ИПМ им.М.В.Келдыша РАН

Абрау-2017 • Труды конференции

Арк. В. Климов

Обзор подходов к созданию мульти-платформенной среды параллельного программирования

Рекомендуемая форма библиографической ссылки

Климов Арк. В. Обзор подходов к созданию мульти-платформенной среды параллельного программирования // Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции (18-23 сентября 2017 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2017. — С. 260-275. — URL: http://keldysh.ru/abrau/2017/19.pdf doi:10.20948/abrau-2017-19

Размещена также презентация к докладу

Обзор подходов к созданию мульти-платформенной среды параллельного программирования

Арк. В. Климов

Институт проблем проектирования в микроэлектронике (ИППМ) РАН

Аннотация. Трудности параллельного программирования многократно отягощаются наличием множества качественно различных платформ параллельных вычислений, для каждой из которых требуется писать программу практически заново. В мире имеется ряд проектов, адресующих эту проблему и активно развиваемых в настоящее время. Как правило, каждый из них предлагает некоторый свой язык для выражения алгоритма решения задачи в наиболее абстрактной форме с возможностью его дальнейшей трансляции в коды выбранной платформы. При этом компилятору может предоставляться дополнительная информация об отображении вычисления на ресурсы системы. В статье рассматриваются несколько таких подходов, включая авторский.

Ключевые слова: параллельное программирования, мульти-платформенные компиляторы, языки программирования, потоковая модель вычислений

Введение

После того, как в нулевые годы был исчерпан ресурс повышения производительности одного процессора за счет повышения частоты, начался стремительный рост вычислительных систем "вширь" в сторону расширения параллелизма. Соответственно, развивались и средства параллельного программирования.

Казалось, проблема будет решаться просто расширением привычных последовательных языков возможностями создания многих последовательных процессов (нитей управления) и организации их взаимодействия. И появятся единообразные подходы к расширениям разных последовательных языков параллельными средствами. Но не тут-то было. Даже для этой "простой" схемы появились различные не похожие друг на друга механизмы: OpenMP, Shmem, LINDA, Occam, MPI, не говоря о появлении многочисленных технологий, основанных уже на совсем иных и очень разных принципах: Charm++, CUDA, ZPL, GreenMarl, MapReduce. И это далеко не полный список.

Аппаратчики тоже "подбросили дров". Наиболее ярким стал пример GPU, с параллелизмом типа SIMD на массово-параллельной памяти, для которого потребовались и особые средства программирования (CUDA, OpenCL). Различия между общей и распределенной памятью также отобразились в

принципиально разные подходы к написанию программ (MPI vs OpenMP, а где-то между ними — PGAS и NUMA). Процесс экспериментирования продолжается (Cell, XEON PHI, FPGA). Не забудем и многочисленные формы языков типа dataflow и их аппаратной поддержки.

Основной вопрос, который решают разработчики всех этих платформ (языков и систем) — поиск компромисса между эффективностью и удобством программирования (продуктивностью). Чем лучше язык отражает аппаратные возможности, тем выше эффективность, но тем сложнее программировать и тем хуже обстоят дела с переносимостью, и наоборот.

В результате перед практиками встает сложная проблема: надо раз и навсегда определиться с выбором языковых и аппаратных средств и отказаться от других вариантов. А в случае необходимости сменить решение, придется тратить дополнительные силы и средства на перепрограммирование.

Хотелось бы необходимости такого выбора избежать. А для этого нужен единый универсальный язык для математического описания алгоритмов, с которого можно автоматически (или с подсказками человека) транслировать в существующие языки существующих платформ. Важно подчеркнуть, что исходное описание должно быть максимально абстрактным, чтобы не содержать деталей, относящихся к особенностям выполнения на той или иной конкретной платформе. Однако, оно должно быть выполнимым, хотя бы в принципе (быть может не эффективно). Только тогда у нас будет шанс добиться как удобства и легкости программирования, так и желаемой переносимости.

Однако, для получения хорошей эффективной программы для конкретной платформы может потребоваться дополнительная информация от человека. Будем говорить о ней как об *отображении* абстрактного алгоритма на конкретную платформу. В нем может содержаться информация о порядке вычислений, распределении вычислений по процессорным ядрам, о группировании в блоки вычислений, данных, сообщений и т.п. Важно, чтобы изменения в отображении не нарушали корректности алгоритма и его программы.

Проблемой такого языка люди озаботились давно. Предложены различные подходы и решения, которые мы и обсудим в этой статье. Важно, что мы включаем сюда только те проекты, авторы которых явно декларируют их нацеленность на данную проблему, а не просто предлагающие некий свой подход к параллельному программированию. Также могут быть включены те, которые могли бы так быть позиционированы с точки зрения автора данной статьи. При этом не утверждается, что будут рассмотрены все существующие такие проекты и подходы.

Далее каждому рассматриваемому проекту посвящается один раздел: 1 — SaC, 2 — LabVIEW, 3 — CnC, 4 — UPL(G). В заключительном разделе 5 обсуждаются критерии и проводится сравнение рассмотренных проектов.

1. SaC

SaC (Single Assignment C) — чисто функциональный язык программирования, имеющий C-подобный синтаксис. Его разработал Sven-Bodo Scholz в Кильском университете, Германия, в 1994 году [1,2]. Его целью было распространить функциональное программирование на область численных приложений. Также декларировалась задача обеспечения эффективной трансляции на различные параллельные платформы.

Область численных задач требует удобства и эффективности работы с многомерными массивами. Важным достижением языка является возможность записывать алгоритмы в не зависящей от размерности (dimension-independent) форме. Для этого введено индексирование через индекс-вектор, который сам является массивом (одномерным). Форма (shape) произвольного массива, также выражается одномерным массивом. Система типов для массивов позволяет гибко переходить от массива произвольной размерности к массивам известной размерности, но неизвестного размера по каждой размерности, и, наконец, к массивам с известными размерами по всем размерностей.

Язык SaC — это язык декларативного программирования, на нем пишутся определения именованных функций и величин (данных), вообще говоря, рекурсивные. Каждое именованная величина имеет для своей области действия единственное, статически заданное, определение. Для массивов введены специальные конструкции "одновременного" определения всех элементов with. Допускаются циклы похожие на императивные, но с соблюдением ограничений, позволяющих статически различить текущее и предыдущее значения переменных. Вычисление заключается в раскрутке всех определений, начиная с головного вызова. По классификации раздела 5 язык относится к парадигме *сбора*.

В качестве примера рассмотрим шаг алгоритма задачи N тел (All-pairs-N-body), заимствованный из работы [3] (см. рис.1).

Функция **advance** применяет параллельно (одновременно) ко всем телам функцию **acceleration**, которая вычисляет сумму всех сил, действующих на данное тело со стороны всех других тел. Этот параллелизм выражен конструкцией with. Она порождает новый массив, при этом указывается область изменения индекса (вообще говоря, многомерного) для каждого из которых тело задает элемент, a shape результата задается либо явно, как здесь, в конструкции genarray, либо указанием массива-прототипа в конструкции modarray. Внутри тела между "{" и "}" могут находиться несколько форм вида

(генератор) : выражение ;

для различных, возможно перекрывающихся, диапазонов.

Но сама функция **acceleration** здесь проходит по всем (другим) телам в простом последовательном for-цикле, вызывая одноименную функцию для каждого тела (здесь имеет место перегрузка), при этом редукция выражена обычным последовательным суммированием. Поэтому этот вариант программы недостаточно абстрактен и допускает не всякий параллелизм. Здесь, по-

видимому, предполагается только распараллеливание по with-циклу в теле функции **advance**.

```
double  sicsaL2Norm ( double [ . ] x )  {
              return  sqrt ( sum ( x ^ 2 ) + 0.0 1 ) ;
             }
double [ 3 ]  acceleration ( double [ 3 ] pos1 , double [ 3 ] pos2 , double mass )  {
   return ( pos2 – pos1 ) * mass / ( sicsaL2Norm ( pos2 – pos1 ) ^ 3 ) ;
   }
double [ 3 ]  acceleration ( double [ 3 ] pos , double [ . , . ] positions , double [ . ] masses )  {
   acc = [ 0.0 , 0.0 , 0.0 ] ;
   for ( i = 0 ; i < length ( masses ) ; i ++)  {
      acc += acceleration ( pos , positions [ i ] , masses [ i ] ) ;
      }
   return acc ;
  }
double [ . , . ] , double [ . , . ]
advance ( double [ . , . ] positions , double [ . , . ] velocities , double [ . ] masses , double dt )  {
   accelerations = with {
       ( [ 0 ] <= [ i ] < shape ( masses ) ) :
            acceleration ( positions [ i ] , positions , masses ) ;
       }: genarray ( shape ( masses ) , [ 0.0 , 0.0 , 0.0 ] ) ;
      velocities += accelerations * dt ;
      positions += velocities * dt ;
      return ( positions , velocities ) ;
   }
```

Рис. 1. Полный код шага итерации для N-BODY на SAC

К сожалению, нельзя указать тип массива с известными размерами для части размерностей. Например, здесь плохо согласуется тип формального аргумента pos2 функции **acceleration** для тела (double [3]) и тип аргумента positions[i] при ее вызове (double [.]). Хотелось бы для формальных параметров positions и velocities указать тип double[.,3].

Цели мульти-платформенности и эффективности языка SAC обеспечиваются продвинутыми техниками анализа и агрессивной оптимизации, опирающимися на чисто-функциональную семантику без побочных эффектов. На данном примере были испытаны возможности SAC-компилятора на платформах OpenMP (до 8 ядер) и GPU (CUDA) и SMP-кластер (255-поточный) из 4-х процессоров SPARC-T4. Примечательно, что конкурентная эффективность была получена путем автоматической компиляции SAC.

На сайте SAC (http://www.sac-home.org) утверждается, что компилятор sac2c обеспечивает автоматическое (из единого исходника) отображение в хорошо адаптированные (higly-tuned) коды для: многоядерных систем с общей памятью, кластеров, ускорителей на базе GPU, FPGA, а также ряд экспериментальных платформ.

2. LabVIEW

Эта широко распространенная в узких инженерных кругах система программирования предлагает писать код в графической форме вместо текстовой. Она разрабатывается с 80-х г.г. компанией National Instruments (NI) [4]. Присущая ей модель вычислений является потоковой, очень похожей на модель SAC. Язык позволяет обходится без имен, вместо которых используются стрелки, соединяющие источник с потребителем. Соединяемые функциональные элементы изображаются прямоугольными блоками, внутри которых обычно изображается пиктограмма функции.

Циклы с накоплением выражаются особыми (в виде как бы "многослойной пачки") прямоугольниками, где каждая изменяемая в цикле переменная отображена как пара "сокетов" слева и справа на одном уровне, соединенных через перевычисляющую значение схему. Толщина линий символизирует размерность передаваемого по ней массива (самая тонкая – скаляр).

Приведем в качестве примера программу вычисления моментов 1-го порядка ("центра тяжести") плоского изображения (черно-белой фотографии) размера *n*x*m* пикселов.

Его координаты (m_x,m_y) определяются формулами

$$m_x = \sum_{i=1}^{n}\sum_{j=1}^{m} iP_{ij}$$

$$m_y = \sum_{i=1}^{n}\sum_{j=1}^{m} jP_{ij}$$

В LabVIEW это вычисление можно выразить графической программой, показанной на рис.2.

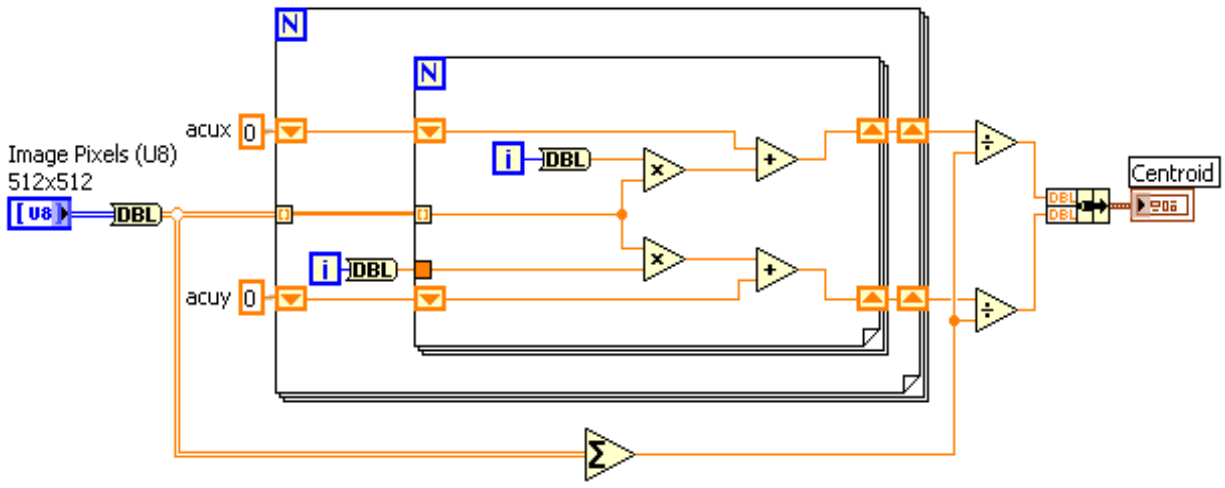


Рис.2. Программа вычисления моментов в системе LabVIEW (взято из сайта [5])

Однако, такое описание недостаточно абстрактно, поскольку оно предписывает определенный порядок вычислений. Заметим, что здесь пара вложенных друг в друга циклов, по строкам и по столбцам, то есть авторам пришлось зафиксировать, что цикл по X будет внутренним, а цикл по Y — внешним. Здесь каждый пиксел умножается (как в формулах) на i или j, а потом произведения складываются.

Семантика цикла также предполагает, что суммирование проводится рекуррентно (то есть последовательно) в порядке от меньших индексов к большим. Входной тип данных здесь – массив массивов. Толщина "проводов" отражает размерность. Предполагается разборка массива при входе в цикл, а при выходе либо сборка, либо вывод последнего значения.

Программа из такого вида легко может быть преобразована в обычный последовательный язык, например C, с сохранением предписанного порядка. Для выбора другого, более эффективного в данной аппаратной среде порядка, компилятору должны быть доступны глубокие эквивалентные преобразования. В принципе данный язык позволяет их в достаточное мере, поскольку он функциональный (без побочных эффектов) и потоковый.

3. CnC

Concurrent Collections (CnC) — это модель параллельного программирования, основанная на потоковой (dataflow) модели вычислений, в которой вычислительные действия активируются готовностью аргументов.

Проект развивается в основном группой университетов США (в первую очередь Rice Univercity, Houston, Texas), а также Intel Corporation. Начальный задел в виде модели вычислений TStreams был произведен в HP Labs [6].

В CnC вычислительные действия (steps) и данные (items) организованы в *коллекции*, индексируемые *тегами* (аналогично многомерным массивам, с тем отличием, что границы значений индексов заранее не фиксируются и их множества могут быть разреженными).

Программа задается статически как конечная совокупность коллекций и отношений между ними. Коллекциям действий приписаны программы действий (step code). В процессе работы всей программы порождаются динамические экземпляры элементов коллекций.

Элементы коллекций записываются указанием имени коллекции и тега (списка индексов) в надлежащих скобках:

- действия — (A: i, j),
- данные — [D: iter, row, col],
- управления — <T: p, q>.

При этом на месте индекса может стоять любое выражение (дискретного типа).

Между коллекциями действий и данных задаются отношения порождения и потребления. Входные данные потребляются действием, а выходные порождаются им. В CnC предлагается задавать сначала domain specification —

граф отношений между коллекциями, где указывается, какие коллекции от каких зависят в принципе. А конкретные виды зависимостей в терминах тегов задаются отдельно, также как и программы действий.

Коллекция действий аналогична процедуре, а экземпляр – вызову процедуры с конкретными аргументами. Основным аргументом является тег, и на его основе могут быть вычислены теги всех его входных и выходных данных (согласно заданным отношениям). Для активации действий предусмотрен еще один сорт коллекций — управления (controls). В программе каждая коллекция действий статически привязана к одной из коллекций управлений, имеющей тег того же типа. Экземпляр действия становится активным (enabled), когда в соответствующую коллекцию управлений вносится элемент с конкретным тегом, который и становится тегом для привязанного к ней действия (действий). После активации действие вычисляет теги своих входных данных и запрашивает их из соответствующих коллекций. Если какие-либо экземпляры входных данных отсутствуют, то экземпляр действия задерживается до тех пор, пока все необходимые данные не появятся. Если все запрошенные данные присутствуют, и только в этом случае, действие выполняется (согласно заданной программе действия), в результате чего могут быть добавлены новые экземпляры в выходные коллекции данных. Кроме того, действие может породить один или несколько новых экземпляров управлений. Весь этот процесс выполняется в среде (environment), которая создает начальные экземпляры данных и управлений и запрашивает вычисленные данные.

Каждый экземпляр коллекции может быть создан не более одного раза. Всякая попытка создать экземпляр (данных или управления) повторно считается ошибкой. Это (вместе с некоторыми другими правилами) гарантирует детерминизм, который считается важным достоинством модели.

Основная мотивировка модели CnC состоит в том, чтобы устранить излишнее навязывание порядка вычислений, как в обычных последовательных или параллельных языках. Иногда эти ограничения связаны с желанием использовать одну и ту же память повторно для других значений. В CnC задаются только необходимые ограничения на порядок, когда по смыслу алгоритма некоторые вычисления должны выполняться раньше других. Не задается, какие операции будут (могут быть) реально параллельны. Проблемный программист избавлен от необходимости принимать лишние решения по этим вопросам.

Рассмотрим, например, рассмотрим условное выполнение. Есть два разных вопроса: выполнять ли некоторый блок B и когда его выполнять. Обычно они объединены в один условный оператор **`if P then B`**, который вынуждает выполнять блок сразу после решения о том, что его нужно выполнять. В CnC блок может быть выполнен в любое более позднее время.

В CnC параллелизм неявный. Задаются лишь ограничения на порядок. Блок B2 должен выполняться строго после блока B1 при наличии хотя бы одного из следующих двух условий:

- Блок B1 производит некоторое значение, которое блок B2 использует (потребляет).
- Блок B1 предписывает выполнить блок B2 в будущем путем порождения тега для B2.

Вторая мотивировка модели CnC (связанная с первой) состоит в разделении ответственности между двумя типами специалистов: специалистом по предметной области (domain expert, предметник) и специалистом по (параллельному и эффективному) выполнению (tuning expert, настройщик).

Задача предметника — содержание алгоритма, его семантическая корректность, зависимости между частями. Он пишет и отлаживает программу так, чтобы она выполнялась функционально правильно, оставляя максимум свободы для настройщика. Задача настройщика — распределение вычислений и данных по процессорам, упорядочение вычислений внутри процессора, параллелизм, локальность, коммуникации, балансировка, размер гранул. Однако, некоторые вопросы входят в сферу обоих экспертов, например, о том, какие данные и когда можно стереть, чтобы переиспользовать память.

Важным требованием является детерминизм. В модели CnC он обеспечивается:

- единственностью присваивания (при повторном определении элемента коллекции с тем же тегом выдается ошибка) и
- недоступностью признака готовности данного (нельзя в рамках программы действия принять решение что-то выполнить на том основании, что какого-то элемента данных еще нет).

Достаточное полное описание CnC и его мотивировок можно найти в [7].

Семантика CnC предполагает, что состояния элементов коллекций в своем жизненном цикле изменяются монотонно и однократно по следующей схеме:

- **данные и управления (теги)**: отсутствует ⟶ готово.
- **действие**: отсутствует ⟶ предписано (создан тег) ⟶ активно (все входные данные готовы, можно выполнять) ⟶ выполнено (программа действия завершена).

Рассмотрим пример: факторизация Холецкого. Это разложение квадратной симметричной положительно определенной матрица A в произведение: $A = L \times L^T$, где L – нижнетреугольная матрица, элементы которой могут быть вычислены по следующим рекуррентным формулам:

$$
\begin{aligned}
&U_{1ij} = A_{ij}, && \text{для } 1 \le j \le i \le N \\
&U_{(k+1)ij} = U_{kij} - L_{ik} L_{jk}, && \text{для } 1 \le k < j \le i \le N \\
&L_{ii} = \sqrt{U_{iii}}, && \text{для } 1 \le i \le N \\
&L_{ij} = U_{jij} \big/ L_{jj}, && \text{для } 1 \le j < i \le N
\end{aligned}
\qquad (*)
$$

Формулы (*) оперируют скалярами, но ничто не мешает аналогичные формулы записать для квадратных блоков. При этом будет использовано разложение Холецкого для блока в качестве квадратного корня (действие C – Cholesky), решение треугольной системы в качестве деления (действие T – TriangSolve), а также матричные умножение и вычитание (действие U – Update). Тегами будут номер итерации *k* и индексы блоков матрицы i,j, а данными – результаты одноименных действий: [C: k], [T: k, i], [U: k, i, j]. Фактические программы действий (step code) могут быть написаны на других языках отдельно, и тогда CnC используется как *язык координации* [8].

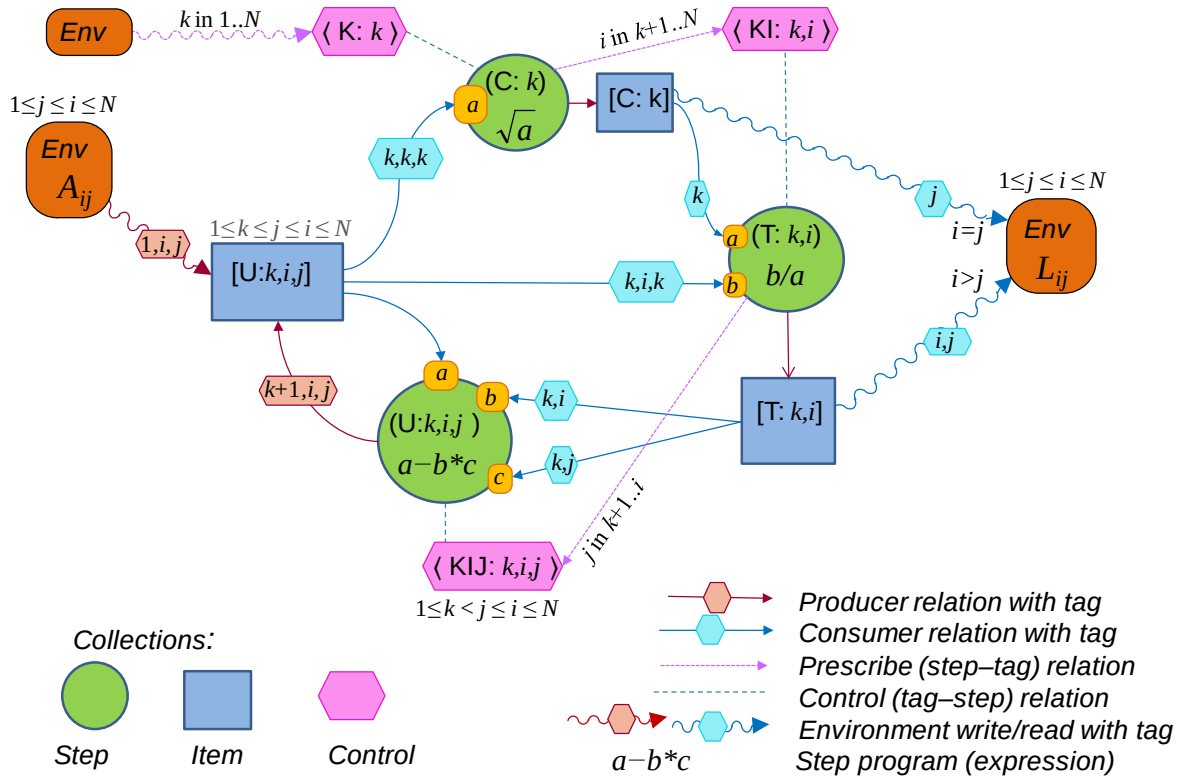


Рис. 3. Полная CnC программа разложения Холецкого.

Полная CnC программа разложения Холецкого изображена на рис.3. Она представляет собой двудольный граф из коллекций действий (C:i), (T:i,j), (U:k,i,j) с одной стороны и коллекций данных и управлений с другой. Компонентами тегов служат номер итерации k и индексы блоков i,j. Входные стрелки идут к действиям от читаемых ими данных, а выходные – от действий к порождаемым ими данным.

Здесь на схеме также приведены конкретные зависимости в виде тегов на стрелках и описания действий (чт***о*** вычисляют действия, условия на зависимостях). Это сделано автором данной статьи, чтобы обеспечить полноту описания на одном рисунке и облегчить сравнение с UPL, хотя в CnC зависимости и прочие условия задают отдельно от схемы, содержащей чистый граф. У действий каждый вход обозначен и поименован, что позволяет их различать и использовать в вычисляемых выражениях. Теги на стрелках, как и

вычисляемые значения, всегда задаются в контексте *действия*, стоящего на одном из концов стрелки, а сам тег задает индекс в коллекции *данных*, стоящей на другом конце. При этом в качестве параметров в тегах могут использоваться только компоненты тега действия (зависимость от данных в CnC не разрешается). Тег на стрелке опускается, если он тривиально образуется из одноименных компонентов тега действия. Для некоторых коллекций дополнительно указаны области значений тегов.

Вычисления будут выполняться следующим образом. Вначале окружение *Env* подает исходную матрицу в виде значений данных [U:1,i,j], а также порождает теги <K:k>, которые запускают действия (C:k) для всех k от 1 до *N*. Но пока только для k=1 данные готовы. Действия (C:1) вычисляют одноименные данные и активируют теги <KI:1,i>, которые запускают действия (T:1,i). Последние, используя данные [U:1,i,1], порождают данные [T:1,i] и теги <KIJ:1,i,j>, которые запускают действия (U:1,i,j), которые породят данные (U:2,i,j) для следующей итерации и т.д.

Можно заметить, что 2-я, 3-я и т.д. итерации могут начинаться задолго до полного окончания 1-й. Это важно, поскольку открывает дополнительные формы параллелизма. Например, можно сгруппировать MxM соседних блоков и M последовательных итераций в один кубический блок MxMxM и выполнять такие блоки независимо на разных процессорах.

В модели CnC автор видит следующие проблемы и недостатки.

1. Необходимость явной активации действия одним источником (внешней средой или другим действием) вынуждает искусственно задавать этого единственного активатора (среду или какое-то действие) для каждого действия, тогда как по смыслу активность может исходить от нескольких источников, как например в случае внешнего произведения двух разреженных векторов.

2. Одна из трудных проблем, которую приходится решать — переиспользование памяти для данных, без которого доступная память будет быстро исчерпываться. Предлагается устанавливать счетчики использований (Get counts), но это весьма хлопотно.

3. В рамках общей модели отсутствует редукция, что вынуждает определять конкретный порядок, например, суммирования при умножении векторов и матриц. Есть решение в рамках реализации INTEL [9], но его сложно признать удачным, поскольку оно вводится как библиотечный подграф, то есть как бы вне самой модели, а потому будет сложно проводить анализ и преобразования таких программ с редукцией. Кроме того, в нем количество редуцируемых элементов должно быть известно заранее. Есть предложение [10], допускающее заранее неизвестное число элементов, основанное на строгой иерархической вложенности редукций, что несколько ограничивает возможности.

4. Жесткий детерминизм сужает область применения, поскольку существуют задачи, сама постановка которых требует недетерминизма, как

например, параллельный недетерминированный обход графа (например, для получения произвольного остовного дерева).

5. Расслоение на действия и данные создает избыточную степень свободы, которой приходится распоряжаться, указывая тег порождаемого действием элемента данных. Чаще всего он тривиально совпадает с тегом порождающего действия, но тогда это приходится повторять.

6. Продукт работы предметника разбит на три части: спецификация области (граф из коллекций), функциональные описания отношений и программы действий, — которые по смыслу сильно связаны и их сложно поддерживать в разделенном виде.

4. UPL(G)

Universal Parallel Language (Graphical) — это проект, в котором задействован автор данной статьи. Он оказался весьма близок к проекту CnC, хотя развивался независимо. Исторически язык UPL возник из проекта потоковой параллельной вычислительной системы [11] путем абстрагирования реализуемой ею модели вычислений от частных технических решений и воплощения этой модели в языке высокого уровня. Начальное представление о нем, правда под другим именем (DFL-G), дается в статье [12].

Как и в CnC, в языке UPL программа представляет собой набор тегированных коллекций. Но в UPL имеется только один сорт коллекций, которые будем называть узлами (узлами-коллекциями), а их элементы — узлами-экземплярами, опуская уточнение, когда это не создает проблем. Они аналогичны действиям CnC, однако, их активация производится не через особые коллекции управления, которых в UPL нет, а просто по готовности всех входных данных. А данные направляются не в специальные коллекции данных, а сразу на входы узлов.

С каждым узлом-коллекцией связана небольшая программа (codelet), которая выполняется при активации узла-экземпляра. Узел с данным тегом активируется, когда на всех его входах присутствуют данные. Тем самым активация возникает не по одному сигналу, а по комбинации нескольких. Тогда узел становится готовым к выполнению, что означает, что в некоторый будущий момент он будет выполнен, рано или поздно. (На самом деле, акт активации является более сложным атомарным действием, изменяющим общее состояние, см. ниже). В результате выполнения будут выданы операции посылки данных на входы других узлов. Таким образом, роль хранимых данных играют состояния входов узлов и только они.

В отличие от CnC, узел не выполняет присвоение значения входу другого узла, а посылает на него токен — небольшое сообщение, содержащее значение и полную адресную информацию. Этот токен попадет на вход узла в некоторый более поздний момент (при этом сама посылка токена неблокирующая: токен отправляется и программа узла продолжает работу, не дожидаясь прихода токена к получателю). Значение токена станет значением входа, доступным

программе узла, только в момент активации, а до этого на входах узла происходит накапливание множества токенов. Причем и на одном входе могут накопиться несколько токенов. После прихода каждого токена проверяется, не возникло ли условие активации. В типичном случае это, например, что на каждом входе присутствует хотя бы один токен. Если да, то одномоментно (атомарно) формируется пакет с заданием на выполнение программы узла с готовыми значениями на всех входах, и при этом участвовавшие в этой активации токены, как правило, удаляются из входной копилки. Те, что остались, могут участвовать в дальнейших активациях.

В качестве примера рассмотрим описание на UPL(G) алгоритма разложения Холецкого (рис.4). Он основан на тех же формулах (*), которые перепишем в виде:

$$U_{ij} = A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk}, \text{ для } 1 \le j \le i \le N$$

$$L_{ii} = \sqrt{U_{ii}}, \qquad \text{для } 1 \le i \le N \qquad (**)$$

$$L_{ij} = U_{ij} / L_{jj}, \qquad \text{для } 1 \le j < i \le N$$

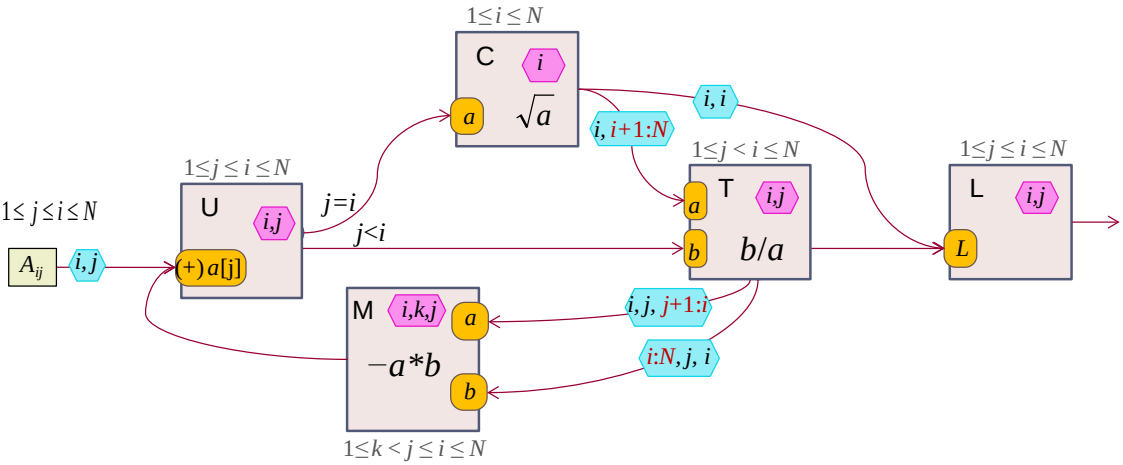


Рис. 4. Программа разложения Холецкого на UPL(G).
Допустимые области значений тегов узлов являются комментариями

Схема (рис.4) реализует формулы (**) почти непосредственно. Узлы C⟨*i*⟩ и T⟨*i*,*j*⟩ представляют, соответственно, диагональ L_{ii} и остальную часть результата L_{ij}, узел M⟨*i*,*k*,*j*⟩ — произведение под знаком Σ. Для реализации суммирования мы применили специальный суммирующий вход узла U, вычисляющий редукцию: (+)*a*[*j*]. Перед именем *a* в скобках стоит знак операции, после имени в квадратных скобках – число элементов. Здесь оно выражено через индекс. Такая запись означает, что когда на вход U.a⟨*i*,*j*⟩ придет ровно *j* токенов, значение входа (равное сумме значений всех пришедших токенов) станет готовым, в результате чего узел будет активирован. В

частности, когда на узел U⟨*i*,1⟩ придет токен из среды с элементом входной матрицы A_{i1}, то активация произойдет немедленно (для каждого *i*).

В началах стрелок из узла U стоят условия, указывающие, что значение *a* при *i=j* будет отправлено на узел C, в остальных случаях — на T. В обоих случаях теги по умолчанию будут, соответственно, ⟨*i*⟩ и ⟨*i*,*j*⟩. Некоторые индексы на стрелках заданы диапазоном, например: *i*+1:*N*. Такая запись указывает, во-первых, что этот токен направляется на каждый из узлов с индексом из этого диапазона, а во-вторых, что ему будет приписана кратность по числу элементов диапазона, в данном случае *N–i*. Это значит, что данный токен будет взаимодействовать многократно, то есть при каждой активации он не будет удаляться из входной копилки, пока не будут совершены такое количество активаций. Особенно интересен случай узла M, на оба входа которого много токенов с диапазонами. Рассмотрим, например, токен на M.a⟨10,5,6:10⟩ и токен на M.b⟨8:100,5,8⟩. Они породят активацию узла M⟨10,5,8⟩, но при этом оба останутся на входах для других взаимодействий (если кратность еще не исчерпана). Повторное взаимодействие той же пары семантика запрещает. Таким образом, для данного *j* придут *N–j* токенов на вход *a* и столько же на вход *b*, все пары будут пытаться взаимодействовать, но с учетом диапазонов произойдет только (*N–j*)(*N–j*+1)/2 взаимодействий. Например, токенам на M.a⟨10,5,6:10⟩ и M.b⟨20:100,5,20⟩ контроль диапазонов не позволит взаимодействовать.

Указывать диапазон не обязательно: может быть задан просто индекс *, что означает "любое значение" в этом месте. Оно будет определяться набором токенов на другом входе (входах)[1]. Тогда лишние активации узла M придется подавить, вводя в его программу дополнительное условие: **if** (j<i)…, и задавая явно кратность #(*N–j*) (в контексте узла-отправителя T) на обоих входах.

Задание точной кратности – залог своевременной очистки памяти от данных, которые больше не понадобятся. Когда она неизвестна, можно указать ее бесконечной (##), но потом позаботиться об отправке специального токена чистки, типа **clear** M⟨*,j,*⟩, в подходящий момент (здесь это можно сделать из узла C⟨j+1⟩).

Как в CnC, пользователь-настройщик может управлять фактическим параллелизмом, задавая функции распределения узлов по процессорным ядрам (пространству) и по этапам (времени). Эти функции зависят от тега (для каждой коллекции узлов, вообще говоря, по-своему). Через выбор функций распределения решаются вопросы балансировки нагрузки, снижения нагрузки на сеть, экономии памяти и т.п. Компиляторам функции распределения помогут генерировать эффективный код для выбранной платформы. Более того, разные функции распределения для одной и той же схемы приводят к разным алгоритмам, как показано в [13] для умножения матриц.

[1] Наличие * в одной позиции одновременно на всех входах является ошибкой.

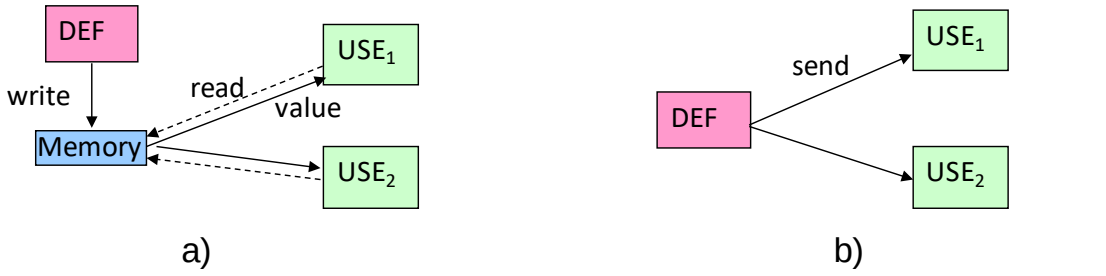
5. Сравнение и выводы

Были рассмотрены несколько проектов, нацеленных на решение проблемы многообразия языков и платформ параллельного программирования. При этом они были представлены только с точки зрения языковых средств и предоставляемых возможностей для программиста. Вопросы отображения на разные платформы и его эффективности почти не затрагивались. Поэтому и их сравнение будет ограничено сферой языка и его семантики.

Все описанные проекты основаны на потоковой модели вычислений в ее "абстрактном" понимании: продолжение вычисления определяется готовностью данных. При этом все, кроме UPL, провозглашают и следуют принципу единственного присваивания (определения). На нем, в частности, базируется их детерминизм.

Далее начинаются отличия UPL. Один из ключевых аспектов — какая поддерживается парадигма: сбора или раздачи. Мы говорим, что модель вычислений, или язык поддерживает *парадигму сбора*, если за использование значений отвечает их потребитель, а производитель лишь сохраняет их в условленном месте (рис. 5а). При *парадигме раздачи*, наоборот, потребителю данные кладутся "в руки" производителем (рис. 5б). Парадигма сбора является более привычной, на ней основаны практически все традиционные языки программирования: Фортран, Алгол, С и т.п. Сама запись арифметического выражения: (ax+by)/(x+y) – апеллирует к парадигме сбора: имя переменной это имя места, откуда нужно «достать» значение. К этому мы привыкли со школы.

В парадигме раздачи производитель отвечает за рассылку, раздачу произведенных им данных потребителям. Он должен «знать» всех своих потребителей и позаботится о передаче им своих результатов. Реально «знать», конечно, должен программист, но он должен будет указать потребителей при описании производителя.



Производитель сохраняет свой результат в памяти по некоторому адресу, откуда его запрашивают потребители по мере надобности (по тому же адресу).

Производитель «знает» (вычисляет) сам «адреса» всех потребителей, на которые и рассылает свой результат.

Рис.5. Парадигмы сбора (a) и раздачи (b).

Классические dataflow системы, к которым относится LabVIEW, являются смешанными, в том смысле, что в них обе стороны "осведомлены" о другой:

обычно говорят, что они как бы связаны "проводом" (wire), по которому "текут" данные. Эта модель близка аппаратуре. Dataflow языки более высокого уровня: SISAL, VAL, ID привержены парадигме сбора. Сюда же относится и SAC.

CnC поддерживает парадигму сбора благодаря разделению коллекций на действия и данные. Действие-производитель кладет результат в определенное место, а потом действия-потребители будут сами нужные им данные забирать. В UPL нет отдельных коллекций данных, и поэтому узел-производитель обязан позаботиться о рассылке токенов с данными всем потенциальным потребителям. Это в чистом виде парадигма раздачи. Из других языков парадигму раздачи поддерживают Charm++ и некоторые другие, основанные на передаче сообщений, в частности, MPI (отчасти). Легко видеть, что для непосредственного исполнения на распределенной системе парадигма раздачи предпочтительнее, поскольку в ней связь между одним производителем и k потребителями требует k передач, а в парадигме сбора — $2k+1$.

В UPL, в отличие от всех остальных систем нет присущего детерминизма. Например, на один вход могут прийти два токена от разных источников, так что результат будет зависеть от того, который придет раньше. Можно обеспечить детерминизм, придерживаясь определенной дисциплины, но сама по себе система его не гарантирует. Для некоторой дисциплины возможна run-time верификация, которая может гарантировать, что произведенное вычисление при любых допустимых отклонениях в расписании даст тот же результат. Есть основания полагать, что любая программа на CnC может быть переведена адекватно в UPL так, что детерминизм будет подтвержден. Но UPL, будучи свободным от необходимости гарантировать детерминизм, дает и более широкие возможности, которые могут иметь практическую ценность. Например, это возможность активации по приходу токенов на часть входов, которая использована нами в алгоритме сложения разреженных векторов [12].

Как UPL, так и CnC «страдают» от присущей мелкозернистости (если алгоритм описан в терминах естественных для задачи элементов), что создает проблему высоких накладных расходов при исполнении. Стандартный путь их преодоления, используемый в CnC, – укрупнять элементы, группируя вычисления и данные в блоки. При этом программы действий могут задаваться на других языках, а уровень CnC тогда рассматривается как язык координации [15]. Для UPL этот вариант тоже возможен, но мы предпочитаем более интересный и перспективный путь, который включает две альтернативы:

1. Аппаратная поддержка всех операций по синхронизации, взаимодействию и запуску действий, чтобы свести к минимуму накладные расходы. Этот путь требует разработки специальной архитектуры процессоров.
2. Отображение (трансляция) в обычные языки с автоматическим порождением блочных кодов. В частности, можно получить блочный алгоритм для метода Холецкого автоматически из скалярного.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект 17–07–00324.

Литература

1. Scholz S.-B. Single Assignment C — Functional Programming Using Imperative Style // In Proceedings of the 6th International Workshop on Implementation of Functional Languages (IFL'94), Norwich, England, UK, pp.21.1–21.13, University of East Anglia, 1994.
2. Grelck C., Scholz S.-B. SAC – A Functional Array Language for Efficient Multi-threaded Execution // International Journal of Parallel Programming, Vol. 34, No. 4, pages 383–427, August 2006.
3. Sinkarovs A., Scholz S.-B., Bernecky R., Douma R., Grelck C. SAC/C Formulations of the All-Pairs N-Body Problem and their Performance on SMPs and GPGPUs // Concurrency and Computation: Practice and Experience, Vol. 26, Issue 4, pages 952–971, March 2014.
4. National Instruments (In Russian). — URL: http://russia.ni.com/ (13.04.2017).
5. LabVIEW (In Russian). — URL: http://labview-rus.blogspot.ru/ (13.07.2017).
6. Knobe K., Offner K.D. Tstreams: A model of parallel computation (preliminary report) // Technical Report HPL–2004–78, HP Labs.
7. Budimlic Z., Knobe K. CnC: A Dependence Programming Model // In Sixth International Workshop on Data Flow Models for Extreme-Scale Computing (DFM 2016), September 15, 2016, Haifa, Israel. — URL: http://www.cs.ucy.ac.cy /dfmworkshop/wp-content/uploads/2014/05/Budimlic_2016.pdf (26.07.2017).
8. Zaichenkov P., Gijsbers B., Grelck C., Tveretina O., Shafarenko A. A Case Study in Coordination Programming: Performance Evaluation of S-Net vs Intel's Concurrent Collections // IPDPSW'14: Paral. & Distrib. Proc. Symposium Workshops, May 19–23, 2014, PHOEIX (Arizona) USA, pages 1059-1067.
9. Intel(R) Concurrent Collections C++API. — URL: https://icnc.github.io/api/ reuse.html (13.07.2017).
10. Budimlic Z., Burke M., et. al. Deterministic Reductions in an Asynchronous Parallel Language // WoDet'11: Proceedings of the 2nd Workshop on Determinism and Correctness in Parallel Programming, March 2011.
11. Бурцев В.С. Выбор новой системы организации выполнения высокопараллельных вычислительных процессов, примеры возможных архитектурных решения построения суперЭВМ // В.С. Бурцев, ред., *Параллелизм вычислительных процессов и развитие архитектуры суперЭВМ*. — ИВВС РАН, Москва, 1997. — С. 41–78.
12. Климов А.В., Окунев А.С. Графический потоковый метаязык для асинхронного распределенного программирования // МЭС–2016: Проблемы разработки перспективных микро- и наноэлектронных систем – 2016. Сборник трудов. Часть II. — М.: ИППМ РАН, 2016 — С. 151–158. — URL: http://www.mes-conference.ru/data/year2016/pdf/D149.pdf (26.07.2017).
13. Климов А.В. О парадигме универсального языка параллельного программирования // Языки программирования и компиляторы – 2017, труды конф. под ред. Д.В. Дуброва. Ростов-на-Дону, ЮФУ, 2017. — С. 141–146. — URL: http://plc.sfedu.ru/files/PLC-2017-proceedings.pdf (26.07.2017).