УДК 681.3

УТОЧНЕНИЕ ЗАВИСИМОСТЕЙ ПРОГРАММЫ В ДВОР

Штейнберг Б.Я.
Южный федеральный университет
Россия, 344006, Ростов-на-Дону, Большая Садовая ул., 105/42
E-mail: borsteinb@mail.ru

Абрамов А.А.
Южный федеральный университет
Россия, 344006, Ростов-на-Дону, Большая Садовая ул., 105/42
E-mail: Azef88@yandex.ru

Баглий А.П.
Южный федеральный университет
Россия, 344006, Ростов-на-Дону, Большая Садовая ул., 105/42
E-mail: TAccessViolation@yandex.ru

Морылев Р.И.
Южный федеральный университет
Россия, 344006, Ростов-на-Дону, Большая Садовая ул., 105/42
E-mail: frg10@yandex.ru

Петренко В.В.
Южный федеральный университет
Россия, 344006, Ростов-на-Дону, Большая Садовая ул., 105/42
E-mail: vpetrenko@gmail.com

Полуян С.В.
Южный федеральный университет
Россия, 344006, Ростов-на-Дону, Большая Садовая ул., 105/42
E-mail: steka@front.ru

Штейнберг Р.Б.
Южный федеральный университет
Россия, 344006, Ростов-на-Дону, Большая Садовая ул., 105/42
E-mail: romanofficial@yandex.ru

Работа поддержана ФЦП «Научные и научно-педагогические кадры инновационной России», Государственный контракт № 02.740.11.0208 от 7 июля 2009 г.

Ключевые слова: распараллеливание, автоматическое распараллеливание, распараллеливание программ, преобразования программ, информационные зависимости
Keywords: parallelizing, automatic parallelizing, program parallelization, program transformations, data dependencies

В основе оптимизирующих и распараллеливающих преобразования программ лежит анализ информационных зависимостей [1]. Не всегда возможно точно определить такие зависимости с помощью быстрых алгоритмов (неравенства Банерджи и др.) [2]. В данной работе описываются методы уточнения графа информационных зависимостей, реализуемые в системе ДВОР (Диалоговый высокоуровневый оптимизирующий распараллеливатель).

PROGRAM DEPENDENCIES REFINEMENT IN HIGH-LEVEL DIALOG BASED OPTIMIZNG PARALLELIZER / B. Ya. Steinberg (Southern Federal University, 105 Sadovaya, Rostov-on-Don 344006, Russia, E-mail: borsteinb@mail.ru), A.A. Abramov (Southern Federal University, 105 Sadovaya, Rostov-on-Don 344006, Russia, E-mail: Azef88@yandex.ru), A.P. Baglij (Southern Federal University, 105 Sadovaya, Rostov-on-Don 344006, Russia, E-mail: TAccessViolation@yandex.ru), R.I. Morylev (Southern Federal University, 105 Sadovaya, Rostov-on-Don 344006, Russia, E-mail: frg10@yandex.ru), V.V. Petrenko (Southern Federal University, 105 Sadovaya, Rostov-on-Don 344006, Russia, E-mail: vpetrenko@gmail.com), S.V. Poluyan (Southern Federal University, 105 Sadovaya, Rostov-on-Don 344006, Russia, E-mail: steka@front.ru), R.B. Steinberg (Southern Federal University, 105 Sadovaya, Rostov-on-Don 344006, Russia, E-mail: romanofficial@yandex.ru). Optimizing and parallelizing program transformations are based on data dependency analysis. It is not always possible to determine such dependencies with fast algorithms (like Banerjee inequalities). This paper describes methods of data dependencies graph refinement implemented in High-level dialog based optimizing parallelizer.

1. Внутреннее представление ДВОР

К проекту Диалогового высокоуровневого оптимизирующего распараллеливателя (ДВОР) предъявлены требования многоязыковости, общей библиотеки преобразований для различных языков, возможности диалоговой компиляции. Эти требования иногда противоречат друг другу и влияют на определение информационных зависимостей. Представление является многоуровневым, с разной детализацией программы на каждом уровне. Верхний уровень для каждого входного языка индивидуальный и удобен для разбора. Второй уровень общий для всех входных языков, информация о зависимостях собирается, в основном, на нем. Например, в языках Си и Фортран по-разному определены массивы. Второй уровень представления сглаживает эти различия, чтобы можно было применить общий алгоритм для определения зависимостей.

2. Граф информационных связей

Граф информационных связей (ГИС) – это ориентированный граф, вершинами которого являются вхождения переменных, а дуга соединяет пару вершин *(v,u)*, если эти вхождения обращаются к одной и той же ячейке памяти (т.е. порождают информационную зависимость), причем вхождение *v* раньше, чем *u* и хотя бы одно из этих вхождений является генератором.

Для выявления зависимостей между парами вхождений необходимо анализировать индексные выражения массивов и выражения для границ циклов, охватывающих данные вхождения. Эти выражения должны быть линейными относительно счетчиков циклов, то есть они должны иметь следующий вид: $c_1*i_1+c_2*i_2+\ldots+c_n*i_n+c$, где $i_1,i_2,\ldots,i_n$ счетчики циклов, охватывающих данные вхождения; $c,c_1,c_2,\ldots,c_n$ – целые числа. В противном случае считается, что данные вхождения порождают информационную зависимость. Бывают случаи, когда в анализируемых выражениях присутствуют внешние переменные, но применяя символьный анализ иногда удается установить отсутствие зависимости между вхождениями:

Пример 1.

```
for(int i=0; i<100; i++)
  A[i+N] = A[i+N-1] + B[i];
     0         1      2
```

Очевидно, что граф информационных связей для данного примера будет состоять из одной дуги, идущей их вхождения A[i+N] в вхождение A[i+N-1]. Но индексные выражения обоих вхождений являются нелинейными, поэтому если не применять символьный анализ, то анализ информационной зависимости построит граф в котором будет 4 дуги: из вхождения A[i+N] в вхождение A[i+N-1], из A[i+N-1] в A[i+N] и две петли самозависимости вхождений A[i+N] и A[i+N-1] .

В ДВОР для определения зависимости между парой вхождений реализованы два теста: НОД-тест и тест на основе неравенств Банержи. Одной из важнейших характеристик метода является его сложность. Указанные методы имеют два основных достоинства: это простота программной реализации и линейная сложность. Существенным недостатком этих методов является их неточность определения информационной зависимости. В некоторых случаях, НОД-тест и тест Банержи ошибочно определяют наличие зависимости между парой вхождений.

В ДВОР реализован метод уточнения графа информационных связей с помощью элементарных решетчатых графов [3]. Суть его заключается в том, что для выбранной дуги графа информационных связей строится элементарный решетчатый граф, на основании которого делается вывод о существовании данной зависимости:

Пример 2.

```
for(i=1;i<99;i=i+1)
{
  a[i+1][i]=b[i]+a[i][100-i]+c[i];
}
```

В данном примере при построении графа информационных связей с помощью неравенств Банержи и НОД теста будет неверно определено наличие зависимости a[i][100-i] от a[i+1][i]. Если для построения графа использовать элементарные решетчатые графы, то будет верно установлено ее отсутствие.

2.1. Анализ указателей

Анализ указателей – это глобальный анализ программы, позволяющий для каждой точки программы, содержащей указатель, вычислить множество фрагментов памяти, адреса которых может принимать указатель в этой точке программы. Обычно результаты анализа указателей используется для уточнения потока данных [4], а в ДВОР анализ указателей используется для уточнения информационных зависимостей на графе информационных связей.

С помощью анализа указателей можно получить новые информационные зависимости, обусловленные наличием в программах разных имен одной и той же ячейки памяти, которые не учитываются такими методами, как НОД тест и неравенства Банержи, Омега тест и другими, определяющими зависимости между вхождениями переменных при определенных значениях индексных выражений. Для сбора информации об указателях в ДВОР реализован контекстно-чувствительный межпроцедурный анализ указателей [5], который позволяет получить результаты анализа для исходной, а не преобразованной (например, к SSA-форме [6]) программы, так как для анализа информационных зависимостей нужна информация об указателях для исходной программы.

На основе результатов анализа указателей производится уточнение информационных зависимостей на графе информационных связей. В процессе уточнения зависимостей к графу информационных связей добавляются новые дуги, соединяющие значения указателей с соответствующими вхождениями:

Пример 3.

```
void f(int *p, int *q, int *w)
{
  *p=*q;
  *q=*w;
```

```
}
```

Изначально на графе информационных связей для этого примеры была одна дуга от q к q. Проведя анализ указателей, может оказаться так, что в зависимости от контекста вызова функции f, переменные p и q, q и w, w и q или они все будут указывать на одну и ту же ячейку памяти. Например, для вызова f(a,b,c) новых дуг на графе информационных связей не будет, а для вызова f(a,b,a) добавится дуга от p к w.

Актуальность задачи уточнения информационных зависимостей с помощью анализа указателей обусловлена тем, что на данный момент разработано множество преобразований программ для распараллеливающих компиляторов, но часть из них способна корректно работать лишь с переменными элементарных типов данных, в то время как появление указателей в программах делает эти преобразования способными нарушить корректность программ, что не допускается после уточнения информационных зависимостей.

2.2. Анализ и линеаризация выражений

Определим, что такое анализ линейности выражения и сопутствующие ему термины. Предположим, дано выражение, состоящее из арифметических операций, переменных и констант. Линеаризацией выражения относительно набора переменных a_i называется преобразование произвольного выражения к виду: $e_1*a_1+e_2*a_2+\ldots+e_n*a_n+e_{n+1}$, где e_i – выражения, a_i – переменные, входящие в исходное выражение и при этом ни одна из них не входит ни в одно из выражений e_i. Анализом линейности выражения называется анализ возможности осуществления линеаризации. Будем называть полученное представление линейной формой выражения (линейным разложением). Будем называть базой линеаризации набор переменных a_i, e_i – коэффициентами, и при этом e_{n+1} – свободным членом линейного разложения.

Разбор линейных выражений, в том числе и индексных выражений массивов необходим для реализации следующих частей ДВОР:

- граф информационных связей (граф зависимостей по данным);
- решетчатый граф;
- распараллеливание рекуррентных циклов ;
- генерация MPI-кода.

Разработка алгоритмов анализа и линеаризации выражений началось с простого случая – разбор индексных выражений в массивах с константными коэффициентами. Например, дано выражение a[i+j-2]. Для построения графа зависимостей и решетчатого графа необходимо определить коэффициенты при счетчиках цикла и свободный член. В результате разбора индексного выражения пользователь получает информацию в виде массива чисел {1,1,-2}. А также есть возможность проверить является ли выражение линейным по некоторому набору переменных (база линеаризации).

После создания первой версии библиотеки классов решающих задачу в указанном случае появилась возможность запустить проекты «граф зависимостей по данным» и «решетчатый граф», но только для вхождений массивов содержащих выражения, зависящих только от счетчиков циклов. Реализация первой версии библиотеки позволила определить проблемы в решении задачи в целом и дальнейшие направления развития.

Во-первых, если пользователь пытается построить любой из упомянутых графов с параметрами, то и разбор выражений должен подразумевать возможность параметрических линейных выражений. Например, верхняя граница цикла может быть выражением, зависящим от параметров и граф зависимостей, и решетчатый граф должны суметь получить нужную им информацию об этих выражениях.

Во-вторых, не всегда в качестве базы линеаризации выступают счетчики циклов. Например, для распараллеливания рекуррентных циклов база линеаризации состоит из вхождений массивов.

В-третьих, при приведении подобных над константами выполняются арифметические операции и необходимо учесть семантику языка при вычислении коэффициентов. Например, если дано выражение i+127 – (j-1) надо учитывать типы коэффициентов и в зависимости от

типов линейное выражение будет иметь либо коэффициенты {1,-1,-128} в 8-битной арифметике, либо {1,-1,128} в 16-битной (или более) арифметике. Таким образом, потребовалось усовершенствовать библиотеку классов так, чтобы она предусматривала упомянутые проблемы первой версии библиотеки.

Во второй версии библиотеки появилась возможность разбирать параметрические линейные выражения и отличать выражения с параметрами от выражений зависящих только от переменных входящих в базу линеаризации. Например, если для выражения i+2*j-n в качестве базы линеаризации выбраны переменные i,j, то оно будет считаться зависящим от внешнего параметра n, а если в качестве базы линеаризации выбрать переменные i,j,n, то будет считаться не параметрическим.

Опыт разработки этой версии библиотеки позволил выделить еще несколько проблем в решении задачи в целом.

Во-первых, в качестве элементов базы линеаризации по-прежнему нельзя выбирать вхождения массивов, т.к. a[i] и a[i-1] не отличаются именем переменной, а отличаются индексными выражениями. Например, если дано рекуррентное выражение вида a[i] = a[i-1] + a[i-2] – 2*a[i-1], то собрать коэффициенты при разных вхождениях массива *a* невозможно. Но с использованием функций второй версии библиотеки появилась возможность разработать механизм сравнения на равенство двух вхождений массива *a* и учитывать при этом наличие параметров.

Во-вторых, при разработке второй версии библиотеки стало понятно, что для выполнения операций над константами разных типов необходимо универсальное представление констант с использованием длинных вычислений. Кроме того, необходимо разработать механизм преобразований констант разных типов с учетом семантики языка C, а в будущем и Fortran.

В-третьих, косвенная адресация не разбирается второй версией библиотеки.

Описанные проблемы решаются в реализации третьей версии библиотеки классов, позволяющих представить выражение в линейном виде.

3. Диалоговое уточнение зависимостей

Преобразования программ нацелены на изменение качества программ: увеличение производительности, уменьшение требований к памяти или уменьшение размера исполнимого кода. При выполнении преобразований важно, чтобы не нарушалась семантическая или синтаксическая корректность программы. Для этого, перед преобразованием необходимо выполнить ряд проверок, например, на основе графа информационных зависимостей [7]. Необходимо так же проверять и целесообразность выполнения преобразования, т.к. оно может как улучшить качество программы, так и ухудшить. Однако, для оценки целесообразности необходимо учитывать множество особенностей целевой архитектуры.

При автоматической компиляции оценка целесообразности использования преобразования выполняется компилятором. В режиме диалоговой компиляции, у пользователя появляется возможность опробовать различные комбинации преобразований и то, как они влияют на качество программы.

Значения многих переменных не могут быть определены на этапе компиляции, например, некоторые переменные могут вычисляться или считываться из входных файлов. Это может помешать правильному определению зависимостей в программе, т.к. при наличии неопределенностей всегда предполагается наличие зависимости. Однако пользователю могут быть известны возможные диапазоны значений этих переменных из их назначения в программе. Эти знания позволяют сократить число выявленных информационных зависимостей в программе. Уменьшение количества информационных зависимостей позволяет выполнять некоторые оптимизирующие преобразования или применять другие методы распараллеливания.

Пример 4.

```
A[1] = ....
.... = A[N]
```

В данном фрагменте кода неизвестно значение переменной N, поэтому будет предположено наличие зависимости. Но если пользователю известно, что N никогда не принимает значения равные 1, то зависимость может быть удалена.

Основа диалоговых преобразований – использование знаний пользователя о переменных для уменьшения числа зависимостей в программе. Это позволяет повысить эффективность распараллеливания программ.

Для получения необходимой информации пользователю предлагается ответить на ряд вопросов о возможных значениях переменных. Эти вопросы автоматически генерируются в ДВОР во время анализа зависимостей. В ряде случаев, на основе информации, полученной из ответов пользователя, производится удаление дуг графа информационных зависимостей программы.

Выполнение диалоговых преобразований на основе графа информационных связей (ГИС) можно разбить на следующие шаги:

1. Пользователь выбирает из списка интересующее его преобразование и выделяет фрагмент программы, к которому это преобразование нужно применить.
2. Если в ГИС выделенного фрагмента программы существуют дуги с внешними переменными, то автоматически генерируется список уточняющих вопросов.
3. Пользователь отвечает на предложенные вопросы
4. В случае если введенная информация позволяет выполнить преобразование, то оно выполняется.

От пользователя не требуется ответить на все предложенные вопросы, но чем больше информации будет сообщено, тем более точный анализ будет произведен.

Рассмотрим выполнение диалогового преобразования для следующего примера:

```
for(i = 0; i < 10; i=i+1)
{
    z[i] = x[i-1] * y[i];
    t[i] = a * z[i+m];
}
```

Рис. 1-3 представляют собой копии экранных форм демонстрационного приложения ДВОР, в рамках которого реализованы диалоговые преобразования.

На рис. 1 видно, что пользователь загрузил пример (слева) и построил для него ГИС (справа).

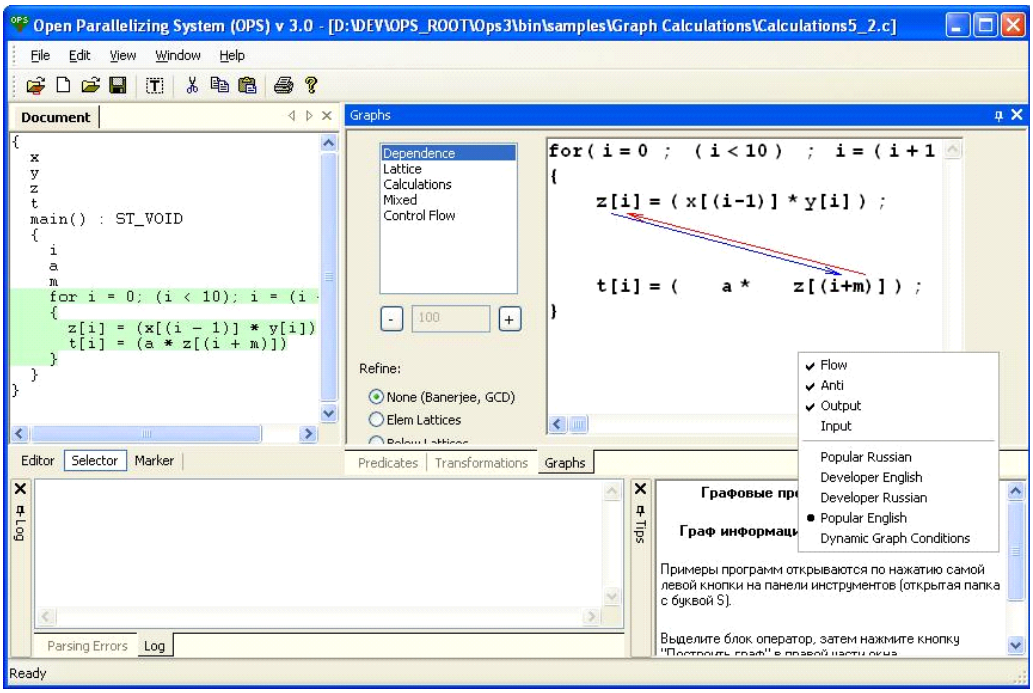

Рис. 1. Граф информационных зависимостей фрагмента программы. Контур из дуг истинной и анти зависимостей мешает разрезанию цикла.

Представим, что пользователю необходимо выполнить преобразование «Разбиение цикла». На рис. 1 видно, что применению выбранного преобразования к фрагменту из примера мешают дуги истинной и анти зависимостей. Список автоматически сгенерированных вопросов изображен на рис. 2.

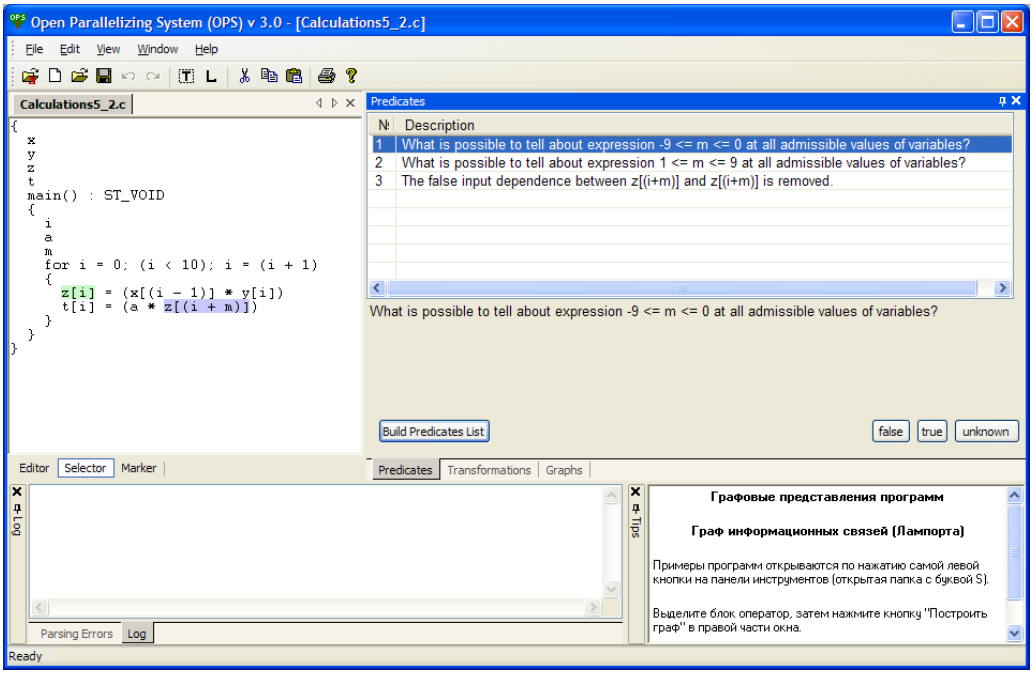

Рис. 2. Вопросы к пользователю, сгенерированные ДВОР.

На рис. 3 изображен уточненный ГИС для случая, когда пользователь отвечает на первый вопрос «true».

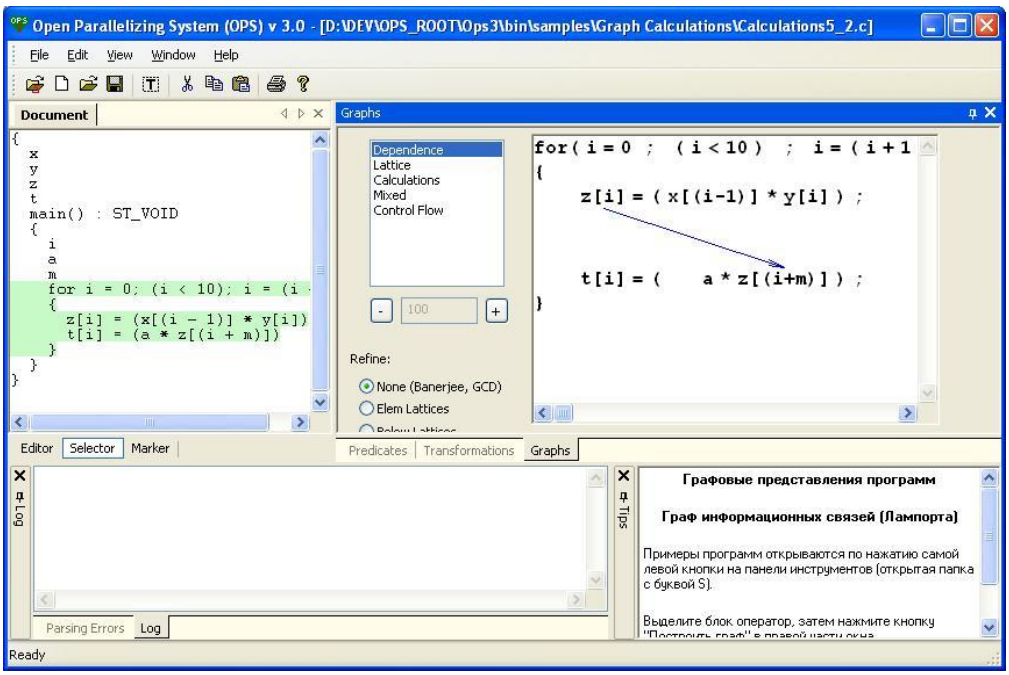

Рис. 3. Граф информационных зависимостей для случая –9<=m<=0. Теперь цикл можно разрезать.

На рис. 3 видно, что на ГИС осталась только дуга истинной информационной зависимости, ведущая сверху вниз, поэтому цикл можно разрезать.

4. Межпроцедурный анализ

Межпроцедурный анализ – способ анализа программы, учитывающий ее разделение на подпрограммы. При межпроцедурном анализе [8] программа рассматривается целиком как набор взаимосвязанных подпрограмм, в отличие от других видов анализа, которые обычно имеют дело с отдельными блоками кода или операторами, как, например, преобразования циклов. Если требуется проводить преобразования участков программы, содержащих вызовы функций, межпроцедурный анализ становится необходим инструментом. Так, если в цикле встречается вызов функции, то для определения того, является ли цикл распараллеливаемым, нужно проанализировать поведение вызываемой функции. Например, имеет ли эта функция побочные эффекты (меняет ли глобальные переменные) и какие из переданных ей параметров она изменяет при работе. Это в свою очередь может потребовать анализа псевдонимов: какие переменные могу ссылаться на один и тот же участок памяти. Ответ на этот вопрос позволит точнее определить возможные воздействия вызова функции на тот набор переменных, которые были доступны ей при выполнении. Все это служит одной цели: более точного построения графа информационных зависимостей. Так, вызов некоторой функции с переданным ей указателем на переменную в следующем участке программы:

```
int x, y, z;
function(&x, &y, &z) ;
```

может быть на самом деле генератором этой переменной. Межпроцедурный анализ программы в этом случае может ответить на вопрос, является ли оператор вызова функции в нашем примере генератором для переменных x, y и z. Вполне возможно, что в теле функции function значение первого параметра не меняется, а меняется, скажем, значение третьего. В таком случае этот вызов функции – генератор переменной z. Эту информацию следует учи-

тывать при построении графа информационных связей. Без межпроцедурного анализа мы будем вынуждены или не рассматривать программы с вызовами функций вообще, что недопустимо, или предполагать наихудший сценарий: любой вызов функции может влиять на любые глобальные переменные и на любые переданные не по значению параметры.

Для проведения межпроцедурного анализа необходимы некоторые инструменты. Так, это граф передачи управления и граф вызовов подпрограмм. В рамках проекта ДВОР реализовано построение этих графов. Граф потока управления с вершинами - операторами и дугами – передачами управления между операторами предназначен для использования при анализе программ, в том числе с целью более точного определения информационных зависимостей. Этот граф определяет порядок выполнения операторов программы при ее последовательном выполнении. Граф вызовов подпрограмм, в котором вершины соответствуют подпрограммам, а дуги – их вызовам, используется при межпроцедурном анализе для определения информационных зависимостей между отдельными подпрограммами. Этот граф хранит информацию обо всех вызовах подпрограмм, совершаемых в теле каждой отдельной подпрограммы.

Проведение межпроцедурного анализа позволяет построить улучшенный граф информационных связей, в котором каждый вызов функции может быть генератором какого-либо переданного ей аргумента или глобальной переменной. При этом дуги зависимости могут быть между вызовом функции и другим оператором и между двумя вызовами. Это может быть в случае, если одна из функций использует значение некоторой глобальной переменной, которой другая функция что-то присваивает внутри своего тела или если первая функция меняет значения параметров. Используя граф вызовов подпрограмм, граф передач управления и улучшенный граф информационных связей, можно осуществить распараллеливание исходной программы, при котором минимальными единицами выполнения будут подпрограммы. То есть, имея некоторый набор вычислительных устройств, мы сможем назначить каждому вызову функции процессор, на котором она будет выполняться. При этом внутри тел функций нужно расставить операторы синхронизации. Синхронизации должны находиться перед теми операторами, которые используют результаты выполнения другой функции. Эти операторы синхронизации должны находиться на каждом пути по графу передачи управления от генератора к ближайшему (на этом пути) использованию. Эти операторы можно расставить по дугам графа информационных связей, ведущим из вызовов функций или из операторов с вызовами функций. Рассмотрим, например, следующий код:

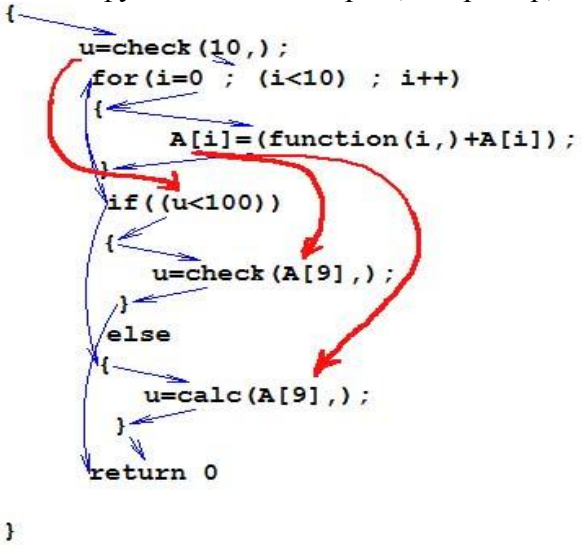

Рис. 4. Участок программы с дугами графа передачи управления и дугами зависимости по данным. Из генератора переменной u дуга зависимости идет в оператор, следующий за циклом. Поэтому вычисление значения check(10) можно выполнять параллельно с операторами цикла;

На рис. 4 можно добавить оператор синхронизации перед условным оператором, и тогда можно будет выполнять цикл параллельно с вычислением значения функции check() вызванной до этого цикла. Сам цикл тоже можно вычислять параллельно, но и в ветке then, и в ветке else нужны операторы синхронизации для того, чтобы значение A[9] было вычислено до операторов ветки.

Список литературы

1. Серебрянский А.И., Сыч Г.А. Обзор распараллеливающих компиляторов. // Системное программирование. 2008. Вып. 3. СПб. С. 157-177
2. R. Allen, K. Kennedy Optimizing compilers for Mordern Architetures // Morgan Kaufmann Publisher, Academic Press, USA, 2002. 790 p.
3. Воеводин В.В. Математические модели и методы в параллельных процессах// М.: Наука, гл. ред. физ.-мат. лит., 1986, 296 с.
4. Ахо, Альфред В., Лам, Моника С., Сети, Равви, Ульман, Джеффри Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2008. 1184 с.
5. Дроздов А.Ю, Владиславлев В.Е. Межпроцедурный анализ указателей // Информационные технологии. Приложение № 2. 2005.
6. Steven Muchnick. Advanced compiler design and implementation. Morgan Kaufmann, 3rd ed., 1997. 856 p.
7. Арапбаев Р.Н. Анализ зависимостей по данным: Тесты на зависимость и стратегии тестирования. Диссертация на соискание ученой степени кандидата физико-математических наук. ИСИ СО РАН им. А.П. Ершова, г. Новосибирск, 2008, 116 с.
8. А. С. Антонов. Современные методы межпроцедурного анализа программ. // Программирование, 1998 г. № 5.