Майкл Восс
Рафаэль Асенхо
Джеймс Рейндерс

Параллельное программирование на C++ с помощью библиотеки TBB

ДМК ПРЕСС
издательство

УДК 004.4
ББК 32.973.202
В76

Восс М., Асенхо Р., Рейндерс Дж.
В76 Параллельное программирование на C++ с помощью библиотеки TBB / 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 C++ программистов, которые хотят научиться работать с библиотекой Threading Building Blocks (TBB). Написанная экспертами по TBB и параллельному программированию, она вобрала в себя их многолетний коллективный опыт разработки и преподавания параллельного программирования с помощью TBB. Излагаемый материал представлен в доступной форме. В книге имеются многочисленные примеры и рекомендации, которые помогут вам в полной мере овладеть TBB и задействовать всю мощь параллельных систем.

Книга начинается с описания базовых параллельных алгоритмов и средств распараллеливания, имеющихся в стандартной библиотеке шаблонов C++. 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 TBB к программированию гетерогенных систем и систем на кристалле.

УДК 004.4
ББК 32.973.202

First published in English under the title «Pro TBB; C++ Parallel Programming with Threading Building Blocks» by Michael Voss, Rafael Asenjo and James Reinders, edition: 1.

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature.

APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

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

Содержание

От издательства ... 14

Об авторах ... 15

Благодарности ... 16

Предисловие ... 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

Глава 1. Приступаем: «Hello, TBB!» ... 60
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 parallel_for ... 80
Добавление уровня SIMD с помощью функции transform из Parallel STL ... 81
Резюме ... 84

Глава 2. Обобщенные параллельные алгоритмы ... 85
Функциональный параллелизм на уровне задач ... 88
Чуть более сложный пример: параллельная реализация быстрой сортировки ... 90
Циклы: parallel_for, parallel_reduce и parallel_scan ... 92
parallel_for: применение тела к каждому элементу диапазона ... 92
parallel_reduce: вычисление одного результата для всего диапазона ... 95
parallel_scan: редукция с промежуточными значениями ... 100
Как это работает? ... 102
Более сложный пример: линия прямой видимости ... 103
Варить до готовности: parallel_do и parallel_pipeline ... 105
parallel_do: применять тело, пока имеются элементы ... 106
parallel_pipeline: обработка несколькими фильтрами ... 113
Резюме ... 120
Дополнительная информация ... 120

Глава 3. Потоковые графы ... 122
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++ ... 145

Какое отношение библиотека STL имеет к этой книге? ... 145
Аналогия для осмысления политик выполнения в Parallel STL ... 147
Простой пример – алгоритм std::for_each ... 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
std::for_each, std::for_each_n ... 156
std::transform ... 158
std::reduce ... 159
std::transform_reduce ... 160
Политики выполнения в деталях ... 162
sequenced_policy ... 162
parallel_policy ... 163
unsequenced_policy ... 163
parallel_unsequenced_policy ... 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 ... 167

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 enumerable_thread_specific ... 190
Тип combinable ... 192

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

Глава 6. Структуры данных для конкурентного программирования....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....224

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 std::allocator<T>....236
scalable_allocator....236
tbb_allocator....237
zero_allocator....237
cached_aligned_allocator....237
Поддержка пула памяти: memory_pool_allocator....238
Поддержка выделения памяти для массивов: aligned_space....238
Избирательная подмена new и delete....239
Настройка производительности: некоторые рычаги управления....242
Что такое большие страницы?....242
Поддержка больших страниц в TBB....242
scalable_allocation_mode(int mode, intptr_t value)....243
TBBMALLOC_USE_HUGE_PAGES....243
TBBMALLOC_SET_SOFT_HEAP_LIMIT....243
int scalable_allocation_command(int cmd, void *param)....244
TBBMALLOC_CLEAN_ALL_BUFFERS....244
TBBMALLOC_CLEAN_THREAD_BUFFERS....244
Резюме....244

Глава 8. TBB и параллельные паттерны 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

Глава 9. Столпы компонуемости 261
Что такое компонуемость? 262
Вложенная композиция 263
Конкурентная композиция 265
Последовательная композиция 266
Благодаря каким особенностям библиотека TBB является компонуемой 268
Пул потоков TBB (рынок) и арены задач 268
Диспетчер задач в TBB: 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

Глава 10. Использование задач для создания собственных алгоритмов 283
Сквозной пример: вычисление последовательности 283
Высокоуровневый подход: parallel_invoke 285
Высший среди низших: task_group 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

Глава 11. Управление количеством потоков 312
Краткий обзор архитектуры планировщика TBB 313
Интерфейсы для управления количеством задач 314
Управление количеством потоков с помощью task_scheduler_init 314
Управление количеством потоков с помощью task_arena 315
Управление количеством потоков с помощью global_control 316
Сводка концепций и классов 316
Рекомендации по заданию количества потоков 317
Использование одного объекта task_scheduler_init в простом приложении 318
Использование нескольких объектов task_scheduler_init в простом приложении 320
Использование нескольких арен с разным числом слотов, чтобы подсказать TBB, куда направлять рабочие потоки 321
Использование global_control для управления количеством потоков, доступных для занятия слотов на аренах 324
Использование global_control с целью временно ограничить количество доступных потоков 326
Когда НЕ следует управлять количеством потоков 328
Что не так? 329
Резюме 330

Глава 12. Применение изоляции работы для обеспечения корректности и повышения производительности 331
Изоляция работ для обеспечения корректности 332
Создание изолированного региона с помощью this_task_arena::isolate 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

Глава 13. Привязка потока к ядру и задачи к потоку 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

Глава 14. Приоритеты задач 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

Глава 15. Отмена и обработка исключений 369
Как отменить коллективную работу 370
Отмена задач в деталях 371
Явное назначение TGC 373

Назначение TGC по умолчанию ... 375
Обработка исключений в TBB ... 379
Написание собственных классов исключений TBB ... 381
Соберем все вместе: компонуемость, 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 static_partitioner ... 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 Pipeline и фильтры, привязанные к потоку ... 414
Резюме ... 418
Дополнительная информация ... 418

Глава 17. Потоковые 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 join_node, sequencer_node или multifunction_node для восстановления порядка в потоковом графе, когда это необходимо ... 445
Полезно: использовать функцию isolate для вложенного параллелизма ... 448
Полезно: использовать отмену и обработку исключений в потоковых графах ... 450
Полезно: задавать приоритеты для графа, в котором используется task_group_context ... 454
Вредно: создавать ребро между узлами разных графов ... 454
Полезно: использовать try_put для передачи информации между графами ... 456
Полезно: использовать composite_node 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

Глава 18. Дополнение потоковых графов асинхронными узлами 471
Пример из асинхронного мира 472
Зачем и когда использовать async_node? 476
Более реалистичный пример 478
Резюме 486
Дополнительная информация 487

Глава 19. Накачанные потоковые графы: узлы OpenCL 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

Глава 20. TBB в системах с архитектурой NUMA 525
Определение топологии платформы 527
Каковы затраты на доступ к памяти 530
Базовый пример 531
Мастерство размещения данных и привязки к процессору 533
Привлекаем hwloc и TBB к совместной работе 538
Более сложные альтернативы 543
Резюме 544
Дополнительная информация 545

Приложение A. История и предшественники 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

Приложение B. TBB в кратком изложении 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

Глоссарий 655

Предметный указатель 668

Глава 1

Приступаем: «Hello, TBB!»

Спустя 10 лет после выпуска первой версии библиотека Threading Building Blocks (TBB) стала одной из самых широко распространенных библиотек для параллельного программирования на C++. Сохранив базовую философию и средства, она продолжает расширяться, включая новые возможности и отвечая на новые вызовы по мере их возникновения.

В этой главе мы обсудим, почему была создана TBB, дадим краткий обзор ее компонентов, опишем, как установить библиотеку, а затем перейдем к нескольким простым примерам.

Почему именно Threading Building Blocks?

Параллельное программирование имеет долгую историю, уходящую в 1950-е годы и еще раньше. На протяжении многих десятилетий ученые разрабатывали крупномасштабные параллельные модели для суперкомпьютеров, а компании создавали приложения масштаба предприятия для больших многопроцессорных машин. Но лишь чуть больше 10 лет назад на рынке появились первые многоядерные кристаллы, предназначенные для настольных компьютеров и ноутбуков. И это радикально изменило ситуацию.

В первых многоядерных системах количество процессоров было невелико – всего два ядра, – но число программистов, вовлеченных в параллельное программирование, было огромным. Если многоядерным процессорам суждено стать обыденностью, то такая же судьба ждет и параллельное программирование, особенно среди тех, кто ценит высокую производительность.

Первая версия библиотеки TBB вышла в сентябре 2006 года в ответ на уникальные вызовы выходящего на широкую арену параллельного программирования. И тогда, и сейчас ее главная цель – предложить разработчикам простой и эффективный способ создавать приложения, которые хорошо масштабировались бы на новые платформы с другими архитектурами и увеличенным количеством ядер. Эта «ориентация на будущее» принесла плоды, когда количество ядер в наиболее популярных процессорах увеличилось с двух в 2006 году до 64 и более в 2018-м!

Чтобы достичь поставленной цели, в основу философии TBB была положена идея сделать так, чтобы разработчику было легко выразить параллелизм своего приложения, ограничив в то же время средства управления отображением этого параллелизма на имеющееся оборудование. Некоторым опытным парал-

лельным программистам эта философия может показаться противоречащей интуиции. Если вы считаете, что параллельное программирование должно любой ценой выжать всю производительность до последней капли и для этой цели годится программирование «на уровне железа», ручная настройка и оптимизация, то TBB, наверное, не для вас – а для тех разработчиков, которые хотят писать высокопроизводительные приложения на сегодняшних платформах, но готовы поступиться толикой производительности ради того, чтобы эти приложения хорошо работали и в будущих системах.

Для этого интерфейсы TBB позволяют выразить параллелизм приложения, но оставляют библиотеке достаточную гибкость, чтобы она могла эффективно отобразить этот параллелизм на текущие и будущие платформы и адаптироваться к динамическому изменению системных ресурсов во время выполнения.

Производительность: низкие накладные расходы, большое преимущество у C++

Мы не хотим придавать чрезмерно большое значение потере производительности и не собираемся отрицать ее. Для простой программы на C++, написанной в стиле Fortran, когда имеются одноуровневые хорошо сбалансированные параллельные циклы, динамическая природа TBB и не нужна вовсе. Однако ограничения такого стиля программирования – важная причина существования TBB. Библиотека TBB спроектирована так, чтобы эффективно поддерживать композицию вложенных, конкурентных и последовательных конструкций и динамически отображать такой параллелизм на целевую платформу. Используя *компонуемую* библиотеку типа TBB, разработчик может создавать приложения путем комбинирования компонентов и библиотек, содержащих внутри себя параллелизм, не заботясь о возможном негативном влиянии этих компонентов друг на друга. Важно, что TBB не требует от нас ограничить возможные виды параллелизма, чтобы избежать проблем. Поэтому для больших и сложных приложений, написанных на C++, TBB можно рекомендовать без всяких оговорок.

Библиотека TBB развивается уже несколько лет, не только поддерживая новые платформы, но и реагируя на пожелания программистов, которые хотят иметь немного больше контроля над решениями, принимаемыми библиотекой при отображении параллелизма на оборудование. Если в TBB 1.0 было совсем мало средств управления производительностью со стороны пользователя, то в TBB 2019 их намного больше, включая управление привязкой к процессору, конструкции для изоляции работ, методы закрепления ядер за потоками и т. д. Разработчики TBB усердно трудятся, чтобы новые средства обеспечили подходящий уровень контроля, не жертвуя компонуемостью.

Предлагаемые библиотекой интерфейсы образуют несколько уровней. В TBB имеются высокоуровневые шаблоны, отвечающие потребностям большинства программистов и покрывающие наиболее распространенные случаи. Но есть также интерфейсы низкого уровня, чтобы можно было создавать решения, точно настроенные на нужды конкретных приложений. TBB берет лучшее

из обоих миров. Как правило, мы полагаемся на то, что библиотека выбирает по умолчанию, но для достижения максимальной производительности можем углубиться в детали.

Эволюция поддержки параллелизма в TBB и C++

С момента появления первой версии TBB и сама библиотека, и язык C++ прошли большой путь. В 2006 году в C++ не было поддержки параллельного программирования на уровне языка, а многие библиотеки, включая STL, было трудно использовать в параллельных программах, т. к. они не были потокобезопасными.

Комитет по языку C++ озаботился включением средств многопоточности прямо в язык и библиотеку STL. На рис. 1.1 показаны новые и планируемые средства C++, относящиеся к параллелизму.

	C++11/14	C++17	C++2x (предложения)
Верхний уровень (события, сообщения, потоковые графы)	`std::async`, `std::future`	`std::async`, `std::future`	Возобновляемые функции, исполнители
Разветвление–соединение, многопоточность	`std::thread` + синхронизация	STL с политикой `par`	Блок задачи, цикл `for`
SIMD/векторизация	ничего	STL с политикой `par_unseq`	STL с политиками `unseq` и `vec`, векторные типы SIMD

Рис. 1.1 ❖ Средства, определенные в стандарте C++, и новые предложения

Несмотря на всю нашу любовь к TBB, мы предпочли бы, чтобы вся фундаментальная поддержка параллелизма была встроена в сам язык C++. Это позволило бы TBB опереться на прочное основание и уже на нем возводить высокоуровневые абстракции параллелизма. В первых версиях TBB приходилось компенсировать отсутствие поддержки на уровне языка, но в этой области стандарт C++ значительно усовершенствован и теперь заполняет те пробелы, которые раньше приходилось восполнять TBB, например переносимые блокировки и атомарные типы. К сожалению для разработчиков на C++, в стандарте все еще недостает некоторых средств для полной поддержки параллельного программирования. Но к радости читателей, это означает, что TBB по-прежнему актуальна и важна для эффективного программирования многопоточности на C++ и, вероятно, останется таковой еще много лет.

Очень важно понимать, что мы не сетуем по поводу процесса разработки стандарта C++. При добавлении новых средств в стандарт языка следует продвигаться с большой осторожностью и тщательно анализировать все предложения. Например, комитет по стандартизации C++11 потратил очень много усилий на утверждение модели памяти. Этот вопрос критически важен для любой библиотеки параллельного программирования, основанной на стандарте. Существуют также некоторые пределы тому, что должен включать и поддерживать стандарт языка. Мы полагаем, что система задач и потоковые графы, присутствующие в TBB, не должны входить в стандарт напрямую. Даже если мы не правы, случится это очень не скоро.

Недавние добавления в C++, относящиеся к параллелизму

Как показано на рис. 1.1, в стандарт C++11 включены некоторые низкоуровневые средства поддержки многопоточности, в т. ч. `std::async`, `std::future` и `std::thread`. Включены также атомарные переменные, мьютексы и условные переменные. Чтобы с помощью этих расширений создать высокоуровневые абстракции, программисту придется немало поработать, но они все же позволяют выразить простой параллелизм непосредственно на C++. Стандарт C++11, безусловно, стал значительным достижением с точки зрения многопоточности, но он не предоставляет высокоуровневых средств, позволяющих легко писать переносимый и эффективный параллельный код. В нем также отсутствуют задачи и планировщик с заимствованием работ.

Стандарт C++17 поднял уровень абстракции на новую высоту, позволив выражать параллелизм, не беспокоясь о мельчайших низкоуровневых деталях. Но, как мы увидим ниже, по-прежнему имеются существенные ограничения, так что новые средства пока недостаточно выразительны и не могут похвастаться высочайшей производительностью – авторам стандарта C++ предстоит еще много работы.

Из добавлений в C++17 нас больше всего интересуют *политики выполнения*, которые можно использовать совместно с алгоритмами из библиотеки STL. Они позволяют выбрать режим работы алгоритма: безопасно распараллелить, векторизовать, распараллелить и векторизовать или же сохранить первоначальную последовательную семантику. Реализацию STL, поддерживающую эти политики, мы называем Parallel STL.

В будущем в стандарт C++, возможно, будут включены дополнительные средства параллелизма, как то: возобновляемые функции, исполнители, блоки задач, параллельные циклы for, векторные типы SIMD и дополнительные политики выполнения для алгоритмов STL.

Библиотека Threading Building Blocks (TBB)

Threading Building Blocks (TBB) – написанная на C++ библиотека, выполняющая две основные функции:

1) заполняет пробелы в поддержке параллелизма там, где стандарт C++ еще недостаточно развит или новые средства не полностью поддерживаются компиляторами;
2) предоставляет высокоуровневые абстракции параллелизма, которые, вероятно, никогда не будут включены в стандарт C++. На рис. 1.2 показаны средства, реализованные TBB.

Эти средства можно отнести к двум большим группам: интерфейсы для выражения параллельных вычислений и интерфейсы, не зависящие от модели выполнения.

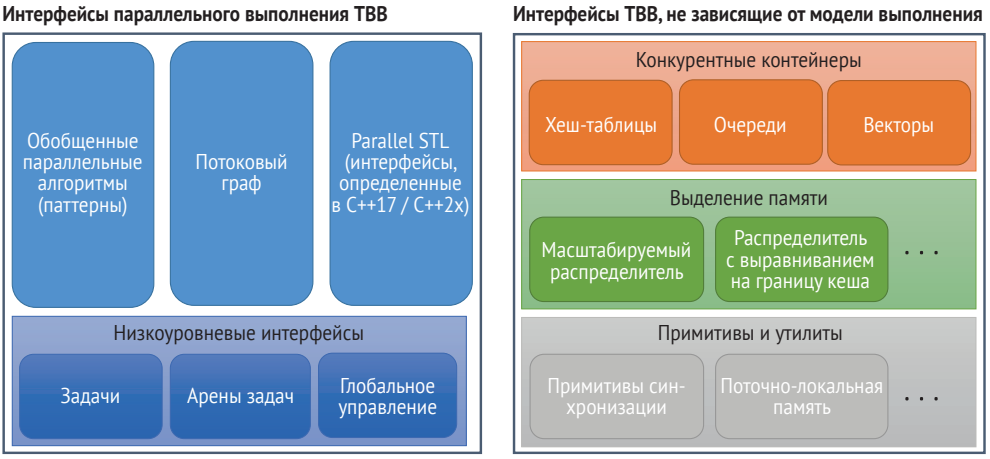


Рис. 1.2 ❖ Средства библиотеки TBB

Интерфейсы параллельного выполнения

Используя TBB для создания параллельных программ, мы выражаем параллелизм приложения с помощью одного из высокоуровневых интерфейсов или непосредственно в виде задач. Задачи мы подробно обсудим ниже, а пока будем рассматривать TBB-задачу как облегченный объект, который определяет небольшое вычисление и связанные с ним данные. Работая с TBB, мы описываем свое приложение с помощью задач – прямо или опосредованно через включенные в TBB алгоритмы, – а библиотека планирует, как выполнить эти задачи, располагая предлагаемыми платформой аппаратными ресурсами.

Важно отметить, что у разработчика может возникнуть желание выразить различные виды параллелизма. На рис. 1.3 показаны три самых распространенных уровня параллелизма в приложениях. Одни приложения включают все три уровня, другие – только один или два. Один из самых «крутых» аспектов TBB – предоставление высокоуровневых интерфейсов для всех трех уровней, что позволяет использовать одну библиотеку во всех случаях.

Уровень обмена сообщениями выражает параллелизм в случае, когда программа структурирована как относительно крупные вычисления, обменивающиеся между собой явно определенными сообщениями. На этом уровне возникают такие паттерны, как потоковые диаграммы, графы потоков данных и графы зависимостей. В TBB все они поддерживаются интерфейсами потокового графа (Flow Graph), описанными в главе 3.

Уровень разветвления–соединения включает паттерны, в которых последовательное вычисление разветвляется на множество параллельных задач и продолжается, после того как все параллельные подвычисления завершатся. Примерами таких паттернов могут служить функциональный параллелизм (параллелизм задач), параллельные циклы, параллельная редукция и конвейеры. TBB поддерживает их с помощью обобщенных параллельных алгоритмов (глава 2).

Наконец, на уровне SIMD параллелизм выражается в применении одной и той же операции одновременно к нескольким элементам данных. Этот тип

параллелизма часто реализуется с помощью таких векторных расширений, как AVX, AVX2 или AVX-512, которые используют устройства векторных вычислений, имеющиеся в каждом процессорном ядре. Библиотека Parallel STL (глава 4), включенная во все дистрибутивы TBB, содержит среди прочего векторные реализации, опирающиеся на эти расширения.

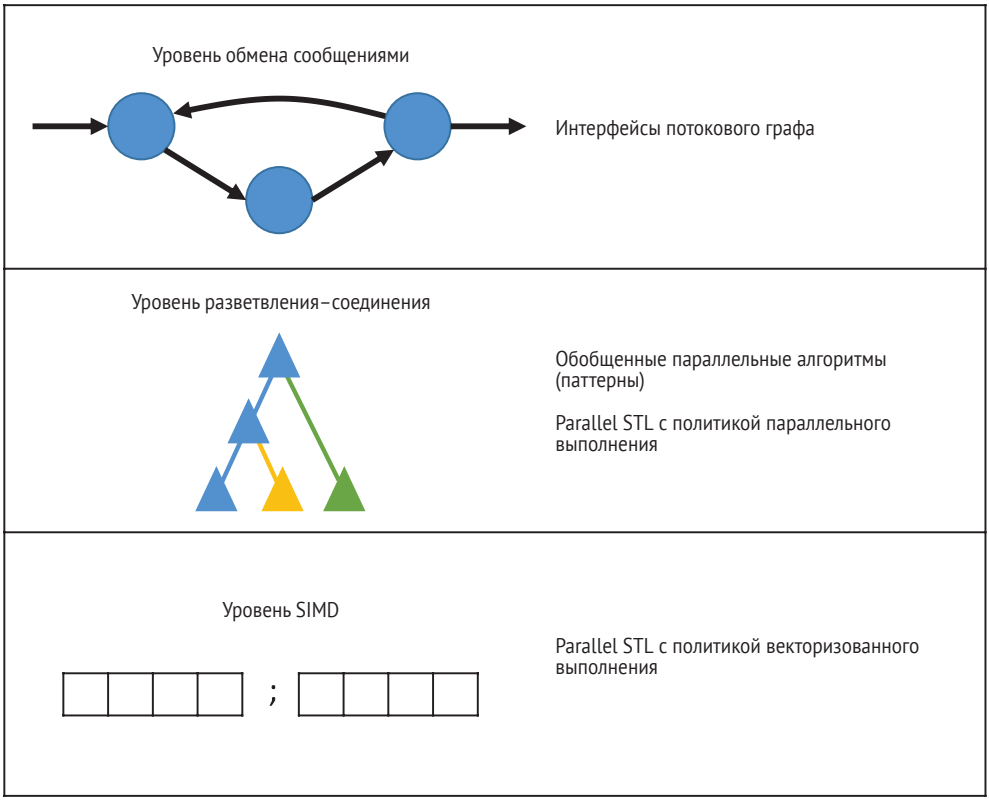


Рис. 1.3 ❖ Три уровня параллелизма, часто встречающихся в приложениях, и их отображение на высокоуровневые интерфейсы параллельного выполнения в TBB

TBB предлагает высокоуровневые интерфейсы для многих распространенных параллельных паттернов, но все же бывают случаи, когда ни один готовый интерфейс не подходит. Тогда можно напрямую использовать в алгоритмах задачи TBB.

Истинная мощь интерфейсов параллельного выполнения в TBB проистекает из возможности комбинировать их, это свойство обычно называют «компонуемостью». Мы можем создавать приложения, на верхнем уровне которых находится потоковый граф, а в его узлах – вложенные обобщенные параллельные алгоритмы. Каждый такой алгоритм может при необходимости включать алгоритмы из библиотеки Parallel STL. Поскольку параллелизм, выражаемый с помощью любого из трех уровней, виден библиотеке TBB, она может эффективно планировать соответствующие задачи так, что они будут допускать композицию и использовать платформенные ресурсы оптимальным способом.

Одно из основных свойств TBB, благодаря которому она компонуема, – поддержка *ослабленной последовательной семантики*. Это означает, что параллелизм, выраженный средствами TBB, – не более чем подсказка библиотеке, и нет никакой гарантии, что любая конкретная задача будет выполняться параллельно с другими. Это открывает перед TBB невероятно гибкие возможности для планирования задач с целью повышения производительности. Благодаря этой гибкости библиотека обеспечивает переносимую производительность на разных системах – с одним, 8 или 80 ядрами. Она также позволяет библиотеке адаптироваться к динамической нагрузке на платформу; например, если одно ядро перегружено работой, то TBB может запланировать больше работы на других ядрах или даже решить, что параллельный алгоритм будет выполняться только одним ядром. Подробнее о том, почему TBB считается компонуемой библиотекой, мы поговорим в главе 9.

Интерфейсы, не зависящие от модели выполнения

В отличие от интерфейсов параллельного выполнения, средства из второй большой группы на рис. 1.2 вообще не зависят от модели выполнения и задач TBB. Эти средства полезны в приложениях, где используются платформенные потоки, например `pthreads` или `WinThreads`, а также в приложениях с задачами TBB.

К числу этих средств относятся конкурентные контейнеры, которые предоставляют потокобезопасные интерфейсы к таким популярным структурам данных, как хеш-таблицы, очереди и векторы. Сюда же относятся средства выделения памяти – масштабируемый распределитель памяти TBB и распределитель с выравниванием на границу кеша (оба описаны в главе 7). В эту группу входят и такие низкоуровневые средства, как примитивы синхронизации и поточно-локальная память.

Использование строительных блоков в TBB

Разработчик может выбирать, какие части TBB полезны в его приложении. Например, можно использовать масштабируемый распределитель памяти (глава 7) и больше ничего. Или воспользоваться конкурентными контейнерами (глава 6) и несколькими обобщенными параллельными алгоритмами (глава 2). И конечно, мы вправе построить приложение, в котором сочетаются все три высокоуровневых интерфейса выполнения, и пользоваться масштабируемым распределителем памяти, конкурентными контейнерами и многими другими возможностями, предлагаемыми библиотекой.

Да начнем же уже!

Получение библиотеки TBB

Первым делом мы должны откуда-то получить копию библиотеки. Путей несколько, перечислим те, что существовали на момент написания книги.

- Перейти по ссылкам на сайте www.threadingbuildingblocks.org или https://software.intel.com/intel-tbb и получить бесплатную библиотеку TBB непосредственно от Intel. Существуют готовые откомпилированные версии для Windows, Linux и macOS. В самые последние пакеты включена как сама TBB, так и реализация алгоритмов из библиотеки Parallel STL, в которой для организации многопоточности используется TBB.
- Зайти на сайт https://github.com/intel/tbb и скачать бесплатную версию TBB с открытым исходным кодом. Эта не облегченная версия, она включает все те же средства, что и коммерческая. Вы можете извлечь исходный код из репозитория и собрать библиотеку самостоятельно или просто щелкнуть по ссылке «releases» и скачать версию, уже собранную и протестированную Intel. На GitHub имеются собранные и протестированные версии для Windows, Linux, macOS и Android. Как и в первом случае, последние пакеты содержат как саму библиотеку TBB, так и реализацию Parallel STL, в которой используется TBB. Если вам нужен исходный код Parallel STL, скачайте его отдельно на странице https://github.com/intel/parallelstl.
- Можно скачать комплект инструментов Intel Parallel Studio XE с сайта https://software.intel.com/intel-parallel-studio-xe. В настоящее время TBB и версия Parallel STL, использующая TBB, включены во все издания этого комплекта, в т. ч. самое маленькое – Composer Edition. Если на вашей машине установлена недавняя версия компилятора Intel C++, то, вероятно, установлена и TBB.

Предлагаем читателям самостоятельно выбрать подходящий способ получения TBB и выполнить инструкции по установке пакета, выложенные на соответствующем сайте.

Получение кода примеров

Код всех примеров, встречающихся в этой книге, выложен по адресу https://github.com/Apress/pro-TBB. В этом репозитории каждой главе соответствует отдельный каталог. Многие исходные файлы названы по номеру соответствующего рисунка, например файл `ch01/fig_1_04.cpp` содержит код, приведенный на рис. 1.4 в этой главе.

Написание первого примера «Hello, TBB!»

На рис. 1.4 приведен небольшой пример, в котором функция `tbb::parallel_invoke` выполняет две функции, одна из которых печатает `Hello`, а вторая параллельно печатает `TBB!`. Пример тривиален и от распараллеливания ничего не выигрывает, но позволяет убедиться, что среда правильно настроена для работы TBB. На рис. 1.4 включен заголовок tbb.h, открывающий доступ к функциям и классам TBB, которые находятся в пространстве имен `tbb`. Обращение к `parallel_invoke` сообщает библиотеке TBB, что обе переданные функции независимы друг от друга, поэтому их можно безопасно выполнять параллельно на разных ядрах или в разных потоках в любом порядке. При таких ограничениях первым может быть напечатано как слово `Hello`, так и слово `TBB!`. Может даже случиться,

что между этими словами нет знака новой строки, а оба таких знака оказались напечатаны в самом конце, поскольку печать строки и следующего за ней символа `std::endl` – не атомарная операция.

```
#include <iostream>
#include <tbb/tbb.h>

int main() {
  tbb::parallel_invoke(
    []() { std::cout << " Hello " << std::endl; },
    []() { std::cout << " TBB! " << std::endl; }
  );
  return 0;
}
```

Рис. 1.4 ❖ Пример Hello TBB

На рис. 1.5 приведен пример, в котором алгоритм Parallel STL `std::for_each` применяет функцию параллельно к двум элементам `std::vector`. Передача политики `pstl::execution::par` алгоритму `std::for_each` подтверждает безопасность применения указанной функции параллельно на разных ядрах или в разных потоках к результату разыменования каждого итератора в диапазоне `[v.begin(), v.end())`. Как и в предыдущем примере, первой может быть напечатана любая из двух строк.

```
#include <pstl/algorithm>
#include <pstl/execution>
#include <iostream>
#include <vector>

int main() {
  std::vector<std::string> v = { " Hello ", " Parallel STL! " };
  std::for_each(pstl::execution::par, v.begin(), v.end(),
    [](std::string& s) { std::cout << s << std::endl; }
  );
  return 0;
}
```

Рис. 1.5 ❖ Пример Hello Parallel STL

На обоих рис. 1.4 и 1.5 для задания функций используются *лямбда-выражения C++*. При работе с библиотеками типа TBB лямбда-выражения очень полезны, когда нужно задать пользовательский код, выполняемый как задача. На врезке «Введение в лямбда-выражения C++» приведен краткий обзор этой важной особенности современного C++.

Введение в лямбда-выражения C++

Поддержка лямбда-выражений впервые появилась в C++11. Они служат для создания анонимных объектов-функций (которые при желании можно присвоить именованным переменным), способных захватывать переменные из объемлющей области видимости.

Базовый синтаксис лямбда-выражений выглядит так:

[*список-захвата* **] (** *параметры* **) ->** *возвр* **{** *тело* **}**

где:

- *список-захвата* – список захватываемых переменных через запятую. Чтобы захватить переменную по значению, ее имя следует указать в *списке-захвата*. Чтобы захватить переменную по ссылке, ее имени нужно предпослать знак амперсанда, например `&v`. Этот способ можно также использовать для захвата по ссылке текущего объекта `this`. Существуют также умолчания: [=] означает захват всех автоматических переменных, встречающихся в теле, по значению, а текущего объекта – по ссылке; [&] означает захват по ссылке всех автоматических переменных, встречающихся в теле, и текущего объекта; [] означает, что не нужно захватывать ничего;
- *параметры* – список параметров, такой же, как для именованной функции;
- *возвр* – тип возвращаемого значения. Если часть ->*возвр* опущена, она выводится из предложений `return` в теле функции;
- *тело* – тело функции.

В следующем примере показано лямбда-выражение C++, захватывающее переменную `i` по значению, а переменную `j` по ссылке. У него также имеется параметр `k0`, передаваемый по значению, и параметр `l0`, передаваемый по ссылке.

```
int i = 1, j = 10, k = 100, l = 1000;
auto lambdaExpression = [i, &j] (int k0, int &l0) -> int {
  j = 2 * j;
  k0 = 2 * k0;
  l0 = 2 * l0;
return i + j + k0 + l0;
  };

printValues(i, j, k, l);
std::cout << "First call returned " << lambdaExpression(k, l) << std::endl;
printValues(i, j, k, l);
std::cout << "Second call returned " << lambdaExpression(k, l) << std::endl;
printValues(i, j, k, l);
return 0;
```

При выполнении этого примера печатается:

```
i == 1
j == 10
k == 100
l == 1000
First call returned 2221
i == 1
j == 20
k == 100
l == 2000
Second call returned 4241
i == 1
j == 40
k == 100
l == 4000
```

Лямбда-выражение можно рассматривать как экземпляр объекта-функции, только определение класса создает компилятор. Например, рассмотренное выше лямбда-выражение эквивалентно экземпляру такого класса:

```
class Functor {
  int my_i;
  int &my_jRef;

public:
  Functor(int i, int &j) : my_i{i}, my_jRef{j} { }

  int operator()(int k0, int &l0) {
    my_jRef = 2 * my_jRef;
    k0 = 2 * k0;
    l0 = 2 * l0;
    return my_i + my_jRef + k0 + l0;
  }
};
```

Всякий раз, видя лямбда-выражение C++, мы можем подставить вместо него экземпляр объекта-функции, аналогичного показанному выше. На самом деле библиотека TBB предвосхитила стандарт C++11 и все его интерфейсы, в которых требуется передавать экземпляры объектов пользовательских классов. Лямбда-выражения C++ упрощают работу с TBB, устраняя лишний шаг – определение отдельного класса для каждого использования алгоритма TBB.

Сборка простых примеров

Написав код примеров на рис. 1.4 и 1.5, мы должны собрать для них исполняемые файлы. Инструкции по сборке приложения с использованием TBB зависят от ОС и компилятора. Но всегда есть два шага, необходимых для правильной настройки среды.

Настройка среды

1. Мы должны сообщить компилятору, где находятся заголовки и библиотеки, входящие в состав TBB. Если используются интерфейсы библиотеки Parallel STL, необходимо еще сообщить, где находятся ее заголовки.
2. Следует настроить среду, так чтобы запущенное приложение могло найти библиотеку TBB. TBB поставляется в виде динамически компонуемой библиотеки, т. е. не встраивается непосредственно в приложение; вместо этого приложение находит и загружает ее на этапе выполнения. Для интерфейсов Parallel STL не нужна собственная динамически компонуемая библиотека, но они зависят от библиотеки TBB.

Теперь мы кратко обсудим, как эти шаги выполняются в Windows и в Linux. Инструкции для macOS такие же, как для Linux. В документации, поставляемой вместе с TBB, описаны дополнительные случаи и даны более подробные указания.

Сборка в Windows в Microsoft Visual Studio

Если была скачана версия TBB с коммерческой поддержкой или версия Intel Parallel Studio XE, то библиотеку TBB можно интегрировать с Microsoft Visual Studio на этапе установки, а затем без труда использовать из Visual Studio.

Для создания проекта «Hello, TBB!» создайте проект в Visual Studio как обычно, добавьте cpp-файл с кодом на рис. 1.4 или 1.5, а затем перейдите на страницы свойств проекта, выполните команду **Configuration Properties ➤ Intel Performance Libraries** (Свойства конфигурации ➤ Intel Performance Libraries) и измените флажок **Use TBB** (Использовать TBB) на **Yes** (Да), как показано на рис. 1.6. На этом шаг 1 можно считать выполненным. Теперь Visual Studio будет компоновать проект с библиотекой TBB, поскольку знает о путях к заголовочным файлам и библиотекам. При этом также устанавливаются правильные пути к заголовкам Parallel STL.

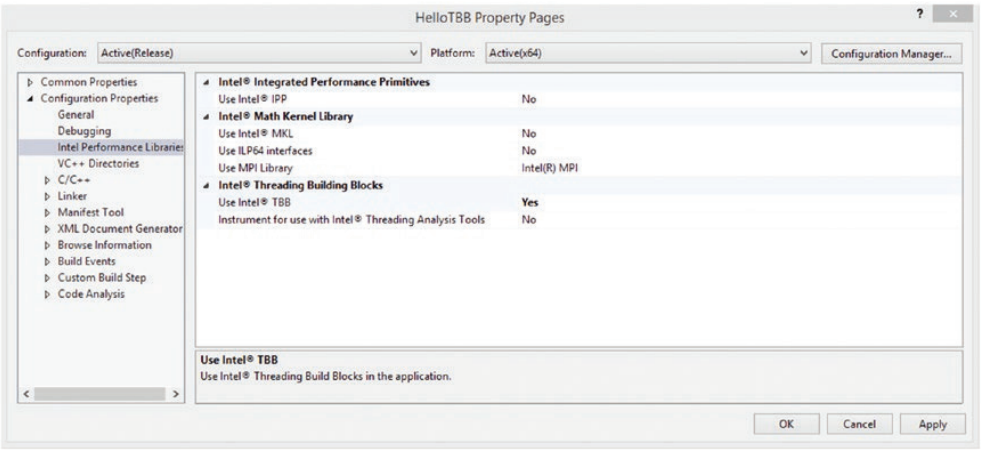


Рис. 1.6 ❖ Задание флажка **Использовать TBB** в страницах свойств проекта в Visual Studio

В системах Windows библиотеки TBB, которые динамически загружаются исполняемым файлом приложения на этапе выполнения, – это dll-файлы. Чтобы выполнить шаг 3 подготовки среды, необходимо добавить путь к этим файлам в переменную среды PATH. Годится как пользовательская, так и системная переменная PATH. Один из способов найти эти параметры – открыть панель управления Windows, пройти по пути **Система ➤ Дополнительные параметры системы ➤ Переменные среды**. Где находятся dll-файлы TBB, можно узнать из документации.

Примечание. Измененная переменная PATH становится видна среде Microsoft Visual Studio только после ее перезагрузки.

Итак, исходный код введен, флажок **Использовать TBB** установлен в **Да**, и путь к dll-файлам TBB прописан в переменной PATH. Теперь можно собрать и выполнить программу, нажав **Ctrl**+**F5**.

Сборка на платформе Linux из терминала

Использование компилятора Intel

При использовании компилятора Intel C++ процесс компиляции упрощается, потому что библиотека входит в дистрибутив компилятора и поддерживается флаг `-tbb`, который правильно устанавливает пути к библиотеке и заголовочным файлам. Поэтому для компиляции примеров нужно только добавить в командную строку флаг `-tbb`:

```
icpc -std=c++11 -tbb -o fig_1_04 fig_1_04.cpp
icpc -std=c++11 -tbb -o fig_1_05 fig_1_05.cpp
```

Скрипты tbbvars и pstlvars

Если компилятор Intel C++ не используется, то для настройки среды можно использовать скрипты, входящие в состав дистрибутивов TBB и Parallel STL. Эти скрипты модифицируют переменные среды `CPATH`, `LIBRARY_PATH` и `LD_LIBRARY_PATH`, включая в них каталоги, необходимые для сборки и выполнения приложений с TBB и Parallel STL. В переменную `CPATH` добавляются каталоги, которые компилятор просматривает, когда ищет заголовочные файлы. В переменную `LIBRARY_PATH` добавляются каталоги, которые компилятор просматривает, когда ищет библиотеки, компонуемые на этапе сборки. А в переменную `LD_LIBRARY_PATH` добавляются каталоги, которые компилятор просматривает, когда ищет библиотеки, динамически загружаемые на этапе выполнения.

Предположим, что корневой каталог нашей установки TBB записан в переменную среды `TBB_ROOT`. В дистрибутив библиотеки TBB входит набор скриптов в каталоге `${TBB_ROOT}/bin`, при выполнении которых правильно настраивается среда. Скрипту нужно передать тип архитектуры [`ia32|intel64|mic`]. Кроме того, на этапе компиляции нужно указать флаг, разрешающий использовать средства из стандарта C++11, в частности лямбда-выражения.

Хотя заголовки Parallel STL включаются во все недавние дистрибутивы библиотеки TBB, все равно необходим дополнительный шаг, чтобы добавить их в среду. Как и для TBB, для Parallel STL имеется набор скриптов в каталоге `${PSTL_ROOT}/bin`. Обычно каталог `PSTL_ROOT` расположен на одном уровне с `TBB_ROOT`. Для использования Parallel STL необходимо также передать тип архитектуры и разрешить использование средств C++11.

Шаги сборки и выполнения примера 1.4 на платформе Linux с 64-разрядным процессором Intel выглядят так:

```
source ${TBB_ROOT}/bin/tbbvars.sh intel64 linux auto_tbbroot
g++ -std=c++11 -o fig_1_04 fig_1_04.cpp -ltbb
./fig_1_04
```

Шаги сборки и выполнения примера 1.5 на той платформе выглядят так:

```
source ${TBB_ROOT}/bin/tbbvars.sh intel64 linux auto_tbbroot
source ${PSTL_ROOT}/bin/pstlvars.sh intel64 auto_pstlroot
g++ -std=c++11 -o fig_1_05 fig_1_05.cpp -ltbb
./fig_1_05
```

Примечание. Все больше дистрибутивов Linux уже включают библиотеку TBB. На этих платформах компилятор GCC может осуществлять компоновку с платформенной версией TBB, а не с той, которая включена в переменную LIBRARY_PATH скриптом tbbvars. Если возникают проблемы с использованием TBB, то причина может быть именно в этом. В таком случае можно добавить явный путь к библиотеке в командную строку вызова компилятора, указав конкретную версию TBB, например:

```
g++ -L${TBB_ROOT}/lib/intel64/gcc4.7 -ltbb ...
```

Если включить в командную строку g++ флаги -Wl,--verbose, то будет выдан отчет обо всех скомпонованных библиотеках, что поможет диагностировать эту проблему.

Хотя в примерах выше приведены команды для g++, они будут такими же (за исключением названия компилятора) для компиляторов Intel (icpc) и LLVM (clang++).

Задание переменных среды вручную без использования скрипта tbbvars или компилятора Intel

Иногда мы не хотим использовать скрипт tbbvars – то ли потому, что не знаем точно, какие переменные он устанавливает, то ли потому, что нужно интегрироваться с системой сборки. Если к вам это не относится, можете пропустить этот раздел и вернуться к нему, когда возникнет необходимость в ручной настройке.

Но раз уж вы читаете данный раздел, давайте посмотрим, как произвести сборку и выполнение из командной строки, не прибегая к скрипту tbbvars. Ни один компилятор, кроме Intel, не поддерживает флаг -tbb, так что придется задавать пути к заголовкам и разделяемым библиотекам TBB.

Если корневой каталог установки TBB записан в переменную среды TBB_ROOT, то заголовки находятся в каталоге ${TBB_ROOT}/include, а разделяемые библиотеки – в каталоге ${TBB_ROOT}/lib/${ARCH}/${GCC_LIB_VERSION}, где ARCH – системная архитектура [ia32|intel64|mic], а GCC_LIB_VERSION – версия библиотеки TBB, совместимая с установленным компилятором GCC или clang.

Различие между версиями библиотеки TBB связано с зависимостями от средств библиотек времени выполнения C++ (таких как libstdc++ или libc++).

Обычно для нахождения подходящей версии TBB мы можем использовать команду gcc -version в окне терминала. Затем нужно выбрать ближайшую из доступных версий в каталоге ${TBB_ROOT}/lib/${ARCH}, но не более позднюю, чем установленная версия GCC (обычно это работает даже при использовании компилятора clang++). Но поскольку установка может зависеть от машины, а мы вправе выбирать различные комбинации компилятора и среды выполнения C++, этот простой подход не всегда срабатывает. Если что-то пошло не так, обратитесь к документации по TBB за дополнительными инструкциями.

Например, в системе, где установлен компилятор GCC 5.4.0, мы компилировали пример на рис. 1.4 командой

```
g++ -std=c++11 -o fig_1_04 fig_1_04.cpp    \
   -I ${TBB_ROOT}/include                  \
   -L ${TBB_ROOT}/lib/intel64/gcc4.7 -ltbb
```

И при использовании clang++ использовали ту же самую версию TBB:

```
clang++ -std=c++11 -o fig_1_04 fig_1_04.cpp \
   -I ${TBB_ROOT}/include                   \
   -L ${TBB_ROOT}/lib/intel64/gcc-4.7 -ltbb
```

Для компиляции примера на рис. 1.5 нужно еще указать путь к каталогу заголовочных файлов Parallel STL:

```
g++ -std=c++11 -o fig_1_05 fig_1_05.cpp     \
   -I ${TBB_ROOT}/include                   \
   -I ${PSTL_ROOT}/include                  \
   -L ${TBB_ROOT}/lib/intel64/gcc4.7 -ltbb
```

Вне зависимости от используемого компилятора (Intel, gcc или clang++) необходимо указать путь к разделяемой библиотеке TBB в переменной `LD_LIBRARY_PATH`, чтобы система могла найти библиотеку во время выполнения приложения. Опять же, предполагая, что корневой каталог нашей установки TBB записан в переменную `TBB_ROOT`, мы можем сделать это следующим образом:

```
export LD_LIBRARY_PATH=${TBB_ROOT}/lib/${ARCH}/${GCC_LIB_VERSION}:${LD_LIBRARY_PATH}
```

Откомпилировав приложение компилятором Intel, gcc или clang++ и установив переменную среды `LD_LIBRARY_PATH`, мы можем запустить приложение из командной строки:

```
./fig_1_04
```

В результате должно быть напечатано что-то вроде

```
Hello
Parallel STL!
```

Более полный пример

В предыдущих разделах описаны шаги написания, сборки и выполнения простых приложений TBB и Parallel STL, которые печатают две строки текста. В этом разделе мы напишем программу побольше, в которой воспользуемся всеми тремя высокоуровневыми параллельными интерфейсами, показанными на рис. 1.2. Мы не объясняем все детали алгоритмов и средства, использованные при создании примера, а просто хотим показать в деле различные уровни параллелизма, выразимые с помощью TBB. Пример намеренно искусственный. Он достаточно прост, чтобы объяснения уместились в нескольких абзацах, но довольно сложен, чтобы продемонстрировать описанные выше уровни параллелизма. Окончательный код следует рассматривать как иллюстрацию синтаксиса, а не как руководство по написанию оптимального TBB-приложения. В следующих главах мы подробно рассмотрим все средства, встречающиеся в этом примере, и расскажем, как их использовать для достижения высокой производительности в реальных приложениях.

Начинаем с последовательной реализации

Начнем с последовательной реализации, показанной на рис. 1.7. В этом примере к каждому изображению, хранящемуся в векторе, применяется гамма-коррекция и подкраска, а затем результат записывается в файл. Выделенная функция `fig_1_7` содержит цикл for, в котором обрабатываются элементы вектора путем вызова функций `applyGamma`, `applyTint` и `writeImage` для каждого изображения. На рис. 1.7 показаны также последовательные реализации каждой из этих функций. Определения представления изображений и некоторых вспомогательных функций содержатся в файле `ch01.h`. Этот заголовочный файл, как и весь остальной код примера, имеется по адресу https://github.com/Apress/threading-building-blocks.

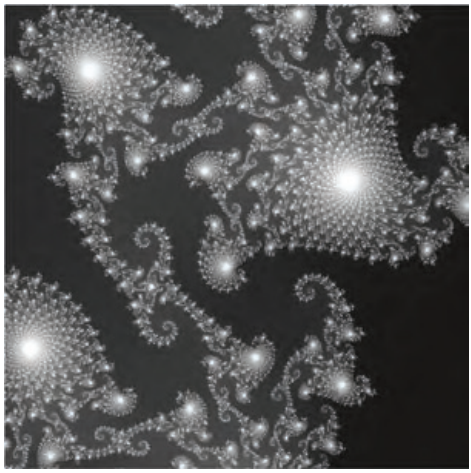
```
#include <iostream>
#include <vector>
#include <tbb/tbb.h>
#include "ch01.h"

using ImagePtr = std::shared_ptr<ch01::Image>;

ImagePtr applyGamma(ImagePtr image_ptr, double gamma);
ImagePtr applyTint(ImagePtr image_ptr, const double *tints);
void writeImage(ImagePtr image_ptr);

void fig_1_7(const std::vector<ImagePtr>& image_vector) {
  const double tint_array[] = {0.75, 0, 0};
  for (ImagePtr img : image_vector) {
    img = applyGamma(img, 1.4);
    img = applyTint(img, tint_array);
    writeImage(img);
  }
}

ImagePtr applyGamma(ImagePtr image_ptr, double gamma) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_gamma",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  for ( int i = 0; i < height; ++i ) {
    for ( int j = 0; j < width; ++j ) {
      const ch01::Image::Pixel& p = in_rows[i][j];
      double v = 0.3*p.bgra[2] + 0.59*p.bgra[1] + 0.11*p.bgra[0];
      double res = pow(v, gamma);
      if(res > ch01::MAX_BGR_VALUE) res = ch01::MAX_BGR_VALUE;
      out_rows[i][j] = ch01::Image::Pixel(res, res, res);
```

Рис. 1.7 ❖ Последовательная реализация примера, в котором к вектору изображений применяется гамма-коррекция и подкраска

```
    }
  }
  return output_image_ptr;
}

ImagePtr applyTint(ImagePtr image_ptr, const double *tints) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_tinted",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  for ( int i = 0; i < height; ++i ) {
    for ( int j = 0; j < width; ++j ) {
      const ch01::Image::Pixel& p = in_rows[i][j];
      std::uint8_t b = (double)p.bgra[0] +
                       (ch01::MAX_BGR_VALUE-p.bgra[0])*tints[0];
      std::uint8_t g = (double)p.bgra[1] +
                       (ch01::MAX_BGR_VALUE-p.bgra[1])*tints[1];
      std::uint8_t r = (double)p.bgra[2] +
                       (ch01::MAX_BGR_VALUE-p.bgra[2])*tints[2];
      out_rows[i][j] =
        ch01::Image::Pixel(
          (b > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : b,
          (g > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : g,
          (r > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : r
      );
    }
  }
  return output_image_ptr;
}

void writeImage(ImagePtr image_ptr) {
  image_ptr->write( (image_ptr->name() + ".bmp").c_str());
}

int main(int argc, char* argv[]) {
  std::vector<ImagePtr> image_vector;

  for ( int i = 2000; i < 20000000; i *= 10 )
    image_vector.push_back(ch01::makeFractalImage(i));

  tbb::tick_count t0 = tbb::tick_count::now();
  fig_1_7(image_vector);
  std::cout << "Time : " << (tbb::tick_count::now()-t0).seconds()
            << " seconds" << std::endl;
  return 0;
}
```

Рис. 1.7 (продолжение)

Функции `applyGamma` и `applyTint` перебирают строки изображения во внешнем цикле for и пиксели одной строки – во внутреннем цикле for. Вычисляются и записываются в выходное изображение новые значения пикселей. Функция `applyGamma` применяет гамма-коррекцию, а функция `applyTint` подкрашивает изображение синим цветом. Обе функции получают и возвращают объекты типа `std::shared_ptr`, чтобы упростить управление памятью. Читатели, незнакомые с типом `std::shared_ptr`, могут обратиться к врезке «Замечание об интеллектуальных указателях». На рис. 1.8 показаны примеры входного и выходного изображений.



(a) Исходное (i == 2 000 000)



(b) После гамма-коррекции



(c) После гамма-коррекции и подкраски

Рис. 1.8 ❖ Примеры изображений:
(a) исходное изображение, сгенерированное функцией `ch01::makeFractalImage(2000000)`;
(b) изображение после гамма-коррекции;
(c) изображение после гамма-коррекции и подкраски

Замечание об интеллектуальных указателях

Одна из самых трудных сторон программирования на C/C++ – динамическое управление памятью. Каждому вызову `new` (`malloc`) должен соответствовать вызов `delete` (`free`), чтобы не было утечек памяти и двойного освобождения. В стандарт C++11 включены интеллектуальные указатели `unique_ptr`, `shared_ptr` и `weak_ptr`, поддерживающие автоматическое управление памятью, безопасное относительно исключений. Например, если память для объекта выделена вызовом функции `make_shared`, то мы получаем интеллектуальный указатель на объект. Когда мы присваиваем этот разделяемый указатель другому разделяемому указателю, библиотека C++ производит подсчет ссылок. Если на объект не ссылается ни один интеллектуальный указатель, то объект автоматически освобождается. В большинстве примеров в этой книге, в т. ч. на рис. 1.7, мы используем не простые, а интеллектуальные указатели. Это позволяет не думать обо всех местах, где надо было бы вставить вызов `free` или `delete`, – мы знаем, что интеллектуальные указатели свою работу выполнят.

Добавление уровня обмена сообщениями с помощью потокового графа

Применив нисходящий подход, мы можем заменить внешний цикл в функции `fig_1_07` на рис. 1.7 потоковым графом TBB, в котором изображения проходят через ряд фильтров, как показано на рис. 1.9. Мы признаем, что это самое неестественное из всех принятых в этом примере решений. В этом случае легко было бы использовать параллельный цикл или объединить вложенные циклы гамма-коррекции и подкраски. Но для демонстрации воспользуемся графом отдельных узлов, чтобы показать, как TBB можно использовать для выражения высокоуровневого параллелизма путем обмена сообщениями. В главе 3 мы изучим интерфейсы потокового графа TBB и познакомимся с более естественными применениями обмена сообщениями.

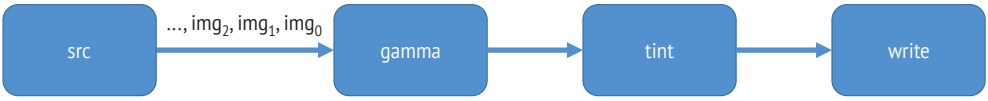


Рис. 1.9 ❖ Граф потока данных с четырьмя узлами:
(1) узел, который получает или генерирует изображения;
(2) узел, который применяет гамма-коррекцию; (3) узел, который применяет подкраску;
(4) узел, который записывает результирующее изображение

С помощью графа потока данных на рис. 1.9 мы можем организовать перекрытие различных этапов конвейера, применяемого к изображениям. Например, когда обработка первого изображения img_0 в узле `gamma` заканчивается, результат передается узлу `tint`, а в узел `gamma` поступает новое изображение img_1. Аналогично после следующего этапа img_0, прошедшее узлы `gamma` и `tint`, передается узлу `write`. Тем временем img_1 передается узлу `tint`, а в узле `gamma` начинается обработка нового изображения img_2. На каждом шаге выполнение фильтров независимо друг друга, поэтому вычисления можно разнести по раз-