5.1. Предпосылки к автоматизации параллельного программирования

Вопрос об автоматизации программирования встал практически одновременно с вопросом о программировании. Чтобы понять причины такого положения дел, обратим внимание на то, что результатом программирования как такового является, в конечном счете, машинный код – огромное упорядоченное множество очень маленьких и простых машинных команд. Работать с таким сложным и объемным объектом напрямую для человека невозможно, особенно если учесть, что ошибка даже в одном символе недопустима. С другой стороны, создание машинного кода – задача во многом (но далеко не во всем!) рутинная, механическая. Поэтому человек практически сразу стал привлекать к процессу конструирования программы компьютер, который с тех пор всегда является ассистентом и соавтором его программ. Наиболее рельефно это видно в таком известном инструменте из арсенала программиста, как компилятор.

Не стало исключением и параллельное программирование. Создание параллельной программы – задача существенно более сложная, чем создание программы последовательной, и обусловлено это несколькими важными факторами. Во-первых, параллельный вычислитель – система гораздо более сложная, чем последовательный компьютер. Параллельная программа должна до примитивных деталей «растолковать» каждому вычислительному ядру, входящему в его состав, что делать, как взаимодействовать с другими ядрами, как совместно бесконфликтно использовать разделяемые ресурсы (например, общую память узла), как организовать сетевое сообщение, как распределить работу равномерно по доступным ресурсам, как распределить данные по вычислительным узлам и многое другое.

Во-вторых, параллельные программы обладают **недетерминизмом** – неоднозначностью исполнения. Дело в том, что каждый поток управления в параллельной программе не синхронизирован жестко с другими потоками, поэтому заранее невозможно однозначно определить, как пойдет исполнение параллельной программы. Из-за этого, в частности, усложняется и

отладка, потому что ошибка в программе может проявляться далеко не каждый раз при ее выполнении, даже если программа выполняется на тех же самых входных данных и том же оборудовании. Типичный пример – это гонка данных (см. п. 2.1). Недетерминизм присутствует и в последовательных программах, но в параллельных масштаб этого фактора несравнимо больше.

В-третьих, организовать работу параллельного компьютера в динамике человеку сложно. Известны примеры очень простых параллельных программ, в которых и ошибке-то негде спрятаться, но тем не менее содержащих ошибку, которая проявится, как это бывает, в самый неподходящий момент.

К сказанному выше стоит добавить, что подавляющее большинство современных компьютеров общего назначения являются параллельными, и этот тренд со временем только усиливается. Даже смартфоны сегодня строятся на многоядерных процессорах. Это связано с тем, что наращивать производительность компьютеров в рамках модели последовательных вычислений с каждым годом становится все сложнее из-за физических и технологических ограничений.

В качестве иллюстрации приведем известный пример роста тактовой частоты процессора. До определенного времени тактовая частота являлась ключевым показателем производительности процессора (чем выше тактовая частота, тем больше операций в единицу времени выполняет процессор), и тактовая частота росла с середины XX века из года в год, достигнув показателя в нескольких гигагерц в первом десятилетии XXI века. Но после этого рост тактовой частоты прекратился и остается примерно тем же до сих пор. Причиной тому стало тепловыделение, которое увеличивалось вместе с тактовой частотой[1]. А раз тактовую частоту увеличивать больше не получалось, то пришлось искать другие пути увеличения производительности компьютеров. И переход к параллельным вычислениям в массовом секторе стал неизбежным выходом.

[1] Это, в свою очередь, потребовало совершенствования систем охлаждения, иначе перегрев приводит к разрушению оборудования.

Проблема с тактовой частотой была важной, но далеко не единственной предпосылкой к параллельным вычислениям.

5.2. Проблематика автоматического конструирования параллельных программ

Автоматизации поддается труд механический, рутинный, строго подчиняющийся известным правилам. При том что задача программирования (в том числе параллельного) требует, вообще говоря, творческого усилия и не имеет удовлетворительного универсального решения, потенциал к автоматизации очень велик.

Для того чтобы часть работы по созданию параллельной программы мог сделать компьютер, ему должна быть достаточно точно сформулирована задача. Для этого и нужны языки программирования высокого уровня. Известно, что не всякая задача может быть решена компьютером – вообще или достаточно хорошо. Например, даже в самых современных компиляторах отсутствует такая полезная функциональность, как проверка программы на зависания. Эта проблема, известная как проблема останова, является алгоритмически неразрешимой. Есть и алгоритмически труднорешаемые задачи, которые в принципе могут быть решены компьютером, но решение потребует слишком много ресурсов (времени, памяти и т. п.). Например, создание параллельной программы, работающей за минимальное время по ее спецификации на высокоуровневом языке. Сложность этой задачи сравнима со сложностью взлома какого-нибудь современного военного шифра.

Чтобы разобраться с тем, что и как может быть автоматизировано в параллельном программировании, разложим эту задачу на несколько уровней (рис. 5.1). Для ясности можно считать, что задать спецификацию программы – значит определить функцию, которую программа должна вычислить. Действительно, программа принимает какие-то данные на вход и какие-то выдает на выход. Получается, что программа определяет отображение входных

данных в выходные, т. е. вычисляет некоторую функцию[2]. В качестве примера можно взять функцию сортировки массива.

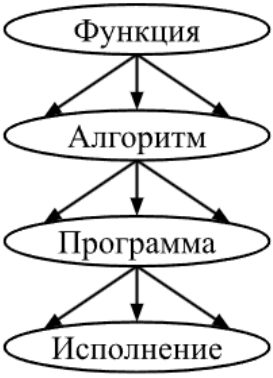

Рис 5.1. Этапы конструирования и исполнения параллельной программы

Любая вычислимая функция может быть вычислена множеством способов (счетным, вообще говоря). Например, читателю наверняка знакомы алгоритмы сортировки пузырьком, быстрой сортировки, сортировки подсчетом элементов, сортировки тасованием и пр. Множественность алгоритмов, вычисляющих заданную функцию, показана на рис. 5.1 расходящимися стрелками. Любой алгоритм, в свою очередь, может быть запрограммирован (счетным) множеством способов. Программа отличается от алгоритма наличием управления и распределения ресурсов. Например, если всем студентам одного курса дать задание запрограммировать сортировку пузырьком, то полученные программы могут отличаться временем выполнения, расходом памяти и т. п. И, наконец, программа обладает недетерминизмом исполнения, т. е. может быть исполнена различными (но правильными) способами. Например, параллельная программа быстрой сортировки массива может параллельно в разных потоках обрабатывать две половины массива, и заранее неизвестно, какой поток закончит обработку раньше.

[2] Читатель может заметить, что не всякая функция сгодится в качестве спецификации программы, а только алгоритмически вычислимая.

На каждом уровне мы видим переход от одного элемента ко многим. Именно в этой множественности скрыта проблема автоматизации программирования. И дело не столько в том, чтобы найти алгоритм, вычисляющий функцию (или программу, реализующую алгоритм), это относительно несложно. Можно даже перебрать все счетное множество алгоритмов. Настоящая проблема состоит в том, чтобы найти достаточно хороший алгоритм (программу) в этом множестве. Например, известно, что алгоритм быстрой сортировки работает быстрее, чем сортировка пузырьком на больших массивах данных, но на небольших массивах сортировка пузырьком может работать быстрее. Этот факт известен из опыта программирования, но вот вывести это автоматически проблематично.

Переход на каждый уровень вниз от функции к эффективному исполнению программы является алгоритмически труднорешаемой задачей. В практике автоматизации параллельного программирования это означает, что языки и системы конструирования параллельных программ не берутся за общую задачу, а решают ее в частном, урезанном виде. Например, берутся за конструирование параллельных программ только для вычислений в области матрично-векторных операций на системах с общей памятью. Поэтому, в частности, существует (и будет существовать в будущем) множество разных языков и систем параллельного программирования, ориентированных на разные классы приложений.

5.3. О повышении уровня программирования

Под повышением уровня программирования обычно подразумевается повышение уровня абстракций, используемых при написании программ. «Единственно эффективное направление работы со сложными системами основывается на иерархическом подходе. Динамическая система составляется и осмысливается в терминах понятий высокого уровня, которые в свою очередь составляются и осмысливаются в терминах понятий более низкого уровня и т. д. Этот подход должен быть отражен в структуре программы, определяющей динамическую систему; в том или

другом аспекте понятия более высокого уровня будут соответствовать программным компонентам» [12].

Вообще говоря, программирование в абстракциях низкого уровня является вынужденной мерой. Программисты предпочли бы писать программы, состоящие из одного оператора вида «реши вот такую задачу». Или, говоря по-другому, программисты хотели бы, чтобы программа решения задачи генерировалась автоматически по спецификации задачи (например, по описанию функции). Из параграфа 5.2 ясно, почему это недостижимо на практике. Поэтому программистам приходится самим реализовывать алгоритмы, т. е. оперировать не терминами уровня постановки задачи, а терминами на уровне элементов алгоритма, понижая уровень абстракции. Таких переходов с понижением уровня абстракции может быть сделано несколько, например, если элементарное действие в терминах одного алгоритма не имеет готовой реализации и должно быть раскрыто другим алгоритмом.

Другая задача, решаемая при программировании, – это запись алгоритма в тех терминах, которые вычислительная система способна «понимать». Когда-то, на заре развития вычислительной техники, единственным уровнем абстракции, который «понимал» компьютер, был уровень машинных команд. Затем, с развитием методов и средств трансляции, уровень средств программирования повышался, появились так называемые «языки программирования высокого уровня». Такие языки предоставляют набор базовых средств, которых достаточно, чтобы из них конструировать многие полезные программы. Кроме того, языки программирования обычно позволяют формировать абстрактные понятия еще более высокого уровня, используя языковые средства типа подпрограмм, классов и т. п. Сконструированные таким образом абстрактные понятия можно объединять в библиотеки, которые позволяют еще повысить уровень программирования в той или иной предметной области.

Повышение уровня программирования имеет два положительных следствия. Во-первых, повышается удобство программирования. Используются более емкие по смыслу операторы, соответственно программы становятся короче. При удачном выборе операторов высокого уровня, например, в

соответствии с терминами актуальной предметной области, программы становятся более понятными, их проще писать, читать, отлаживать и модифицировать. Кроме того, уменьшается вероятность при написании программы совершить ошибку.

Во-вторых, в среднем повышается качество программ. Так как программа пишется в более емких терминах, то остается свобода выбора в том, как соответствующие операторы будут реализованы на низком уровне. С одной стороны, выбор реализации может сильно влиять на нефункциональные свойства программы, характеризующие ее качество, такие как производительность, эффективность использования ресурсов, устойчивость к сбоям и другие. Поэтому важно выбрать «хорошую» реализацию. С другой стороны, выбор «хорошей» реализации сложен, так как зависит от многих факторов, таких как характеристики используемого оборудования и программного окружения, типичные входные данные, загрузка вычислительной системы другими программами и т. п. (см. предыдущий параграф). Умелый программист, конечно, учтет все эти факторы и напишет низкоуровневую программу, которая в ряде ситуаций будет работать эффективно. Но если существуют средства, позволяющие делать выбор реализации высокоуровневых конструкций автоматически (например, оптимизирующий компилятор), то даже менее умелый среднестатистический программист сможет получать программы достаточно хорошего качества.

Современные компиляторы и библиотеки в целом закрывают проблему эффективного отображения абстракций уровня алгоритма на машинные команды, что позволяет большинству программистов заниматься, по сути, записью алгоритмов, а не машинным кодированием. Но так обстоят дела только в области последовательного программирования, когда у нас есть единственный вычислитель[3]. В случае параллельной вычислительной системы появляется задача распределения данных

[3] Да и то лишь для программ, относительно детально описывающих управление и распределение ресурсов. Например, программы на языке Python в массе работают гораздо медленнее, чем аналогичные программы на C++.

и операций алгоритма по имеющимся ресурсам, а также связанная с ней задача задания порядка вычислений. Даже в статической постановке эти задачи являются алгоритмически труднорешаемыми. В случае же заранее неизвестных или динамически меняющихся условий, что является типичной ситуацией в параллельном программировании, автоматически решить эти задачи не представляется возможным. Поэтому в области параллельного программирования до сих пор распространено ручное кодирование алгоритмов на уровне машинных абстракций, таких как ядра, процессы, сообщения, блокировки и т. п.

5.4. Обзор высокоуровневых средств параллельного программирования

Многие параллельные вычислители являются на самом деле множеством связанных некоторым образом последовательных вычислителей (например, процессоры, связанные через общую память или сеть). Параллельную программу для них можно рассматривать как множество взаимодействующих последовательных программ. Наиболее низким уровнем представления параллельной программы (если не принимать во внимание уровень представления составляющих ее последовательных программ) будет такой, при котором взаимодействие между последовательными программами задано явно в терминах соответствующего оборудования. Например, низкоуровневые параллельные программы для систем с общей памятью будут явно содержать команды обращения к общей памяти и средства их упорядочения, для систем с распределенной памятью – команды передачи и приема сообщений по сети. Порождение и завершение процессов на низком уровне осуществляется с помощью специальных системных вызовов уровня операционной системы.

Наиболее очевидный способ повышения уровня параллельного программирования – это повышение уровня средств, использующихся для описания взаимодействия последовательных программ. Например, для взаимодействия процессов над общей памятью были в разное время предложены такие средства, как

критические секции, события и операторы ожидания, futures и promises, потокобезопасные абстрактные типы данных (мониторы) и др. Для управления порождением и завершением параллельных процессов были предложены, например, параллельные секции, явно заданные задачи, порождающие отдельный процесс. Для систем с распределенной памятью были предложены такие абстракции, как каналы, коллективные коммуникационные операции, удаленный вызов подпрограмм (RPC), совмещающий передачу сообщений с порождением процесса, и др. Хороший экскурс в историю развития средств и концепций параллельного программирования дан в работе [53].

В основе систем автоматизации конструирования параллельных программ лежит некоторая модель вычислений, в терминах которой пользователь описывает прикладной (численный) алгоритм. От этой модели вычислений зависит, какие требуемые нефункциональные свойства конструируемой программы потенциально могут быть обеспечены автоматически, а от алгоритмов системы зависит, насколько этот потенциал реализуется. Модель вычислений и системные алгоритмы, таким образом, определяют область применения системы. Рассмотрим существующие языки, системы и средства параллельного программирования по основным их классам на примере характерных представителей с точки зрения автоматизации конструирования эффективных параллельных программ численного моделирования на мультикомпьютерах.

Специализированные модели вычислений

Подходы к автоматизации конструирования параллельных программ на основе частных моделей вычислений, например, MapReduce [47], Hadoop [74], Anthill [50], обеспечивают хорошие результаты как по производительности, так и по другим свойствам (отказоустойчивость, динамическая балансировка загрузки и т. п.), но имеют ограниченную область применения ввиду неуниверсальности модели вычислений. Эти ограничения не могут быть преодолены в полной мере путем развития системных алгоритмов без изменения модели вычислений, лежащей в основе системы.

Распределенная общая память

В рамках систем с общей распределенной памятью (PGAS [31] – partitioned global address space, распределенное глобальное адресное пространство), таких как Titanium [77], Unified Parallel C [45], Co-Array Fortran [46], X10 [72], HPX [57] и пр., разработчики создают иллюзию общей памяти на системах с распределенной памятью. При этом неизбежно возникает неоднородность памяти с точки зрения времени доступа к ней и пропускной способности. Игнорирование этого факта со стороны пользователей системы приводит к неэффективным программам, что практически не может быть преодолено со стороны системы ввиду сложности этой задачи. Как следствие, пользователям приходится учитывать фактическую неоднородность памяти в программах, что снижает их переносимость. Разработчики систем применяют частные и эвристические подходы, обеспечивающие удовлетворительное качество конструируемых программ, что определяет область практического применения систем (их «нишу»).

Исполнительные системы и параллелизм задач

В подходах, основанных на параллелизме задач (task-based), декомпозиция данных и вычислений осуществляется пользователем, а планирование вычислений и распределение данных и вычислений по узлам осуществляется автоматически. В Иллинойском университете еще с 1980-х годов развивается исполнительная система Charm++ [58] для исполнения параллельных программ, представленных в виде множества мигрирующих взаимодействующих объектов (L. Kale et al.), что позволяет автоматически обеспечивать динамическую балансировку нагрузки на узлы и другие свойства программы. Подобный подход используется и в Т-системе [1, 2].

Управляемое исполнение

Общей слабостью большинства подходов является использование такого представления прикладного алгоритма, эффективное исполнение которого обеспечивается частными и

эвристическими подходами лишь на ограниченном круге задач, который и составляет область практического применения системы. Из этого правила есть по меньшей мере три исключения, описанных ниже. В этих системах, помимо собственно представления прикладного алгоритма, имеются отдельные средства, позволяющие описать, как прикладной алгоритм следует исполнять. Это позволяет явно разделить функциональную отладку программы и обеспечение ее нефункциональных свойств. В частности, над повышением эффективности программы может работать отдельный специалист в области системного параллельного программирования без риска внести функциональную ошибку в программу.

Так, в университете Теннесси для поддержки библиотеки матрично-векторных операций DPLASMA [42] развивается подход (J. Dongarra et al.) к конструированию и исполнению параллельных программ на основе машинно-независимого представления прикладного алгоритма в виде бесконтурного ориентированного графа (Directed Acyclic Graph, DAG), что позволяет обеспечивать высокопроизводительную реализацию представленных прикладных алгоритмов благодаря планированию вычислений и динамической поддержке исполнительной системы (PaRSEC [43]).

В Стэндфордском университете (A. Aiken) разрабатывается подход, при котором отдельно описывается прикладной алгоритм и способ его отображения на вычислительные ресурсы. Коллективом разработана система Legion [41], реализующая этот подход. Недостатком системы является отсутствие автоматизации в конструировании способа отображения прикладного алгоритма на вычислительные ресурсы.

Во фреймворке AllScale [56] явно разделяются уровни описания вычислительной части в терминах предметной области, уровень описания параллелизма и локальности данных, а также системный уровень отображения данных и вычислений на аппаратные ресурсы.

Системы автоматизированного распараллеливания последовательного кода

В системах HPF (K. Kennedy) [65], DVM/САПФОР (В. А. Крюков) [7,59], XcallableMP [62] исследуются подходы к

автоматизации конструирования параллельных программ из последовательных путем статического анализа последовательного кода, возможно, аннотированного. Подход позволяет относительно малыми усилиями пользователя получать параллельный код из последовательного, но возникающие при этом проблемы выявления параллелизма в последовательном коде, декомпозиции данных и вычислений, планирования вычислений и распределения нагрузки существенно ограничивают область практического применения системы, так как являются алгоритмически труднорешаемыми. Частично это преодолевается аннотированием кода.

Высокоуровневые языки и системы

Существуют языки и системы, которые ставят задачу автоматического конструирования параллельной программы по высокоуровневому описанию алгоритма в общем виде. Такие системы характеризуются ресурсно независимым описанием алгоритма или схемы вычислений в терминах отдельных переменных (целых и вещественных чисел) и операций (сложения, умножения и т. п.), математических уравнений, систем рекуррентных соотношений и т. п. К таким работам относятся языки Пифагор [15], Sisal [14], Cilk [51], Model [70].

Также сюда можно отнести «чистые» функциональные языки, такие как Haskell [18, 49]. Показательно, что такие языки и системы обычно не выходят за пределы вычислителей с общей памятью, так как конструирование эффективной распределенной программы по такому описанию алгоритма слишком сложно.

Итоги

Обзор существующих языков и систем показывает, что разнообразие подходов действительно велико, количество усилий, прикладываемых человечеством к автоматизации программирования, велико, а проблема далеко не закрыта. Это в полной мере соответствует обстоятельствам, изложенным в параграфе 5.2. Имеется множество частных подходов, обеспечивающих приемлемое по качеству конструирование параллельных программ в отдельных предметных областях, но

доминирующим в научном компьютерном моделировании до сих пор является низкоуровневое программирование с использованием различных коммуникационных библиотек. Это означает, что высокоуровневые средства программирования еще далеки от удовлетворения потребностей пользователей суперЭВМ.

5.5. Знакомство с технологией фрагментированного программирования и системой LuNA

Целью данного параграфа является знакомство с одной из систем автоматического конструирования параллельных программ численного моделирования, развиваемой в ИВМиМГ СО РАН – системой LuNA[4]. Система LuNA базируется на теории структурного синтеза параллельных программ на вычислительных моделях [8] и концепции активных знаний [64].

Термины и определения

В основе системы LuNA лежит модель вычислений, основанная на задачах (task-based). Вычислительный процесс описывается в крупноблочно-параллельной форме в виде множества информационно зависимых задач, называемых фрагментами вычислений (ФВ). Каждый ФВ – это последовательный вычислительный процесс, задаваемый последовательной процедурой на языке C++. Эта процедура не должна иметь побочных эффектов. Для каждого ФВ явно указываются его входные и выходные аргументы, называемые фрагментами данных (ФД). ФД может быть вычисленным или невычисленным, и если он был вычислен, то его значение уже не изменяется. Процесс вычислений состоит в исполнении ФВ по мере готовности (вычисленности) их входных ФД, и в результате исполнения ФВ его выходные ФД получают свои значения (становятся вычисленными). Собственно LuNA-программа представляет собой описание множества ФВ и ФД.

Исполнение LuNA-программы осуществляет система LuNA, которая сначала транслирует исходный код в исполняемое

[4] **LuNA** — **L**ang**u**age for **N**umerical **A**lgorithms, язык для численных алгоритмов.

представление, а затем исполняет его на мультикомпьютере с помощью распределенной исполнительной системы. При этом отдельные ФВ могут выполняться параллельно (в многопоточном режиме или на разных узлах мультикомпьютера), если это не противоречит информационным зависимостям между ФВ. Система LuNA определяет, на каких узлах исполнять ФВ, передает значения ФД по сети в случае необходимости, осуществляет сборку мусора и выполняет ряд других системных задач. Таким образом, программист освобождается от этой работы, и его роль сводится лишь к описанию множества ФВ и ФД и предоставлению последовательных процедур, реализующих ФВ. Параллельное программирование как таковое отсутствует, что и является основным преимуществом системы LuNA. Ценой за такое упрощение программирования является снижение эффективности конструируемой программы. И хотя при разработке параллельной программы вручную в принципе достижима более высокая эффективность, но на практике средства автоматического конструирования программ по мере своего развития обгоняют ручное программирование так, как это произошло с компиляторами в последовательном программировании.

Язык LuNA

Сначала рассмотрим на примерах основные средства языка LuNA. В листинге 5.1 представлен простой программы типа «Hello, World!».

Листинг 5.1. LuNA-программа «Hello World!»

```
01 C++ sub print_hello(string who) ${
02   printf("Hello, %s!\n", who);
03 $}
04 sub main() {
05   print_hello("World");
06 }
```

Единственная фрагментированная подпрограмма `main` (строки 4–6) содержит описание единственного фрагмента вычислений. Тело этого фрагмента определено в строках 1–3 на языке C++. Аргументом служит строка `"World"`, которая и выводится на экран

с приветственным сообщением. Синтаксически программа похожа на программу на языке C++, но это лишь внешнее сходство. Принципиальное отличие состоит в том, что LuNA-программа *перечисляет* множество фрагментов, но не предписывает порядка их выполнения. Реальный порядок выполнения фрагментов выбирает система исходя из собственных соображений, но с учетом информационных зависимостей между фрагментами. Так, например, в программе на листинге 5.2 первым обязательно исполнится фрагмент `b`, а затем фрагменты `a` и `c` в произвольном порядке. В том числе, `a` и `c` могут исполниться параллельно на разных ядрах или узлах мультикомпьютера. При этом их порядок следования в LuNA-программе значения не имеет.

Листинг 5.2. LuNA-программа «Hello World!»

```
01 C++ sub init(name what) ${
02   what = 10;
03 $}
04 C++ sub show(string message, int value) ${
05   printf("%s = %d\n", message, value);
06 $}
07 sub main() {
08   df x;
09   cf a: show("first", x);
10   cf b: init(x);
11   cf c: show("second", x);
12 }
```

В этом примере (листинг 5.2) имеется две C++-процедуры, с помощью которых определяется три фрагмента вычислений. В терминологии системы LuNA такая C++-процедура называется фрагментом кода[5] (ФК). ФК init устанавливает значение параметра в целое число 10, а вторая выводит на экран значение целого числа

[5] Не следует путать фрагмент кода и фрагмент вычислений. Фрагмент кода находится в таком же соотношении к фрагменту вычислений, как в языке C++ соотносятся понятия функции и вызова функции. То есть фрагмент вычислений — это применение фрагмента кода к конкретным аргументам.

вместе со строковым значением. Единственный фрагмент данных `x` объявлен в начале подпрограммы `main`.

В примерах выше описание C++-процедур включалось в текст LuNA-программы, а маркеры `${...$}` использовались для обозначения секции C++-кода. Если C++-процедура определена в отдельном cpp-файле, то для ее использования необходимо ее объявить:

```
import proc_name(value, name) as code_name;
```

В скобках перечислены типы аргументов, которые могут принимать значения `int`, `real`, `string`, `value`, `name`, соответствующие C++-типам `int`, `double`, `const char*`, `const DF &` и `DF &`. Класс `DF` является системным классом, представляющим фрагмент данных. Количество параметров может быть любым конечным (0 и более). Далее, после терминала `as` может следовать синоним процедуры. Это имя будет использоваться в LuNA-программе для обозначения этой процедуры, в то время как `proc_name` будет идентифицировать C++-процедуру во внешнем файле.

Множество имен, используемых в блоке, описывается следующим образом:

```
df x, N, some_df;
```

Тут за терминалом `df` следует перечень имен через запятую. Все имена ФД должны быть перечислены таким образом. При этом отдельный ФД может быть обозначен как именем целиком, так и именем с индексами, например `x[10]` или `x[20+y][10]`. Использование индексированных имен (называемых также ссылками) является характерной чертой языка LuNA и похоже на массивы в традиционных языках, но лишь внешне. Различие состоит в том, что наличие ФД с индексированным именем не обязательно подразумевает наличие других ФД с другими значениями индексов. Можно рассматривать это как безразмерные «массивы с дырами».

Оператор исполнения описывается следующим образом:

```
cf a: oper(x, y[N], z[y[N]]);
```

Тут за нетерминалом `oper`, идентифицирующим ФК, следует перечень аргументов (ФД, выражений, констант) ФВ в порядке,

соответствующем порядку параметров ФК в его определении. Метка ФВ (`cf a:`) является опциональной.

Оператор арифметического цикла описывается следующим образом:

```
for i=x..y {...}
```

Тут за терминалом `for` следует нетерминал `i`, определяющий имя индексной переменной (счетчик цикла), `x` и `y` – ссылки, определяющие первое и последнее значения индексной переменной, а фигурные скобки содержат описание множества операторов, составляющих тело цикла.

Цикл в языке LuNA также отличается от циклов в процедурных языках, таких как C++. Различие состоит в том, что итерации цикла не упорядочены последовательно относительно друг друга. Вместо этого цикл задает *множество* копий своих тел (содержимого фигурных скобок), где счетчик цикла принимает различные значения. То есть цикл описывает множество, а не предписывает порядок. То же самое справедливо и для цикла с предусловием:

```
while y[i] i=x ..out N {...}
```

Тут за терминалом **`while`** следует предусловие, после чего нетерминалом задается индексная переменная и начальное значение индексной переменной. Далее следует терминал `out,` за которым следует ссылка на выходной ФД, куда будет записано первое значение индексной переменной, для которого условие оказалось ложным. Завершает описание блок – множество операторов, составляющих тело цикла и обрамленных в фигурные скобки.

И хотя цикл `while` вычисляет значение всех предусловий последовательно, но вычисление предусловия для следующего значения счетчика цикла вовсе не подразумевает, что тело на предыдущей итерации будет исполнено перед этим. Поэтому цикл `while` может породить сразу большое множество своих тел, например, если вычисление предусловия на очередной итерации не зависит от того, что вычисляется в теле предыдущей. Также особенностью является наличие явного выходного аргумента этого оператора – ссылка, обозначенная терминалом `out`. Так можно задать и использовать в дальнейших вычислениях фактическое количество итераций, которое отработал этот оператор.

В качестве иллюстрации рассмотрим пример LuNA-программы для вычисления числа Фибоначчи с заданным номером `n` (листинг 5.3).

Листинг 5.3. LuNA-программа вычисления числа Фибоначчи.

```
01 import init(int expr, name val);
02 sub main() {
03   df F, n;
04    init(F[0], 1);
05    init(F[1], 1);
06   for i=2..n init(F[i-1]+F[i-2], F[i]);
07 }
```

Рассмотрим дополнительные средства языка LuNA.

Комментарии. В целом язык LuNA имеет C-подобный синтаксис (фигурные скобки, операторы `for`/`while`, точки с запятой как разделители операторов, и т. п.), хотя его семантика существенно отличается. В частности, в язык были добавлены C-подобные комментарии – концевые и многострочные. Также были добавлены однословные комментарии, полезные для коротких ремарок. Слово, начинающееся с символа ` (обратный апостроф) игнорируется (так называемые «однословные комментарии», не имеющие аналогов в большинстве языков, но оказавшиеся полезными на практике):

```
df N; // N denotes the problem size
init(`in m, `out N); /* Note: `in and `out
    are one-word comments */
```

Директивы препроцессора. Введена возможность определения макросов, похожих на макросы препроцессора C, но с некоторыми отличиями. Директива определяется так:

```
#define greet(arg1, arg2) Hello, $arg1 \
      and $arg2.
#define N 10
#define FLAG
```

Отличие состоит в том, что в теле макроса параметры подставляются по именам параметра, предваренным символом `$`.

Символ \ в конце строки позволяет определять многострочные макросы. Предусматриваются также условные подстановки вида:

```
#ifdef <macro_name><body1> [#else <body2>] #endif
```

и

```
#ifndef <macro_name><body1> [#else <body2>] #endif
```

Применение макросов также использует символ $:

```
$N // gives 10
$greet(John, Marry) // gives: Hello, John
//  and  Marry.
```

Включение. Для поддержки модульности на уровне файлов добавлена возможность включения одних файлов в другие с помощью механизма, повторяющего механизм препроцессора C++ – директиву #include:

```
#include "path/to/file"
```

Она включает содержимое файла по указанному в кавычках пути в текущее место.

Внешние блоки (foreign blocks). Концепция внешнего блока подразумевает вставку в исходный код на языке LuNA фрагмента кода на некотором другом языке. Благодаря этому механизму возможно прямо в теле LuNA-программы определять процедуры, реализующие ФК. В связи с тем, что внешний блок может содержать, вообще говоря, произвольные последовательности символов, маркер окончания блока является параметром конструкции, что позволяет гибким образом избежать коллизий:

```
${<END}some textEND
```

Тут последовательность символов ${<...} является маркером начала внешнего блока, а вместо многоточия указывается произвольная последовательность символов, кроме символа }, которая должна считаться маркером конца блока. В примере выше это три символа END. Телом внешнего блока, соответственно, будет являться последовательность из 9 символов *«some text»*.

Базовые типы данных и выражения. Для удобства введены базовые типы данных – целочисленный (int), вещественный (real)

и строковой (`string`). Принадлежность ФД к тому или иному типу означает, что его значение является, соответственно, целым числом, вещественным числом или строкой. В противном случае ФД считается «черным ящиком», значение которого используется только внутри ФК. Те же типы характеризуют и выражения, которые могут быть использованы в качестве значений параметров. Семантика стандартных операций (`+`, `-`, `*`, `/`, `%`) соответствует их семантике в языке C. Типы данных могут быть использованы в подпрограммах в качестве входных.

Подпрограммы. Для поддержки модульности на уровне подпрограмм введена возможность определения подпрограмм. На верхнем структурном уровне исходного кода пользователь может определять ФК и подпрограммы, причем одна из подпрограмм считается головной (аналог функции `main` в языке C), и ее тело и задает множество операторов, составляющих LuNA-программу. По умолчанию головной подпрограммой считается подпрограмма с именем `main`.

Подпрограммы могут иметь параметры, причем в подпрограммах тип параметра `name` означает ссылочный тип, т. е. тип, который может быть использован как ссылка, в том числе для построения новых ссылок путем добавления в конец ссылки индексного выражения. В ФК ссылочный тип `name` означает выходной параметр, но в подпрограммах ссылка может быть использована как для обозначения входного, так и выходного параметров (в теле подпрограммы).

Логические выражения и условный оператор. Вводится условный оператор следующего вида:

```
if (cond) {...}
if (cond) {...} else {...}
```

Тут `cond` – целочисленное выражение, которое считается ложным, если его значение равно 0, и истинным в противном случае. Если условие истинно, то операторы, определенные в теле цикла, будут исполнены, а если ложно – то нет. Если определена ветка `else`, то в случае ложного условия исполнены будут ее операторы. Для работы с логическими выражениями имеются

базовые операции `&&`, `||`, `!`, аналогичные соответствующим операторам в языке C.

Рекомендации. Виды рекомендаций, их смысл и практическое применение рассматривать не будем, лишь отметим, что существует такая синтаксическая конструкция. Рекомендации нужны для того, чтобы управлять поведением LuNA-программы. При начальном знакомстве с языком и системой рекомендации можно игнорировать. Для обозначения рекомендаций используется символ `@` с последующим блоком в фигурных скобках.

Листинг 5.4. Примеры рекомендаций

```
01 sub main() {
02   ...
03   cf a: compute("some_text", N[K+L], M) @ {
04     request K, L;
05     request N[K+L];
06     req_count N: 5;
07     stealable, log;
08  };
09   ...
10 }@ {
11  locator_cyclic N: 5;
12  locator_cyclic x[i]: i/2;
13 }
```

Параметры командной строки. Предусмотрена возможность передачи аргументов запуска программы из командной строки. Для этого в головной подпрограмме (`main`) должны быть перечислены параметры базовых типов. Каждый параметр будет считан с командной строки и приведен к соответствующему базовому типу. Если количество параметров не совпадает с количеством аргументов, то программа не будет запущена, а система выдаст сообщение об ошибке:

```
sub main(string first_arg, int second arg) {...}
```

5.6. Пример разработки LuNA-программы

Рассмотрим задачу скалярного умножения двух вещественнозначных векторов. Сначала требуется выполнить

декомпозицию данных и вычислений задачи на фрагменты, чтобы выявить независимые подзадачи и тем самым дать возможность их параллельной обработки. На рис. 5.2 представлена схема скалярного умножения, представленная в виде множества фрагментов.

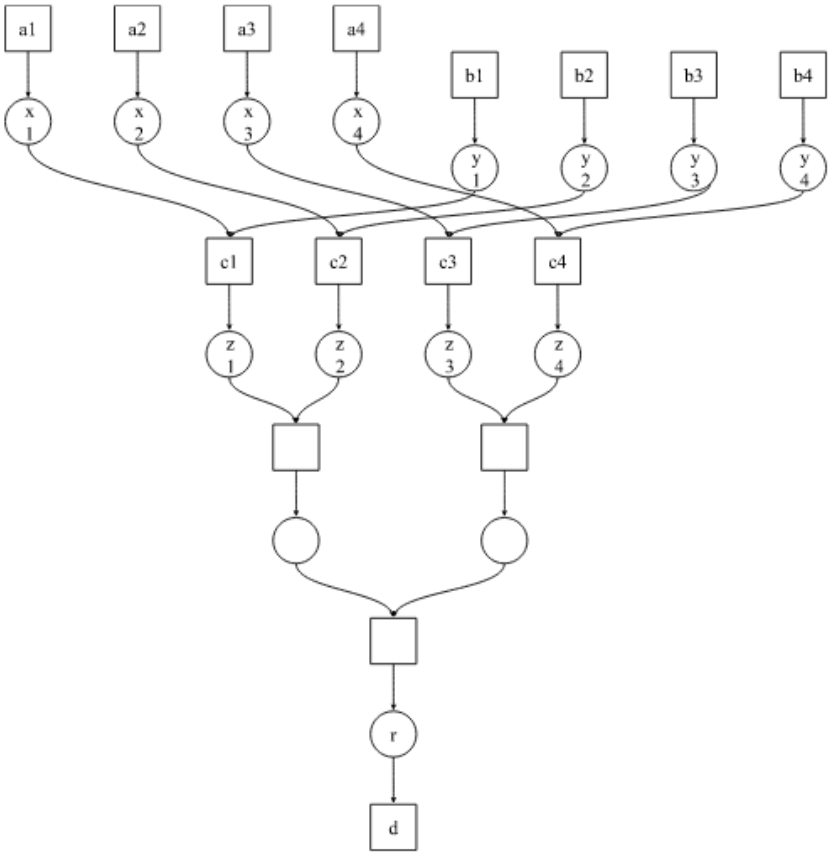

Рис. 5.2. Схема LuNA-программы скалярного умножения двух векторов

Пусть имеется два вектора размера n, которые требуется скалярно умножить. Разобьем эти вектора на m покомпонентно равных частей. Назовем соответствующие части векторов (это будут

ФД) *x1*, ..., *xm* и *y1*, ..., *ym*. Для простоты будем считать, что *n* кратно m, и что размеры всех частей равны *n/m*. Определим ФВ, инициализирующие соответствующие части векторов как *a1*, ..., *am* и *b1*, ..., *bm*. Далее вычислим квадрат скалярного произведения каждой части с помощью ФВ *c1*, ..., *cm*, а результатом будет новый набор ФД *z1*, ..., *zm*. Отметим, что в отличие от ФД *xi* и *yi* размер каждого ФД *zi* равен единице (каждый *zi* – это одиночное вещественное число). Далее просуммируем все ФД *z1*, ..., *zm* (на рис. 5.2 изображено бинарное дерево редукции) в одно число *r* – квадрат искомого скалярного произведения. ФД *r* подается на вход в ФВ *d*, который извлекает корень и выводит результат на экран.

Теперь, когда множество ФВ и ФД определено, опишем его на языке LuNA (листинг 5.5).

Листинг 5.5. LuNA-программа скалярного произведения векторов

```
01 #define VECTOR_PART_SIZE 1000
02 #define PARTS_COUNT 10

03 import init_vector_part(int vector_id,
     int part_number, int part_size, name vector) as
     init_vector_part;
04 import print_result(real n) as print_result;
05 import smulv2_part(value x, value y, name z,
     int part_size) as smulv2_part;
06 import init(name x, real val) as init;

07 sub mulv(name x, name y, name z) {
08   for i=0..$PARTS_COUNT-1
09     smulv2_part(x[i], y[i], z[i],
         $VECTOR_PART_SIZE);
10 }

11 sub reduce_sum(name arr, int begin, int end,
     name result)  {
12   if (end-begin==1) {
13     init(result, arr[begin]);
14   }
```

```
15   if (end-begin>1) {
16     df tmp1, tmp2;
17     reduce_sum(arr, begin, begin+(end-begin)/2,
         tmp1);
18     reduce_sum(arr, begin+(end-begin)/2, end, tmp2);
19     init(result, tmp1+tmp2);
20   }
21 }

22 sub main() {
23   df x, y, z, result;

24   for i=0..$PARTS_COUNT-1 {
25     init_vector_part(0, i, $VECTOR_PART_SIZE,
         x[i]);
26     init_vector_part(1, i, $VECTOR_PART_SIZE,
         y[i]);
27   }
28   print_result(result);
29   mulv(x, y, z);
30   reduce_sum(z, 0, $PARTS_COUNT, result);
31 }
```

В строках 1–2 средствами препроцессора определены константы, задающие количество частей (`m`) и размер одной части (`n/m`). Далее в 3–6 перечислены сигнатуры C++-процедур (ФК), которые будут использоваться для реализации ФВ. В строках 22–31 определена головная подпрограмма `main`. В строке 23 перечислены имена, которые будут использоваться для адресации ФД (в ссылках). Далее (24–27) следует описание множеств ФВ `a[1]`, ..., `a[m]` и `b[1]`, ..., `b[m]`, которые инициализируют значения входных векторов `x[1]`, ..., `x[m]` и `y[1]`, ..., `y[m]` соответственно. В строках 29–30 вызываются подпрограммы `mulv` и `reduce_sum` для умножения и суммирования, а в строке 28 описывается ФВ `d` для извлечения корня и вывода результата на экран. Подпрограммы `mulv` и `reduce_sum` определены выше, их смысл должен быть ясен по аналогии. Отметим только, что подпрограмма `reduce_sum`

содержит рекурсивный вызов самой себя для первой и второй половин редуцируемого массива.

Язык LuNA похож на процедурный, но лишь внешне. Операторы языка лишь описывают множества, но не предписывают последовательного исполнения. Все описанные ФВ будут исполняться не в том порядке, в котором они описаны в листинге, а в соответствии с имеющимися информационными зависимостями. Обратим внимание на то, что ФВ `d`, описанный в строке 28, будет исполняться после ФВ, описанных ниже, в строках 29–30, потому что он информационно от них зависит. Порядок следования операторов в LuNA-программе значения не имеет. Также отметим, что для ФВ `a[1]`, ..., `a[m]` и `b[1]`, ..., `b[m]` использована одна и та же C++-процедура `init_vector_part`, в которой имеется параметр `vector_id`, специфицирующий, какой из векторов требуется инициализировать.

Далее требуется предоставить C++-процедуры, перечисленные в строках 3–6 (листинг 5.6).

Листинг 5.6. Исходный C++-код

```
01 #include "ucenv.h"
02 #include <cmath>

03 extern "C" {
04 void init_vector_part(int vector_id,
     int part_number, int part_size, DF &vector) {
05   double *data = vector.create<double>(part_size);

06   for (int i=0; i<part_size; i++) {
07     if (vector_id==0)
08       data[i]=part_number * part_size + i;
09     else
10       data[i]=(part_number * part_size + i) * 0.1;
11   }
12 }

13 void print_result(double n) {
14   printf("Result: %lf\n", sqrt(n));
15 }
```

```
16 void smulv2_part(const DF &x, const DF &y, DF &z,
     int part_size) {
17   const double *part1=x.getData<double>();
18   const double *part2=y.getData<double>();

19   double result=0;

20   for (int i=0; i<part_size; i++)
21     result += part1[i] * part2[i];
22   z=result;
23 }

24 void init(DF &x, double val) {
25   x=val;
26 }
27 } // end of extern "C"
```

В строке 1 подключаются заголовки с необходимыми определениями программного окружения системы LuNA (класс *DF* и пр.) Все процедуры, которые будут применяться для реализации ФВ, должны быть отмечены как `extern "C"` (строки 3 и 27). В строках 4–12 определена процедура `init_vector_part`. Сначала (строка 5) выделяется память под результирующее значение, затем инициализируется собственно часть массива. Формула для инициализации определяется параметром *vector_id*. Аналогичным образом устроена процедура `smulv2_part` (строки 16–23), но память под результат выделяется автоматически в строке 22 как часть оператора присваивания базового типа (*double*). В строках 17–18 осуществляется получения указателя на данные входных ФД.

Далее LuNA-программа в составе двух файлов (листинги 5.5 и 5.6) передается на исполнение системе LuNA командой:

```
luna <file.fa>
```

где `<file.fa>` – это имя файла с листингом 5.5, а листинг 5.6 должен быть сохранен в той же директории с расширением `*.cpp`. После этого в консоли должен появиться результат выполнения программы.

Практические задания

1. Модифицировать программу из примера (листинги 5.5 и 5.6) так, чтобы параметром задавался не размер части вектора (***n/m***), а общий размер вектора (***n***). При этом ***n*** может быть не кратно ***m***.
2. Разработать последовательную программу, эквивалентную программе из примера, которая выполняет ту же самую работу исключительно путем вызова процедур из листинга 5.6. Последовательная программа может содержать циклы, подпрограммы и вспомогательный код, но она не должна выполнять ту работу, которая содержится в процедурах из листинга 5.6.
3. Аналогично заданию 2, но программа должна быть не последовательной, а параллельной – либо в общей памяти (многопоточная), либо в распределенной (на основе MPI или его аналогов).
4. Разработать LuNA-программу умножения матрицы на вектор. Количество фрагментов, на которые разрезается матрица и/или вектор, должно быть параметром. Можно считать размер матрицы кратным количеству фрагментов.
5. Сравнить производительность LuNA-программы на различном количестве потоков и/или процессов с производительностью программы, написанной вручную.

Примечание. Работа может выполняться на вычислителе, где установлена система LuNA. Также студент может самостоятельно установить систему LuNA на свой компьютер с помощью руководства по установке и эксплуатации системы на сайте проекта.

Вопросы для самоконтроля

1. Зачем нужны системы автоматического конструирования параллельных программ?
2. Почему не существует универсальной системы, а существуют много «нишевых», специализированных систем и языков?

3. Что означает, что система программирования имеет высокий уровень? Уровень чего?
4. Зачем нужна технология фрагментированного программирования? На что ориентирована ее модель вычислений?
5. Почему порядок выполнения фрагментов вычислений не зависит от порядка их определения в теле LuNA-программы?
6. Как добиться того, чтобы фрагменты выполнялись по порядку, если они информационно независимы друг от друга?
7. Почему язык LuNA использует в качестве строительных блоков процедуры на последовательном языке программирования?