УДК 004.432.4

Графический потоковый метаязык для асинхронного распределенного программирования

А.В. Климов, А.С. Окунев

Институт проблем проектирования в микроэлектронике РАН, klimov@ippm.ru, oku@ippm.ru

***Аннотация* — Существует много разных языковых парадигм параллельного программирования и каждая диктует свои ограничения на структуру алгоритмов, в связи с чем для каждой из них приходится придумывать алгоритм практически заново. Предлагается новый язык, на котором будет возможно описать алгоритм один раз, после чего из этого описания можно будет систематически выводить эффективные коды в разных парадигмах программирования. Язык ориентирован на асинхронные распределенные алгоритмы и основан на принципе управления потоком данных (dataflow): вычислительный фрагмент выполняется, когда готовы его входные данные. Классическая схема потокового языка расширена введением индексирования узлов, что привело к радикальному изменения самого стиля программирования. Также предлагается графическая форма языка, облегчающая его изучение и использование. В статье язык объясняется на простых примерах. Показано, как индексирование помогает выражать сложные структуры данных, составные операции, позволяет гибко управлять распределением вычислений. Также проводится сравнение нашего проекта с другими системами и подходами, такими как LabVIEW и AM (Active Messages). Вводится понятие о парадигмах сбора и раздачи, помогающее видеть некоторые сходства и различия между языками и системами программирования.**

***Ключевые слова* — графическое программирование, асинхронные распределенные вычисления, метаязык, потоковая модель вычислений, информационная структура алгоритма, активные сообщения, парадигма раздачи, парадигма сбора.**

I. Введение

В настоящее время существуют различные языковые парадигмы программирования высокопроизводительных вычислений на суперкомпьютерах. Среди них – MPI, Shmem, CUDA, CSP, Active Messages. В каждой из них предъявляются свои требования к структуре и свойствам алгоритма, в связи с чем для каждой из этих парадигм программирования приходится придумывать алгоритм практически заново.

Предлагая проект нового языка и инструментальной системы параллельного программирования, мы ставим задачу обеспечить возможность написать один раз математическое описание алгоритма, и затем систематически выводить из него эффективные программы для различных вычислительных платформ. Поэтому мы говорим о таком языке как о метаязыке.

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 (информационную) структуру алгоритма, его вычислительный граф. Иначе говоря, это что от чего и как зависит, а не порядок вычислений или иные аспекты их организации (распределение по процессорам, блочность и т.п.). 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 дополнения не должны нарушать его правильность, что должно гарантироваться системой.

Главной особенностью высокопроизводительных вычислений является их распределенность, то есть, вычисление разбивается на фрагменты, выполняемые в разных местах вычислительной системы и координирующие свою работу через коммуникационную сеть. Это общее свойство всех упомянутых выше языков и оно может служить основой для метаязыка. Мы не рассматриваем парадигмы, основанные на общей памяти, такие как OpenMP.

Многие парадигмы параллельного программирования опираются на глобальную синхронизацию. Есть общее название таких моделей – Bulk Synchronous Parallel (BSP) [1]. Эта модель предполагает глобально синхронное чередование фаз независимых локальных вычислений и коммуникаций, что само по себе привносит серьезные ограничения в работу программ. В частности, они влекут необходимость обеспечивать равномерность загрузки в каждом супершаге. Кроме того, сама глобальная синхронизация на больших системах несет в себе серьезные накладные расходы.

Поэтому мы исходим из того, что универсальный язык распределенного программирования должен допускать асинхронную работу компонентов, когда разные фрагменты могут работать независимо, а сигналом к началу выполнения некоторого вычислительного фрагмента является готовность всех его входных данных и только их. Модель вычислений с такими свойствами называют потоковой или моделью вычислений с управлением потоком данных (dataflow computation model).

Ниже дается общее представление о модели потока данных, которая расширена индексированием. Затем будут описаны изобразительные средства языка и приведены примеры их использования. Сначала мы рассмотрим на одном примере сравнение нашего языка с другой существующей системой графического потокового программирования – LabVIEW ([2], [3]). Обсудим их особенности и различия. Далее на небольшом примере – сложение разреженных векторов – демонстрируются новые уникальные возможности языка. В заключение приводится сопоставление нашего языка с другими аналогами.

II. Потоковая модель вычислений с индексированием

В потоковой модели вычисления разбиваются на семантически мотивированные 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. Чтобы подчеркнуть эту более широкую роль контекста, будем называть его индексом, в общем случае многомерным. Каждый программный узел теперь представляет семейство экземпляров узла, индексированное указанным в описании набором индексов. При посылке токена на узел отправитель указывает явно индексы экземпляра, которому токен предназначен. Каждый экземпляр активируется независимо от других, когда на него придут все требуемые входные токены.

Мы не стремимся обеспечить кажущееся сходство с привычным программированием. Оно в лучшем случае сохраняется внутри программных узлов при записи программы, которая исполняется при активации каждого экземпляра этого узла.

В рамках данной статьи наш язык будем называть DFL. А его графический вариант – DFL-G.

III. Графическое программирование

Поскольку вычисления задаются как не последовательные, то и линейная текстовая форма записи имеет мало содержательного смысла. Более естественной является графическая форма, где вычислительным фрагментам соответствуют блоки, а зависимостям по данным – соединяющие их стрелки. Мы здесь тоже не первые: например, система LabVIEW компании NI (National Instruments [2]) 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 поддержана 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 есть новый элемент – явное использование контекста, который мы называем индексом. Циклы можно рассматривать лишь как надстройку, выражающую определенный шаблон программирования в 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. Язык DFL-G 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 (m_x,m_y) определяются формулами

$$m_x = \sum_{i=1}^{n}\sum_{j=1}^{m} iP_{ij}\ ,\quad m_y = \sum_{i=1}^{n}\sum_{j=1}^{m} jP_{ij}\ .$$

В LabVIEW это вычисление можно выразить графической программой, показанной на рис. 1.

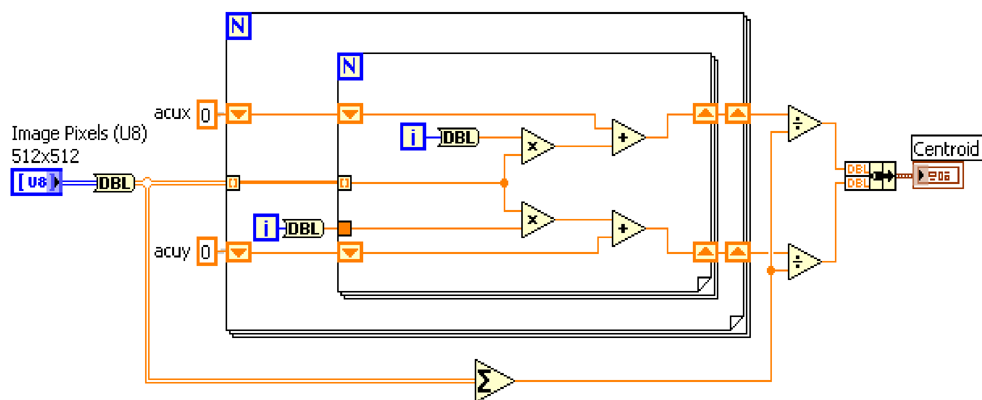

Рис. 1. Программа вычисления моментов в системе LabVIEW (взято с сайта [3])

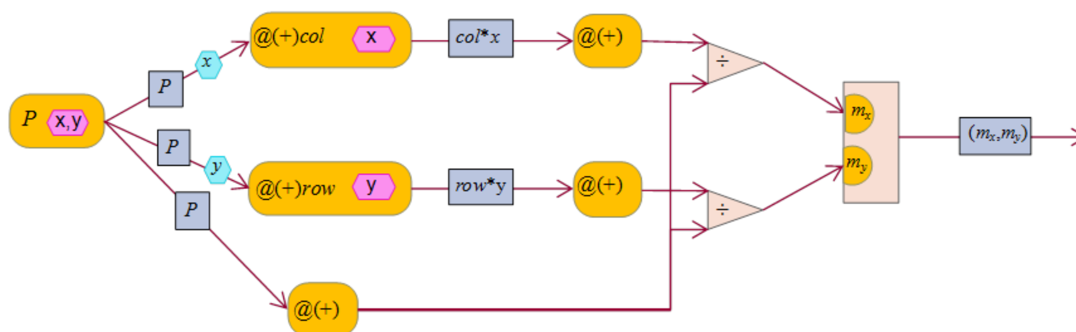

Рис.2. Программа вычисления моментов изображения в языке DFL-G

Здесь имеется 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 столбца, 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 просто 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, Си. 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.

Блоки на стрелках показывают пересылаемые данные – токены. Шестиугольники содержат значение индекса целевого узла. Сами необходимые индексы узлов описаны внутри блоков узлов в красных шестиугольниках. Индекс на стрелке можно опускать, когда он извлекается из окружения по простому правилу. Например, если узел исхода имеет индекс $\langle x,y \rangle$, а узел прихода – индекс $\langle y \rangle$, то по умолчанию на стрелке индекс будет $\langle y \rangle$. Элемент данных тоже можно не указывать, если он совпадает со значением первого (единственного) входа узла-источника или последней формулы (например, P). Значение как данного, так и контекста токена формируются в узле-отправителе на основе имеющихся в нем значений входов и контекста.

В общем случае вычислительный узел изображается блоком, содержащим помеченные входы (каждый в форме усеченного кружка), в которые идут стрелки от других узлов. Внутри блока могут быть записаны формулы для вычисления величин, используемых при формировании данных и индексов новых токенов. В частном случае узла с одним входом и без формул используется овал. В случае одной бинарной операции – треугольник. У основания стрелки может быть записано условие посылки токена. Один узел может вырабатывать несколько токенов (или ни одного), один вход 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 почти все узлы одновходовые, но некоторые из входов – суммирующие, что выражается префиксом (+). Это значит, что на такой вход могут прийти несколько (или нуль) значений и все они будут просуммированы. Обычно, сравнивая индексы, можно понять, что это за множество. Например, на вход *col* с индексом ⟨*x*⟩ поступают токены из узла *P* с индексом ⟨*x*,*y*⟩. Значит, на входе *col*⟨*x*⟩ суммируются значения *P*⟨*x*,*y*⟩ с различными *y*, поступающие в некотором порядке. На месте знака + может находиться другая ассоциативная и коммутативная операция.

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". Определение наступления этого условия может потребовать от исполняющей системы определенных ухищрений, например, обнаружения "тишины" (quiescence).

Мы поместили умножения после сложений по строкам и по столбцам, хотя ничто не мешало поставить их и перед сложениями как в схеме на рис. 1. 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 языке DFL-G – явное формирование индексов в посылаемых токенах. Это существенно расширяет возможности языка, радикально меняя самый стиль програмирования. Индекс в токене аналогичен индексам в обращении к массиву, разница в том, что у нас каждый элемент существует независимо от других, неся при себе свой индекс. Массив целиком может никогда не существовать, быть неполным или разреженным. Такая операция как "изменить элемент массива" в нашей парадигме не нужна, 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 организовать обход блоками. Эти аспекты задаются отдельно от программы – через распределение вычислений в пространстве и времени.

Индекс предоставляет удобные возможности для управления распределением вычислений по физическому пространству процессорных элементов. Распределение задается как функция, зависящая от индекса узла и выдающая номер процессора (*place*) или номер этапа (*stage*). 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). Задавая функцию *stage*, программист указывает предпочтительный порядок обхода. Подробно этот вопрос рассмотрен в работе [4].

IV. Парадигма раздачи и парадигма сбора

Будем говорить, что язык или модель вычислений основаны на парадигме сбора, если при описании некоторого вычислительного фрагмента мы задаем явно, откуда взять аргументы, но не сообщаем, где будут использованы результаты. 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 это обычно место для сохранения результата. Где, кем и когда он будет использоваться, станет известно в момент использования (в виде запроса на чтение). Это для нас естественно, мы еще со школы к этому привыкли.

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 дело дойдет 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 индекса целевого узла. Все выражения, включая индекс целевого узла, заданы в терминах входных величин и индекса узла-источника. Тем самым отправитель "знает", кто получатель, но получатель, вообще говоря, "не знает", кто отправитель. А это и есть парадигма раздачи. В парадигме сбора было бы наоборот.

Если индекс узла источника обозначить I , а индекс узла получателя J , то на стрелке будет написано $\langle F(I) \rangle$, имея в виду приравнивание

$J = F(I)$. 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 функции 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, которые можно вычислять как функциональную программу – от конца. В такой форме удобно проводить некоторые эквивалентные преобразования, в частности объединение узлов. 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. При этом точно известно, ***что*** надо предзагрузить, но только не вполне известно, *когда*. Если загрузить слишком рано, то может не хватить объема кэша. Возможная стратегия разрешения этого вопроса предложена в [4].

V. Активация по частичному набору токенов

Обычно предполагается, что активация экземпляра узла происходит, когда на всех входах имеются токены. Мы ослабляем это правило, допуская возможность нескольких вариантов активаций – ветвей. При этом для каждой ветви задано свое подмножество входов, наличие токенов на которых вызывает ее активацию. Между ветвями задан (частичный) порядок, который определяет выбор ветви при наличии нескольких готовых к активации. Для несравнимых ветвей выбор недетерминирован.

В качестве примера узла с ветвями рассмотрим задачу сложения двух разреженных векторов a и b. Вектор представляется множеством токенов, направленных на вход узла с индексом, индексирующим элементы вектора. В этом множестве могут отсутствовать нулевые элементы. Кроме того, известно общее количество присутствующих элементов, которое направлено на некоторый узел без данного индекса. (У всех узлов могут быть также другие общие индексы – номер шага, поколения и т.п.). Результат сложения должен представляться аналогично.

Мы хотим добиться полной асинхронности. Поэтому перед нами две проблемы: 1) определить, когда закончатся все сложения элементов с общими индексами и 2) посчитать количество элементов n_c результирующего вектора c. Если было бы известно количество общих элементов m, то

$$n_c = n_a + n_b - m .$$

Решение показано на рис. 3. На входы a и b узла AddSV⟨i⟩ приходят элементы входных векторов a_i и b_i. Количества n_b и n_b складываются, и сумма направляется на вход *n* узла Cnt. На его входе (+)*k*[*n*] производится подсчет числа элементов результата. На него придет ровно по одному токену от каждой активации одной из первых трех ветвей узла AddSV.

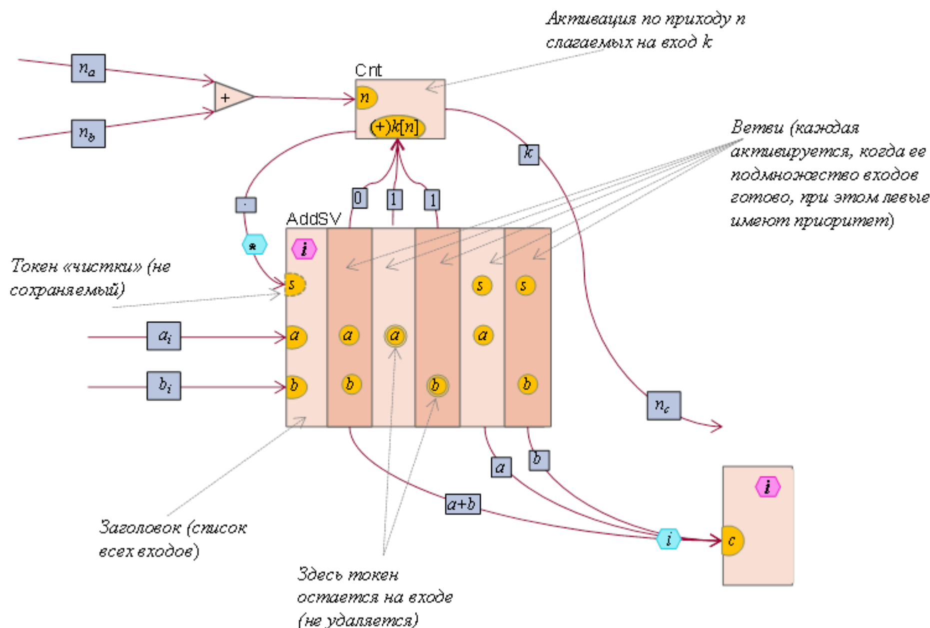

Рис. 3. Граф-схема алгоритма суммирования разреженных векторов

Левая колонка - это заголовок. В ней перечислены все входы узла и его индекс. Каждая следующая колонка соответствует одной ветви. В ней повторены входы, необходимые для активации. Ветви упорядочены слева направо. Первая ветвь сработает при наличии элементов *a* и *b*. При этом они будут удалены, поскольку вход помечен одиночным кружком. Вторая ветвь сработает при наличии входа *a*. При этом, поскольку она стоит после первой, ее срабатывание предполагает, что элемент *b* отсутствует. Вход *a* во второй ветви помечен двойным кружком, означающим, что после ее срабатывания токен на этом входе будет сохранен. Поэтому, если затем придет токен на вход *b*, сработает первая ветвь. Аналогично, третья ветвь сработает, когда первым придет токен на вход *b*. С одним и тем же набором токенов каждая ветвь может сработать не более одного раза.

При каждом срабатывании одной из первых трех ветвей посылается слагаемое на узел Cnt. Поскольку в данном случае каждый приход токена вызывает ровно одну активацию, то число слагаемых равно $n = n_a + n_b$. Мы посылаем 0 с первой ветви и 1 – с двух других ветвей, чтобы сумма в конце концов стала равной в точности числу элементов результата $n_c = n_a + n_b - m$, где m – число срабатываний 1-й ветви, то есть число фактических сложений.

В описании суммирующего входа *k* указано количество слагаемых n, которое задано одноименным входом. Когда на узел Cnt придет токен *n* и *n* токенов на суммирующий вход *k*, узел активируется и пошлет значение суммы *k* дальше в качестве числа элементов результата. Но нам еще нужно «разморозить» непарные элементы векторов a и b, которые пока остаются на входе a или b узлов AddSV⟨i⟩ в ожидании своей пары. Для этого узел Cnt посылает сигнальный токен на дополнительный вход *s*. Это «глобальный» токен, один на все узлы AddSV⟨i⟩. В нем индекс задан как ⟨*⟩. Реакция на него задана в последних двух ветвях. Пунктир вокруг входа *s* в заголовке указывает на то, что токен на нем не будет сохраняться, и потому он может взаимодействовать только с другими токенами на входах *a* или *b*, пришедшими ранее, и после всех взаимодействий он будет удален.

При распределенном вычислении элементы с индексом *i* поступают в процессор *place*(*i*), где и выполняется активация узла AddSV⟨i⟩. Предполагается, что они поступают в случайном порядке, и что каждый отдельный элемент может в принципе задержаться на неопределенное время – это может задержать его результат, но не должно нарушать работу алгоритма в целом. Парные элементы смогут сразу породить элемент результата и выдать его дальше. Непарные элементы будут вынуждены дожидаться момента, когда станет ясно, что все элементы получены. Это можно узнать только после получения всех элементов всеми процессорами и подсчета их количества узлом Cnt. После достижения нужного числа слагаемых, сумма передается как результат и порождается глобальный токен на вход *s* узла AddSV. Поскольку в индексе целевого узла в нем стоит *, то копия этого токена посылается во все процессоры.

Узел Cnt не имеет индекса, поэтому формально он находится в каком-то одном процессоре. И на этот процессор должны будут прийти $n_a + n_b$ токенов-слагаемых для входа *k*. Возникает проблема «узкого горла». Для ее решения можно использовать технику «активных сообщений»: слагаемые с общим целевым адресом накапливаются и суммируются (агрегируются) перед отправкой. Это может значительно снизить нагрузку на входе целевого узла.

Другой вариант решения – распределенный счетчик. Узел Cnt распределяется на все процессоры, при этом система контролирует достижение заданного числа слагаемых глобально. После этого запускается, с одной стороны, суммирование локальных сумм по двоичному дереву (фактически это уже сделано контролирующей системой), а с другой стороны, в каждом процессоре локально создается копия глобального токена на вход s. Более продвинутый исполнитель (компилятор) может перейти на распределенное представление количеств элементов n_a, n_b, n_c, что позволит полностью исключить нелокальные взаимодействия.

VI. Автоматическое распараллеливание

Задача автоматического распараллеливания обычно ставится как задача преобразования исходного 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 расширениях. Метод состоит в том, что сначала для программы строится ее так называемая полиэдральная модель [5], которая анализируется (на предмет построения функций *place* и *stage*) и затем отображается в параллельный код.

Оказывается [6], 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 википедия (в статье для "Visual prorgamming language" [7]):

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, вероятно, станет одной из самых больших проблем программирования будущего).

VII. Аналоги

Существует огромное число работ как по языкам и системам потока данных, так и языкам в графической форме. Оба свойства сочетает уже упомянутая система LabVIEW.

Важным аналогом также является парадигма параллельного программирования, известная под такими названиями, как модель акторов или агентов [8], Active Messages (AM) [9], polyphonic C# [10], Charm++ [11]. Их объединяет объектный подход, состоящий в том, что в распределенном пространстве независимо действуют объекты, посылая друг другу сообщения. Объекты имеют состояния, и обработка сообщения состоит в изменении состояния и посылке новых сообщений. Программа заключается в реакциях на все виды сообщений.

В языке DFL-G узлы не имеют состояний, но узел принимает и реагирует не на отдельные сообщения, а сразу на несколько. В частности, это позволяет один из входов рассматривать как состояние, посылая на него новое значение после каждой обработки. Тем самым отсутствие явных состояний не является принципиальным ограничением. А возможность обрабатывать сразу несколько сообщений, особенно по неполному набору входов, дает дополнительный выразительный механизм структурирования обработки. Язык Polyphonic C# также содержит подобный механизм.

Еще одно важное различие состоит в способах адресации. В объектном подходе, надо сначала объект создать (привязав к процессору), сохранить ссылку на него и только потом по ссылке посылать сообщения. Сложные сети объектов образуются посредством хранения ссылок внутри объектов. У нас узлы имеют явную адресацию посредством индексов, возможно многомерных. Чтобы послать токен, не надо иметь ссылку, надо просто скомпоновать индекс, причем экземпляр узла физически будет создан по приходу первого токена на любой из его входов. В некоторых вариациях объектного подхода [8] возможность индексировать и создавать объект автоматически по первому сообщению также имеется.

Мы не рассматриваем названные системы как конкурентов. Скорее наоборот, наш язык, будучи хорошо инструментированным, мог бы стать удобной надстройкой для прототипирования и облегчения программирования в этих системах.

VIII. Заключение

Предложенное расширение языка потока данных средствами индексирования узлов и токенов значительно повышает выразительные возможности языка. В частности, оно позволяет естественно и компактно выражать обработку сложных структурных данных, такие как массивы, ключевые множества (хеш-таблицы), разреженные вектора и матрицы и т.п. Также оно дает удобное средство эффективного управления распределением вычислений по пространству и времени.

Практика экспериментального использования этого языка показала, что в нем удобно и компактно выражаются довольно сложные алгоритмы. Среди них разные методы сортировки, разностные методы, задачи линейной алгебры, быстрое преобразование Фурье, молекулярная динамика, SAT-задача, ряд важных алгоритмов обработки графов (поиск вширь, нахождение кратчайшего пути, построение остовного дерева, разбиения на сообщества).

Особо следует отметить заключенный в этом языке потенциал осуществления семантического анализа и эквивалентных преобразований, на основе которого могут быть построены инструменты для извлечения из заданных на этом языке алгоритмов эффективных программных кодов для различных иных аппаратно-программных платформ. Этот подход можно назвать "автоматическим распараллеливанием наоборот", поскольку мы начинаем не с последовательной программы, а наоборот, с предельно параллельной и мелкозернистой, и выводим из нее другие программы, менее параллельные, после наложения ряда дополнительных ограничений на управление порядком вычислений, распределение вычислений и т.п.

Литература

[1] Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990.

[2] National Instruments site. URL http://russia.ni.com (дата обращения 04.04.2016).

[3] LabVIEW. URL: http://labview-rus.blogspot.ru/ (дата обращения 04.04.2016).

[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: электрон. научн. журн. 2014. Т. 5. № 1(19). С. 15–36. URL: http://psta.psiras.ru/read/psta2014_1_15-36.pdf (дата обращения 04.04.2016).

[5] P. Feautrier. Array dataflow analysis. In: Compiler Optimizations for Scalable Parallel Systems. LNCS, vol. 1808, pp. 173–219, Springer, Heidelberg (2001).

[6] A.V. Klimov. Construction of exact polyhedral model for affine programs with data dependent conditions. In: Proc. of the Forth International Valentine Turchin Workshop on Metacomputation, META'14, (June 29 – July 3, 2014), pages 136–160. University of Pereslavl, 2014.

[7] Visual programming language. URL: https://en.wikipedia.org/wiki/Visual_programming_language (дата обращения 04.04.2016).

[8] 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.

[9] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. AM++: A generalized active message framework. In PACT'10, NY, USA, 2010, pages 401–410.

[10] Introduction to Polyphonic C#. URL: http://research.microsoft.com/en-

us/um/people/nick/polyphony/intro.htm (дата обращения 04.04.2016).
[11] 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.

A graphical dataflow meta-language for asynchronous distributed programming

A.V. Klimov, A.S. Okunev

Institute for Design Problems in Microelectronics of RAS, klimov@ippm.ru, oku@ippm.ru

Keywords **— graphical programming, asynchronous distributed computing, meta-language, dataflow computation model, indexing, information structure of algorithm, LabVIEW, Active Messages, scatter paradigm, gather paradigm.**

Abstract

There exists a variety of programming paradigms for development of parallel software. Each paradigm dictates its own requirements on the structure of the algorithm, which in practice means the need to develop an algorithm within each paradigm almost from scratch. We propose a new language in which one will be able to write an algorithm just once, and then to systematically derive efficient code in different programming paradigms. The language is intended for writing asynchronous distributed algorithms and is based on the dataflow principle: an operation fires when all its arguments are ready.

We have extended the classical dataflow paradigm by adding a node indexing, which radically changed the very style of programming. We also propose a graphical form of the language, which makes it easier to learn and use. The language is explained by simple examples. We show how indexing helps to express complex data structures, compound operations, allows to flexibly manage the distribution of computations.

The concepts of *scatter* paradigm and *gather* paradigm are introduced, which help to classify programming languages and styles. In gather paradigm the consumer points out the sources of its input arguments, while in less familiar scatter paradigm inherent to our language the producer of a data points out all its consumers. We claim that scatter paradigm better fits the domain of distributed programming.

The paper also provides comparison of our project with existing systems and approaches including LabVIEW and Active Messages.

References

[1] Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990.
[2] National Instruments site (in Russian). Available at: http://russia.ni.com (accessed 04.04.2016).
[3] LabVIEW (in Russian). Available at: http://labview-rus.blogspot.ru/ (accessed 04.04.2016).
[4] A.V. Klimov, N.N. Levchenko, A.S. Okunev, A.L. Stempkovsky. Supercomputers, memory hierarchy and dataflow computation model. // “Programmnye systemy: teoriya i prilozheniya”: Electronic Sci. J., V.5, No.1 (19), pages 15-36 (2014) (in Russian). Available at: http://psta.psiras.ru/read/psta2014_1_15-36.pdf (accessed 04.04.2016)
[5] P. Feautrier. Array dataflow analysis. In: Compiler Optimizations for Scalable Parallel Systems. LNCS, vol. 1808, pp. 173-219, Springer, Heidelberg (2001).
[6] A.V. Klimov. Construction of exact polyhedral model for affine programs with data dependent conditions. Proc. of the Forth International Valentine Turchin Workshop on Metacomputation, META'14, (June 29 – July 3, 2014), pages 136--160. University of Pereslavl, 2014.
[7] Visual programming language. Available at: https://en.wikipedia.org/wiki/Visual_programming_language (accessed 04.04.2016).
[8] 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.
[9] J.J. Willcock, T. Hoefler, N.G. Edmonds, and A. Lumsdaine. AM++: A generalized active message framework. In PACT'10, NY, USA, 2010, pages 401–410.
[10] Introduction to Polyphonic C#. Available at: http://research.microsoft.com/en-us/um/people/nick/polyphony/intro.htm (accessed 04.04.2016).
[11] 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.