УДК 004.415.2

АЛГОРИТМ ДЕКОМПОЗИЦИИ ПОСЛЕДОВАТЕЛЬНЫХ ПРОГРАММ НА ПАРАЛЛЕЛЬНЫЕ СОСТАВЛЯЮЩИЕ

А.А. Афоняшин[1]

ОАО "НИИ суперЭВМ", 117437, г. Москва, ул. Академика Волгина, 33

В данной статье описывается алгоритм декомпозиции программ, написанных на стандартном языке программирования, на части, выполняющиеся параллельно и время выполнения которых не превышает заданного времени T.

Ключевые слова: многопроцессорные вычислительные системы и параллельные вычисления; программный комплекс, автоматически декомпозирующий программу.

ВВЕДЕНИЕ

Всё большее распространение и применение получает использование *многопроцессорных* вычислительных систем и параллельных вычислений. Оба этих фактора неотделимы друг от друга, так как если нужно добиться повышения производительности и значительного *сокращения времени вычисления и/или* обработки данных, то без распараллеливания задач на различные вычислительные узлы никак не обойтись. При этом остро встаёт вопрос о координации вычислительных узлов между собой, так как на производительность системы влияет огромное количество факторов, и все их требуется учесть и найти оптимальное решение.

Но не всегда есть возможность использования многопроцессорной системы. Это может быть обусловлено как отсутствием времени на изучение и дальнейшее распараллеливание задачи для конкретной системы, *так и отсутствием знаний* и практики использования данных типов систем. Возможным решением данной проблемы может стать программный комплекс, который будет автоматически декомпозировать программу, написанную на стандартном языке, на части, выполняющиеся независимо друг от друга. Данное решение позволит снизить требования для пользователей и программ, выполняемых на многопроцессорной вычислительной системе.

1. ВХОДНЫЕ ДАННЫЕ АЛГОРИТМА

Входными данными для алгоритма является программа пользователя. После того, как программа пользователя загружена в комплекс, её необходимо представить в таком виде, который позволил бы проанализировать *её и успешно декомпозировать*. В качестве такого представления был выбран граф зависимости по данным (DDG – Data Dependence Graph). Узлом данного графа может являться:

- *с*трока, содержащая простые операторы (+, *, -, = и т.д.);
- строка, содержащая сложные операторы (операторы цикла, операторы ветвления и т.д.).

Построение графа зависимости по данным (DDG).

Вначале строится граф и определяются узлы графа (рис. 1).

```
int fib(int n)
{
  int a, b, c;
  a = b = c = 1;
  for (; n > 1; n--) {
    c = a + b;
    a = b;
    b = c;
  }
  return c;
}
```

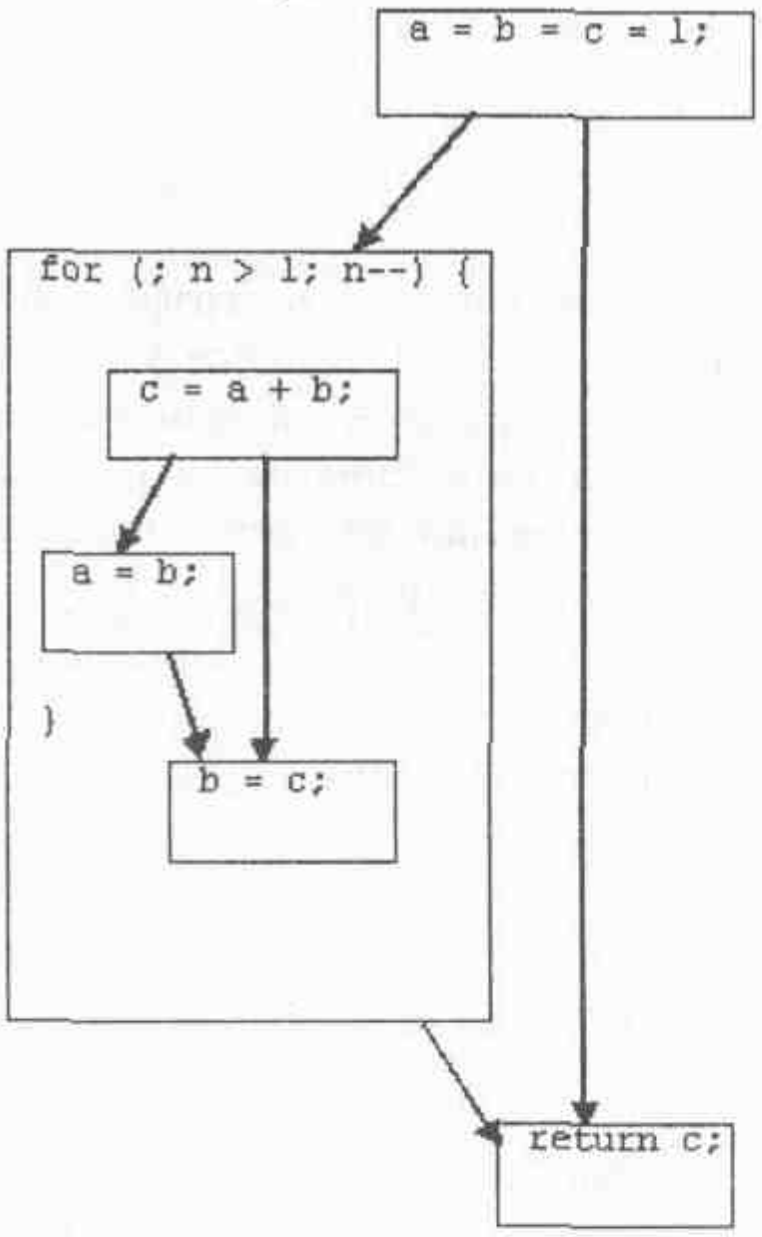


Рис. 1. Пример функции и ее DDG

Узлом такого графа может являться:

- простой оператор (сложение, умножение, сравнение, присваивание и т.д.);
- более сложный оператор (условный оператор, оператор цикла и т.д.);
- граф зависимостей по данным следующего уровня, инкапсулирующий свойства соответствующего программного блока.

Дуги графа DDG представляют собой зависимости по данным между узлами. Более формально, пусть u и v – узлы DDG, причем в последовательной программе u предшествует v. Дуга (u, v) входит в граф тогда и только тогда, когда между u и v есть зависимость по данным одного из трех типов:

- «запись-чтение» (в узле v необходимы результаты вычислений узла u);
- «чтение-запись» (в узле v записывается переменная, значение которой считывается в u);
- «запись-запись» (оба узла записывают одну и ту же переменную).

Наличие одной из указанных зависимостей по данным между узлами говорит о том, что при параллельном выполнении программы для получения результатов, совпадающих с последовательной версией, необходимо выполнить u раньше, чем v.

Граф зависимостей по данным является ориентированным ациклическим графом. Это объясняется тем, что циклы в DDG означают наличие циклических зависимостей по данным, возможным в свою очередь только в операторах цикла исходной программы. Но циклы, как и другие сложные операторы, раскрываются на более низком уровне иерархии, обеспечивая разрыв таких зависимостей по данным.

Пример функции и ее графа зависимостей по данным приведен на рис. 1. DDG, состоит из трех узлов: двух простых узлов и оператора цикла, раскрывающегося в DDG второго уровня.

Граф зависимостей по данным строится для каждой функции программы. Алгоритм построения состоит из следующих этапов:

- построение графа потока управления программы;
- выбор программных блоков, которые будут узлами текущего уровня иерархии DDG;

- нахождение зависимостей по данным между этими узлами с помощью алгоритма достигающих определений;
- если необходимо продвинуться на следующий уровень иерархии и достроить граф.

Для того чтобы отразить на графе побочные эффекты работы функции, в графе вводится специальный узел EXIT. Все узлы, генерирующие побочные эффекты (например, осуществляющие запись в глобальную переменную), связаны дугой с узлом EXIT. Все этапы алгоритма разделения на нити, описанные ниже, работают с представлением программы в виде графа зависимостей по данным.

2. ПОСТРОЕНИЕ РАСПИСАНИЯ

В предыдущих разделах был описан процесс построения DDG-графа программы и вычислены времена выполнения узлов данного графа. Теперь наша задача поместить данный граф на требуемое число процессоров и постараться уложиться в заданное время T.

Для того чтобы уложиться в требуемое время выполнения, программа должна быть некоторым образом разбита на подзадачи, которые и будут выполняться на вычислительных узлах многопроцессорной системы. При построении такого расписания мы должны учитывать не только время выполнения отдельных задач на вычислительных узлах системы, но также учитывать время обмена данными между задачами. Для решения данной проблемы был выбран генетический алгоритм построения расписания, так как:

- генетические алгоритмы достаточно эффективны при решении дискретных экстремальных задач, плохо поддающихся решению традиционными методами;
- стохастика, используемая генетическими алгоритмами, позволяет надеяться, что мы не пропустим решения, «не поддающегося» той или иной эвристике;
- вычислительная сложность генетических алгоритмов для большинства приложений растёт практически линейно с ростом размера задачи и числа оптимизируемых параметров;
- простота кодируемости решения задачи позволяет сконцентрироваться на реализации самого алгоритма.

Расписание выполнения задач для данного DDG-графа будет кодировать следующим образом:

$$\text{<битовая строка>} \equiv \left(\bigcup_{i=1}^{N} \text{<поле процесса>}_i\right);$$

$$\text{<поле процесса>} \equiv (\text{<номер процессора>} \bigcup \text{<приоритеты интервалов>});$$

$$\text{<приоритеты интервалов>} \equiv \left(\bigcup_{j=1}^{I_i} \text{<приоритет интервала>}_j\right),$$

где $\bigcup$ – операция конкатенации (склейки) битовых строк.

С учётом следующих особенностей автоматизированное выделение процессов (задач) в графе DDG сама по себе – очень сложная и трудоёмкая задача. К тому же, очень сложно при формировании задач учесть объёмы локальной памяти вычислительного узла и соотнести их с формируемой задачей. Поэтому мы рассматриваем рабочий интервал – интервал времени, в течение которого выполняемый код не обращается к другим частям кода, как самостоятельную и независимую задачу. Таким образом, мы устраняемся от проблемы объединения узлов графа DDG программы в задаче и переходим к проблеме разбиения узлов графа на рабочие интервалы. Проблема разбиения узлов на рабочие интервалы сводится к проблеме разбиения сложных операторов, таких как операторы цикла, операторы условия и т.д., на рабочие интервалы, удовлетворяющие определению.

Так как основным параметром нашей поставленной задачей является оптимизация времени выполнения программы, то в связи с этим все циклы должны иметь заранее фиксированную длину, т.е. перед оператором цикла в комментариях должно быть указано

максимальное количество итераций. Так как иначе, если не будет указано максимальное количество итераций для данного цикла, мы не сможем однозначно определить время его выполнения, что приведёт к неоднозначности в решении. К примеру:

```
// Start = 0 End = 10
For (i=Start; i<End; ++i) {
…
}
…
```

Зная максимальное время выполнения каждого цикла, мы можем повторить тело цикла соответствующее количество раз с учётом счётчика цикла, таким образом уходя от сложных операторов и переходя к рабочим интервалам (рис. 2).

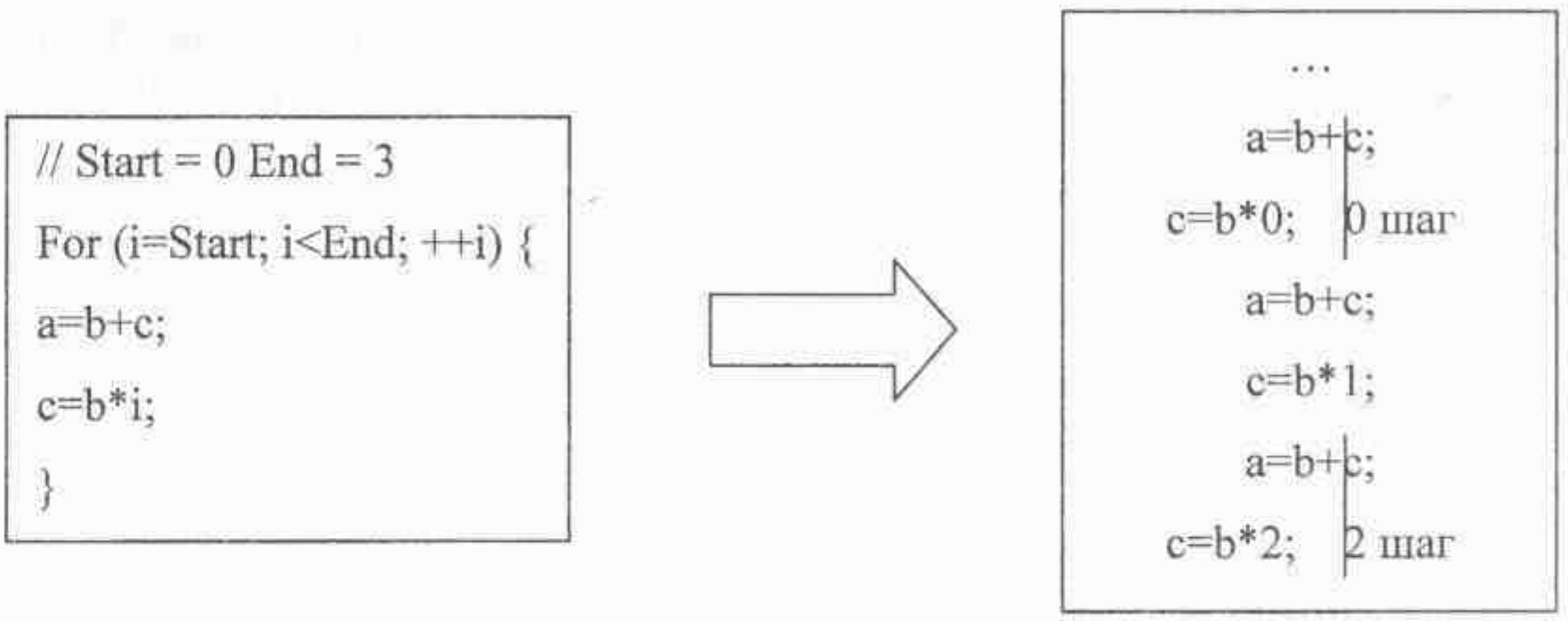


Рис. 2. Переход от сложных операторов к рабочим интервалам

Устранившись таким образом от оператора цикла, мы всё-таки должны соблюсти очередность выполнения повторяемых частей. Несмотря на то, что внутри каждой из таких частей сохраняется собственная зависимость, у нас непосредственно отсутствует зависимость по данным между повторяемым телом цикла. Для того чтобы соблюсти порядок выполнения тела цикла, мы должны определить зависимости между повторяемым кодом, как это показано ниже на рис. 3.

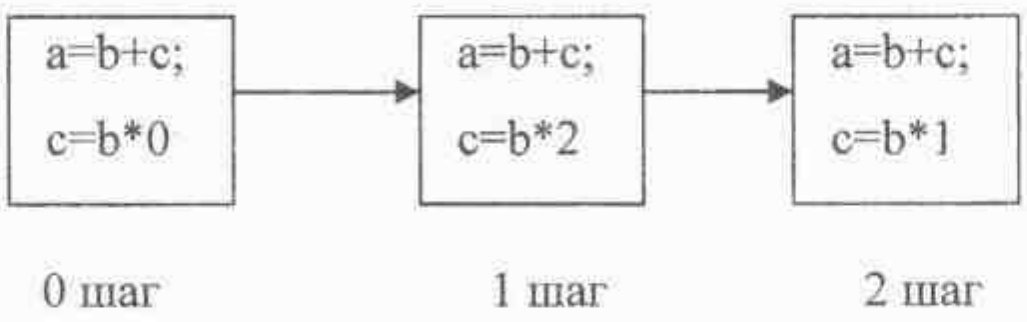


Рис. 3. Определение зависимостей между итерациями цикла

Несмотря на то, что мы связываем итерации цикла между собой, мы должны соблюсти очередность выполнения операций, т.е. первая итерация не может начаться, пока не поступят все данные, необходимые для данного цикла, что отображено в графе DDG, где цикл является собственным узлом графа. Для этого мы создадим некую фиктивную вершину, входные ребра которой будут соответствовать зависимостям по данным между оператором цикла и предыдущими узлами. Так же зависимые от оператора цикла следующие узлы не могут начать своё выполнение, пока не будет завершена последняя итерация цикла, таким образом первую и последнюю итерации цикла мы соединяем с фиктивными вершинами (рис. 4).

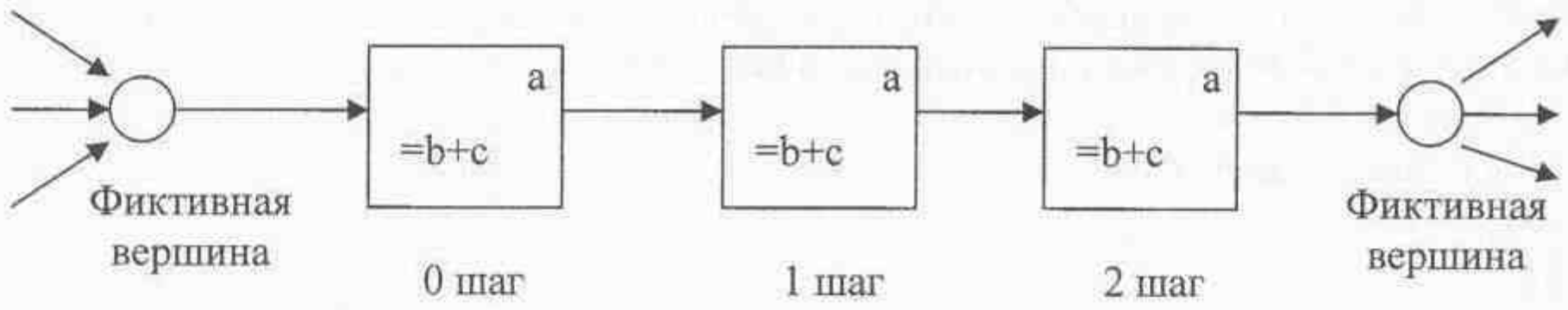


Рис. 4. Ввод фиктивных вершин

Таким же образом можно поступить и с остальными сложными операторами и функциями, так же вводя фиктивные вершины начала и конца и соединяя их со всеми узлами сложного оператора (не цикла) или функции (рис. 5).

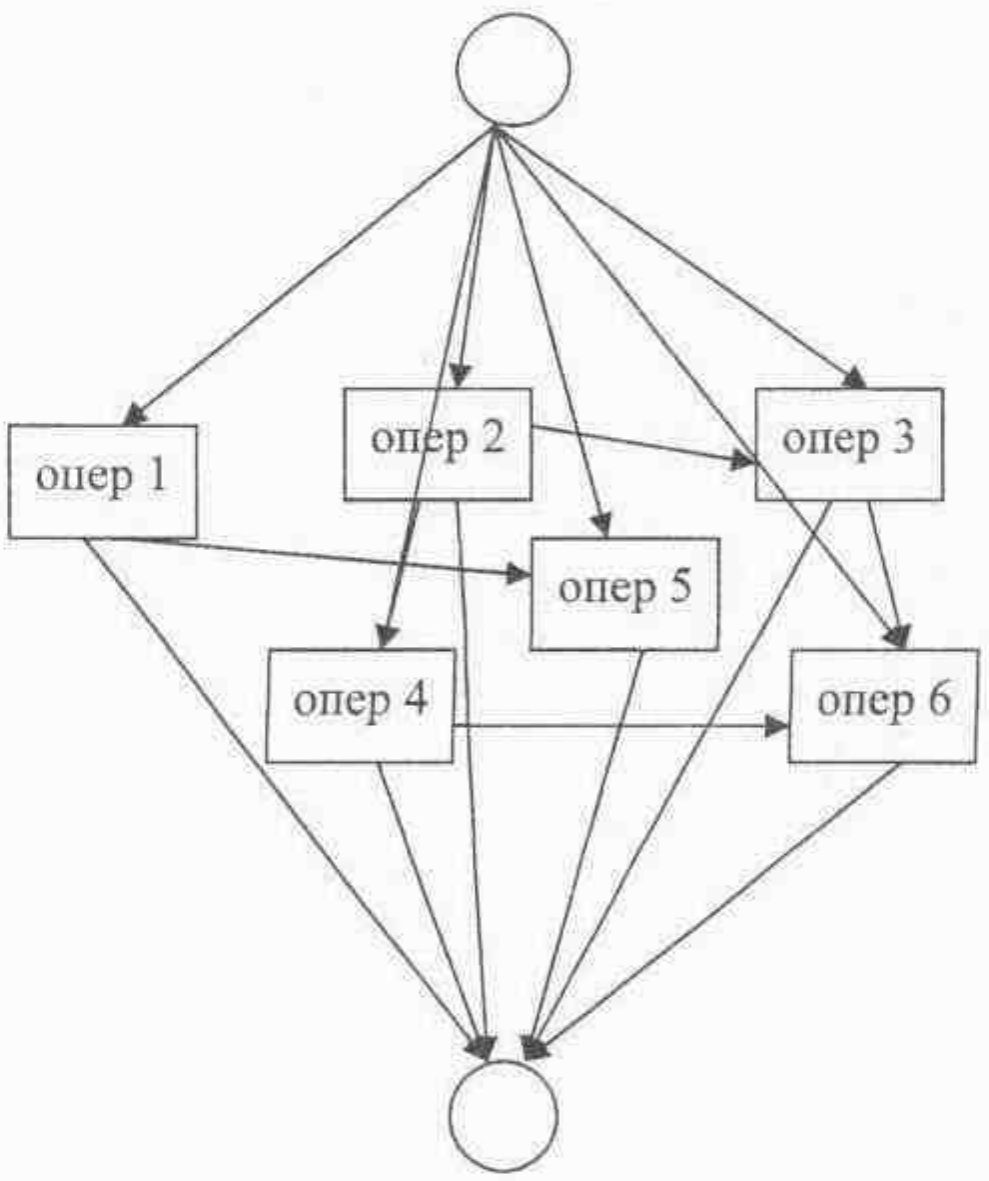


Рис. 5. Использование фиктивных вершин в функции

Используя фиктивные вершины, мы таким образом сохраняем частичную упорядоченность графа DDG-программы.

После того, как мы подобным образом преобразовали сложные операторы и функции, можно преобразовать способ кодирования решения для генетического алгоритма. Теперь он будет выглядеть следующим образом:

$$\text{<битовая строка>} \equiv \left(\bigcup_{i=1}^{N} \text{<номер процессора>}_i\right),$$

где $\bigcup$ – операция конкатенации (склейки) битовых строк.

Так как мы разбили весь наш граф DDG-программы на рабочие интервалы, следовательно, нам не нужно использовать поле <приоритет интервалов> и <поле процесса>,

а сразу назначать рабочему интервалу номер процессора, таким образом упрощая способ кодирования решения. Хотя, конечно, мы и увеличиваем размер битовой строки, так как мы увеличили число рабочих интервалов.

Выбрав способ кодирования решения, мы также должны и суметь восстановить расписание по битовой строке. Процедура восстановления расписания и получения оценки времени его выполнения по данной кодировке выглядит следующим образом. Все рабочие интервалы W графа DDG-программы разбиваем на три множества:

$$W = W1 \bigcup W2 \bigcup W3,$$

где $W1$ – множество рабочих интервалов, для которых определено время начала выполнения, $W2$ – множество рабочих интервалов, у которых все предшественники (в соответствии с отношениями E принадлежат множеству $W1$, $W3$ – множество рабочих интервалов, у которых хотя бы один из предшественников не принадлежит множеству $W1$. Приведём алгоритм восстановления расписания по битовой строке.

1. Полагаем $W1 = \varnothing$, $W2$ содержит рабочие интервалы без предшественников, $W3$ – все остальные рабочие интервалы.

2. Находим в $W2$ рабочий интервал с наибольшим приоритетом и переносим его в $W1$, одновременно определяем время его инициализации (см. ниже).

3. Проверяем $W3$ с целью возможности переноса рабочих интервалов в $W2$ (все предшественники принадлежат $W1$).

4. Если $W2 \neq \varnothing$ или $W3 \neq \varnothing$, то переходим к шагу 2, иначе завершаем работу.

Одновременно с определением порядка осуществляется оценка времени выполнения прикладной программы:

$T = \max_{i \in M} (T_i)$ – время выполнения программы $PR`$;

$T_i = \max_{j \in N_i} (t_j^{ini} + t_j), \quad i \in M$ – время работы i-го процессора;

$t_j^{ini} = \max(t_j^d, (t_{j-1}^{ini} + t_{j-1})), \quad j \in N_i, \quad i \in M$ – время инициализации j-го рабочего интервала;

$t_j^d = \max_{k \in \prec_{in}} (t_k^{ini} + t_k), \quad j \in N_i, \quad i \in M$ – время получения данных j-м рабочим интервалом,

где t_j^{ini} – время инициализации j-го рабочего интервала, j – порядковый номер рабочего интервала на процессоре (соответствие между порядковым номером интервала на процессоре и его номером в графе поведения прикладной программы определено однозначно), t_j – время выполнения j-го рабочего интервала, t_j^d – время получения рабочим интервалом j данных от других рабочих интервалов, находящихся с ним в отношении порядка (множество предшественников $\prec_{in}$ – определяется в соответствии с отношениями $\{R \succ (PR`)\}$ и $\{R \rightarrow (PR`)\}$), M – число процессоров в ВС, N_i – число рабочих интервалов на i-ом процессоре, $PR`$ – новое множество узлов в соответствии с приведёнными выше преобразованиями.

Критерии останова и целевая функция. В качестве целевой функции была выбрана функция вида:

$$F(t) = T,$$

где T вычисляется способом, описанным выше. Минимизация функции данного вида и является нашей прямой задачей.

Критерий останова выглядит следующим образом: алгоритм останавливает свою работу, как только находится решение, удовлетворяющее условию

$$F(T) \leq T_{заданное.}$$

В противном случае алгоритм завершает свою работу, если за заданное число итераций K^* значение целевой функции улучшено не было.

Селекция. Метод селекции представляет собой двухэтапный метод. На первом этапе селекция происходит по так называемой *пропорциональной селекции*, т.е. битовая строка со значением целевой функции F_i даёт строку в новую популяцию $F_i/\overline{F}$ потомков, где $\overline{F}$ – среднее значение целевой функции всей популяции. В результате применения данного типа селекции в новую популяцию строки со значением целевой функции выше среднего дадут более одного потомка, а строки с меньшим значением целевой функции – менее одного потомка. Очевидно, что пропорциональная селекция даёт дробное число потомков в новую популяцию. Чтобы дополнить новую популяцию и разнообразить число потомков, будем использовать метод рулетки, который добавит некоторый элемент случайности в новую популяцию, тем самым привнеся в неё разнообразие возможных решений.

В методе рулетки для каждой строки выделим сектор с центральным углом $2\pi F_i/\overline{F}$. Строка получает потомков, если случайно выбранное число от 0 до 2π попадает в сектор, соответствующий этой строке. Данный метод выполняется до тех пор, пока мы не насытим новую популяцию необходимым количеством потомков.

Ниже приведён пример операции селекции.

Допустим у нас есть 5 битовых строк, со значениями целевой функции

$F_1 = 10$ $F_2 = 15$ $F_3 = 8$ $F_4 = 33$ $F_5 = 24$;

среднее значение целевой функции $\overline{F} = 18$;

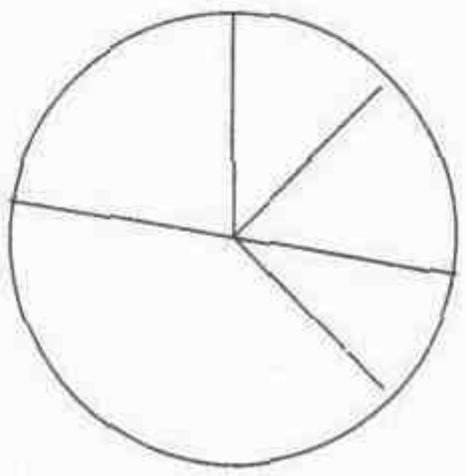
вычисляем значения $F_i/\overline{F}$ для каждой битовой строки соответственно:

1-ай строка = 0,56;
2-ая строка = 0,83;
3-я строка = 0,44;
4-ая строка = 1,83;
5-ая строка = 1,33.

Это означает, что в новую популяцию строки 4-ая и 5-ая дадут хотя бы 1-го потомка.

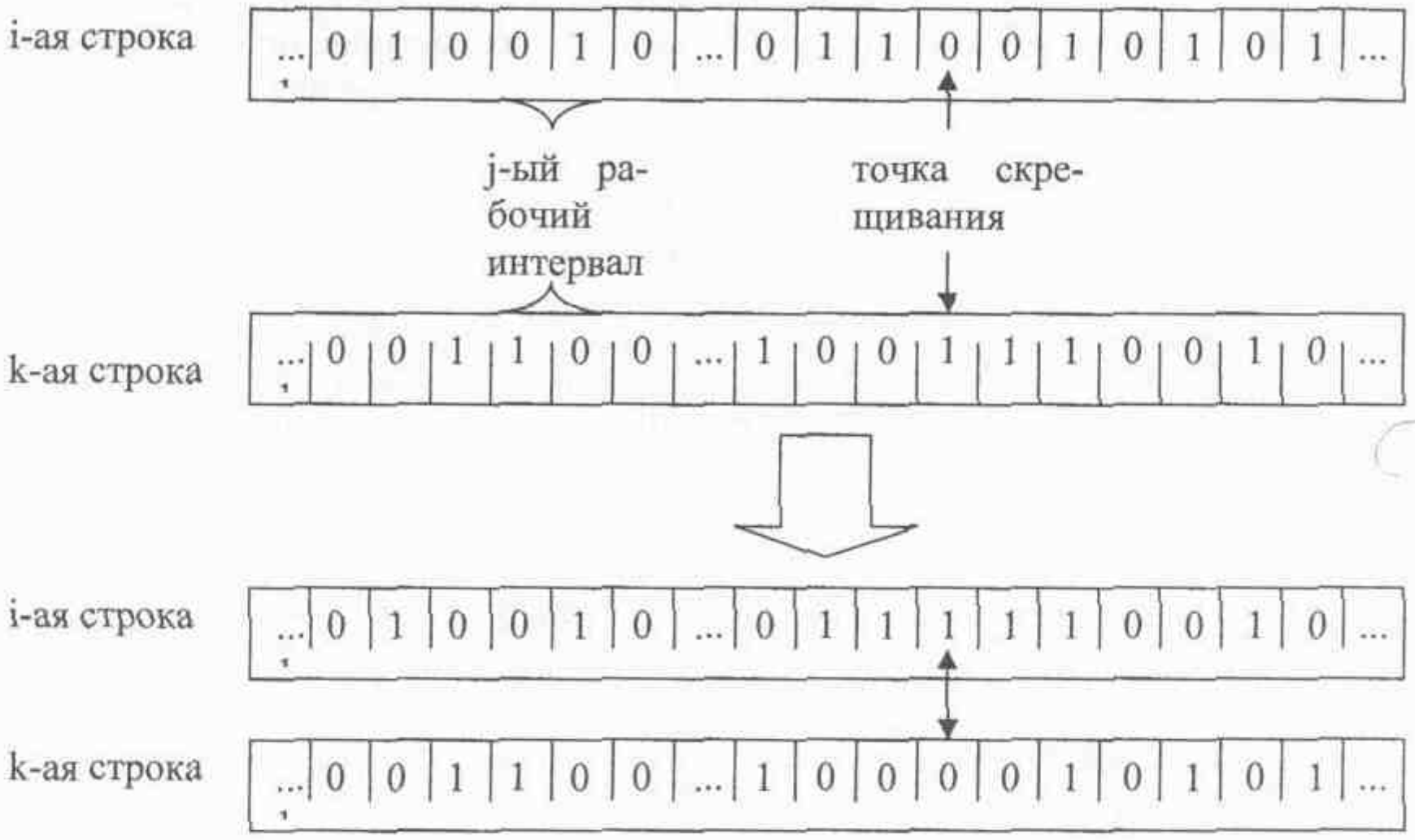
Теперь применяем метод рулетки для добирания числа потомков в новую популяцию. Для этого вычисляем для каждой строки величину $2\pi F_i/\overline{F}$:

1-ай строка = 2π*5/9;
2-ая строка = 2π*5/6;
3-я строка = 2π*4/9;
4-ая строка = 2π*11/6;
5-ая строка = 2π*4/3.



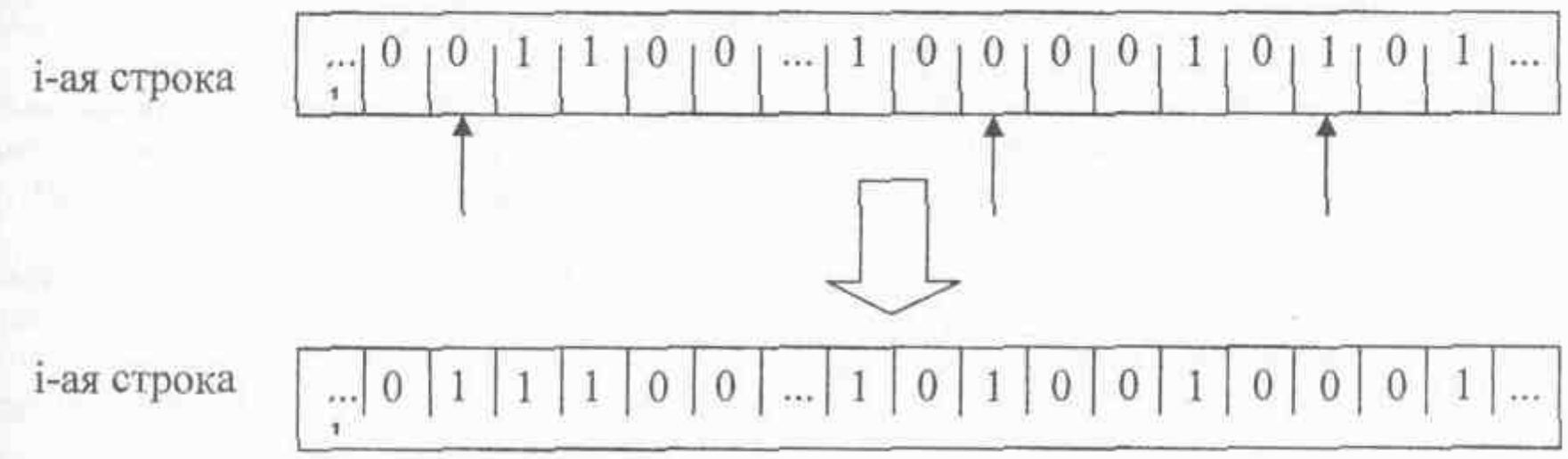
Далее генерируем случайное число от 0 до 2π, в сектор какой строки оно попало, ту строку и переносим в новую популяцию. И так до тех пор, пока окончательно не сформируем новую популяцию.

Скрещивание. В данном алгоритме используется операция одноточечного скрещивания со случайным выбором битовых срок для скрещивания и точки скрещивания. Ниже приведен пример одноточечного скрещивания двух битовых строк.



Использование многоточечного скрещивания существенно не повысило качество получаемого результата, но повысило вычислительную сложность.

Мутация. Программно операция мутации была реализована с помощью генератора случайных чисел. Для каждого бита битовой строки генерировалось случайное число; если сгенерированное число оказывалось выше порога мутации p_m, то бит инвертировался, если ниже, то оставался неизменным. На примере стрелками указаны инвертируемые биты, для которых сгенерированное число оказалось выше порога мутации p_m.



Определившись со способом кодировки, алгоритмом восстановления расписания по битовой строке и операциями генетического алгоритма можно всё это объединить в один единый алгоритм.

3. АЛГОРИТМ ПОСТРОЕНИЯ РАСПИСАНИЯ

1) Из представления программы в форме $\{R \succ (PR)\}$ и $\{R \rightarrow (PR)\}$) перейти к представлению $\{R \succ (PR`)\}$ и $\{R \rightarrow (PR`)\}$);

2) пронумеровать все рабочие интервалы, т.е. присвоить каждому узлу графа свой уникальный номер, который в дальнейшем не может быть изменён;

3) сгенерировать случайным образом популяцию размера P;

4) оценить здоровье популяции, т.е. для каждой битовой строки вычислить значение целевой функции (провести процедуру восстановления расписания);

5) выполнить операцию селекции и отобрать особи в новую популяцию;

6) выполнить операцию скрещивания: случайным образом выбрать пары для скрещивания, затем для каждой скрещиваемой пары сгенерировать случайное число. Если число больше порога скрещивания, то происходит скрещивание, процедура которого описана выше. Если сгенерированное число меньше порога скрещивания, то пара остается неизменной;

7) если критерий останова не достигнут, то возвращаемся к шагу 4.

ЗАКЛЮЧЕНИЕ

В результате тестирования были получены следующие диапазоны параметров генетического алгоритма, которые оптимальным образом сочетают в себе время, затрачиваемое на поиск удовлетворительного решения, и качество получаемого решения.

Таким образом:

- разработанный программный комплекс позволяет упростить и ускорить процесс разработки программ для многопроцессорных вычислительных систем;
- решены проблемы кодирования решения для генетического алгоритма;
- получен набор параметров, которые оптимальным образом позволяют найти решение.

Алгоритм может быть использован для различных многопроцессорных систем. Универсальность алгоритма заключается в том, что нужно всего лишь указать синтаксис языка программирования, а также основные характеристики вычислительной системы, концепция же алгоритма остаётся неизменной для любого языка программирования и вида многопроцессорной вычислительной системы.

Литература

1. Костенко В.А. Генетические алгоритмы: синтез структур вычислительных систем / В.А. Костенко, Р.Л. Смелянский, А.Г. Трекин: тезисы докладов Всероссийской научной конференции // Фундаментальные и прикладные аспекты разработки больших распределенных программных комплексов (сентябрь 1998 г., Новороссийск). – М.: Изд-во МГУ, 1998 – С. 35-41.

2. Сальников А.Н. Прототип системы разработки параллельных программ для гетерогенных многопроцессорных систем / А.Н. Сальников, А.Н. Сазонов, М.В.Карев.

3. Сазонов А.Н. Статистическое построение расписания выполнения параллельной программы с использование генетических алгоритмов / А.Н. Сазонов: материалы Междунар. науч.-практ. семинара // Высокопроизводительные параллельные вычисления на кластерных системах. – Н. Новгород, 2001. – С. 149-158.

4. Гайсарян С.С. О некоторых задачах анализа и трансформации программ / С.С. Гайсарян, А.В. Чернов, А.А. Белеванцев, О.Р. Маликов, Д.М. Мельник, А.В. Меньшикова: Труды Ин-та системного программирования РАН.

DECOMPOSITION'S ALGORITHM OF THE CONSECUTIVE PROGRAMS TO PARALLEL COMPONENTS

A.A. Afonjashin

OAO of "SUPERCOMPUTER NII", 117437, Moscow, street Academician Volgina, 33

The article describes the decomposition's algorithm of the programs written on the standard programming language on parts which are carried out in parallel and time of which performance does not exceed the given time T.

Key words: multiprocessor computer systems and parallel calculations; programmatic complex, automatically dekompose program.