Оглавление

Предисловие

От авторов

Боб Роби, Лос-Аламос, Нью-Мексико

> *Выходить за дверь – опасно, Фродо. Выходишь на дорогу, и, если не удержаться на ногах, неизвестно, куда тебя унесет.*
>
> – Бильбо Бэггинс

Я не мог предвидеть, куда нас приведет это путешествие в параллельные вычисления. Я говорю «мы», потому что на протяжении многих лет это путешествие делили со мной многочисленные коллеги. Мое же путешествие в параллельные вычисления началось в начале 1990-х годов, когда я учился в Университете Нью-Мексико. Я написал несколько программ по динамике сжимаемой жидкости для симулирования экспериментов с ударной трубкой и запускал их в каждой системе, которая попадала мне в руки. В результате меня вместе с Брайаном Смитом (Brian Smith), Джоном Соболевски (John Sobolewski) и Фрэнком Гилфезером (Frank Gilfeather) попросили представить предложение по созданию центра высокопроизводительных вычислений. Мы выиграли грант и в 1993 году создали Центр высокопроизводительных вычислений Maui Мое участие в проекте состояло в том, чтобы предлагать курсы и возглавлять подготовку 20 аспирантов по разработке параллельных вычислений в Университете Нью-Мексико в Альбукерке.

1990-е годы были временем становления параллельных вычислений. Я помню выступление Эла Гейста (Al Geist), одного из первых разработчиков параллельной виртуальной машины (PVM) и члена комитета по стандартам MPI[1], как говорил о готовящемся к выпуску стандарте MPI

[1] Message Passing Interface (MPI) – программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. – *Прим. перев.*

(июнь 1994 года). Он сказал тогда, что он никуда не приведет, потому что он слишком сложен. Насчет сложности Эл был прав, но, несмотря на это, он взмыл ввысь, и в течение нескольких месяцев он уже применялся почти во всех параллельных приложениях. Одна из причин успеха MPI заключается в наличии готовых имплементаций. Аргоннская национальная лаборатория разрабатывала Chameleon, инструмент переносимости, который в то время транслировал языки передачи сообщений, включая P4, PVM, MPL и многие другие. Этот проект был быстро изменен на MPICH, став первой высококачественной имплементацией MPI. На протяжении более десяти лет MPI стал синонимом параллельных вычислений. Почти каждое параллельное приложение было построено поверх библиотек MPI.

Теперь давайте перенесемся в 2010 год и увидим появление графических процессоров, GPU. Как-то я наткнулся на статью в журнале Dr. Dobb об использовании суммы Кахана для компенсировпания единственной арифметики с одинарной точностью, доступной на GPU. Я подумал, что, возможно, этот подход мог бы помочь решить давнюю проблему параллельных вычислений, когда глобальная сумма массива меняется в зависимости от числа процессоров. Намереваясь проверить это на практике, я подумал о программном коде по гидродинамике, который мой сын Джон написал в средней школе. Он тестировал сохранение массы и энергии в задаче с течением времени и останавливался, выходя из программы, если масса изменялась более чем на указанную величину. Пока он был дома на весенних каникулах после своего первого курса в Вашингтонском университете, мы опробовали этот метод и были приятно удивлены тем, насколько улучшилось сохранение массы. Для производственных исходных кодов влияние этого простого технического приема окажется важным. Мы рассмотрим алгоритм суммирования с повышенной точностью для параллельных глобальных сумм в разделе 5.7 этой книги.

В 2011 году я организовал летний проект с тремя студентами, Нилом Дэвисом (Neal Davis), Дэвидом Николаеффом (David Nicholaeff) и Деннисом Трухильо (Dennis Trujillo), чтобы попробовать, сможем ли разработать более сложные программные коды, такие как адаптивная детализация расчетной сетки (AMR) и неструктурированные произвольные лагранжево–эйлеровы (ALE) приложения для работы на GPU. Результатом стало CLAMR, мини-приложение AMR, которое полностью работало на GPU. Бóльшая часть приложения была легко переносима. Самой сложной его частью было определение соседа для каждой ячейки. В изначальном CPU-коде использовался алгоритм k-D-дерева, но алгоритмы, основанные на дереве, трудно переносятся на GPU. Через две недели после начала летнего проекта на холмах над Лос-Аламосом вспыхнул пожар в Лас-Кончасе, и город был эвакуирован. Мы уехали в Санта-Фе, и студенты разбежались. Во время эвакуации я встретился с Дэвидом Николаеффом в центре Санта-Фе, чтобы обсудить вопрос переноса на GPU. Он предложил нам попробовать использовать алгоритм хеширования, чтобы заменить древесный исходный код отыскания соседей. В то время я наблюдал за пылающим над городом огнем, и меня не отпускало

беспокойство, что огонь доберется до моего дома. Несмотря на все эти перипетии, я согласился попробовать, и алгоритм хеширования привел к тому, что весь код был запущен на GPU. Техника хеширования была обобщена Дэвидом, моей дочерью Рейчел, когда она училась в средней школе, и мной. Эти алгоритмы хеширования формируют основу для многих алгоритмов, представленных в главе 5.

В последующие годы технические приемы компактного хеширования были разработаны Ребеккой Тамблин (Rebecka Tumblin), Питером Аренсом (Peter Ahrens) и Сарой Харце (Sara Hartse). Более сложная задача компактного хеширования для операций перекомпоновывания (remapping) на CPU и GPU была решена Джеральдом Коломом (Gerald Collom) и Колином Редманом (Colin Redman), когда они только что закончили среднюю школу. Этими прорывами в параллельных алгоритмах для GPU были устранены препятствия для выполнения многих научных приложений на GPU.

В 2016 году я начал программу летней исследовательской стажировки по параллельным вычислениям (PCSRI) в Национальной лаборатории Лос-Аламоса (LANL) вместе с моими соучредителями Хай А Нам (Hai Ah Nam) и Гейбом Рокфеллером (Gabe Rockefeller). Целью программы параллельных вычислений было решение проблемы растущей сложности систем высокопроизводительных вычислений. Программа представляет собой 10-недельную летнюю стажировку с лекциями по различным темам параллельных вычислений с последующим исследовательским проектом под руководством сотрудников Национальной лаборатории Лос-Аламоса. У нас в летней программе участвовало от 12 до 18 студентов, и многие использовали ее в качестве трамплина для своей карьеры. С помощью этой программы мы продолжаем решать некоторые новейшие задачи, стоящие перед параллельными и высокопроизводительными вычислениями.

Джули Замора, Чикагский университет, Иллинойс

> *Если есть книга, которую вы хотите прочитать, но она еще не написана, то вы должны ее написать.*
>
> – Тони Моррисон

Мое знакомство с параллельными вычислениями началось со слов: «Перед тем как начнете, зайдите в комнату в конце 4-го этажа и установите вот эти процессоры Knights Corner в нашем кластере». Эта просьба профессора Корнельского университета побудила меня попробовать что-то новое. То, что я считала простым делом, превратилось в бурное путешествие в высокопроизводительные вычисления. Я начала с изучения основ – с выяснения физического принципа функционирования малого кластера, поднимая 40-фунтовые сервера для работы с BIOS и выполнения моего первого приложения, а затем оптимизации этих приложений по всем установленным мной узлам.

После короткого семейного перерыва, каким бы обескураживающим он ни был, я подала заявление на научно-исследовательскую стажиров-

ку. Будучи принятым в первую программу летней исследовательской стажировки по параллельным вычислениям в Нью-Мексико, я получила возможность разведать тонкости параллельных вычислений на современном оборудовании, и именно там я познакомилась с Бобом. Я пришла в восторг от повышения производительности, которое было возможно при наличии лишь небольших знаний о правильном написании параллельного кода. Я лично провела разведывательный анализ процесса написания более эффективного кода OpenMP. Мое волнение и прогресс в оптимизации приложений открыли двери для других возможностей, таких как участие в конференциях и представление моей работы на собрании группы пользователей Intel и на стенде Intel по суперкомпьютерным вычислениям. В 2017 году меня также пригласили принять участие и выступить на конференции в Салишане. Это была прекрасная возможность обменяться идеями с несколькими ведущими провидцами высокопроизводительных вычислений.

Еще одним замечательным опытом была подача заявки на участие в хакатоне GPU и участие в нем. На хакатоне мы перенесли код на OpenACC, и в течение недели этот код был ускорен в 60 раз. Вы только подумайте – расчет, который раньше занимал месяц, теперь может выполняться за одну ночь. Полностью погрузившись в потенциал долгосрочных исследований, я подала заявление в аспирантуру и выбрала Чикагский университет, осознавая его тесную связь с Аргоннской национальной лабораторией. В Чикагском университете меня консультировали Ян Фостер (Ian Foster) и Генри Хоффман (Henry Hoffmann).

Из своего опыта я поняла, насколько ценным является личное взаимодействие, когда учишься писать параллельный код. Я также была разочарована отсутствием учебника или справочника, в котором обсуждалось бы текущее оборудование. В целях восполнения этого пробела мы написали эту книгу, чтобы сделать ее намного проще для тех, кто только начинает свою деятельность в параллельных и высокопроизводительных вычислениях. Решение задачи создания и преподавания введения в информатику для поступающих студентов Чикагского университета помогло мне получить представление о тех, кто знакомится с этой областью впервые. С другой стороны, объяснение технических приемов параллельного программирования в качестве ассистента преподавателя в курсе «Продвинутые распределенные системы» позволило мне работать со студентами с более высоким уровнем понимания. Оба этих опыта помогли мне обрести способность объяснять сложные темы на разных уровнях.

Я считаю, что у каждого должна быть возможность изучить этот важный материал по написанию высокопроизводительного кода и что он должен быть легко доступен для всех. Мне посчастливилось иметь наставников и советников, которые направляли меня по правильным ссылкам на веб-сайты или передавали мне свои старые рукописи для чтения и изучения. Хотя некоторые технические приемы, возможно, будут сложными, более серьезной проблемой является отсутствие согласованной документации или доступа к ведущим ученым в этой области в качестве наставников. Я понимаю, что не у всех есть одинаковые ре-

сурсы, и поэтому я надеюсь, что создание этой книги заполнит пустоту, которая существует в настоящее время.

Как мы пришли к написанию этой книги

Начиная с 2016 года команда ученых LANL во главе с Бобом Роби разработала лекционные материалы для летней исследовательской стажировки по параллельным вычислениям (PCSRI) в Лос-Аламосской национальной лаборатории (LANL). Большая часть этого материала посвящена новейшему оборудованию, которое быстро выходит на рынок. Параллельные вычисления меняются быстрыми темпами, и к ним прилагается мало документации. Книга, охватывающая эти материалы, была явно необходима. Именно в этот момент издательство Manning связалось с Бобом по поводу написания книги о параллельных вычислениях. У нас был лишь черновой набросок материалов, и работа над ним вылилась в двухлетние усилия по переводу всего этого в формат высокого качества.

Темы и план глав были четко определены на ранней стадии и основывались на лекциях для нашей летней программы. Многие идеи и технические приемы были позаимствованы из более широкого сообщества высокопроизводительных вычислений, поскольку мы стремимся к более высокому уровню вычислений – тысячекратному повышению производительности вычислений по сравнению с предыдущим эталоном петафлопсного масштаба. Это сообщество включает Центры передового опыта Министерства энергетики (DOE), проект по экзомасштабным вычислениям (Exascale Computing Project) и серию семинаров по производительности, переносимости и производительности. Широта и глубина материалов наших компьютерных лекций отражают глубокие проблемы сложных гетерогенных вычислительных архитектур.

Мы называем материал этой книги «глубоким введением». Она начинается с основ параллельных и высокопроизводительных вычислений, но без знания вычислительной архитектуры невозможно достичь оптимальной производительности. По ходу дела мы стараемся осветить проблематику на более глубоком уровне понимания, потому что недостаточно просто двигаться по тропе, не имея ни малейшего представления о том, где вы находитесь или куда направляетесь. Мы предоставляем инструменты для разработки карты и для того, чтобы показать удаленность цели, к которой мы стремимся.

В начале этой книги Джо Шоновер (Joe Schoonover) был привлечен для написания материалов, связанных с GPU, а Джули Замора – главы по OpenMP. Джо предоставил дизайн и компоновку разделов по GPU, но ему пришлось быстро отказаться. Джули написала статьи и предоставила массу иллюстраций о том, как OpenMP вписывается в этот дивный новый мир масштабных вычислений, поэтому указанный материал особенно хорошо подошел для главы книги об OpenMP. Глубокое понимание Джули проблем, связанных с экзомасштабными вычислениями, и ее способность разбирать их по полочкам для новичков в этой области стали решающим вкладом в создание этой книги.

О книге

Одна из самых важных задач для исследователя-первопроходца состоит в том, чтобы начертить карту для тех, кто последует за ним. Это особенно верно для тех из нас, кто раздвигает границы науки и техники. Нашей целью в этой книге было предложение дорожной карты для тех, кто только начинает изучать параллельные и высокопроизводительные вычисления, а также для тех, кто хочет расширить свои знания в этой области. Высокопроизводительные вычисления – это быстро эволюционирующая область, где языки и технологии находятся в постоянном потоке. По этой причине мы сосредоточимся на фундаментальных принципах, которые остаются неизменными с течением времени. В компьютерных языках для CPU и GPU мы подчеркиваем общие закономерности во многих языках, чтобы дать вам возможность быстро выбирать наиболее подходящий язык для вашей текущей задачи.

Кто должен прочитать эту книгу

Эта книга предназначена как для студентов продвинутых курсов по параллельным вычислениям, так и в качестве современной литературы для специалистов в области вычислительной техники. Если вас интересует производительность, будь то время выполнения, масштаб или мощность, то эта книга предоставит вам инструменты для усовершенствования вашего приложения и повышения вашей конкурентноспособности. Имея процессоры, которые достигают пределов масштаба, тепла и мощности, мы не можем рассчитывать на компьютер следующего поколения в деле ускорения наших приложений. Все чаще высококвалифицированные и знающие программисты имеют решающее значение для достижения максимальной производительности современных приложений.

В этой книге мы надеемся изложить ключевые идеи, применимые к современному оборудованию высокопроизводительных вычислений. Они являются базовыми истинами программирования для повышения производительности. Указанные темы лежат в основе всей книги.

В высокопроизводительных вычислениях важно не то, как быстро вы пишете код, а то, как быстро выполняется написанный вами код.

Эта мысль подводит итог тому, что значит писать приложения для высокопроизводительных вычислений. В большинстве других приложений основное внимание уделяется быстроте написания приложения. Сегодня компьютерные языки, как правило, предназначены для ускорения программирования, а не для повышения производительности кода. Хотя этот подход к программированию уже давно используется в приложениях высокопроизводительных вычислений, он не был широко задокументирован или описан. В главе 4 мы обсудим этот другой фокус внимания в методологии программирования, который недавно получил свой термин «дизайн с ориентацией на данные».

Все дело в памяти: сколько вы ее используете и как часто загружаете.

Даже если вы знаете, что доступная память и операции с памятью почти всегда являются лимитирующим фактором производительности, мы все равно склонны тратить много времени на размышления об операциях с плавающей точкой. При наличии способности большинства современных вычислительных устройств выполнять 50 операций с плавающей точкой для каждой загрузки в память операции с плавающей точкой являются второстепенным вопросом. Почти в каждой главе мы используем нашу имплементацию потокового сравнительного теста (STREAM benchmark), теста производительности памяти с целью верификации получения нами разумной производительности от оборудования и языка программирования.

Если вы загружаете одно значение, то получаете восемь или шестнадцать.

Это все равно что покупать яйца. Невозможно взять только одно. Загрузка в память выполняется строками кеша по 512 бит. Для 8-байтового значения двойной точности будет загружено восемь значений, хотите вы этого или нет. В целях достижения наилучшей производительности следует планировать так, чтобы ваша программа использовала более одного значения, предпочтительно восемь смежных значений. И пока вы этим занимаетесь, она использовала остальные яйца.

Если в вашем коде есть какие-либо изъяны, то параллелизация их проявит.

Качество кода требует большего внимания при высокопроизводительных вычислениях, чем в сопоставимом последовательном приложении. Это относится ко всем этапам: до начала параллелизации, во время и после. При параллелизации вы с большей вероятностью запустите в своей программе изъян, а также обнаружите, что проводить отлаживание будет сложно, в особенности в крупном масштабе. Мы представляем

технические приемы повышения качества программного обеспечения в главе 2, затем на протяжении всех глав упоминаем важные инструменты, и, наконец, в главе 17 мы перечисляем другие инструменты, которые могут оказаться полезными.

Эти ключевые темы выходят за рамки типов оборудования, одинаково применимых ко всем CPU и GPU. Они существуют из-за текущих физических ограничений, накладываемых на аппаратное обеспечение.

Как эта книга организована: дорожная карта

Эта книга не предполагает, что вы обладаете какими-либо знаниями в области параллельного программирования. Ожидается, что читатели являются опытными программистами, предпочтительно на компилируемом языке высокопроизводительных вычислений, таком как C, C++ или Fortran. Также ожидается, что читатели будут обладать некоторыми знаниями в области компьютерной терминологии, основ операционных систем и сетей. Читатели также должны иметь возможность ориентироваться на своем компьютере, включая инсталлирование программного обеспечения и легкие задачи системного администрирования.

Знание компьютерного оборудования, пожалуй, является самым важным требованием к читателям. Мы рекомендуем открыть корпус вашего компьютера, изучить каждый компонент и получить представление о его физических характеристиках. Если вы не можете открыть корпус своего компьютера, то рекомендуем посмотреть фотографии типичной настольной системы в конце приложения C к книге. Например, посмотрите на нижнюю часть процессора на рис. C.2 и на лес контактных штырьков, входящих в чип. Вам не удастся вставить туда даже булавку. Теперь вы можете понять, почему существует физический предел того, сколько данных может быть передано процессору из других частей системы. Рекомендуем возвращаться к этим фотографиям и глоссарию в приложении A, если у вас возникает необходимость лучше понять компьютерное оборудование или компьютерную терминологию.

Мы разделили эту книгу на четыре части, которые охватывают мир высокопроизводительных вычислений. Указанные части таковы:

- часть I «Введение в параллельные вычисления» (главы 1–5);
- часть II «Технологии центрального процессора (CPU)» (главы 6–8);
- часть III «Технологии графического процессора (GPU)» (главы 9–13);
- часть IV «Экосистемы высокопроизводительных вычислений (HPC)» (главы 14–17).

Порядок тем ориентирован на тех, кто занимается проектом, связанным с высокопроизводительными вычислениями. Например, для прикладного проекта темы разработки программного обеспечения в главе 2 необходимы перед запуском проекта. По завершении разработки программного обеспечения следующими техническими решениями станут структуры данных и алгоритмы. Затем идут имплементации для CPU и GPU. Наконец, приложение адаптируется для параллельной файловой

системы и других уникальных характеристик системы высокопроизводительных вычислений.

С другой стороны, некоторые из наших читателей больше заинтересованы в приобретении фундаментальных навыков параллельного программирования и, возможно, захотят перейти непосредственно к главам MPI или OpenMP. Но не стоит останавливаться на достигнутом. Сегодня существует гораздо больше возможностей для параллельных вычислений. От GPU, которые могут ускорять ваше приложение на порядок, до инструментов, которые могут улучшать качество вашего кода или указывать разделы кода, которые подлежат оптимизации, – потенциальные выгоды ограничены только вашим временем и опытом.

Если вы используете эту книгу для занятий по параллельным вычислениям, объема материала хватит как минимум на два семестра. Вы можете рассматривать эту книгу как коллекцию материалов, которые могут быть подобраны индивидуально под аудиторию. Выбрав темы для изучения, вы можете настроить его для своих собственных курсовых целей. Вот возможная последовательность изложения материала:

- глава 1 содержит введение в параллельные вычисления;
- в главе 3 рассматриваются подходы к измерению производительности оборудования и приложений;
- разделы 4.1-4.2 описывают концепцию программирования, ориентированного на данные, многомерные массивы и основы кеширования;
- глава 7 посвящена OpenMP (открытой мультипроцессорной обработке) для обеспечения параллелизма на узлах;
- в главе 8 рассматривается MPI (Интерфейс передачи сообщений) для обеспечения распределенного параллелизма между несколькими узлами;
- в разделах 14.1–14.5 представлены концепции аффинности и мест размещения процессов;
- главы 9 и 10 описывают оборудование GPU и модели программирования;
- разделы 11.1-11.2 посвящены OpenACC для обеспечения работы приложений на GPU.

Вы можете добавить в этот список такие темы, как алгоритмы, векторизация, параллельная обработка файлов или несколько других языков GPU. Или же удалить какую-то тему, чтобы потратить больше времени на остальные темы. Еще есть дополнительные главы, которые побудят студентов продолжить самостоятельное разведывание мира параллельных вычислений.

Об исходном коде

Невозможно научиться параллельным вычислениям, не написав код и не выполнив его. Для этой цели мы приводим большой набор примеров, прилагаемых к этой книге. Примеры находятся в свободном доступе

по адресу https://github.com/EssentialsOfParallelComputing. Вы можете скачать эти примеры в виде полного комплекта либо по отдельности по главам.

Также весь используемый в этой книге исходный код для книг издательства «ДМК Пресс» можно найти на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Учитывая объем примеров, аппаратное и программное обеспечение, в прилагаемых примерах неизбежно будут присутствовать изъяны и ошибки. Если вы обнаружите что-то, что является ошибкой, или просто пример будет неполным, то рекомендуем внести свой вклад в эти примеры. Мы уже объединили несколько ценных запросов на внесение изменений, поступивших от читателей. Кроме того, хранилище исходного кода будет наилучшим местом для поиска исправлений и обсуждений исходного кода.

Требования к программному/аппаратному обеспечению

Возможно, самой большой проблемой параллельных и высокопроизводительных вычислений является широкий спектр используемого аппаратного и программного обеспечения. В прошлом эти специализированные системы были доступны только на специфичных веб-сайтах. В последнее время аппаратное и программное обеспечение стало более демократичным и широко доступным даже на уровне настольных компьютеров или ноутбуков. Произошел существенный сдвиг, который значительно упрощает разработку программного обеспечения для высокопроизводительных вычислений. Однако настройка аппаратной и программной среды является самой сложной частью задачи. Если у вас есть доступ к параллельно-вычислительному кластеру, где они уже настроены, то мы рекомендуем вам воспользоваться этим преимуществом. В конце концов вы, возможно, захотите настроить свою собственную систему. Примеры проще всего использовать в системах Linux или Unix, но во многих случаях они должны работать и в Windows, и macOS при некоторых дополнительных усилиях. На случай если вы обнаружите, что пример не запускается в вашей системе, мы предоставили альтернативы заготовкам контейнеров Docker и настроечным скриптам VirtualBox.

Для выполнения упражнений с GPU требуются GPU разных производителей, включая NVIDIA, AMD Radeon и Intel. Любой, кто пытался установить графические драйверы GPU в своей системе, не удивится, что они представляют наибольшую трудность в настройке вашей локальной системы для примеров. Некоторые языки GPU также могут работать на CPU, позволяя разрабатывать код в локальной системе для оборудования, которого у вас нет. Вы также, возможно, обнаружите, что отладка на CPU проходит проще. Но в целях ознакомления с реальной производительностью вам потребуется реальное оборудование GPU.

Другие примеры, требующие специальной инсталляции, включают примеры пакетной системы и параллельных файлов. Для пакетной системы требуется больше, чем один ноутбук или рабочая станция, чтобы

инсталляция выглядела как настоящая. Аналогичным образом примеры параллельных файлов лучше всего работают со специализированной файловой системой, такой как Lustre, хотя базовые примеры будут работать на ноутбуке или рабочей станции.

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 на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Дискуссионный форум liveBook

Покупка книги «Параллельные и высокопризводительные вычисления» включает в себя бесплатный доступ к приватному веб-форуму издательства Manning Publications, где вы можете комментировать книгу, зада-

вать технические вопросы и получать помощь от авторов и других пользователей. В целях получения доступа к указанному форуму перейдите по ссылке https://livebook.manning.com/#!/book/parallel-and-high-performance-computing/discussion. О форумах издательства Manning и правилах поведения можно узнать больше на веб-странице https://livebook.manning.com/#!/discussion.

Другие онлайновые ресурсы

Публикации Manning также предоставляют онлайновый дискуссионный форум под названием livebook для каждой книги. Наш веб-сайт находится по адресу https://livebook.manning.com/book/parallel-and-high-performance-computing. Это самое подходящее место для добавления комментариев или расширения материалов в главах.

Об авторах

Роберт (Боб) Роби работает техническим сотрудником отдела вычислительной физики Национальной лаборатории Лос-Аламоса и является адъюнкт-исследователем Университета Нью-Мексико. Он является основателем летней исследовательской стажировки по параллельным вычислениям, которая началась в 2016 году. Он участвует в учебной инициативе NSF/IEEE-TCPP по параллельным и распределенным вычислениям. Боб также является членом правления New Mexico Supercomputing Challenge, образовательной программы для средней и старшей школы, которой уже 30 лет. На протяжении многих лет он был наставником для сотен студентов и дважды был признан выдающимся наставником студентов Лос-Аламоса. Боб был соучредителем курса по параллельным вычислениям в Университете Нью-Мексико и читал гостевые лекции в других университетах.

Боб начал свою научную карьеру с эксплуатации взрывоопасных и сжимаемых газовых ударных труб в Университете Нью-Мексико. Его работа была связана с самой большой в мире ударной трубой со взрывным приводом диаметром 20 футов и длиной более 800 футов. Он провел сотни экспериментов со взрывами и ударными волнами. В целях поддержания своей экспериментальной работы Боб написал несколько программ по гидродинамике сжимаемой жидкости с начала 1990-х годов и является автором многих статей в международных журналах и публикациях. Полное 3D-моделирование в то время было редкостью и требовало предельного напряжения вычислительных ресурсов. Поиск дополнительных вычислительных ресурсов привел его к участию в исследованиях в сфере высокопроизводительных вычислений.

Боб проработал 12 лет в Университете Нью-Мексико, проводя эксперименты, записывая и проводя симулирование гидродинамики сжимаемой жидкости, и основал центр высокопроизводительных вычислений. Он был ведущим автором предложений и принес университету десятки миллионов долларов исследовательских грантов. С 1998 года он занимает должность в Национальной лаборатории Лос-Аламоса. Находясь там, он внес важный вклад в создание больших мультифизических исходных кодов, работающих на самом разном новейшем оборудовании.

Боб является байдарочником мирового класса, впервые спустившимся по ранее неизведанным рекам Мексики и Нью-Мексико. Он также увлекается альпинизмом и имеет за своими плечами восхождения на вершины трех континентов на высоту более 18 000 футов. Он является лидером команды студентов-однокурсников Лос-Аламоса и помогает в многодневных поездках по западным рекам.

Боб окончил Техасский университет A&M со степенью магистра делового администрирования и степенью бакалавра в области машиностроения. Он прошел аспирантуру в Университете Нью-Мексико на математическом факультете.

Джулиана (Джули) Замора заканчивает докторскую степень по информатике в Чикагском университете. Джули является стипендиатом 2017 года в Центре неостанавливаемых вычислений ЦЕРЕРЫ при Чикагском университете и аспирантом Национального консорциума физических наук (NPSC).

Джули работала в Лос-Аламосской национальной лаборатории и стажировалась в Аргоннской национальной лаборатории. В Лос-Аламосской национальной лаборатории она занималась оптимизированием исходного кода Higrad Firetec, используемого для симулирования лесных пожаров и другой физики атмосферы для некоторых из самых лучших систем высокопроизводительных вычислений. В Аргоннской национальной лаборатории она работала на стыке высокопроизводительных вычислений и машинного обучения. Она работала над проектами, начиная от предсказания производительности GPU-процессоров NVIDIA и заканчивая суррогатными моделями машинного обучения для научных приложений.

Джули разработала и преподавала курс «Введение в информатику» для поступающих студентов Чикагского университета. Она включила в свой учебный материал многие главные концепции основ параллельных вычислений. Ее курс был настолько успешным, что ее просили преподавать его снова и снова. Желая получить больше педагогического опыта, она добровольно вызвалась на должность ассистента преподавателя на курсе «Продвинутые распределенные системы» в Чикагском университете.

Степень бакалавра Джули получила в области гражданского строительства в Корнельском университете. Она получила степень магистра компьютерных наук в Чикагском университете и вскоре защитит докторскую диссертацию по информатике, также в Чикагском университете.

Часть I

Введение в параллельные вычисления

Первая часть этой книги охватывает темы, имеющие общую значимость для параллельных вычислений. Эти темы включают:

- понимание ресурсов параллельного компьютера;
- оценивание производительности и ускорения приложений;
- обзор потребностей, специфичных для параллельных вычислений, со стороны разработчиков программного обеспечения;
- рассмотрение вариантов структур данных;
- отбор алгоритмов, которые показывают хорошую производительность и хорошо параллелизуются.

Хотя эти темы должны рассматриваться параллельным программистом в первую очередь, они не будут иметь одинаковой значимости для всех читателей этой книги. Для разработчика параллельных приложений все главы этой части посвящены первоочередным задачам успешного проекта. Проект нуждается в отборе правильного оборудования, правильного типа параллелизма и правильных ожиданий. Прежде чем приступать к параллелизации, следует определить соответствующие структуры данных и алгоритмы; позже их будет гораздо сложнее изменить.

Даже если вы являетесь разработчиком параллельных приложений, вам, возможно, не понадобится вся глубина обсуждаемого материала. Те, кто желает лишь скромного параллелизма или выполняет ту или иную роль в коллективе разработчиков, могут счесть достаточным беглое понимание содержимого. Если вы просто хотите разведать тему параллель-

ных вычислений, то мы предлагаем прочитать главу 1 и главу 5, а затем просмотреть остальные, чтобы освоиться с терминологией, используемой при обсуждении параллельных вычислений.

Мы включаем главу 2 для тех, у кого, возможно, не будет опыта разработки программного обеспечения, или для тех, кто просто нуждается в освежении своих знаний. Если вы новичок во всех деталях оборудования CPU, то вам, возможно, потребуется читать главу 3 малыми порциями. В целях повышения производительности важно понимать современное компьютерное оборудование и ваше приложение, но это не обязательно должно происходить сразу. Обязательно вернитесь к главе 3, когда будете готовы приобрести свою следующую компьютерную систему, чтобы иметь возможность разобраться во всех маркетинговых заявлениях в отношении того, что действительно имеет важность для вашего приложения.

Обсуждение тем дизайна данных и моделирования производительности в главе 4 может оказаться сложным для восприятия, поскольку для того, чтобы оценить его по достоинству, требуется понимание деталей оборудования, его производительности и компиляторов. Хотя эта тема важна по причине последствий оптимизации кеша и компилятора на производительность, она не обязательна для написания простой параллельной программы.

Мы рекомендуем вам сверяться с прилагаемыми к книге примерами. Рекомендуем вам также потратить немного времени на разведку многочисленных примеров из репозиториев программного обеспечения, расположенных по адресу https://github.com/EssentialsOfParallelComputing.

Примеры организованы по главам и содержат подробную информацию о настройке различного оборудования и операционных систем. Для оказания помощи в решении вопросов переносимости есть примеры контейнерных сборок для дистрибутивов Ubuntu в Docker. Существуют также инструкции по настройке виртуальной машины с помощью VirtualBox. Если у вас есть необходимость в настройке собственной системы, то вы можете прочитать раздел о Docker и виртуальных машинах в главе 13. Но контейнеры и виртуальные машины поставляются с ограниченными средами, которые нелегко обходить.

Наша работа в отношении контейнерных сборок и других настроек системной среды в целях организации надлежащей работы для многих возможных системных конфигураций не прекращается. Правильное инсталлирование системного программного обеспечения, в особенности драйвера GPU и связанного с ним программного обеспечения, является самой сложной частью путешествия. Большое разнообразие операционных систем, оборудования, включая графические процессоры (GPU), и часто упускаемое из виду качество инсталляционного программного обеспечения делают эту работу сложной. Альтернативой является использование кластера, в котором программное обеспечение уже установлено. Тем не менее в какой-то момент полезно инсталлировать некоторое программное обеспечение на свой ноутбук или настольный компьютер с целью обеспечения более удобного ресурса разработки. Теперь самое время перевернуть страницу и войти в мир параллельных вычислений. Это мир почти неограниченной производительности и потенциала.

1 Зачем нужны параллельные вычисления?

Эта глава охватывает следующие ниже темы:

- что такое параллельные вычисления и почему они приобретают все бóльшую значимость;
- где в современном оборудовании существует параллелизм;
- почему важен параллелизм приложений;
- программные подходы для задействования параллелизма.

В современном мире вы столкнетесь со многими задачами, требующими широкого и эффективного использования вычислительных ресурсов. Большинство приложений, требующих производительности, традиционно относится к научной области. Однако, согласно прогнозам, приложения искусственного интеллекта (ИИ) и машинного обучения станут преобладающими пользователями крупномасштабных вычислений. Несколько примеров таких приложений включают:

- моделирование мегапожаров для оказания помощи пожарным командам и населению;
- моделирование цунами и штормовых волн от ураганов (см. главу 13, в которой приводится простая модель цунами);
- распознавание голоса для компьютерных интерфейсов;
- моделирование распространения вируса и разработка вакцин;
- моделирование климатических условий на десятилетия и столетия;

- распознавание изображений для технологии беспилотных автомобилей;
- оснащение аварийных бригад работающими симуляциями таких источников опасности, как наводнение;
- снижение энергопотребления мобильных устройств.

С помощью описанной в этой книге технологии вы сможете справляться с более крупными задачами и наборами данных, одновременно выполняя симуляции в десять, сто или даже в тысячу раз быстрее. Типичные приложения оставляют большую часть вычислительных способностей современных компьютеров незадействованными. Параллельные вычисления представляют собой ключ к выявлению потенциала ваших компьютерных ресурсов. Итак, что такое параллельные вычисления и как их можно использовать, чтобы подстегнуть свои приложения?

Параллельные вычисления – это исполнение большого числа операций в отдельный момент времени. Полное эксплуатирование параллельных вычислений не происходит автоматически. Это требует некоторых усилий от программиста. И прежде всего вы должны определить и выявить потенциал параллелизма в приложении. Потенциальный параллелизм, или конкурентность, означает, что вы подтверждаете безопасность выполнения операций в любом порядке по мере появления системных ресурсов. А при параллельных вычислениях существует еще одно, дополнительное требование: эти операции должны выполняться в одно и то же время. Для того чтобы это происходило, вы также должны правильно задействовать ресурсы с целью одновременного исполнения этих операций.

Параллельные вычисления вводят новые трудности, которых нет в последовательном мире. Для того чтобы приспособиться к дополнительным сложностям параллельного исполнения, нам необходимо поменять наши мыслительные процессы, но с практикой это будет становиться второй натурой. Данная книга начинает ваше открытие в том, как получать доступ к мощи параллельных вычислений.

Жизнь представляет массу примеров параллельной обработки, и эти примеры часто становятся основой для вычислительных стратегий. На рис. 1.1 показана кассовая линия супермаркета, где цель состоит в том, чтобы клиенты быстро оплачивали товары, которые они хотят приобрести. Это можно сделать, наняв нескольких кассиров для обработки или проверки клиентов по одному за раз. В этом случае квалифицированные кассиры смогут быстрее исполнять процесс оплаты покупок, в результате чего клиенты будут покидать магазин быстрее. Еще одна стратегия состоит в задействовании большого числа касс самообслуживания и разрешении клиентам исполнять процесс самостоятельно. Эта стратегия требует меньше человеческих ресурсов от супермаркета и позволяет открыть больше линий по обработке клиентов. Клиенты, возможно, не смогут оплачивать покупки так же эффективно, как обученный кассир, но, возможно, больше клиентов сможет быстро выполнять оплату из-за увеличения параллелизма, что приведет к сокращению очередей.

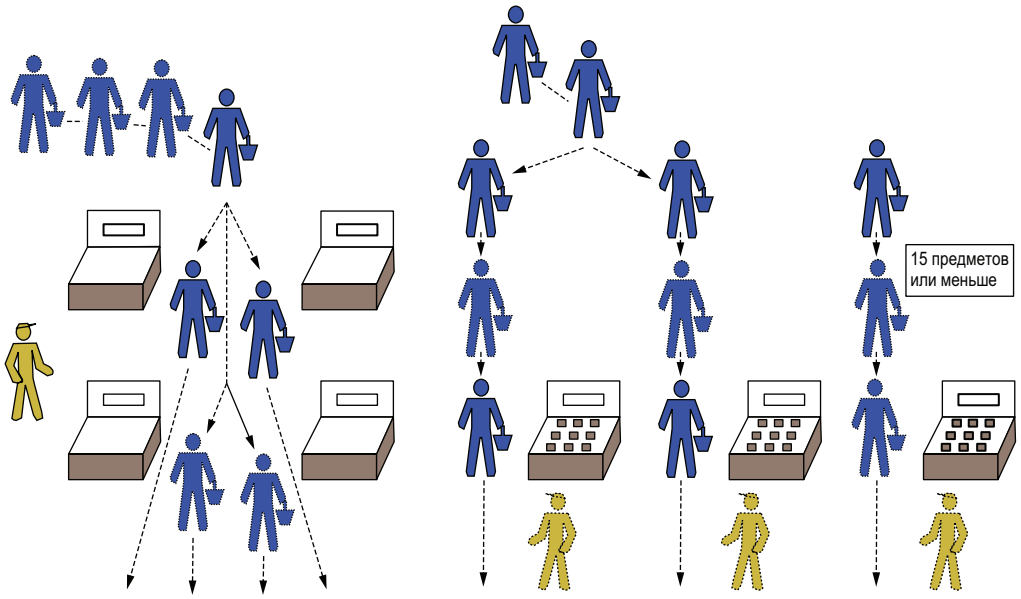


Рис. 1.1 Повседневный параллелизм в очередях в кассы супермаркетов. Кассиры (с кепками) обрабатывают свою очередь клиентов (с корзинами). Слева один кассир одновременно обрабатывает четыре полосы для каждой самостоятельной оплаты покупки. Справа один кассир требуется для каждой полосы оплаты покупки. Каждый вариант влияет на затраты супермаркета и темпы оплаты покупок

Мы решаем вычислительные задачи, разрабатывая *алгоритмы*, перечень шагов для достижения желаемого результата. В аналогии с супермаркетом процесс самообслуживания при оплата покупки представляет собой алгоритм. В данном случае сюда входит выгрузка товаров из корзины, сканирование товаров для получения цены и оплата товаров. Указанный алгоритм является последовательным (или сериальным); он должен следовать этому порядку. Если данная задача должна исполняться сотнями клиентов, то алгоритм оформления покупок многочисленных клиентов содержит параллелизм, которым можно воспользоваться. Теоретически нет никакой зависимости между любыми двумя клиентами, которые проходят процесс оплаты покупки. Используя несколько кассовых линий или станций самообслуживания, супермаркеты демонстрируют параллелизм, тем самым увеличивая скорость, с которой покупатели покупают товары и покидают магазин. Каждый вариант выбора в том, как мы имплементируем этот параллелизм, приводит к разным затратам и выгодам.

ОПРЕДЕЛЕНИЕ *Параллельные вычисления* – это практика выявления и раскрытия параллелизма в алгоритмах с выражением их в нашем программном обеспечении и пониманием затрат, преимуществ и ограничений выбранной имплементации.

В конечном итоге параллельные вычисления всецело касаются производительности. Сюда входит не только скорость, но и масштаб задачи и энергоэффективность. Наша цель в этой книге состоит в том, чтобы дать вам понимание широты современной области параллельных вычислений и познакомить с достаточным числом наиболее часто используемых языков, технических приемов и инструментов, чтобы обеспечить возможность уверенно браться за параллельно-вычислительный проект. Важные решения о том, как встраивать параллелизм, часто принимаются в начале проекта. Продуманный дизайн является важным шагом на пути к успеху. Уклонение от шага дизайна может привести к проблемам гораздо позже. Не менее важно сохранять реалистичность ожиданий и знать имеющиеся ресурсы и характер проекта.

Еще одна цель этой главы состоит в ознакомлении вас с терминологией, используемой в параллельных вычислениях. Время от времени это будет делаться, направляя вас к глоссарию в приложении C с целью краткого ознакомления с терминологией, когда вы будете читать эту книгу. Поскольку эта область и технология постепенно расширяются, использование многих терминов участниками параллельного сообщества зачастую бывает небрежным и неточным. В связи с возросшей сложностью оборудования и параллелизма в приложениях важно с самого начала применять терминологию четко и недвусмысленно.

Добро пожаловать в мир параллельных вычислений! По мере того как вы будете углубляться, технические приемы и подходы будут становиться более естественными, и вы обнаружите, что их сила завораживает. Задачи, к решению которых вы никогда не думали приступать, станут обыденным делом.

1.1 Почему вы должны изучить параллельные вычисления?

Будущее будет параллельным. Увеличение последовательной производительности достигло плато, поскольку варианты дизайна процессоров достигли пределов миниатюризации, тактовой частоты, мощности и даже тепла. На рис. 1.2 показаны тренды тактовой частоты (скорости, с которой команда может исполняться), энергопотребления, числа вычислительных ядер (или просто ядер для краткости) и производительности оборудования с течением времени для товарных процессоров.

В 2005 году число ядер резко возросло с одного до нескольких. В то же время тактовая частота и энергопотребление выровнялись. Теоретическая производительность неуклонно повышалась, поскольку производительность пропорциональна произведению тактовой частоты и числа ядер. Этот сдвиг в сторону увеличения числа ядер, а не тактовой частоты указывает на то, что достигуть наиболее идеальную производительность центрального процессора (CPU) можно только за счет параллельных вычислений.

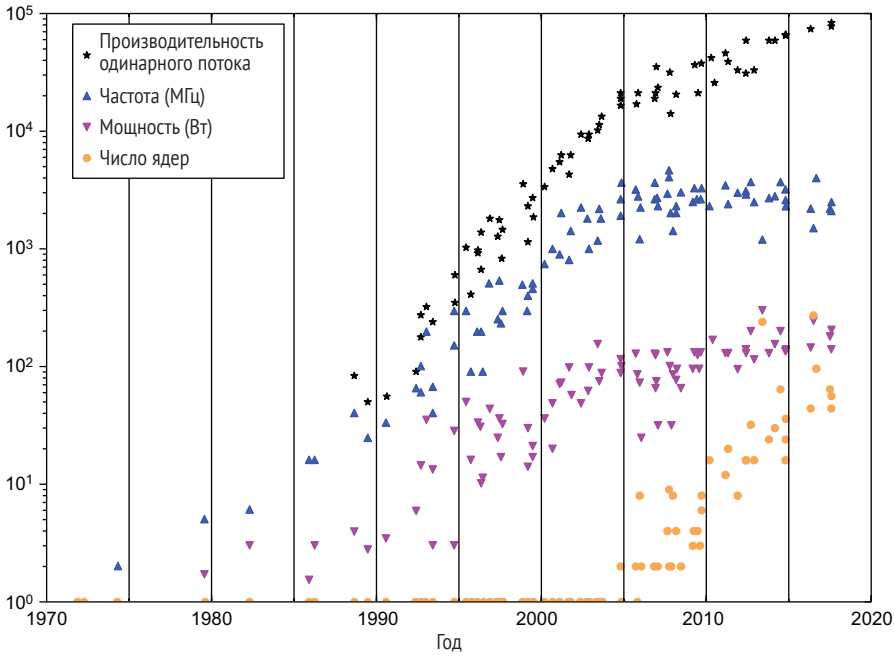


Рис. 1.2 Производительность одинарного потока, тактовая частота процессора (МГц), потребляемая мощность процессора (Вт) и число ядер процессора с 1970 по 2018 год. Эра параллельных вычислений начинается примерно в 2005 году, когда число ядер в процессорных микросхемах начинает расти, в то время как тактовая частота и энергопотребление снижаются, но производительность неуклонно растет (Горовиц и соавт. и Рупп, https://github.com/karlrupp/microprocessor-trend-data**)**

Современное вычислительное оборудование потребительского класса оснащается несколькими центральными процессорами (CPU) и/или графическими процессорами (GPU), которые обрабатывают несколько наборов команд одновременно. Эти небольшие системы часто соперничают по вычислительной мощности с суперкомпьютерами двадцатилетней давности. В целях полного использования вычислительных ресурсов (на ноутбуках, рабочих станциях, смартфонах и т. д.) требуется, чтобы вы, программист, обладали практическими знаниями об инструментах, служащих для написания параллельных приложений. Вы также должны понимать аппаратные функциональности, которые подстегивают параллелизм.

Поскольку существует много разных параллельных аппаратных функциональностей, это создает программисту новые сложности. Одной из таких функциональностей является гиперпотокообразование (hyper-threading)[1], представленное компанией Intel. Наличие двух очередей

[1] Гиперпотокообразование (Hyper-Threading) – это термин, пришедший из компании Intel, для обозначения одновременного множественного потокообразования (simultaneous multithreading, аббр. SMT). Это процесс, в котором центральный процессор (CPU) разбивает каждое свое физическое ядро на

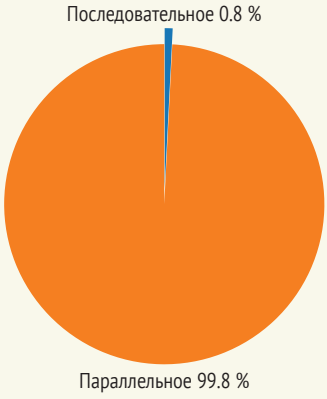
команд, чередующих работу с аппаратными логическими модулями, позволяет одному физическому ядру выглядеть в операционной системе (OS) как два ядра. Векторные процессоры являются еще одной аппаратной функциональностью, которая начала появляться в товарных процессорах примерно в 2000 году. Они исполняют несколько инструкций сразу. Ширина в битах векторного процессора (также именуемого модулем векторной обработки) определяет число команд, которые должны исполняться одновременно. Отсюда модуль векторной обработки шириной 256 бит может исполнять одновременно четыре 64-битовые команды (двойной точности) или восемь 32-битовых команд (одинарной точности).

Пример

Давайте возьмем 16-ядерный CPU с гиперпотокообразованием и векторным модулем шириной 256 бит, который обычно используется в домашних настольных компьютерах. Последовательная программа, использующая одинарное ядро и не имеющая векторизации, использует только 0.8 % теоретических обрабатывающих способностей этого процессора! Расчет выглядит так:

16 ядер × 2 гиперпотока × (векторный модуль шириной 256 бит) / (64-бит двойной точности) = 128-путный параллелизм,

где 1 последовательный путь / 128 параллельных путей = 0.008, или 0.8 %. На следующем ниже рисунке показано, что это составляет мизерную долю суммарной вычислительной мощности процессора.



Последовательное приложение обращается только к 0.8 % вычислительной мощности 16-ядерного процессора

Умение проводить расчет теоретических и реалистичных ожиданий последовательной и параллельной производительности, как показано в этом примере, имеет большую важность. Мы обсудим это подробнее в главе 3.

виртуальные ядра, именуемые потоками (threads). Например, в большинстве процессоров Intel с двумя ядрами гиперпотокообразование используется для обеспечения четырех потоков. – *Прим. перев.*

Несколько усовершенствований в инструментах разработки программного обеспечения помогло добавить параллелизм в наши инструментарии, и в настоящее время научно-исследовательское сообщество делает больше, но до устранения разрыва в производительности еще далеко. Это ложится большой нагрузкой на нас, разработчиков программного обеспечения, чтобы получать максимальную отдачу от процессоров нового поколения.

К сожалению, разработчики программного обеспечения отстали в адаптации к этому фундаментальному изменению вычислительной мощности. Кроме того, транзит существующих приложений в сторону использования современных параллельных архитектур может оказаться обескураживающе сложным из-за бурного развития новых языков программирования и интерфейсов прикладного программирования (API). Но хорошая работающая осведомленность о вашем приложении, способность видеть и выявлять параллелизм, а также глубокое понимание имеющихся инструментов обязательно принесут существенные преимущества. Какие именно преимущества увидят приложения? Давайте посмотрим поближе.

1.1.1 Каковы потенциальные преимущества параллельных вычислений?

Параллельные вычисления могут сокращать время до решения[1], повышать энергоэффективность вашего приложения и позволять решать более масштабные задачи на существующем в настоящее время оборудовании. Сегодня параллельные вычисления больше не являются единственной областью деятельности крупнейших вычислительных систем. Указанная технология теперь присутствует на всех настольных компьютерах или ноутбуках и даже на портативных устройствах. Она позволяет каждому разработчику программного обеспечения создавать параллельное программное обеспечение в своих локальных системах, тем самым значительно расширяя возможности для новых приложений.

Передовые исследования как в промышленности, так и в научных кругах открывают новые области для параллельных вычислений по мере того, как интерес расширяется от научных вычислений до машинного обучения, больших данных, компьютерной графики и потребительских приложений. Появление новых технологий, таких как самоуправляемые автомобили, компьютерное зрение, распознавание голоса и искусственный интеллект, требует крупных вычислительных способностей как внутри потребительских устройств, так и в сфере разработки, где необходимо использовать и обрабатывать массивные тренировочные наборы данных. И в научных вычислениях, которые долгое время были исключительной областью параллельных вычислений, тоже появляются новые, захваты-

[1] Время до решения (time to solution) – это продолжительность времении между предобработкой и постобработкой данных при решении поставленной задачи. – *Прим. перев.*

вающие возможности. Распространение удаленных датчиков и портативных устройств, которые могут подавать данные в более масштабные и реалистичные вычисления для более оптимального информирования при принятии решений, имеющих отношение к природным и техногенным катастрофам, позволяет получать более обширные данные.

Следует помнить, что параллельные вычисления как таковые не являются самоцелью. Правильне сказать, цели – это то, что является результатом параллельных вычислений: сокращение времени выполнения, выполнение более крупных вычислений или снижение энергопотребления.

Более быстрое время выполнения с бóльшим числом вычислительных ядер

Сокращение времени выполнения приложения, или его ускорение, часто считается первостепенной целью параллельных вычислений. И действительно обычно это имеет наибольшие последствия. Параллельные вычисления способны ускорять интенсивные вычисления, обработку мультимедиа и операции с большими данными независимо от того, требуются ли для обработки ваших приложений дни или даже недели, или же результаты необходимы в реальном времени.

В прошлом программист тратил больше усилий на последовательную оптимизацию, чтобы выжать несколько процентных улучшений. Теперь есть потенциал для улучшения на порядки, с многочисленными вариантами на выбор. Это создает новую проблему в разведывании возможных параллельных парадигм – больше возможностей, чем численность программистов. Но глубокое знание вашего приложения и понимание возможностей параллелизма непременно поведут вас по более ясному пути к сокращению времени выполнения вашего приложения.

Более крупные размеры задач с бóльшим числом вычислительных узлов

Выявляя параллелизм в своем приложении, вы можете масштабировать размер вашей задачи вертикально до размерностей, недоступных для последовательного приложения. Это связано с тем, что объем вычислительных ресурсов обуславливает то, что можно сделать, а параллелизм позволяет работать с большими ресурсами, предоставляя возможности, которые раньше никогда не рассматривались. Более крупные размеры обеспечиваются за счет большего объема основной памяти, дискового хранилища, пропускной способности сетей и диска, а также центральных процессоров (CPU). По аналогии с супермаркетом, как упоминалось ранее, выявление параллелизма эквивалентно найму большего числа кассиров либо открытию большего числа касс самостоятельной оплаты покупок с целью обслуживания большего и растущего числа клиентов.

Энергоэффективность: делая больше с меньшими затратами

Одной из новых областей влияния параллельных вычислений является энергоэффективность. С появлением параллельных ресурсов в наладонных устройствах параллелизм может ускорять работу приложений. Это позволяет устройству быстрее возвращаться в спящий режим и дает

возможность использовать более медленные, но более параллельные процессоры, потребляющие меньше энергии. Отсюда перенос работы тяжеловесных мультимедийных приложений на GPU-процессоры может оказать более существенное влияние на энергоэффективность, а также значительно повысить производительность. Чистый результат использования параллелизма снижает энергопотребление и продлевает срок службы батареи, что является сильным конкурентным преимуществом в этой рыночной нише.

Еще одна область, в которой важность энергоэффективности неоспорима, – это удаленные датчики, сетевые устройства и операционные устройства, развернутые на местах, таких как удаленные метеостанции. Нередко без больших источников питания эти устройства должны быть способны функционировать малыми пакетами с небольшим объемом ресурсов. Параллелизм расширяет возможности, которые могут имплементироваться на этих устройствах, и разгружает работу центральной вычислительной системы в растущем тренде, который называется периферийным вычислением (edge compute). Перемещение вычислений на самый край сети обеспечивает возможность вести обработку в источнике данных, сжимая их в меньший результирующий набор, который легче отправлять по сети.

Точный расчет энергетических затрат приложения является сложной задачей без прямых измерений энергопотребления. Однако вы можете оценить затраты путем умножения расчетной тепловой мощности производителя на время выполнения приложения и число используемых процессоров. Расчетная тепловая мощность (thermal design power, аббр. TDP) – это скорость, с которой энергия расходуется при типичных эксплуатационных нагрузках. Потребление энергии для вашего приложения можно оценить по формуле:

$$P = (N \text{ Процессоров}) \times (R \text{ Ватт/Процессоры}) \times (T \text{ часов}),$$

где P – это потребление энергии, N – число процессоров, R – расчетная тепловая мощность и T – время выполнения приложения.

Пример

16-ядерный процессор Intel Xeon E5-4660 имеет расчетную тепловую мощность 120 Вт. Предположим, что ваше приложение использует 20 из этих процессоров в течение 24 часов до полного завершения работы. Оценочное потребление энергии для вашего приложения составляет:

$$P = (20 \text{ Процессоров}) \times (120 \text{ Вт/Процессоры}) \times (24 \text{ часа}) = 57.60 \text{ кВт/ч}.$$

В общем случае GPU имеют более высокую расчетную тепловую мощность, чем современные CPU, но потенциально способны сократить время работы или потребовать всего нескольких GPU для получения того же результата. Можно применить ту же формулу, что и раньше, где N теперь рассматривается как число GPU.

Пример

Предположим, что вы портировали свое приложение на мульти-GPU-платформу. Теперь вы можете выполнять свое приложение на четырех GPU NVIDIA Tesla V100 за 24 часа! GPU Tesla V100 от NVIDIA имеет максимальную тепловую мощность 300 Вт. Оценочное потребление энергии для вашего приложения составляет:

P = (4 GPU-процессора) × (300 Вт/GPU-процессоры) × (24 часа) = 28.80 кВт/ч.

В этом примере приложение с ускорением на основе GPU работает с вдвое меньшими энергозатратами, чем версия только для CPU. Обратите внимание, что в этом случае, даже несмотря на то, что время до решения остается прежним, затраты энергии сокращаются вдвое!

Достижение снижения энергозатрат с помощью ускорителей, таких как GPU-процессоры, требует наличия в приложении достаточного параллелизма, который можно выявить. Это позволяет эффективно пользоваться ресурсами устройства.

Параллельные вычисления могут снизить затраты

Фактические денежные затраты становятся все более заметной озабоченностью для коллективов разработчиков программного обеспечения, пользователей программного обеспечения и исследователей в равной степени. По мере роста размеров приложений и систем нам необходимо проводить анализ затрат и выгод на имеющихся у нас ресурсах. Например, в следующих крупных системах высокопроизводительных вычислений (HPC) затраты на электроэнергию, по прогнозам, в три раза превысят стоимость приобретения оборудования.

Затраты на использование также способствовали развитию облачных вычислений в качестве альтернативы, которая все чаще внедряется в научных кругах, стартапах и отраслях промышленности. Как правило, поставщики облачных служб выставляют счета в зависимости от типа и объема используемых ресурсов, а также количества времени, потраченного на их использование. Хотя GPU-процессоры, как правило, стоят дороже, чем CPU в расчете на единицу времени, в некоторых приложениях могут использоваться GPU-ускорители, чтобы обеспечивать достаточное сокращение времени выполнения по сравнению с расходами на CPU с целью снижения затрат.

1.1.2 Предостережения, связанные с параллельными вычислениями

Параллельные вычисления не являются панацеей. Многие приложения недостаточно велики или не требуют достаточного времени выполнения, чтобы нуждаться в параллельных вычислениях. У некоторых может даже не быть достаточного внутреннего параллелизма, который можно

было бы эксплуатировать. Кроме того, транзит приложений на использование мультиядерного и многоядерного оборудования (GPU) требует выделенных усилий, которые могут временно отвлекать внимание от прямых исследований или продуктовых целей. Сначала следует осознать целесообразность затрат времени и усилий. Всегда важнее обеспечить функционирование приложения и генерирование им желаемого результата, прежде чем его ускорять и масштабировать его вертикально до более крупных задач.

Мы настоятельно рекомендуем вам начать свой параллельно-вычислительный проект с плана. Важно знать варианты, которые доступны для ускорения приложения, а затем выбрать наиболее подходящий для вашего проекта. После этого крайне важно иметь разумную оценку затраченных усилий и потенциальных выгод (с точки зрения стоимости в долларах, энергопотребления, времени до решения и других показателей, которые могут представлять важность). В этой главе мы начнем давать вам авансом знания и навыки для принятия решений по параллельно-вычислительным проектам.

1.2 Фундаментальные законы параллельных вычислений

В последовательных вычислениях все операции ускоряются по мере увеличения тактовой частоты. В отличие от них в случае параллельных вычислений нам необходимо немного поразмыслить и модифицировать наши приложения так, чтобы задействовать параллельное оборудование в полной мере. Почему так важен объем параллелизма? Для того чтобы в этом разобраться, давайте взглянем на законы параллельных вычислений.

1.2.1 Предел на параллельные вычисления: закон Амдала

Нам необходим подход к расчету потенциального ускорения вычисления, основываясь на объеме параллельного кода. Это можно сделать, используя Закон Амдала, предложенный Джином Амдалом в 1967 году. Указанный закон описывает ускорение задачи фиксированного размера по мере увеличения числа процессоров. Следующее ниже уравнение это показывает, где P – это параллельная доля кода, S – последовательная доля, означающая, что $P + S = 1$, и N – это число процессоров:

$$\text{Ускорение}(N) = \frac{1}{S + \frac{P}{N}}.$$

Закон Амдала подчеркивает, что независимо от того, какой быстрой мы делаем параллельную часть кода, мы всегда будем лимитированы

последовательной частью. На рис. 1.3 показан этот предел. Такое масштабирование задачи фиксированного размера называется сильным масштабированием.

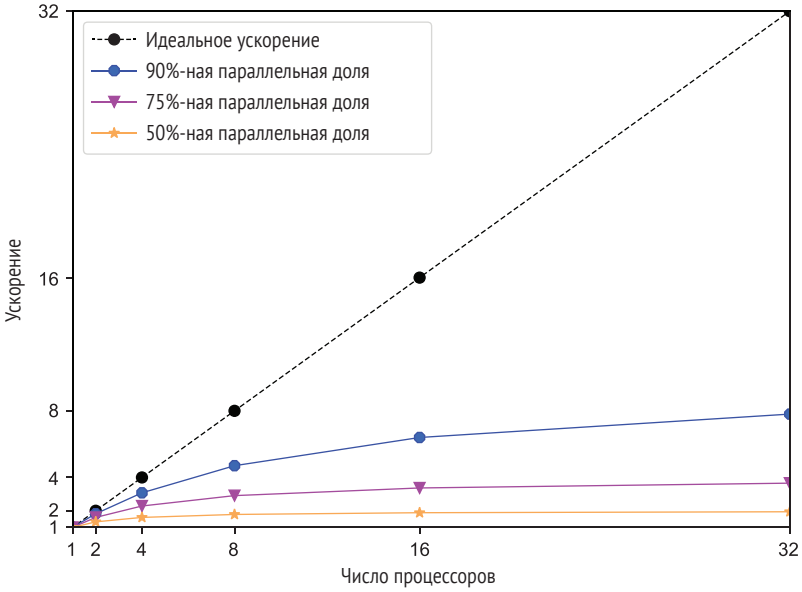


Рис. 1.3 Ускорение для задачи фиксированного размера в соответствии с законом Амдала показано как функция от числа процессоров. Линии показывают идеальное ускорение, когда параллелизуется 100 % алгоритма, а также для 90, 75 и 50 %. Закон Амдала гласит, что ускорение лимитировано долями кода, которые остаются последовательными

ОПРЕДЕЛЕНИЕ Сильное масштабирование представляет время до решения в зависимости от числа процессоров для задачи фиксированного суммарного размера.

1.2.2 Преодоление параллельного предела: закон Густафсона–Барсиса

В 1988 году Густафсон и Барсис указали на то, что прогоны параллельного кода должны увеличивать размер задачи по мере добавления большего числа процессоров. Это дает нам альтернативный подход к расчету потенциального ускорения нашего приложения. Если размер задачи растет пропорционально числу процессоров, то ускорение теперь выражается как

$$\text{Ускорение}(N) = N - S \times (N - 1),$$

где N – это число процессоров, а S – последовательная доля, как и раньше. Как следствие, более крупная задача может быть решена за одно и то

же время с использованием большего числа процессоров. Это предоставляет дополнительные возможности для привлечения параллелизма. И действительно увеличение размера задачи вместе с числом процессоров имеет смысл, потому что пользователь приложения хочет извлекать выгоду не только из мощности дополнительного процессора, но и хочет использовать дополнительную память. Масштабирование времени выполнения для этого сценария, показанное на рис. 1.4, называется слабым масштабированием.

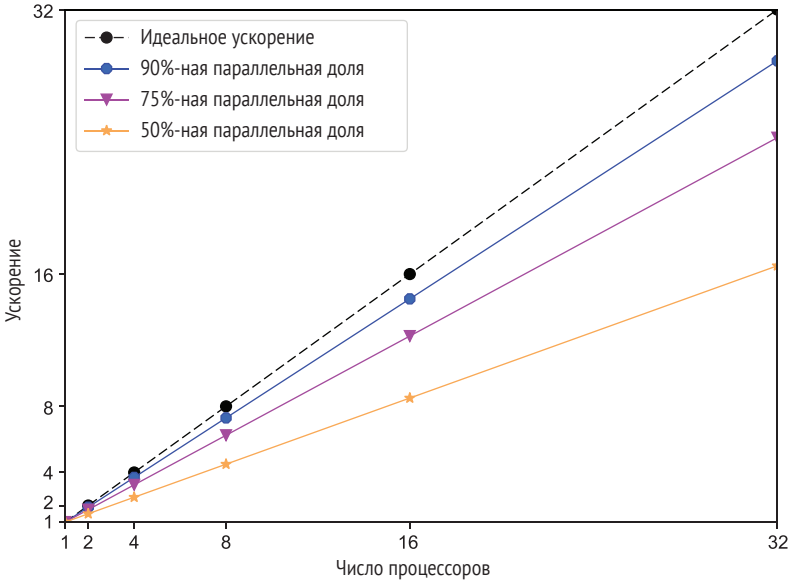


Рис. 1.4 Ускорение для случаев, когда размер задачи увеличивается вместе с увеличением числа доступных процессоров в соответствии с законом Густафсона–Барсиса, показано как функция числа процессоров. Линии показывают идеальное ускорение, когда параллелизуется 100 % алгоритма, а также для 90, 75 и 50 %

ОПРЕДЕЛЕНИЕ Слабое масштабирование представляет время до решения в зависимости от числа процессоров для задачи фиксированного размера в расчете на процессор.

На рис. 1.5 наглядно показана разница между сильным и слабым масштабированием. Аргументация слабого масштабирования в отношении того, что размер вычислительной сетки должен оставаться постоянным на каждом процессоре, позволяет эффективно пользоваться ресурсами дополнительного процессора. С позиций сильного масштабирования все внимание сконцентрировано в первую очередь на ускорении вычисления. На практике важны как сильное, так и слабое масштабирование, поскольку они решают разные пользовательские сценарии.

Термин «масштабируемость» часто используется для обозначения возможности добавлять больше параллелизма в аппаратное или про-

граммное обеспечение и существования совокупного предела возможного улучшения. В то время как традиционный фокус внимания лежит на масштабировании времени выполнения, мы выдвигаем аргумент, что масштабирование памяти часто имеет бóльшую значимость.

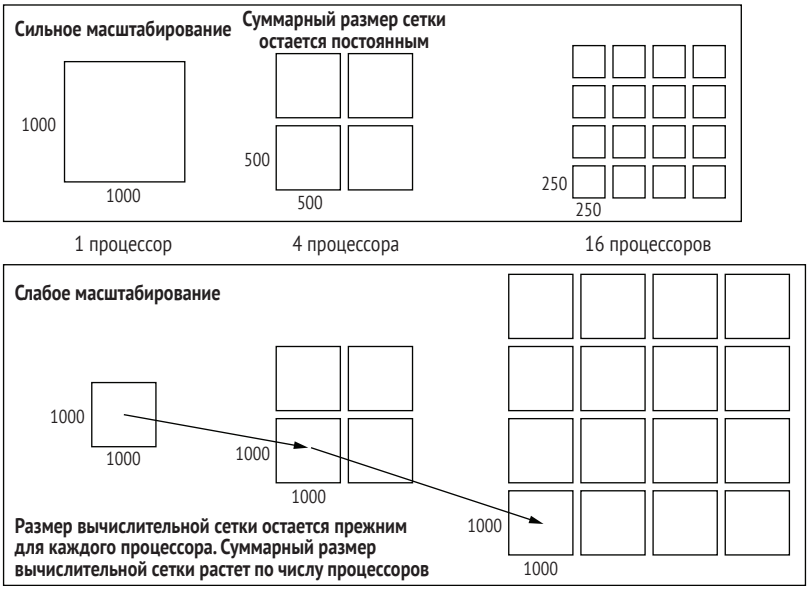


Рис. 1.5 Сильное масштабирование сохраняет тот же совокупный размер задачи и распределяет ее по дополнительным процессорам. При слабом масштабировании размер вычислительной сетки остается одинаковым для каждого процессора, а суммарный размер увеличивается

На рис. 1.6 показано приложение с лимитированной масштабируемостью памяти. Реплицированный массив (R) – это набор данных, который дублируется во всех процессорах. Распределенный массив (D) разбивается на разделы и распределяется между процессорами.

Например, в игровом симуляторе 100 персонажей могут быть распределены между 4 процессорами по 25 персонажей на каждом процессоре. Но карта игрового поля может быть скопирована на каждый процессор. На рис. 1.6 реплицированный массив дублируется по всей вычислительной сетке. Поскольку этот рисунок относится к слабому масштабированию, размер задачи увеличивается вместе с увеличением числа процессоров. Для 4 процессоров массив в 4 раза больше на каждом процессоре.

Поскольку размер задачи растет, вскоре на процессоре не хватит памяти для выполнения задания. Лимитированное масштабирование времени выполнения означает, что задание выполняется медленно; лимитированное масштабирование памяти означает, что задание не может выполняться вообще. Это также тот случай, что если память приложения может быть распределена, то время выполнения обычно масштабируется тоже. Однако обратное не обязательно верно.

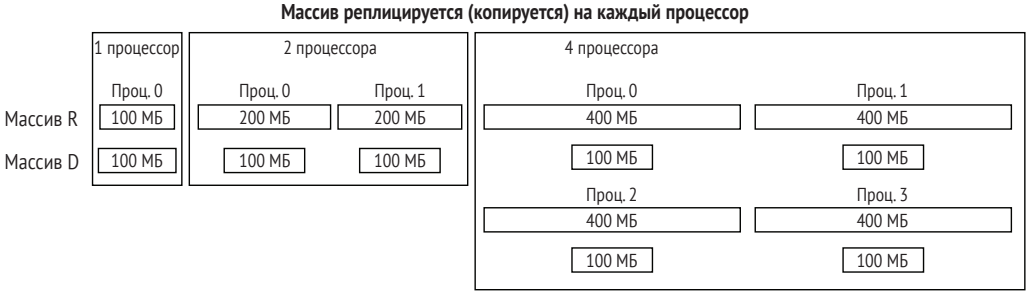


Рис. 1.6 Распределенные массивы остаются того же размера, что и задача, а число процессоров удваивается (слабое масштабирование). Но реплицированные (скопированные) массивы нуждаются во всех данных на каждом процессоре, при этом память быстро растет вместе с увеличением числа процессоров. Даже если время выполнения масштабируется слабо (остается постоянным), требования к памяти лимитируют масштабируемость

Одна из точек зрения на интенсивное вычислительное задание состоит в том, что каждый байт памяти затрагивается в каждом цикле обработки, а время выполнения является функцией от размера памяти. Сокращение размера памяти обязательно приведет к сокращению времени выполнения. Таким образом, первоначальное внимание в параллелизме должно быть направлено на сокращение размера памяти по мере увеличения числа процессоров.

1.3 Как работают параллельные вычисления?

Параллельные вычисления требуют сочетания понимания оборудования, программного обеспечения и параллелизма при разработке приложения. Это больше, чем просто передача сообщений или потокообразование. Современное оборудование и программное обеспечение предоставляют массу разных возможностей для параллелизации вашего приложения. Некоторые из этих возможностей могут быть скомбинированы, чтобы обеспечить еще большую эффективность и ускорение.

Важно иметь понимание параллелизации в вашем приложении и того, как разные аппаратные компоненты позволяют вам ее выявлять. Кроме того, разработчики должны понимать, что между вашим исходным кодом и оборудованием ваше приложение должно проходить дополнительные слои, включая компилятор и операционную систему (рис. 1.7).

Как разработчик, вы несете ответственность за слой прикладного программного обеспечения, в котором содержится ваш исходный код. В исходном коде вы выбираете язык программирования и параллельные программные интерфейсы, которые используете для задействования опорного оборудования. В дополнение вы принимаете решение о том, как разбить свою работу на параллельные модули. Компилятор служит для транслирования вашего исходного кода в форму, исполнимую вашим оборудованием. Получив в свое распоряжение эти команды,

операционная система управляет их исполнением на компьютерном оборудовании.

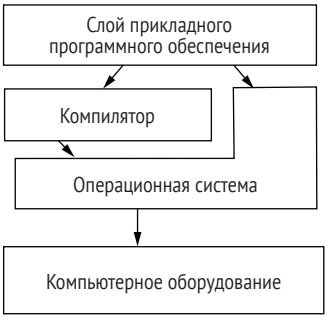


Рис. 1.7 Параллелизация выражается в слое прикладного программного обеспечения, который соотносится с компьютерным оборудованием через компилятор и OS

Мы покажем вам на примере процесс введения параллелизации в алгоритм с помощью прототипного приложения. Указанный процесс происходит в слое прикладного программного обеспечения, но требует понимания компьютерного оборудования. Пока что мы воздержимся от обсуждения выбора компилятора и операционной системы. Мы будем поступательно добавлять каждый слой параллелизации, чтобы у вас имелась возможность видеть принцип ее работы. В каждой параллельной стратегии мы объясним характер влияния доступного оборудования на сделанный выбор. Цель этого состоит в том, чтобы продемонстрировать то, как аппаратные функциональности влияют на параллельные стратегии. Мы классифицируем параллельные подходы, которые разработчик может принять, на:

- параллелизацию на основе процессов (process-based parallelization);
- параллелизацию на основе потоков (thread-based parallelization);
- векторизацию (vectorization);
- обработку в потоковом режиме (stream processing).

Следуя этому примеру, мы введем модель, которая поможет вам рассуждать о современном оборудовании. Эта модель разбивает современное вычислительное оборудование на отдельные компоненты и различные вычислительные устройства. В эту главу включен упрощенный взгляд на память. Более подробный обзор иерархии памяти представлен в главах 3 и 4. Наконец, мы подробнее обсудим слои приложения и программное обеспечение.

Как уже упоминалось, мы классифицируем параллельные подходы, которые разработчик может принять, на параллелизацию на основе процессов, параллелизацию на основе потоков (т. е. виртуальных ядер), векторизацию и обработку в потоковом режиме. Параллелизация на основе отдельных процессов с их собственными пространствами памяти может представлять собой распределенную память на разных узлах компьютера или внутри узла. Обработка в потоковом режиме обычно связана с GPU-процессорами. Модель современного оборудования и прикладного программного обеспечения поможет вам лучше понять принцип

планирования переноса вашего приложения на текущее параллельное оборудование.

1.3.1 Пошаговое ознакомление с примером приложения

В целях этого введения в параллелизацию мы рассмотрим подход на основе параллелизма данных. Это одна из наиболее распространенных стратегий применения параллельных вычислений. Мы выполним вычисления на пространственной вычислительной сетке, состоящей из регулярной двумерной (2D) решетки прямоугольных элементов или ячеек. Шаги (резюмированные здесь и подробно описанные позже) для создания пространственной расчетной сетки и подготовки к вычислению таковы:

1. дискретизировать (подразделить) задачу на более мелкие ячейки или элементы;
2. определить вычислительное ядро (операцию) для выполнения на каждом элементе вычислительной сетки;
3. добавить следующие слои параллелизации на CPU-процессорах и GPU-процессорах для выполнения расчета:
   - векторизацию – работать на более одной единице данных за один раз;
   - потоки – развертывать более одного вычислительного маршрута для привлечения большего числа процессорных ядер;
   - процессы – отделять программные экземпляры для распространения расчета по отдельным пространствам памяти;
   - выгрузку расчета на GPU-процессоры – отправлять данные в GPU для расчета.

Мы начинаем с двумерной задачной области участка пространства. Для целей иллюстрации мы будем использовать 2D-изображение вулкана Кракатау (рис. 1.8) в качестве примера. Целью наших расчетов может быть моделирование шлейфа вулканического выброса, возникающего в результате цунами или раннее обнаружение извержения вулкана с использованием машинного обучения. Для всех этих вариантов скорость вычислений имеет решающее значение, если мы хотим, чтобы реальновременные результаты информировали наши решения.

Рис. 1.8 Пример двухмерной пространственной области для численной симуляции. Численные симуляции обычно предусматривают стенсильные операции (см. рис. 1.11) или большие матрично-векторные системы. Эти типы операций часто используются при моделировании жидкостей для продуцирования предсказаний времени прибытия цунами, прогнозов погоды, распространения дымового шлейфа и других процессов, необходимых для принятия обоснованных решений

Шаг 1. Дискретизировать задачу на более мелкие ячейки или элементы

Для любого подробного расчета мы должны сначала разбить область задачи на более мелкие части (рис. 1.9). Указанный процесс называется дискретизацией. При обработке изображений это часто просто пикселы в растровом изображении. Для вычислительной области они называются ячейками или элементами. Коллекция ячеек или элементов образует вычислительную сетку, которая охватывает пространственный участок для симуляции. Значения данных для каждой ячейки могут быть целыми числами, вещественными числами или числами двойной точности.

Рис. 1.9 Область дискретизирована на ячейки. Для каждой ячейки в вычислительной области такие свойства, как высота волны, скорость жидкости или плотность дыма, решаются в соответствии с физическими законами. В конечном счете стенсильная операция или матрично-векторная система представляет эту дискретную схему

Шаг 2. Определить вычислительное ядро или операцию, необходимую для выполнения на каждом элементе сетки

Расчеты на этих дискретизированных данных часто являются некоторой формой стенсильной операции, так именуемой, потому что она предусматривает шаблон смежных ячеек для вычисления нового значения для каждой ячейки. Это может быть среднее значение (операция размытия, которая размывает изображение или делает его более расплывчатым), градиент (обнаружение краев, которое делает края изображения более четкими) или другая более сложная операция, связанная с решением физических систем, описываемых уравнениями в частных производных (PDE). На рис. 1.10 показана стенсильная операция в виде пятиточечного трафарета, который выполняет операцию размытия, используя средневзвешенное значение стенсильных значений.

Но что это за уравнения в частных производных? Давайте вернемся к нашему примеру и представим, что на этот раз это цветное изображение, состоящее из отдельных красных, зеленых и синих массивов, составляющих цветовую модель RGB. Термин «частный» здесь означает, что существует более одной переменной и что мы отделяем изменение красного

цвета в пространстве и времени от изменения зеленого и синего. Затем мы выполняем оператор размытия отдельно для каждого из этих цветов.

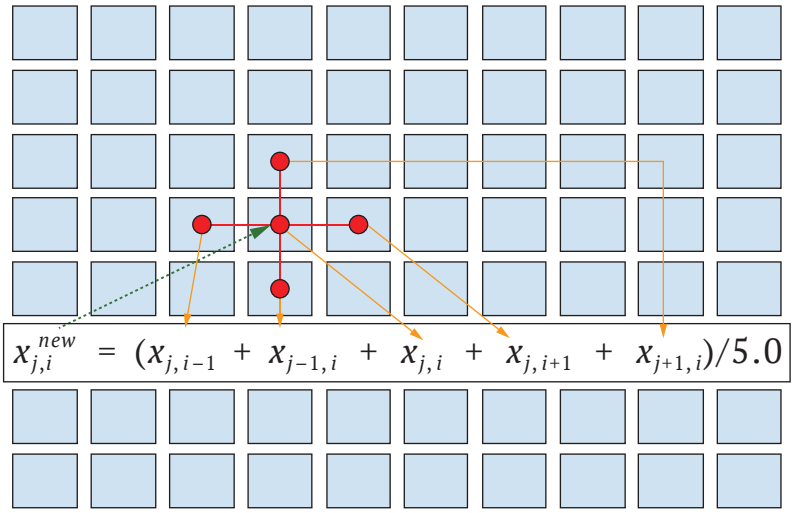


Рис. 1.10 Пятиточечный стенсильный оператор в виде поперечного шаблона на вычислительной сетке. Отмеченные трафаретом данные считываются во время операции и сохраняются в центральной ячейке. Этот шаблон повторяется для каждой ячейки. Оператор размытия, один из более простых стенсильных операторов, представляет собой взвешенную сумму пяти точек, отмеченных большими точками, и обновляет значение в центральной точке трафарета. Этот тип операций выполняется для операций сглаживания или численных симуляций распространения волн

Есть еще одно требование: нам нужно применять скорость изменения во времени и пространстве. Другими словами, красный цвет будет распространяться с одной скоростью, а зеленый и синий – с другой. Это может делаться с целью продуцирования на изображении специального эффекта или может описывать то, как реальные цвета просачиваются и сливаются в фотографическом изображении во время проявления. В научном мире вместо красного, зеленого и синего мы могли бы иметь массу и скорость x и y. Если добавить немного больше физики, то мы могли бы получить движение волны или пепельного шлейфа.

Шаг 3. Векторизация для работы с более чем одной единицей данных за один раз

Мы начинаем вводить параллелизацию с рассмотрения векторизации. Что такое векторизация? Некоторые процессоры имеют возможность оперировать на более чем одной порции данных за один раз; эта возможность называется векторными операциями. Затененные блоки на рис. 1.11 иллюстрируют то, как несколько значений данных обрабатываются одновременно в модуле векторной обработки в процессоре одной командой за один тактовый цикл.

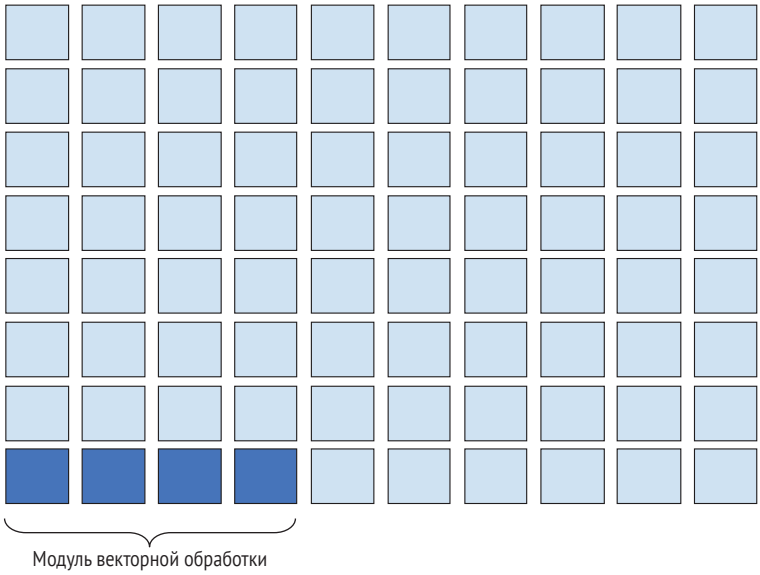


Рис. 1.11 Специальная векторная операция выполняется на четырех значениях двойной точности. Эта операция может исполняться за один тактовый цикл с небольшими дополнительными энергозатратами на последовательную операцию

Шаг 4. Потоки для развертывания более одного вычислительного маршрута с целью привлечения большего числа обрабатывающих ядер

Поскольку большинство CPU сегодня имеет по крайней мере четыре обрабатывающих ядра, мы используем потокообразование для оперирования ядрами в одновременном режиме в четырех строках за один раз. Этот процесс показан на рис. 1.12.

Шаг 5. Процессы для распространения вычислений по отдельным пространствам памяти

Мы можем еще больше разбить работу между процессорами на два настольных компьютера, в параллельной обработке часто именуемых узлами. Когда работа разбита на узлы, пространства памяти под каждый узел отличимы и отделены. На это указывает наличие промежутка между строками, как показано на рис. 1.13.

Даже для этого довольно скромного аппаратного сценария существует потенциальное ускорение в 32 раза. Об этом свидетельствуют следующие ниже данные:

2 настольных компьютера (узла) × 4 ядра × (модуль векторной обработки шириной 256 бит)/(64-битное значение двойной точности) = 32-кратное потенциальное ускорение.

Если взять высококлассный кластер с 16 узлами, 36 ядрами на узел и 512-битовым векторным процессором, то потенциальное теоретиче-

ское ускорение будет 4608-кратным по сравнению с последовательным процессом:

16 узлов × 36 ядер × (модуль векторной обработки шириной 512 бит)/ (64-битное значение двойной точности) = 4.608-кратное потенциальное ускорение.

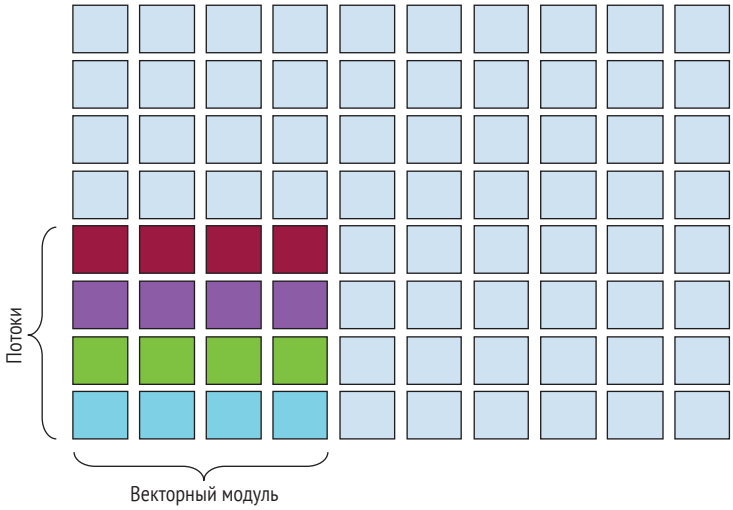


Рис. 1.12 Четыре потока обрабатывают четыре строки векторных модулей одновременно

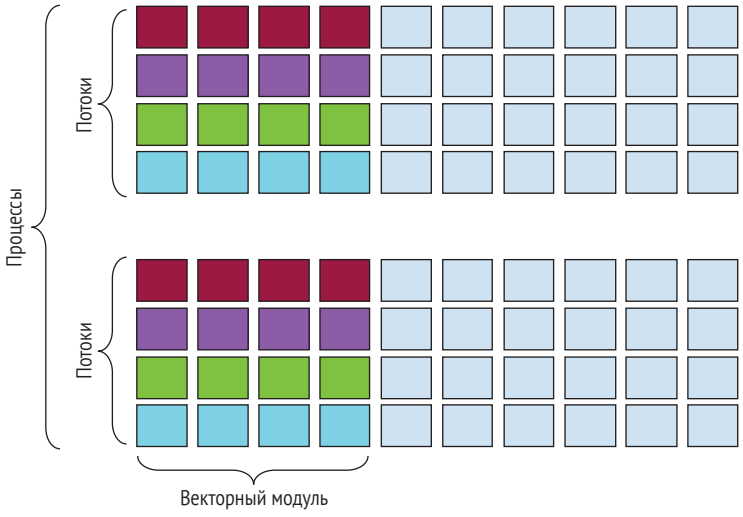


Рис. 1.13 Этот алгоритм можно параллелизовать еще больше, распределив блоки 4×4 между отличимыми процессами. В каждом процессе используется четыре потока, каждый из которых обрабатывает векторный модуль шириной четыре узла за один тактовый цикл. Дополнительное пустое пространство на рисунке иллюстрирует границы процесса

Шаг 6. Выгрузка расчета на GPU-процессоры

GPU – это еще один аппаратный ресурс для подстегивания параллелизации. С помощью GPU-процессоров мы можем привлекать к работе большое число потоковых мультипроцессоров (streaming multiprocessors, аббр. SMs). Например, на рис. 1.14 показан принцип разделения работы отдельно на плитки размером 8×8. Используя технические характеристики оборудования для GPU NVIDIA Volta, на этих плитках могут оперировать 32 ядра двойной точности, распределенные по 84 потоковым мультипроцессорам, что дает нам в общей сложности 2688 ядер двойной точности, работающих одновременно. Если у нас есть один GPU на узел в 16-узловом кластере, каждый с 2688 потоковыми мультипроцессорами двойной точности, то это составит 43 008-путную параллелизацию из 16 GPU-процессоров.

Рис. 1.14 На GPU длина вектора намного больше, чем на CPU. Здесь плитки размером 8×8 распределены по рабочим группам GPU

Эти цифры впечатляют, но пока что мы должны умерить ожидания, признав, что фактическое ускорение значительно отстает от этого полного потенциала. Теперь нашей задачей становится организация таких экстремальных и разрозненных слоев параллелизации, чтобы добиться как можно большего ускорения.

В этом пошаговом ознакомлении с высокоуровневым приложением мы опустили массу важных деталей, которые рассмотрим в последующих главах. Но даже этот номинальный уровень детализации подчеркивает несколько стратегий для выявления параллелизации алгоритма. Для того чтобы иметь возможность разрабатывать похожие стратегии для решения других задач, необходимо понимание современного ап-

паратного и программного обеспечения. Теперь мы глубже погрузимся в текущие аппаратные и программные модели. Эти концептуальные модели являются упрощенными представлениями разнообразного реального аппаратного обеспечения, чтобы избежать сложности и сохранить общность в быстро эволюционирующих системах.

1.3.2 Аппаратная модель для современных гетерогенных параллельных систем

С целью получения базового понимания принципа работы параллельных вычислений мы объясним компоненты современного оборудования. Начнем с того, что в динамической оперативной памяти, именуемой DRAM, хранится информация, или данные. Ядро процессора, или просто ядро, выполняет арифметические операции (сложение, вычитание, умножение, деление), оценивает логические инструкции, загружает и сохраняет данные из DRAM. Когда операция выполняется на данных, команды и данные загружаются из памяти в ядро, обрабатываются и сохраняются обратно в память. Современные CPU, часто именуемые просто процессорами, оснащены многочисленными ядрами, способными исполнять эти операции в параллельном режиме. Также получают распространение системы, оснащенные ускорительным оборудованием, таким как GPU-процессоры. GPU-процессоры оснащены тысячами ядер и пространством памяти, которое отделено от DRAM центрального процессора, CPU.

Комбинация процессора (или двух), DRAM и ускорителя составляет вычислительный узел, на который можно ссылаться в контексте одного домашнего настольного компьютера или «стойки» в суперкомпьютере. Вычислительные узлы могут соединяться друг с другом одной или несколькими сетями. Такое соединение иногда именуется межсоединением. Концептуально узел запускает один экземпляр операционной системы, который управляет и контролирует все аппаратные ресурсы. Поскольку оборудование становится все сложнее и неоднороднее, мы начнем с упрощенных моделей компонентов системы, чтобы каждый из них был более очевидным.

Архитектура на основе распределенной памяти: перекрестно-узловой параллельный метод

Одним из первых и наиболее масштабируемых подходов к параллельным вычислениям является кластер распределенной памяти (рис. 1.15). Каждый CPU имеет свою собственную локальную память, состоящую из DRAM, и соединен с другими CPU коммуникационной сетью. Хорошая масштабируемость кластеров распределенной памяти обусловлена их кажущейся безграничной способностью включать в себя большее число узлов.

Эта архитектура также обеспечивает некоторую локальность памяти, разделяя суммарную адресную память на меньшие подпространства для

каждого узла, что делает доступ к памяти вне узла явно отличным от доступа на узле. Это вынуждает программиста явно обращаться к разным участкам памяти. Недостатком этой архитектуры является то, что программист должен управлять подразделением пространств памяти в самом начале приложения.

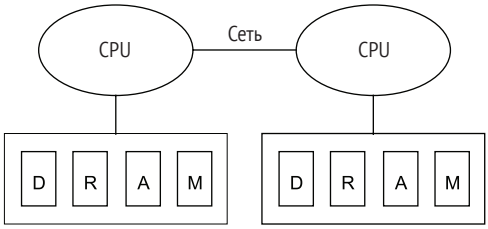


Рис. 1.15 Архитектура на основе распределенной памяти связывает узлы, состоящие из отдельных пространств памяти. Этими узлами могут быть рабочие станции или стойки

Архитектура на основе совместной памяти: наузловой параллельный метод

Альтернативный подход присоединяет два CPU напрямую к одной и той же совместной памяти (рис. 1.16). Сила этого подхода заключается в том, что процессоры используют совместно одно и то же адресное пространство, что упрощает программирование. Но он вводит потенциальные конфликты памяти, что приводит к проблемам с правильностью и производительностью. Синхронизирование доступа к памяти и значений между CPU или обрабатывающими ядрами на мультиядерном CPU усложнено и обходится дорого.

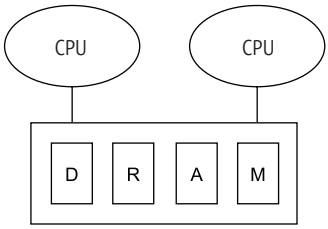


Рис. 1.16 Архитектура на основе совместной памяти обеспечивает параллелизацию внутри узла

Добавление большего числа CPU и обрабатывающих ядер не увеличивает объем доступной приложению памяти. Эти затраты и затраты на синхронизацию лимитируют масштабируемость архитектуры на основе совместной памяти.

Модули векторной обработки: несколько операций с одной командой

Почему бы просто не увеличить тактовую частоту процессора, чтобы получить более выскую мощность, как это делалось в прошлом? Самым большим ограничением в увеличении тактовой частоты CPU является то, что он требует больше энергии и выделяет больше тепла. Будь то супер-

компьютерный HPC-центр с пределами на инсталлированные силовые линии либо ваш мобильный телефон с лимитированной емкостью батареи, все устройства сегодня имеют пределы по электропитанию. Эта проблема называется силовой стеной.

Вместо того, чтобы увеличивать тактовую частоту, почему бы не выполнять более одной операции за цикл? Это идея лежит в основе возрождения векторизации на многих процессорах. Для выполнения нескольких операций в модуле векторной обработки требуется лишь немного больше энергии, чем для одной операции (более формально именуемой скалярной операцией). С помощью векторизации мы можем обрабатывать больше данных за один тактовый цикл, чем при последовательном процессе. Требования к питанию для нескольких операций практически не меняются (по сравнению с одной), а сокращение времени выполнения может привести к снижению энергопотребления приложения. Во многом подобно четырехполосной автостраде, которая позволяет четырем автомобилям двигаться одновременно, по сравнению с однополосной дорогой, векторная операция обеспечивает более высое значение мощности обработки. И действительно четыре маршрута через модуль векторной обработки, показанные в разных оттенках на рис. 1.17, обычно называются полосами векторной операции.

Большинство CPU и GPU имеет некоторые способности по векторизации или эквивалентные операции. Объем данных, обрабатываемых за один тактовый цикл, длина вектора, зависит от размера модулей векторной обработки на процессоре. В настоящее время наиболее распространенная длина вектора составляет 256 бит. Если дискретизированные данные равны 64-битным значениям двойной точности, то мы можем выполнять четыре операции с плавающей точкой одновременно как векторную операцию. Как показано на рис. 1.17, аппаратные модули векторной обработки загружают по одному блоку данных за раз, одновременно выполняют одну операцию с данными, а затем сохраняют результат.

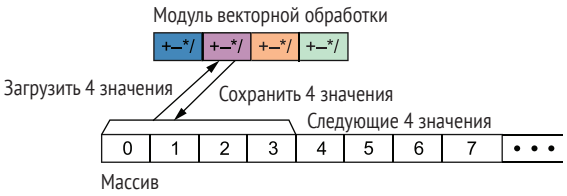


Рис. 1.17 Пример векторной обработки, при которой манипулирование четырьмя векторными модулями массива осуществляется одновременно

Ускорительное устройство: узкоцелевой надстраиваемый процессор

Ускорительное устройство – это дискретное оборудование, предназначенное для быстрого исполнения специфических задач. Наиболее распространенным ускорителем является GPU. При использовании для вычислений это устройство иногда называют общецелевым графическим процессором (general-purpose GPU или GPGPU). GPU содержит много ма-

лых обрабатывающих ядер, именуемых потоковыми мультипроцессорами (SM). Хотя SM-процессоры проще, чем ядро CPU, они обеспечивают огромный объем вычислительной мощности. Обычно малый интегрированный GPU можно найти прямо на CPU.

Большинство современных компьютеров также имеет отдельный дискретный GPU, подключенный к CPU интерфейсной шиной для подключения периферийных компонентов (PCI-шиной) (рис. 1.18). Эта шина увеличивает затраты на передачу данных и команд, но дискретная карта часто является более мощной, чем интегрированное устройство. Например, в системах высокого класса NVIDIA использует NVLink, а AMD Radeon использует свою Infinity Fabric для снижения затрат на передачу данных, но эти затраты по-прежнему значительны. Мы подробнее обсудим эту интересную архитектуру GPU в главах 9–12.

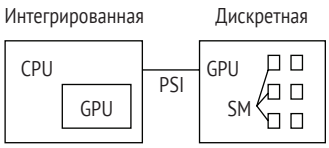


Рис. 1.18 GPU-процессоры бывают двух видов: интегрированные и дискретные. Дискретные или выделенные GPU обычно имеют большое число потоковых мультипроцессоров и собственную DRAM. Для доступа к данным на дискретном GPU требуется связь по шине PCI

Общая гетерогенная параллельная архитектурная модель

Теперь давайте скомбинируем все эти разные аппаратные архитектуры в одну модель (рис. 1.19). Два узла, каждый с двумя CPU, совместно используют одну и ту же память DRAM. Каждый CPU является двухъядерным процессором с интегрированным GPU. Дискретный GPU на шине PCI также подключается к одному из CPU. Хотя CPU используют основную память совместно, они обычно находятся в разных участках неравномерного доступа к памяти (NUMA). Это означает, что доступ к памяти второго CPU обходится дороже, чем доступ к его собственной памяти.

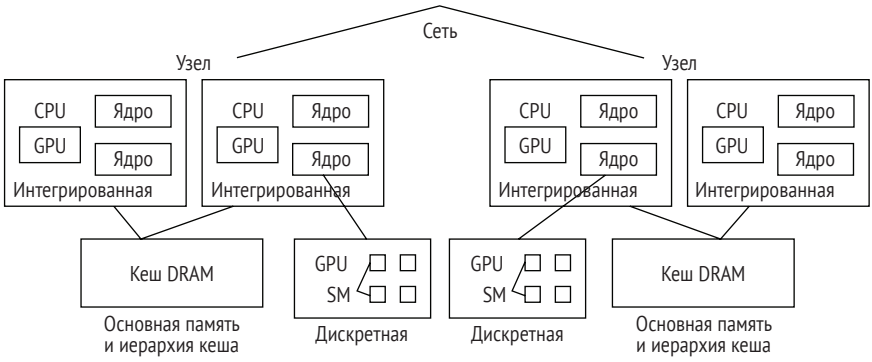


Рис. 1.19 Общая гетерогенная параллельная архитектурная модель, состоящая из двух узлов, соединенных сетью. Каждый узел имеет мультиядерный CPU с интегрированным и дискретным GPU и некоторой памятью (DRAM). Современное вычислительное оборудование обычно имеет ту или иную расстановку этих компонентов

На протяжении всего этого обсуждения оборудования мы упоминали упрощенную модель иерархии памяти, показывающую только DRAM или основную память. Мы показали кеш в комбинированной модели (рис. 1.19), но без подробностей о его составе или о том, как он функционирует. Мы оставляем наше обсуждение сложностей управления памятью, включая несколько уровней кеша, для главы 3. В этом разделе мы просто представили модель современного оборудования, чтобы помочь вам идентифицировать имеющиеся компоненты, чтобы иметь возможность выбрать параллельную стратегию, наиболее подходящую для вашего приложения и оборудования.

1.3.3 Прикладная/программная модель для современных гетерогенных параллельных систем

Программная модель для параллельных вычислений с необходимостью обуславливается опорным оборудованием, но тем не менее отличается от него. Операционная система обеспечивает интерфейс между ними. Параллельные операции не возникают сами по себе; вернее сказать, исходный код должен указывать на то, как распараллеливать работу, порождая процессы или потоки; выгружая данные, работу и команды на вычислительное устройство или оперируя на блоках данных одновременно. Программист должен сначала выявить параллелизацию, определить наилучший технический прием для функционирования в параллельном режиме, а затем в явной форме направить его работу безопасным, правильным и эффективным способом. Следующие ниже методы являются наиболее распространенными техническими приемами параллелизации. Далее мы подробно рассмотрим каждый из них.

- Параллелизация на основе процессов – передача сообщений.
- Параллелизация на основе потоков – совместные данные через память.
- Векторизация – несколько операций с одной командой.
- Обработка в потоковом режиме – через специализированные процессоры.

Параллелизация на основе процессов: передача сообщений

Подход на основе передачи сообщений был разработан для архитектур на основе распределенной памяти, в которых для перемещения данных между процессами используются явные сообщения. В этой модели ваше приложение создает отдельные процессы, именуемые рангами в передаче сообщений, с собственным пространством памяти и конвейером команд (рис. 1.20). На рисунке также показано, что процессы передаются в OS для размещения на процессорах. Приложение обитает в части диаграммы, помеченной как пользовательское пространство, где у пользователя есть разрешение на работу. Часть ниже – это ядерное пространство, которое защищено от опасных операций со стороны пользователя.

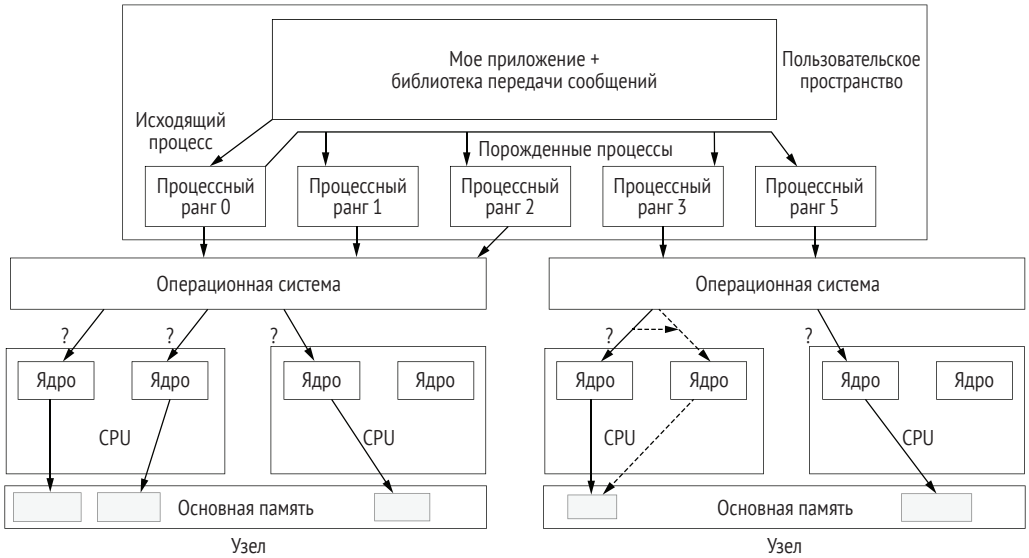


Рис. 1.20 Библиотека передачи сообщений порождает процессы. Операционная система размещает процессы на ядрах двух узлов. Вопросительные знаки указывают на то, что OS управляет размещением процессов и может перемещать их во время выполнения, как указано пунктирными стрелками. OS также выделяет память под каждый процесс из основной памяти узла

Имейте в виду, что процессоры – CPU – имеют несколько обрабатывающих ядер, которые не эквивалентны процессам. Процессы – это концепция операционной системы, а процессоры – это аппаратный компонент. Запуск любого числа порождаемых приложением процессов планируется операционной системой для обрабатывающих ядер. На самом деле вы можете запустить восемь процессов на своем четырехъядерном ноутбуке, и они будут просто переключаться между обрабатывающими ядрами. По этой причине были разработаны механизмы, указывающие операционной системе на то, как размещать процессы и следует ли «привязывать» процесс к обрабатывающему ядру. Контролирование привязки подробнее обсуждается в главе 14.

В целях перемещения данных между процессами вам нужно будет запрограммировать явные сообщения в своем приложении. Эти сообщения могут отправляться по сети либо через совместную память. В 1992 году многие библиотеки передачи сообщений образовали стандарт под названием «Интерфейс передачи сообщений» (MPI). С тех пор MPI занял эту нишу и присутствует почти во всех параллельных приложениях, масштабируемых за пределами одного узла. И – да, вы также найдете много разных имплементаций библиотек MPI.

Распределенные вычисления против параллельных вычислений

В некоторых параллельных приложениях используется более низкоуровневый подход к параллелизации, именуемый распределенными вычислениями. Мы определяем распределенные вычисления как множество

слабо сцепленных процессов, которые взаимодействуют с помощью вызовов на уровне операционной системы. Хотя распределенные вычисления являются подмножеством параллельных вычислений, это различие важно понимать. Примерами приложений для распределенных вычислений являются одноранговые сети, Всемирная паутина и интернет-почта. Поиск внеземного разума (Search for Extraterrestial Intelligence, SETI@home) является лишь одним из примеров многих научных приложений для распределенных вычислений.

Место каждого процесса обычно находится на отдельном узле и создается с помощью операционной системы с использованием чего-то вроде удаленного вызова процедур (RPC) либо сетевого протокола. Затем процессы обмениваются данными посредством передачи сообщений между процессами посредством межпроцессного взаимодействия (IPC), которого существует несколько разновидностей. В простых параллельных приложениях часто используется подход на основе распределенных вычислений, но нередко с помощью языка более высокого уровня, такого как Python, и специализированных параллельных модулей или библиотек.

Параллизация на основе потоков: совместные данные через память

Подход на основе потоков (т. е. на основе threads – виртуальных ядер) к параллелизации порождает отдельные указатели команд в рамках одного и того же процесса (рис. 1.21). В результате вы можете легко делиться порциями процессной памяти между потоками. Но это сопровождается подвохами, связанными с правильностью и производительностью. Программисту остается определять разделы набора команд и данных, которые независимы и могут поддерживать потокообразование. Эти соображения подробнее обсуждаются в главе 7, где мы рассмотрим OpenMP, одну из ведущих систем потокообразования. OpenMP предоставляет возможность порождать потоки и распределять работу между этими потоками.

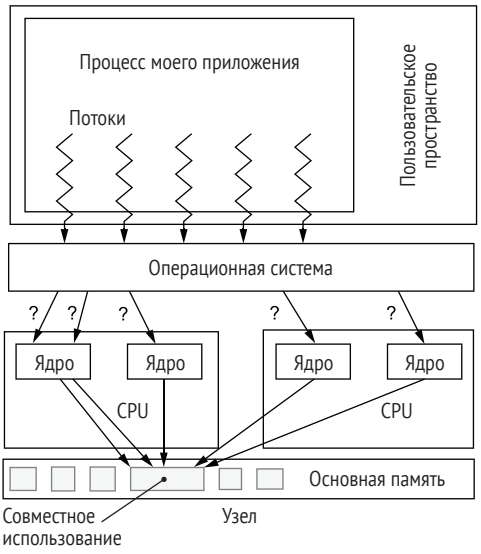


Рис. 1.21 Процесс приложения в поточном (thread-based) подходе к параллелизации порождает потоки (threads). Указанные потоки ограничены доменом узла. Вопросительные знаки показывают, что OS выбирает место размещения потоков. Некоторая память используется между потоками совместно

Существуют самые разнообразные подходы к потокообразованию, от тяжеловесных до легковесных, управляемых пользовательским пространством либо операционной системой. Хотя системы потокообразования лимитированы масштабированием в пределах одного узла, они являются привлекательным вариантом для умеренного ускорения. Однако пределы памяти одного узла имеют более серьезные последствия для приложения.

Векторизация: несколько операции с одной командой

Векторизирование приложения бывает гораздо эффективнее с точки зрения затрат, чем расширение вычислительных ресурсов в центре HPC, и этот метод бывает абсолютно необходим на портативных устройствах, таких как мобильные телефоны. При векторизировании работа выполняется блоками по 2–16 элементов данных за один раз. Более формальным термином для этой классификации операций является «Одна команда, несколько элементов данных» (single instruction, multiple data, аббр. SIMD). Термин SIMD часто используется, когда речь заходит о векторизации. SIMD – это всего лишь одна из категорий параллельных архитектур, которые будут обсуждаться далее в разделе 1.4.

Инициирование векторизации из пользовательского приложения чаще всего выполняется с помощью прагм исходного кода или с помощью компиляторного анализа. Прагмы и директивы – это подсказки, которые даются компилятору с целью сориентировать его в отношении параллелизирования или векторизирования раздела исходного кода. Как прагмы, так и компиляторный анализ сильно зависят от способностей компилятора (рис. 1.22). Здесь мы зависим от компилятора, где предыдущие параллельные механизмы зависели от операционной системы. Кроме того, без явно заданных компиляторных флагов сгенерированный код предназначен для наименее мощного процессора и длины вектора, что значительно снижает эффективность векторизации. Существуют механизмы, с помощью которых компилятор можно обойти, но они требуют гораздо больших усилий по программированию и не являются переносимыми.

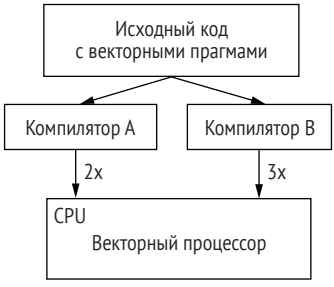


Рис. 1.22 Векторные команды в исходном коде, возвращающие разные уровни производительности из компиляторов

Обработка потоков операций посредством специализированных процессоров

Обработка в потоковом режиме (stream processing) – это концепция циркуляции данных, в которой поток данных обрабатывается более простым узкоцелевым процессором. Долгое время применявшаяся во встроенных вычислениях, эта технология была адаптирована под визуальную обработку крупных наборов геометрических объектов для компьютерных дисплеев в специализированном процессоре, GPU. Эти GPU наполнялись широким набором арифметических операций и несколькими потоковыми мультипроцессорами (SM) для обработки геометрических данных в параллельном режиме. Научные программисты вскоре нашли способы адаптировать обработку потоков данных к крупным наборам симуляционных данных, таким как ячейки, расширив роль GPU до GPGPU.

На рис. 1.23 показаны данные и вычислительное ядро, выгруженные по шине PCI в GPU для вычислений. GPU-процессоры по-прежнему лимитированы в функциональности, по сравнению с CPU, но там, где можно использовать специализированную функциональность, они обеспечивают исключительные вычислительные способности при более низких требованиях к мощности. К этой категории подходят и другие специализированные процессоры, хотя в наших обсуждениях мы сосредоточимся на GPU.

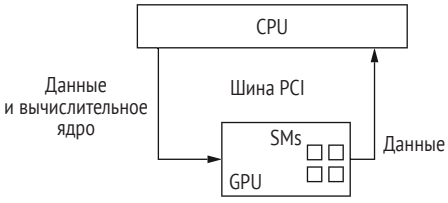


Рис. 1.23 В подходе на основе обработки потоков данных данные и вычислительное ядро выгружаются в GPU и его потоковые мультипроцессоры. Обработанные или выходные данные передаются обратно в CPU для файлового ввода-вывода или другой работы

1.4 Классифицирование параллельных подходов

Если вы прочитаете больше о параллельных вычислениях, то столкнетесь с такими аббревиатурами, как SIMD (Одна команда, несколько элементов данных) и MIMD (Несколько команд, несколько элементов данных). Эти термины относятся к категориям компьютерных архитектур, предложенных Майклом Флинном (Michael Flynn) в 1966 году, что потом стало известно как таксономия Флинна. Эти классы помогают по-разному рассматривать потенциальную параллелизацию в архитектурах. Указанная классификация основана на разбивке команд и данных на одиночные либо многочисленные операции (рис. 1.24). Имейте в виду, что, хотя эта таксономия и полезна, некоторые архитектуры и алгоритмы не очень хорошо вписываются в категорию. Ее полезность заключается в распознавании шаблонов в таких категориях, как SIMD, у которых могут иметься трудности с условными инструкциями. Это связано с тем, что каждый

элемент данных может нуждаться в другом блоке кода, но потоки должны исполнять ту же самую команду.

		Команда	
		Одиночная	Многочисленная
Данные	Одиночные	SISD Одна команда, один элемент данных	MISD Многочисленные команды, один элемент данных
	Многочисленные	SIMD Одна команда, многочисленные элементы данных	MIMD Многочисленные команды, один многочисленные элементы данных

Рис. 1.24 Таксономия Флинна классифицирует разные параллельные архитектуры. Последовательная архитектура – это один элемент данных, одна команда (SISD). Две категории имеют лишь частичную параллелизацию в том смысле, что либо команды, либо данные являются параллельными, но другая часть является последовательной

В случае, когда существует более одной последовательности команд, категория называется «Несколько команд, один элемент данных» (multiple instruction, single data, аббр. MISD). Эта архитектура не так распространена; самым лучшим примером может служить избыточное вычисление на одних и тех же данных. Оно используется в высокоустойчивых подходах, таких как контроллеры космических аппаратов. Поскольку космические аппараты находятся в условиях высокой радиации, они часто выполняют две копии каждого расчета и сравнивают результаты обоих.

Векторизация является ярким примером SIMD, в которой одна и та же команда исполняется для многочисленных элементов данных. Вариантом SIMD является «Одна команда, мультипоток» (single instruction, multi-thread, аббр. SIMT), который широко используется для описания рабочих групп GPU.

Последняя категория имеет параллелизацию как в командах, так и в данных и называется MIMD. Эта категория описывает мультиядерные параллельные архитектуры, которые составляют большинство крупных параллельных систем.

1.5 Параллельные стратегии

До сих пор в нашем первоначальном примере в разделе 1.3.1 мы рассматривали параллелизацию данных для ячеек или пикселов. Но параллелизация данных может использоваться и для частиц и других объектов данных. Параллелизация данных является наиболее распространенным подходом и часто самым простым. По сути, каждый процесс выполняет одну и ту же программу, но оперирует уникальным подмножеством

данных, как показано в правом верхнем углу рис. 1.25. Параллельный подход к обработке данных имеет то преимущество, что он хорошо масштабируется по мере того, как увеличивается размер задачи и число процессоров.

Еще один подход представлен параллелизмом на уровне операционных задач. Это включает в себя главный контроллер со стратегиями на основе потоков-работников (worker threads), конвейера или ведерной бригады, также показанными на рис. 1.25. Подход в виде турбопровода (т. е. как бы в виде трубы, по которой равномерно течет вода) используется в суперскалярных процессорах, где вычисления адресов и целых чисел выполняются отдельным логическим модулем, а не обработчиком данных с плавающей точкой, что позволяет выполнять эти вычисления в параллельном режиме. В ведерной бригаде (т. е. как бы в виде цепочки людей, передающих ведра с водой на пожаре) каждый процессор используется для обработки и преобразования данных в последовательности операций. При подходе на основе главного работника один процессор планирует и распределяет задачи для всех работников, и каждый работник проверяет наличие следующего элемента работы, возвращая предыдущую выполненную задачу. Также существует возможность комбинировать разные параллельные стратегии, чтобы выявить более высокую степень параллелизма.

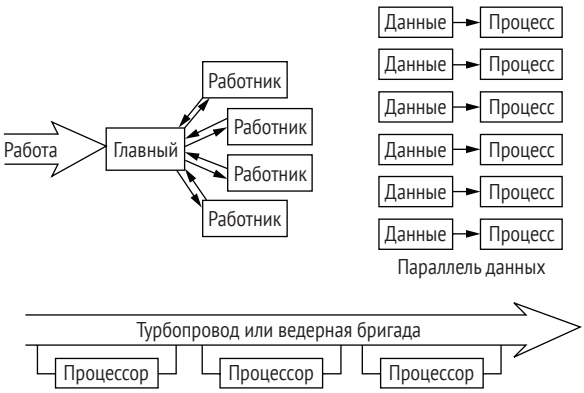


Рис. 1.25 Различные стратегии параллельности задач и данных, включая главного работника, конвейер или ведерную бригаду, и параллелизм данных

1.6 Параллельное ускорение против сравнительного ускорения: две разные меры

На протяжении всей этой книги мы будем представлять ряд сравнительных показателей производительности и ускорений. Нередко термин «ускорение» используется для сравнения двух разных времен выполнения с небольшим объяснением или контекстом в целях полного понима-

ния того, что это значит. Ускорение является общим термином, который используется во многих контекстах, например для количественного оценивания эффектов оптимизации. В целях прояснения разницы между двумя главными категориями показателей параллельной производительности мы определим два разных термина.

- Параллельное ускорение. На самом деле мы должны назвать этот термин параллельным ускорением по сравнению с последовательным. Ускорение происходит, по сравнению с базовым последовательным прогоном на стандартной платформе, обычно на одном CPU. Параллельное ускорение может обуславливаться работой на GPU либо с пакетом OpenMP, либо MPI на всех ядрах узла компьютерной системы.
- Сравнительное ускорение. На самом деле мы должны назвать этот термин сравнительным ускорением между архитектурами. Обычно это сравнение производительности между двумя параллельными имплементациями либо другое сравнение между достаточно ограниченными комплектами оборудования. Например, оно может быть между параллельной имплементацией MPI на всех ядрах узла компьютера в сравнении с GPU-процессором(ами) на узле.

Эти две категории сравнений производительности представляют две разные цели. Первая – понять, насколько можно ускорить процесс, добавив тот или иной тип параллелизма. Однако это сравнение будет необъективным между архитектурами. Речь идет о параллельном ускорении. Например, сравнение времени работы GPU с последовательным прогоном CPU не является объективным сравнением между мультиядерным CPU и GPU. Сравнительные ускорения между архитектурами более уместны при попытке сравнить мультиядерный CPU с производительностью одного или нескольких GPU на узле.

В последние годы эти две архитектуры были нормализованы, вследствие чего относительная производительность сравнивается для схожих требований к мощности или энергии, а не для произвольного узла. Тем не менее существует столь много разных архитектур и возможных комбинаций, что для обоснования вывода можно получать любые показатели производительности. Вы можете подобрать быстрый GPU и медленный CPU либо четырехъядерный CPU для сравнения с 16-ядерным процессором. Поэтому мы предлагаем вам добавлять следующие ниже термины в скобках для сравнения производительности, чтобы придавать им больший контекст.

- Добавлять «(лучшее в 2016 году)» в каждый термин. Например, параллельное ускорение (лучшее в 2016 году) и сравнительное ускорение (лучшее в 2016 году) указывают на то, что сравнение проводится между лучшим оборудованием, выпущенным в определенном году (в данном примере в 2016 году), где вы можете сравнивать высококлассный GPU с высококлассным CPU.
- Добавлять «(общедоступное в 2016 году)» или «(2016)», если две архитектуры были выпущены в 2016 году, но не являются оборудо-

ванием высшего класса. Это бывает актуально для разработчиков и пользователей, у которых больше массовых компонентов, чем в топовых системах.
- Добавлять «(Mac 2016 года)», если GPU и CPU были выпущены в ноутбуке или настольном компьютере Mac 2016 года или в чем-то подобном для других брендов с фиксированными компонентами в течение определенного периода времени (в данном примере в 2016 году). Сравнение производительности такого типа полезно для пользователей общедоступной системы.
- Добавлять «(GPU 2016:CPU 2013)», чтобы показывать, что существует возможное несоответствие в годе выпуска оборудования (в данном примере 2016 год по сравнению с 2013 годом) у сравниваемых компонентов.
- В сравнительные данные никаких квалификаций не добавляется. Кто знает, что эти цифры означают?

Из-за резкого роста моделей CPU и GPU показатели производительности обязательно будут больше похожи на сравнение яблок и апельсинов, чем на четко определенную метрику. Но для более формальных условий сравнения мы должны, по крайней мере, указывать характер сравнения, чтобы другие лучше понимали смысл цифр и чтобы быть объективнее к поставщикам оборудования.

1.7 Чему вы научитесь в этой книге?

Эта книга написана, имея в виду разработчика прикладного кода, и никаких предварительных знаний о параллельных вычислениях не предполагается. У вас просто должно быть желание повысить производительность и масштабируемость вашего приложения. Области применения включают научные вычисления, машинное обучение и анализ больших данных в системах, начиная от настольных компьютеров и заканчивая крупнейшими суперкомпьютерами.

В целях извлечения пользы из этой книги в полной мере читатели должны быть опытными программистами, предпочтительно владеющими компилируемым языком HPC, таким как C, C++ или Fortran. Мы также исходим из наличия элементарный знаний аппаратных архитектур. В дополнение к этому читатели должны быть знакомы с терминами компьютерных технологий, такими как биты, байты, операции, кеш, оперативная память и т. д. Также полезно иметь базовое понимание функций операционной системы и того, как она управляет аппаратными компонентами и взаимодействует с ними. После прочтения этой книги вы приобретете несколько навыков, в том числе:

- определение того, когда передача сообщений (MPI) более уместна, чем потокообразование (пакет OpenMP), и наоборот;
- оценивание того, насколько возможно ускорение при векторизации;

- распознавание того, какие разделы вашего приложения обладают наибольшим потенциалом для ускорения;
- принятие решения о том, когда бывает полезно использовать GPU для ускорения вашего приложения;
- установление максимальной потенциальной производительности для вашего приложения;
- оценивание энергозатрат для вашего приложения.

Даже после этой первой главы вы должны почувствовать себя комфортно с разными подходами к параллельному программированию. Мы предлагаем вам прорабатывать упражнения в каждой главе, которые помогут вам интегрировать многие вводимые нами концепции. Если вы начинаете чувствовать себя немного подавленным сложностью современных параллельных архитектур, то вы не одиноки. Сложно ухватить все возможности сразу. В следующих главах мы будем разбирать их по частям, чтобы вам было проще.

1.7.1 Дополнительное чтение

Хорошее базовое введение в параллельные вычисления можно найти на веб-сайте Национальной лаборатории Лоуренса Ливермора:

- Блейз Барни, «Введение в параллельные вычисления» (Blaise Barney, Introduction to Parallel Computing), https://computing.llnl.gov/tutorials/parallel_comp/.

1.7.2 Упражнения

1. Каковы другие примеры параллельных операций в вашей повседневной жизни? Как бы вы классифицировали свой пример? Под что, по вашему мнению, оптимизируется параллельный дизайн? Можете ли вы вычислить параллельное ускорение для этого примера?
2. Какова теоретическая мощность параллельной обработки вашей системы (будь то настольный компьютер, ноутбук или мобильный телефон) по сравнению с ее мощностью последовательной обработки? Какие виды параллельного оборудования в ней присутствуют?
3. Какие параллельные стратегии вы видите в примере с оплатой покупок в магазине на рис. 1.1? Существуют ли какие-то нынешние параллельные стратегии, которые не показаны? Как насчет примеров из упражнения 1?
4. У вас есть приложение для обработки изображений, которому необходимо ежедневно обрабатывать 1000 изображений размером 4 мебибайт (МiВ, 2^{20}, или 1 048 576 байт) каждое. Для последовательной обработки каждого изображения требуется 10 мин. Ваш кластер состоит из мультиядерных узлов с 16 ядрами и общим объемом 16 гибибайт (ГиБ, 2^{30} байт, или 1024 мебибайт) основной памяти в расчете на узел. (Обратите внимание, что мы используем правильные двоичные термины МиБ и ГиБ, а не Мб и Гб, которые являются метрическими терминами соответственно для 10^6 и 10^9 байт.)

a Какой дизайн параллельной обработки лучше всего справляется с этой рабочей нагрузкой?
b Теперь потребительский спрос увеличивается в 10 раз. Справляется ли с этим ваш дизайн? Какие изменения вам пришлось бы внести?

5 Процессор Intel Xeon E5-4660 имеет расчетную тепловую мощность 130 Вт; это средняя потребляемая мощность при использовании всех 16 ядер. GPU NVIDIA Tesla V100 и GPU AMD MI25 Radeon имеют расчетную тепловую мощность 300 Вт. Предположим, вы портируете свое программное обеспечение для использования одного из этих GPU. Насколько быстрее должно работать ваше приложение на GPU, чтобы считаться более энергоэффективным, чем ваше приложение с 16-ядерным CPU?

Резюме

- Поскольку наступила эпоха, когда большая часть вычислительных способностей оборудования доступна только через параллелизм, программисты должны хорошо разбираться в технических приемах, используемых для эксплуатирования параллелизма.
- Приложения должны иметь параллельную работу. Самая важная задача параллельного программиста заключается в выявлении большего параллелизма.
- Усовершенствования оборудования всецело касается почти тотального усовершенствоваания параллельных компонентов. Опора на повышение последовательной производительности не приведет к ускорению в будущем. Ключ к повышению производительности приложений будет лежать в параллельной сфере.
- Появляются самые разные языки параллельного программного обеспечения, которые помогают получать доступ к возможностям оборудования. Программисты должны разбираться в том, какие из них подходят для разных ситуаций.