ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ № ?(??), 2015

УДК 004.42-43

А. В. Климов

Параллельное потоковое графическое программирование для суперкомпьютеров

Аннотация. Предлагается проект языка и инструментальной системы параллельного программирования. Проект ставит целью позволить, написав один раз математическое описание алгоритма, в дальнейшем систематически выводить из него эффективные программы для различных вычислительных платформ. Предлагаемый язык основан на потоковой модели вычислений и допускает удобное и выразительное представление в графической форме.

Ключевые слова и фразы: потоковая модель вычислений, парадигма раздачи, графическое программирование, параллельное программирование, мелкозернистый параллелизм.

Введение

Предлагается проект языка и инструментальной системы параллельного программирования. Проект ставит целью позволить, написав один раз математическое описание алгоритма, в дальнейшем систематически выводить из него эффективные программы для различных вычислительных платформ.

Существующие средства и языки параллельного программирования плохо отвечают данной цели. Это связано с тем, что они основаны на модели последовательного программирования, и потому они вынуждают программиста при записи алгоритма принимать много лишних решений о порядке вычислений. Это осложняет дальнейшую задачу анализа и оптимизации кода, не говоря об увеличении трудозатрат. Хорошая форма записи должна побуждать автора фиксировать только математическую структуру алгоритма, его вычислительный граф. Попросту говоря, это что от чего и как зависит, а не порядок вычислений или иные аспекты их организа-

© А. В. Климов, 2015
© ИППМ РАН, 2015
© Программные системы: теория и приложения, 2015

ции (распределение по процессорам, блочность и т.п.). Все эти дополнительные аспекты должны привноситься в программы на более поздних стадиях, с учетом характеристик используемой вычислительной платформы и, по возможности, автоматически. И если математический алгоритм был отлажен, то эти вариации не должны нарушать его правильность (и это должно гарантироваться).

1. Потоковая модель вычислений с динамически формируемым контекстом

Мы полагаем, что искомая форма записи алгоритмов может быть основана только на модели вычислений с управлением потоком данных (dataflow). В ней вычисления разбиваются на фрагменты, выполнение которых организуется по принципу готовности данных: когда все входные данные некоторого фрагмента готовы, он может выполняться, и его выполнение приводит к вычислению новых данных, которые будут входными для других фрагментов и так далее. Для такой работы программа должна задавать только вычислительный граф (зависимостей), а порядок выполнения выстраивается автоматически в динамике.

В принципе идея управления потоком данных (dataflow) уже давно разрабатывалась и породила ряд языков и систем. Но в них был один существенный недостаток: стремясь уподобиться по форме существующим языкам структурного программирования (Паскаль, С), они унаследовали от них и склонность к последовательной организации вычислений – через такие конструкции как циклы и вызовы процедур. На эти конструкции опиралось использование в реализациях этих языков понятия *контекста*, позволяющего одному фрагменту узла работать многократно, параллельно с другими экземплярами себя же, без риска перепутывания данных.

В нашей версии потоковой модели мы даем возможность использовать контекст как открытый ресурс, явно задавая все его изменения. И мы уже не стремимся обеспечить кажущееся сходство с привычным программированием. Оно в лучшем случае сохра-

няется внутри небольших вычислительных фрагментов – узлов, между которыми действует принцип управления потоками данных.

2. Графическое программирование

Поскольку вычисления задаются как не последовательные, то и линейная текстовая форма записи имеет мало содержательного смысла. Более естественной будет графическая форма, где вычислительным фрагментам соответствуют блоки, а зависимостям по данным – соединяющие их стрелки. Мы здесь тоже не первые - есть, например, система LabVIEW компании NI (National Instruments [1]), которая также предлагает графическое программирование на основе потоковой организации. Она развивается уже на протяжении 30 лет и имеет довольно широкое распространение в инженерных кругах. Но, как и прочие потоковые системы или языки, она унаследовала многое от последовательных языков, считая это своим достоинством. В частности в ней присутствует структурность, представленная главным образом циклами и вызовами процедур.

Мы в принципе можем заимствовать все удобные средства из графического языка G системы LabVIEW, в частности, циклы, когда они присущи самому алгоритму, например для итерационных алгоритмов. Но в нашем графическом языке будет новый элемент – явное использование контекста. Циклы можно рассматривать лишь как надстройку, выражающую определенный шаблон программирования в базовом языке, DFL-G.

Циклы и структурность также вынуждают программиста принимать несущественные решения. Например, задавая двойной цикл, он вынужден один из циклов сделать внешним, а другой внутренним, что может иметь далеко идущие последствия. Наш язык в таких случаях позволяет обойтись без таких излишеств. Продемонстрируем это на конкретном примере, заодно показав характерные отличия нашего предложения от системы LabVIEW.

Пусть имеется изображение, например черно-белая фотография, размером nxm пикселов. Требуется вычислить ее момент 1-го

порядка, иначе говоря – центр тяжести. Его координаты (mx,my) 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. 1.

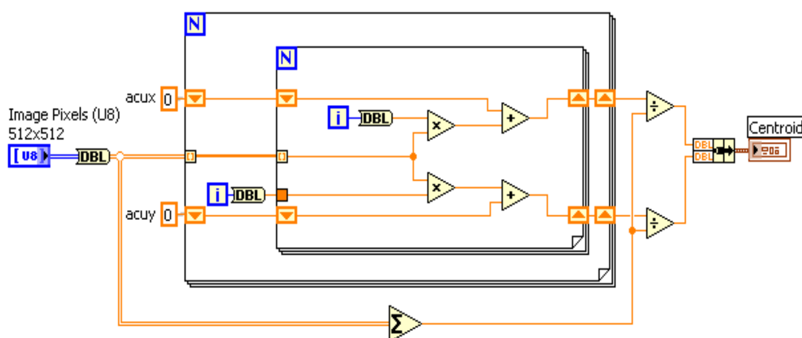

Рис. 1. Программа вычисления моментов в системе LabVIEW (взято из сайта [2])

Обратите внимание, 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. Также заметим, что здесь каждый пиксел умножается (как в формулах) на i или j, а потом произведения складываются. Попытаемся сделать улучшение, чтобы делать один раз умножение на x для всего стролбца, и на y для всей строки. Здесь это легко сделать только для строк, "обведя" внешний индекс i вокруг внутреннего цикла, бегущего вдоль строк. А для столбцов так же про-

сто уже не получается. Придется нарисовать две пары циклов: в одной паре внутренний цикл по строкам, а в другой по столбцам. 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. Полезность графики в основном только в том, что связи по данным выражены более наглядно линиями, а не именами переменных.

В языке DFL-G аналогичную задачу можно выразить, как показано на Рис. 2.

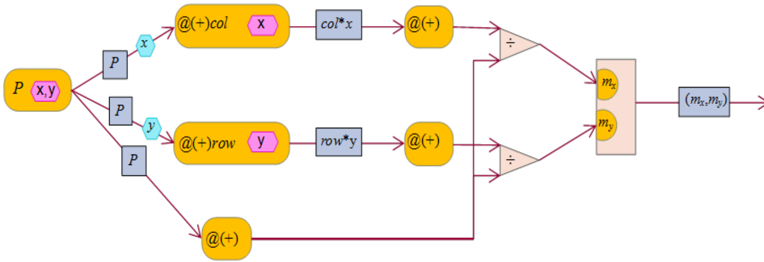

Рис. 2. Программа вычисления моментов в языке DFL-G

Синие блоки на стрелках показывают пересылаемые данные – токены. Голубые шестиугольники – значение контекста. Для узлов также задается контекст (в красном шестиугольнике). Контест на стрелке можно опускать, когда он извлекаются из окружения по простому правилу. Например, если узел исхода имеет контекст {x,y}, а узел прихода – контекст {y}, то по умолчанию на стрелке контекст будет {y}. Элемент данных тоже можно не указывать, если он совпадает со значением первого (единственного) входа узла-источника или его последней формулы (например, P). Значение

и данного и контекста токена формируются в узле-отправителе из имеющихся в нем значений входов и контекста. В общем случае вычислительный узел изображается блоком (бледно-розовым), содержащим помеченные входы (желтые усеченные кружки), в которые идут стрелки от других узлов и контекст (красный шестиугольник). Внутри блока могут быть записаны формулы для вычисления величин, используемых в формировании данных и контекста новых токенов. В частном случае узла с одним входом и без формул используется один желтый овал. В случае одной бинарной операции используется треугольник. У основания стрелки может быть записано условие посылки токена (вычисляемое также исключительно на базе входов и контекста узла-источника). Один узел может вырабатывать несколько токенов (или ни одного), один вход также может принимать токены от разных узлов. Узел "не знает", от кого придут токены на его вход. Каждый пришедший токен может породить активацию (согласно правилам активации).

В примере на Рис. 2 почти все узлы одновходовые, но некоторые из входов – суммирующие, что выражается префиксом (+) перед именем (там, где имя входа не используется потом, оно и не вводится). Это значит, что на такой вход могут прийти несколько значений (или нуль) и все они будут просуммированы. Обычно из сравнения контекстов понятно, что это за множество. Например, на вход col с контекстом $<x>$ поступают токены из узла P с контекстом $<x,y>$. Значит, на входе $col<x>$ суммируются значения $P<x,y>$ с различными y, поступающие в неизвестном порядке.

Возникает вопрос – как исполнителю определить, что пора считать сбор слагаемых завершенным. В данном примере число слагаемых известно заранее – это константы m или n, и мы могли бы этим воспользоваться. А в общем случае можно поставить знак @, который означает "ждать всех", и исполнитель должен суметь определить, когда эти "все" придут. Например, это может быть некоторый способ распознавания "тишины" [3] – ситуации отсутствия активности некоторого вида, когда можно гарантировать, что больше токенов на таком-то входе не будет. (Это легкий для про-

граммиста, но тяжелый для вычислителя способ задать количество суммируемых слагаемых). Например, на Рис.2 сказано, что когда на строку $row<y>$ придут все элементы, их сумма будет умножена на y и отправлена дальше на суммирование.

Мы сразу поместили умножения после сложений по строкам и по столбцам, хотя ничто не мешало поставить их и перед сложениями как в схеме на Рис. 1. Теперь здесь нет никаких циклов, все они неявные, определяемые потоком токенов. В каком порядке они будут приходить, в таком и будут суммироваться. Если системе будет известна дополнительная информация о диапазонах для x и y, то может быть применен параллельный вариант суммирования методом "сдваивания". В исходном коде нет указаний на конкретный способ подсчета сумм.

Основной новый элемент в нашем языке – явное формирование контекстов в посылаемых токенах. Это радикально расширяет возможности программирования, меняя самый его стиль. Контекст аналогичен индексам массивов, разница в том, что у нас каждый элементы существуют независимо от других, неся при себе свои индексы. Массив целиком может никогда не существовать, быть неполным или разреженным. Такая операция как "изменить элемент массива" в нашей парадигме не имеет смысла, тогда как в классических dataflow системах, включая LabVIEW, наоборот, новый массив часто приходится строить путем последовательных модификаций старого.

Код на Рис. 2 получился симметричным по строкам и столбцам. Решение о том, по какой из координат бежит внешний цикл, по какой – внутренний может быть отложено. Можно даже организовать обход по блокам. Эти аспекты определяются отдельно от программы – через механизм распределения вычислений по пространству и времени. Основу для управления распределением дает опять же контекст. Распределение задается как функция (вообще говоря, своя для каждого узла) зависящая от контекста и выдающая номер процессора (пространство) или номер этапа (время). Функция $place(<...>)$ задает распределение по процессорам, то есть по пространству, или *placement*. Функция $stage(<...>)$ задает рас-

пределение по этапам, то есть по времени, или *scheduling*. Например, функция $place(<x,y>)=(zip(x,y)/256)\%N_P$ задает распределение двумерного пространства блоками 16x16. (*zip* - это операция скрещивания двоичных представлений целых чисел. Завершающее взятие по модулю числа процессоров, $\%N_{P,}$ можно опустить: оно применяется по умолчанию.) Задавая функцию *stage*, программист указывает предпочтительный порядок обхода. Подробно этот вопрос рассмотрен в работе [4].

3. Парадигма раздачи или сбора?

Будем говорить, что язык или модель вычислений основаны на парадигме сбора, если при описании некоторого вычислительного фрагмента мы не интересуемся, где будут использованы его результаты, но при этом должны явно задать, откуда взять аргументы. Соответственно, для парадигмы раздачи все наоборот: при записи фрагмента мы имеем все аргументы уже "под рукой", но при этом должны позаботиться о направлении результата в правильные места. Традиционное программирование основано на парадигме сбора: когда мы пишем, например,

```
C[i,j] = A[i,k]+B[k,j]
```

то именами и индексами в правой части мы указываем, откуда надо взять слагаемые, а левая часть обычно есть просто место для сохранения результата этого оператора. Где он будет использоваться, будет записано в момент использования. Это очень естественно для нас, мы привыкли к этому еще со школы. Предлагаемая модель вычислений основана на парадигме раздачи. В программе узла аргументы именуются локальными именами входов, не важно откуда они придут. А результаты, наоборот должны быть сразу направлены туда, где они будут использованы: на входы соответствующих узлов. Это кажется неудобным, непривычным, но это – наиболее точно отвечает эффективной работе при распределенном вычислении: вычисленное значение сразу направляется в указанное место, а когда дойдет дело до его использования, все необходимые значения уже будут "под рукой".

В графической записи передача выражается стрелкой: у нее есть начало и конец. То есть сама по себе стрелка ни одну из двух парадигм не выделяет. Асимметрия проявляется в метках на стрелках. Это условие передачи, выражение для значения и выражения для полей контекста целевого узла. И все это выражено в терминах входных величин и контекста узла-источника. В том числе контекст целевого узла. Тем самым отправитель "знает", кто получатель, но получатель "не знает", кто отправитель. А это и есть парадигма раздачи. В парадигме сбора было бы наоборот.

Если контекст узла источника обозначить I, а контекст узла получателя J, то на стрелке будет написано $F(I)$, имея в виду присваивание $J = F(I)$. Чтобы перейти к парадигме сбора, надо взять обратную функцию и написать $I = F^{-1}(J)$. Но обратная функция не всегда возможна, например для ее вычисления в узле-получателе может не хватить информации. Но если все F на стрелках обратимы, то мы можем говорить об обратимости потокового графа.

Программы с обратимым графом образуют важный подкласс, открывающий интересные возможности анализа и преобразований. Например, он позволяет смотреть на программу, как на систему рекуррентных уравнений (System of Recurrence Equations – SRE), которые можно вычислять как функциональную программу – от конца (не путать с обращением функции, которую вычисляет сама программа!). В такой форме удобно проводить некоторые эквивалентные преобразования, в частности объединение узлов (*fuse*). Поэтому для обратимых программ это преобразование может быть применено и к программе в парадигме раздачи.

Для исполняющей аппаратуры парадигма раздачи имеет одно важное преимущество: снимается проблема предсказания для предзагрузки данных в кэш. Точнее, нам точно известно, *что* надо предзагрузить, только не вполне известно, *когда*. Если загружать слишком рано, то может не хватить объема кэша. Возможная стратегия разрешения этого вопроса предложена в [4].

4. Задача автоматического распараллеливания

Обыкновенно эта задача ставится как задача преобразования исходно чисто последовательного кода к параллельному. В последние десятилетия был достигнут значительный прогресс в этой задаче – но лишь для ограниченного класса программ. Это класс так называемых аффинных программ, основанный на аффинных циклах, и некоторых его расширениях. Метод состоит в том, что сначала для программы строится ее полиэдральная модель [5], которая анализируется (на предмет построения функций для placement и scheduling) и затем отображается в параллельный код.

Оказывается, полиэдральная модель есть не что иное, как потоковая программа, причем с обратимым графом! Для полиэдральных моделей уже наработаны техники отображения их в эффективный код для разных моделей параллельного программирования: OpenMP, MPI, DVM, CUDA и т.п. Обобщение этих техник на более общий случай потоковых программ – актуальнейшая задача. Вот что говорит об этом википедия (в статье для "Visual prorgamming language" [6]):

Dataflow languages also allow automatic parallelization, which is likely to become one of the greatest programming challenges of the future (Потоковые языки также допускают автоматическое распараллеливание, что похоже будет одной из крупнейших проблем программирования в будущем).

5. Прообразы потокового языка PolyDFL(-G)

Мы также находим элементы потокового языка PolyDFL(-G) в следующих языках и моделях программирования:

- TTDF (Tagged Tokens Dataflow Architecture) [7].
- LINDA [8].
- CHARM++ [9]. Этот язык и система наиболее близки нашей модели. Они также опираются на парадигму dataflow, которая, по мнению авторов, сближает структуру программы со структурой решаемой проблемы [10]. Наша работа

может рассматриваться как возможная надстройка над этой системой. Необходимо рассмотреть и сравнить более подробно черты этой и нашей систем.

- Polyphonic C# (J-calculus) [11]
- LabVIEW [2]
- Intel TBB [12]

6. Заключение

Введение в узлах и токенах контекста (с возможностью его явного формирования и использования) дает возможность "убить" сразу несколько "зайцев":

- Представлять сложные структуры данных: массивы, ключевые таблицы, разреженные матрицы и т.п.
- Лаконично и выразительно описывать обработку таких структур без явных циклов.
- Гибко управлять распределением вычислений и данных по пространству и времени.
- Раскрыть возможности data-параллелизма во всей полноте и многообразии.
- Анализировать и оптимизировать локальность и коммуникационную сложность алгоритмов.

Кроме того, язык позволяет:

- выражать решение "на уровне спецификаций"
- выражать решение в наглядной графической форме
- проводить анализ и преобразование программ
- наложив те или иные ограничения, извлекать код с явным параллелизмов для конкретных архитектур

Язык был опробован и хорошо показал себя в ряде прикладных областей, особенно в областях с нерегулярной обработкой нерегулярной информации:

- Параллельная сортировка (пузырьком, быстрая, слиянием)
- Обработка изображений.
- Линейная алгебра, разностные методы (явная, неявная схемы), разложение Холецкого.

- Быстрое преобразование Фурье.
- Молекулярная динамика (асинхронная, задача N тел).
- SAT-задача (Servey Propagation method).
- Обработка графов (поиск вширь, алгоритм Дейкстры нахождения кратчайшего пути, разбиение на сообщества Лувенским методом [13]).

Список литературы

[1] http://russia.ni.com/ .
[2] http://labview-rus.blogspot.ru/ .
[3] Amitabh B. Sinha, Laxmikant V. Kalé, Balkrishna Ramkumar. A Dynamic and Adaptive Quiescence Detection Algorithm. Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 1993.
[4] А. В. Климов, Н. Н. Левченко, А. С. Окунев, А. Л. Стемпковский. Суперкомпьютеры, иерархия памяти и потоковая модель вычислений // Программные системы: теория и приложения: элек-трон. научн. журн. 2014. Т. 5, № 1(19), с. 15–36.
[5] Paul Feautrier. Array dataflow analysis. In Santosh Pande and Dharma P. Agrawal, editors, Compiler Optimizations for Scalable Parallel Systems, pages 173–219. Springer-Verlag New York, Inc., New York, NY, USA, 2001.
[6] https://en.wikipedia.org/wiki/Visual_programming_language .
[7] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture, IEEE Transactions on Computers, vol. 39, no. 3, 1990.
[8] D. Gelertner, N. Carriero, S. Chandran, S. Chang. Parallel programming in Linda. In International Conference on Parallel Processing, pages 255–263, Aug 1985.
[9] L.V.Kale. The Chare Kernel parallel programming language and system. In Proceedings of the International Conference on Parallel Processing, volume II, pages 17–25, 1990.
[10] J. Yelon and L. V. Kale. Agents: An undistorted representation of problem structure. In Lecture Notes in Computer Science, volume 1033, pages 551–565. Springer-Verlag, August 1995.

[11] http://research.microsoft.com/en-us/um/people/nick/polyphony/intro.htm.
[12] https://software.intel.com/en-us/intel-tbb
[13] Xinyu Que Fabio Checconi Fabrizio Petrini John A. Gunnels. Scalable Community Detection with the Louvain Algorithm. 2015 IEEE 29th International Parallel and Distributed Processing Symposium, pages 28–37, May 2015.

Об авторе:

Аркадий Валентинович Климов

Старший научный сотрудник Института проблем проектирования в микроэлектронике РАН.
e-mail: arkady.klimov@gmail.com

Образец ссылки на публикацию:

А. В. Климов. *Параллельное потоковое графическое программирование для суперкомпьютеров* // Программные системы: теория и приложения: электрон. научн. журн. 2015. Т. ??, № ??(??), с. ??–??.

A. V. Klimov. Parallel dataflow graphical programming for supercomputers.

Abstract. A draft language and development system for parallel programming is proposed. The project aims to allow programmers to write once a mathematical description of the algorithm and then to systematically derive from it efficient programs for a variety of computing platforms. The proposed language is based on the dataflow computation model and allows for a convenient and expressive representation in graphical form.

Key Words and Phrases: dataflow computation model, scattering paradigm, graphical programming, parallel programming, fine grained parallelism.