АВТОМАТИЧЕСКОЕ РАСПАРАЛЛЕЛИВАНИЕ ЛИНЕЙНЫХ ЦИКЛОВ ПУТЕМ ТРАНСЛЯЦИИ В ЯЗЫК ПОТОКА ДАННЫХ

Арк.В.Климов, А.С.Окунев, Н.Н.Левченко

ИППМ РАН, Москва

Введение. Проблема автоматического распараллеливания программ хорошо известна. Существующие компиляторы, распараллеливающие на уровне циклов, решают ее путем выяснения для каждого цикла, является ли он параллельным (более точно: допускает ли он параллельное выполнение своих итераций). Для этого они проверяют, нет ли зависимостей между операциями чтения и записи, мешающих параллельному выполнению цикла. Цикл не может выполняться как параллельный, если на разных его итерациях какие-то две операции, из которых хотя бы одна является записью, читают или пишут в один и тот же элемент какого-то массива. При этом важно обнаружить сам факт наличия или отсутствия зависимости, а какие конкретно операции пишут и читают и в какие элементы – неважно. Иначе говоря, такой анализ вправе быть неполным [1, гл.11]. Более продвинутые компиляторы пытаются преобразовать циклы так, чтобы какой-то из них (желательно, внешний) стал параллельным.

Такой подход, если и дает результат, то только для SMP (многоядерные, многопоточные процессоры с общей памятью). Для распределенных систем (кластеры с MPI) или систем с графическим сопроцессором (CUDA) такой подход малоэффективен. Причина – необходимость перехода к модели вычислений с явным параллелизмом, а это требует привлечения дополнительного знания об алгоритме.

Предлагаемый нами подход также основан на переходе в другую модель вычисления, но с неявным параллелизмом, извлекаемым динамически по принципу готовности данных. И для перевода в нее не требуется дополнительной информации, кроме той, которую хороший анализатор способен извлечь из исходной программы. Это – точное описание графа потоковых (истинных) зависимостей между экземплярами операций с памятью. (Правда, еще могут потребоваться сведения, касающиеся распределения вычислений по вычислительным узлам, но это будет лишь внешнее по отношению к основному результату перевода дополнение, которое влияет лишь на эффективность, и которое легко может быть добавлено автором программы в терминах ее исходного кода. Но этот вопрос выходит за рамки данной работы.)

Данная статья имеет следующую структуру. В разделе 1 вводится модель вычислений, которую мы по традиции называем потоковой (dataflow). Она во многом подобна классической модели dataflow[2], но имеет и существенные отличия. Основное – это наличие контекста, который полностью контролируется (задается) программистом. Описывается язык программирования DFL, соответствующий этой модели вычислений. В разделе 2 описывается архитектура вычислителя, способного выполнять программу на языке DFL. Затем, в разделе 3, ставится формальная задача перевода последовательных программ в эту модель вычислений. При этом накладываются определенные ограничения на допустимые исходные программы и вводится понятие графа алгоритма, описание которого может быть автоматически построено для любой допустимой программы. По описанию графа уже легко строится программа на DFL. В разделе 4 работа компилятора демонстрируется на примере программы LU-декомпозиции матриц.

1. Модель вычислений. Вычисление, рассматриваемое как целое, разбивается на вычислительные элементы, каждый из которых является экземпляром одного из типовых узлов. Между элементами возникают связи по данным: одни элементы вычисляют некоторые данные, другие эти данные используют. Интерфейс и поведение типового узла задаются в его описании.

Конечный набор описаний типовых узлов и является программой в этой модели вычислений. Описание типового узла состоит из заголовка и программы узла. В заголовке указываются входы (имя и тип каждого) и контекст, задаваемый как список имен полей целого типа. В программах узлов для записи основных вычислений используется подмножество языка Паскаль. При этом в качестве внешних данных могут использоваться только значения входов, полей контекста и констант, засылаемых из хост-процессора перед началом работы, а внешним результатом работы может быть только посылка одного или нескольких токенов на входы других узлов (или хост-процессор). Оператор посылки токена имеет вид:

$$v \rightarrow N.p\{e_1,\ldots,e_k\}$$

где v – посылаемое значение, N – имя узла, p – имя входа, e_i – целые выражения, в совокупности задающие контекст целевого виртуального узла. Читается: послать значение v на вход p виртуального узла $N\{e_1,\ldots,e_k\}$. Или короче: послать v на $N.p\{e_1,\ldots,e_k\}$. Виртуальным узлом мы называем экземпляр типового узла с конкретным набором значений полей контекста. Несколько токенов, направленных на разные входы одного и того же виртуального узла, должны встретиться вместе, и когда соберутся токены для всех входов, образуется пакет – задание на выполнение программы узла. Созданные пакеты друг от друга не зависят и могут выполняться в произвольном порядке или в параллель. Для каждого пакета выполняется программа соответствующего типового узла, в результате чего могут появиться новые токены. Некоторые токены посылаются «наружу» (на хост-процессор). Это такие, у которых целевой узел имеет имя с суффиксом _out,

ровно один вход и пустую программу. Чтобы «процесс пошел» надо, чтобы вначале извне (из хост-процессора) поступило некоторое количество входных токенов с начальными данными. Процесс завершается, когда в системе больше нет готовых пакетов (как нет и наборов токенов, которые их могут породить). Набор посланных на хост-процессор токенов является результатом работы.

Для иллюстрации модели вычислений приведем простой пример: вычисление массива частичных сумм произведений двух массивов. На рисунке 1 слева приводится спецификация алгоритма в виде фрагмента на фортране, а справа – соответствующий фрагмент на DFL, состоящий из одного трехвходового узла P.

```
REAL A(N),B(N),C(N),S
S = 0.0
DO I = 1,N
  S = S + A(I) * B(I)
  C(I) = S
ENDDO
```

```
node P(s:real,a:real,b:real) {i};
 var v:real;
 begin
   v := s+a*b;
   v -> P.s{i+1};
   v -> C.s{i};
 end;
```

Слева – фортран, справа – DFL. Предполагается, что исходные данные засылаются на входы a и b узлов P{i}, i=1,…,N, а результаты будут на C.s{i}. Кроме того, на P.s{1} надо послать 0 «для затравки».

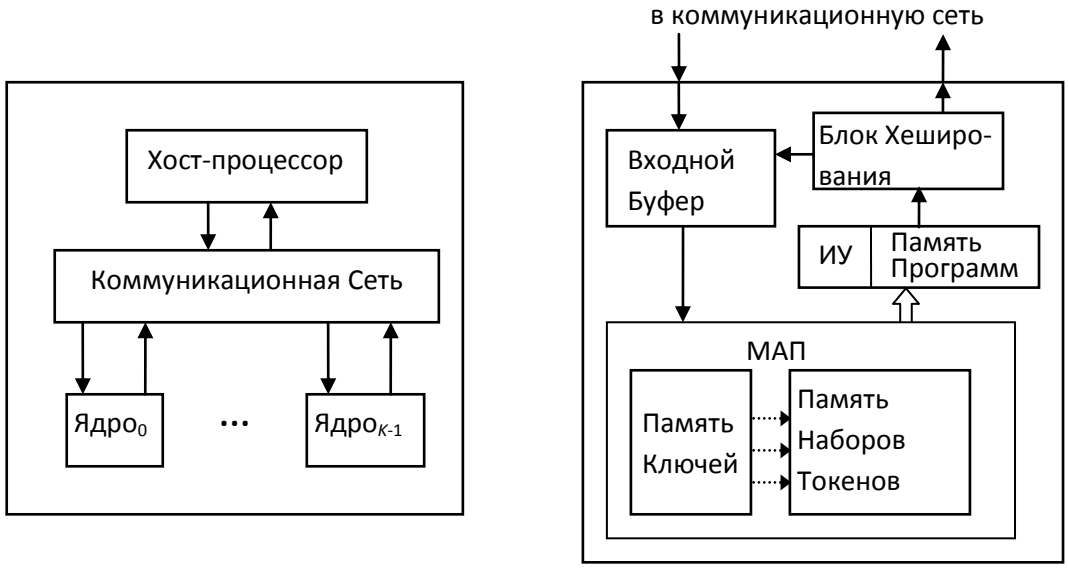
Рисунок 1. Фрагмент программы вычисления частичных сумм произведений двух векторов

Здесь параллелизма нет: узел P{i+1} не может работать, пока узел P{i} не отправил ему свою частичную сумму s. Но можно делать умножения параллельно, если выделить его в отдельный узел. Оставляем это читателю в качестве упражнения.

2. Архитектура вычислителя. Архитектура вычислителя для данной модели вычислений была предложена в [2] и далее развита в [3]. Для полноты нашего изложения приведем здесь его краткое описание. Вычислитель состоит из некоторого числа *K* ядер, соединенных коммуникационной сетью (Рис. 2). Ядро состоит из модуля ассоциативной памяти (МАП) и исполнительного устройства (ИУ). Токены через коммуникационную сеть приходят в МАП и могут там находиться. МАП состоит из ассоциативной памяти (АП) ключей, где хранятся ключи со ссылками на наборы токенов, и память для наборов токенов (ПТ). Ключ токена состоит из кода (кодирующего имя) узла и контекста. В реализации они упаковываются в одно 64-битное слово. Вхо-

дящий токен сравнивается в АП по ключу со всеми присутствующими ключами, и если обнаруживается совпадение, то новый токен вносится в соответствующий хранимый набор на место указанного в нем входа. Если набор оказывается полным, выполняется процедура образования пакета. Если совпадения ключа входящего токена с хранимым ключом не обнаружено, в АП заносится новый ключ и в ПТ отводится место для нового набора, куда помещается сам токен. Если узел одновходовой, то сразу выполняется процедура образования пакета.

Токены с общим ключом должны непременно оказаться в одном и том же МАП. Для этого номер целевого МАП вычисляется при создании токена посредством вычисления определенной функции распределения, зависящей только от ключа. Это принципиальный момент: им обеспечивается возможность эффективной и распределенной реализации ассоциативного поиска. Стандартная функция распределения (хеш-функция) обеспечивает достаточно хорошее перемешивание, но разрушает потенциальную локальность. Пользователь может указать или задать свою функцию распределения. Если при хорошем балансе нагрузки удается обеспечить локальность (чтобы токены чаще посылались в свое ядро), то сокращается нагрузка на коммуникационную сеть и, как следствие, повышается общая производительность системы.



а) структура системы б) структура ядра

Простые стрелки показывают движение токенов, двойная – пакетов.

Рисунок 2. Архитектура потокового вычислителя.

Сформированный пакет, состоящий из токенов набора и ключа, поступает через буфер пакетов в ИУ. В норме на каждый вход должно прийти ровно по одному токену. После формирования пакета набор и ключ удаляются из МАП. Пока набор токенов не полон, он продолжает храниться в МАП в памяти наборов токенов, а общий ключ токенов – в памяти ключей.

Для каждого поступающего пакета в ИУ выполняется программа соответствующего узла. При этом могут образоваться новые токены, которые через буфер и блок хеширования (БХ) поступают на вход коммуникационной сети.

В памяти программ всех ИУ хранятся копии программы с константами. Они заносятся туда из хост-процессора перед началом работы.

3. Трансляция последовательных программ в DFL. Для перевода заданной последовательной программы P в язык DFL необходимо построить так называемый граф алгоритма [4, гл.6]. Точнее, для заданной программы нужно построить описание графа в конечной компактной форме. Оно в параметрическом виде описывает любой граф вычисления данной программы, определяемый по прогону программы при конкретных входных данных. Это двудольный граф: он состоит из вершин двух типов. Вершины первого типа (записи) соответствуют исполнениям операций записи в память, а второго типа (чтения) операциям чтения из памяти. Из вершины-записи идет дуга в вершину-чтение, если эта операция чтения считывает из памяти именно то значение, которое было записано данной операцией записи.

Отдельную операцию записи или чтения можно идентифицировать парой $(M,\boldsymbol{I})$, где M – место в программе, где эта операция находится, а $\boldsymbol{I}$ – набор значений параметров внешних (для этого места) циклов. Используя эти пары как идентификаторы вершин, мы можем поставить задачу описать отображение

$$\boldsymbol{F}: (R, \boldsymbol{I}_R) \rightarrow (W, \boldsymbol{J}_W) \qquad (1)$$

которое по идентификатору чтения $(R, \boldsymbol{I}_R)$ выдает идентификатор записи $(W, \boldsymbol{J}_W)$, записавшей читаемое значение (или говорит, что записей не было, а читается изначальное значение).

Но для перевода в DFL нам требуется обратное: для каждой операции записи найти все (их может быть несколько или ни одной) операции чтения, которые именно это значение читают, то есть нужна многозначное функция

$$\boldsymbol{G}: (W, \boldsymbol{J}_W) \text{ -> } \{ (R, \boldsymbol{I}_R) \} \qquad (2)$$

которая по идентификатору записи $(W,\boldsymbol{J}_W)$ выдает множество идентификаторов чтений $\{(R,\boldsymbol{I}_R)\}$, читающих записываемое значение.

К сожалению, выразить в явной форме эти отображения удается далеко не всегда. Но есть хорошо определенный класс программ, для которых это возможно: это *линейные* программы (см. [4, гл.6]). Они составлены из следующих конструкций:

- операторы присваивания с линейными индексными выражениями во всех обращениях к массивам;
- правильно вложенные циклы do-enddo с единичным шагом, линейными выражениями для верхней и нижней границы и без преждевременных выходов из цикла;

- правильно вложенные условные операторы if-then-else-endif с условиями вида e=0 или e>0, где e – линейное выражение.

Выражения называются *линейными*, если они являются линейными формами с целыми коэффициентами, где в качестве переменных используются лишь только параметры внешних циклов и некоторые фиксированные параметры (которые определяются вне участка, подлежащего анализу и преобразованию, и в нем не изменяются).

Пока мы рассматриваем только строго линейные программы. Впоследствии некоторые из ограничений могут быть сняты. Но в любом случае, общая схема преобразования исходной программы такая: транслятор ее сканирует и ищет участки, которые удовлетворяют условиям допустимости – и только их переводит в DFL. А на их место вставляет новый код, который посылает данные на входные узлы и принимает результаты с выходных. Сама внешняя программа выполняется на хост-процессоре, а ее преобразованная параллельная часть – на параллельной потоковой системе. Таким образом, в худшем случае, наш транслятор ничего не сделает, и программа останется последовательной. Но на практике вычислительные ядра очень многих реальных программ удовлетворяют условиям допустимости (будущим транслятором). Пока же требуется, чтобы вся транслируемая исходная подпрограмма была строго линейной.

Существует теория о том, что для любой линейной программы ее граф может быть описан при помощи систем линейных равенств и неравенств, расширенных операцией целочисленного деления с остатком. Она восходит к теореме о разрешимости арифметики Пресбургера[5]. Применимость ее к анализу программ для нужд распараллеливания была показана в работе [6].

Опираясь на эти теоретические предпосылки, одним из авторов в работе [7] было введено понятие дерева выбора и показано, как могут быть описаны с его помощью однозначные функции типа (1). Там же представлен алгоритм построения описания графа в виде набора деревьев выбора, однозначно указывающих для каждой операции чтения соответствующий оператор записи. Для представления обратного графа понятие дерева выбора расширено до многозначных деревьев, путем добавления двух новых видов вершин: AND-вершин и ANY-вершин.

Напомним строение обычного (однозначного) дерева выбора T. В нем есть вершины вида (C–>T_1 : T_2), где C – условие вида e=0 или e>0 с линейными e (к ним легко сводятся и другие операции сравнения), а T_1 и T_2 – поддеревья для случаев True и False. Еще есть вершины «деления» вида ($e = m\boldsymbol{q}+\boldsymbol{r}$ –> T), где m – целое, а $\boldsymbol{q}$ и $\boldsymbol{r}$ – новые переменные. В листьях стоит либо терм None (который означает, что записей не было), либо терм вида $W\{e_1,e_2,\ldots\}$, где W – программное имя оператора записи (присваивания), а e_i – линейные выражения. В расширенном дереве допускаются также вершины вида (AND T_1 T_2 … T_n), n≥2, и вида (ANY v T_1). Значением многозначного дерева является не одиночный атом (терм с числовыми аргументами вместо выраже-

ний e_i), а множество атомов. Значением AND-вершины является объединение значений деревьев T_1, T_2, … T_n. А значением ANY-вершины является объединение значений дерева T_1 (зависящего, вообще говоря, от переменной v) по всем v. (При этом для ANY-вершины вычисляется также диапазон значений переменной v с границами в виде линейных выражений от остальных переменных, вне которого значений у дерева T_1 нет.) Такими деревьями для любой линейной программы могут быть выражены функции вида (2), задающие граф алгоритма, в котором для каждой операции записи указывается множество всех соответственных операций чтения. В нем имя R операции чтения кодируется парой S.p, где S – имя оператора присваивания, p – локальное имя данного чтения внутри оператора. Для каждого оператора присваивания S в описании графа имеется свое дерево T_S. На рисунке 3 в качестве примера приведено дерево для оператора A1 из примера, показанного на рисунке 4, в том виде, как его построил анализатор. В ANY-вершине анализатором вставлен диапазон изменений переменной. В первой строке, кроме имени и параметров вершины имеется список условий, которым эти параметры должны удовлетворять.

```
((A_1 K I) (K>=1) (K-N<=-1) (I-K>=1) (I-N<=0)
   (AND
     A.OUT{I,K}
     (ANY J (K+1..N)
       A_2.A_2{K,I,J} )
   )
)
```

Рисунок 3. Пример многозначного дерева

Теперь, когда имеется описание графа ***G***, программа переводится в DFL следующим образом. Для каждого оператора присваивания S вводится отдельный DFL-узел с именем S. Входами станут локальные имена операций чтения для всех входящих в правую часть переменных или элементов массива (без дублирования). Контекстом будет список имен переменных всех внешних циклов данного оператора. Например, для исходного оператора присваивания

```
A(i,j) = B(i) * B(j)
```

стоящего внутри двойного цикла с параметрами i,j, может быть построен заголовок

node A6 (b1:**real**, b2: **real**) {i,j};

Далее строится тело программы узла. В начало помещается оператор вычисления правой части, где вместо переменных и обращений к массивам стоит локальное имя входа:

a:= b1*b2;

Затем из дерева T , порожденного анализатором для операции записи в A(i,j), генерируется оператор <T>, в котором:

- каждый терм R_p(e_1,…,e_k) превращается в оператор посылки a→R.p(e_1,…,e_k),
- терм None порождает пустой оператор,
- вершина (C–>T_1 : T_2) превращается в условный оператор **if** C **then** <T_1> **else** <T_2>,
- вершина ($e = mq+r$ –> T_1) порождает блок **begin** q:=e **div** m; r:=e **mod** m; <T_1> **end**,
- вершина (AND T_1 T_2 … T_n) порождает блок **begin** <T_1>; <T_2>; … <T_n> **end**,
- вершина (ANY v T_1) порождает цикл **for** v:= l **to** u **do** <T_1>, где l и u – линейные выражения для нижней и верхней границ диапазона, вне которого T обращается в None.

Это простейший вариант трансляции. Более «умный» транслятор для сокращения накладных расходов будет искать возможности склеивать несколько операторов в один узел, а также несколько входов в один вход с составным значением.

4. Пример трансляции с Фортрана в DFL. На рисунке 4 вверху слева показан пример исходной линейной подпрограммы на Фортране, которая выполняет LU-разложение матрицы A, получая ее на входе в виде двумерного массива A(n,n) и возвращая в нем же обе матрицы L и U. Остальные фрагменты (серые) – результат трансляции. Вверху справа – вариант подпрограммы, которым подменяется исходная. Здесь язык Фортран расширен операторами SEND и RECEIVE, которые, соответственно, посылают исходные данные в параллельную подсистему и принимают ее результаты. Оператор SEND посылает все элементы каждого указанного в нем массива X в виде токенов с индексами элемента в качестве контекста на узел X_in: X(i,j) -> X_in{i,j}. Оператор RECEIVE принимает токены и заносит их значения в соответствующий массив на указанное контекстом место. Здесь при имени массива в скобках указывается количество принимаемых токенов (оно позволяет определить момент завершения приема).

На рисунке 4 внизу показан перевод тела этой подпрограммы в DFL. В подпрограмме два оператора присваивания, назовем их A1 и A2. Первый вложен в два цикла, второй в три. Соответственно узел A1 имеет контекст из двух полей, узел A2 – из трех. Тело каждого узла содержит образ самого оператора присваивания, за которым следуют операторы рассылки вычисленного значения. Узел A_in принимает входные элементы и рассылает их куда требуется, а через узел A_out проходит возврат результатов.

Заметим, что в полученной программе почти нет циклов. Те, что есть, связаны с необходимостью разослать одно значение сразу на много узлов. Но исходные циклы исчезли. Вместо них в хост-процессоре создается поток элементов входных массивов, которым активизируется сразу множество узлов. И далее данные сами передают активность другим узлам, порождая тот же объем вычислений, который в исходной программе порождался циклами. Поэтому мы говорим о нашем вычислителе, как о системе, управляемой потоком данных.

При выполнении на системе с достаточно большим числом ядер в нашем примере достигается параллелизм порядка $O(n^2)$, в основном за счет возможности одновременно выполнять все узлы A2{k,i,j} при одном k. Причем могут совмещаться также и различающиеся в значениях i,j узлы A2 с различными k, а узлы A1 - с узлами A2 для предыдущих k.

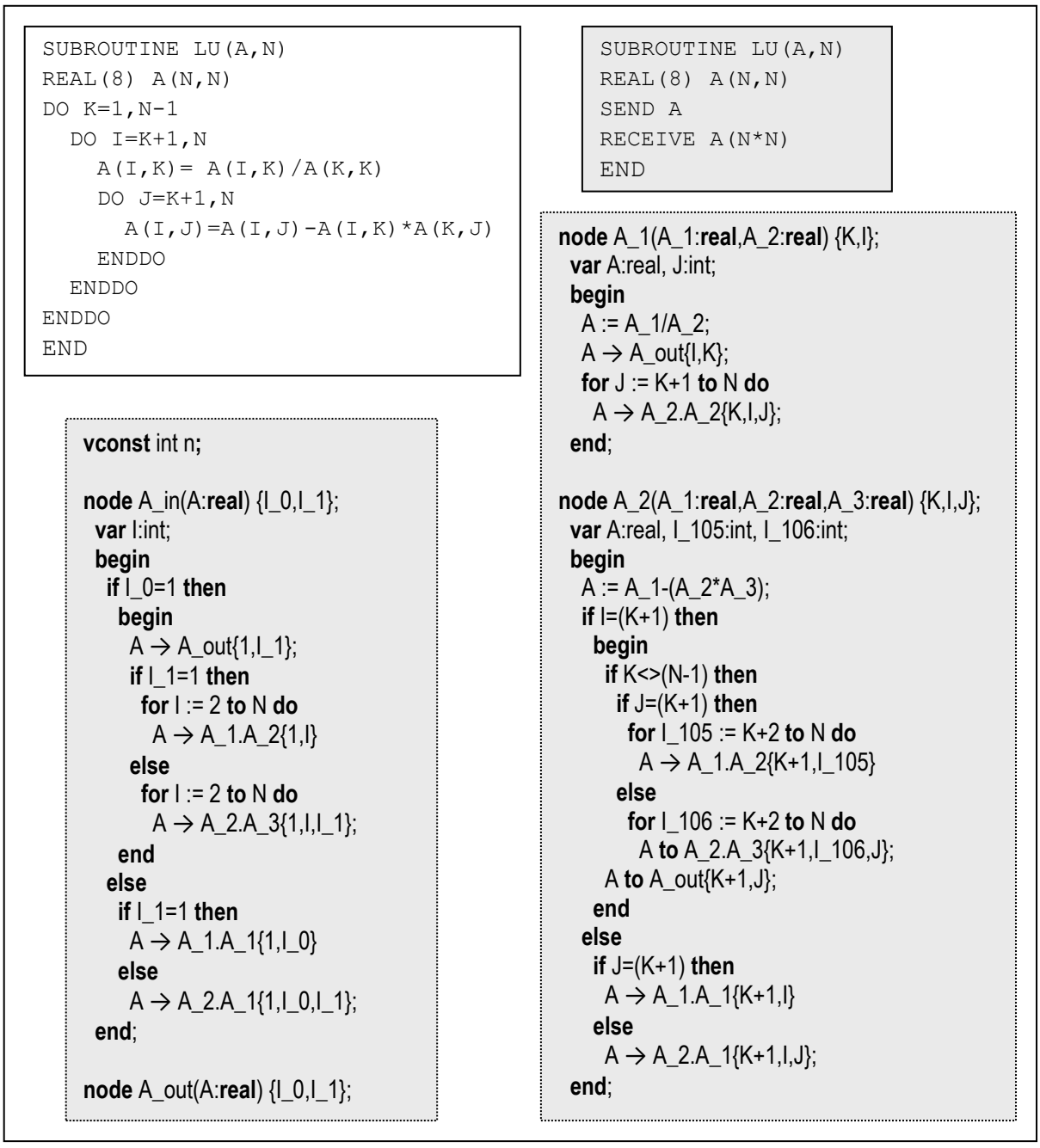


Рисунок. 4. Пример трансляции: подпрограмма LU-разложения

Выводы. Наш компилятор выявляет потоковые зависимости алгоритма и переводит программу в такую форму, где они выражены явно. При этом, глядя на результирующий код, мы не можем сказать, что с чем будет выполняться параллельно. Но в этой форме заложен потенциальный параллелизм, который может проявиться при исполнении на соответствующей аппаратуре. Таким образом, автоматическое распараллеливание достигается совместными усилиями компилятора и исполняющей системы. Компилятор лишь выявляет полный граф потоковых зависимостей и выражает его в языке DFL. Параллелизм явно компилятором не планируется, а выявляется уже при выполнении. Это отличает наш подход от традиционных распараллеливателей, которые стремятся выявить параллелизм на стадии компиляции, используя модели вычислений (Open MP, MPI, CUDA), требующие явно выраженного параллелизма. И это позволяет нам, в конечном счете, извлекать из программы больше параллелизма, чем это делают традиционные распараллеливающие компиляторы.

Список литературы

1. Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: пер. с англ. М: «И.Д. Вильямс», 2008.
2. Бурцев В.С. Выбор новой системы организации выполнения высокопараллельных вычислительных процессов, примеры возможных архитектурных решений построения суперЭВМ, в сб. Параллелизм вычислительных процессов и развитие архитектуры суперЭВМ, ИВВС РАН, М.:1997, с.41-78.
3. 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.
4. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб: БХВ-Петербург, 2004.
5. Presburger M. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. // In *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*. Warszawa, 1929, p.92–101.
6. Pugh W. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. // Comm.of the ACM, August 1992.
7. Климов Арк.В. Использование деревьев выбора для описания состояний в распараллеливающем компиляторе. // Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность. Труды Всероссийской суперкомпьютерной конференции, М.: Изд-во МГУ, 2009, с.238-240.