.1. Схема Кека для одного процессора

Схема Кека для последовательного массива на системе с одним процессором. Нижние индексы означают уровень иерархии памяти (P_{ID} при необходимости может быть обозначен верхним индексом). На рисунке изображена система с одноуровневым процессором/памятью. Прямоугольник в овале означает, что массив **A** находится в памяти, обозначенной M_0

«Картой» в вышеуказанной строке является $P()\times$. Аналогичным образом матрица, которая распределена таким образом, что каждый процессор хранит блок столбцов, задается как

$$\mathbf{A} : R^{N\times P(N)}.$$

«Картой» в вышеуказанной строке является $\times P()$. Декомпозиция вдоль как строк, так и столбцов, может быть записана как

$$\mathbf{A} : R^{P(N)\times P(N)} \text{ или } \mathbf{A} : R^{P(N\times N)}.$$

Эти две карты эквивалентны; другими словами, $P()\times P() = P(\times)$. В данном случае также может понадобиться описать процессорную «сетку» (например, $Np/4\times 4$). Эти виды распределений показаны на рисунке А.2.

Доступ к определенному элементу данных в распределенном массиве задается обычным обозначением подмассива. Если $\mathbf{A} : \mathrm{R}^{P(N)\times N}$, то запись $\mathbf{A}(i, j)$ заставит процессор, который обладает элементом с индексами i, j матрицы **A**, отправить данное значение всем другим процессорам. Точно также, при заданных двух матрицах с различными картами распределения $\mathbf{A} : \mathrm{R}^{P(N)\times N}$ и $\mathbf{B} : \mathrm{R}^{P(N)\times N}$ выражение

$$\mathbf{B} = \mathbf{A},$$

вызовет перераспределение данных из **A** в новое распределение матрицы **B**.

Доступ только к локальной части распределенного массива обозначается с помощью дополнения *.loc*. Для $\mathbf{A} : \mathrm{R}^{P(N)\times N}$ локальной частью является $\mathbf{A}.loc : \mathrm{R}^{(N/Np)\times N}$. Данное обозначение очень полезно при определении операций, являющихся полностью локальными для каждого процессора и не требующих взаимодействия. При использовании этих обозначений простая параллельная по данным реализация примера с двумерным фильтрующим алгоритмом может быть записана, как показано в Алгоритме 1.2.

Алгоритм А.2. Параллельный алгоритм 2D-фильтра

Каждый этап приложения изменен для работы только над теми данными, которые являются локальными для каждого процессора.

```
Y : C^{N×P(M)} = FilterParallel(X : C^{P(N)×M})
1  c : C^M
2  for i = 1 : size(X.loc, 1)
3        do          ▷ Stage 1.
4             X.loc(i, :) = IFFT(FFT(X.loc(i, :)) .* c)
5  Y = X             ▷ Cornerturn data.
6  for j = 1 : size(Y.loc, 2)
7        do          ▷ Stage 2.
8             Y.loc(:, j) = FFT(Y.loc(j, :))
```

Перевод

3 Этап 1;

5 Перераспределение с поворотом угла;

7 Этап 2.

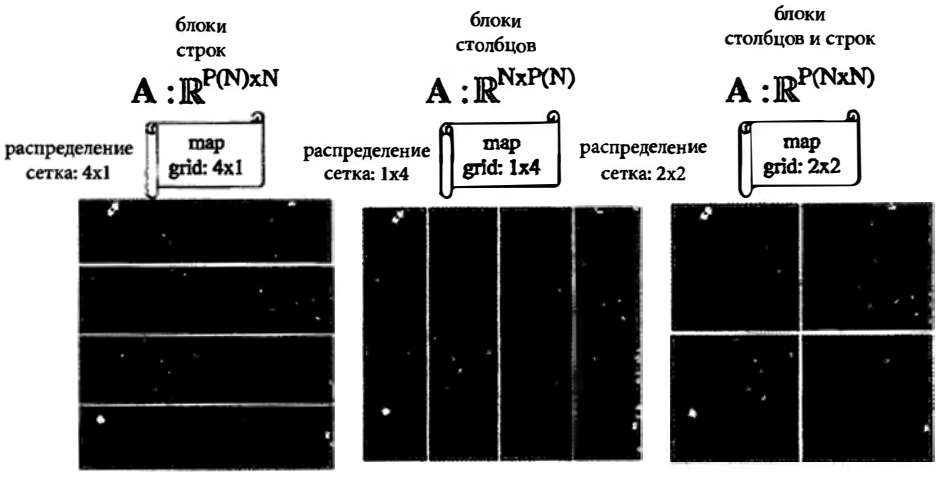


Рисунок А.2. Параллельные карты распределения
Набор карт распределений, поддерживаемых типичной средой программирования на основе распределенных массивов

Ссылки с определенным индексом в локальной части даются обычным способом: **A.** *loc*(*i, j*). Как правило глобальный индекс **A**(*i, j*) отличается в массиве от локального индекса **A.** *loc*(*i, j*). Если необходимо получить локальную часть от определенного процессора, ее можно обозначить как **A.** loc_p. Наконец, часто требуется получить глобальный индекс определенного локального значения. Он обозначается с использованием **A.** *loc*(*i, j*). Очевидно, (*i,j*) = **A**(*i,j*). В более общем виде следующее равенство будет верным:

$$\mathbf{A}(\mathbf{A}.\ loc(i, j)) = \mathbf{A}.\ loc(i, j).$$

С точки зрения параллельной памяти, физическое расположение распределенного массива может быть изображено при помощи схемы Кека (см. рисунок А.3).

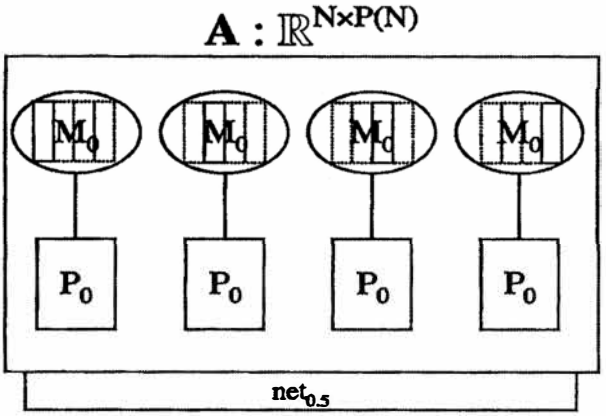


Рисунок А.3. Схема Кека для параллельных процессоров
Схема Кека для распределенного массива на системе параллельных процессоров. Нижние индексы означают уровень иерархии памяти (`Pid` при необходимости может быть обозначен верхним индексом). $net_{0.5}$ означает сеть, которая соединяет различные устройства памяти на нулевом уровне. Пунктирный прямоугольник в каждом овале означает весь массив А. Сплошной прямоугольник в каждом овале изображает локальную часть массива **А**. *loc*, находящуюся в локальной памяти каждого процессора

А.2.3. Блочный параллельный алгоритм

Упомянутая ранее запись *P()* неявно подразумевала, что данные разбиваются на логически смежные блоки данных. Это можно выразить явно с использованием обозначения $P_b()$, которое различает блочное распределение и циклическое распределение, обозначаемое $P_c()$. В циклическом распределении элементы данных, которыми владеет определенный процессор, определяются таким образом:

$$(i-1+P_{ID}) \mod Np, \quad i = 1 : N.$$

Обобщение такого распределения должно распределять элементы в смежных подблоках. Такое блочно-циклическое распределение обозначается $P_{c(n)}()$, где N – это размер подблоков. Ссылка на k-й локальный подблок на процессоре обозначается как $A.loc.blk_k$ Циклические распределения часто используются для восстановления равномерности нагрузки (т. е. для равномерного распределения работы на множество процессоров). Блочно-циклические распределения часто используются для оптимизации производительности алгоритма для иерархии памяти. Работая по алгоритму при помощи блоков, обычно можно переместить данные в процессор, выполнить ряд операций над этим блоком и затем переместить данные из процессора. Используя данную

запись, простую блочно-циклическую параллельную реализацию примера с двумерным фильтрующим алгоритмом можно записать, как показано в Алгоритме А.3.

Алгоритм А.3. Блочный параллельный алгоритм 2D-фильтра

Каждый этап приложения изменен для работы над одним блоком в каждый момент времени.

$\mathbf{Y} : \mathbb{C}^{N \times P_{c(n)}(M)} = \textsc{FilterBlockParallel}(\mathbf{X} : \mathbb{C}^{P_{c(n)}(N) \times M})$

1 $\mathbf{c} : \mathbb{C}^{M}$
2 **for** $k = 1 : \mathbf{X}.loc.N_{blk},\ i = 1 : \textsc{size}(\mathbf{X}.loc.blk_k, 1)$
3 **do** ▷ Stage 1.
4 $\mathbf{X}.loc.blk_k(i,:) = \mathrm{IFFT}(\mathrm{FFT}(\mathbf{X}.loc.blk_k(i,:)) .* \mathbf{c})$
5 $\mathbf{Y} = \mathbf{X}$ ▷ Cornerturn data.
6 **for** $k = 1 : \mathbf{Y}.loc.N_{blk},\ k = 1 : \textsc{size}(\mathbf{Y}.loc.blk_k, 2)$
7 **do** ▷ Stage 2.
8 $\mathbf{Y}.loc.blk_k(:,j) = \mathrm{FFT}(\mathbf{Y}.loc.blk_k(:,j))$

Перевод
3 Этап 1;
5 Переразмещение с поворотом угла;
7 Этап 2.

А.2.4. Иерархический параллельный алгоритм

В предыдущих подразделах была представлена система обозначений для описания канонического «плоского» распределенного массива. Такая запись может использоваться для описания любого N-мерного блочно-циклического распределения данных. Многоядерные процессоры со сложными иерархиями памяти часто требуют алгоритмов, которые выходят за пределы того, что может быть описано при помощи этих обозначений. Рассмотрим случай, где каждый P_{ID} в наборе процессоров P несет ответственность за набор сопроцессоров $\bar{P}$, каждый из которых идентифицируется соответствующим $\bar{P}_{ID}$, находящимся в пределах от 0 до $N_{\bar{P}}$. Распределение массива по иерархическому набору процессоров требует расширения системы обозначений следующим образом. Матрица, которая распределена таким образом, что каждый P_{ID} имеет блок строк и в свою очередь каждый P_{ID} имеет блок строк, обозначается как

$$\mathbf{A} : \mathrm{R}^{P(P(N)) \times N}.$$

«Картой» в вышеуказанной строке является $P(\bar{P}()) \times$. Аналогичным образом матрица, которая распределяется иерархически по столбцам, обозначается как

$$\mathbf{A} : \mathrm{R}^{N \times P(P(N))}.$$

«Картой» в вышеуказанном выражении является $\times P(\bar{P}())$. $P()$ и $\bar{P}()$ не обязательно применять к одинаковым измерениям. Матрица, которая распределена таким об-

разом, что каждый P_{ID} хранит блок строк и в свою очередь каждый $\bar{P}_{ID}$ хранит блок столбцов из этого блока строк, обозначается как

$$\mathbf{A} : \mathrm{R}^{P(N)\times P(N)}.$$

«Картой» в вышеуказанном выражении опять является $P()\times\bar{P}()$.

Ссылка на $\bar{p}$ -тый локальный подблок на процессоре обозначается $\mathbf{A}.loc.blk_{\bar{p}}$. Операции на иерархическом массиве неявно выполняются с процессорами на самом нижнем уровне иерархии (в данном случае процессоры из множества $\bar{p}$). Если при этом требуется передача данных в другое адресное пространство, это происходит неявно. Первый доступ $\mathbf{X}.loc.loc$ приведет к тому, что данные будут загружены в память процессоров из множества $\bar{p}$. При необходимости это может быть сделано явно в коде. Используя эти обозначения, двумерный фильтрующий алгоритм можно записать иерархически, как приведено в Алгоритме А.4.

Алгоритм А.4. Иерархический параллельный алгоритм 2D-фильтра

Распределенные массивы по двум уровням процессоров P и $\bar{P}$.

$\mathbf{Y} : \mathbb{C}^{N\times P(\bar{P}_{c(n)}(M))} = \textsc{FilterHierParallel}(\mathbf{X} : \mathbb{C}^{P(\bar{P}_{c(n)}(N))\times M})$

1 $\mathbf{c} : \mathbb{C}^{M}$
2 **for** $i = 1 : \textsc{size}(\mathbf{X}.loc.loc, 1)$
3 **do** ▷ Stage 1.
4 $\mathbf{X}.loc.loc(i,:) = \mathrm{IFFT}(\mathrm{FFT}(\mathbf{X}.loc.loc(i,:)) .* \mathbf{c})$
5 $\mathbf{Y} = \mathbf{X}$ ▷ Cornerturn data.
6 **for** $j = 1 : \textsc{size}(\mathbf{Y}.loc.loc, 2)$
7 **do** ▷ Stage 2.
8 $\mathbf{Y}.loc.loc(:,j) = \mathrm{FFT}(\mathbf{Y}.loc.loc(:,j))$

Перевод
3 Этап 1;
5 Переразмещение с поворотом угла;
7 Этап 2.

С точки зрения памяти, физическое расположение иерархического распределенного массива может быть изображено при помощи схемы Кека (см. рисунок А.4).

А.2.5. Иерархический блочный параллельный алгоритм

Иерархия и разбиение на блоки часто комбинируются с целью учета ограничений иерархии памяти. В частности, когда локальные запоминающие устройства процессоров множества $\bar{P}$ малы и не могут вместить все части $\mathbf{A}.loc.loc$, тогда $\bar{P}$ может быть в дальнейшем разбит на блоки. Это обозначается так:

$$\mathbf{A} : \mathbb{R}^{P(\bar{P}_{b(n)}(N)) \times N}.$$

$$\mathbf{A} : \mathbb{R}^{N \times P(\bar{P}(N))}$$

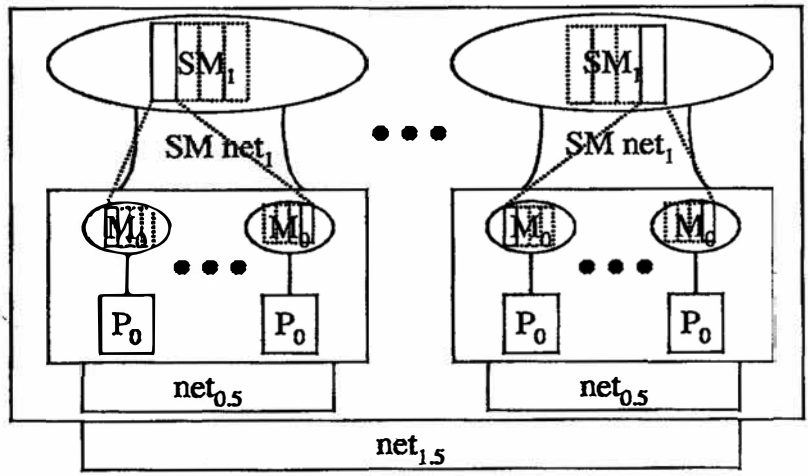


Рисунок А.4. Схема Кека для иерархического параллельного процессора
Схема Кека для иерархического распределенного массива на системе иерархических параллельных процессоров. Нижние индексы означают уровень иерархии памяти (P_{ID} при необходимости может быть обозначен верхним индексом). $net_{0.5}$ означает сеть, которая соединяет различные устройства памяти на 0-м уровне. $SM\ net_1$ означает сеть, которая соединяет процессоры на 0-м уровне с совместно используемой памятью на 1-м уровне. $net_{1.5}$ соединяет устройства памяти на 1-м уровне. **A**, **A**.*loc* и **A**.*loc.loc* изображены соответствующими сплошными и пунктирными прямоугольниками в овале

K-й блок на субпроцессоре $\bar{p}$ на процессоре *p* обозначается как $\mathbf{A}.loc_p.locblk_k$. Это допускает перемещение данных в память множества процессоров $\bar{p}$ блоком такого размера, который в ней может уместиться. С использованием этих обозначений двумерный фильтрующий алгоритм можно переписать, как показано в Алгоритме А.5.

При необходимости массив может быть объявлен массивом только для чтения посредством объявления:

$$\mathrm{A} :_{\Rightarrow} \mathrm{R}^{N \times M}$$

или только для записи с использованием:

$$\mathrm{A} : \mathrm{R}_{\Rightarrow}^{N \times M}.$$

Первый доступ к блоку проведет к загрузке данных в память множества процессоров $\bar{P}$. Обычно чтение следующего блока повлечет за собой запись предыдущего блока обратно в память. Однако если этот массив предназначен только для чтения, теку-

щий блок будет стерт. Аналогично, если массив предназначен только для записи, доступ к следующему блоку не считывает блок, а запишет текущий блок.

Алгоритм А.5. Иерархический блочный параллельный алгоритм 2D-фильтра

Распределенный массив разбивается на меньшие блоки, которые помещаются в памяти каждого P_{ID}.

$\mathbf{Y} : \mathbb{C}^{N \times P(\bar{P}_{b(n)}(M))} = \text{FILTERHIERBLOCKPARALLEL}(\mathbf{X} : \mathbb{C}^{P(\bar{P}_{b(n)}(N)) \times M})$

1 $\mathbf{c} : \mathbb{C}^{M}$

2 for $k = 1 : \mathbf{X}.loc.N_{blk}$, $i = 1 : \text{SIZE}(\mathbf{X}.loc.loc, 1)$

3 do ▷ Stage 1.

4 $\mathbf{X}.loc.loc.blk_k(i, :) = \text{IFFT}(\text{FFT}(\mathbf{X}.loc.loc.blk_k(i, :)) .* \mathbf{c})$

5 $\mathbf{Y} = \mathbf{X}$ ▷ Cornerturn data.

6 for $k = 1 : \mathbf{Y}.loc.N_{blk}$, $j = 1 : \text{SIZE}(\mathbf{Y}.loc.loc, 2)$

7 do ▷ Stage 2.

8 $\mathbf{Y}.loc.loc.blk_k(:, j) = \text{FFT}(\mathbf{Y}..loc.loc.blk_k(:, j))$

Перевод

3 Этап 1;

5 Переразмещение с поворотом угла;

7 Этап 2.

А.3. Конвейерный параллелизм

Конвейерный параллелизм часто используется в многоступенчатых алгоритмах с множеством наборов входных данных. Это позволяет одному набору процессоров выполнить одну ступень алгоритма и затем направить данные следующему набору процессоров для выполнения следующей ступени алгоритма. Таким образом, в любой момент времени разные наборы процессоров работают на разных экземплярах данных. Модель SPMD обычно используется в рамках каждого набора процессоров, работающего на определенной ступени алгоритма. Основная выгода конвейерных алгоритмов видна, когда требуется перемещение данных между ступенями (например, переразмещение матрицы с поворотом угла). В данной ситуации можно скрыть время этого взаимодействия путем одновременного выполнения вычислений и взаимодействия.

В первом подразделе описывается параллельно-конвейерная реализация двумерного фильтрующего алгоритма с использованием неявного метода путем распределения массивов по различным наборам процессоров. Второй подраздел представляет концепции «задач» и «каналов», которые явно поддерживают ковейерность. Третий подраздел описывает использование конвейерного метода на уровне более мелкой ячеистости. Четвертый подраздел иллюстрирует использование конвейерных конструкций для активизации обобщенного исполнения вычислительных ядер на ие-

A.3.1. Неявный конвейерный параллелизм

Параллельные конвейеры могут быть настроены неявно путем создания распределенного массива с множеством процессоров:

$$A : R^{P^{S1}(N)\times N},$$

где $S1$ – множество экземпляров P_{ID}. Например, чтобы распределить **A** на первой половине процессоров в множестве P, $S1$ будет определяться как $S1 = \{0, \ldots [Np/2]\}$. В этом случае **A** существует на всех процессорах, но хранит данные только на процессорах в множестве $S1$, т. е. $A.loc_{S1} = 0$. Используя правило «вычисляет владелец», любой процессор, который не владеет какими-либо данными в массиве, пропустит операцию и перейдет к следующему выражению в алгоритме.

Распределение массива по второй половине P обозначается как

$$B : R^{N\times P^{S2}(N)},$$

где $S2 = \{[Np/2],\ldots,Np\}$. В выражении присвоения от одного набора процессоров другому набору процессоров

$$\mathbf{B} = \mathbf{A}$$

процессоры в $S1$ отправят свои данные соответствующим процессорам в $S2$ и затем перейдут к следующим действиям. Аналогично процессоры из $S2$ перед переходом к следующим действиям будут ждать до тех пор, пока не получат данные от $S1$. Используя наборы процессоров, можно преобразовать двумерный фильтрующий алгоритм в параллельно-конвейерный алгоритм (см. Алгоритм A.6).

Алгоритм A.6. Неявный конвейерный алгоритм 2D-фильтра

Конвейерность имеет место, когда распределенные массивы назначаются различным множествам процессоров.

$\mathbf{Y} : \mathbb{C}^{N\times P^{S2}(M)} = \textsc{FilterImplicitPipe}(\mathbf{X} : \mathbb{C}^{P^{S1}(N)\times M})$

1 $\mathbf{c} : \mathbb{C}^{M}$
2 **for** $i = 1 : \textsc{size}(\mathbf{X}.loc, 1)$
3 **do** ▷ Stage 1 on processor set 1.
4 $\mathbf{X}.loc(i,:) = \mathrm{IFFT}(\mathrm{FFT}(\mathbf{X}.loc(i,:)) .* \mathbf{c})$
5 $\mathbf{Y} = \mathbf{X}$ ▷ Send from set 1 to set 2.
6 **for** $j = 1 : \textsc{size}(\mathbf{Y}.loc, 2)$
7 **do** ▷ Stage 2 on processor set 2.
8 $\mathbf{Y}.loc(:,j) = \mathrm{FFT}(\mathbf{Y}.loc(j,:))$

Перевод

3 Этап 1 на множестве процессоров 1;

5 Отправить данные от множества 1 к множеству 2;

7 Этап 2 на множестве процессоров 2.

А.3.2. Конвейерный параллелизм по задачам

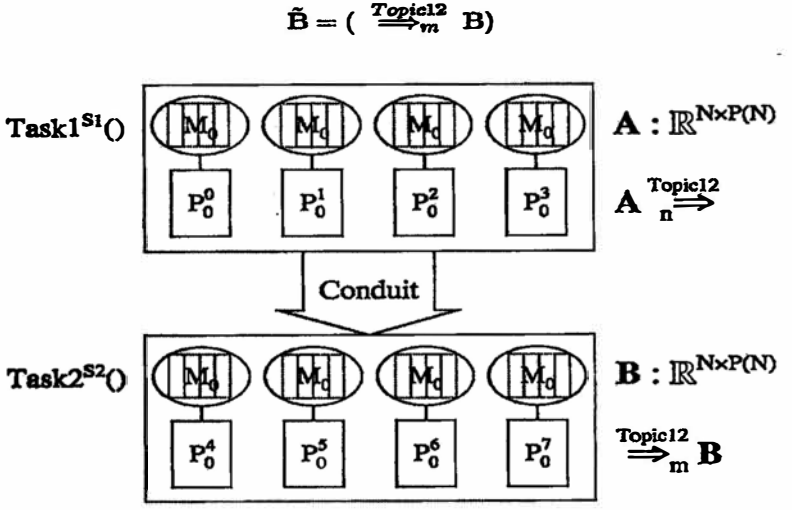
Наборы процессоров могут использоваться для неявной установки простых параллельных конвейеров. Однако обстоятельства часто требуют использовать более сложные конвейеры (например, обрабатывающие этапы карусельной диспетчеризации, конвейеры с использованием нескольких буферов и т. д.) «Задачи» и «каналы» представлены для описания этих более сложных случаев. Говоря более простыми словами, задачи – это просто «прямоугольники» на сигнальном ориентированном графе, а каналы – это «стрелки». В частности, задача создает подпрограмму SPMD, распределенную по особым наборам процессоров. В рамках этой программы все дальнейшие карты задаются относительно родительского набора процессоров.

На рисунке А.5 приведены основные обозначения задачи и канала и их влияние на ход выполнения программы. Задача обозначается посредством проставления у функции верхнего индекса с множеством процессоров, на которых должна выполняться эта функция, как то $\text{TASK}^{S1}()$, где $S1 = \{0, 1, 2, 3\}$. Канал создается между двумя задачами в первую очередь посредством назначения одной задачи в качестве записывающей для канала с использованием конструкции:

$$\overline{\mathrm{A}} = (A_n^{T_{\Rightarrow}^{optic12}}),$$

где **A** – распределенный массив, определяющий размер, тип и карту входных данных для канала, «Topic12» – это идентификатор канала, N означает количество буферов записи и $\tilde{\mathbf{A}}$ – первый доступный буфер записи от канала. Читатель канала объявляется с использованием соответствующей конструкции:

$$\overline{\mathrm{B}} = (T_{n\Rightarrow}^{optic12}\,\mathrm{B}),$$



Перевод
Task – задача;
Conduit – канал.

Рисунок А.5. Две задачи, соединенные каналом

Задача Task1 распределяется на множество процессоров $S1 = \{0, 1, 2, 3\}$ и содержит распределенный массив **A**. Задача Task2 распределяется на множество процессоров $S2 = \{4, 5, 6, 7\}$ и содержит распределенный массив **B**. Данные из **A** передаются в B посредством канала, обозначенного «Topic12»

Алгоритм А.7 является двумерным фильтрующим алгоритмом, переписанным с использованием задач и каналов. Он разбивается на две отдельные функции: FILTERSTAGE1 и FILTERSTAGE2. Функции преобразуются в задачи путем выполнения их на определенных множествах процессоров через запись FILTERSTAGE1^{S1}() и FILTERSTAGE2^{S2}(). Внутри этих функций распределение выполняется относительно этих множеств процессоров. После вызова этих функций задачи находятся вне контекста SPMD друг друга. Ни функция FILTERSTAGE1, ни функция FILTERSTAGE2 не осведомлена о том, что происходит внутри другой.

Прохождение данных между задачами осуществляется через конструкцию канала $\Longrightarrow$. Чтобы использовать канал, задача должна сначала подписаться на «topic (тему)» канала. Задача может либо записывать, либо считывать из канала. Чтобы соединить писателя и читателя, необходимо существование посредника, который следит за всеми темами и устанавливает все двухточечные перемещения данных, необходимые для перераспределения данных. После подписки на канал, задача, которая записывает в канал, «заимствует» следующий доступный буфер записи из этого канала. Она может заполнить этот буфер данными и отправить их считывающей задаче. Аналогично задача, которая считывает из канала, первоначально получит «неправильный» буфер. Затем она ожидает, пока буфер наполнится и станет «правильным».

Алгоритм А.7. Параллельный по задачам алгоритм 2D-фильтра

Верхние индексы, определяющие множества процессоров, означают, что функция должна выполняться только на этом множестве процессоров.

```
1  FilterStage1^{S1}() ▷ Stage 1 on processor set 1.
2  FilterStage2^{S2}() ▷ Stage 2 on processor set 2.

FilterStage1()
1  X : C^{P(N)×M}          c : C^M
2  X̃ = (X ⟹_n^{Topic12} ) ▷ Init conduit.
3  for i = 1 : size(X.loc, 1)
4        do                ▷ Stage 1 on processor set 1.
5           X̃.loc(i, :) = IFFT(FFT(X.loc(i, :)) .* c)
6  X̃ ⇒                     ▷ Write to conduit.

FilterStage2()
1  Y : C^{N×P(M)}
2  Ỹ = ( ⟹_m^{Topic12} Y) ▷ Init conduit.
3  ⇒ Ỹ                     ▷ Read from conduit.
4  for j = 1 : size(Y.loc, 2)
5        do                ▷ Stage 2 on processor set 2.
6           Y.loc(:, j) = FFT(Ỹ.loc(j, :))
```

Перевод
1 Этап 1 на множестве процессоров 1;
2 Этап 2 на множестве процессоров 2.

2 Иниц. канал;
4 Этап 1 на множестве процессоров 1;
6 Записать в канал.

2 Иниц. канал;
3 Считать с канала;
5 Этап 2 на множестве процессоров 2.

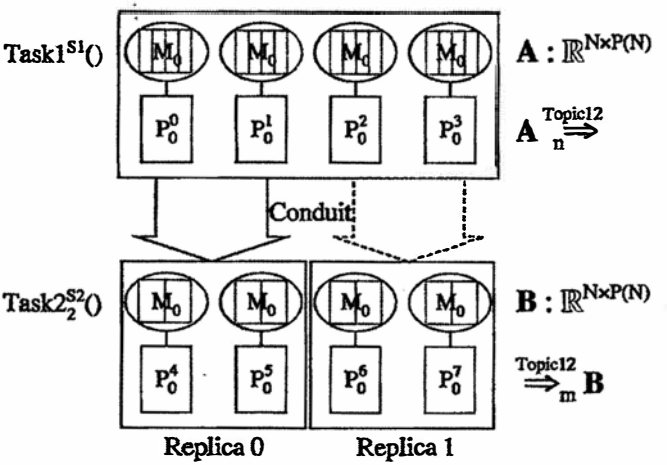
Сторона записи канала инициализируется и используется следующим образом. В алгоритме А.7 строка 2 функции FILTERSTAGE1 инициализирует сторону записи канала. Тема канала – «Topic12». Тип, размер и распределение данных, подлежащих записи в канал, задается распределенным массивом **X**. Количество буферов записи задается нижним индексом N перед $\Longrightarrow$. Первый доступный буфер возвращается в виде распределенного массива $\tilde{\mathbf{X}}$. Этот буфер наполняется данными в строке 5. Строка 6 записывает этот буфер в канал и затем заимствует следующий доступный буфер записи.

Сторона считывания канала инициализируется и используется следующим образом. В Алгоритме А.7 строка 2 функции FILTERSTAGE1 инициализирует сторону счи-

тывания канала. Тема канала – «Topic12». Тип, размер и распределение данных, подлежащих считыванию из канала, задается распределенным массивом Y. Количество буферов считывания задается нижним индексом **m** после $\Longrightarrow$. «Неправильный» буфер возвращается в виде распределенного массива $\tilde{\mathbf{Y}}$. Этот буфер заполняется данными в строке 3. Строка 6 использует этот буфер для выполнения вычислений.

А.3.3. Мелкозернистый конвейерный параллелизм по задачам

Задачи и каналы обычно используются для организации крупномодульного параллелизма, но их также можно использовать при организации параллелизма мелкозернистого. Например, предположим, в двумерном фильтрующем алгоритме первая ступень обрабатывает каждую строку данных по мере считывания и затем незамедлительно отправляет результаты на следующую ступень. Эта функциональность может быть реализована с задачами и каналами, но при этом необходимо добавить концепцию дублированной задачи (см. рисунок А.6). Дублированная задача, обозначенная $\text{TASK}^{S1}()$, создает R точных копий задач, при этом каждая копия содержит N_{S_1}/R процессоров. Канал, приходящий от дублированной задачи, методом карусельной диспетчеризации направляет выходные данные канала последующим задачам.



Перевод
Task – задача;
Conduit – канал;
Replica – копия.

Рисунок А.6. Дублированная задача, связанная с недублированной задачей
Задача 1 распределяется на множество процессоров $S1 = \{0, 1, 2, 3\}$ и содержит две копии. Копия 0 находится на множестве процессоров $\{0, 1\}$, а копия 1 находится на множестве процессоров $\{2, 3\}$. Каждая содержит распределенных массив **A**.

Задача 2 распределяется на множество процессоров $S2 = \{4, 5, 6, 7\}$ и содержит распределенный массив B. Данные в массиве A передаются в массив B через канал, обозначенный «Topic12». Задача2 по очереди получает данные от копий 0 и 1

Алгоритм А.8 представляет собой двумерный фильтрующий алгоритм, переписанный с использованием мелкозернистых задач и каналов. Функции преобразуются в задачи посредством их выполнения на определенном множестве процессоров через обозначения FILTERSTAGE2^{S1}() и FILTERSTAGE2^{S2}(). Первая ступень выполняется с коэффициентом повторения N_{s1}, так что каждая копия задачи выполняется только на одном процессоре. Внутри FILTERSTAGE1 все операции совершаются над вектором X. Вторая ступень собирает выходные векторы всех результатов в матрицу Y и затем выполняет вторую ступень.

Алгоритм А.8. Мелкозернистый параллельный по задачам алгоритм 2D-фильтра

Задачи и каналы используются для перемещения строк матрицы

```
FILTERFINE1^{S1}_{N_{S1}}() ▷ Replicated Stage 1 on processor set 1.
FILTERFINE2^{S2}() ▷ Stage 2 on processor set 2.
```

```
FILTERFINE1()
x : C^M)          c : C^M
x̃_0 = ( ==Topic01=>_m x) ▷ Init input conduit.
x̃_1 = (x ==Topic12=>_n ) ▷ Init output conduit.
⇒ x̃_0                ▷ Read from input conduit.
while x̃_0
      do             ▷ Stage 1 on processor set 1.
         x̃_1 = IFFT(FFT(x̃_0 .* c)
         x̃_1 ⇒                ▷ Write to output conduit.
```

```
FILTERFINE2()
Y : C^{N×P^{S2}(M)},        y : C^{P(M)}
ỹ = ( ==Topic12=>_m y) ▷ Init conduit.
for i = 1 : N
      do             ▷ Stage 2 on processor set 2.
         ⇒ ỹ                ▷ Read from conduit.
         Y.loc(i,:) = ỹ.loc
for j = 1 : SIZE(Y.loc, 2)
      do             ▷ Stage 2 on processor set 2.
         Y.loc(:,j) = FFT(Ỹ.loc(j,:))
```

Перевод

1 Дублированный этап 1 на множестве процессоров 1;

2 Этап 2 на множестве процессоров 2;

2 Иниц. входной канал;
3 Иниц. выходной канал;
4 Считать из входного канала;
6 Этап 1 на множестве процессоров 1;
8 Записать в выходной канал;
2 Иниц. канал;
4 Этап 2 на наборе процессоров 2;
5 Этап из канала;
8 Этап 2 на множестве процессоров 2.

А.3.4. Обобщенный иерархический блочный параллельный алгоритм

Процесс определения точной параллельной операции для каждой функции может быть слишком утомительным, особенно если определенные шаблоны перемещения данных являются общими для всего множества функций. Например, общей операцией является выполнение некоторой функции-ядра (kernel) над иерархическом распределенным массивом. Типичный шаблон выполнения будет заключаться в том, чтобы каждый $\overline{P}_{ID}$ выполнил функцию–ядро над данными, распределенными на нем. Неявное выполнение было представлено в Алгоритме А.5. Однако зачастую каждый $\overline{P}_{ID}$ может обрабатывать один блок своих данных в один момент времени. В этой ситуации желательно установить каналы, по которым данные направятся в каждый $\overline{P}_{ID}$, произойдет обработка данных, и они направятся наружу. Алгоритм А.9 иллюстрирует данный метод с использованием обобщенных вспомогательных функций, представленных в Алгоритме А.10. Для заданной процессорной архитектуры оптимальный способ применения этой функции-ядра может быть общим для многих алгоритмов.

Алгоритм А.9. Обобщенный иерархический блочный параллельный алгоритм 2D-фильтра

Вспомогательные функции используются для выполнения специальных функций-ядер на иерархических распределенных массивах.

```
Y : C^{N×P(M)} = FilterExpHierBlockParallel(X : C^{P(P̄(N))×M})
1  HierRun(Kernel1, X)
2  Y = X              ▷ Cornerturn data.
3  HierRun(Kernel2, Y)

X1 : C^{n×N)} = Kernel1(X0 : C^{n×M})
1  c : C^M
2  for i = 1 : size(X0, 1)
3        do
4           X1(i,:) = IFFT(FFT(X0(i,:)) .* c)

Y1 : C^{N×n)} = Kernel2(Y0 : C^{N×n})
1  for j = 1 : size(Y0, 2)
2        do
3           Y1(:,j) = FFT(Y0(:,j))
```

Перевод

2 Переразмещение данных с поворотом угла.

Функция HierRun использует функцию KernelTask для запуска ядра в качестве набора независимых задач на $\bar{P}$. Кроме того, обе функции HierRun и KernelTask устанавливают необходимые каналы к ядру и от ядра, чтобы организовать поток данных внутрь и наружу. В результате пользователю необходимо только написать высокоуровневую программу и низкоуровневые функции-ядра. Поток данных определяется картой иерархического распределенного массива и автоматически выполняется посредством вспомогательных функций.

Алгоритм А.10. Иерархические вспомогательные функции

Функции выполнения и задачи для составления иерархических программ.

```
HierRun(Kernel, X)
 1  X̃0 = (X.loc.loc.blk ⟹(Topic01) ) ▷ Init input conduit.
 2  X̃1 = ( ⟹(Topic12) X.loc.loc.blk) ▷ Init output conduit.
 3  KernelTask^p̄_{N_p̄}(Kernel, X)
 4  ⇒ X̃0                ▷ Read from input conduit.
 5  for k = 1 : X.loc.N_blk
 6        do
 7           for p̄ = 1 : N_p̄
 8                 do            ▷ Stage 1.
 9                    X̃0 = X.loc.loc_p̄.blk_k
10                    X̃0 ⇒
11           for p̄ = 1 : N_p̄
12                 do            ▷ Stage 1.
13                    ⇒ X̃1
14                    X.loc.loc_p̄.blk_k = X̃1
```

```
KernelTask^p̄_{N_p̄}(Kernel), X)
1  X̃0 = ( ⟹(Topic01) X.loc.loc.blk) ▷ Init input conduit.
2  X̃1 = (X.loc.loc.blk ⟹(Topic12) ) ▷ Init output conduit.
3  while X̃0
4        do
5           ⇒ X̃0
6           X̃1 = Kernel(X̃0)
```

Перевод

1 Иниц. входной канал;
2 Иниц. выходной канал;
4 Считать из входного канала;
8 Этап 1;
12 Этап 1.

1 Иниц. входной канал;
2 Иниц. выходной канал.

Список литературы

1. *[Choy & Edelman 1999] Ron Choy and Alan Edelman, Parallel MATLAB: Doing it right, Proceedings of the IEEE (Feb. 2005) Vol. 93, No. 2, pp. 331–341.*

2. *[Flynn 1972] Michael Flynn, Some computer organizations and their effectiveness, IEEE Trans. Comput., Vol. C-21, p. 948, 1972.*

Указатель

Np.....24, 271
*Np**1.....25
P_{ID}.....23, 271
`Np`.....26
`Pid;`.....26
`RecvMsg`.....27
`SendMsg`.....27
`agg`.....27
`global_block_range`.....224
`global_block_ranges`.....224
`global_ind`.....27
`local`.....27
`map`.....27, 111
`pRUN`.....29
`put_local`.....27
`synch`.....95
`transpose_grid`.....242
1**Np*.....25

LU-разложение – LV
Алгоритм – Algorithm.....256
Параллельное – Parallel.....255
Производительность – Performance... 266, 267, 268

MATLAB Использование – MATLAB Usage.....27
MatlabMPI – MatlabMPI.....124
Разработка языка параллельного программирования
Matlab – pMatlab Development.....21
StarP – StarP.....21
Барьерная синхронизация –
Barrier synchronization.....207

Быстрое преобразование Фурье (БПФ) –
Fast Fourier transform (FFT)
1D (одномерное) – 1D.....238
2D (двумерное) – 2D.....265, 271
Алгоритм – Algorithm.....181
Гигафлопс (производительность) – Gigaflops..237
Пропускная способность – Bandwidth.....237

Векторизировать – Vectorize.....193
Вертикальное масштабирование – Scaling up.....210
Временная локальность – Temporal locality.....153
Временный показатель – Temporal score.....153
Время – Time.....79
Время вычисления – Compute time.....146

Высокопроизводительная Linpack –
High Performance Linpack (HPL)
Алгоритм – Algorithm.....183, 253
Гигафлопс (производительность) – Gigaflops...237

Задержка – Latency.....127
Задержка в зависимости от размера сообщения –
Latency versus message size.....127
Локальность – Locality.....114, 272
Неравномерность нагрузки –
Load imbalance.....110
Обратная пропускная способность –
Inverse bandwidth.....158
Распределение нагрузки – Load balancing....195
Схема Кека – Kuck diagram....159, 161, 272, 275

Высокопроизводительная вычислительная обработка – High Performance Computing (HPC) Challenge
Высокопроизводительная Встроенная
Вычислительная Обработка – High Performance
Embedded Computing (HPEC) Challenge.....271
Иерархия памяти – Memory hierarchy.....154
Пространственная/временная –
Spatial/temporal.....178
Схема Кека – Kuck diagram.....159
Характеристика – Characteristics.....176

Вычислительная наука – Computational science..176
Вычисляет владелец – Owner computes.86, 162, 270
Горизонтальное масштабирование –
Scaling out.....211
Дискретное преобразование Фурье (ДПФ) –
Discrete Fourier transform.....236
Доля Амдала – Amdahl fraction.....149
Зависимость пропускной способности от размера сообщений –
Bandwidth versus message size.....125
Закон Амдала – Amdahl's law.....147
Иерархические – Hierarchical.....269
Иерархия памяти –
Memory hierarchy.....154, 155, 159, 174, 178
Изменение количества процессоров –
Processor sizing.....142
Индекс – Index.....224
Индексирование– Indexing.....90

Интерфейс передачи сообщений (MPI)
Многоузловой – Multinode.....98
Многоядерный – Multicore.....161
Размер – Message Passing Interface (MPI) size..... 86
Ранг – Message Passing Interface (MPI) rank..... 86

Карта размещения – Map. 28, 48, 55, 57, 68, 86, 129, 266
Клиент/сервер – Client/server.....133
Количество строк кода
Software Lines of Code SLOC.....130, 167

Комбинированные – Hybrid
.....172, 213, 217, 229, 254, 263
Коммуникация «каждый к каждому» –
All-to all communication 126, 128, 197

Коммуникация – Communication
Ближайшая соседняя точка – Nearest–neighbor .. 84
Время – Time..... 83, 154
Задержка – Latency 154
Издержки – Overhead 131
Конвейер – Pipelinec..... 84
Пропускная способность – Bandwidth..... 154
Распределение/сбор – Scatter/gather..... 84
Сеть – Network..... 154
Сложность – Complexity 137
Шаблон – Pattern..... 83

Конвейерный – Pipeline 75, 109, 142, 279
Коэффициент ускорения, ускорение –
Speedup.....41, 149
Критический путь – Critical path 257
Крупнозернистый (карусельная диспетчеризация) –
Coarse grained block.....142
Линейное – Linear 130, 149
Локальная часть – Local part 26, 65, 99, 275
Масштабирование – Scaling.....132
Масштабированный размер задачи –
Scaled problem size 151
Менеджер/исполнитель – Manager/worker
.....78, 86, 87, 92, 161
Метрики – Metrics..... 136
Многоэтапный параллелизм – Multistage 60
Многоузловой – Multinode.....233, 247, 267
Многоядерный – Multicore.....231, 246, 265
Множество Мандельброта – Mandelbrot set 33
«Наука моделирования» – Simulation science 176
Модель – Model 84, 86, 159
Модель аппаратного обеспечения –
Hardware model.....176
Модель стоимости и кодирования (CoCoMo) –
Code and Cost Modeling (CoCoMo) 167
Модель фон Неймана – Von Neumann model 151
Насыщение – Saturation 149
Обмен сообщениями –
Message passing.....18, 78, 85, 93, 108, 166
Одна инструкция, множество потоков данных –
Single Instruction – Multipla Data flows SIMD 175
Одна программа, несколько потоков данных –
Single Program – Multiople Data flows SPMD.. 24, 271
Отношение «вычисление-коммуникация» –
Computation-to-communication 89, 115, 137, 184
Оценка Estimating..... 188
Памяти Memory 79

Параллелизм – Concurrency 114, 243
Параллелизм – Parallel
Выполнение – Execution.....29, 120
Канал – Conduit 282
Крупнозернистый тривиальный массовый
параллелизм – Coarse-grained 42
Массовый – Embarrassingly.....31, 42
Мелкозернистый – Fine–grained 31
По данным – Data.....142
Простые приложения – Easy applications.....177
Сложные приложения – Hard applications 177

Перегрузка функции – Function overloading 117
Перекрытие – Overlap..... 90, 107, 111
Переразмещение с поворотом угла –
Corner turn.....159
Перераспределение – Redistribution.....25, 102, 121
По задачам – Task.....75, 142, 281
Зависимость производительности от трудозатрат –
Versus effort 18, 170, 195
Полиморфизм – Polymorphism.....118
Последовательная – Serial.....81

Поток – Stream
Алгоритм – Algorithm 181, 203
«Каждый к каждому» – All-to-all.....180
Массово-параллельные приложения –
Embarrassingly parallel 181
Пропускная способность – Bandwidth 178
Сильное масштабирвоание – Strong scaling .151
Супер-ЭВМ – Supercomputer 175

Программирование на основе карты размещения –
Map-based programming.....28
Продуктивность – Productivity.....169, 199
Производительность – Performance.....128

Произвольный доступ – RandomAccess
Алгоритм – Algorithm 182, 216
Миллиардах обновлений в секунду (GUPS) –
Giga Updates Per Second (CUPS) 214

Пропускная способность – Bandwidth 141, 157
Относительная пропускная способность передачи с
переупорядочиванием данных – Throughput 163
Пропускная способность локальной памяти –
Memory bandwidth.....236
Пространственная локальность –
Spatial locality 152

Пространственная фильтрация – Beamforming 63
Пространственный показатель – Spatial score.....152
Профилирование – Profiling 79
Процессорное теговое индексированное семейство линейных сегментов – Processor Index tagged Family of Line Segments PITFALLS....121
Размер сообщений – Message size 127, 157, 248
Размер, вес и мощность – Size, weight and power....163

Распределенные массивы – Distributed arrays
Блочно-циклические – Block-cyclic 108, 265
Блочные – Block.... 102, 109, 251
Конструктор – Constructor 28
«Плоского» распределенного массива – Flat 276
Цели реализации – Design goals.... 106
Циклические – Cyclic.... 102, 107, 275

Результат – Output.... 119
P-запись – P-notation 25, 272
Свертка – Convolution....44, 80
Скорость кодирования – Coding rate 168
Слабое масштабирование – Weak scaling 121
Сложность вычислений – Computational complexity....137,184

Степени параллелизма – Degrees of parallelism 137, 140, 185
Затраты на программное обеспечение – Software cost....167
Структуры данных – Data structures 137
Сублинейное – Sublinear....149
Суперлинейное – Superlinear....149
Топология сети – Network topology....156
Уровень реализации – Implementation level.... 116
Файловый ввод/вывод – File IO 51
Фиксированный размер задачи – Fixed problem size....247
Фильтрация в двух измерениях – Filtering in two dimensions 123
Фрагментированные – Fragmented 104
Характеристики приложений – Application characteristics....139
Чистые – Pure 103
Шаблоны программирования – Programming patterns....83
Шаги отладки – Debugging steps....39, 100, 130

Эффективность – Efficiency
Вычисления – Compute 176, 185
Коммуникации – Communication 186
Программного обеспечения – Software 161

Jeremy Kepner

Parallel MATLAB for Multicore and Multinode Computers

MATLAB is currently the dominant language of technical computing with approximately one million users worldwide, many of whom can benefit from the increased power offered by widely available multicore processors and multimode computing clusters. MATLAB is also an ideal environment for learning about parallel computing, allowing the user to focus on parallel algorithms instead of the details of the implementation. The succinctness of MATLAB allows many specific examples to be presented to illustrate parallel programming concepts.

The book is organized around two central concepts: the core programming process (i.e., design, code, debug, and test) and the core parallel programming models (i.e., distributed arrays, manager/worker, and message passing). Throughout the book the approach is to first present concrete examples and then discuss in detail the more general parallel programming concepts these examples illustrate. The book aims to bring these ideas to bear on the specific challenge of writing parallel programs in MATLAB.

The target audience of the book is anyone who needs to adapt their serial MATLAB program to a parallel environment. In addition, this book is suitable as either the primary book in a parallel computing class or as a supplementary text in a numerical computing class or a computer science algorithms class.

Contents

I. Fundamentals

1. Primer: Notation and Interfaces 22
2. Introduction to pMatLab 30
3. Interacting with Distributed Arrays 59

II. Advanced Techniques

4. Parallel Programming Models 77
5. Advanced Distributed Array Programming 101
6. Performance Metrics and Software Architecture 135

III. Case Studies

7. Parallel Application Analysis 173
8. Stream 201
9. Random Access 213
10. Fast Fourier Transform 236
11. High Performance Linpack 250

Appendix. Notation for Hierarchical Parallel Multicore Algorithms

A.1. Introduction 269
A.2. Data Parallelism 270
A.3. Pipeline Parallelism 279

Index 290

Учебное издание

Джереми Кепнер

Параллельное программирование в среде MATLAB для многоядерных и многоузловых вычислительных машин

Научный редактор *Д. В. Дубров*

Редактор *М. С. Сафарова*
Оформление переплета *П. А. Брызгалов, Ю. Н. Симоненко*
Перевод и верстка выполнены
ООО «Инновационные Лингвистические Технологии – РЭС»
www.ilt-res.com
Корректор *М. С. Сафарова*

Подписано в печать 30.11.2012.
Формат 70×100/16. Гарнитура Minion Pro.
Офсетная печать. Бумага офсетная № 1.
Усл. печ. л. 23,87. Тираж 1800 экз.
Изд. № 9726. Заказ № 2208

Ордена «Знак Почета» Издательство Московского университета
125009, Москва, ул. Б. Никитская 5/7.

Тел.: (495) 629-50-91. *Факс:* (495) 697-66-71
(495) 939-34-93 *(отдел реализации)*
E-mail: secretary-msu-press@yandex.ru

Отпечатано в ППП «Типография „Наука“»
121099, Москва, Шубинский пер. 6