ЗАДАЧИ СТАТИЧЕСКОГО РАСПАРАЛЛЕЛИВАНИЯ ПОСЛЕДОВАТЕЛЬНЫХ ПРОГРАММ

(Основные задачи статического распараллеливания, возникающие при отображении алгоритмов, задаваемых последовательными программами, на параллельные компьютеры с распределенной памятью)

Будем в качестве целевого суперкомпьютера[1] рассматривать многопроцессорный компьютер с распределенной памятью. К компьютерам с распределенной памятью относятся, в частности, мультикомпьютерные вычислительные системы типа белорусско-российского суперкомпьютера СКИФ. Во многом мы будет также ориентироваться на многоядерные персональные компьютеры и на графические процессоры.

На практике параллельные алгоритмы получают преобразованием разработанных ранее последовательных алгоритмов. Новые оригинальные параллельные алгоритмы, за исключением процесса сдваивания и алгоритма параллельной матричной прогонки, оказались не пригодными для практического использования. Они требуют слишком большого числа процессоров, очень много памяти, труднореализуемые коммуникации.

Даже математически эквивалентные последовательные алгоритмы могут иметь разные вычислительные свойства, в том числе и разные параллельные свойства. Но алгоритмов разработано очень много. Из них, как правило, можно выбрать подходящий для распараллеливания.

Будем считать, что алгоритм задан последовательной программой. Такое представление позволяет точно изложить содержание математической записи алгоритма, не оставляет неопределенности в отношении порядка выполнения операций. Будем также считать, что рассматриваемые программы и фрагменты программ принадлежат линейному классу[2]. Это означает:

– исполняемым оператором может быть только оператор присваивания, правая часть которого является арифметическим выражением; операторы-вызовы функций запрещены;

– повторяющиеся операции описываются только с помощью циклов, эквивалентных do-циклам языка программирования Fortran; шаги изменения параметров циклов равны +1;

– операторы перехода передают управление "вниз" по тексту (но не создают циклических конструкций);

– индексные выражения переменных, границы изменения параметров циклов и условия передачи управления задаются, в общем случае, аффинны-

[1]Целевой суперкомпьютер — это компьютер, на котором требуется реализовать параллельную версию алгоритма.

[2]В литературе используются также термины "аффинные гнезда циклов" и "алгоритмы с аффинными зависимостями".

ми функциями от параметров циклов и внешних переменных[3], т.е. неоднородными формами, линейными по совокупности параметров циклов и внешних переменных программы (запрещены фрагменты: $i * j$, $N_1 * N_2$, if$(a(i, j)...)$);

– параметры функций и внешние переменные программы целочисленные.

Многие программы принадлежат линейному классу или могут быть приведены к линейным с помощью различных преобразований. Примеры таких преобразований: прямая подстановка переменных при вычислении параметров циклов (чтобы все переменные зависели только от параметров циклов, а не от других переменных: выражения типа $a(b(i, j))$ запрещены), замена циклических конструкций типа goto на do-циклы, замена произведения внешних переменных новой внешней переменной.

Для отображения алгоритмов, задаваемых последовательными программами, на параллельные компьютеры с распределенной памятью, требуется

1) распределить операции алгоритма между процессорами,

2) распределить данные (элементы массивов) между процессорами,

3) установить порядок выполнения операций в каждом процессоре,

4) организовать обмен данными.

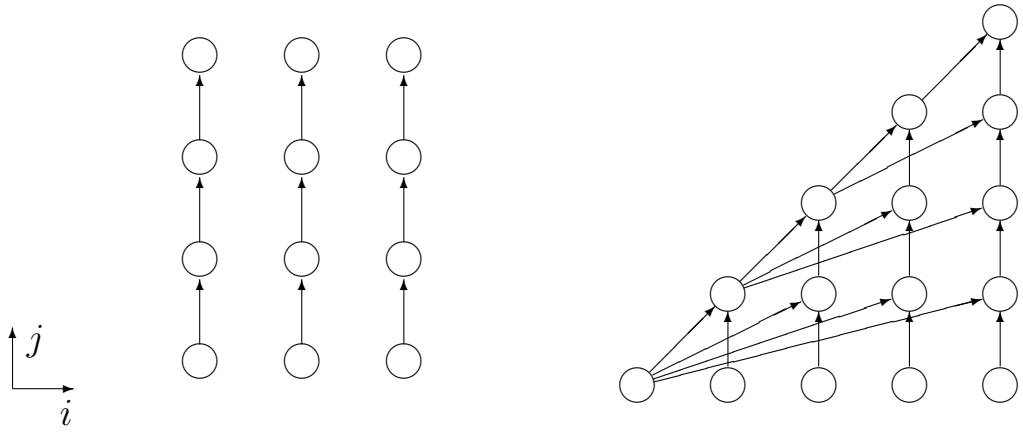
Это задачи общие. Далее мы рассмотрим более конкретные задачи. Основное внимание следует уделять распараллеливанию гнезд циклов, так как именно циклами часто описывается большая часть вычислений, и основной ресурс параллелизма относится к циклам.

1. Поиск информационных зависимостей, получение информационной структуры алгоритма

Любые процедуры распараллеливания должны опираться на точное или избыточное описание зависимых операций алгоритма. Если для распараллеливания использовать неполное описание зависимостей, то необходима проверка корректности полученного параллельного алгоритма.

Знание одних только зависимостей позволяет определять параллельные циклы, во многих случаях организовывать параллельные процессы вычислений, выявлять избыточные вычисления, восстанавливать из программы математические соотношения.

[3]Внешние переменные программы — переменные, значение которых не изменяется и не определено до начала выполнения программы.



Пусть в первом из алгоритмов, представленных изображенными графами, внешним циклом является цикл по i. Анализ зависимостей позволяет сделать вывод, что нет зависимостей между операциями, приписанными точкам с различным значением i. Следовательно, внешний цикл является параллельным.

Пусть во втором алгоритме внешним циклом является цикл по j. При фиксированном j зависимости между операциями отсутствуют. Следовательно, внутренний цикл является параллельным.

Во многих случаях получение информационной структуры алгоритма является трудной задачей и требует применения специальных методов.

2. Обнаружение потенциального параллелизма

Согласно третьему закону Амдала ускорение вычислительной системы не может превзойти обратной величины доли последовательных вычислений. Например, если доля последовательных вычислений составляет всего 1%, то ни при каком числе процессоров нельзя добиться ускорения более чем в 100 раз.

Таким образом, прежде чем параллелизм использовать, его нужно выявить, желательно как можно больше, чтобы знать возможности выполнения операций алгоритма одновременно.

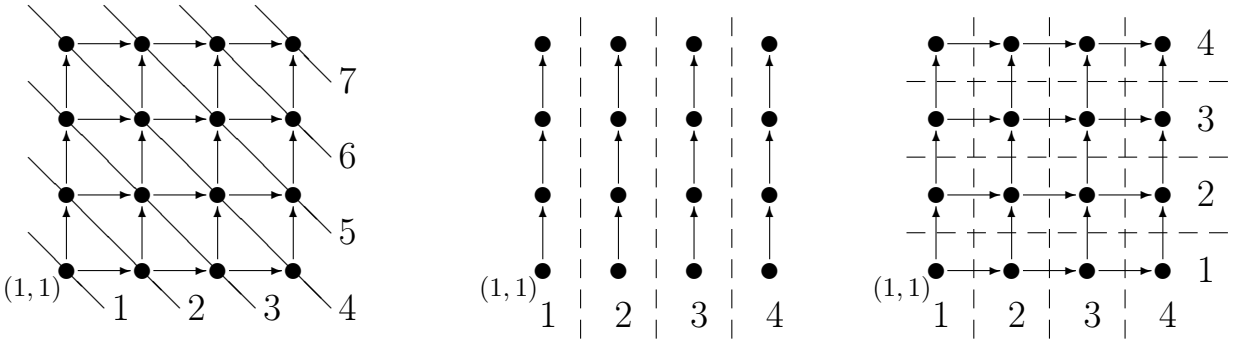
Мы уже видели, что параллельные циклы можно обнаруживать посредством анализа зависимостей.

Более общий математический аппарат для распараллеливания дает таймирование (scheduling). Методы таймирования позволяют с помощью так называемых таймирующих функций получать параллельные множества операций, организовывать параллельные вычислительные процессы, строить параллельные формы алгоритма. Здесь мы будем использовать равносильное таймирующей функции понятие — развертка графа алгоритма.

Рассмотрим вещественный функционал, определенный на вершинах

графа алгоритма. Функционал называется разверткой графа, если он не убывает вдоль дуг графа. Развертка называется строгой, если функционал вдоль дуг графа возрастает. Развертка называется расщепляющей, если ее значения равны на любых двух вершинах, соединенных дугой.

Строгие развертки задают параллельные формы алгоритма. К одному ярусу параллельной формы относятся поверхности уровня развертки. Расщепляющие развертки задают параллельные множества. Знание хотя бы двух независимых разверток позволяет задать параллельные последовательности вычислений, конвейерный параллелизм. Сказанное иллюстрируется с помощью рисунков.

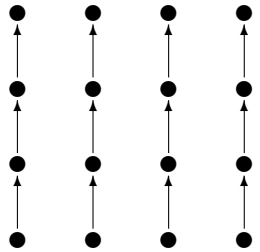


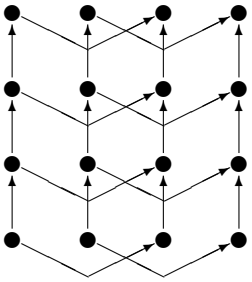
На первом рисунке ярусы параллельной формы алгоритма задаются строгой разверткой $t(i,j) = i + j - 1$. На втором рисунке параллельные множества задаются расщепляющей разверткой $t(i,j) = i$. Третий рисунок поясняет задание конвейерного параллелизма с помощью двух независимых разверток $t(i,j) = i$ и $t(i,j) = j$. Одна из разверток задает номер процессора, другая — порядок выполнения процессором операций.

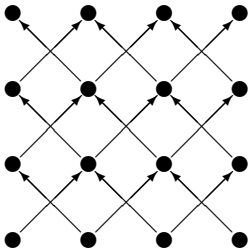
3. Выявление независимых частей программы

Выявление независимых частей программы, то есть параллельных множеств, — случай обнаружения параллелизма, особенно важного для компьютеров с распределенной памятью.

В ряде случаев каждое параллельное множество может быть представлено гиперплоскостью пространства итераций; в этом случае число независимых частей пропорционально размеру задачи. Способы получения такого параллелизма называются методами гиперплоскостей (Lamport). В других случаях, независимо от размера задачи, параллельных множеств два или несколько. Методы, позволяющие выявлять такие разбиения алгоритма на независимые части называются методами параллелепипедов. Такое название обусловлено тем, что получаемые в результате области независимых итераций можно представить в виде параллелепипедов или в виде вырожденных параллелепипедов.







Параллельные множества в методах гиперплоскостей задаются аффинными таймирующими функциями, а в методах параллелепипедов — модульными аффинными таймирующими функциями.

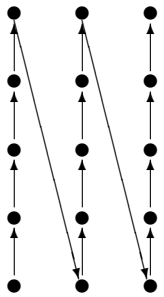
В реальных алгоритмах, представленных программной единицей, декомпозиция на независимые части возможна не часто. Примерами могут быть алгоритмы умножение матрицы на вектор (см. первый рисунок) и матрицы на матрицу, а также алгоритм Якоби решения задачи Дирихле для уравнения Пуассона (разбивается на два блока независимых вычислений, см. третий рисунок).

4. Предварительное преобразование алгоритма[4]

Предварительное преобразование может применяться для улучшения параллельных свойств алгоритма.

Приведем пример. Рассмотрим один из алгоритмов умножения матрицы порядка N на вектор и несколько упрощенный соответствующий граф зависимостей ($N = 3$) :

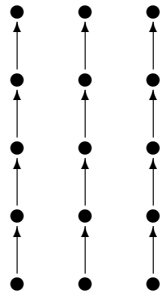
do $i = 1, N$
  $r = 0$
  do $j = 1, N$
    $r = r + a(i, j)b(j)$
  enddo
  $c(i) = r$
enddo



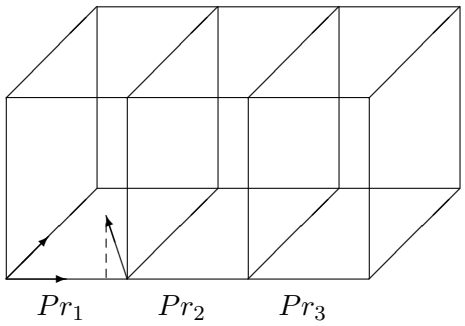
Алгоритм является последовательным: переменная r порождает зависимости (в том числе и так называемые ложные зависимости), связывающие последовательно все итерации гнезда циклов. Поэтому нельзя организовать параллельные вычисления, использующие общую память (нельзя, например, запустить несколько потоков на многоядерном процессоре). После применения стандартного приема увеличения размерности массива получаем хорошо распараллеливаемый алгоритм:

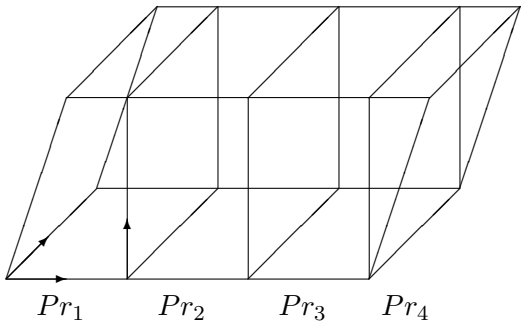
[4]Этот этап получения параллельных алгоритмов может отсутствовать.

```
do  i = 1, N
    r(i) = 0
    do  j = 1, N
        r(i) = r(i) + a(i, j)b(j)
    enddo
    c(i) = r(i)
enddo
```



Предварительное преобразование применяется также для улучшения коммуникационной структуры синтезируемого параллельного алгоритма и (или) для возможности получения координатных информационных разрезов алгоритма.





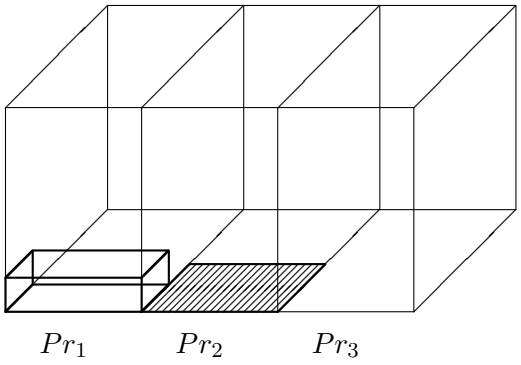
На рисунках схематично представлены множества операций, выполняемых на одном процессоре, некоторого исходного и преобразованного алгоритмов; векторы символизируют дуги в графе алгоритма. Для реализации преобразованного алгоритма требуется меньше коммуникаций. Кроме того, преобразованный алгоритм допускает информационные разрезы по всем координатам итерационного пространства, что важно для получения зернистых алгоритмов (т.е. алгоритмов, операции которых объединенными в макрооперации).

5. Тайлинг

Тайлинг (tiling) — получение макроопераций (получение зерна вычислений, тайлов). Под зерном вычислений (или тайлом) понимается множество операций алгоритма, выполняемых атомарно: вычисления, принадлежащие одному зерну, не могут прерываться синхронизацией или обменом данными, требуемыми для выполнения этих операций. Обычно результаты вычислений одного тайла формируют пакет данных для пересылки другому процессору. Тайлинг очень часто применяется как при использовании компьютеров с распределенной памятью, так и при использовании многоядерных персональные компьютеры и графических процессоров.

При тайлинге каждый цикл разбивается на два цикла: глобальный, параметр которого определяет на данном уровне вложенности порядок вычисления тайлов, и локальный, в котором параметр исходного цикла изменяется

в границах одного тайла. Допускается вырожденное разбиение цикла, при котором все итерации относятся к глобальному циклу или все итерации относятся к локальному циклу. Локальные циклы переставляются с глобальными и становятся самыми внутренними.

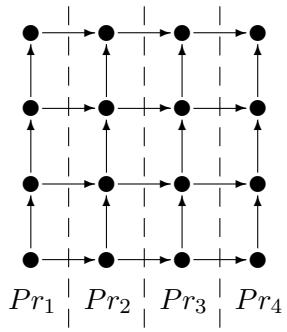


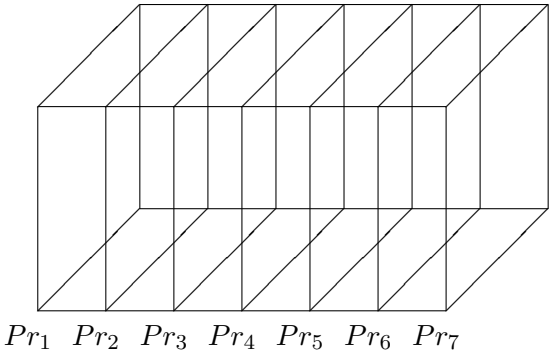
На рисунке приведены примеры трехмерного и двумерного тайлов.

6. Организация параллельных вычислительных процессов

Пусть алгоритм, заданный гнездом циклов, имеет один или несколько параллельных самых внешних циклов. В этом случае в явном виде заданы параллельные множества операций, и алгоритм разбивается на независимые части. Если параллельными являются один или несколько самых внутренних циклов, то в явном виде задана параллельная форма алгоритма. В обоих случаях параллельные вычислительные процессы или параллельные потоки (нити) можно организовать, выполняя независимо операции параллельных циклов.

Если алгоритм не имеет параллельных циклов, но обладает внутренним параллелизмом (достаточным условием для этого является существование двух независимых разверток графа алгоритма), то можно выделить параллельные последовательности вычислений, т.е. можно указать упорядоченные множества операций алгоритма, которые могут выполняться одновременно. В этом случае явного указания операций, выполняемых одновременно, не требуется.





Пример — рассмотренное ранее для двумерного алгоритма получение конвейерного параллелизма с помощью двух независимых разверток. Опера-

ции, приписанные точкам на рисунке, можно выполнять параллельно, если одна координата точек задает номер процесса, другая — порядок выполнения операций. После выполнения операции данные передаются процессу с большим номером. Аналогично осуществляется организация параллельных вычислительных процессов и для алгоритмов большей размерности.

Как правило, для эффективной реализации алгоритма на параллельном компьютере следует сначала осуществить тайлинг, получить макрооперации и для распараллеливания рассматривать только глобальные циклы.

7. Согласование распределения операций и данных между процессорами. Приватизация массивов

Согласование распределения операций и данных по процессорам (alignment) необходимо для минимизации числа и объема обменов данными. Требуется распределить операции и массивы данных так, чтобы коммуникаций было как можно меньше.

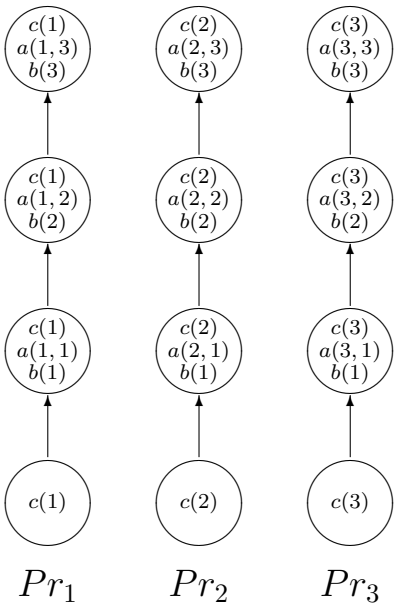
Операции обмена данными выполняются гораздо дольше вычислительных операций, поэтому производительность многопроцессорной вычислительной системы с распределенной памятью в значительной степени зависит от оптимальности решения задачи распределения данных.

Наиболее благоприятным результатом согласования распределения операций и данных является приватизации данных, т.е. выделение массивов или частей массивов, локализованных в процессорах. Элементы приватизированного массива требуются для вычислений только в одном процессоре и поэтому не участвуют в операциях обмена данными.

Приведем пример приватизации данных. Рассмотрим алгоритм перемножения матрицы A и вектора b порядка N; выходными данными алгоритма являются координаты вектора $c : c_i = \sum_{j=1}^{N} a_{ij} b_j$, $1 \leqslant i \leqslant N$.

$$
\begin{array}{l}
\text{do } \ i = 1, N \\
\quad S_1 : \ c(i) = 0 \\
\quad \text{do } \ j = 1, N \\
\qquad S_2 : \ c(i) = c(i) + a(i,j)b(j) \\
\quad \text{enddo} \\
\text{enddo}
\end{array}
$$

Изобразим граф алгоритма ($N = 3$), вершины помечены необходимыми для выполнения операций элементами массивов.



Пусть элементы c_i массива c вычисляются в процессоре Pr_i. Тогда процессор Pr_i приватизирует элемент c_i и i-ю строку массива a. В каждом процессоре используется весь массив b.

8. Выделение используемых процессорами массивов

Для каждого процессора необходимо определить массивы или части массивов, используемые процессором в вычислениях. Соответственно изменяются размеры массивов и текст программы. В примере из прошлого пункта в каждом процессоре вместо двумерного массива a будет использоваться одномерный массив.

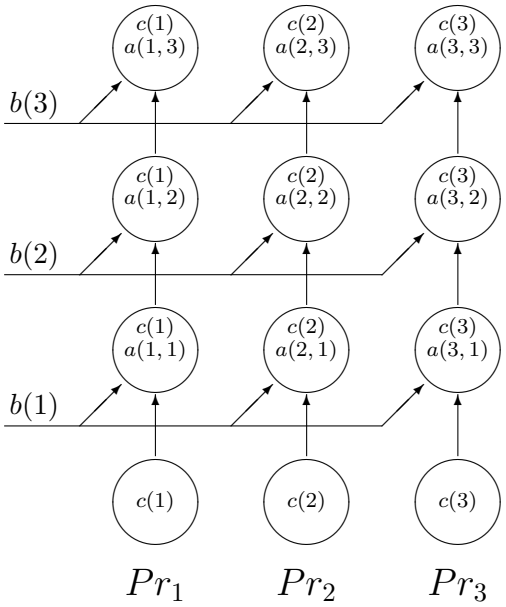
9. Оптимизация числа и объема временных массивов

Объем массивов данных существенно влияет на скорость работы с памятью. С другой стороны, уменьшение объема массивов может ухудшить параллельные свойства алгоритма. Например, в рассмотренном выше примере умножения матрицы на вектор использование массива r меньшего объема уменьшает ресурс параллелизма. Поэтому при извлечении параллелизма важно уметь оптимизировать размеры массивов за счет расширения или сжатия массив. Под сжатием массива понимается аннулирование какой-либо размерности массива и замена его в коде программы новым массивом меньшей размерности. Подразумевается, что такая замена допустима и результаты вычислений не изменятся. Отметим, что задача выявления массивов, для которых возможно сжатие, является NP-полной.

10. Получение коммуникационных операций. Структурирование коммуникаций

На практике даже оптимальное согласование распределения операций и данных между процессорами не исключает необходимости вводить коммуникации. Возникает задача генерации (включения) коммуникационных операций в параллельный алгоритм. Теория включения в код коммуникационных операций разработана недостаточно, поэтому для организации обмена данными требуется хорошее понимание реализуемого алгоритма.

На параллельных компьютерах с распределенной памятью структурированные коммуникации, например бродкаст (broadcast), а также трансляция (translation) данных выполняются быстрее, чем большое количество коммуникаций точка-точка (point-to-point) между процессором, в локальной памяти которого хранится данное, и каждым процессором, который использует или вычисляет это данное. Поэтому желательно выявлять возможность организации таких коммуникаций, и только в случаях, когда не удается структурировать, использовать коммуникации точка-точка. В примере перемножения матрицы и вектора (пункт 7) массив b можно не хранить в локальной памяти процессоров, а осуществить бродкаст каждого данного b_1, b_2, b_3 ко всем процессорам:



11. Улучшение локальности данных

Локальность данного при выполнении какой-либо операции означает расположение (хранение) его непосредственно перед использованием в памяти с быстрым доступом, где оно оказалось вследствие участия в более ранней операции. При многопроцессорной обработке памятью с быстрым доступом считается локальная память процессора, при однопроцессорной — кэш. Локальность алгоритма — это вычислительное свойство алгоритма, отражаю-

щее совокупность сведений о локальности его данных. Улучшить локальность означает реорганизовать вычисления таким образом, чтобы добиться более частого использования памяти с быстрым доступом. Улучшение локальности данных называют еще локализацией данных.

Локальность параллельного алгоритма, предназначенного для реализации на компьютерах с распределенной памятью, определяет коммуникационные затраты: чем лучше локальность, тем меньше аргументов операций требуется доставлять в процессоры.

Для улучшение локальности при однопроцессорной обработке необходимо, чтобы операции алгоритма, которые используют одни и те же или соседние с точки зрения хранения в памяти элементы массива данных, выполнялись бы друг за другом. Если хранение элементов массива осуществляется по строкам, то близко расположенными в памяти элементами массива являются элементы, отличающиеся только последней координатой.

Пусть A — квадратная матрица порядка N. Рассмотрим алгоритм LU разложения матрицы компактной схемой Гаусса:

```
do i = 1,N-1
   S1 : b(i)=1/a(i,i)
   do j =i+1,N
      S2 : a(j,i)=a(j,i)b(i)
      do k =i+1,N
         S3 : a(j,k)=a(j,k)-a(j,i)a(i,k)
      enddo
   enddo
enddo
```

Основные вычисления определяются оператором S_3. Если хранение элементов массива осуществляется по строкам, то на всех вхождениях массива a в оператор S_3 данные с точки зрения локальности данных используются эффективно. Действительно, элементы $a(j,k)$ при изменении параметра k самого внутреннего цикла являются элементами одной строки массива. То же самое можно сказать и относительно элемента $a(i,k)$, а элемент $a(j,i)$ вообще не изменяется на итерациях k. Но если переставить местами циклы по i и j, то получим некоторое улучшение свойства локальности: к прежним достоинствам добавится использование элементов одной строки $a(j,i)$ при смене параметра внутреннего цикла по i. Соответствующее допустимое преобразование задается многомерным таймированием: $t^{(1)}(i) = (i,i,i),\ t^{(2)}(i,j) = (j,i,i),\ t^{(3)}(i,j,k) = (j,i+1,k)$, и приводит исходный алгоритм к следующему виду:

```
S'_1 : b(1)=1/a(1,1)
do j = 1,N-1
   S'_2 : a(j,1)=a(i,1)b(1)
   do i =2,j-1
      S_2 : a(j,i)=a(j,i)b(i)
      do k =i,N
         S_3 : a(j,k)=a(j,k)-a(j,i-1)a(i-1,k)
      enddo
   enddo
   S'_3 : a(j,j)=a(j,j)-a(j,j-1)a(j-1,j)
   S_1 : b(j)=1/a(j,j)
   do k =j+1,N
      S''_3 : a(j,k)=a(j,k)-a(j,j-1)a(j-1,k)
   enddo
enddo
S''_2 : a(N,1)=a(N,1)b(1)
do i =2,N-1
   S'''_2 : a(N,i)=a(N,i)b(i)
   do k =i,N
      S'''_3 : a(N,k)=a(N,k)-a(N,i-1)a(i-1,k)
   enddo
enddo
S''''_3 : a(N,N)=a(N,N)-a(N,N-1)a(N-1,N)
```

С ростом N время вычислений преобразованного и непреобразованного алгоритмов на однопроцессорном компьютере может отличаться существенно.

12. Генерация кода

Генерация кода после распараллеливающих и (или) улучшающих структуру программы преобразований во многих случаях вызывает затруднение. В рассмотренном примере LU-разложения код значительно усложнился после сравнительно несложного преобразования — перестановки циклов со сдвигом одного из циклов. На практике, по-видимому, разумно получать пусть не оптимальные, но приемлемые по выбранным критериям преобразования циклов, приводящие к упрощенной генерации кода. Можно воспользоваться средствами автоматизации генерации кода [4, 5].

13. Автоматизация распараллеливания

Получение и программная реализация параллельных алгоритмов задача, как правило, довольно трудоемкая; эффективный параллельный алгоритм удается получить не всегда. Обращаться к суперкомпьютеру следует

тогда, когда последовательные реализации алгоритма себя полностью исчерпали. В связи с этим особую важность и актуальность приобретает разработка методов автоматизированого распараллеливания последовательных программ, а также создание систем автоматизированного распараллеливания, адаптирующих программы к компьютерам с параллельной архитектурой.

Литература

1. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002. – 600 с.

2. Воеводин В. В. Вычислительная математика и структура алгоритмов. – Москва: Изд-во МГУ, 2006. – 112 с. http://parallel.ru/info/parallel/voevodin/

3. Антонов А. С., Воеводин Вл. В. Эффективная адаптация последовательных программ для современных векторно-конвейерных и массивно-параллельных супер-ЭВМ // Программирование. 1996. № 4. С. 37–51.

4. CLooG: The Chunky Loop Generator. http://www.cloog.org

5. TLOG: A parameterized tiled loop generator. http://www.cs.colostate.edu/ ln/TLOG/