ИПМ им.М.В.Келдыша РАН

Абрау-2019 • Труды конференции

Арк.В. Климов

О возможности генерации параллельных программ по спецификации графа алгоритма

Рекомендуемая форма библиографической ссылки

Климов Арк.В. О возможности генерации параллельных программ по спецификации графа алгоритма // Научный сервис в сети Интернет: труды XXI Всероссийской научной конференции (23-28 сентября 2019 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2019. — С. 432-449. — URL: http://keldysh.ru/abrau/2019/theses/88.pdf doi:10.20948/abrau-2019-88

Размещена также презентация к докладу

О возможности генерации параллельных программ по спецификации графа алгоритма

Арк. В. Климов

Институт проблем проектирования в микроэлектронике (ИППМ) РАН

Аннотация. Для решения актуальной задачи автоматизации параллельного программирования предлагается язык UPL для записи M-графа алгоритма и намечается путь для получения на его основе в автоматическом режиме программ для различных параллельных платформ. Для этой цели пользователь должен задать дополнительно функции распределения, отображающие вычисления на ресурсы платформы. Описывается алгоритм порождения гнезда циклов по M-графу алгоритма и комбинированной функции распределения и на простом примере демонстрируется его работа.

Ключевые слова: параллельное программирование, потоковая модель вычислений, граф алгоритма, язык представления графа, функция распределения, автоматизация порождения программ.

On the possibility of parallel programs synthesis from algorithm graph specification

Ark. V. Klimov

Institute of Design Problems in Microelectronics (IPPM) of RAS

Abstract. To solve the actual problem of automated parallel programming, the UPL language is proposed for specifying the M-graph of the algorithm and a roadmap for generating programs on its basis for various parallel platforms in automatic mode is presented. For this purpose, the user must specify in addition the distribution functions that map calculations to the platform resources. The algorithm for generating a loop nest by the M-graph of the algorithm and the combined distribution function is described, and its operation is demonstrated by a simple example.

Keywords: parallel programming, dataflow computation model, algorithm graph, graph representation language, distribution function, automated program synthesis.

1. Введение

В настоящее время достижение эффективности программ невозможно без параллелизма. Но, в отличии от классического последовательного вычислителя фон-неймановского типа, до сих пор так и не сложилось какой-либо единой для

всех параллельных вычислителей модели вычислений и, соответственно, языка программирования, который мог бы с приемлемой эффективностью (как язык C) автоматически транслироваться на все существующие платформы параллельных и распределенных вычислений. Хуже того, возникло много разных типов платформ (OpenMP, MPI, CUDA, TBB, ..., FPGA), для каждой из которых приходится радикально перестраивать весь алгоритм.

Как ответ на эту ситуацию появился проект AlgoWiki [1], в котором заявлена цель представить все основные известные вычислительные алгоритмы так, чтобы сначала описать алгоритм в некотором абстрактном, не зависящем от платформ виде, а затем во второй части описать особенности различных программных реализаций для разных платформ. Так описано уже более сотни различных алгоритмов.

Для абстрактного представления алгоритмов введено понятие графа алгоритма, в котором явно выражены информационные связи между вычислительными узлами: в узлах производятся вычисления, входы узлов связаны дугами[1] с выходами узлов или входными узлами, а выходы связаны с входами узлов или выходными узлами.

К сожалению, для описания графа абстрактного алгоритма в проекте AlgoWiki не предлагается никакого формализованного языка, чтобы можно было ставить вопросы о верификации и дальнейших преобразованиях алгоритмов, включая генерацию программ для различных платформ. Здесь мы попытаемся восполнить этот пробел. Для этого, уточнив понятие графа алгоритма (раздел 2), предложим язык представления графа алгоритма UPL (раздел 3). Этот язык может рассматриваться как язык программирования, имеющий полноценную семантику, позволяющую его выполнять как алгоритм на некоторой абстрактной машине (раздел 4). Для превращения графа в эффективно работающий код нам нужно прежде всего уметь распределять граф вычислений по пространству (placement, раздел 5) и времени (scheduling, раздел 6). В разделе 7 вводится понятие объединенной функции распределения, а в разделе 8 описывается общий метод порождения кода по ней. Метод иллюстрируется на простом примере. В отличие от многих работ и систем по автоматическому распараллеливанию, где эти распределения порождаются автоматически, мы рассматриваем их как инструмент управления трансляцией: задавая эти функции, пользователь управляет порождением кода, а его эффективность зависит от выбора пользователя. В статье мы пытаемся показать, что функция распределения как понятие не только позволяет автоматически порождать работающий согласно ей код, но и является удобным и адекватным средством управления работой порождаемого кода.

[1]Мы используем термины «узел» и «дуга» как аналоги терминов «вершина» и «ребро» в абстрактной теории графов.

2. Понятие графа алгоритма

Прежде всего, обратим внимание на двусмысленность, связанную с понятием графа алгоритма. Следует различать граф вычисления, в котором узлами является отдельные вычислительные акты, и собственно граф алгоритма, который в компактной форме описывает всевозможные графы вычислений, возникающие при работе данного алгоритма с конкретными входными данными. Первый может быть огромным, тогда как второй обычно может уместиться в одну или несколько страниц. Именно второй, на наш взгляд, заслуживает название «*граф алгоритма*», поскольку для данного алгоритма он один. Первый же будем именовать *вычислительный граф* (алгоритма). Он определяется не только алгоритмом, но и его входными данными, по крайней мере их частью, состоящей из целочисленных параметров, задающей размеры массивов. Для краткости и определенности, мы будем называть его *R-граф* (run-time граф, развернутый граф, а также – *решетчатый граф* в терминологии Ростовской школы [3]). А собственно граф алгоритма будем именовать *M-граф* (макро-граф, мета-граф и т.п.). Во избежание двусмысленностей узлы R-графа будем также называть *виртуальными*[2] *узлами*. В литературе также встречаются термины *граф информационных зависимостей* или *информационный граф*. Под ними может пониматься как M-граф, так и R-граф в зависимости от контекста.

M-граф может быть построен автоматически по фрагменту программы на языке C или Фортран, если фрагмент относится к линейному классу. Такой фрагмент может содержать только циклы и операторы присваивания, где все обращения к массивам и границы циклов являются линейными выражениями от индексов объемлющих циклов и фиксированных *структурных* параметров (например, `2*i+j`, `N-i`). Также допускаются условные операторы с линейными условиями (типа `i<j`, `i==N`). В рамках линейных выражений можно допускать также операцию деления нацело на целочисленную константу. В англоязычной литературе такого рода фрагмент называют Static Control Program (SCOP) а соответствующий ему M-граф – *полиэдральной моделью* [12].

Каждый узел X полиэдральной модели (M-графа) возникает из некоторого оператора присваивания (переменной или элементу массива) и характеризуется:

- уникальным именем (образованном по имени целевого массива),
- набором индексов $I=(i_1,i_2,...,i_k)$, соответствующим параметрам объемлющих циклов,

[2]Поскольку мы можем заранее не знать, какие из виртуальных узлов появятся в конкретном варианте R-графа. Поэтому они все априори считаются *виртуально* существующими.

- доменом D_X (множеством значений набора индексов), заданным кусочно-квази-аффинным[3] (ККА) многогранником [4], зависящим (линейно) от структурных параметров,
- набором входов (портов) $V=(v_1,v_2,...,v_p)$, соответствующим обращениям к массивам в правой части,
- вычислительной программой, построенной по правой части.

Из каждого узла X в M-графе (не считая выходных узлов) исходит одна или несколько дуг. Каждая дуга идет в некоторый входной порт некоторого узла Y. Каждая дуга характеризуется:

- передаваемым значением, вычисленным программой узла X,
- индексом узла-получателя, вычисляемым отправителем по заданной формуле $\varphi(I,V)$, зависящей от собственных индексов I и входов V,
- условием выдачи (а в общем случае – генератором множества выходных значений).

На возможности извлечения M-графа из последовательной программы построены многие системы распараллеливания, использующие обычный последовательный C или Фортран в качестве входного языка (OPC [2], PLUTO [11]). Мы полагаем, однако, что использование в качестве отправной точки единого универсального языка представления самого M-графа будет более плодотворным, в частности, поскольку отображения удобнее задавать в терминах узлов и их индексов, а не параметров циклов, операций и массивов.

R-граф может быть построен по M-графу, если задать необходимые входные данные. Обычно для этого достаточно задать только структурные параметры, такие как N и M на Рис.2 (в частности, если индексы получателей на дугах зависят только от собственных индексов I, но не от входов V, что выполняется для полиэдральных моделей). При этом каждый узел X в M-графе порождает семейство виртуальных узлов $X\langle I\rangle$ в R-графе, где $I\in D_X$. Каждая дуга в M-графе (идущая из узла X в порт P узла Y) породит семейство дуг в R-графе, из которых каждая соединяет, если выполнено соответствующее *условие*, свой экземпляр узла $X\langle I\rangle$ с портом P экземпляра узла $Y\langle J\rangle$, где $J=\varphi(I,V(X\langle I\rangle))$.

Заметим, что в R-графе нет циклов – иначе какой-то узел получает на вход данные, зависящие от его же результата. Но M-граф обычно содержит какие-то циклы, при этом на них индексы меняются так, чтобы в порожденном R-графе циклов не было (как, например, на дуге с условием $t<M$ на Рис.2 слева). Данное условие равносильно тому, что каждый узел R-графа активируется (см. ниже) не более одного раза.

[3]Приставка «квази-» указывает на то, что в определяющих формулах было использовано деление на число.

3. UPL – язык представления М-графа

Для записи М-графа используется разработанный автором графический язык UPL (Universal Parallel Language) [5,6]. Он может рассматриваться как обобщение языка полиэдральных моделей [12], в котором нет требования на линейность (квази-аффинность) используемых выражений для индексов-получателей и не требуется спецификация домена.

В качестве иллюстрации и рабочего примера рассмотрим задачу решения уравнения теплопроводности в одномерном случае классическим методом с использованием простой трехточечной схемы. Соответствующий код на языке С приведен на рис. 1.

```
for(int t=1; t<=M; t++){
  for(int i=1; i<N; i++)
    B[i] = 0.33333*(A[i-1]+A[i]+A[i+1]);
  for(int i=1; i<N; i++)
    A[i] = B[i];
};
```

Рис. 1. Программа Heat1 на языке С

По этому коду автоматически строится М-граф, в графической форме показанный на рис. 2 (слева). Он получен в предположении, что $M > 1$ и $N > 2$, входом является весь массив `A[0:N]`, и он же является выходом. Данный М-граф соответствует программе Heat1 с точностью до передач через *транзитные* узлы, такие как узел A (они имеют единственный вход, передаваемый на выход без изменений).

На рис. 2 справа приведен соответствующий R-граф для N=5 и M=3. Все узлы здесь имеют индекс из одного или двух компонентов – по числу циклов, объемлющих соответствующий оператор. При записи виртуальных узлов в тексте индексы записываются в угловых скобках, а на схеме – в шестиугольниках. Обычный узел изображается прямоугольником с одним или несколькими именованными входами, имеющими форму усеченного желтого круга на краю прямоугольника. Транзитный узел изображаются желтым овалом. В узлах индексы объявлены в розовых шестиугольниках. На каждой дуге имеется пометка со списком индексных выражений на фоне голубого шестиугольника – это индекс целевого узла. В начале дуги может стоять условие выдачи. На дуге также может стоять формула для вычисления значения, либо она стоит в узле отправителе, либо это значение первого (или единственного) входа.

Все индексные выражения, стоящие на дугах, вычисляют индекс получателя, опираясь на контекст отправителя. Мы говорим, что М-граф выполнен в *парадигме раздачи*, в том смысле, что каждый узел знает, куда и кому следует отдать вычисленное им значение. Если все стрелки обратить, сменив в них выражения на вычисляющие индексы отправителя в контексте получателя, то получится граф в *парадигме сбора*, где каждый узел (элемент

вычислений) знает, где взять нужные ему параметры, но не знает куда передавать результаты – он просто должен положить их в свой выходной массив данных под своим индексом, а кому надо сам их оттуда заберет. Для полиэдральных моделей такое преобразование всегда возможно, а в общем случае не всегда[4]. Парадигма раздачи, в принципе, лучше соответствует нуждам параллелизма и распределенности, поскольку в ней по дугам производится односторонняя передача данных, тогда как в парадигме сбора данные передаются как бы «в ответ на запрос».

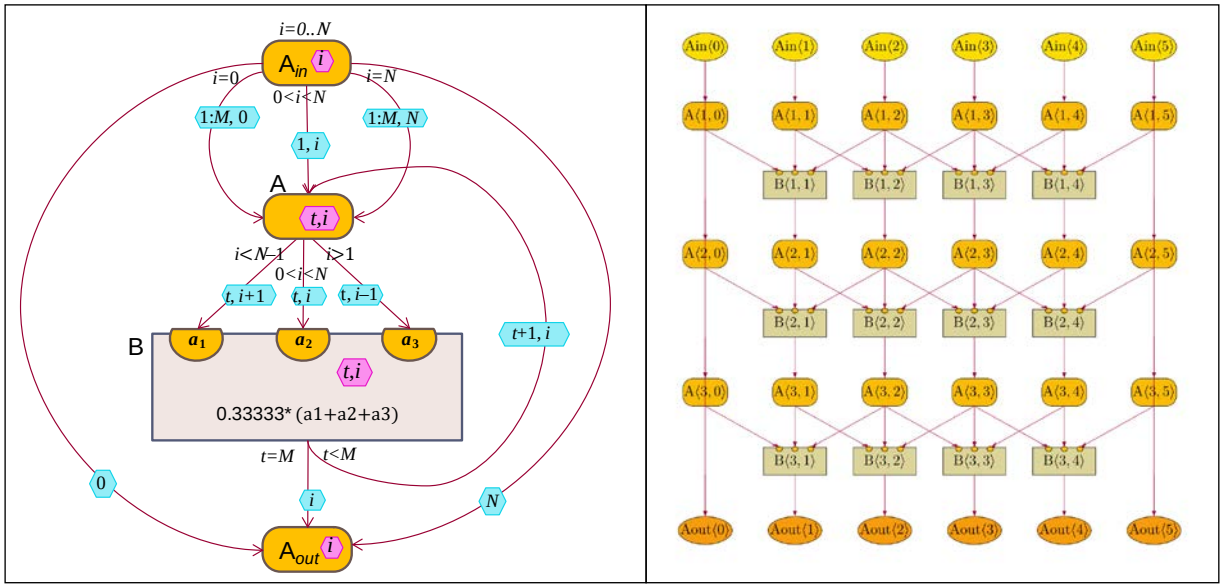

Рис.2. Граф алгоритма Heat1: слева – собственно граф алгоритма, или M-граф, справа – вычислительный R-граф для N=5, M=3

Заметим, что в M-графе утрачена информация о том, как информация была размещена в массивах (здесь A и B). Вместо этого появились «массивы узлов» и связанные с ними «массивы входов и выходов». Забегая вперед, укажем, что исходная последовательная программа может быть восстановлена (с точностью до мелочей) по подходящей функции распределения по времени.

4. Потоковая модель вычислений

M-граф может рассматриваться как самостоятельная программа, в которой достаточно информации для ее вычисления. При этом используется потоковая модель вычислений, в которой происходит активация узлов по мере готовности данных на его входах. Процесс выполнения M-графа может рассматриваться как процесс порождения соответствующего R-графа. Сам R-граф может трактоваться как *трасса* выполнения M-графа при некоторых

[4]В литературе по полиэдральным моделям основным считается представление в парадигме сбора.

входных данных. И наоборот: всякому вычислению М-графа соответствует определенный R-граф.

Данные (как начальные, так и порождаемые внутри) поступают в виде *токенов*, каждый из которых несет в себе значение (элемент данных) и адрес, состоящий из имени целевого узла, имени его входа и набора значений индексов. Когда на все входы определенного экземпляра узла (с конкретными индексами) поступят токены, данный экземпляр активируется, в результате чего будет исполнена его программа. В программе узла вычисляются его выходные значения и формируются новые токены, посылаемые по исходящим дугам на входы других узлов. Индексы целевых узлов также вычисляются в программе узла.

М-граф программы Heat1 работает следующим образом. На входные узлы Ain⟨i⟩ поступают токены, содержащие начальное значение A[i], i=0..N. Каждый узел Ain⟨i⟩ подает свое значение на транзитный узел A⟨1,i⟩, но крайние узлы, при i=0 или i=N, – сразу на все A⟨t,i⟩ (в позиции для t можно написать * или диапазон 1:M). Каждый узел A⟨t,i⟩ передает свой аргумент на три входа трех соседних узлов B (крайние – лишь на 1 или 2 из них). Кроме того, элементы Ain⟨0⟩ и Ain⟨N⟩ подаются также на крайние выходные узлы Aout⟨0⟩ и Aout⟨N⟩. Каждый узел B⟨t,i⟩, получив три входных значения, вычисляет результат по формуле и при t<M передает его на узел A⟨t+1, i⟩. А при t=M – на выходной узел Aout⟨i⟩.

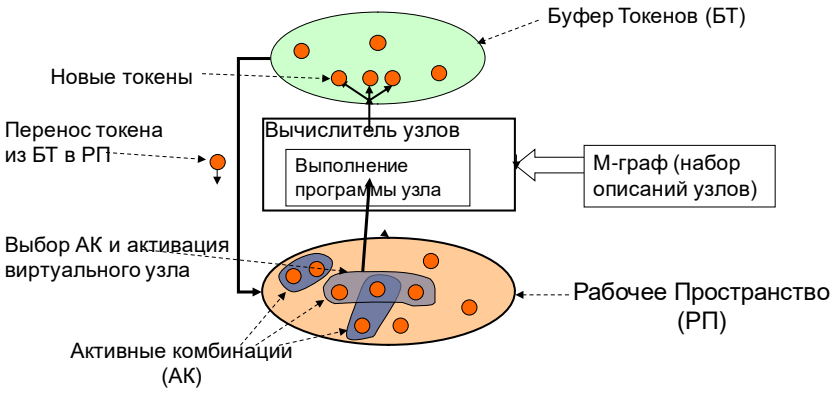

Рис.3. Абстрактный вычислитель

Абстрактный вычислитель, который выполняет описанную работу, содержит запоминающее устройство, называемое Рабочее Пространство (РП), в которое поступают токены (см. Рис. 3). В нем по определенным правилам отыскиваются *активные комбинации* – наборы токенов направленные на входы одного виртуального узла, по одному на каждый вход. Шаг работы вычислителя состоит в выборе одной активной комбинации в РП и в активации соответствующего виртуального узла. При этом использованные токены одномоментно (атомарно) удаляются из РП (или, если у токена кратность была больше 1, то из нее вычитается 1), после чего на Вычислитель Узлов (ВУ)

передается пакет (задания на выполнение программы узла) со значениями входов и индексов. В процессе выполнения могут могут быть порождены новые токены, которые через Буфер Токенов (БТ) с какой-то задержкой передаются (по одному) в РП. Работа заканчивается, когда ВБ пуст, и РП не содержит ни одной АК. Токены, оставшиеся в РП, образуют результат работы.

Описанный абстрактный вычислитель позволяет произвести вычисление по М-графу. Его можно реализовать либо аппаратно [8], либо в режиме эмуляции на одно- или много-процессорной системе [9]. Это удобно для отладки и исследования алгоритма, но может быть не эффективно из-за высоких накладных расходов (на передачу токена в РП, поиск АК, активацию, передачу пакета в ВУ и т.п.). Поэтому было бы полезно иметь альтернативные механизмы отображения М-графа в эффективные программы на существующие (супер-)ЭВМ. Для этого может потребоваться дополнительная информация. Ниже рассмотрим возможный механизм такого отображения на основе информации о распределении виртуальных узлов по пространству и времени.

5. Распределение вычислений по пространству

Абстрактный вычислитель языка UPL, работу которого мы неформально описали выше, обладает замечательным свойством – он может быть распределенным.

Пусть у нас есть P вычислителей, связанных коммуникационной сетью. Будем называть их процессорами или ядрами. Пусть для каждого узла М-графа X задана функция распределения Π_X, которая каждому набору индексов I узла ставит в соответствие некоторый номер ядра $p = \Pi_X(I)$ от 0 до P–1. При порождении каждого нового токена, направляемого на узел $X\langle I\rangle$, будем вычислять соответствующую функцию $\Pi_X(I)$ и направлять токен в ядро с полученным номером. Понятно, что токены, направленные на общий виртуальный узел $X\langle I\rangle$, попадут все в какое-то одно ядро с номером $\Pi_X(I)$ и там активируют узел $X\langle I\rangle$. А значит, все будет работать так же, как если бы использовалось одно единственное ядро[5].

В модели вычислений есть два вида недетерминизма выполнения: 1) при выборе токена из БТ для переноса в РП и 2) при выборе очередной АК для активации, когда есть несколько готовых АК. Распределенное выполнение позволяет обратить этот недетерминизм в параллелизм. Такое обращение корректно, если результат параллельного выполнения совпадает с результатом какого-либо варианта выбора порядка последовательного выполнения. В данной модели так и будет, поскольку разные виртуальные узлы активируются независимо друг от друга.

[5] Возможны проблемы с токенами, направленными сразу на много виртуальных узлов (как, например, токены с диапазоном 1:M), для их разрешения могут накладываться некоторые ограничения.

С учетом возможных ограничений можно считать, что функцию П можно выбирать произвольно: программа (М-граф) в любом случае будет работать корректно, разве что затраты на передачу токенов и объемы простоев из-за неравномерности загрузки могут различаться. Поэтому надо стараться выбирать функцию распределения П так, чтобы объем передач между ядрами был поменьше. И не только объем: важно также, чтобы на критических путях (цепочках передающих друг другу данные виртуальных узлов) было меньше передач между ядрами, создающими большие задержки. При наличии иерархии в структуре сети потребуется оптимизировать на всех уровнях. Также важен баланс загрузки. Все три фактора: объем передач, задержки на критических путях и баланс загрузки – тесно взаимосвязаны и при выборе функции П должны оптимизироваться совместно.

6. Планирование вычислений – распределение по времени

В соответствии с потоковой моделью вычислений каждый виртуальный узел может выполняться, когда доставлены все его аргументы. Будем говорить, что виртуальный узел Y *зависит* от виртуального узла X, обозначая это как $X \rightarrow Y$, если X создает токен, используемый в активации Y. Транзитивное замыкание этого отношения (обозначаемое $\twoheadrightarrow$) создает *базовый* (строгий) частичный порядок на множестве всех виртуальных узлов, которые возникают в данной задаче. Линейный или частичный порядок ($<$) на множестве виртуальных узлов является *допустимым*, если он уточняет базовый порядок, то есть $X \rightarrow Y \Rightarrow X < Y$.

Если порядок линейный, то он задает строго последовательный обход (выполнение) графа. Если порядок частичный, то он оставляет место для параллелизма. Таким образом, фиксация порядка налагает ограничение на программу, вычисляющую М-граф.

Подобно тому, как распределение по ядрам, то есть по *пространству*, можно задать функцией распределения П, порядок вычислений можно задать функцией распределения по *времени* Θ, которая также зависит от узла и его индексов и выдает номер этапа, на котором этот узел надо выполнять.

Составление законченной программы обхода R-графа (последовательной или параллельной) тоже является способом описания порядка его вычислений. Но нам представляется, что составить программу обхода на некотором языке программирования человеку сложнее, чем выписать функцию распределения. Именно поэтому мы считаем актуальной задачу автоматической генерации программы по заданным функциям распределения по пространству и времени.

7. Объединенная функция распределения

Часто может рассматриваться многомерное пространство процессоров или тредов, когда процессоры расположены в узлах многомерной решетки, и номер каждого задается вектором целых чисел. В этом случае функция П

вырабатывает векторные значения. Аналогично время можно считать многомерным, задавая функцию Θ как векторную. В таком случае подразумевается лексикографический порядок. Мы вводим комбинированную многомерную функцию Ψ, все компоненты которой поделены на временные и пространственные. Последние будем выделять подчеркиванием.

На множестве значений функции Ψ определим *частичный лексикографический порядок* ≻ следующим образом. Для сравнения двух векторов надо найти первый (слева) компонент, в котором они различаются. Если этот компонент временной, то значения сравнимы и больше тот вектор, у которого этот компонент больше. В противном случае значения несравнимы (равные также считаем несравнимыми).

Функция Ψ (как и функция Θ) должна быть монотонной. Формально:

$$A\langle I\rangle \rightarrow B\langle J\rangle \Rightarrow \Psi_B(J) \succ \Psi_A(I).$$

Соблюдение этого требования необходимо верифицировать, прежде чем использовать функцию Ψ для генерации кода.

8. Порождение программы по функции распределения

Рассмотрим работу предполагаемого генератора кода на примере той же задачи Heat1. На Рис.4. показан R-граф задачи, размеченный значениями функций Π=*p*(*t*,*x*) и Θ=*s*(*t*,*x*). Функции *p* и *s* выбраны так, чтобы внутренний цикл проходил по блокам (плиткам), показанным на рисунке. Значения (*p*,*s*) указаны на плитках. Пример взят из нашей работы [7], где исследуется вариант (а). При таком способе обхода производительность многократно повышается за счет более эффективного использования кэша.

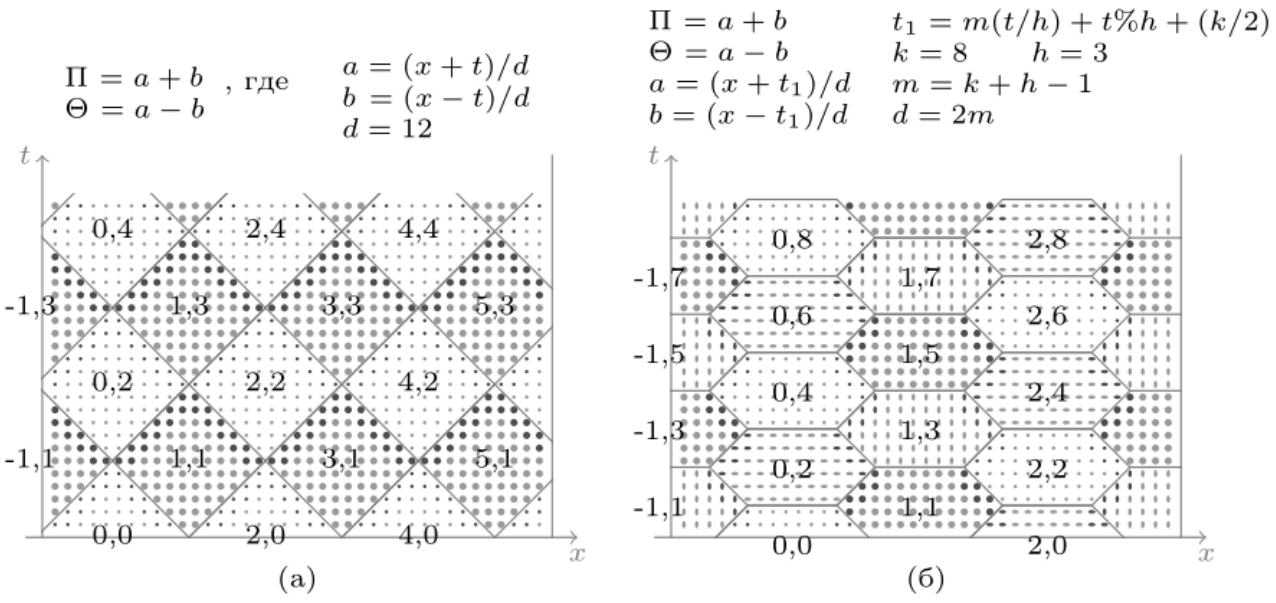

Рис.4. Варианты распределений узлов сетки по пространству и времени в задаче Heat1

Теперь рассмотрим вариант (б), порожденный следующими функциями:

$$\Pi(t,x) = p(t,x) = (x+t_1)/d + (x-t_1)/d,$$
$$\Theta(t,x) = s(t,x) = (x+t_1)/d - (x-t_1)/d, \quad (1)$$
где
$$t_1 = m(t/h) + t\%h + k/2,\ d = m,\ m = k + h - 1,\ k = 8,\ h = 3$$

Здесь / и % – целочисленное деление для получение частного и остатка соответственно, остаток имеет знак делителя. От варианта (а) он отличается «заменой времени» $t \leftarrow t_1$. Ее смысл в том, что как бы стираются горизонтали, проходящие через углы «ромбов», и еще несколько соседних, так что остаются полоски ширины h (то есть h это высота элементарных «трапеций», а m – их средняя линия). Этот вариант мотивируется тем, что, в отличие от варианта (а) в нем не будет коротких циклов по x.

Рассмотрим объединенную функцию распределения[6]

$$\Psi_B(t,x) = \langle \Theta, \underline{\Pi}, t, x \rangle.$$

По ней порождается гнездо циклов вида:

```
int s,p,t,x;
for (s=□;s<□;s++)
  #pragma omp parallel for
  for (p=□;p<□;p++)
    if (exist(s,p))
      for(t=□;t<□;t++)
        for(x=□;x<□;x++)
          B(t,x)=0.33333*(B(t-1,x-1)+B(t-1,x)+B(t-1,x+1));
```

Здесь значком □ помечены места для границ циклов, которые предстоит определить. Перед телом каждого цикла в общем случае может стоять проверка предиката непустоты exist, но здесь он только в одном случае будет отличен от **True**, а именно, блок для индексов (s,p) существует только тогда, когда s и p имеют одинаковую четность.

Для определения границ циклов рассмотрим пространство итераций

$$D = \{ (t,x) \mid 1 \leqslant t \leqslant M,\ 1 \leqslant x < N \}$$

и его образ под действием функции Ψ:

$$\Psi(D) = \{ (s,\underline{p},t,x) \mid 1 \leqslant t \leqslant M,\ 1 \leqslant x < N,\ p=p(t,x),\ s=s(t,x) \}. \quad (2)$$

Это квази-аффинный многогранник в пространстве I^4.

Наш алгоритм построения гнезда циклов построен в духе работы [10], но с некоторыми модификациями. В качестве переменных циклов обхода многогранника D выступают символьные обозначения выражений компонентов $\Psi(D)$. Здесь это $s,\underline{p},t,x$. В такой последовательности и будут расположены вложенные циклы (возможна перестановка рядом стоящих подчеркнутых символов).

[6] Имеет смысл только функция для узла B, поскольку остальные узлы транзитные.

Поскольку многогранник *D* задан системой квази-аффинных неравенств, для определения границы цикла для переменной, например *t*, достаточно выбрать неравенство с нужным направлением, в которое входят только эта и более левые переменные, в данном случае – *s*,*p*,*t*. Для получения таких неравенств необходимо спроектировать многогранник *D* на подпространство значений этих переменных. Это можно сделать с помощью алгоритма Фурье-Моцкина последовательного исключения «лишних» переменных. Обозначим его как

D_{vars} = Fourier(*D*,*vars*)

где *vars* – список остающихся переменных.

Его результат D_{vars} может быть аппроксимацией «сверху». Он также представлен в виде системы аффинных неравенств. Если есть несколько неравенств, подходящих для выбора границ, то можно взять любую их непустую часть, выразив из них выбранную переменную через остальные, используя max или min. Оставшиеся неравенства будут использованы внутри циклов в форме ограничивающих условий. Здесь остается простор для оптимизации.

В алгоритме также используется функция ElimRedund(*CTX, D*), которая удаляет избыточные ограничения в *D* при условии *CTX* (те, которые являются следствиями условий из контекста *CTX* и остальных условий из *D*).

Алгоритм построения гнезда циклов

Дано:

pars – список статических параметров;

vars = $v_1,v_2,...,v_k$;

D (квази-аффинная система неравенств, зависящая от *pars*+*vars*);

Результат:

разложение $D = D_1+D_2+...+D_k$, где каждое D_i зависит только от переменных $v_1, ..., v_i$ (и параметров из списка pars). Каждое D_i содержит хотя бы одно неравенство, ограничивающее сверху/снизу переменную v_i.

начало

CTX := список ограничений на параметры *pars*.

для всех *i* **от** 1 **до** *k*

vs = start(*vars*, *i*) ; //– первые i переменных списка *vars*

D_{vs} := Fourier (*D*, *pars*+*vs*) ;

D_i := ElimRedund (*CTX*, D_{vs}) ;

CTX := *CTX*+ D_i ;

D := ElimRedund (*CTX*, *D*) ;

конец

Теперь найдем границы цикла по переменной v_i. Для этого достаточно для нижней/верхней границы выбрать из D_i одно или несколько неравенств, ограничивающих v_i снизу/сверху и выразить это ограничение через остальные

переменные v_1, ..., v_{i-1} (объединив результаты, если неравенств несколько, через max/min).

В применении к функции (2) алгоритм сработал не очень удачно – результат сложен и неэффективен. Помешала сложная форма блоков: ограничения сверху и снизу для *x* выражаются двумя неравенствами. Чтобы это обойти, разобьем каждую шестиугольную плитку с индексом (*p*,*s*) на две половинки – верхнюю и нижнюю. Тогда в каждой части будет только одно неравенство слева и справа. Чтобы это выразить через функцию распределения Ψ, введем дополнительную переменную, например q=t/h–s, h=3, и включим ее в Ψ сразу после *p*:

$$\Psi(\mathrm{D}) = \{\ (s,\underline{p},[q],t,x) \mid 1\le t\le M,\ 1\le x<N,\ p=p(t,x),\ s=s(t,x),\ q=t/h-s\ \}. \qquad (3)$$

Чтобы получить интересующий нас эффект, мы выделили элемент q квадратными скобками, что является указанием статически «размотать» (unroll) соответствующий цикл в последовательность из конечного числа вариантов тела цикла. Компилятор сам определит границы для *q* (здесь –1..0) и прооптимизирует каждый вариант (через ElimRedund) с учетом его специфики.

В результате порождается[7] код на C (с директивой OpenMP), показанный на рис. 5.

```
#define B(t,x) BB[t%2,x]
double A[N+1],BB[2,N+1];
int s,p,t,x;
for (int x=0; x<=N; x++)
  B(0,x)=A[x];
for (int s=0; s<=(M+3)/3; s++)
  #pragma omp parallel for
  for (int p=-1; p<=(N-4)/10; p++)
    if ((p+s)%2 == 0) {
      // q=-1
      for(int t=max(1,3*s-3); t<=min(M,3*s-1); t++)
        for(int x=max(1,10*p-t+3*s+3);
                x<=min(N-1,10*p+t-3*s+16); x++)
          B(t,x)=0.33333*(B(t-1,x-1)+B(t-1,x)+B(t-1,x+1));
      // q=0
      for(int t=max(1,3*s); t<=min(M,3*s+2); t++)
        for(int x=max(1,10*p+t-3*s+4);
                x<=min(N-1,10*p-t+3*s+15); x++)
          B(t,x)=0.33333*(B(t-1,x-1)+B(t-1,x)+B(t-1,x+1));
      };
for (int x=0; x<=N; x++)
  A[x]=B(M,x);
```

Рис. 5. Результат трансляции графа в код для OpenMP по функции (3)

[7] Пока у нас автоматически порождаются только выражения, а их вставку в заголовки циклов мы делали руками.

Изначально в построенном коде используется обращение к двумерному массиву B(t,x), представляющему выход виртуального узла B⟨t,x⟩. Его размер чрезвычайно велик, поскольку каждый элемент присваивается не более одного раза. При отображении его на реальный массив элементы последнего надо использовать многократно, но так, чтобы не затирались нужные значения. В данном примере воспользуемся наблюдением[8], что при записи элемента `B(t+2,x)` элемент `B(t,x)` уже заведомо не нужен (поскольку все использования последнего находятся среди элементов, от которых зависит первый). Поэтому ничто не мешает виртуальные элементы `B(t,x)` отобразить на один реальный: `BB[t%2,x]`.

Видно, что даже для столь простого примера сложность написания кода заметно превосходит сложность написания соответствующей функции распределения (3), не говоря о высокой вероятности ошибок. А уже для такой же задачи размерности 2 или 3 написать подобный код вручную практически невозможно. Отметим, что ошибки в функции распределения, не нарушающие порядок, при корректном генераторе не приведут к неправильной программе.

Столь замысловатые распределения оправданны, если они дают выигрыш. На рис. 6 приводятся результаты измерений производительности для следующих вариантов:

- Base1 – исходный код на рис. 1 с распараллеливанием обоих внутренних циклов,
- Base2 – то же, но вместо цикла копирования выполняется шаг вычисления с подменой массивов A и B,
- Rhomb – распределение как на Рис. 4(а),
- Trap – распределение как на Рис. 4(б).

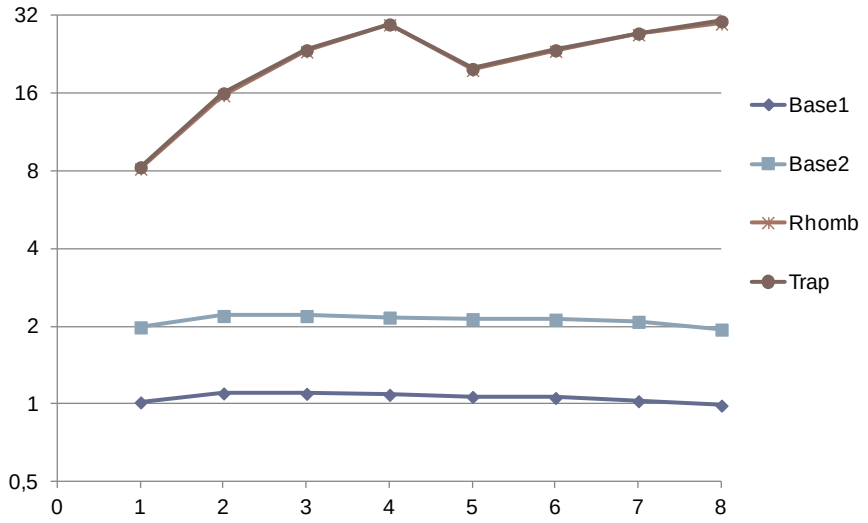

Рис. 6. Производительность (GFlops) в зависимости от числа потоков

[8] Автоматизация задачи отображения будет предметом дальнейших исследований.

Измерения проводились на ПК с 4-ядерным процессором Intel Core i7-2600 3.4GHz на компиляторе из Microsoft Visual Studio Community-2019 с распараллеливанием на 1–8 потоков с параметрами: N=2000000, M=5000, d=300, h=125. Видим, что базовые варианты вообще не распараллеливаются, поскольку работают на пределе скорости доступа к памяти. Здесь важно, что данные задачи не помещается в кэш L3 целиком. При этом Base2 работает вдвое быстрее, поскольку не совершает холостого переписывания. Варианты Rhomb и Trap дают одинаковые ускорения, причем 4 раза (по сравнению с Base2) дает само блочное распределение, а еще 4 раза дает распараллеливание по ядрам и потокам. Оказалось, что вариант Trap ничем не лучше Rhomb, но чтобы это узнать, надо было сделать код и испытать его. Для этого, собственно, и нужна автоматическая генерация кода по заданной функции распределения.

9. Заключение

В работе были продемонстрированы возможность и метод автоматического синтеза параллельных программ (здесь для платформы OpenMP) по ее спецификации в форме потокового графа алгоритма. Для этого пользователем дополнительно должно быть задано отображение на ресурсы машины (прежде всего пространство и время) в форме функций распределения квазиаффинного класса. Важно, что выбор отображения не может нарушить правильности программы, если, конечно, компилятор корректен. Чтобы правильно осуществлять построение кода, компилятор должен уметь эффективно выполнять разрешающие процедуры в логике первого порядка для квазилинейных целочисленных форм с операциями сравнения (=,<).

Есть много работ по автоматическому синтезу параллельных программ, в которых функции распределения выбираются тоже автоматически. Однако класс таких функций, как правило, ограничен чисто линейными, с возможным делением только в качестве последнего шага. Мы работаем с более широким классом функций, среди которых имеются, возможно, дающие более высокие результаты по производительности. Но задавать их предлагаем вручную, хотя не исключаем и автоматизации в этом деле. В статье [6] автор дает некоторый обзор аналогичных работ и сравнение с ними собственного подхода.

Отметим проблемы, которые требуют дальнейшей работы:

1. Выдача результирующего кода на C.
2. Задача отображения виртуальных массивов в реальные.
3. Сохранение символических параметров, когда они не являются линейными (здесь это параметры d и h).
4. Переход к размерностям 2 и 3 в этой задаче и к другим задачам.
5. Сравнение по получаемой производительности с аналогичными работами других авторов.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проекты 17-07-00324 и 17-07-00478.

Литература

1. Проект AlgoWiki. — URL: http://algowiki-project.org .
2. Штейнберг Б. Я. Открытая распараллеливающая система // PACO'2001/ Труды международной конференции «Параллельные вычисления и задачи управления». — М.: ИПУ РАН, 2001. — С. 214–220.
3. Шульженко А. М. Решетчатый граф и использующие его преобразования программ в Открытой распараллеливающей системе // Труды Всероссийской научной конференции «Научный сервис в сети Интернет: технологии распределенных вычислений», 19-24 сентября 2005 г., г. Новороссийск. — С. 82–85.
4. Гуда С. А. Операции над представлениями кусочно-квазиаффинных функций в виде деревьев // Информатика и ее применения, том 7, выпуск 1, 2013. — С. 58–69.
5. Климов Арк. В. О парадигме универсального языка параллельного программирования // Языки программирования и компиляторы — 2017 (PLC–2017). Труды конференции. — Ростов-на-Дону: ЮФУ, 2017. — С. 141–146.
6. 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 .
7. Климов Арк. В. К автоматическому порождению программ трафаретных вычислений с улучшенной временной локальностью // Программные системы: теория и приложения, 2018, 9:4(39). — С. 493–508. — DOI: 10.25209/2079-3316-2018-9-4-493-508 . — URL: http://psta.psiras.ru/read/psta2018_4_493-508.pdf
8. Стемпковский А.Л., Левченко Н.Н., Окунев А.С, Цветков В.В. Параллельная потоковая вычислительная система — дальнейшее развитие архитектуры и структурной организации вычислительной системы с автоматическим распределением ресурсов // "Информационные технологии" №10, 2008. — С. 2 – 7.
9. Змеев Д.Н., Климов А.В., Левченко Н.Н., Окунев А.С., Стемпковский А.Л. Эмуляция аппаратно-программных средств параллельной потоковой вычислительной системы «Буран» // «Информационные технологии», т. 21, №10, 2015. — С. 757-762.
10. Ancourt C., Irigoin F. Scanning polyhedra with DO loops. Principles and Pratice of Parallel Programming, PPoPP'91, Apr 1991, Williamsburg, Virginia, United States. Volume 26 (Issue 7). Pages 39–50, 1991. — DOI: 10.1145/109626.109631 .
11. Bondhugula U., Ramanujam J., Sadayappan P. PLuTo: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-5/07-TR70. — The Ohio State University, Oct. 2007.
12. Feautrier P., Lengauer C. Polyhedron Model // Encyclopedia of Parallel Computing. — Springer, 2011. — Pages 1581–1592.

References

1. AlgoWiki Project. — URL: http://algowiki-project.org .
2. Stejnberg B.Ya. Otkrytaja rasparallelivajushchaja sistema // PACO'2001/ Trudy mezhdunarodnoj konferencii "Parallelnye vychislenija i zadachi upravlenija". — M.: IPU RAS, 2001. — Pages 214–220.
3. Shulzhenko A. M. Reshetchatyj graf i ispolzujushchie ego preobrazovanija v Otkrytoj rasparallelivajushchej sisteme // Nauchnyi servis v seti Internet: trudy Vserossiiskoi nauchnoi konferentsii (19-24 sentiabria 2005 g., g. Novorossiisk). — Pages 82–85.
4. Guda S.A. Operacii nad predstavlenijami kusochno-kvaziaffinnykh funkcij v vide derev'ev // Informatics and applications, Volume 7, Issue 1, 2013. — Pages 58–69.
5. Klimov Ark. V. O paradigme universal'nogo yazyka parallel'nogo programmirovaniya // Yazyki programmirovanija i kompilyatory (Programming Languages and Compilers) — 2017. Trudy konferencii. — Rostov-na-Donu: YuFU, 2017. — P. 141–146.
6. Klimov Ark. V. Obzor podkhodov k sozdaniyu multi-platformennoj sredy parallelnogo programmirovaniya // Nauchnyi servis v seti Internet: trudy XIX Vserossiiskoi nauchnoi konferentsii (18–23 sentiabria 2017 g., g. Novorossijsk). — M.: KIAM, 2017. — Pages 260–275.
7. Klimov Ark. V. Towards automatic generation of stencil programs with enhanced temporal locality // Program Systems: Theory and Applications, 2018, 9:4(39). — Pages 493–508. (In Russian). — DOI: 10.25209/2079-3316-2018-9-4-493-508. — URL: http://psta.psiras.ru/read/psta2018_4_493-508.pdf .
8. Stempkovskij A.L., Levchenko N.N., Okunev A.S., Cvetkov V.V. Parallel Dataflow Computing System: Further Development of Architecture and Structural Organization of the Computing System with Automatic Distribution of Resources // «Informacionnye tehnologii», 2008, No. 10. — Pages 2-7 (In Russian).
9. Zmeev D.N., Klimov A.V., Levchenko N.N., Okunev A.S., Stempkovskij A.L. Emulation on Hardware and Software of the Parallel Dataflow Computing System "Buran" // «Informacionnye tehnologii», vol. 21, No. 10, 2015. — Pages 757-762 (In Russian).
10. Ancourt C., Irigoin F. Scanning polyhedra with DO loops. Principles and Practice of Parallel Programming, PPoPP'91, Apr 1991, Williamsburg, Virginia, United States. Volume 26 (Issue 7). Pages 39–50, 1991. — DOI: 10.1145/109626.109631 .
11. Bondhugula U., Ramanujam J., Sadayappan P. PLuTo: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-5/07-TR70. — The Ohio State University, Oct. 2007.
12. Feautrier P., Lengauer C. Polyhedron Model // Encyclopedia of Parallel Computing. — Springer, 2011. — Pages 1581–1592.