

УДК 004.4  
ББК 32.973.202  
В76

**Восс М., Асенхо Р., Рейндерс Дж.**

В76 Параллельное программирование на С++ с помощью библиотеки ТВВ / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2020. – 674 с.: ил.

**ISBN 978-5-97060-864-7**

Эта книга представляет собой современное руководство для всех пишущих на С++ программистов, которые хотят научиться работать с библиотекой Threading Building Blocks (ТВВ). Написанная экспертами по ТВВ и параллельному программированию, она вобрала в себя их многолетний коллективный опыт разработки и преподавания параллельного программирования с помощью ТВВ. Излагаемый материал представлен в доступной форме. В книге имеются многочисленные примеры и рекомендации, которые помогут вам в полной мере овладеть ТВВ и задействовать всю мощь параллельных систем.

Книга начинается с описания базовых параллельных алгоритмов и средств распараллеливания, имеющих в стандартной библиотеке шаблонов С++. Вы узнаете об основах управления памятью, работе со структурами данных и решении типичных проблем синхронизации. Затем эти идеи применяются к более сложным системам, на примере которых объясняются компромиссы во имя производительности, общеупотребительные паттерны параллельного программирования, управление потоками и накладные расходы, а также применение ТВВ к программированию гетерогенных систем и систем на кристалле.

УДК 004.4  
ББК 32.973.202

Original English language edition printed on acid-free paper. Copyright © 2019 by Intel Corporation. Russian language edition copyright © 2020 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-4842-4397-8 (англ.)  
ISBN 978-5-97060-864-7 (рус.)

Copyright © 2019 by Intel Corporation  
© Оформление, издание, перевод,  
ДМК Пресс, 2020

# Содержание

<b>От издательства</b> .....	14
<b>Об авторах</b> .....	15
<b>Благодарности</b> .....	16
<b>Предисловие</b> .....	18
Мыслите параллельно .....	18
Что такое TBB.....	18
Структура книги и предисловия.....	18
Мыслите параллельно.....	19
Мотивы, стоящие за библиотекой TBB.....	19
Программирование с применением задач, а не потоков .....	20
Компонуемость: параллельное программирование необязательно должно быть запутанным.....	21
Масштабируемость, производительность и погоня за переносимой производительностью .....	22
Введение в параллельное программирование .....	23
Параллелизм вокруг нас .....	24
Конкурентность и параллелизм.....	24
Враги параллелизма.....	25
Терминология параллелизма .....	26
Сколько параллелизма в приложении? .....	33
Что такое потоки? .....	38
Что такое SIMD?.....	40
Безопасность в условиях конкурентности .....	41
Взаимное исключение и блокировки .....	41
Корректность .....	43
Взаимоблокировка .....	44
Состояния гонки.....	45
Нестабильность (недетерминированность) результатов .....	45
Уровень абстракции.....	46
Паттерны .....	46
Локальность и месть кешей .....	46
Аппаратное обоснование .....	47
Локальность ссылок .....	48
Строки кеша, выравнивание, разделение, взаимное исключение и ложное разделение .....	49
TBB помнит о кешах.....	53
Введение в векторизацию (SIMD).....	53
Введение в средства C++ (в объеме, необходимом для работы с TBB).....	55
Лямбда-функции.....	55
Обобщенное программирование.....	55
Контейнеры .....	56

Шаблоны .....	56
STL.....	56
Перегрузка .....	57
Диапазоны и итераторы .....	57
Резюме.....	58
Дополнительная информация .....	58

## ЧАСТЬ I

<b>Глава 1. Приступаем: «Hello, TBB!» .....</b>	<b>60</b>
Почему именно Threading Building Blocks? .....	60
Производительность: низкие накладные расходы, большое преимущество у C++ ..	61
Эволюция поддержки параллелизма в TBB и C++ .....	62
Недавние добавления в C++, относящиеся к параллелизму .....	63
Библиотека Threading Building Blocks (TBB) .....	63
Интерфейсы параллельного выполнения .....	64
Интерфейсы, не зависящие от модели выполнения .....	66
Использование строительных блоков в TBB .....	66
Да начнем же уже! .....	66
Получение библиотеки TBB.....	66
Получение кода примеров.....	67
Написание первого примера «Hello, TBB!» .....	67
Сборка простых примеров .....	70
Сборка в Windows в Microsoft Visual Studio .....	70
Сборка на платформе Linux из терминала .....	72
Более полный пример .....	74
Начинаем с последовательной реализации .....	75
Добавление уровня обмена сообщениями с помощью потокового графа .....	78
Добавление уровня разветвления–соединения с помощью <code>parallel_for</code> .....	80
Добавление уровня SIMD с помощью функции <code>transform</code> из Parallel STL .....	81
Резюме.....	84
<b>Глава 2. Обобщенные параллельные алгоритмы .....</b>	<b>85</b>
Функциональный параллелизм на уровне задач .....	88
Чуть более сложный пример: параллельная реализация быстрой сортировки .....	90
Циклы: <code>parallel_for</code> , <code>parallel_reduce</code> и <code>parallel_scan</code> .....	92
<code>parallel_for</code> : применение тела к каждому элементу диапазона .....	92
<code>parallel_reduce</code> : вычисление одного результата для всего диапазона .....	95
<code>parallel_scan</code> : редукция с промежуточными значениями .....	100
Как это работает? .....	102
Более сложный пример: линия прямой видимости .....	103
Варить до готовности: <code>parallel_do</code> и <code>parallel_pipeline</code> .....	105
<code>parallel_do</code> : применять тело, пока имеются элементы .....	106
<code>parallel_pipeline</code> : обработка несколькими фильтрами .....	113
Резюме.....	120
Дополнительная информация .....	120
<b>Глава 3. Потоковые графы .....</b>	<b>122</b>
Зачем использовать графы для выражения параллелизма? .....	123
Основы интерфейса потоковых графов в TBB.....	124
Шаг 1: создать объект графа .....	125

Шаг 2: создать узлы .....	126
Шаг 3: добавить ребра .....	128
Шаг 4: запустить граф .....	128
Шаг 5: ждать завершения выполнения графа .....	131
Более сложный пример потокового графа данных .....	131
Реализация примера в виде потокового графа TBB .....	133
Производительность потокового графа данных .....	134
Частный случай – графы зависимостей .....	136
Реализация графа зависимостей .....	138
Оценка масштабируемости графа зависимостей .....	143
Дополнительные сведения о потоковых графах в TBB .....	143
Резюме .....	144

## **Глава 4. TBB и параллельные алгоритмы стандартной библиотеки шаблонов C++ .....**

Какое отношение библиотека STL имеет к этой книге? .....	145
Аналогия для осмысления политик выполнения в Parallel STL .....	147
Простой пример – алгоритм <code>std::for_each</code> .....	148
Какие алгоритмы предоставляет реализация Parallel STL? .....	151
Как получить и использовать копию библиотеки STL, в которой применяется TBB .....	151
Алгоритмы в библиотеке Intel Parallel STL .....	152
Нестандартные итераторы открывают дополнительные способы использования .....	153
Некоторые наиболее полезные алгоритмы .....	156
<code>std::for_each</code> , <code>std::for_each_n</code> .....	156
<code>std::transform</code> .....	158
<code>std::reduce</code> .....	159
<code>std::transform_reduce</code> .....	160
Политики выполнения в деталях .....	162
<code>sequenced_policy</code> .....	162
<code>parallel_policy</code> .....	163
<code>unsequenced_policy</code> .....	163
<code>parallel_unsequenced_policy</code> .....	164
Какую политику выполнения использовать? .....	164
Другие способы ввести SIMD-параллелизм .....	165
Резюме .....	166
Дополнительная информация .....	166

## **Глава 5. Синхронизация – почему ее нужно избегать и как это сделать .....**

Сквозной пример: гистограмма изображения .....	167
Небезопасная параллельная реализация .....	170
Первая безопасная параллельная реализация: крупнозернистая блокировка .....	173
Варианты мьютексов .....	178
Вторая безопасная параллельная реализация: мелкозернистая блокировка .....	180
Третья потокобезопасная параллельная реализация: атомарные переменные .....	184
Улучшенная параллельная реализация: приватизация и редукция .....	188
Поточно-локальная память .....	189
Класс <code>enumerable_thread_specific</code> .....	190
Тип <code>combinable</code> .....	192

Самая простая параллельная реализация: шаблон редукции.....	194
Подведем итоги .....	196
Резюме.....	200
Дополнительная информация .....	200

## Глава 6. Структуры данных для конкурентного

<b>программирования</b> .....	201
Основы важнейших структур данных .....	202
Неупорядоченные ассоциативные контейнеры .....	202
Отображение или множество .....	203
Несколько значений.....	203
Хеширование .....	203
Неупорядоченность.....	204
Конкурентные контейнеры.....	204
Конкурентные неупорядоченные ассоциативные контейнеры .....	206
Конкурентные очереди: обычные, ограниченные и с приоритетами .....	212
Конкурентный вектор.....	220
Резюме.....	223

## Глава 7. Масштабируемое выделение памяти .....

Выделение памяти в современном C++ .....	224
Масштабируемое выделение памяти: что .....	225
Масштабируемое выделение памяти: почему .....	226
Избежание ложного разделения с помощью дополнения .....	227
Альтернативы масштабируемому выделению памяти: какие.....	229
К вопросу о компиляции.....	230
Самый популярный способ использования (библиотека прокси для C/C++): как .....	230
Linux: использование библиотеки прокси .....	231
macOS: использование библиотеки прокси .....	232
Windows: использование библиотеки прокси.....	232
Тестирование библиотеки прокси .....	233
Функции C: масштабируемые распределители памяти для C .....	234
Классы C++: масштабируемые распределители памяти для C++ .....	235
Распределители с сигнатурой <code>std::allocator&lt;T&gt;</code> .....	236
<code>scalable_allocator</code> .....	236
<code>tbb_allocator</code> .....	237
<code>zero_allocator</code> .....	237
<code>cached_aligned_allocator</code> .....	237
Поддержка пула памяти: <code>memory_pool_allocator</code> .....	238
Поддержка выделения памяти для массивов: <code>aligned_space</code> .....	238
Избирательная подмена <code>new</code> и <code>delete</code> .....	239
Настройка производительности: некоторые рычаги управления .....	242
Что такое большие страницы? .....	242
Поддержка больших страниц в TBB .....	242
<code>scalable_allocation_mode(int mode, intptr_t value)</code> .....	243
<code>TBBMALLOC_USE_HUGE_PAGES</code> .....	243
<code>TBBMALLOC_SET_SOFT_HEAP_LIMIT</code> .....	243
<code>int scalable_allocation_command(int cmd, void *param)</code> .....	244
<code>TBBMALLOC_CLEAN_ALL_BUFFERS</code> .....	244
<code>TBBMALLOC_CLEAN_THREAD_BUFFERS</code> .....	244
Резюме.....	244

<b>Глава 8. ТВВ и параллельные паттерны</b> .....	245
Параллельные паттерны и параллельные алгоритмы.....	245
Паттерны определяют классификацию алгоритмов, проектных решений и т. д. ....	247
Паттерны, которые работают .....	248
Параллелизм данных одерживает победу .....	249
Паттерн Вложенность.....	249
Паттерн Отображение .....	251
Паттерн Куча работ.....	252
Паттерны редукции (Редукция и Сканирование) .....	252
Паттерн Разветвление–соединение.....	253
Паттерн Разделяй и властвуй .....	256
Паттерн Ветви и границы .....	256
Паттерн Конвейер.....	257
Паттерн Событийно-управляемая координация (реактивные потоки).....	258
Резюме .....	259
Дополнительная информация .....	259

## ЧАСТЬ II

<b>Глава 9. Столпы компонуемости</b> .....	261
Что такое компонуемость?.....	262
Вложенная композиция .....	263
Конкурентная композиция .....	265
Последовательная композиция.....	266
Благодаря каким особенностям библиотека ТВВ является компонуемой .....	268
Пул потоков ТВВ (рынок) и аренды задач .....	268
Диспетчер задач в ТВВ: заимствование работ, и не только .....	271
Соберем все вместе.....	277
Забегая вперед .....	281
Управление количеством потоков .....	281
Изоляция работ .....	281
Привязка задачи к потоку и потока к ядру .....	281
Приоритеты задач.....	281
Резюме .....	282
Дополнительная информация .....	282

<b>Глава 10. Использование задач для создания собственных алгоритмов</b> .....	283
Сквозной пример: вычисление последовательности .....	283
Высокоуровневый подход: <code>parallel_invoke</code> .....	285
Высший среди низших: <code>task_group</code> .....	287
Низкоуровневый интерфейс: часть первая – блокировка задач.....	289
Низкоуровневый интерфейс задач: часть вторая – продолжение задачи .....	293
Обход планировщика.....	299
Низкоуровневый интерфейс задач: часть третья – рециклинг задач.....	300
Контрольный список для интерфейса задач .....	302
И еще одно: FIFO-задачи (типа запустил и забыл) .....	303
Применение низкоуровневых средств на практике .....	304
Резюме .....	310
Дополнительная информация .....	311

<b>Глава 11. Управление количеством потоков</b> .....	312
Краткий обзор архитектуры планировщика TBB .....	313
Интерфейсы для управления количеством задач .....	314
Управление количеством потоков с помощью <code>task_scheduler_init</code> .....	314
Управление количеством потоков с помощью <code>task_arena</code> .....	315
Управление количеством потоков с помощью <code>global_control</code> .....	316
Сводка концепций и классов .....	316
Рекомендации по заданию количества потоков .....	317
Использование одного объекта <code>task_scheduler_init</code> в простом приложении .....	318
Использование нескольких объектов <code>task_scheduler_init</code> в простом приложении .....	320
Использование нескольких арен с разным числом слотов, чтобы подсказать TBB, куда направлять рабочие потоки .....	321
Использование <code>global_control</code> для управления количеством потоков, доступных для занятия слотов на аренах .....	324
Использование <code>global_control</code> с целью временно ограничить количество доступных потоков .....	326
Когда НЕ следует управлять количеством потоков .....	328
Что не так? .....	329
Резюме .....	330
<b>Глава 12. Применение изоляции работы для обеспечения корректности и повышения производительности</b> .....	331
Изоляция работ для обеспечения корректности .....	332
Создание изолированного региона с помощью <code>this_task_arena::isolate</code> .....	336
Использование арен задач для изоляции: обоюдоострый меч .....	341
Не поддавайтесь искушению использовать арены задач для изоляции ради корректности .....	344
Резюме .....	347
Дополнительная литература .....	347
<b>Глава 13. Привязка потока к ядру и задачи к потоку</b> .....	348
Создание привязки потока к ядру .....	349
Создание привязки задачи к потоку .....	351
Когда и как следует использовать средства привязки в TBB? .....	357
Резюме .....	358
Дополнительная информация .....	358
<b>Глава 14. Приоритеты задач</b> .....	359
Поддержка невытесняющих приоритетов в классе задач TBB .....	359
Задание статических и динамических приоритетов .....	361
Два простых примера .....	362
Реализация приоритетов без поддержки со стороны задач TBB .....	365
Резюме .....	367
Дополнительная информация .....	368
<b>Глава 15. Отмена и обработка исключений</b> .....	369
Как отменить коллективную работу .....	370
Отмена задач в деталях .....	371
Явное назначение TGS .....	373

Назначение TGC по умолчанию .....	375
Обработка исключений в TBB .....	379
Написание собственных классов исключений TBB .....	381
Соберем все вместе: компоуемость, отмена и обработка исключений .....	384
Резюме .....	386
Дополнительная информация .....	387

## **Глава 16. Настройка TBB-алгоритмов: зернистость, локальность, параллелизм и детерминированность**.....388

Зернистость задач: какой размер достаточен? .....	389
Выбор диапазонов и разбивателей для циклов .....	390
Обзор разбивателей .....	391
Выбирать ли степень детализации для управления зернистостью задач .....	392
Диапазоны, разбиватели и производительность кеша данных .....	395
Использование <code>static_partitioner</code> .....	402
Ограничение планировщика ради детерминированности .....	404
Настройка конвейеров в TBB: количество фильтров, режимы и маркеры .....	406
Сбалансированный конвейер .....	407
Несбалансированный конвейер .....	409
Конвейеры, локальность данных и привязка к потоку .....	410
В глубоких водах .....	411
Создание собственного типа диапазона .....	411
Класс <code>Pipeline</code> и фильтры, привязанные к потоку .....	414
Резюме .....	418
Дополнительная информация .....	418

## **Глава 17. Поточковые графы: дополнительные сведения**.....419

Оптимизация зернистости, локальности и степени параллелизма .....	419
Зернистость узла: какой будет достаточно? .....	420
Потребление памяти и локальность данных .....	428
Арены задач и потоковый граф .....	441
Рекомендации по работе с потоковыми графами: что полезно, а что вредно .....	444
Полезно: использовать вложенный параллелизм .....	444
Вредно: использовать многофункциональные узлы вместо вложенного параллелизма .....	444
Полезно: использовать узлы <code>join_node</code> , <code>sequencer_node</code> или <code>multifunction_node</code> для восстановления порядка в потоковом графе, когда это необходимо .....	445
Полезно: использовать функцию <code>isolate</code> для вложенного параллелизма .....	448
Полезно: использовать отмену и обработку исключений в потоковых графах .....	450
Полезно: задавать приоритеты для графа, в котором используется <code>task_group_context</code> .....	454
Вредно: создавать ребро между узлами разных графов .....	454
Полезно: использовать <code>try_put</code> для передачи информации между графами .....	456
Полезно: использовать <code>composite_node</code> для инкапсуляции группы узлов .....	458
Введение в Intel Advisor: Flow Graph Analyzer .....	462
Процесс проектирования в FGA .....	462
Процесс анализа в FGA .....	465
Диагностика проблем производительности с помощью FGA .....	467
Резюме .....	470
Дополнительная информация .....	470



<b>Глава 18. Дополнение потоковых графов асинхронными узлами</b> .....	471
Пример из асинхронного мира .....	472
Зачем и когда использовать <code>async_node</code> ? .....	476
Более реалистичный пример .....	478
Резюме .....	486
Дополнительная информация .....	487
<b>Глава 19. Накачаные потоковые графы: узлы OpenCL</b> .....	488
Пример «Hello OpenCL_Node» .....	489
Где исполняется наше ядро? .....	496
Возвращаясь к более реалистичному примеру из главы 18 .....	502
Дьявол кроется в деталях .....	509
Концепция NDRange .....	511
Поиграем со смещением .....	515
Задание ядра OpenCL .....	516
Еще о выборе устройства .....	517
Предупреждение по поводу порядка .....	520
Резюме .....	523
Дополнительная информация .....	524
<b>Глава 20. TBB в системах с архитектурой NUMA</b> .....	525
Определение топологии платформы .....	527
Каковы затраты на доступ к памяти .....	530
Базовый пример .....	531
Мастерство размещения данных и привязки к процессору .....	533
Привлекаем <code>hwloc</code> и TBB к совместной работе .....	538
Более сложные альтернативы .....	543
Резюме .....	544
Дополнительная информация .....	545
<b>Приложение А. История и предшественники</b> .....	546
Десятилетие «от птенца к орлу» .....	546
1. Революция TBB внутри Intel .....	546
2. Первая революция TBB в сфере параллелизма .....	547
3. Вторая революция TBB в сфере параллелизма .....	548
4. Птички TBB .....	549
Источники идей TBB .....	551
Модель ослабленного последовательного выполнения .....	552
Библиотеки, оказавшие влияние .....	552
Языки, оказавшие влияние .....	554
Прагмы, оказавшие влияние .....	554
Влияние обобщенного программирования .....	555
Учет кешей .....	555
Учет стоимости квантования времени .....	556
Литература для дополнительного чтения .....	557

<b>Приложение В. ТВВ в кратком изложении</b> .....	560
Отладка и условный код .....	560
Макросы ознакомительных средств .....	562
Диапазоны .....	562
Разбиватели .....	563
Алгоритмы .....	564
Алгоритм: parallel_do .....	564
Алгоритм: parallel_for .....	567
Алгоритм: parallel_for_each .....	569
Алгоритм: parallel_invoke .....	571
Алгоритм: parallel_pipeline .....	572
Алгоритм: parallel_reduce и parallel_deterministic_reduce .....	574
Алгоритм: parallel_scan .....	578
Алгоритм: parallel_sort .....	581
Алгоритм: pipeline .....	583
Потоковый граф .....	585
Потоковый граф: класс graph .....	586
Потоковый граф: порты и ребра .....	587
Потоковый граф: узлы .....	587
Выделение памяти .....	597
Контейнеры .....	602
Синхронизация .....	620
Поточно-локальная память (TLS) .....	626
Хронометраж .....	634
Группы задач: использование планировщика с заимствованием задач .....	635
Планировщик задач: точный контроль над планировщиком с заимствованием задач .....	636
Настройки плавающей точки .....	647
Исключения .....	649
Потоки .....	651
Parallel STL .....	652
<b>Глоссарий</b> .....	655
<b>Предметный указатель</b> .....	668

# От издательства

## Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте [www.dmkpress.com](http://www.dmkpress.com), зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com), при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу [http://dmkpress.com/authors/publish\\_book/](http://dmkpress.com/authors/publish_book/) или напишите в издательство по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

## Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте [www.dmkpress.com](http://www.dmkpress.com) на странице с описанием соответствующей книги.

## Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com), и мы исправим это в следующих тиражах.

## Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Apress очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com) со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

# Об авторах

**Майкл Восс** – главный инженер группы архитектуры, графики и программного обеспечения в компании Intel. Он входил в команду разработки ТВВ еще до выхода версии 1.0 в 2006 году и был первым архитектором API потокового графа ТВВ. Также является одним из ведущих разработчиков программы Flow Graph Analyzer – графического инструмента для анализа потоков данных на однородных и гетерогенных платформах. Автор и соавтор свыше 40 печатных работ по вопросам параллельного программирования, часто консультирует заказчиков по широкому кругу проблем и помогает им эффективно использовать потоковые библиотеки Intel. До поступления на работу в Intel в 2006 году был доцентом факультета электронной и вычислительной техники имени Эдварда С. Роджерса в Торонтском университете. Получил степень доктора философии в школе электронной и вычислительной техники при университете Пердью в 2001 году.

**Рафаэль Асенхо** – профессор компьютерной архитектуры в Малагском университете, Испания. Получил степень доктора философии по технике связи в 1997 году и работал доцентом на факультете компьютерной архитектуры с 2001 по 2017 год. В 1996 и 1997 годах был приглашенным преподавателем в Иллинойском университете в Урбана-Шампейне, а в 1998 году – приглашенным научным сотрудником в том же университете. Также работал приглашенным научным сотрудником в Исследовательском центре Томаса Уотсона компании IBM в 2008 году и в компании Cray Inc. в 2011 году. Использует ТВВ начиная с 2008 года, а в последние пять лет занимается промышленным применением гетерогенных кристаллов, в которых ТВВ служит в качестве координирующего каркаса. В 2013 и 2014 годах приезжал в Иллинойский университет в Урбана-Шампейне для работы над кристаллами, объединяющими CPU и GPU. В 2015 и 2016 годах начал исследовательскую работу по кристаллам, объединяющим CPU и ППВМ (программируемая пользователем вентиляционная матрица, англ. FPGA), во время работы в Бристольском университете. Исполнял обязанности председателя на конференции ACM PPoPP'16 (по принципам и практике параллельного программирования) и был членом оргкомитета, а также членом программного комитета на нескольких конференциях по высокопроизводительным вычислениям (PPoPP, SC, PACT, IPDPS, HPCA, EuroPar и SBAC-PAD). В сферу его профессиональных интересов входят модели и архитектуры гетерогенного программирования, распараллеливание нерегулярного кода и оптимизация энергопотребления.

**Джеймс Рейндерс** – консультант, имеющий за плечами более тридцати лет опыта в области параллельных вычислений. Автор, соавтор или редактор девяти технических книг по параллельному программированию. Принимал участие в разработке ключевых элементов двух самых быстрых в мире компьютеров (номер 1 в списке Top500), а также многих других суперкомпьютеров и средств разработки ПО. В середине 2016 года Джеймс отметил 10 001 день (свыше 27 лет) работы в Intel, но и теперь продолжает писать, преподавать, программировать и консультировать в различных областях, связанных с параллельными вычислениями (высокопроизводительные вычисления и искусственный интеллект).

# Благодарности

Два человека поддерживали этот проект с самого начала и до конца – Санджив Шах (Sanjiv Shah) и Херб Хинсторф (Herb Hinstorff). Мы благодарны им за поддержку, ободрение, а временами и ненавязчивое подталкивание.

По-настоящему героические усилия предприняли рецензенты, которые присылали нам содержательные и подробные отзывы на черновики отдельных глав. Благодаря высокому качеству их работы нам пришлось потратить больше времени на редактирование текста, чем первоначально планировалось. Но в результате книга стала лучше.

Круг рецензентов составляла элита пользователей и основных разработчиков ТВВ. Редко бывает, чтобы в доведении книги до ума принимало участие такое энергичное и благосклонно настроенное сообщество. Читатели книги должны знать этих людей поименно: Эдуард Айгуаде (Eduard Ayguade), Кристина Бельдика (Cristina Beldica), Константин Бояринов, Хосе Карлос Кабалеиро Домингес (José Carlos Cabaleiro Domínguez), Брэд Чемберлен (Brad Chamberlain), Джеймс Джен-Чань Чен (James Jen-Chang Chen), Джим Коуни (Jim Cowrie), Сергей Диденко, Алехандро (Алекс) Дюран (Alejandro [Alex] Duran), Михаил Дворский, Рудольф Rudolf (Руди) Эйгенман (Rudolf Eigenmann), Джордж Элькоура (George Elkoura), Андрей Федоров, Алексей Федотов, Томас Фернандес Пена (Tomás Fernández Pena), Элвис Фефей (Elvis Fefey), Евгений Фиксман, Базилио Фрагуэла (Basilio Fraguela), Генри Гэбб (Henry Gabb), Хосе Даниэль Гарсиа Санчес (José Daniel García Sánchez), Мария Хесус Гарзаран (Maria Jesus Garzaran), Александр Герवेशи (Alexander Gerveshi), Дарио Суарес Грасиа (Darío Suárez Gracia), Кристина Керманшахче (Kristina Kermanshahche), Янив Клейн (Yaniv Klein), Марк Лубин (Mark Lubin), Антон Малахов, Марк Маклафлин (Mark McLaughlin), Сюзан Мередит (Susan Meredith), Есер Мезиани (Yeser Meziani), Давид Падуа (David Padua), Никита Пономарев, Ануп Мадхусоодханан Прабха (Anoop Madhusoodhanan Prabha), Пабло Ребле (Pablo Reble), Арч Робисон (Arch Robison), Тимми Смит (Timmie Smith), Рубен Гран Техеро (Rubén Gran Tejero), Вазант Товинкере (Vasanth Tovinkere), Сергей Виноградов, Кайл Уилер (Kyle Wheeler) и Флориан Зитцельбергер (Florian Zitzelsberger).

Мы искренне благодарны всем помощникам и приносим извинения тем, кого забыли упомянуть.

Майк (а вместе с ним Рафа и Джеймс!) благодарят всех принимавших участие в работе над ТВВ на протяжении многих лет: многочисленных разработчиков в Intel, оставивших свой след в библиотеке; Алексея Куканова, который рассказывал нам о ее внутреннем устройстве; сообщество разработчиков программ с открытым исходным кодом; технических писателей и специалистов по маркетингу, которые трудились над документацией и распространением информации о ТВВ; технических консультантов и прикладных программистов, которые помогают пользователям применять ТВВ к их задачам; менеджеров, не дававших нам сбиться с пути; а особенно пользователей ТВВ, которые присылали отзывы о библиотеке и ее функциональности, подсказывавшие нам,

в каком направлении двигаться. А больше всех Майк благодарит свою жену Натали и детей, Ника, Али и Люка, за поддержку и терпение на протяжении вечеров и выходных, проведенных в работе над книгой.

Рафа благодарит своих аспирантов и коллег за советы о том, как понятнее донести идеи ТВВ: Хосе Карлоса Ромеро (José Carlos Romero), Франсиско Корберу (Francisco Corbera), Алехандро Виллегаса (Alejandro Villegas), Денизу Андреа КонстантINESКУ (Denisa Andreea Constantinescu), Анжелес Наварро (Angeles Navarro). Особая благодарность Хосе Даниэлю Гарсиа за увлекательные и информативные беседы по поводу C++11, 14, 17 и 20, а также Алексею Федотову и Пабло Ребле (Pablo Reble) за помощь с примерами применения `OpenCL_node`, и прежде всего своей жене Анжелес Наварро за поддержку и выполнение некоторых его обязанностей во время работы над книгой.

Джеймс благодарит свою жену Сюзан Мередит – без ее терпеливой и неослабной поддержки книга была бы невозможна. Ко всему прочему детальная правка, когда за красными чернилами иногда не видно было оригинального текста, сделала ее одним из самых ценных наших рецензентов.

Будучи соавторами, мы не находим слов, чтобы воздать должное друг другу. Майк и Джеймс много лет знают друг друга по работе в Intel, и большая удача, что они сошлись в этом проекте. Трудно выразить, как Майк и Джеймс ценят Рафу! Как же повезло его студентам иметь такого энергичного и знающего профессора! Без Рафы читать эту книгу было бы далеко не так приятно. Благодаря знаниям Рафы о ТВВ книга стала гораздо лучше, а английским он владеет так хорошо, что не раз исправлял ошибки своих англоязычных коллег (Майка и Джеймса). Мы все трое работали над книгой с удовольствием и определенно подстегивали друг друга. Это было прекрасное сотрудничество.

Мы благодарны Тодду Грину (Todd Green), который привел нас в издательство Apress. Спасибо Натали Пао (Natalie Pao) из Apress и Джону Сомоса (John Somoza) из Intel, которые цементировали отношения между Intel и Apress в этом проекте. Мы высоко ценим тяжелый труд всего коллектива Apress над заключением контракта, редактированием и производством.

Спасибо всем!

Майк Восс, Рафаэль Асенхо и Джеймс Рейндерс

# Предисловие

## МЫСЛИТЕ ПАРАЛЛЕЛЬНО

Мы ставили себе целью сделать эту книгу полезной как начинающим, так и искушенным в параллельном программировании. Мы также хотели, чтобы книга была доступна как тем, кто владеет только программированием на С, так и тем, кто уверенно пишет на С++.

Для охвата столь широкой аудитории без «оболванивания» книги мы и написали это предисловие, чтобы уравнять правила игры.

## Что такое ТВВ

ТВВ – это библиотека для написания параллельных программ на С++, ставшая самым популярным решением и могущая похвастаться отличной поддержкой. Она широко используется – и не без причины. Созданная более десяти лет назад, ТВВ прошла испытание временем и учитывалась при включении поддержки параллельного программирования в стандарт С++. Хотя С++11 содержит много добавлений, связанных с параллельным программированием, а С++17 и С++2х продвинулись еще дальше в этом направлении, ТВВ предлагает куда больше, чем стандарт языка. Первая версия ТВВ была выпущена в 2006 году, поэтому библиотека по-прежнему поддерживает компиляторы, предшествующие выходу С++11. Но мы упростили себе задачу, приняв современный взгляд на ТВВ и предполагая, что все функции, описанные в С++11, наличествуют. В наши дни дают такой совет: «если у тебя нет компилятора С++11, поставь его». Если сравнить с книгой о ТВВ, вышедшей в 2007 году, то, на наш взгляд, С++11, а особенно поддержка лямбда-выражений, расширяют функциональность ТВВ, а также упрощают ее понимание и использование.

Проще говоря, ТВВ – лучший способ написать параллельную программу на С++, и мы полагаем, что с ТВВ ваша продуктивность резко возрастет.

## СТРУКТУРА КНИГИ И ПРЕДИСЛОВИЯ

В книге четыре основные части.

1. Предисловие. Базовые сведения, полезные для понимания остальной части книги. Содержит обоснование модели параллельного программирования, выбранной в ТВВ, введение в параллельное программирование, вопросы локальности, кеши, векторные вычисления (набор команд SIMD) и основные средства С++ (сверх имеющихся в языке С), которые поддерживаются или используются ТВВ.

- II. Главы 1–8. Собственно книга о ТВВ. Включает введение в ТВВ в объеме, достаточном для эффективного параллельного программирования.
- III. Главы 9–20. Включает специальные темы, углубляющие понимание ТВВ и параллельного программирования. Рассматриваются тонкие нюансы того и другого.
- IV. Приложения А и В и глоссарий. Собрание полезных сведений о ТВВ, которые могут показаться вам интересными, в том числе история (приложение А) и полное справочное руководство (приложение В).

## Мыслите параллельно

Для незнакомых с параллельным программированием мы написали это введение, в котором излагаются базовые сведения, делающие книгу более понятной, полезной и независимой. Мы предполагаем только владение языком С на базовом уровне и знакомим с ключевыми элементами С++, которые ТВВ поддерживает и на которые опирается. Мы рассматриваем параллельное программирование с практической точки зрения, подчеркивая те черты, которые делают параллельные программы более эффективными. Надеемся, что для опытных программистов это предисловие станет полезным напоминанием о терминологии и способах рассуждений, позволяющих извлекать максимум из параллельного оборудования.

Прочитав предисловие, вы сможете объяснить, что значит «мыслить параллельно» в терминах декомпозиции, масштабирования, корректности, абстрагирования и паттернов. Вы будете понимать, что ключом ко всему параллельному программированию является локальность. Вам раскроется философия программирования на уровне задач, а не на уровне потоков – *революционное достижение концепции параллельного программирования, поддерживаемой ТВВ*. Вы познакомитесь с теми элементами программирования на С++ сверх известного по программированию на С, которые необходимы для использования ТВВ.

Предисловие состоит из пяти частей:

- 1) объяснение мотивов, стоящих за ТВВ;
- 2) введение в параллельное программирование;
- 3) введение в локальность и кеши – аспект оборудования, который, на наш взгляд, неотделим от достижения максимальной производительности средствами параллельного программирования;
- 4) введение в векторизацию (набор команд SIMD);
- 5) введение в языковые средства С++ (сверх унаследованных от С), которые используются или поддерживаются библиотекой ТВВ.

## МОТИВЫ, СТОЯЩИЕ ЗА БИБЛИОТЕКОЙ ТВВ

Библиотека ТВВ появилась в 2006 году. Ее создали специалисты по параллельному программированию из компании Intel, и за плечами многих из них были десятки лет работы с моделями параллельного программирования, в т. ч. OpenMP. Многие члены команды ТВВ потратили годы, чтобы OpenMP могла добиться



впечатляющих успехов, для чего разрабатывали и поддерживали различные реализации OpenMP. Приложение А посвящено истории ТВВ и ее ключевых концепций, в т. ч. прорывной идее планировщиков с заимствованием задач.

Появившись на заре создания многоядерных процессоров, ТВВ быстро превратилась в самую популярную у программистов на C++ модель параллельного программирования. На протяжении первого десятилетия после рождения ТВВ впитывала в себя разнообразные дополнения, сделавшие ее очевидным выбором для параллельного программирования равно у начинающих и опытных пользователей. Будучи проектом с открытым исходным кодом, ТВВ получала отклики и дополнения со всего мира.

ТВВ продвигает революционную идею: параллельное программирование должно дать программисту возможность без колебаний выявлять места, подходящие для распараллеливания, а реализация базовой модели программирования (ТВВ) должна отображать его желания на аппаратные средства во время выполнения.

В основе важности и ценности ТВВ лежит понимание трех вещей: (1) программирование с применением задач, а не потоков; (2) модели параллельного программирования необязательно должны быть запутанными; (3) как добиться масштабируемости, высокой производительности и переносимости при работе с переносимыми моделями параллельного программирования с низкими накладными расходами, примером которых является ТВВ. Далее мы рассмотрим все три этих крайне важных аспекта! Можно с уверенностью сказать, что до того, как они стали краеугольными камнями эффективного и структурированного программирования, их важность долгое время недооценивалась.

## Программирование с применением задач, а не потоков

Программировать параллельно всегда следует в терминах *задач*, а не *потоков*. В конце этого предисловия мы процитируем авторитетный и глубокий анализ этого положения Эдвардом Ли. В 2006 году он заметил: «Чтобы конкурентное программирование стало обыденностью, необходимо отказаться от потоков как модели программирования».

Параллельное программирование в терминах *потоков* – это упражнение на тему отображения приложения на некоторое число параллельных потоков выполнения на той машине, где выполняется программа. Параллельное программирование в терминах *задач* – это упражнение на тему выявления возможных мест для распараллеливания, после чего исполняющая среда (например, среда ТВВ) отображает задачи на оборудование во время выполнения, не вынуждая программиста усложнять логику приложения.

Логический *поток* исполняется аппаратным потоком в течение кванта времени, а в будущих квантах времени может быть назначен другому аппаратному потоку. Модель параллельного программирования в терминах потоков терпит провал, потому что часто используется как взаимно однозначное соответствие между логическими потоками и аппаратными потоками (например, процессорными ядрами). Аппаратный поток – это физическое свойство, от машины к машине меняется их количество, равно как и некоторые тонкие характеристики различных реализаций потоков.

Напротив, *задачи* представляют *возможные места* распараллеливания. Разбиение на задачи можно использовать по мере необходимости с учетом количества доступных аппаратных потоков.

Имея в виду эти определения, можно сказать, что программа, написанная в терминах потоков, должна отображать каждый алгоритм на конкретную систему, состоящую из аппаратного и программного обеспечения. Это не только отвлекает внимание, но и порождает целый ряд проблем, из-за которых параллельное программирование оказывается более трудным, менее эффективным и гораздо менее переносимым.

В то же время программа, написанная в терминах задач, допускает наличие механизма времени выполнения, например исполняющей среды TBB, который отображает задачи на реально имеющееся оборудование. Это позволяет не думать о том, каким количеством аппаратных потоков в действительности располагает система. Но важнее то, что на практике это единственный метод, позволяющий эффективно использовать вложенный параллелизм. Это настолько важная возможность, что мы будем возвращаться к ней в нескольких главах.

## Компонуемость: параллельное программирование необязательно должно быть запутанным

Библиотека TBB обеспечивает *компонуемость* в параллельном программировании, а это меняет все. Компонуемость означает, что мы можем совместно использовать различные средства TBB без ограничений. А самое главное – она допускает вложенность. В частности, ничто не запрещает поместить один цикл `parallel_for` внутрь другого. Из цикла `parallel_for` можно также вызвать подпрограмму, внутри которой имеется другой цикл `parallel_for`.

Поддержка компонуемого вложенного параллелизма в высшей степени желательна, поскольку открывает больше возможностей для распараллеливания, а это, в свою очередь, позволяет создавать более масштабируемые приложения. Например, система OpenMP не является компонуемой относительно вложенности, т. к. каждый уровень вложенности может приводить к значительным накладным расходам и потреблению ресурсов, что станет причиной истощения ресурсов и аварийного завершения программы. Серьезность этой проблемы становится очевидной при попытке использовать библиотечную подпрограмму, содержащую параллельный код. В TBB подобной проблемы нет, поскольку она поддерживает компонуемость. Отчасти она решается благодаря тому, что TBB позволяет программисту указать места распараллеливания (задачи), а сама во время выполнения решает, как отобразить их на аппаратные средства (потоки).

Это важнейшее преимущество кодирования в терминах задач (доступный, но необязательный параллелизм (см. раздел об «ослабленной последовательной семантике» в главе 2)), а не потоков (принудительный параллелизм). Если бы цикл `parallel_for` считался обязательным, то вложенность привела бы к взрывному росту количества потоков вместе с ворохом проблем неконтролируемого потребления ресурсов, которые легко могут вызвать (и часто вы-

зывают) крах программы. Если же `parallel_for` рассматривается как доступный, но необязательный параллелизм, то исполняющая среда вправе использовать эту информацию при выборе наиболее эффективного отображения на аппаратные средства компьютера.

Мы привыкли ожидать компонуемости от языков программирования, но в большинстве моделей параллельного программирования это свойство утрачено (к счастью, ТВВ является исключением!). Рассмотрим, к примеру, предложения `if` и `while`. В языках C и C++ они могут сочетаться и вкладываться как угодно. Но представим себе, что это не так – что мы живем в мире, где функция, вызванная из предложения `if`, не может содержать предложения `while`! Сама мысль о таком ограничении кажется нелепой. ТВВ привносит такого рода компонуемость в *параллельное программирование*, разрешая произвольно сочетать параллельные конструкции без опасения вызвать проблемы.

## Масштабируемость, производительность и погоня за переносимой производительностью

Быть может, самым важным преимуществом программирования с использованием библиотеки ТВВ является то, что она помогает создавать приложения с переносимой производительностью. Мы определяем *переносимую производительность* как характеристику, благодаря которой у программы оказывается похожий «процент пиковой производительности» на различных машинах (с различным оборудованием, разными операционными системами или тем и другим сразу). Мы хотели бы, чтобы высокий процент пиковой производительности имел место на самых разных машинах без необходимости изменять код.

Мы также хотели бы видеть 16-кратный прирост производительности на машине с 64 ядрами по сравнению с четырехъядерной машиной. По различным причинам такое идеальное ускорение на практике почти никогда не наблюдается (но никогда не говори никогда: в некоторых ситуациях благодаря увеличению совокупного размера кеша наблюдается даже более чем идеальное ускорение – это называется *сверхлинейным ускорением*).

### Что такое ускорение?

Изначально ускорение определяется как время последовательного выполнения программы, поделенное на время ее параллельного выполнения. Если обычно моя программа работает 3 с, а на четырехъядерном процессоре всего 1 с, то говорят, что ускорение трехкратное. Иногда употребляют термин *эффективность* – это ускорение, поделенное на количество процессорных ядер. Таким образом, трехкратное ускорение эквивалентно эффективности распараллеливания 75 %.

Идеал – 16-кратный прирост производительности при переходе от четырехъядерной машины к 64-ядерной – называется линейной, или идеальной, *масштабируемостью*.

Для его достижения необходимо обеспечить полную занятость всех ядер при увеличении их количества – цель, требующая большого уровня доступного па-

раллелизма. Понятие доступного параллелизма мы более внимательно рассмотрим ниже при обсуждении закона Амдала и следствий из него.

Пока же важно знать, что ТВВ поддерживает высокопроизводительное программирование и помогает в достижении переносимой производительности. Высокая производительность проистекает из того, что ТВВ не вносит почти никаких накладных расходов, что позволяет беспрепятственно масштабировать программу. Переносимая производительность позволяет приложению задействовать весь доступный параллелизм, предлагаемый современными компьютерами.

Уверенно делая такие заявления, мы предполагали, что незначительность дополнительных накладных расходов на планирование задач обеспечивает максимальную эффективность выявления и использования возможностей распараллеливания. У этого предположения есть один изъян: если мы напишем программу, идеально соответствующую оборудованию, но без возможности динамической подстройки, то, возможно, сумеем увеличить производительность на несколько процентов. Традиционная модель высокопроизводительных вычислений (High-Performance Computing – HPC), применявшаяся для программирования интенсивных массивно параллельных вычислений на самых больших в мире компьютерах, давно уже обладала такой характеристикой. Разработчик, привыкший к HPC, использующий систему OpenMP со статическим планированием и довольный ее производительностью, вероятно, обнаружит, что ТВВ со своей динамичностью несколько снижает производительность. Но преимущества такого статического планирования по различным причинам постепенно сходят на нет. По мере усложнения программ на основе модели HPC требуется поддержка вложенного и динамичного параллелизма. Мы видим это во всех аспектах HPC-программирования: появление больших мультифизических моделей, включение искусственного интеллекта (ИИ) и использование методов машинного обучения (МО). Одна из главных причин дополнительной сложности – применение разнообразного оборудования, в результате чего на одной машине появляется гетерогенная вычислительная среда. ТВВ предлагает средства справиться с этими сложностями, в т. ч. потоковый граф, который мы будем изучать в главе 3.

---

Очевидно, что для эффективного параллельного программирования необходимо разделять выявление мест распараллеливания в форме задач (функция программиста) и отображение задач на аппаратные потоки (функция реализации модели программирования).

---

## **ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ**

Прежде чем сорвать покров тайны с терминологии и основных понятий параллельного программирования, сделаем смелое заявление: параллельное программирование интуитивно более понятно, чем последовательное. Параллелизм окружает нас в повседневной жизни, делать свои дела шаг за шагом – роскошь, которую мы далеко не всегда можем себе позволить. Параллелизм не является чем-то неизведанным и не должен быть таким в программировании.

## Параллелизм вокруг нас

В повседневной жизни мы часто задумываемся о параллелизме. Вот несколько примеров:

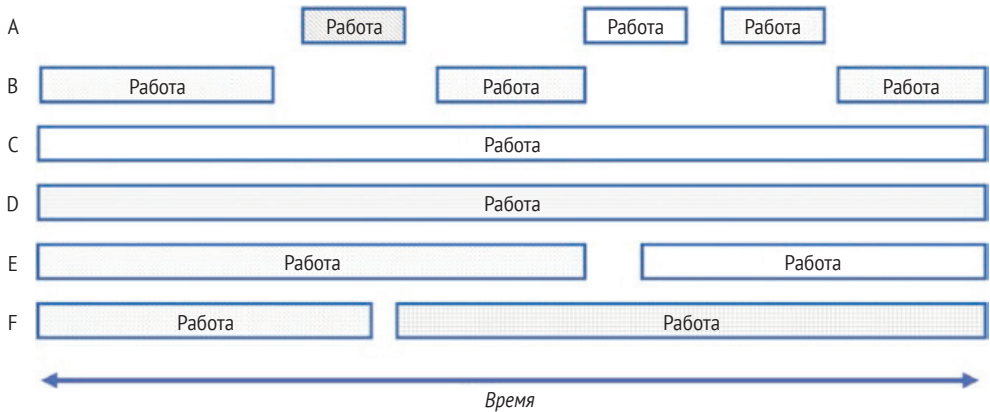
- длинные очереди: стоя в длинной очереди, вы наверняка хотели бы, чтобы было несколько очередей покороче (и побыстрее) или чтобы человек, обслуживающий очередь, работал пошустрее. Очереди на кассе в бакалейной лавке, очереди за билетами на электричку, очереди за кофе – все это хорошо знакомые примеры;
- большой объем однообразной работы: когда перед вами стоит большая задача, над которой одновременно могло бы работать много людей, вы, несомненно, хотели бы иметь побольше помощников. Примерами могут служить перевоз пожитков из старой квартиры в новую, раскладывание писем по конвертам для массовой рассылки или установка одной и той же программы на все новые компьютеры в учебном классе. Пословица «Берись дружно, не будет грузно» относится и к компьютерам.

Начав использовать параллелизм, вы станете Мыслить Параллельно. Вы научитесь сначала думать о возможностях распараллеливания в своем проекте и только потом – о кодировании.

Йель Пат (Yale Pat), знаменитый компьютерный архитектор, как-то заметил:  
 Проблема традиционной мудрости заключается в вере, что  
 Мыслить параллельно трудно,  
 Быть может, потому что мыслить вообще трудно!  
 Как нам убедить людей в том, что  
 Мыслить параллельно естественно  
 (и с этим сложно не согласиться!).

## Конкурентность и параллелизм

Стоит отметить, что термины *конкурентный* и *параллельный* связаны, но имеют тонкое отличие. Конкурентный означает «происходящий в течение одного и того же промежутка времени», а параллельный – «происходящий в одно и то же время (по крайней мере, часть времени)». Конкурентность сродни характеру действий человека, пытающегося решать сразу несколько задач, а параллельность – действиям нескольких людей, работающих вместе. На рис. Р.1 иллюстрируется различие между конкурентностью и параллелизмом. При создании эффективных параллельных программ нашей целью является не просто достижение конкурентности. Когда говорят о конкурентности, часто не ожидают, что значительная часть работы будет выполняться по-настоящему параллельно – т. е. теоретически два исполнителя необязательно выполнят больше работы, чем один (см. задачи А и В на рис. Р.1). Поскольку работа не делается быстрее, конкурентность не уменьшает задержку задачи (время до начала выполнения задачи). Употребление термина *параллельный* подразумевает уменьшение задержки и повышение пропускной способности (объем работы, выполненной в единицу времени). Мы еще вернемся к этому вопросу, когда будем обсуждать пределы параллелизма и очень важный закон Амдала.



**Рис. Р.1** ❖ Параллелизм и конкурентность:  
задачи (А) и (В) конкурентны друг относительно друга, но не параллельны.  
Все остальные комбинации одновременно конкурентны и параллельны

Враги параллелизма: блокировки, разделяемое изменяемое состояние, синхронизация, неумение «мыслить параллельно» и пренебрежение верховенством алгоритма.

## Враги параллелизма

Помня о *врагах параллельного программирования*, вы легче воспримете наши аргументы в пользу определенных методов программирования. Вот перечень главных врагов параллельного программирования.

- Блокировки. В параллельном программировании блокировки, или мьютексы, используются, чтобы предоставить потоку монопольный доступ к ресурсу – запретить другим потокам одновременно обращаться к одному и тому же ресурсу. Блокировки – самый распространенный явный способ гарантировать контролируемое (в отличие от хаотического) обновление разделяемых данных несколькими параллельными потоками. Мы ненавидим блокировки, потому что они сериализуют выполнение некоторых участков программы, ограничивая масштабируемость. Это чувство – ненависть к блокировкам – красной нитью проходит через всю книгу. Мы надеемся внедрить эту мантру и в ваши умы, не упуская при этом из виду необходимость надлежащей синхронизации. Поэтому оговоримся: вообще-то мы любим блокировки, когда они необходимы, поскольку без них грядут страшные беды. Эту смесь любви и ненависти к блокировкам необходимо четко осознать.
- Разделяемое изменяемое состояние. Такое состояние возникает всякий раз, как данные сообща используются несколькими потоками, и их разрешено изменять. Такое разделение (или обобществление) либо уменьшает масштабируемость, если необходима синхронизация и она организована правильно, либо приводит к некорректности (состояниям гонки или взаимоблокировкам), если синхронизация (например, блокировка) организована неправильно. Но надо смотреть на жизнь реально – при написании интересных приложений без разделяемого изменяемого со-



стояния не обойтись. Обдумывание аккуратных способов обращения с ним – простой способ понять наше двойственное отношение к блокировкам. В конечном итоге мы научимся «управлять» разделяемым изменяемым состоянием и взаимным исключением (в т. ч. блокировками), так чтобы они работали, как нам нужно.

- Неумение (или нежелание) мыслить параллельно. Хитроумные заплатки и пластыри не смогут компенсировать отсутствие продуманной стратегии использования масштабируемых алгоритмов. Выявлять места, где возможно распараллеливание, следует до начала реализации. Попытка добавить параллелизм в уже написанное приложение чревато рисками. Иногда существующий код сравнительно легко поддается распараллеливанию, но, как правило, необходимо полностью переосмыслить алгоритмы.
- Пренебрежение верховенством алгоритма. Это еще один способ сказать: «Мысли параллельно». От выбора алгоритмов очень сильно зависит масштабируемость приложения. Выбор алгоритма определяет разбиение на задачи, способ доступа к структурам данных и тактику объединения результатов. Оптимальный алгоритм – основа оптимального решения. Оптимальное решение – это сплав подходящего алгоритма, наилучшим образом соответствующего параллельной структуре данных, и наилучшего способа планирования применяемых к данным вычислений. Поиск лучших алгоритмов – нескончаемая задача всех программистов. А параллельные программисты должны включить в определение *лучшего* еще и *масштабируемость*.

---

Блокировки – и без них никак, и с ними плохо.

---

## Терминология параллелизма

Терминологический словарь параллельного программирования необходимо знать, чтобы общаться с другими программистами. Никаких особенно трудных понятий в нем нет, но очень важно «записать их на подкорку». Любой программист, в т. ч. параллельных приложений (для краткости будем в дальнейшем называть их «параллельными программистами»), тратит годы на овладение своим ремеслом на интуитивном уровне, пусть даже объяснить его основы достаточно просто.

Мы обсудим разложение работы на параллельные задачи, терминологию масштабирования, вопросы корректности и важность локальности, связанную прежде всего с эффектами кеширования.

Как мы находим места для распараллеливания в своем приложении?

На самом верхнем уровне параллелизм существует либо в виде параллельно обрабатываемых данных, либо в виде параллельно исполняемых задач. То и другое *не* является взаимно исключающим. В некотором смысле любой сколько-нибудь важный вид параллелизма – это параллелизм данных. Тем не менее мы вводим оба понятия, поскольку так удобнее рассуждать. При обсуждении масштабируемости и закона Амдала предпочтение, которое мы отдаем *параллелизму данных*, станет более понятным.

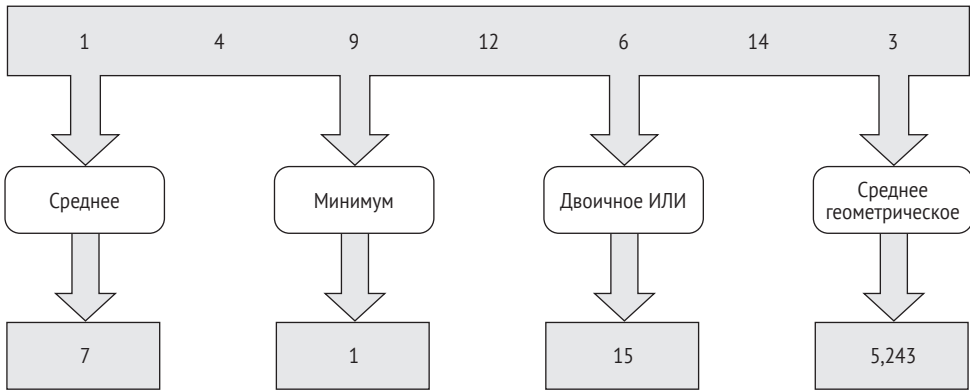


Рис. Р.2 ❖ Параллелизм задач

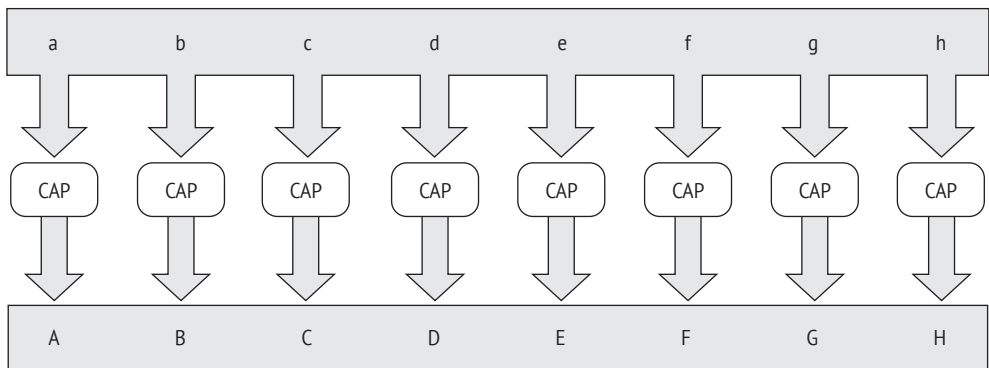


Рис. Р.3 ❖ Параллелизм данных

### **Терминология: параллелизм задач**

*Параллелизм задач* относится к выполнению различных независимых задач. Это показано на рис. Р.2 на примере математических операций, которые можно применить к одному и тому же набору данных для вычисления независимых значений – в данном случае среднего арифметического, минимума, двоичного ИЛИ и среднего геометрического. При нахождении работ, которые можно распараллелить по задачам, мы ограничены количеством полезных независимых операций.

Выше мы ратовали за *задачи* и против *потоков*. Сейчас, сопоставляя параллелизм *данных* и *задач*, мы оказались в двусмысленной ситуации, потому что в этом контексте слово *задача* употребляется в другом смысле. О каком бы параллелизме ни шла речь – *задач* или *данных*, мы все равно будем программировать в терминах *задач*, а не *потоков*. Так уж устроен словарь параллельных программистов.

### **Терминология: параллелизм данных**

*Параллелизм данных* (рис. Р.3) изобразить легко: надо взять много данных и применить в каждой порции одно и то же преобразование. На рис. Р.3 каждая



буква преобразуется в верхний регистр. В этом простом примере мы имеем набор данных и операцию, применяемую к каждому его элементу. Программисты, пишущие код для суперкомпьютеров, обожают такого рода задачи, поскольку распараллелить их настолько просто, что они даже получили специальное название – *естественно параллельные*. Совет: если налицо массовый параллелизм данных, не смущайтесь – пользуйтесь и будьте счастливы. Считайте это *счастливым параллелизмом*.

С точки зрения поиска работы, допускающей распараллеливание, подход на основе параллелизма данных ограничен объемом имеющихся данных. А подходы на основе одного лишь параллелизма задач – числом типов задач. Хотя оба вида параллелизма важны и имеют право на существование, для создания истинно масштабируемой параллельной программы критическим аспектом является выявление параллелизма обрабатываемых данных. Масштабируемость означает, что производительность приложения можно повысить путем добавления оборудования (например, дополнительных процессорных ядер) при условии, что данных достаточно. В век больших данных оказывается, что большие данные и параллельное программирование созданы друг для друга. Мы еще вернемся к этому наблюдению при обсуждении закона Амдала.



Рис. Р.4 ❖ Конвейер



Рис. Р.5 ❖ Представьте себе, что каждая позиция – это новая машина на одном из этапов сборки – именно так конвейер воздействует на проходящие через него данные

### **Терминология: конвейерная обработка**

Хотя обнаружить параллелизм задач труднее, чем параллелизм данных, есть один вид такого параллелизма, который стоит отметить особо: *конвейерная обработка*. В этом случае к потоку данных необходимо применить несколько независимых задач. Каждый элемент подвергается обработке на каждом этапе, что обозначено буквой А на рис. Р.4. При использовании конвейера поток данных можно обработать быстро, потому что разные элементы могут одновременно находиться на разных этапах, как показано на рис. Р.5. В этих примерах время получения окончательного результата (так называемая задержка, измеряемая как время от поступления входных данных до выдачи выходных), возможно, и не уменьшится, но пропускная способность, измеряемая количеством обработанных элементов в единицу времени, возрастет. Благодаря конвейерному параллелизму пропускная способность увеличивается по сравнению с последовательной обработкой. Конвейер может быть устроен и сложнее: он может изменять маршрут прохождения данных или пропускать шаги для некоторых элементов. В ТВВ имеется специальная поддержка для простых конвейеров (глава 2) и для очень сложных конвейеров (глава 3). Разумеется, на каждом шаге конвейера разрешается использовать параллелизм данных или

задач. Благодаря встроенной в ТВВ компонентности с этим не возникает никаких проблем.

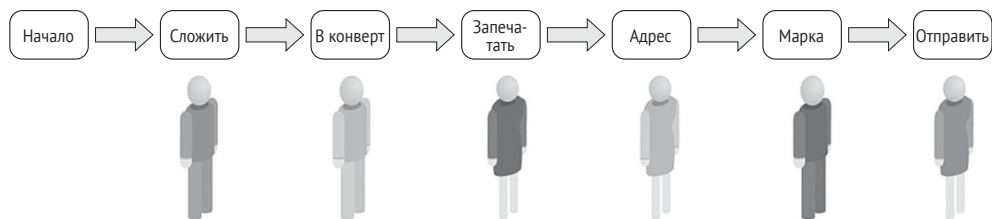


Рис. Р.6 ❖ Конвейер – у каждого человека своя работа

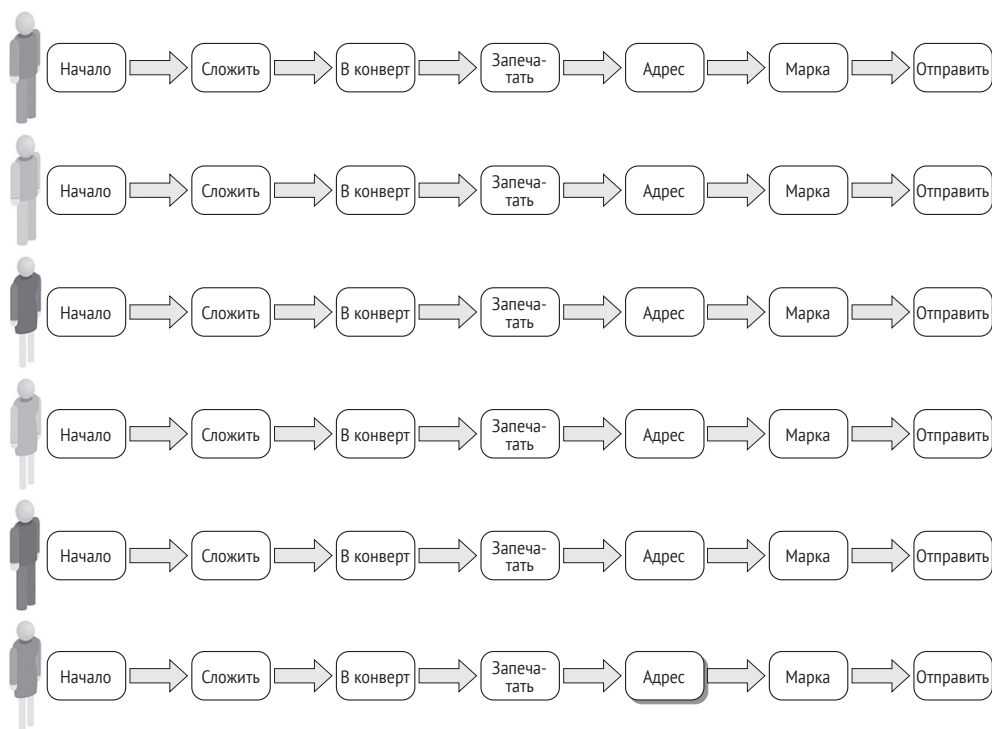


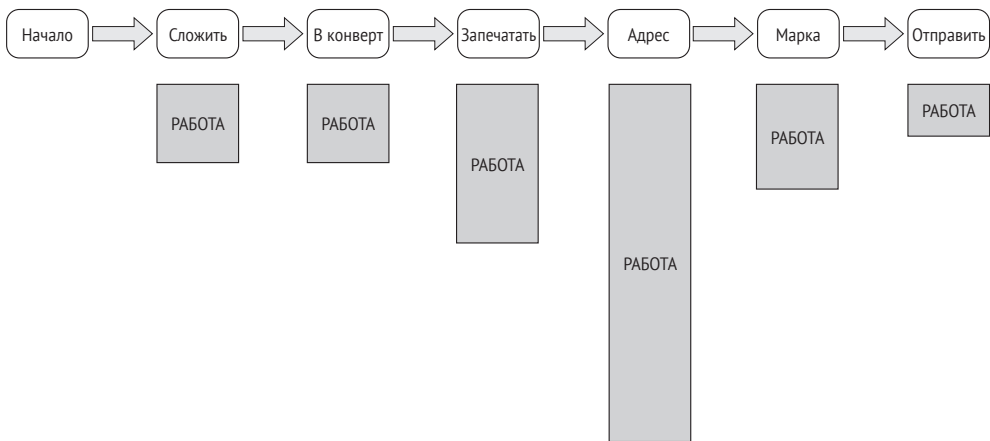
Рис. Р.7 ❖ Параллелизм данных – все выполняют одну и ту же работу

### **Пример использования смешанного параллелизма**

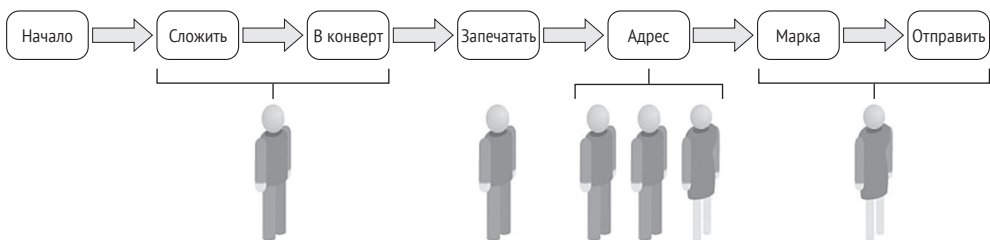
Рассмотрим задачу, состоящую из следующих этапов: складывание письма, его помещение в конверт, запечатывание конверта, написание адреса, наклеивание марки и отправка. Если собрать для ее решения шесть человек, то мы можем поручить каждому одну операцию, организовав конвейер (рис. Р.6). Это прямая противоположность параллелизму данных, когда мы делим все множество писем на равные части и поручаем каждому человеку одну часть (рис. Р.7). После этого каждый человек выполняет все операции с переданными ему материалами.

Ситуацию на рис. Р.7, безусловно, следует предпочесть, если работники находятся далеко друг от друга. Это называется *крупнозернистым* параллелизмом, потому что взаимодействия между задачами происходят редко (работники сходятся вместе, только чтобы получить конверты, а затем расходятся, и каждый занимается своей задачей, включающей и отправку письма). Другой вариант, показанный на рис. Р.6, иллюстрирует *мелкозернистый* параллелизм, когда взаимодействия происходят часто (каждый конверт передается каждому работнику на разных этапах конвейера).

Ни одна из крайностей не отражает реальность точно, хотя иногда аппроксимация может быть достаточно близкой, чтобы оказаться полезной. В нашем примере может случиться, что надписывание конверта занимает так много времени, что для этой операции нужно три человека, тогда как с двумя первыми и двумя последними шагами может справиться один человек. На рис. Р.8 шаги обозначены прямоугольниками, размер которых зависит от трудоемкости работы. Мы можем заключить, что если назначить на каждую операцию ровно одного человека, как на рис. Р.6, то некоторые работники будут простаивать в ожидании работы. Можно сказать, что имеет место скрытая «неполная занятость». Для достижения разумного баланса мы решаем организовать конвейер (рис. Р.9) как гибрид параллелизма данных и задач.



**Рис. Р.8** ❖ Неравные по трудоемкости задачи лучше объединить или разделить с учетом наличных трудовых ресурсов



**Рис. Р.9** ❖ Поскольку задачи не эквивалентны, направляем больше людей на надписывание конвертов

## Достижение параллелизма

Координацию работы людей над подготовкой и отправкой конвертов легко описать следующими двумя концептуальными шагами:

1. Назначить людям задания (и при необходимости перемещать их с одного задания на другое для балансировки загруженности).
2. Вначале назначить на каждое из шести заданий по одному человеку, но быть готовым разбить любую задачу на части, чтобы над ней могло работать несколько людей.

Шесть задач – это складывание письма, помещение его в конверт, запечатывание, надписывание адреса, наклеивание марки и отправка. Для выполнения этой работы в нашем распоряжении имеется шесть человек (ресурсов). Именно так рекомендуется работать с ТВВ: определить задачи и данные на уровне, допускающем объяснение, а затем разбивать или объединять данные в соответствии с наличными ресурсами.

Первый шаг при написании параллельной программы – определить места для распараллеливания. Во многих учебниках параллелизм задач и данных трактуется так, будто между ними имеется четкое разделение. ТВВ допускает любую комбинацию.

---

Мы далеки от осуждения хаоса – мы любим хаос, когда множество некоординируемых задач трудятся над некоторой работой, не утруждаясь взаимной сверкой (синхронизацией). Это так называемое «слабо связанное» параллельное программирование – отличная вещь! Еще больше, чем блокировки, мы ненавидим синхронизацию, т. к. она означает, что задачи должны ждать друг друга. Задачи должны работать, а не простаивать в ожидании!

---

Если нам повезет, то в программе окажется достаточно параллелизма данных – бери и пользуйся. Чтобы упростить эту работу, ТВВ требует от нас только одного: определить задачи и способ их разделения. В случае, когда имеется чистый параллелизм данных, мы определяем одну задачу, которой передаем все данные. Затем задача автоматически разделится, чтобы задействовать весь имеющийся аппаратный параллелизм. Неявная синхронизация (в отличие от синхронизации, которую мы явно запрашиваем в коде) зачастую устраняет необходимость в использовании блокировок. Возвращаясь к списку врагов и нашей ненависти к блокировкам, отметим, что неявная синхронизация – это хорошо. Что мы понимаем под «неявной» синхронизацией? Обычно это значит, что синхронизация имеет место, но явно нигде не упоминается в коде. На первый взгляд, это кажется обманом. Ведь синхронизация все-таки произошла – и кто-то должен был ее запросить! В некотором смысле мы рассчитываем на то, что неявные синхронизации планируются и реализуются более тщательно. Чем чаще мы будем использовать стандартные методы ТВВ и чем реже будем писать собственный код блокировки, тем лучше – в общем случае.

Поручая ТВВ управлять работой, мы делаем ее ответственной за разбиение работы на части и за синхронизацию. А неявная синхронизация, выполняемая библиотекой, часто устраняет необходимость в написании явного кода синхронизации (см. главу 5).

Мы настоятельно рекомендуем этим и ограничиться и прибегать к явной синхронизации (глава 5) только в случае крайней необходимости или когда это приносит очевидную пользу. По своему опыту мы можем сказать, что даже когда кажется, что нечто подобное необходимо, на деле может оказаться не так. Мы вас предупредили. Но если вы похожи на нас, то иногда игнорируете предупреждения и обжигаетесь. С нами так бывало.

Люди применяют декомпозицию много десятилетий, так что уже выработались определенные паттерны. Мы рассмотрим их позже, когда будем обсуждать паттерны параллельного программирования.

---

Эффективное параллельное программирование сводится к тому, чтобы обеспечить все задачи полезной работой в каждый момент времени; выявление и исключение холостого простоя – ключ к основной цели: добиться значительного ускорения.

---

### ***Терминология: масштабируемость и ускорение***

Под масштабируемостью программы понимается мера ускорения ее работы при увеличении вычислительной мощности. Ускорением называется отношение времени работы без параллелизма к времени работы распараллеленной программы. Четырехкратное ускорение (4×) означает, что параллельной программе требуется в четыре раза меньше времени. Например, если последовательной программе на однопроцессорной машине нужно 100 с, то той же программе на четырехъядерной машине понадобится 25 с.

Ожидается, что программа, работающая на двух процессорных ядрах, будет быстрее, чем при работе на одном ядре. Аналогично программа, работающая на четырех процессорных ядрах, должна быть быстрее, чем работающая на двух ядрах.

Для любой программы отдача от увеличения степени параллелизма постепенно уменьшается. Нередко производительность не просто выходит на плато, а даже падает, если мы даем программе слишком много вычислительных ресурсов и требуем, чтобы все они использовались. Уровень зернистости, при котором разбиение работы следует прекратить, называется *степенью детализации* (grain size). В ТВВ это понятие используется, чтобы ограничить разделение данных на части до разумного уровня и тем самым избежать падения производительности. Обычно степень детализации определяется автоматически встроенным в ТВВ разбивателем, который применяет комбинацию эвристик, чтобы определить начальное значение и затем динамически корректировать его по мере выполнения. Но при желании можно управлять степенью детализации и вручную. Мы не поощряем такой подход в этой книге, поскольку явное задание редко позволяет добиться лучшей производительности по сравнению с автоматическим разбивателем. К тому же оптимальное значение зависит от машины, так что явное задание отрицательно сказывается на переносимости производительности.

Когда параллельно мышление выйдет на интуитивный уровень, структурирование программы с учетом масштабируемости станет для вас обыденным делом.

## Сколько параллелизма в приложении?

Вопрос о том, сколько параллелизма можно выдать из приложения, вызывал много споров, а ответ на него зависит от обстоятельств.

Разумеется, это зависит от размера решаемой задачи и от нашей способности найти подходящий алгоритм (и структуры данных), позволяющий задействовать параллелизм. До появления многоядерных процессоров эти споры относились к вопросу о том, как писать эффективные и достойные программы для дорогих и редких параллельных компьютеров. Но с появлением многоядерных процессоров само определение размера, требуемой эффективности и стоимости компьютера изменилось. Мы должны отойти немного назад и оценить, где находимся сейчас. Мир изменился.

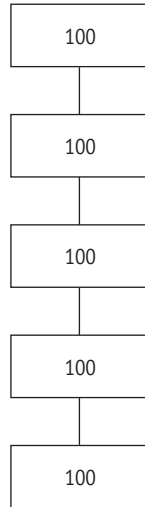
### Закон Амдала

Известный компьютерный архитектор Джин Амдал (Gene Amdahl) сделал наблюдение, касающееся максимального ожидаемого ускорения компьютерной системы в случае, когда улучшается только часть системы. Это наблюдение, сделанное в 1967 г., получило название закона Амдала. Он утверждает, что если ускорить все части программы в два раза, то можно ожидать, что программа станет работать в два раза быстрее. Но если повысить вдвое производительность только  $2/5$  программы, то общая производительность возрастет только в 1,25 раза.

Закон Амдала легко проиллюстрировать наглядно. Представьте себе программу, состоящую из пяти равных частей, которая работает 500 с (рис. P.10). Если ускорить две части в 2 и в 4 раза, как показано на рис. P.11, то вместо 500 с мы получим 400 (ускорение  $1,25\times$ ) и 350 с (ускорение  $1,4\times$ ) соответственно. Чем дальше, тем больше дают о себе знать ограничения тех частей, которые не ускорились в результате распараллеливания. И сколько бы процессорных ядер ни было, последовательные части образуют барьер в 300 с, преодолеть который невозможно (см. рис. P.12), так что максимальное ускорение составляет всего  $1,7\times$ . Если мы можем распараллелить только  $2/5$  части программы, то никогда не увеличим производительность более чем в 1,7 раза!

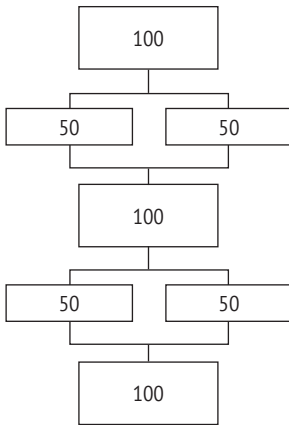
Параллельные программисты давно уже используют закон Амдала для предсказания максимального ускорения, которого можно ожидать при использовании нескольких процессоров. По существу, он говорит, что программа никогда не будет работать быстрее суммы частей, работающих последовательно, сколько процессоров ни добавляй.

Многие, опираясь на закон Амдала, предрекали параллельным компьютерам печальную участь, но на вещи можно взглянуть по-другому, и этот взгляд сулит куда более радужные перспективы.

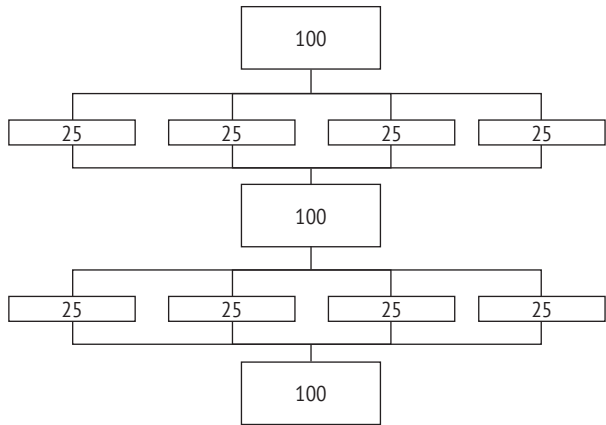


Работа 500 Время 500  
Ускорение 1×

Рис. Р.10 ❖ Исходная программа без параллелизма



Работа 500 Время 400  
Ускорение 1,25×



Работа 500 Время 350  
Ускорение 1,4×

Рис. Р.11 ❖ Постепенное увеличение уровня параллелизма

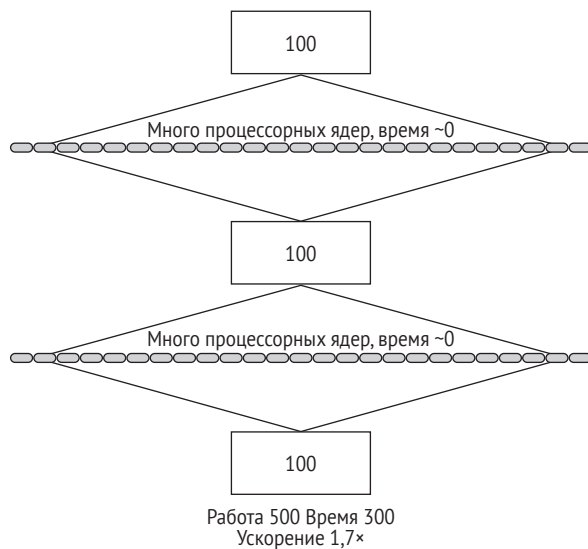


Рис. Р.12 ❖ Ограничения, налагаемые законом Амдала

### **Наблюдения Густафсона, касающиеся закона Амдала**

Закон Амдала считает, что программы фиксированы, а изменениям подвергается сам компьютер. Но опыт показывает, что когда компьютеры приобретают новые возможности, приложения изменяются и начинают ими пользоваться. Большинство современных приложений не смогли бы работать на компьютерах десятилетней давности, а многие плохо работали бы уже на машинах, которым всего пять лет. Это относится не только к таким очевидным случаям, как видеоигры, но и к офисным приложениям, веб-браузерам, программам редактирования фотографий и видео.

С появления закона Амдала прошло больше 20 лет, когда Джон Густафсон, работавший в компании Sandia National Labs, взглянул на проблему под другим углом и предложил пересмотреть выводы из закона Амдала. Густафсон заметил, что параллелизм оказывается более полезен, если учитывать рост рабочей нагрузки со временем. Это означает, что по мере того как компьютеры становятся мощнее, мы поручаем им больше работы, т. е. нагрузка не остается неизменной. Во многих случаях с увеличением размера задачи объем работы, возлагаемой на параллельную часть программы, растет быстрее, чем для части, которая не поддается распараллеливанию. Поэтому с увеличением размера задачи доля последовательной части уменьшается, а, стало быть, согласно закону Амдала, масштабируемость растет. Мы можем начать с приложения, как на рис. Р.10, но если задача масштабируется вместе с располагаемым уровнем параллелизма, то, вероятно, мы увидим улучшения, представленные на рис. Р.13. Если последовательные части занимают столько же времени, сколько раньше, то их процентная доля в общем времени работы программы постепенно снижается. В конечном счете алгоритм достигнет ускорения, показанного на рис. Р.14. Производительность растет с такой же скоростью, как число процессоров ( $n$ ), т. е. мы получаем линейную масштабируемость, что обозначается как  $O(n)$ .