Министерство высшего и среднего специального образования РСФСР

ЛЕНИНГРАДСКИЙ ОРДЕНА ЛЕНИНА
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ
имени М. И. КАЛИНИНА

И. Д. БУТОМО, Д. Ф. ДРОБИНЦЕВ, А. Е. ПИТЬКО

МЕТОДЫ РАСПАРАЛЛЕЛИВАНИЯ АЛГОРИТМОВ И ИХ РЕАЛИЗАЦИЯ В ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ

Учебное пособие

Ленинград
1980

УДК 681.3.06:51

Данное учебное пособие содержит материал по курсам «Вычислительные системы» и «Алгоритмические языки и программирование» и предназначено студентам специальности «Автоматизированные системы управления», а также слушателям факультета повышения квалификации преподавателей по специальности АСУ и специального факультета переподготовки кадров по новым, перспективным направлениям науки и техники.

В пособии излагаются прикладные вопросы теории параллельного программирования, основные приемы построения и анализа параллельных программ, элементы построения операционных систем для параллельного выполнения программ в вычислительных системах.

Игорь Дмитриевич Бутомо, Дмитрий Федорович Дробинцев, Андрей Евгеньевич Питько

Методы распараллеливания алгоритмов и их реализация в вычислительных системах

Учебное пособие

Редактор *Н. В. Бакк*

Корректор *Л. А. Онохова*

Сдано в набор 25.11.80. Подписано к печати 25.12.80. М-28987.
Формат бумаги 60 × 90$^{1}/_{16}$. Бумага тип. № 3. Гарнитура литературная.
Печать высокая. Усл. печ. л. 5. Уч.-изд. л. 5. Тираж 1500. Заказ 1295.
Цена 20 коп. Темплан 1980 г., поз. 1180.
Издание ЛПИ им. М. И. Калинина. 195251, Ленинград, Политехническая, 29.

Лаборатория полиграфических машин ЛПИ им. М. И. Калинина
195251, Ленинград, Политехническая ул., 29.

ВВЕДЕНИЕ

С момента создания ЭВМ перед разработчиками средств вычислительной техники встал вопрос о повышении эффективности использования вычислительных машин и систем на их основе. В качестве критериев эффективности могут использоваться различные показатели в зависимости от назначения вычислительной системы. Для систем, работающих в неоперативном режиме (пакетная обработка), основным показателем эффективности является *производительность* — количество задач, решаемых за единицу времени.

Для систем оперативной обработки, в которых существенным является время реакции системы (системы с разделением времени, управляющие системы), основным показателем эффективности является *время ответа* — интервал между моментом поступления задания в систему и моментом его выполнения. При этом для каждой системы под заданием понимается некая стандартная работа, например, запрос в системе резервирования авиабилетов, обработка сигнала с датчика объекта управления в управляющей системе и т. п.

Средствами, обеспечивающими заданные значения показателей эффективности, являются структурные, аппаратные и программные решения, принимаемые разработчиками при создании конкретной вычислительной системы.

В области программных средств исторически первыми были начаты разработки в области языков программирования и трансляторов, существенно облегчающие труд пользователя и повышающие общую производительность системы «человек—машина». Прогресс в области аппаратуры связан с разработкой быстродействующих элементов и устройств на их основе — операционных блоков, блоков оперативной памяти, блоков сопряжения, периферийных устройств.

Под структурными средствами понимаются приемы и методы, обеспечивающие совмещенное (параллельное) функционирование отдельных узлов и блоков вычислительной системы; здесь существенную роль играет соответствующая программная поддержка. Первой попыткой совмещения в рамках

однопроцессорной ЭВМ можно назвать просмотр команд вперед, при котором выявлялась независимость соседних команд последовательной программы и перекрытие во времени их использования. Такую работу осуществляли блоки опережения, буферирующие команды и данные, позволяющие сократить количество обращений к ОЗУ. Последовательное развитие этого приема привело впоследствии к разработке конвейерных вычислительных систем.

Приемы совмещения (распараллеливания) внутри отдельных блоков ЭВМ относят к так называемому локальному параллелизму. Значительное повышение производительности ЭВМ было получено, когда наряду с основным процессором в состав ЭВМ был введен процессор ввода-вывода, осуществляющий ввод-вывод информации независимо от основного процессора (центрального процессора — ЦП). Такая форма совмещения работы отдельных блоков ЭВМ называется глобальным параллелизмом. Большинство существующих вычислительных систем используют разумное сочетание глобального и локального параллелизма. И если локальный параллелизм обеспечивается в основном аппаратными средствами, то глобальный параллелизм обеспечивается за счет соответствующих программных средств. Таким программным средством в современных ЭВМ является операционная система (ОС). Назначение ОС состоит в таком распределении аппаратных ресурсов — памяти, ЦП, каналов ввода-вывода, внешних устройств и программных ресурсов — обслуживающих программы типа трансляторов, загрузчиков и т. п. между заданиями пользователей, при котором соответствующий показатель эффективности принимал бы экстремальное значение.

Для системы пакетной обработки максимума производительности можно добиться при определенном уровне мультипрограммирования — числе заданий, находящихся одновременно в оперативной памяти системы, организуя специальную смесь из заданий, требующих разного характера обслуживания. Например, смесь можно организовать из пяти заданий, два из которых требуют преимущественно счета на ЦП, два других загружают каналы ввода-вывода, а пятое занимает промежуточное положение. При этом все основные ресурсы будут загружены значительное время, простои оборудования можно свести к минимуму, а, значит, производительность может поддерживаться на высоком уровне.

Для систем оперативной обработки минимум времени ответа получается, если в системе присутствует только одно задание. При этом время ожидания решения равно нулю. И все равно, время ответа может быть недопустимо большим для

обеспечения управления сложным технологическим объектом с помощью ЭВМ обычной структуры.

Выход — в переходе на многопроцессорные структуры, когда ЦП представляет собой блок процессоров, связанных между собой для обеспечения обмена информацией при решении одной задачи, и к другой форме представления программы — параллельному программированию.

В чем заключаются проблемы практического параллельного программирования? Основная трудность — преодоление барьера и приемов последовательного программирования, связанного отчасти с особенностью человеческого мозга — мыслить последовательно. Главной целью последовательного программирования является экономия основных ресурсов ЭВМ — памяти и времени выполнения. Она заставляет программиста организовывать вычисления по возможности в одном цикле, использовать минимальное число переменных величин в программе и т. п. Иными словами, делать программу максимально связной с сильной зависимостью отдельных частей (операторов) через память.

Главной целью параллельного программирования является экономия времени выполнения программы. Эта экономия может быть достигнута только за счет дополнительного расхода ресурса памяти. Как мы увидим в дальнейшем, эта цель заставляет программиста организовывать независимые вычисления в независимых циклах, использовать для всех независимых результатов разные переменные и т. п., иными словами, делать программу минимально связной. Итак, первая проблема связана с формой представления параллельной программы, т. е. языками параллельного программирования. В свою очередь, эффективность той или иной формы зависит существенным образом от структуры вычислительной системы, на которой будет выполняться данная параллельная программа.

Вторая проблема связана с автоматизацией разработки параллельных программ, включающей создание алгоритмов преобразования обычных последовательных программ в параллельные. Эта автоматизация заполняет пробел между возможностями человека в написании параллельных программ и возможностями вычислительной системы реализовать программу в максимально параллельной форме, зависящей, может быть, только от количества процессоров. Третья проблема заключается в построении специальной программы, управляющей распределением фрагментов задач по процессорам системы, следящей за выполнением этих фрагментов на ресурсах системы, регулирующей и исключающей аварийные и конфликтные ситуации параллельного выполнения. Таким образом, речь идет о создании специальной операционной си-

стемы, основная задача которой — управление фрагментами с целью минимизации времени выполнения задания в целом.

В данном учебном пособии с разной степенью подробности отражены все три проблемы.

Для частичной иллюстрации задач и приемов, используемых при решении указанных проблем, рассмотрим следующий пример.

Рассмотрим программу на АЛГОЛе решения задачи вычисления в точке x непрерывной функции $\Phi(x)$, аппроксимирующей функцию $F(x_j)$, заданную на дискретном множестве точек $\{x_j\}$

$$\Phi(x) = \sum_{k=0}^{kmax} a_k P_k(x), \quad a_k = \sum_{l=0}^{lmax} F(x_l) P_k(x_l), \quad P_k(x) = \sum_{m=0}^{k} b_{km} x^m.$$

Здесь $\{P_k(x)\}$ — система полиномов Чебышева, ортогональная на заданном множестве точек $x_0, x_1, \ldots, x_{l_{max}}$.

В программе используются: массив коэффициентов ортогональных полиномов b, массив значений функции F, процедура, вычисляющая значения полиномов степени k в любой точке x

```
begin integer kmax, lmax; ввод (kmax, lmax);
      begin array b [0: kmax, 0: kmax]; real Φ, x, x0, a, h;
            array F[0: lmax]; integer i, j;
            real procedure ПОЛ(k, M, x); value x; real x;
            begin real s, y; integer m; s: = 0; y: = x;
            for m: = 0 step 1 until k do begin s: = s +
            + M[k, m] × y;
            y: = y × x end; ПОЛ: = s end;
            ввод(b, F, x0, h); Φ: = 0;
            for i: = 0 step 1 until kmax do begin a: = 0;
            for j: = 0 step 1 until lmax do
            a: = a + F[j] × ПОЛ(i, b, x0 + j × h);
            Φ: = Φ + a × ПОЛ(i, b, x) end; печать (Φ, x)
      end
end
```

Посмотрим, что можно сделать с данной задачей при попытке распараллелить ее решение. Распараллеливание связано с обеспечением независимого выполнения операторов программы. В нашем случае первым шагом в решении задачи должно быть обеспечение независимого выполнения операторов внешнего цикла по i, т. е. оператора внутреннего цикла по j и оператора Φ: =. Этого можно добиться введением массива a[0: $kmax$].

Фрагмент программы будет теперь выглядеть следующим образом:

for i: = 0 **step** 1 **until** *kmax* **do**
begin $a[i]$: = 0;
for j: = 0 **step** 1 **until** *lmax* **do**
$a[i]$: = $a[i] + F[j] \times$ *ПОЛ* $(i, b, x0 + j \times h)$;
Φ: = $\Phi + a[i] \times$ *ПОЛ* (i, b, x)
end;

Теперь коэффициенты $a[i]$ могут подсчитываться независимо от выполнения оператора Φ: =.

Вторым шагом будет попытка распараллелить выполнение оператора $a[i]$: =.

Для этого заметим, что вычисление полиномов *ПОЛ* можно отделить от выполнения оператора a: =, введя специальный массив R[0: *lmax*, 0: *kmax*]. Иными словами, мы пытаемся разукрупнить сложные операторы присваивания распараллеливанием выражений в правых частях (оператор Φ распараллеливать не будем).

Фрагмент программы будет выглядеть так:

for i: = 0 **step** 1 **until** *kmax* **do begin** $a[i]$: = 0;
z: = $x0$; **for** j: = 0 **step** 1 **until** *lmax* **do begin**
$R[i, j]$: = *ПОЛ* (i, b, z); z: = $z + h$;
$a[i]$: = $a[i] + F[j] \times R[i, j]$ **end**;
Φ: = $\Phi + a[i] \times$ *ПОЛ* (i, b, x) **end**;

Теперь коэффициенты $R[i, j]$ могут подсчитываться независимо от $a[i]$. Правда, необходимо обеспечить выполнение вспомогательного оператора z: = $z + h$ совместно с $R[i, j]$: =. Но как обеспечить параллельное выполнение указанных операторов? Одна из возможностей (третий шаг) — внешний цикл представить в виде трех отдельных блоков A, B и C:

A: **begin integer** i, j; **real** z; **for** i: = 0 **step** 1 **until** *kmax* **do**
begin z: = $x0$; **for** j: = 0 **step** 1 **until** *lmax* **do**
begin $R[i, j]$: = *ПОЛ* (i, b, z); z: = $z + h$ **end end end** A;
B: **begin integer** i, j; **for** i: = 0 **step** 1 **until** *kmax* **do**
begin $a[i]$: = 0; **for** j: = 0 **step** 1 **until** *lmax* **do**
$a[i]$: = $a[i] + F[j] \times R[i, j]$ **end end** B;
C: **begin integer** i; Φ: = 0;
for i: = 0 **step** 1 **until** *kmax* **do**
Φ: = $\Phi + a[i] \times$ *ПОЛ* (i, b, x) **end** C;

Теперь, если блокам A, B, C дать возможность выполняться параллельно, возникают следующие проблемы:

1. Блоки A и C не могут выполняться независимо, поскольку используют общий информационный ресурс — процедуру *ПОЛ*. Необходимы средства для обеспечения синхронизации выполнения блоков A и C при их работе с процедурой *ПОЛ*: если один блок обращается в данный момент к процедуре *ПОЛ*, то другой должен ждать конца этого обращения.

2. Блок B не может выполняться независимо от блока A, поскольку блок A поставляет ему коэффициенты $R[i, j]$. Необходимо обеспечить такую синхронизацию взаимодействия этих блоков, чтобы блок B ожидал, пока блок A не выработает очередной коэффициент $R[i, j]$.

3. Блок C не может выполняться независимо от блока B, так как блок B поставляет ему коэффициенты $a[i]$. Здесь также необходима синхронизация взаимодействий блоков B и C.

По существу, мы пришли к концепции асинхронных процессов, подробному рассмотрению которой посвящена отдельная глава пособия.

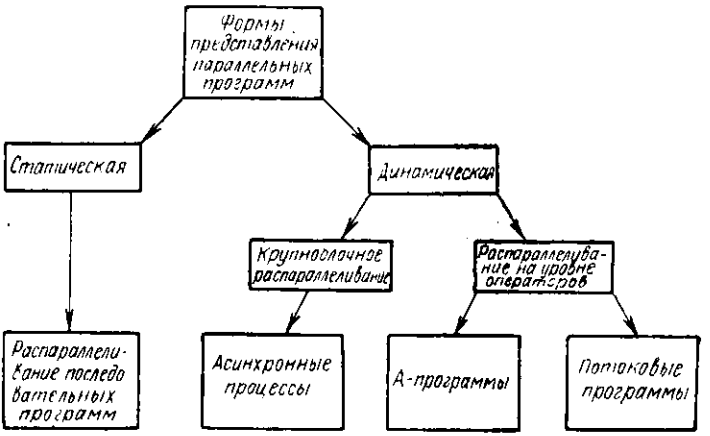


Рис. 1

В заключение введения в проблематику параллельного программирования введем классификацию методов распараллеливания, согласно которой строится материал основных глав пособия. На рис. 1 эта классификация представлена в графической форме. Существует два основных подхода к представлению программ для параллельного исполнения: статическое и динамическое распараллеливание. При статическом распараллеливании программа представляется в виде последовательности участков параллельности, состоящих из параллельных фрагментов (блоков), которые в свою очередь могут содержать участки параллельности и т. д. Язык статического параллельного программирования должен содержать средства явного указания параллельных ветвей и порядка выполнения участков параллельности. При реализации программы управление движется по тексту программы от одного участка параллельности до другого, расщепляясь на каждом участке на число независимых управлений, равное числу параллельных блоков в участке. Переход между участками парал-

лельности осуществляется после того, как все параллельные управления достигнут конца своих блоков.

При динамическом распараллеливании возможность параллельного выполнения участков программы не указывается явно. Возможный параллелизм выполнения участков программы задается в виде специальных управляющих предикатов, значения которых зависят от текущего состояния процесса выполнения программы. Для каждого участка в некоторые моменты времени путем вычисления его предиката (и независимо от других участков) выясняется возможность его выполнения.

Участок программы готов к выполнению, если для него подготовлены все необходимые исходные данные другими участками. Между участками могут существовать связи двух типов: информационные и логические (управляющие).

Предикат участка состоит из логических функций, аргументами которых являются переменные, отражающие эти связи.

При крупноблочном динамическом распараллеливании программу представляют в виде некоторого числа функционально законченных фрагментов (сопрограмм, процессов). Каждый фрагмент представляет собой последовательную программу в обычном понимании, дополненную специальными операторами, осуществляющими информационные и логические связи между отдельными фрагментами. Часто в фрагменте локализован какой-либо циклический участок исходной программы, осуществляющий вычисление сложной или громоздкой функции. Фрагменты выполняются асинхронно; один фрагмент может передать другому в какой-либо момент времени порцию информации, может затормозить или инициировать выполнение программы другого фрагмента. Все такие взаимодействия осуществляются на некоторых общих ресурсах (устройствах, памятях), разделяемых во времени независимо исполняющимися фрагментами. Поэтому требуется синхронизация независимого исполнения фрагментов, которая и реализуется с помощью указанных выше специальных операторов.

При желании использовать весь потенциальный параллелизм, присущий задаче, программу представляют в виде совокупности (набора) так называемых информационных операторов. Это могут быть операторы какого-либо языка высокого уровня (например, АЛГОЛА) или машинные команды.

Каждому информационному оператору приписывается пусковая функция — предикат от так называемых управляющих переменных. Управляющие переменные, выделенные в отдельной памяти, получают значения с помощью специальных управляющих операторов, по одному на каждый инфор-

мационный оператор. Управляющий оператор фиксирует факт выполнения информационного оператора присваиванием значения управляющей переменной, связанной с этим оператором. Таким образом, каждый исходный оператор теперь представляется тройкой: спусковая функция, информационный оператор, управляющий оператор (*А*-программа). Выполнением такой программы можно управлять различным образом. Например, сначала проверяются все спусковые функции. Информационные операторы, у которых значение спусковой функции «истина», выполняются. Затем срабатывают соответствующие управляющие операторы, изменяющие значения управляющих переменных. После этого цикл повторяется. Возможен и другой способ управления, при котором проверка спусковых функций осуществляется всякий раз после срабатывания хотя бы одного информационного (и управляющего) оператора. Практическая реализация *А*-программы наталкивается на промоздкость вычисления спусковых функций всех операторов программы всякий раз, когда необходимо выбрать на выполнение готовые операторы.

Существует подход, который существенно сокращает проверку готовности операторов, — управление выполнением программы потоком данных. Идея подхода состоит в том, чтобы вычисленные к рассматриваемому моменту времени результаты сами выбирали для выполнения операторы, зависящие от этих результатов, т. е. операторы, аргументы которых вычислены. Управление выполнением операторов заключается в фиксации степени готовности каждого из них, направлении готовых операторов на исполнение и фиксации факта выполнения каждого оператора.

Для описания процесса выполнения программы предположим, что каждый результат (и аргумент) имеет уникальный идентификатор и что каждый оператор вырабатывает один результат. С каждым результатом связан список операторов, имеющих этот результат одним из аргументов. С каждым оператором связан список его аргументов. Таким образом, имеется массив списков операторов с числом элементов, равным числу идентификаторов, и массив списков аргументов с числом элементов, равным числу операторов.

При появлении в потоке данных очередного результата управление извлекает из масоива список операторов, имеющих этот результат одним из аргументов; в списке аргументов каждого такого оператора в соответствующем элементе делается пометка о его готовности. При всех готовых аргументах оператор считается готовым и помещается в список готовых для выполнения операторов. Условием корректного выполнения программы является следующее: один и тот же

оператор не может быть назначен для повторного выполнения, если он не окончит предыдущего выполнения.

Цель данного пособия — ознакомить с прикладными вопросами теории параллельного программирования, с основными приемами построения и анализа параллельных программ, с вопросами построения операционных систем для параллельного выполнения программ.

Глава 1. МОДЕЛИ ПРОГРАММ

§ 1. Основные понятия

Определение программы

Алгоритм — точное предписание, определяющее процесс преобразования исходных данных в искомый результат. Это предписание задается с помощью конечной системы правил.

Программа — преобразованная форма алгоритма, записанного либо на языке машины (машинная программа), либо на каком-либо алгоритмическом языке в виде некоторой последовательности правил этого языка (программа на данном алгоритмическом языке). Машинная программа состоит из совокупности объектов, называемых *командами*. Каждая команда заставляет машину выполнить указанное в ней действие. Действие задается одной частью команды, называемой *кодом операции*. Другая часть команды, называемая *адресной частью*, указывает адреса ячеек памяти, используемых в качестве аргументов и результатов команды.

Программа на алгоритмическом языке состоит из совокупности объектов, называемых *операторами*. Различают описательные и исполнительные операторы. Первые представляют собой список величин программы, вторые являются преобразователями информации. Величины программы не имеют, в общем, прямого отношения к ячейкам памяти. Исполнительные операторы выражают, как правило, сложные действия, такие, как вычисление выражений, функций, организация разветвлений, циклов и т. п.

Понятие графа

Пусть дано конечное множество $A = \{a_1, a_2, \ldots, a_n\}$.

Ориентированным графом $G = (A, \Gamma)$ называется набор из A и любого бинарного отношения Γ, заданного на прямом произведении $A \times A = \{(a_i, a_j)\}$. Напомним, что *прямым произведением* $M_1 \times M_2 \times \ldots \times M_n$ n непустых множеств $M_1 = \{m_1\}$, $M_2 = \{m_2\}$, $\ldots$, $M_n = \{m_n\}$ называется множество, элементами которого являются любые наборы вида $(m_1, m_2,$

$\ldots, m_n$). Любое подмножество B прямого произведения двух множеств $M = \{m\}$ и $N = \{n\}$ называется *бинарным отношением*, заданным на множестве $M \times N$. Элементы (m, n) множества B в этом случае называются парами, удовлетворяющими заданному бинарному отношению. Возвращаясь к определению графа, элементы множества A будем называть *вершинами* графа, а пары (a_i, a_j), удовлетворяющие бинарному отношению, — дугами графа. В дуге (a_i, a_j) вершина a_i называется *предшественником* своего *преемника* a_j.

Бинарное отношение Γ, обладающее тем свойством, что если $(a_i, a_j) \in \Gamma$, то и $(a_j, a_i) \in \Gamma$ и (a_i, a_i) не принадлежит Γ, называется *симметричным* и *антирефлексивным*. Любое такое бинарное отношение Γ задает неориентированный граф. Элементы множества A по-прежнему называются вершинами графа, а каждая пара (a_i, a_j) и (a_j, a_i) из Γ вместе называется *ребром*, соединяющим смежные вершины a_i и a_j. Геометрическое представление ориентированного и неориентированного графов на плоскости отображено на рис. 2, *а, б*. Вершины изображены кружками, дуги — стрелками, ребра — линиями.

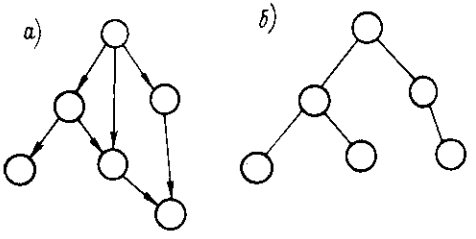


Рис. 2

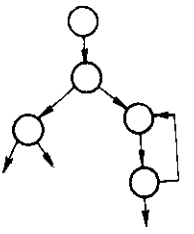


Рис. 3

Операторная схема программы

Основу операторной схемы составляет *граф переходов*, вершинами которого являются операторы программы, а дуги означают возможные передачи управления от одного оператора другому (рис. 3). Вершины делятся на два типа: *преобразователи*, из которых выходит одна дуга и которые преобразуют информацию из памяти, и *распознаватели*, из которых исходят две или более дуги и которые являются предикатами, меняющими ход вычислений. Выделяются частные случаи операторных схем: схемы без распознавателей — линейные и схемы без циклов, не содержащие контуров в графе переходов.

Каждый оператор имеет некоторое количество *аргументов* и *результатов*. Каждый аргумент и результат поименованы, т. е. для них произведено распределение памяти. Кроме связи по управлению операторы связаны информационными связями, т. е. связями между результатами одних операторов и

аргументами других. Информационные связи в операторной схеме группируются в *связки*. В одну связку входят все информационные связи, выходящие из некоторого результата. При распределении памяти связке присваивается уникальное имя, т. е. имя результата и имена аргументов должны совпадать.

Пример операторной схемы представлен на рис. 4. Информационные связи изображены пунктирными стрелками, операторы изображены прямоугольниками. Их обозначения ставятся внутри. Сплошные стрелки обозначают управляющие связи. Малые кружки изображают аргументы и результаты. Имена связок обозначены малыми латинскими буквами.

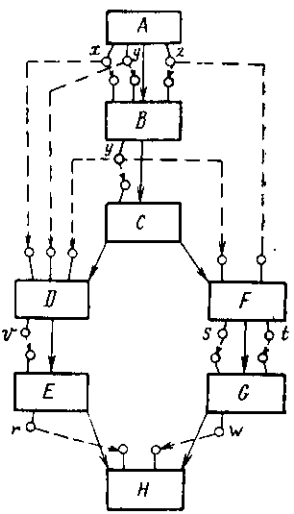


Рис. 4

Для существования информационной связи необходимо наличие хотя бы одного реализующего ее *маршрута*, т. е. цепочки операторов, которую можно проложить в графе переходов от оператора с результатом a к оператору с аргументом a, внутри которой нет ни одного оператора, вырабатывающего a. Распознаватели вырабатывают управляющую информацию, которая не хранится в памяти и не воспринимается на входах операторов, но используется устройством управления ЭВМ для выбора тех операторов, которые будут участвовать в вычислениях. Тогда будем говорить, что между выбираемым оператором и выбирающим распознавателем имеется управляющая связь или что распознаватель является *существенным* для данного оператора. Более строго: дугу (f_k, f_l) и оператор f_k назовем существенными для оператора f_i в операторной схеме, если любой путь в графе переходов схемы из оператора f_l в выходной оператор проходит через вершину f_i и если существует путь в графе переходов схемы из оператора f_k в выходную вершину, минующий f_i. Например, в операторной схеме рис. 4 оператор C и дуга (C, D) являются существенными для операторов D и E, оператор C и дуга (C, F) — для операторов F и G. Назовем дугу (f_k, f_l) *дополнительной* для оператора f_j относительно оператора f_i, если любой путь $(f_1, \ldots, f_i)$ в графе переходов схемы не содержит f_j, но существует путь $(f_k, \ldots, f_i)$, содержащий f_j. Например,

в операторной схеме рис. 4 дуга (C, D) является дополнительной для оператора F (и G) относительно оператора H.

Дадим формальное определение операторной схемы.

Пусть $F = \{f_1, f_2, \ldots, f_n\}$ — конечное множество операторов. Аргументы и результаты объединим общим названием — полюсы. Пусть $A = \{a_1, a_2, \ldots, a_p\}$ — множество аргументов, $R = \{r_1, r_2, \ldots, r_q\}$ — множество результатов, а $P = A \cup R = \{\pi_1, \pi_2, \ldots, \pi_r\}$ — множество полюсов $(r = p + q)$.

Распределение полюсов между операторами записывается в виде отображения V множества полюсов во множество операторов

$$V\colon P \to F.$$

Граф переходов определяется как ориентированный граф $C = (F, J)$, где J — бинарное отношение, задающее для операторов их непосредственных преемников по передаче управления.

Из введенных множеств можно построить скелет

$$S = (F, C, A, R, V).$$

Для превращения скелета в схему необходимы память $X = \{x_1, \ldots, x_m\}$ и распределение памяти между полюсами, которое задается в виде отображения множества полюсов на множество величин

$$L\colon P \to X.$$

Таким образом, операторная схема $G = (S, X, L)$. Необходимо ввести понятие *компоненты связности* графа G. Две вершины в графе a и b называются связанными, если существует последовательность вершин графа $v_1, v_2, \ldots, v_k$, такая, что $v_1 = a$, $v_n = b$, и любая пара (v_i, v_{i+1}) $(i = \overline{1, n-1})$ образована соседними вершинами. Граф называется связным, если в нем любая пара вершин является связанной. Существует теорема, по которой любой непустой граф G можно однозначно представить в виде объединения непустых попарно непересекающихся связных подграфов $G_1, G_2, \ldots, G_l$ $(l \geqslant 1)$. Эти графы и называются *компонентами* связности графа G. В частном случае в графе G может быть лишь одна компонента связности $G_1 = G$.

Компонентой сильной связности ориентированного графа G называется подграф этого графа, образованный множеством вершин V, таким, что v и v' принадлежат V тогда и только тогда, когда существует путь в этом графе из v в v' и существует путь из v' в v.

Рассмотрим ориентированный граф, вершинами которого являются полюсы P, а множество дуг задается бинарным отношением M существования маршрута, реализующего пере-

дачу информации между результатом и аргументом. Этот граф естественно назвать *информационным графом* $I = (P, M)$. Дуги информационного графа — особые. Результат в этих дугах бывает только предшественником, а аргумент — только преемником. По существу, бинарное отношение M задано не на прямом произведении $P \times P$, а на прямом произведении $R \times P$, где $R \subseteq P$, $A \subseteq P$, $R \cap A = \varnothing$. Графы такого рода называются двудольными, т. е. содержащими два типа перемежающихся вершин, один из которых связан с оператором, вырабатывающим результат, а другой — с аргументом (не оператором).

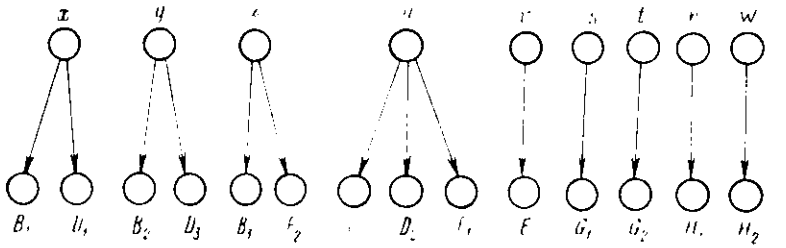


Рис. 5

На рис. 5 представлен информационный граф операторной схемы рис. 4. Компонентами связности информационного графа являются связки информационных связей. Если формально отказаться от двудольности информационного графа, приписав аргумент его оператору, то операторная схема может быть представлена в более компактной форме, в виде информационно - логического графа $G = (F, M, J)$, где F — по-прежнему множество операторов, а M и J — множество дуг, отображающих соответственно информационные и управляющие связи между операторами. Граф G называется биграфом, так как он представляет собой совокупность двух графов: информационного графа $G_1 = I = (F, M)$ и управляющего графа $G_2 = C = (F, J)$.

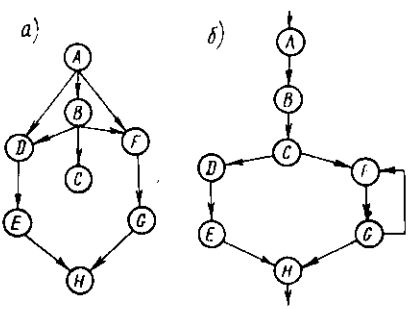


Рис. 6

В графе G_1 существует направленная дуга $(f_i, f_j) \in M$, если между операторами f_i и f_j существует информационная связь, т. е. если результат оператора f_i является аргументом оператора f_j. Будем называть f_i предшественником оператора f_j, а оператор f_j — преемником оператора f_i.

Операторы называются независимыми, если в информационном графе не существует дуги, соединяющей эти операторы.

В графе G_2 существует направленная дуга $(f_p, f_q) \in J$, если имеется управляющая связь, означающая безусловный переход или условный переход от оператора f_p к оператору f_q. Графы G_1 и G_2 схемы, изображенной на рис. 4, приведены на рис. 6, *а*, *б*.

Укрупнение операторов операторной схемы

Если рассмотреть реальные программы, то в них всегда можно заметить такие участки, где вычисления идут подряд, не прерываясь передачами управления. Такие участки называют *линейными участками* программы (операторной схемы, графа переходов). Для операторов линейного участка имеет место один и тот же набор условий перехода, а между операторами существуют только безусловные переходы. Две соседние вершины f_i и f_{i+1} графа G_2 относятся к различным линейным участкам, если: а) отсутствует дуга $(f_i, f_{i+1}) \in J$ или дуга $(f_{i+1}, f_i) \in J$; б) из вершины f_i выходит более, чем одна дуга, т. е. оператор f_i — оператор условного перехода; в) в вершину f_{i+1} входит более, чем одна направленная дуга. На рис. 7 приведено разбиение схемы рис. 3 на линейные участки. Операторную схему можно подвергнуть редукции, состоящей в замене каждого линейного участка одним оператором, где аргументом и результатами являются аргументы и результаты линейного участка, для которых существуют маршруты информационных пар вида (r_i, a) и (r, a_j). Здесь a и r — аргумент и результат, принадлежащие линейному участку, а r_i и a_j — не принадлежащие ему. Будем называть управляющие связи, соответствующие условным и безусловным переходам на границах линейных участков, существенными. Множество соответствующих им дуг $J' \subseteq J$ и множество линейных участков L образуют граф G_3, вершинами которого являются линейные участки, а дугами — существенные управляющие связи. На рис. 8 показан пример редукции исходной операторной схемы в укрупненные операторы.

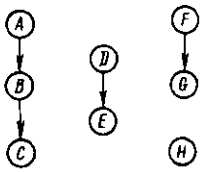


Рис. 7

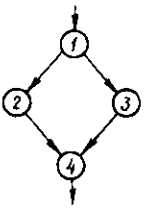


Рис. 8

Непосредственным обобщением этого подхода является факторизация операторной схемы на так называемые *гамаки*. Гамаком называется такой подграф g графа G_2, для которого

существуют две принадлежащие ему вершины, a (вход) и b (выход), обладающие следующими свойствами:

— все дуги из a ведут в гамак;

— любой путь, ведущий в гамак извне, проходит через a;

— все дуги в b ведут из гамака;

— любой путь, идущий из гамака вовне, проходит через b.

Гамак интересен тем, что его можно стянуть в одну вершину, не нарушая отношений смежности между другими вершинами графа.

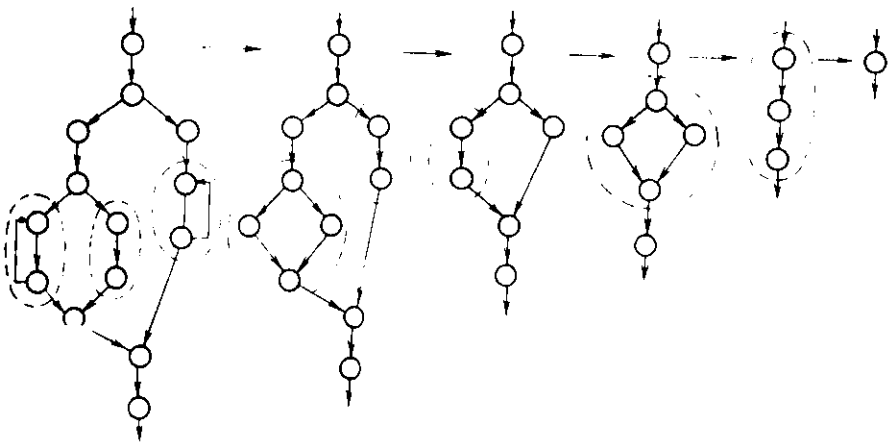


Рис. 9

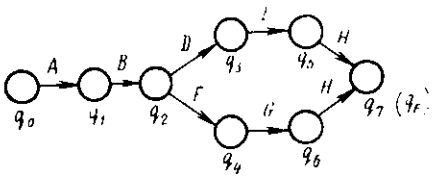
Выделение гамаков и линейных компонент позволяет разложить (факторизовать) исходную операторную схему на более простые части, которые легко анализируются. При этом сначала в схеме выделятся линейные компоненты, которые после анализа стягиваются в один оператор. После этого в схеме выделяются минимальные гамаки, вершинами которых являются «стянутые» линейные участки — операторы. Каждый гамак анализируется отдельно, после чего гамаки первого ранга редуцируются в вершины — элементарные операторы по отношению к остальной схеме. В полученной схеме снова выделяются линейные компоненты, и описанный двухчастный этап повторяется снова. Этот процесс повторяется до тех пор, пока после очередной редукции схема не превратится в одну или несколько изолированных вершин.

Рис. 9 содержит пример факторизации операторной схемы.

Граф состояний. Вычислительный процесс

Понятие процесса выполнения программы связывается обычно с некоторой информационной базой. Пусть $X = \{x_1, x_2, \ldots, x_N\}$ — набор переменных, характеризующих со-

стояние, т. е. некоторая память. Состояние описывается заданием значений всех элементов, входящих в набор. *Пространство состояний* — это множество значений, которое может принимать этот набор переменных. В предельном случае это множество равно прямому произведению множеств значений переменных $x_1, x_2, \ldots, x_N$. *Действие* — это присваивание значений некоторым переменным данного набора. Любое действие порождает новое состояние, т. е. новое содержимое памяти. Действие описывается с помощью *функции действия*, которая применяется к очередному состоянию для получения нового состояния. Функция действия отождествляется в на-



	x	y	z	u	v	s	t	r	w
q_0	0	0	0	0	0	0	0	0	0
q_1	1	1	1	0	0	0	0	0	0
q_2	1	1	1	1	0	0	0	0	0
q_3	1	1	1	1	1	0	0	0	0
q_4	1	1	1	1	0	1	1	0	0
q_5	1	1	1	1	1	0	0	1	0
q_6	1	1	1	1	0	1	1	0	1
q_7									

Рис. 10

шем контексте с программой. Для протекания процесса необходимо задать начальное состояние набору переменных. Итак, процесс P — это тройка

$$P = (X, \Phi, q_0),$$

где X — память, Φ — программа, q_0 — вектор начальных значений X (начальное состояние).

Графом состояний процесса назовем граф $G(Q, \Phi)$, вершинами которого $q_i \in Q$ будут состояния процесса (т. е. конкретные значения элементов x_j; $j = \overline{1, N}$) и дугами $\varphi_i \in \Phi$ — операторы. Дуга $\varphi_i \in \Phi$ исходит из состояния $q_k \in Q$ и приходит в состояние $q_l \in Q$, если в состоянии q_k может быть применен оператор φ_i, вследствие чего процесс переходит в состояние q_l. На рис. 10 изображен граф состояний, соответствующий схеме рис. 4, а в таблице даны отметки присвоения значений компонентам x_j для каждого состояния q_i. Состояние q_7 может быть произвольным; например, если оператор H — печать, то все x_j могут быть обнулены. Вычислительным процессом назовем путь на графе состояний, начинающийся состоянием q_0 и кончающийся q_F, где q_F — конечное (финальное) состояние. Для последовательных программ в качестве вычислительных процессов чаще всего рассматривают *цепочки* — последовательности операторов.

На рис. 9 граф состояний определяет два вычислительных процесса: $q_0, q_1, q_2, q_3, q_5, q_7$ и $q_0, q_1, q_2, q_4, q_6, q_7$.

Параллелизм и асинхронность. Эквивалентность и однозначность

Последовательный вычислительный процесс всегда детерминирован, т. е. последовательность выполнения операторов последовательной программы для одних и тех же начальных данных остается постоянной. При исполнении же параллельной программы для одних и тех же начальных данных могут получаться различные вычислительные процессы, отличающиеся порядком выполнения операторов или числом операторов, выполняемых параллельно. Такие вычислительные процессы называются недетерминированными.

Важным понятием, связанным с недетерминированными процессами, является понятие *эквивалентности*. Существует несколько определений эквивалентности: функциональная, по всем результатам, по истории ячеек, по информационному графу, по информационно-логическому графу. Мы не будем интерпретировать их все, а дадим определение функциональной эквивалентности. Два процесса P и P', заданных над одной и той же памятью, эквивалентны, если при всяком начальном состоянии этой памяти q_0 заключительные значения q_F и q_F' для обоих процессов равны.

Это определение требует, чтобы процессы протекали в одной и той же памяти. Это сильное ограничение можно снять, разрешив определить функциональную эквивалентность по выделенным элементам (т. е. лишь некоторые — «выделенные» — переменные состояния двух процессов должны иметь одинаковые заключительные значения, а для других это неважно, так что в данном случае $q_F \neq q_F'$).

Если вычислительные процессы, генерируемые программой при конкретных начальных данных, не эквивалентны, то программа называется неоднозначной при этих начальных данных, в противном случае — однозначной. Программа, однозначная при любых начальных данных, называется просто однозначной. В связи с недетерминированностью вычислительных процессов, генерируемых программой, возникают новые понятия, не свойственные последовательному программированию. Это — параллелизм и асинхронность программы.

Под *параллелизмом* программы понимают свойство программы допускать одновременное выполнение некоторого числа операторов.

Под *асинхронностью* программы понимают свойство программы допускать различные эквивалентные вычислительные процессы.

Если ввести соответствующую меру параллелизма, то эквивалентные программы можно сравнивать между собой по степени параллелизма. В качестве меры могут быть исполь-

зованы: максимально возможное число одновременно выполняемых операторов, среднее значение числа одновременно выполняемых операторов (полученное усреднением по времени выполнения программы) и т. п.

Аналогично благодаря введению меры асинхронности в виде числа разнообразных вычислительных процессов, допускаемых программой для любых начальных данных, можно сравнивать эквивалентные между собой программы по асинхронности.

Асинхронность программы — свойство более широкое, чем параллелизм. Например, из того факта, что программа P_1 не менее асинхронна, чем P_2, следует, что P_1 не менее параллельна, чем P_2 (но не наоборот).

Другой иллюстрацией важности понятия асинхронности является адаптация программы к наличным ресурсам вычислительной системы. Чем более асинхронна программа, тем более гибко и полно она может использовать ресурсы ВС.

§ 2. Формальные модели программ

В теоретическом программировании используется обычный методологический прием теоретических разделов прикладных наук — замена сложных реальных объектов и явлений более простыми абстрактными моделями. Рассмотрим три модели, получившие широкое распространение во многих работах по последовательному и параллельному программированию.

Схемы программ

Схема, так же как и программа, представляет собой текст на некотором формальном языке. Отличие от программы состоит в следующем. В программе арифметические, логические и другие выражения образуются с помощью символов операций и указателей функций, семантика (смысл) которых фиксирована языком программирования (базовые операции, базовые функции) или выводима по определенным правилам композиции из семантики базовых операций. Поэтому каждый оператор программы однозначным образом задает некоторую функцию над данными, а из этих функций образуется функция, задаваемая программой в целом. В схемах индивидуальные операции и функции заменены абстрактными символами переменных-операций и переменных-предикатов, т. е. функциональными и предикатными символами. Если вместо каждого такого символа подставить конкретную функцию или предикат, то схема превращается в программу. Такая подстановка называется *интерпретацией* схемы.

В отличие от программы схема не задает алгоритма, с ее помощью ничего нельзя вычислить, но она описывает строе-

ние множества программ, получающихся из схемы в результате задания множества интерпретаций. Ниже изображена программа (вычисление факториала) «а» и ее схема «б», написанная на языке АЛГОЛ-60

а) **begin integer** n, m; *ввод* (n);
$m := 1$; P: **if** $n = 0$ **then go to** L;
$m := n \times m$; $n := n - 1$;
go to P; L: *вывод* (m)
end

б) **begin** *ввод* (n);
$m := a$; P: **if** q (n) **then go to** L;
$m := f(m, n)$; $n := g$ (n);
go to P; L: *вывод* (m)
end

Из текста видно, что схема программы представляет собой некоторый формальный объект с незаданной семантикой. Если про программу «а» мы можем сказать, что она вычисляет $n!$, то про схему «б» мы можем сказать, что она описывает структуру, *скелет* программ, подобных программе вычисления факториала. Схемное представление программ становится особенно наглядным, если использовать изображение схемы в виде графа переходов. На рис. 11 представлен граф переходов для приведенной выше схемы. Анализ схемы позволяет сделать заключения, относящиеся ко всем программам, полученным из этой схемы с помощью всех возможных интерпретаций. Между программами и схемами существуют глубокие связи, которые позволяют изучать свойства программ, анализируя схемы-объекты более простой природы. Например, говорят, что схема пуста, если программа, получаемая из нее с помощью любой интерпретации, зацикливается. Две схемы эквивалентны, если программы, полученные из них одинаковой интерпретацией, эквивалентны, т. е. задают одну и ту же функцию.

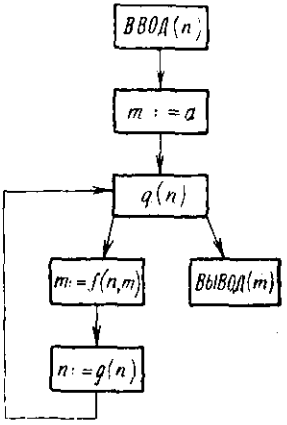


Рис. 11

Вычислительные модели

Вычислительная модель (ВМ) представляется в виде двудольного графа, который состоит из переменных $x_i \in X$ и связывающих их операторов $\varphi_i \in \Phi$. Часто такой граф называют *графом потока данных*. Для решения задачи на ВМ требуется

выполнить некоторое множество операторов $\Phi' \in \Phi$. Такая ВМ, соответствующая определенной задаче и содержащая только операторы Φ', является информационной структурой программы, при выполнении которой решается данная задача. Такая ВМ называется приведенной. Например, на рис. 12 переменные x_1, x_2, x_3 будут исходными, а искомые переменные x_6, x_7 получаются путем применения операторов φ_1, φ_2, φ_3 и φ_4. Ясно, что сама по себе информационная структура программы не является процессом. Для определения вычислительного процесса необходимо составить управляющую структуру на некотором формализованном языке. Для последовательных программ, как мы видели, составляется операторная схема программы (точнее — граф переходов). Для параллельных программ используются другие способы представления управления. Рассмотрим два из них.

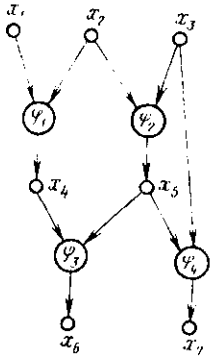


Рис. 12

Сети Петри

При помощи сетей Петри описывается поведение параллельной асинхронной системы. Сеть Петри — двудольный граф, множество вершин которого состоит из позиций $p_i \in P$ и переходов $t_j \in T$. Дуги соединяют позиции с переходами или переходы с позициями. Позиции обозначаются кружками, переходы — черточками, дуги — стрелками (рис. 13). В содержательном плане переходам соответствуют события (операторы), а позициям — условия. Дуги между позициями и переходами выражают связи двух типов: 1) позиция p_i — входная для данного перехода t_j. Для появления события, соответствующего переходу t_j, необходимо выполнение условия, соответствующего позиции p_i; 2) позиция p_i — выходная для данного перехода t_j. При появлении события, соответствующего переходу t_j, выполняется условие, соответствующее позиции p_i. Выполнение условий изображается на графе в виде маркировки — присваивания точек позициям (разметка). Каждая позиция может содержать несколько точек, количество которых показывает, сколько раз можно считать выполненным соответствующее условие. Некоторый переход за-

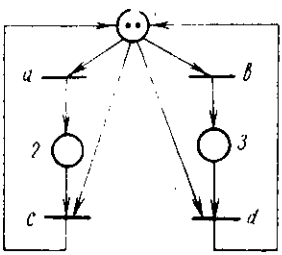


Рис. 13

пускается, а соответствующее событие появляется только тогда, когда все его входные позиции содержат не менее одной точки. При запуске перехода от каждой из его входных позиций отбирается по одной точке, а к каждой выходной — добавляется по одной точке.

Разметка M сети Петри — это функция, ставящая в соответствие позициям неотрицательные целые числа, равные числу точек в позициях. Если N — сеть Петри, P — множество позиций в N, а $n(P)$ — число позиций в P, то каждая позиция имеет номер из набора $\{1, 2, \ldots, n(P)\}$. Разметка M представляется в виде вектора, состоящего из $n(P)$ элементов, в которой i-й элемент означает число точек в i-й позиции. Например, для сети Петри, представленной на рис. 13, разметка $M = [2, 0, 0]$; $P = \{1, 2, 3\}$; $n(P) = 3$. Переходы a и b являются запускаемыми, а c и d — незапускаемыми. В результате запуска перехода a получается разметка $M' = [1, 1, 0]$. Теперь уже запускаемыми являются переходы a, b и c. Запустив c, мы получаем новую разметку $M'' = [2, 0, 0]$. В сетях Петри единственное ограничение на совместное выполнение переходов налагается числом наметок в позиции. То, что переходы являются запускаемыми, не означает, что они могут быть запущены совместно. Если бы разметка на рис. 13 была $M = [1, 0, 0]$, то оба перехода были бы запускаемыми, но не могли быть запущены одновременно.

Последовательность исполнения N — это последовательность разметок $M(0), M(1) \ldots M(i)$, где $M(i + 1)$ получается из $M(i)$ в результате запуска некоторого перехода. Для данной сети Петри с начальной разметкой существует много возможных последовательностей исполнения.

Живучей называется такая разметка, при которой каждый из переходов может быть запущен бесконечное число раз. Если достигнута такая разметка, что ни один из переходов не может быть запущен, говорят, что сеть Петри зависла (попала в тупик). На рис. 13 при начальной разметке $M(0) = [2, 0, 0]$, если запускаются оба перехода a и b, то достигается разметка $M(2) = [0, 1, 1]$. Сеть Петри мертва. Следовательно, разметка $M(0)$ не является живучей.

Размеченный граф — это сеть Петри, в которой с каждой позицией связано в точности по одной входящей и выходящей дуге. Размеченные графы менее общи, чем сети Петри, но легче анализируются. Для них доказан ряд теорем. В частности, было показано, что для сильно связанных размеченных графов мертвый переход существует только в том случае, если существует ориентированный цикл без точек.

Сеть Петри называется *диаграммой переходов*, если каждый переход имеет не более одного входа и одного выхода.

Для диаграмм переходов также получены доказательства ряда теорем.

Для описания взаимосвязей в сложных программах, каковыми являются, например, программы операционных систем, требуется полная описательная мощь общих сетей Петри. Но и общие сети Петри имеют ограничения. Например, оператор «не» нельзя смоделировать с помощью сетей Петри, поэтому используются другие, более общие способы задания управления.

Граф управления

Он определяет последовательность выполнения операторов. Каждый оператор связан с некоторым количеством *управляющих счетчиков*, каждый из которых содержит некоторое неотрицательное целое число. На рис. 14 представлен пример графа управления. Числа, записанные внутри квадратов, являются начальными значениями счетчиков. Если все счетчики, указывающие на оператор, имеют ненулевые значения, то оператор *определен*. Если оператор определен, то он может выполниться в любой момент, изменив значения некоторых переменных в соответствии с графом потока данных. При этом счетчики графа управления изменяются следующим образом: значения всех входных счетчиков оператора уменьшаются на единицу; значение каждого из выходных счетчиков увеличивается на некоторое неотрицательное число. Если для выходного счетчика это приращение постоянно, то соответствующее число записывается рядом с дугой. Для обеспечения возможности влияния данных на последовательность выполнения операторов допускается, что изменение выходного счетчика может задаваться с помощью функции от значений переменных x_i, на которые указывает оператор в графе потока данных. Граф потока данных (ВМ) и граф управления определяют так называемую вычислительную схему (рис. 12 и 14).

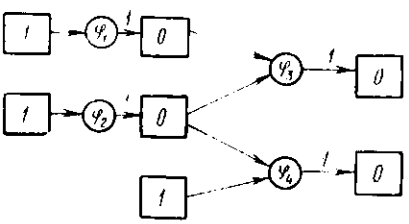


Рис. 14

Последовательностью исполнения схемы называется последовательность имен операторов $\varphi_1, \varphi_2, \ldots$, такая, что каждый оператор φ_i определен при тех значениях счетчиков, которые получаются в результате выполнения предшествующих операторов последовательности. Последовательность $\varphi_1, \varphi_2, \ldots, \varphi_N$ считается завершенной, если ни один оператор не определен после выполнения $\varphi_1, \varphi_2, \ldots, \varphi_N$.

Глава 2. РАСШИРЕНИЕ ЯЗЫКОВ ПОСЛЕДОВАТЕЛЬНОГО ПРОГРАММИРОВАНИЯ

§ 3. Средства распараллеливания в языке АЛГОПП

Перечислим основные требования к языкам с распараллеливанием:

1. Четкое выделение параллельно-выполняемых частей программ.
2. Слияние искомых параллельных ветвей в одну ветвь и обратная процедура — разветвление.
3. Группирование параллельных ветвей, связанных тем или иным способом между собой.
4. Обмен информацией между ветвями.
5. Синхронизация выполнения параллельных ветвей.

Покажем реализацию этих требований на примере средств языка АЛГОПП, являющимся расширением АЛГОЛа-60.

В АЛГОПП включены средства для статического и динамического распараллеливания.

Статическим является такое распараллеливание, когда число параллельно выполняемых частей определяется программистом до начала работы программы и программа выполняется так, что число процессоров соответствует числу параллельных ветвей программы.

При динамическом распараллеливании указания о возможности его осуществления делаются программистом и реализуются в ходе выполнения программы в зависимости от соблюдения тех или иных условий. В АЛГОППе хорошо выражен статический параллелизм, средства динамического распараллеливания довольно скудны.

Оператор разветвления имеет общий вид

do together < список меток >;

Этот оператор указывает начало параллельного участка. Метками помечены участки ветвей, выполняющихся параллельно. В свою очередь, каждая ветвь может ветвиться, т. е. допускаются гнездования произвольной глубины. Общие участки параллельных ветвей могут выноситься из них. Тогда их записывают между оператором разветвления и первой меткой из его списка. Эти общие участки также помечаются. В каждой ветви на них имеется ссылка в виде метки. Область действия данного оператора — до оператора соединения **join** включительно. Каждая параллельная ветвь должна быть логически законченной и оканчиваться оператором перехода к соответствующему оператору **join**. Выполнение параллельной ветви прерывается, если встречается оператор синхронизации, оператор обмена, оператор ожидания или оператор динамического распараллеливания.

Выполнение оператора **do together** заканчивается, если выполнены все параллельные ветви или если закончено выполнение всей программы.

Оператор соединения имеет вид

join <список меток>;

Список меток содержит метки параллельных ветвей, выполнение которых заканчивается данным оператором **join**. Множество меток в этом списке является подмножеством множества меток в списке меток оператора разветвления, в область действия которого входит данный оператор соединения. Оператор **join** должен быть помечен.

Оператор динамического распараллеливания (образование присоединяемого задания) имеет общий вид

task (<идентификатор задания>) **task**
(<идентификатор задания>,
<арифметическое выражение>);

Оператор **task** указывает, что некоторый оператор (или процедура) может быть вызван асинхронно и присоединен в качестве задания. Идентификатор — это метка оператора или название процедуры. Арифметическое выражение служит для вычисления приоритета присоединяемого задания по отношению к присоединяющему заданию или присоединяющей ветви.

По окончании задания результат его используется или запоминается, а память, занятая заданием, освобождается. Синхронизация заданий осуществляется с помощью оператора ожидания (см. ниже). Задания могут гнездоваться. Оператор **task** обычно используется для организации асинхронной работы устройств ввода-вывода. Если приоритет опущен, то он принимается равным нулю. Обмен данными между присоединяющим и присоединяемым заданиями напоминает вызов процедур в АЛГОЛе, т. е. присоединяющее задание засылает исходные данные в присоединяемое задание и получает от него результаты. Отличие состоит в том, что присоединенное задание выполняется параллельно (асинхронно) с присоединяющим, и возможны дополнительные способы обмена (см. ниже). Окончание задания происходит, если: а) в задании встречается оператор, заканчивающий задание; б) закончилось выполнение присоединяющего задания (или ветви); в) закончилось выполнение всей программы.

Операторы обмена информацией между параллельными ветвями

Оператор передачи имеет вид

transmit <совокупность фактических параметров>;

Этот оператор предназначен для передачи данных из одной ветви в одну или более других ветвей, выполняемых параллельно с данной. Передача допускается только в пределах области действия ближайшего оператора разветвления. Невозможна одновременная передача из двух или более ветвей. Оператору передачи может предшествовать оператор синхронизации для согласования во времени выполнения передающей и принимающей ветвей.

Оператор приема имеет вид

receive <совокупность фактических параметров>;

Оператор приема принимает данные, переданные какой-либо ветвью программы. Прием данных происходит во всех ветвях, в которых содержится оператор приема и которые охвачены тем же оператором разветвления, который содержит соответствующий оператор передачи. Выполнение ветви, содержащей оператор приема, не будет продолжено до тех пор, пока не будут получены все данные, указанные в совокупности фактических параметров. Выполнение ветвей, передающих и принимающих данные, может быть синхронизировано, прежде чем ветви передадут управление операторам передачи и приема.

Синхронизация параллельного выполнения

Оператор синхронизации имеет вид

synchronize <список меток>;

Оператор *синхронизации* согласовывает во времени выполнение некоторых операторов из различных параллельных ветвей. Список меток содержит метки операторов, выполнение которых нужно синхронизировать. Оператор синхронизации располагается непосредственно перед тем оператором, выполнение которого должно совпадать во времени с выполнением операторов, метки которых фигурируют в списке меток данного оператора синхронизации. Когда встречается оператор синхронизации, все параллельные ветви останавливаются на метках, указанных в списке меток данного оператора. Затем управление одновременно передается следующему оператору в данной ветви, а в других синхронизируемых ветвях — операторам с метками из списка.

Оператор ожидания имеет вид

expect <список предметов ожидания> | <оператор ожидания>, (<арифметическое выражение>);
<предмет ожидания> : : = <идентификатор задания> | | <именующее выражение> | <предмет ожидания> (<арифметическое выражение>);

Оператор ожидания приостанавливает выполнение программы, пока не будут удовлетворены некоторые условия.

Предметом ожидания являются: а) идентификатор задания, непосредственно присоединенного наименьшим блоком, в который входит оператор ожидания; б) именующее выражение, значение которого есть метка параллельной ветви, содержащейся в области действия ближайшего оператора разветвления. Арифметические выражения служат, с одной стороны, для работы со списками присоединенных заданий, имеющих одинаковые идентификаторы. В этом случае оно служит для вычисления индекса элемента списка. С другой стороны, если арифметическое выражение появляется в операторе ожидания, то оно преобразуется к целому m. Выполнение программы приостанавливается, пока не будут выполнены любые m из n условий, перечисленных в списке предметов ожидания. Когда оператор ожидания встречается с именующими выражениями, выполнение ветви, содержащей этот оператор ожидания, прекращается, пока не будет закончено выполнение ветвей, помеченных метками, являющимися значениями этих именующих выражений. Условие считается удовлетворенным, если задание или ветвь завершены.

§ 4. Пример записи параллельной программы

В качестве примера рассмотрим программу решения системы n линейных алгебраических уравнений методом простой итерации.

```
begin integer n; real q; input (n);
      begin array a[1 : n, 1 : n], b, x, xn[1 : n]; integer i, j, k;
            procedure УМН (y, yn, c, d, i); array y, yn, c, d;
            integer i;
            begin integer j; for j: = 1 step 1 until i — 1, i + 1
                  step 1
                  until n do y[i]: = d[i] + c[i, j] × yn[j]
            end УМН;
            procedure СРАВН (y, yn, g); value y, yn, g;
            array y, yn;
            real g;
            begin real max; integer k; max: = 0;
            for k: = 1 step 1 until n do if max < abs (y[i] —
            — yn[i])
            then max: = abs (y[i] — yn[i]); if max < g then
            go to конец;
            end СРАВН;
      input (a, b, g);
      do together l1, l2, . . . , ln;
```

```
        u1: begin xn[i]: = b[i]; b[i]: = b[i]/a[i, i];
            for j: = 1 step 1 until i — 1, i + 1 step 1 until n do
                a[i, j]: = a[i, j]/a[i, i];
        synhronize l1, l2, . . . , ln;
            for k: = 1 step 1 until n do if k = i then
            transmit (xn (k)) else receive (x[k]);
цикл: УМН (x, xn, a, b, i);
        synchronize l1, l2, . . . , ln;
                for k: = 1 step 1 until n do if k = i then
                transmit (x[k])
                else receive (x[k]);
        task (СРАВН (x, xn, a)); for k: = 1 step 1 until n do
            xn[i]: = x[i];
        go to цикл; end u1;
        l1: i: = 1; u1: ; l2: i: = 2; u1: ; . . . ln: i: = n; u1:;
конец: join l1, l2, . . . , ln;
output (x) end end
```

Отметим некоторые особенности исполнения программы, следующие из текста, и не оговоренные специально в языке.

1. Каждая ветвь копирует все, что необходимо, в свою память. Реентерабельность процедур не находит отражения в программе. Отсутствуют средства работы с общей памятью.

2. Вопрос выхода из ветвей в основную программу и возврат в ветвь проработан слабо, что вызывает необходимость в каждой ветви делать проверку на окончание всего процесса.

3. Характер выполнения операторов обмена и синхронизации скрыт от программиста. Из текста следует, что для обмена все ветви приостанавливаются, организуется некий общий канал, в который каждая ветвь посылает свой результат, и из которого принимает $(n-1)$ значений, выработанных другими ветвями. При этом полагается, что операторы в параллельных ветвях, перед которыми стоит оператор синхронизации, выполняются синхронно.

Из описания АЛГОППа следует, что структура программы представляет собой последовательность участков параллельности, состоящих из параллельных ветвей, которые, в свою очередь, могут включать участки параллельности. При выполнении программы управление движется по тексту программы от участка к участку, расщепляясь на каждом участке на такое количество независимых управлений, каково число параллельных ветвей в участке. Переход от одного участка к другому осуществляется после того, как все параллельные управления достигнут конца своих ветвей. Это обстоятельство резко уменьшает использование внутреннего параллелизма (асинхронности), присущего задаче.

Глава 3. РЕАЛИЗАЦИЯ КРУПНОБЛОЧНОГО РАСПАРАЛЛЕЛИВАНИЯ В ВМ

Программа, являясь записью алгоритма решения задачи на каком-либо языке программирования, представляет собой основное понятие статического описания в программировании. В ходе вычислений (реализации программы на ВМ) возникают динамические зависимости, для анализа которых необходимо другое базовое понятие — процесс. Развиваясь во времени и взаимодействуя между собой в рамках одной ВМ, процессы создают так называемый вычислительный процесс, управление ходом которого должно реализовываться с помощью специальных средств либо программистом, либо ВМ.

В данной главе будут рассмотрены вопросы распараллеливания программы на уровне блоков и основное внимание будет обращено на управление процессами, создаваемыми на этапе выполнения подобной параллельной программы.

Особенностью подобных процессов является то, что они логически зависимы, так как реализуют алгоритмы решения одной задачи, в отличие от независимых процессов, создаваемых для реализации разных задач.

§ 5. Блочное распараллеливание

Возможность организации параллельного выполнения различных ветвей программы в рамках одной ВМ возникла тогда, когда удалось организовать независимую работу нескольких исполнительных устройств.

Элементы параллелизма в выполнении программы появились уже в однопрограммном режиме работы ВМ при совмещении вычислений с вводом или выводом информации. Однако в полном объеме эти вопросы проявились тогда, когда в составе вычислителя появилось несколько процессоров, работающих независимо и функционально эквивалентных.

Подобный вычислитель — вычислительная система (ВС), имеющая в своем составе несколько центральных процессоров (ЦП), может одновременно выполнять и несколько команд (операторов) одной программы.

Разбиение программы на команды или те последовательности (блоки), которые могут в дальнейшем при реализации выполняться одновременно (параллельно), и есть одна из задач параллельного программирования. При ее решении стремятся провести такое разбиение программы, чтобы коэффициент использования устройств ВС был максимален.

Возможности потерь, простои устройств в ходе вычислений обусловлены тем, что устройства будут реализовывать логи-

чески зависимые процессы, а, значит, порядок их развития во времени определен общим алгоритмом решения задачи.

Ниже представлена программа вычисления значения аппроксимирующей функции Φ.

```
begin integer kmax, lmax;
      ввод (kmax, lmax);
      begin real p, b, x0, h; array R[0: lmax; 0: kmax],
            F[0: lmax], A[0: kmax];
            real procedure ПОЛ (k, m, x); integer k; array m;
            begin real s, y; integer n;
            s: = 0; y: = x; for n: = 0 step 1 until k do
            begin s: = s + m[k, n] × y; y: = y × x end;
            ПОЛ: = s
            end;
         ввод (b, x0, h);
      A: begin integer i; real z; z: = x0;
         for i: = 0 step 1 until lmax do
         begin F[i]: = sqrt (z × sin (z)); z: = z + h end;
         end A;
      B: begin integer i, j; real z;
         for i: = 0 step 1 until kmax do begin z: = x0;
         for j: = 0 step 1 until lmax do begin
         R[i, j]: = ПОЛ (i, b, z); z: = z + h end;
         end B;
      C: begin integer i, j;
         for i: = 0 step 1 until kmax do begin
         A[i]: = 0; for j: = 0 step 1 until lmax do
         A[i]: = A[i] + F[j] × R[i, j] end;
         end C;
      D: begin integer i;
         p: = 0; for i: = 0 step 1 until kmax do
         p: = p + ПОЛ (i, b, x) × A[i]
         end D
      end
end
```

Последовательная программа состоит из четырех блоков, реализующих следующие части алгоритма.

1. Блок A. Вычисление аппроксимируемой функции $F(x)$ во всех точках диапазона.

2. Блок *B*. Вычисление значений ортогональных полиномов в точках диапазона.

3. Блок *C*. Вычисление коэффициентов разложения.

4. Блок *D*. Вычисление значения аппроксимирующей функции.

Проведем анализ данной последовательной программы и рассмотрим возможные пути ее преобразования в параллельную.

Прежде всего необходимо отметить, что проведенное уже разбиение на блоки (структурирование) программы облегчит поставленную задачу. Нетрудно заметить, что блоки A и B независимы и могут выполняться параллельно без всяких дополнительных условий. Блоки же C и D — зависимые блоки. Так, блок C может быть запущен в том случае, когда вычислены очередные значения $F[j]$ и $R[i, j]$ (их подготовка производится в блоках A и B соответственно). В блоке D можно выполнять действие при готовности очередного коэффициента (блок C).

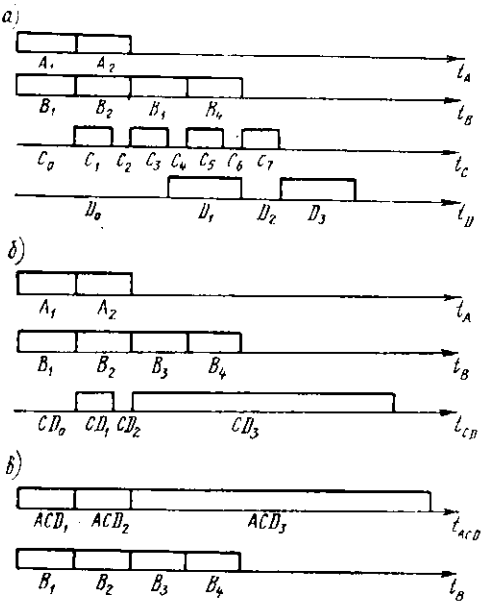


Рис. 15

Рассмотрим временные диаграммы выполнения блоков при условии, что каждый из них будет реализован на отдельном процессоре.

На рис. 15 приведены соответствующие диаграммы с указанием действий, выполняемых на отдельных участках. Рассмотрена задача при $lmax = 1$, $kmax = 1$ и следующих временах выполнения действия в блоках $A — D$:

A — 3 ед. времени на вычисление очередного элемента $F[i]$;
B — 3 „ „ $R[i, j]$;
C — 2 „ „ $A[i]$;
D — 4 „ „ P.

Приведенный рисунок четко определяет последовательность действий и их длительности. Так, независимость блоков A и B выражается в том, что подготовка очередного элемента в них начинается сразу же после выдачи предыдущего, независимо от того, на какой стадии выполнения находятся другие блоки программы. Противоположная картина наблюдается на диаграмме рис. 15, *а* (блок C, D). Участки C_0, C_2, C_4, C_6 — участки ожидания окончания действий в блоках A

и B, участки D_0, D_2 — участки ожидания окончания действий в блоке C.

Наличие участков ожидания и приводит к неполному использованию мощностей устройств ВС. Они тем больше, чем значительнее различаются длительности выполнения соответствующих фаз зависимых параллельных блоков. В нашем примере увеличение времени выполнения в блоке B на две единицы приведет к увеличению каждого участка ожидания на диаграмме рис. 15, *а* на две единицы.

Таким образом, при разбиении на блоки исходной последовательной программы необходимо стремиться к тому, чтобы $T_{ож} = \Sigma t_{i\ участка\ ожидания}$ было минимально. Следует отметить, что для независимых блоков оно равно 0, а для зависимых $\min T_{ож} = t_{0\ уч\ ож}$.

Структура ВС определяет количество параллельных устройств, которые могут быть использованы для реализации программы, и тем самым она также должна учитываться при разработке параллельной программы. Так, при наличии трех исполнительных устройств в рассматриваемом примере возможно объединение блоков C и D в один CD. Диаграммы выполнения, получаемые в этом случае, приведены на рис. 15, *б*. Незначительное увеличение суммарного времени выполнения программы (на 2 ед.) обусловлено тем, что сокращен простой устройств. Таким образом, $T_{реш}(3_{прц}) = T_{реш}(4_{прц}) + 2 = 20$ ед. времени (коэффициенты загрузки устройств сведены в общую таблицу). Видно, что имеется недоиспользование практически 50% мощностей. Исключение одного устройства (укрупнение блока) позволило немного повысить коэффициент использования.

Номер процессора	Вариант		
	I	II	III
1	3/9	3/10	11/11
2	6/9	6/10	6/11
3	4/9	8/10	—
4	4/9	—	—
Среднее	17/36	17/30	17/22

В данном примере ясно, что для повышения такого показателя, как коэффициент использования устройств BC, необходимо блок C реализовывать на устройстве с номером *1*. Тогда получаем средний коэффициент использования 17/22.

Однако решение вопроса о реализации конкретных блоков в более сложных ситуациях должно производиться в динамике.

Таким образом, эффективность распараллеливания программ определяется как разбиением ее на блоки, производимым при написании программы, так и решениями, принимаемыми при их реализации в системе.

В дальнейшем будем рассматривать реализацию вычислительного процесса для данной задачи при условии, что в

исходной последовательной программе выделено 4 блока (A, B, C, D), которые могут выполняться параллельно.

Запись параллельной программы

Реализовать выполнение программы на ВМ — это значит организовать в машине такую процедуру, в результате которой осуществляется изменение набора переменных состояния процесса, соответствующего данной программе. Изображающая точка, отражающая состояние процесса в каждый момент времени, перемещается в пространстве состояний от начальной точки пространства, задаваемой исходными данными, к конечной, которая соответствует получению результата решения исходной задачи.

Продвижение изображающей точки в пространстве состояний, отражающее развитие процесса, происходит в результате выполнения действий, описанных функцией действия процесса (программой вычислений, соответствующей данному процессу). Оно является результатом выполнения арифметических, логических действий в ЦП ВМ, действий по вводу-выводу или любых других, осуществляемых аппаратными или программными средствами ВМ (ВС).

Все аппаратные и программные средства, предназначенные для реализации действий, описанных в процессах, называются аппаратными (программными) ресурсами ВМ (ВС). Таким образом:

1. Все действия выполняются на ресурсах ВМ (ВС).
2. Выполнение любого действия производится в два этапа:
   — захват необходимого ресурса;
   — исполнение действия.

Возможностью захватить ресурс обладает единица независимой работы — процесс.

Поэтому каждый выделенный блок исходной параллельной программы должен быть оформлен на этапе реализации как процесс.

Описание процесса может быть задано явно, и тогда вводится специальный описатель *process* в текст программы. Возможно также разбиение на процессы с помощью указателя *parbegin* (<список>), где в списке указываются метки тех блоков программы, которые должны реализовываться параллельно. Второй способ предполагает, что оформление процессов на этапе выполнения осуществляется средствами ВС. В дальнейшем будет рассматриваться явное задание процессов.

Оформив каждый блок исходной распараллеленной программы в виде процесса, программист получает возможность осуществлять захват различных ресурсов в каждом из них.

Рассматривая временные диаграммы реализации блоков, мы убедились, что необходимо организовать определенную последовательность их выполнения. Это означает, что продвижение процессов в пространстве состояний возможно только при выполнении определенных условий. Например, вычисления блока C (процесса) могут начаться только по завершении подготовки очередных элементов в блоках (процессах) A и B (см. рис. 15, *а*). Подобные условия называются спусковыми функциями, они «спускают» процессы при выполнении условия готовности соответствующих переменных. В нашем примере данная функция представляет собой $f_3 = \Gamma_{F[i]} \wedge \Gamma_{R[i,\,j]}$, где f_3 — спусковая функция процесса C при вычислении $A[i]$; $\Gamma_{F[i]}$ — признак готовности $F[i]$ в процессе A, $\Gamma_{R[i,\,j]}$ — признак готовности $R[i,\ j]$ в процессе B. При $f_3 = 1$ процесс C может осуществить захват ресурсов BC и выполнить свои действия.

Подобные функции реализуют все f_j. Кроме этого, необходимо рассмотреть взаимодействие процессов с процедурой *ПОЛ*. К данной процедуре возможно обращение в один и тот же момент времени нескольких процессов. Подобный программный ресурс — общий ресурс — должен быть также разделен между процессами. Поэтому доступ к нему также необходимо реализовать в режиме разделения, т. е. управлять им спусковой функцией. Программы процессов с учетом этого факта приведены ниже:

```
begin integer kmax, lmax; ввод (kmax, lmax);
      begin real p, b, x0, h; array R[0: lmax, 0: kmax],
            F[0: lmax], A[0: kmax];
            real procedure ПОЛ (k, m, x); integer k; array m;
            begin real s, y; integer n;
            s: = 0; y: = x; for n: = 0 step 1 until k do
            begin s: = s + m[k, n] × y; y: = y × x end;
            ПОЛ: = s
            end;
            ввод (b, x0, h);
process A: begin integer i; real z; z: = x0;
            for i: = 0 step 1 until lmax do
            begin <спусковая функция f1>;
            F[i]: = sqrt (z × sin (z)); z = z + h end;
            end;
process B: begin integer i, j; real z;
            for i: = 0 step 1 until kmax do
            begin z: = x0; for j: = 0 step 1 until lmax do
           begin <спусковая функция f2>;
            R[i, j]: = ПОЛ (i, b, z); z: = z + h end; end;
            end;
```

```
process C: begin integer i, j;
           for i:=0 step 1 until kmax do begin a[i]:=0;
           for j:=0 step 1 until lmax do
           begin <спусковая функция f3>;
           A[i]:=A[i]+F[j]×R[i, j] end; end;
           end;
process D: begin integer i;
           p:=0; for i:=0 step 1 until kmax do
           begin <спусковая функция f4>;
           p:=p+ПОЛ (i, b, x)×A[i] end;
           end
      end
end
```

Реализация спусковых функций в процессе вычислений — это основная задача организации параллельных вычислений. При решении этого вопроса стремятся обеспечить максимальный параллелизм вычислений с наименьшими затратами на его реализацию.

§ 6. Реализация спусковых функций

Параллельные процессы, развивающиеся в ВС, вступают во взаимодействие друг с другом при использовании одного и того же ресурса. Поэтому каждый подобный общий ресурс защищается спусковой функцией.

В § 5 было рассмотрено взаимодействие процессов на информационном ресурсе — разделение общих переменных. Примером подобного взаимодействия является связь между процессами *A*, *B* и *C*. Там же рассмотрено и взаимодействие на общем программном ресурсе — разделение обращений к процедуре *ПОЛ*.

Доступ процесса к любому ресурсу осуществляется в результате выполнения следующих действий:

1) проверки возможности захвата ресурса. Если такая возможность имеется, то переход к п. 3, иначе — к п. 2;

2) помещения процесса в очередь ожидания данного ресурса;

3) фиксации захвата ресурса.

Первое действие предполагает, что ресурс имеет некоторый указатель состояния, который и позволяет в каждый момент времени идентифицировать его состояние. Третье действие есть не что иное, как модификация данного указателя. В случае невозможности захвата ресурса процесс должен быть приостановлен до момента выполнения условия захвата (спусковой функции). Поэтому он помещается в очередь ожи-

дания, которая просматривается каждый раз при изменении состояния ресурса.

Выполнение описанных действий — это и есть реализация спусковых функций в системе.

В данном пособии будут рассмотрены два основных аппарата организации взаимодействий между процессами: семафоры и мониторы. Сразу отметим, что оба они реализуют описанный доступ к ресурсу, но отличаются уровнем централизации управления.

Семафоры

Семафор — целая переменная, значение которой может быть изменено только в результате выполнения над ней двух операций — P и V. Указанные операции являются неделимыми.

Пусть S — семафор. При выполнении операции $P(S)$ значение уменьшается на единицу и далее, если: 1) $S \geqslant 0$, то процесс S продолжает работу; 2) $S < 0$, то процесс останавливается и встает в очередь ожидания, связанную с S. Он останется заблокированным до тех пор, пока операция $V(S)$, выполненная другим процессом, не освободит его. Когда процесс выполняет $V(S)$, значение S увеличивается на единицу и далее, если: 1) $S > 0$, то данный процесс продолжает работу; 2) $S \leqslant 0$, то один из процессов, находящихся в очереди ожидания, связанной с S, получает разрешение продолжить работу и удаляется из нее. Процесс, выполнивший операцию $V(S)$, также может продолжить работу.

Неделимость указанных операций означает, что в каждый момент времени только один из протекающих процессов может выполнять операцию P или V над данным семафором.

Именно поэтому в случае, когда $S = 1$ и два процесса одновременно попытаются выполнить операцию $P(S)$, только один из них сможет продолжить работу. Второй будет помещен в очередь ожидания. Перевод процессов из одного состояния в другое осуществляется с помощью специальных средств — ядра ОС. Так как в ходе выполнения операций P и V необходимо запускать и останавливать процессы, естественным является реализация данных операций в этом ядре.

Семафор S, максимальное значение которого равно единице, называется двоичным семафором. Семафор S с максимальным значением, равным k, — k-й семафор.

Таким образом, семафор — это указатель состояния ресурса, а операции над ним (P и V) обеспечивают как модификацию его значения, так и управление состояниями процессов.

При использовании данного подхода в организации параллельных вычислений необходимо:

— выделить общие ресурсы, разделение которых будет осуществляться процессами;

— сопоставить с каждым выделенным ресурсом указатель (семафор) и сделать его доступным для всех процессов;

— записать в программах процессов операции *P* и *V* так, чтобы их выполнение обеспечивало необходимый порядок взаимодействий процессов.

В рассматриваемом примере осуществляются взаимодействия между процессами *A*—*C*; *B*—*C*; *C*—*D*; *B*—*D* на ресурсах *F*, *R*, *A*, *ПОЛ*.

Сопоставим каждому ресурсу указатель состояния *УКАЗ* 1, *УКАЗ* 2, *УКАЗ* 3, *УКАЗ* 4.

Первые три взаимодействия осуществляются на информационном ресурсе и характеризуются тем, что один из процессов подготавливает информацию (его часто называют поставщиком), а второй использует ее (потребитель).

Значения указателя: 0 — информация не готова; *K* — готово *к* порций информации.

Операция *V* (*УКАЗ i*) должна осуществляться после окончания подготовки очередной порции информации (в процессах *A*, *B*, *C*).

Операция *P* (*УКАЗ i*) — перед использованием очередной порции информации (в процессах *C*, *D*).

Четвертое взаимодействие — разделение общего программного ресурса между процессами. Работа с данным ресурсом должна быть организована так, чтобы в каждый момент времени только один из процессов пользовался данным ресурсом. Поэтому проверка состояния ресурса и его изменение (так, чтобы закрыть доступ другому процессу) должны осуществляться до начала работы с ресурсом. Изменение состояния ресурса, открывающее доступ к нему других процессов, — после окончания работы с ним.

Значения указателя: 0 — ресурс занят процессом; 1 — ресурс свободен.

Операция *P* (*УКАЗ 4*) выполняется до начала действий на ресурсе, операция *V* (*УКАЗ 4*) — после их окончания.

Для того чтобы все указатели были доступны процессам, необходимо описать их в соответствующем внешнем блоке.

Программа, управление в которой осуществляется с помощью семафоров, приведена ниже.

```
begin integer kmax, lmax; семафор УКАЗ 1, УКАЗ 2, УКАЗ 3,
      УКАЗ 4; ввод (kmax, lmax);
      begin real p, b, x0, h; array R[0: lmax, 0: kmax],
                  F[0: lmax], A[kmax];
```

```
            real procedure ПОЛ (k, m, x); integer k; array m;
            begin real s, y; integer n;
            s: = 0; y: = x; for n: = 0 step 1 until k do
            begin s: = s + m[k, n] × y; y: = y × x end;
            ПОЛ: = s
            end;
        ввод (b, x0, h); УКАЗ 1: = 0; УКАЗ 2: = 0; УКАЗ 3: =
        = 0; УКАЗ 4: = 1;

process A: begin integer i; real z; z: = x0;
            for i: = 0 step 1 until lmax do
            begin F[i]: = sqrt (z × sin (z)); V (УКАЗ 1);
            z: = z + h end;
            end

process B: begin integer i, j; real z;
            for i: = 0 step 1 until kmax do
            begin z: = x0; for j: = 0 step 1 until lmax do
            begin P (УКАЗ 4); R[i, j]: = ПОЛ (i, b, z);
            V (УКАЗ 4); V (УКАЗ 2); z: = z + h end; end;
            end

process C: begin integer i, j;
            for i: = 0 step 1 until kmax do begin
            A[i]: = 0; for j: = 0 step 1 until lmax do
            begin P (УКАЗ 1); P (УКАЗ 2);
            A[i]: = A[i] + F[j] × R[i, j]; V (УКАЗ 3) end; end;
            end
process D: begin integer i;
            p: = 0; for i: = 0 step 1 until kmax do
            begin P (УКАЗ 3); P (УКАЗ 4),
            p: = p + ПОЛ (i, b, x) × A[i]; V (УКАЗ 4) end;
            end
        end
    end
```

Мониторы

Наиболее мощным средством управления взаимодействием параллельных процессов являются мониторы.

Монитор представляет собой совокупность процедур, обеспечивающих взаимодействие процессов с общими ресурсами. Данные процедуры входят в состав системного математического обеспечения ВС и могут быть вызваны в любом процессе специальной командой обращения к монитору. Все действия по захвату общего ресурса при этом скрыты от пользователя.

Мониторы, являясь неделимыми процедурами, гарантируют выполнение требований при работе с ресурсами любых типов. Они выполняют все действия с требованиями процессов, а также осуществляют перевод процессов из одного состояния в другое при изменениях состояний ресурсов.

В каждый момент времени монитор обслуживает запрос только одного процесса. Если одновременно приходят запросы от нескольких процессов на один монитор, данный конфликт разрешается аппаратно через систему прерываний.

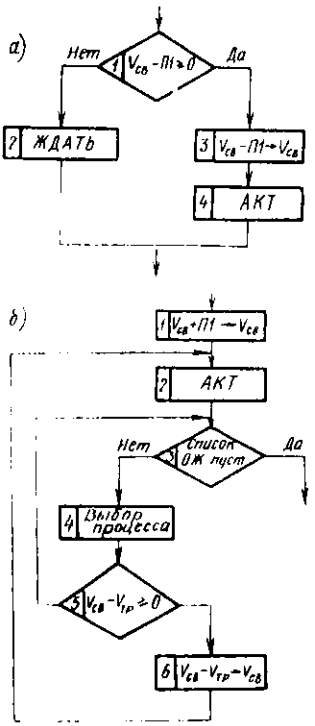


Рис. 16

ТР

№ ресурса	Информация		
	V	$V_{св}$	Ссылка
1			
2			
. . .	. . .	. . .	. . .
N			

Элемент СО

Идентификатор процесса	$V_т$	Ссылка

Элемент СГ

Идентификатор процесса	Ссылка

Рис. 17

Для захвата или освобождения ресурса пользователь помещает в процессе команды:

ЗАХВАТ (*П1*, *П2*); *ОСВОБ* (*П1*, *П2*),

где *П2* — уникальный идентификатор ресурса в системе; *П1* — требуемый или возвращаемый объем ресурса.

Функционирование монитора после получения им управления осуществляется в соответствии с алгоритмами, изображенными на рис. 16, *а*, *б*.

К информационным структурам монитора относятся (рис. 17):

— *ТР* — таблица ресурсов системы, каждый элемент которой содержит три поля — объем данного ресурса в системе *V*, свободный к текущему моменту объем ресурса $V_{св}$, ссылки на

список процессов, находящихся в состоянии ожидания, связанном с данным ресурсом;

— CO_i — список ожидания, связанный с i-м ресурсом. Каждый элемент списка имеет три поля: идентификатор процесса, требуемый объем ресурса, ссылка на следующий элемент списка;

— *СГ* — список готовых к выполнению процессов — процессов, для которых удовлетворены требования на ресурсы.

Работа монитора

В мониторе осуществляется проверка возможности действия над ресурсом, указанного в команде (блок *1*). Если $V_{св} - П1 < 0$, действие невозможно, и процесс помещается в список ожидания, связанный с данным ресурсом (блок *2*).

При $V_{св} - П1 \geqslant 0$ действие возможно. В блоке *3* выполняются операции, изменяющие состояние (параметры) ресурса $V_{св} - П1 \rightarrow V_{св}$. В блоке *4* процесс подключается к *СГ*.

В случае, когда выполняется команда *ОСВОБ* (рис. 16, *б*), работа блока *2* эквивалентна работе блока *4* (рис. 16, *а*), а в блоках *3, 4, 5, 6* осуществляется выбор процесса из списка ожидания и удовлетворение его требования.

Данный блок последовательно выбирает все процессы, требования которых могут быть удовлетворены в рамках освобожденного объема ресурса. Алгоритм выбора в данном блоке может реализовывать приоритетные или бесприоритетные дисциплины. Мы рассмотрели общую схему работы монитора, обеспечивающего взаимодействие параллельных процессов на ресурсах. В некоторых реализациях возможно построение мониторов на базе других алгоритмов, но функционально все алгоритмы эквивалентны.

Рассмотрим работу монитора на примере выполнения команды *ЗАХВАТ* (*П1, П2*).

К началу реализации процессов все таблицы монитора должны быть подготовлены. В *ТР* на основании описания ресурсов заносится информация *V*, каждый ресурс в дальнейшем имеет уникальное имя, соответствующее его номеру в *ТР*. В поле *ссылка ТР* заносится адрес списка ожидания по каждому ресурсу. Остальные поля ТР обнуляются.

На данной информационной базе строится работа монитора:

Блок 1. П2 — указывает номер в *ТР*, по которому можно получить *V* и $V_{св}$.

Блок 2. В случае, когда процесс должен быть помещен в очередь ожидания, его требование (*П1*) и номер процесса помещаются в список ожидания ресурса *П2*.

Блок 3. При наличии возможности обслуживания процесса осуществляется модификация *ТР*.

Блок 4. В случае, когда процесс получает затребованный ресурс, он может быть продолжен. Для этого необходимо перевести его в активное состояние, что и реализуется в данном блоке.

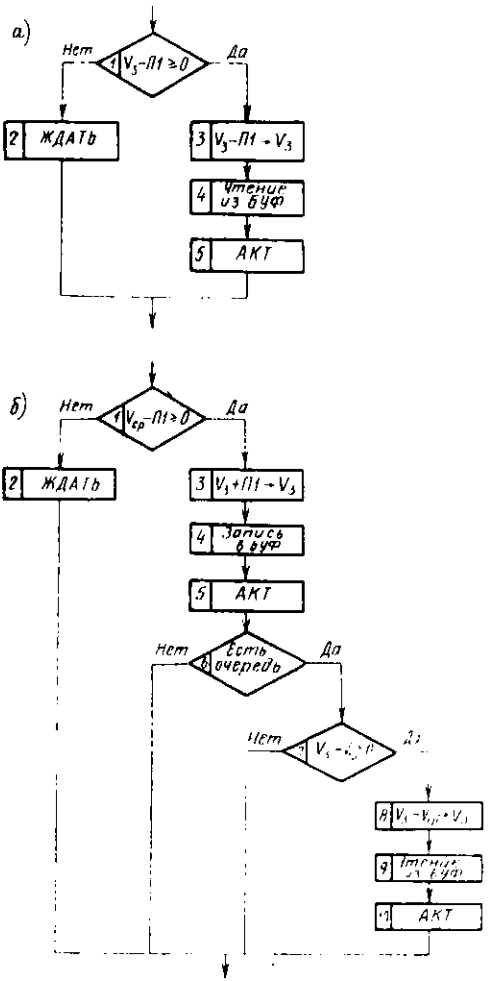


Рис. 18

Обмен информацией — разделение информационного ресурса — есть один тип взаимодействия между процессами в ВС.

Так как подобное взаимодействие есть не что иное, как разделение процессами общего поля памяти, оно может быть обеспечено монитором, в ведении которого находится буфер

обмена. На каждом буфере процессы вступают во взаимодействие по модели «поставщик — потребитель», а количество буферов определяется числом точек обмена информацией между процессами, протекающими в системе.

Назначение буферов точкам обмена осуществляется в каждом процессе явным указанием в команде обмена идентификатора буфера.

В поле описания внешнего блока указываются идентификаторы всех буферов в системе и соответствующие их объемы.

Команды обращения к монитору имеют вид

ПОСЛАТЬ (*П1*, *П2*, *П3*); *ПРИНЯТЬ* (*П1*, *П2*, *П3*),

где *П1* — идентификатор буфера, используемого для данного обмена; *П2* — идентификатор переменной для помещения информации, получаемой (посылаемой) из буфера; *П3* — объем получаемой информации.

Введение параметров *П3* позволяет осуществлять обмен не только одиночными переменными, но и массивами и любыми их частями.

Алгоритмы функционирования монитора при реализации команд приведены на рис. 18. Информационные поля соответствуют описанию, приведенному ранее.

В рассматриваемом примере необходимо реализовать взаимодействия *A*—*C*; *B*—*C*; *C*—*D*; *B*—*D*.

Первые три взаимодействия — обмен, для организации каждого из них выделяется ресурс буферов: *Буф* 1 (*A*—*C*), *Буф* 2 (*B*—*C*), *Буф* 3 (*C*—*D*) объемами V_1, V_2, V_3.

Четвертое взаимодействие — разделение программного ресурса, с которым в каждый момент времени может работать только один процесс. Идентифицируем данный ресурс как *РЕС* 1 с объемом $V = 1$.

Программа, управление в которой осуществлено с помощью мониторов, приведена ниже.

```
begin integer kmax, lmax; ресурсы БУФ 1 (V1), БУФ 2 (V2),
            БУФ 3 (V3), РЕС 1 (1); ввод (kmax, lmax);
      begin real p, b, x0, h; array R[0: lmax, 0: kmax],
                F[0: lmax], A[0: kmax];
            real procedure ПОЛ (k, m, x); integer k; array m;
            begin real s, y; integer n;
            s: = 0; y: = x; for n: = 0 step 1 until k do
            begin s: = s + m[k, n] × y; y: = y × x end;
            ПОЛ: = s
            end
            ввод (b, x0, h);
```

```
process A: begin integer i; real z; z: = x0;
           for i: = step 1 until lmax do
           begin F[i]: = sqrt (z × sin (z)); ПОСЛАТЬ (БУФ 1,
           F[i], 1) end;
           end
process B: begin integer i, j; real z;
           for i: = 0 step 1 until kmax do begin z: = x0;
           for j: = 0 step 1 until lmax do begin
           ЗАХВАТ (РЕС 1, 1); R[i, j]: = ПОЛ (i, b, z);
           ОСВОБ (РЕС 1, 1); ПОСЛАТЬ (БУФ 2, R[i, j], 1)
           z: = z + h end; end;
           end
process C: begin integer i, j;
           for i: = 0 step 1 until kmax do begin A[i]: = 0;
           for j: = 0 step 1 until lmax do begin
           ПРИНЯТЬ (БУФ 1, F[i], 1); ПРИНЯТЬ (БУФ 2,
           R[i, j], 1);
           A[i]: = A[i] + F[j] × R[i, j]; ПОСЛАТЬ (БУФ 3,
           A[i], 1)
           end; end;
           end
process D: begin integer i;
           p: = 0; for i: = 0 step 1 until kmax do
           begin ПРИНЯТЬ (БУФ 3, A[i], 1);
           ЗАХВАТ (РЕС 1, 1); p: = p + ПОЛ (i, b, x) × A[i];
           ОСВОБ (РЕС 1, 1) end;
           end
      end
  end
```

Глава 4. АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ

§ 7. Распараллеливание программ

Понятие асинхронной программы

Статические методы распараллеливания, примеры которого мы разобрали выше, характеризуются тем, что в них в явном виде указываются участки параллельности. Альтернативой такому подходу является динамическое априорное распараллеливание. В результате предварительного анализа программы не устанавливается параллельность операторов, а регистрируется ряд определенных отношений между элементами программы (операторами, входами, выходами операторов и т. п.). На основе этих отношений строятся параллельные программы, не содержащие явных средств указания параллельности операторов. Параллелизм (асинхронность) выпол-

нения неявно содержится в управляющих конструкциях, называемых *спусковыми функциями*. Итак, асинхронная параллельная программа (*А-программа*) представляет собой множество *блоков*, каждый из которых состоит из оператора над общей памятью, предиката (спусковой функции) над специальной управляющей памятью и управляющего оператора над этой же управляющей памятью.

Выполнение А-программы осуществляется следующим образом. В произвольные моменты времени проверяются значения спусковых функций всех невыполняемых блоков. После каждой проверки из блоков, значения спусковых функций которых равны «истине», выбирается произвольное подмножество блоков и инициируется выполнение операторов над общей памятью. Каждый из операторов выполняется произвольное время, после чего изменяет состояние общей памяти. После выполнения каждого оператора выполняется управляющий оператор, изменяющий состояние управляющей памяти.

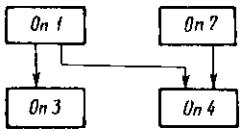


Рис. 19

Выполнение *А*-программы заканчивается, если при очередной проверке спусковых функций ни один из блоков не выполняется и значения всех спусковых функций — «ложь».

Рассмотрим для примера операторную схему, представленную на рис. 19.

Параллельное выполнение операторов *1—4* можно задать с помощью следующей *А*-программы:

$$\{\alpha = 0 \mid \text{оп. } 1;\ \alpha := 1\}$$

$$\{\beta = 0 \mid \text{оп. } 2;\ \beta := 1\}$$

$$\{\alpha = 1 \,\&\, \gamma = 0 \mid \text{оп. } 3;\ \gamma := 1\}$$

$$\{\alpha = 1 \,\&\, \beta = 1 \,\&\, \delta = 0 \mid \text{оп. } 4;\ \delta := 1\}$$

Здесь α, β, γ, δ — управляющие переменные, с помощью которых формируются спусковые функции. Спусковые функции в каждом блоке отделены от операторов вертикальной чертой. Для правильного составления спусковой функции очень важно уметь определять независимость операторов, являющуюся основой любого распараллеливания. Поэтому разберем основные способы и приемы распараллеливания последовательных программ, представленных операторными схемами.

Распараллеливание линейных программ

В этом случае имеем линейную операторную схему без разветвлений. Основным понятием, используемым при распа-

раллеливании, является понятие взаимно независимых операторов, определяемое через *информационный граф* цепочки (линейной схемы). Мы знаем, что информационный граф содержит информацию о выполняемых операциях и о зависимости между операторами по данным, но не содержит сведений о порядке выполнения операторов и распределении памяти по входам и выходам операторов.

На рис. 20, *а* показана линейная операторная схема, а на рис. 20, *б* — ее информационный граф.

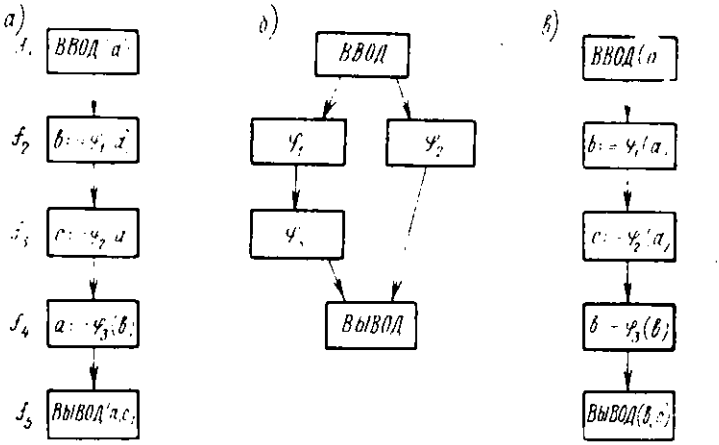


Рис. 20

Пара операторов называется *независимой*, если в информационном графе линейной схемы не существует пути, соединяющего эти операторы. Однако независимость операторов еще не означает, что их можно выполнить параллельно. Существуют еще *связи по памяти*, которые ограничивают параллельность и коммутативность выполнения независимых операторов. Так, в схеме на рис. 20, *а* операторы f_2 и f_3 независимы, и любое их коммутативное или параллельное выполнение сохраняет информационный граф неизменным. Однако независимые операторы f_3 и f_4 должны выполняться так, чтобы выполнение f_3 началось раньше, чем закончится выполнение f_4, иначе информационный граф при выполнении не сохранится.

Таким образом, для параллельности пары операторов можно сформулировать следующие достаточные условия:

а) операторы f и f' независимы;

б) $In f \cap Out f' = \varnothing \,\&\, Out f \cap In f' = \varnothing \,\&\, Out f \cap Out f' = \varnothing$, где $In f$ — входной, а $Out f$ — выходной наборы переменных оператора f. Условия «а» и «б» не равноценны. Линейную

операторную схему можно всегда преобразовать в эквивалентную линейную, в которой для любой пары операторов выполнение условия «а» имплицирует выполнение условия «б». Такую схему (или цепочку) будем называть *приведенной*. Приведенную схему можно получить из исходной перераспределением памяти между операторами. При этом исходная схема преобразуется в эквивалентную по информационному графу таким образом, чтобы для любой пары независимых операторов выполнялось условие «б». Например, в линейной схеме на рис. 20, *в* все ограничения на параллельное выполнение сняты, и независимые операторы f_3 и f_4 параллельны. Информационный граф этой линейной схемы тот же, что и для схемы рис. 20, *а* (см. рис. 20, *б*).

Распараллеливание программ без циклов

В случае операторных схем, содержащих распознаватели, между операторами возникают *управляющие* (логические) связи. Последовательная программа при ее выполнении порождает в общем случае различные *цепочки* при различных начальных данных. Поэтому однозначного ответа о параллельности двух операторов дать нельзя: в одной цепочке они могут быть параллельны, в другой — не могут. Отсюда напрашивается вывод—не ставить вопрос о параллельности отдельных пар операторов, а ввести некоторую глобальную характеристику внутренней асинхронности операторных схем. Для характеристики потенциальной асинхронности используются информационно-логические графы цепочек. Такой граф представляет собой информационный граф с добавленными логическими дугами, отмечающими наличие существенных управляющих связей между операторами. Совокупность информационно-логических графов всех цепочек операторной схемы полностью отражает те ограничения на параллельное выполнение операторов, которые возникают из-за информационных и логических связей.

От ограничений на параллельное выполнение, возникающих из-за распределения памяти, можно избавиться, преобразовав операторную схему в приведенную операторную схему, все цепочки которой являются приведенными. Это преобразование в общем случае требует изменения графа переходов, если мы хотим сохранить потенциальную асинхронность исходной схемы.

В качестве примера рассмотрим операторную схему на рис. 21, *а*. Операторную схему рис. 21, *а* нельзя преобразовать в приведенную, переименовав, например, выход оператора f_2 с b на d, так как полученная схема не будет эквивалентна исходной (будет потеряна информационная связь между f_2

и f_5). Один из выходов показан на рис. 21, *б*, где одновременно с переименованием выхода оператора f_2 введен оператор пересылки $b := d$. Другой выход — «расщепление» оператора f_5 — показан на рис. 21, *в*. Схемы рис. 21, *б* и *в* — приведенные. Приведенные операторные схемы обладают тем свойством, что они могут быть преобразованы в параллельные программы таким образом, чтобы преобразование не затрагивало операторов.

Приведем пример такого преобразования, которое называется *десеквенцией*. В результате десеквенции получаются параллельные программы, представленные в виде *А*-программ.

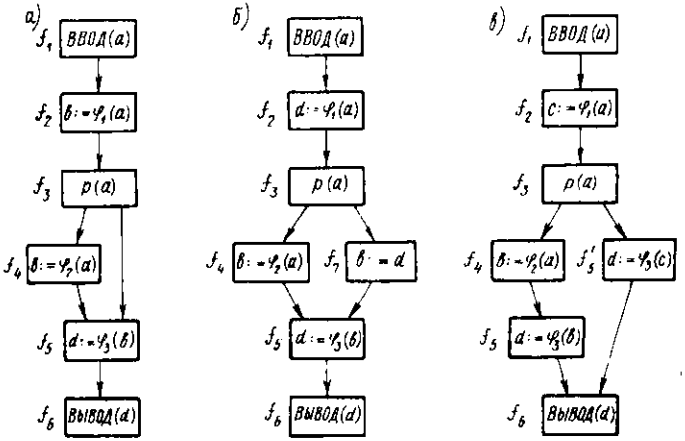


Рис. 21

Предполагается, что в операторной схеме из распознавателя может исходить любое конечное число дуг и все эти дуги пронумерованы от 0 до l. Распознаватель представляет собой, таким образом, многозначный предикат $P(m_1, \ldots, m_k)$ со значениями от 0 до l, где $m_1 \ldots m_k$ — элементы общей памяти. Пусть f_i — оператор заданной операторной схемы, а Inf_i — входной набор длины n оператора f_i. Обозначим через F_{ik} множество тех операторов схемы, которые присваивают некоторое значение переменной $m \in Inf_i$, используемое как аргумент k-го входа оператора f_i. Десеквенция приведенной операторной схемы без циклов осуществляется следующим образом:

1. Организуется управляющая память сопоставлением каждому оператору f_i операторной схемы элемента памяти α_i со значениями 0 и 1, а каждой дуге с номером l, исходящей из распознавателя f_i, — элемента памяти β_{il}, также со значе-

ниями 0 и 1. Элемент α_0 будет соответствовать фиктивному оператору, инициирующему программу.

2. Каждому оператору f_i схемы сопоставим управляющий оператор g_i. Для преобразователя f_i управляющий оператор будет иметь вид $\alpha_i := 1$. Для распознавателя f_i управляющий оператор g_i представляет собой последовательность из $(l+1)$ операторов, где $(l+1)$ — число дуг, исходящих из f_i, и имеет вид

$$\alpha_i := 1;\ \textbf{if}\ P(m_1, \ldots, m_n) = 0\ \textbf{then}\ \beta_{i0} := 1;$$

$$\ldots\ldots\ldots\ldots\ldots\ldots$$

$$\textbf{if}\ P(m_1, \ldots, m_n) = l\ \textbf{then}\ \beta_{il} := 1.$$

Пусть входной набор Inf_i оператора f_i имеет длину n. Оператору f_i сопоставляется спусковая функция y_i, представляющая собой конъюнкцию $n+1$ предиката

$$y_i = P_0{}^i \,\&\, P_1{}^i \,\&\, \ldots \,\&\, P_n{}^i.$$

Предикат $P_0{}^i$ имеет вид

$$\alpha_i < \sum_{(f_s,\, t) \in U} \beta_{st},$$

где U — множество всех дуг, существенных для оператора f_i в схеме; s — номер распознавателя, t — порядковый номер дуги оператора распознавателя. Предикат $P_k{}^i\ (1 \leqslant k \leqslant n)$ соответствует k-му входу оператора f_i и в свою очередь представляет собой конъюнкцию предикатов

$$P_k{}^i = P^i_{k1} \,\&\, P^i_{k2} \,\&\, \ldots \,\&\, P^i_{km},$$

где предикат $P^i_{kj}(1 \leqslant j \leqslant m)$ соответствует оператору f_j из множества F_{ik}, а m — число операторов в этом множестве. Предикат P^i_{kj} имеет вид

$$\alpha_i < \sum_{(f_s,\, t) \in U'} \beta_{st} + \alpha_j,$$

где U' — множество дуг, дополнительных для f_j относительно f_i. Начальные значения всех элементов управляющей памяти, кроме α_0, полагаются равными нулю, а значение α_0 равно единице. Получаемое таким образом множество блоков является однозначной A-программой, эквивалентной исходной операторной схеме. Данная A-программа является максимально асинхронной относительно исходной операторной схемы. Получим A-программу из операторной схемы, представленной на рис. 22. А-программа записана в табл. 1.

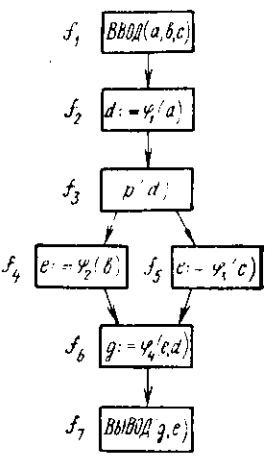


Рис. 22

Таблица 1

Блок	Спусковая функция	Оператор схемы	Управляющий оператор
f_1	$\alpha_1 < \alpha_0$	*Ввод* (a, b, c)	$\alpha_1 := 1$
f_2	$\alpha_2 < \alpha_1$	$d := \varphi_1(a)$	$\alpha_2 := 1$
f_3	$\alpha_3 < \alpha_2$	$p(d)$	$\alpha_3 := 1$; **if** $p(d) = 0$ **then** $\beta_{30} := 1$ **else** $\beta_{31} := 1$
f_4	$\alpha_4 < \alpha_1$ & $\alpha_4 < \beta_{30}$	$e := \varphi_2(b)$	$\alpha_4 := 1$
f_5	$\alpha_5 < \alpha_1$ & $\alpha_5 < \beta_{31}$	$e := \varphi_3(c)$	$\alpha_5 := 1$
f_6	$\alpha_6 < \alpha_2$ & $\alpha_6 < \beta_{30} + \alpha_5$ & $\alpha_6 < \beta_{31} + \alpha_4$	$g := \varphi_4(e, d)$	$\alpha_6 := 1$
f_7	$\alpha_7 < \alpha_6$ & $\alpha_7 < \beta_{30} + \alpha_5$ & $\alpha_7 < \beta_{31} + \alpha_4$	*Вывод* (g, e)	$\alpha_7 := 1$

Из таблицы следует:

А-программа $F = \{f_1, f_2, \ldots, f_7\}$;

управляющая память $M_c = \{\alpha_0, \alpha_1, \ldots, \alpha_7, \beta_{30}, \beta_{31}\}$;

начальное значение $M_c = \{\alpha_0 = 1, \alpha_1 = \alpha_2 = \ldots = \beta_{30} = \beta_{31} = 0\}$.

Анализ показывает, что A-программа допускает только два последовательных вычислительных процесса: f_1, f_2, f_3, f_4, f_6, f_7 и f_1, f_2, f_3, f_5, f_6, f_7.

§ 8. Распараллеливание программ, содержащих циклы

Наличие циклов в программах приводит к значительным трудностям при распараллеливании. На практике используется ряд стратегий:

1. Каждый цикл стягивается в один оператор, после чего получается схема без циклов.

2. Каждый цикл с известным числом повторений развертывается, после чего опять получается схема без циклов.

3. Анализ возможностей параллельного выполнения разных повторений цикла.

4. Анализ программ, при котором не выясняется параллельность пар операторов, а между элементами программы устанавливаются отношения, на основе которых и строятся параллельные программы. При этом не требуется явного выделения циклов, так как анализ проводится на множествах цепочек. Как мы увидим далее, теряется привычное представление о характере выполнения последовательного цикла.

Рассмотрим эту стратегию подробней. Операторная схема с циклами рассматривается как обобщение операторной схемы без циклов. Это обобщение состоит в том, что множество

цепочек схемы может быть бесконечным. Это приводит к различным трудностям при анализе схем и преобразовании их в параллельные программы. Так, например, для произвольной операторной схемы не существует конечной приведенной операторной схемы над общей памятью, эквивалентной исходной схеме, если потребовать, чтобы их потенциальные асинхронности были равны.

На рис. 23, *а* показана операторная схема, не являющаяся приведенной. Для нее можно построить эквивалентную приведенную операторную схему с той же потенциальной асинхронностью, введя операторы над массивами, как показано на рис. 23, *б*.

Опишем преобразование операторных схем с циклами в параллельные асинхронные программы (*A*-программы). При организации управляющей памяти для схем с циклами каждому оператору также сопоставляются элементы управляющей памяти. Спусковые

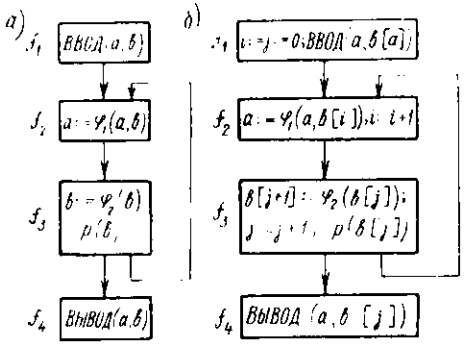


Рис. 23

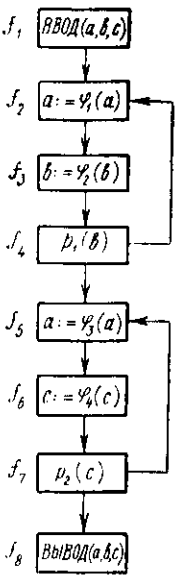


Рис. 24

функции имеют ту же структуру, что и в случае схем без циклов, но вид предикатов P_{kj} в спусковой функции y_i зависит от соотношения глубин операторов f_i и f_j. Предикат P_{kj} в спусковой функции оператора f_i может зависеть не только от элементов управляющей памяти, сопоставленных оператору f_i, дугам, существенным для f_i, и дугам, дополнительным для оператора f_j относительно f_i, но и от элементов управляющей памяти, сопоставленных компонентам, охватывающим операторы f_i и f_j (вложенные циклы). Рассмотрим простой случай операторной схемы с циклами, пример которой показан на рис. 24. В табл. 2 приведена *A*-программа, соответствующая этой операторной схеме. Символ $\dot{-}$ в спусковых функциях представляет знак операции усеченной разности:

$$a \dot{-} b = a - b, \text{ если } a > b;$$
$$a \dot{-} b = 0, \quad \text{если } a \leqslant b.$$

Таблица 2

Блок	Спусковая функция	Оператор программы	Управляющий оператор
f_1	$\alpha_1 < \alpha_0$	*Ввод* (a, b, c)	$\alpha_1 := 1$
f_2	$\alpha_2 < \alpha_1 + \beta_{11}$	$a := \varphi_1(a)$	$\alpha_2 := \alpha_2 + 1$
f_3	$\alpha_3 < \alpha_1 + \beta_{41}$	$b := \varphi_2(b)$	$\alpha_3 := \alpha_3 + 1$
f_4	$\alpha_4 < \alpha_3$ & $\alpha_4 < 1 + \beta_{41}$	$p_1(b)$	$\alpha_4 := \alpha_4 + 1$; **if** $p_1(b)=0$ **then** $\beta_{40} := 1$ **else** $\beta_{41} := \beta_{41} + 1$
f_5	$\alpha_5 < 1 + \beta_{71}$ & $\alpha_5 \div \beta_{71} < (\alpha_2 + \beta_{40}) \div (1 + \beta_{41})$	$a := \varphi_3(a)$	$\alpha_5 := \alpha_5 + 1$
f_6	$\alpha_6 < \alpha_1 + \beta_{71}$	$c := \varphi_4(c)$	$\alpha_6 := \alpha_6 + 1$
f_7	$\alpha_7 < \alpha_6$ & $\alpha_7 < 1 + \beta_{71}$	$p_2(c)$	$\alpha_7 := \alpha_7 + 1$; **if** $p_2(c)=0$ **then** $\beta_{70} := 1$ **else** $\beta_{71} := \beta_{71} + 1$
f_8	$\alpha_8 < (\alpha_5 + \beta_{70}) \div (1 + \beta_{71})$ & $\alpha_8 < (\alpha_3 + \beta_{40}) \div (1 + \beta_{41})$ & $\alpha_8 < (\alpha_6 + \beta_{70}) \div (1 + \beta_{71})$	*Вывод* (a, b, c)	$\alpha_8 := 1$

Поскольку в схемах с циклами необходимо следить за числом выполнений каждого оператора, управляющие операторы, входящие в цикл, имеют вид

$$\alpha_i := \alpha_i + 1; \quad \beta_{il} := \beta_{il} + 1.$$

Рассмотрим, например, спусковую функцию блока f_5. Это — конъюнкция двух предикатов. Первый предикат $\alpha_5 < 1 + \beta_{71}$ показывает, что блок f_5 не может выполняться большее число раз, чем повторяется охватывающий его цикл. Второй предикат $\alpha_5 \div \beta_{71} < (\alpha_2 + \beta_{40}) \div (1 + \beta_{41})$ разрешает первое выполнение блока f_5 только после того, как выполнится блок f_2 столько раз, каково число повторений первого цикла.

В качестве примера построения A-программы рассмотрим вычисление определенного интеграла по формуле трапеций от функции $\sqrt{x \cdot \sin(x)}$. Последовательная программа записана на языке ФОРТРАН (с упрощенными операторами ввода-вывода)

```
READ (A, B, N)
H = (B — A)/N
X = A
S = 0
K = N — 1
I = 0
Y = SQRT (A * SIN (A))
Z = SQRT (B * SIN (B))
10 I = I + 1
   X = X + H
   S = S + SQRT (X * SIN (X))
   IF (I.NE.K) GOTO 10
   TR = H * (S + 0,5 * (Y + Z))
   WRITE (A, B, TR)
   STOP
   END
```

A-программа представлена в табл. 3.

Таблица 3

Блок	Спусковая функция	Оператор программы	Управляющий оператор
1	$\alpha_1 < \alpha_0$	READ (A, B, N)	$\alpha_1 := 1$
2	$\alpha_2 < \alpha_1$	H=(B—A)/N	$\alpha_2 := 1$
3	$\alpha_3 < \alpha_1$	X=A	$\alpha_3 := 1$
4	$\alpha_4 < \alpha_0$	S=0	$\alpha_4 := 1$
5	$\alpha_5 < \alpha_1$	K=N — 1	$\alpha_5 := 1$
6	$\alpha_6 < \alpha_0$	I=0	$\alpha_6 := 1$
7	$\alpha_7 < \alpha_1$	Y=SQRT (A*SIN(A))	$\alpha_7 := 1$
8	$\alpha_8 < \alpha_1$	Z=SQRT (B*SIN(B))	$\alpha_8 := 1$
9	$\alpha_9 < \alpha_6 + \beta_{121}$	10 I=I+1	$\alpha_9 := \alpha_9 + 1$
10	$\alpha_{10} < \alpha_2 + \beta_{121}$ & $\alpha_{10} < \alpha_3 + \beta_{121}$ & $\alpha_{10} < 1 + \alpha_{11}$	X=X+H	$\alpha_{10} := \alpha_{10} + 1$
11	$\alpha_{11} < \alpha_{10}$ & $\alpha_{11} < \alpha_4 + \beta_{121}$	S=S+SQRT (X*SIN(X))	$\alpha_{11} := \alpha_{11} + 1$
12	$\alpha_{12} < \alpha_9$ & $\alpha_{12} < \alpha_5 + \beta_{121}$	IF (I. NE. K) GOTO 10	$\alpha_{12} := \alpha_{12} + 1$; if I=k then $\beta_{120} := 1$ else $\beta_{121} := \beta_{121} + 1$
13	$\alpha_{13} < \alpha_2$ & $\alpha_{13} < \alpha_7$ & $\alpha_{13} < \alpha_8$ & $\alpha_{13} < (\alpha_{11} + \beta_{120}) \dot{-} (1 + \beta_{121})$	TR=H*(S+0,5*(Y+Z))	$\alpha_{13} := 1$
14	$\alpha_{14} < \alpha_1$ & $\alpha_{14} < \alpha_{13}$	WRITE (A, B, TR)	$\alpha_{14} := 1$
15	$\alpha_{15} < \alpha_{14}$	STOP	$\alpha_{15} := 1$

Данная *A*-программа не обладает той потенциальной асинхронностью, которая содержится в исходной программе. Например, блок 10 не может выполняться независимо от блока 11, который использует результаты его работы.

Потенциальную асинхронность можно сохранить, если ввести массив *X*. Измененный фрагмент программы представлен в табл. 4.

Таблица 4

9	$\alpha_9 < \alpha_6 + \beta_{121}$	10 I=I+1	$\alpha_9 := \alpha_9 + 1$
10	$\alpha_{10} < \alpha_2 + \beta_{121}$ & $\alpha_{10} < \alpha_3 + \beta_{121}$	X(α_{10}+1)=X(α_{10})+H	$\alpha_{10} := \alpha_{10} + 1$
11	$\alpha_{11} < \alpha_{10}$ & $\alpha_{11} < \alpha_4 + \beta_{121}$	S=S+SQRT (X(α_{11}+1)* SIN (X(α_{11}+1)))	$\alpha_{11} := \alpha_{11} + 1$
12	$\alpha_{12} < \alpha_5 + \beta_{121}$ & $\alpha_{12} < \alpha_9$	IF (I. NE. K) GOTO 10	$\alpha_{12} = \alpha_{12} + 1$; if I=K then $\beta_{120} := 1$ else $\beta_{121} := \beta_{121} + 1$

При этом упрощаются спусковые функции, и управляющие переменные можно использовать для индексации в операторах программы.

Теперь блок 10 может выполняться независимо от блока 11, и вообще, теряется привычное для последовательного программирования понятие цикла: блоки 9 и 12 выполняются независимо от блоков 10 и 11. Управляющая переменная β_{121} «открывает дорогу» столькими вычислениям всех операторов, каково число повторений цикла.

В данной программе наибольшую трудоемкость исполнения имеют блоки 7, 8, 11; содержащие вычисления элементарных функций SIN и SQRT. Покажем на примере блока 11, что может дать распараллеливание выражений. Естественно, что при этом асинхронность программы должна возрасти.

Оператор $S = S + SQRT(X \times SIN(X))$ можно представить в виде следующих операторов:

$U = SIN(X)$

$R = X * U$

$T = SQRT(R)$

$S = S + T$

Введем массивы U, R, T и управляющие переменные γ_1, γ_2, γ_3. Теперь блок 11 будет выглядеть, как представлено в табл. 5.

Таблица 5

11а	$\gamma_1 < \alpha_{10}$ & $\gamma_1 < 1 + \beta_{121}$	$U(\gamma_1+1) = SIN(X(\gamma_1+1))$	$\gamma_1 := \gamma_1 + 1$
11б	$\gamma_2 < \alpha_{10}$ & $\gamma_2 < \gamma_1$ & $\gamma_2 < 1 + \beta_{121}$	$R(\gamma_2+1) = X(\gamma_2+1) * U(\gamma_2+1)$	$\gamma_2 := \gamma_2 + 1$
11в	$\gamma_3 < \gamma_2$ & $\gamma_3 < 1 + \beta_{121}$	$T(\gamma_3+1) = SQRT(R(\gamma_3+1))$	$\gamma_3 := \gamma_3 + 1$
11г	$\alpha_{11} < \alpha_4 + \beta_{121}$ & $\alpha_{11} < \gamma_3$	$S = S + T(\alpha_{11}+1)$	$\alpha_{11} := \alpha_{11} + 1$

Все операторы могут выполняться параллельно, контроль за количеством вычислений осуществляется с помощью управляющей переменной β_{121}. Управляющие переменные γ_i и α_i фиксируют фактическое число вычисленных операторов.

Распараллеливание выражений — отдельная тема, и мы здесь не будем ее подробно рассматривать. Отметим только, что для задачи распараллеливания выражений хорошо подходят алгоритмы синтаксического анализа, используемые в компиляторах. Например, синтаксические анализаторы выдают на выходе текст на некотором языке, который можно представить в виде ярусного графа. При анализе выражений на каждом ярусе располагаются операции, которые можно выполнять параллельно.

Глава 5. РЕАЛИЗАЦИЯ ВЫЧИСЛИТЕЛЬНОГО ПРОЦЕССА, УПРАВЛЯЕМОГО ПОТОКОМ ДАННЫХ

Трудности, которые возникают при переходе от решения задачи на одной машине к решению задачи, состоящей из взаимозависимых частей (сегментов), на нескольких машинах, в основном определяются тем, что существующие методы описания вычислительного процесса носят последовательный характер, а большинство существующих способов задания программ также подразумевают последовательный процесс реализации алгоритмов.

В последнее время в языках программирования появились средства, которые позволяют отразить при написании программы возможность распараллеливания процесса реализации программы. Хотя эти средства (см. гл. 2) дают программисту весьма широкие возможности для организации параллельно-последовательного вычислительного процесса, однако программист должен предварительно проделать весьма большую работу по выделению сегментов программы и допустимых ветвей, идентифицировать эти ветви и указать, когда и какую именно ветвь процесса следует начать выполнять, обеспечить необходимую синхронизацию и взаимное управление ветвями. Все это делает работу программиста чрезвычайно трудоемкой и не позволяет на практике полностью использовать возможности параллельной реализации вычислительного процесса. Поэтому необходимо переложить работу по распараллеливанию вычислительного процесса на саму вычислительную систему.

В настоящей главе речь пойдет о путях решения этой проблемы на основе использования принципа управления вычислениями потоком данных.

В отличие от классического принципа управления, когда порядок выполнения операторов в программе задан заранее, при управлении потоком данных порядок выполнения операторов в программе определяется динамически (выполняются те операторы, для которых подготовлены данные). В каждый момент времени осуществляется проверка условий готовности операторов к работе и выполнение тех операторов, которые удовлетворяют условиям готовности. Такой способ управления позволяет сохранить при составлении программы и легко использовать при выполнении естественный параллелизм, присущий практически любой задаче. При этом полностью исключается проблема синхронизации, так как выполняются всегда только те операторы, для которых уже закончился процесс вычисления необходимых данных.

Такой подход сильно отличается от ставшего традиционным обычного представления.

Обычные программы считаются по умалчиванию последовательными, использующими «естественный» порядок выполнения команд (следующая команда выбирается по значению счетчика команд). Управление от данных использует противоположный подход — любая программа считается по умалчиванию параллельной.

Что же представляет из себя программа, использующая принцип управления потоком данных?

Рассмотрим задачу определения решения системы двух линейных неоднородных уравнений

$$\left.\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned}\right\}.$$

В случае, если

$$\Delta = \begin{vmatrix} a_{11}a_{12} \\ a_{21}a_{22} \end{vmatrix} = a_{11} \cdot a_{22} - a_{12} \cdot a_{21} \neq 0,$$

то данная система имеет решение

$$x_1 = \frac{1}{\Delta}\,(a_{22}b_1 - a_{12}b_2),$$

$$x_2 = \frac{1}{\Delta}\,(a_{11}b_2 - a_{21}b_1).$$

На рис. 25, *а* представлена блок-схема обычной программы вычисления значений x_1 и x_2, а на рис. 25, *б* представлен граф потока данных программы вычисления значений x_1 и x_2.

В вершинах графа находятся команды, а данные для них располагаются на ребрах. При таком представлении связь между командами программы осуществляется только через данные. Таким образом, программа потока данных представляет собой граф, вершинами которого являются операторы, соединенные ребрами, которые определяют данные. Операции, указанные в вершинах, выполняются только над данными на входах вершин и не зависят от соседних вершин. Ребра графа имеют обозначения, соответствующие значениям данных. Эти значения данных представляют собой промежуточный результат вычислений, который еще не был использован. Появление необходимых для выполнения данной команды операндов приводит к выполнению этой команды. Таким образом, в этом случае порядок выполнения команд определяется динамически в зависимости от наличия необходимых данных. Представление, использующее управление потоком данных, позволяет добиться максимально возможного распараллеливания программы. В любой программе всегда имеются независимые команды, которые могут выполняться одновременно. Но выделение таких команд в обычной программе сопряжено с очень большими трудностями, определяемыми статическим,

наперед заданным порядком выполнения команд, в то время как в программе, использующей управление потоком данных, определение команд, которые могут быть выполнены в данный момент времени, является естественной частью процесса выполнения программы.

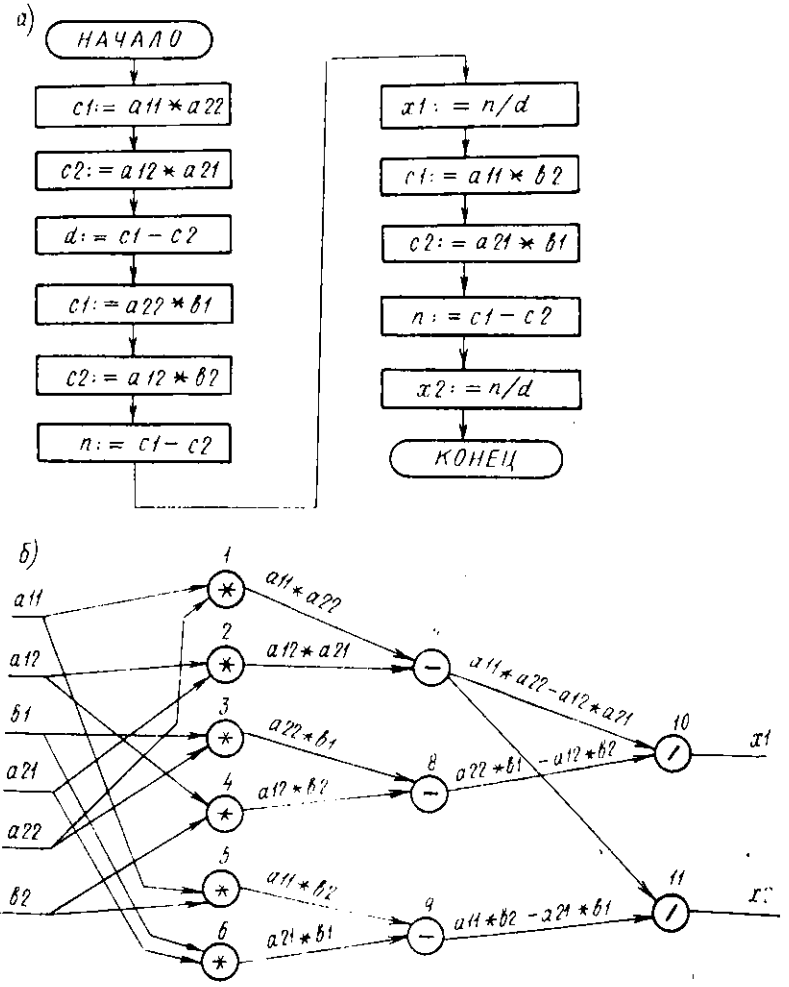


Рис. 25

Рассмотрим основные черты механизма выбора на исполнение команд программы, использующей управление потоком данных. Как уже отмечалось выше, программа представлена в таком виде, когда связь между командами осуществляется через данные. В этом случае связь между командами указа-

на явно, т. е. известно, какой команде должен быть передан результат выполнения данной команды. Из примера рис. 25, *б* видно, что для выполнения команды *7* необходимы только результаты выполнения команд *1* и *2*, а, в свою очередь, результат выполнения команды *7* необходим для выполнения команд *10* и *11*. Команда считается готовой и направляется на выполнение, когда поступили все необходимые для данной команды операнды.

Таким образом, задача определения готовых к выполнению команд (команд, которые можно выполнять одновременно) сводится к задаче сравнения числа операндов, поступивших для данной команды, с числом операндов, необходимых для ее выполнения. И в случае совпадения этих чисел команда может быть выполнена.

§ 9. Команды языка потока данных

Будем говорить, что ребро графа потока данных приобретает значение, когда соответствующее этому ребру данное становится определенным. Перед началом выполнения программы определены только исходные данные и константы.

Команда становится активной, когда все ребра, входящие в соответствующую данной команде вершину графа, имеют значение. Команда становится активизированной, когда хотя бы одно из ребер, входящих в вершину, имеет значение. Будем говорить, что команда неактивна, когда ни одно из входящих в нее ребер не имеет значения. При выполнении программы команда может быть выполнена только в том случае, когда она активна. Все активные команды могут выполняться независимо друг от друга и параллельно. Активная команда вычисляет выходные значения по входным значениям данных, снимает значения с входящих в нее ребер, выставляет значения выходящим из нее ребрам и становится неактивной. Таким образом, выполнение команды зависит только от информации, локальной по отношению к данной команде. В программе обработки потока данных последовательность выполнения активных команд не влияет на конечный результат. А поскольку команды выполняются по мере готовности операндов на их входах, то рассматриваемая форма представления не накладывает каких-либо ограничений на последовательность выполнения команд.

Рассмотрим структуру команд и данных в вычислительной машине, реализующей программы, представленные в виде графа потока данных. Рассматриваемый командный уровень соответствует традиционному машинному уровню в обычных машинах. Форматы команд показаны на рис. 26, *а, б*. Команды состоят из четырех или шести полей. Поле *КОП* содержит

код операции, поле *ПА* — признак активности команды. Поле *АДР* содержит адрес (адреса) последующей команды (команд), поле *ПО* — признаки очередности операндов. Например, для команд, показанных на рис. 25, *б*, команда *2* будет содержать в поле *КОП* ∗, а в поле *АДР* — указание на команду 7, т. е. будет выглядеть так:

∗	0	7	1

а команда *7* будет выглядеть так:

—	0	10	0	11	0

Значения полей *АДР 1* и *АДР 2* — 10 и 11 — это номера следующих за командой *7* команд.

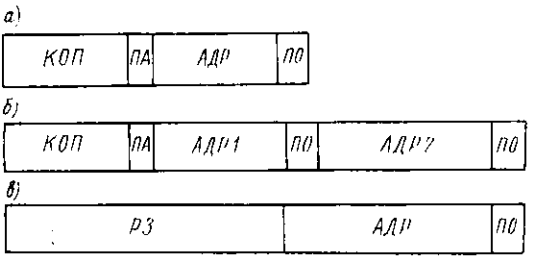


Рис. 26

Формат данного показан на рис. 26, *в*. Поле *РЗ* содержит результат выполнения команды, а поля *АДР* и *ПО*, соответственно, адрес команды и признак очередности операндов. Так, например, данное, полученное после выполнения команды *2*, в поле *РЗ* будет содержать результат, в поле *АДР* — 7, а в поле *ПО* содержать 1 (данное является вторым операндом в команде *7*). Таким образом, данное, вычисленное командой *2*, будет выглядеть так:

$a12 * a21$	7	1

После выполнения команды *1* появится такое данное:

$a11 * a22$	7	0

где 0 в поле *ПО* указывает на то, что это данное является первым операндом команды 7. Когда на процессор поступает данное, то по содержимому поля *АДР* осуществляется чтение команды. Производится анализ признака активности (для унарных операций признак активности всегда находится в состоянии 1). Для бинарной операции состояние признака

активности 0 означает, что поступившее данное является первым из двух необходимых для выполнения команды операндов. В этом случае значение поступившего данного запоминается вместе с командой в ассоциативном запоминающем устройстве (АЗУ), а признак активности в команде устанавливается в состояние 1.

В случае, если команда была активизирована ранее, т. е. поступившее данное является вторым из двух необходимых для выполнения команды операндов, команда становится активной и направляется на выполнение.

Выполнение команды осуществляется в два этапа: на первом этапе по входным значениям данных вычисляется результат. При этом в поле *АДР* и *ПО* результата помещается содержимое полей *АДР* и *ПО* команды, а в поле *РЗ* помещается результат выполнения указанной в поле *КОП* операции. Признак активности в команде устанавливается в состояние 0 (команда становится неактивной). На втором этапе осуществляется вызов и передача значения результата команде (командам), адрес(а) которой (ых) указан(ы) в поле *АДР* Например, до выполнения команда *3* (рис. 25, *б*) выглядит так:

*	0	8	0

а результат команды *3* отсутствует (не имеет значения). После выполнения команды 3 появится результат

*a*22 * *b*1	8	0

До выполнения команда *7* выглядит так:

—	0	10	0	11	0

В результате выполнения команды *7* появляются данные (результаты):

*a*11 * *a*22	10	0

*a*11 * *a*22	11	0

В программах, представленных в виде графа потока данных, не делается различия между командами, принадлежащими информационному графу и графу управления, так как выполнение тех и других команд происходит одинаково. Единственное отличие состоит в том, что в случае выполнения команды, принадлежащей информационному графу, мы говорим, что получили данное, а в случае выполнения команды, принадлежащей графу управления, мы говорим, что получили управляющее данное.

Основной набор команд включает в себя следующие типы команд:

— команды чтения и записи;
— бинарные операции, например, +, —, *, / и т. п.;
— унарные операции;
— команды сравнения.

Команды чтения и записи имеют вид рис. 26, *б*. По команде *ЧТЕНИЕ* осуществляется занесение содержимого ячейки ОЗУ, адрес которой указан в поле *АДР 1* команды, в поле *РЗ* результата. В поле *АДР* результата заносится значение поля *АДР 2* команды. Команда *ЧТЕНИЕ* становится активной при поступлении на нее любого данного.

По команде *ЗАПИСЬ* содержимое поля *РЗ* поступившего данного запоминается в ячейке ОЗУ, по адресу, указанному в поле *АДР 1*. Формируется выходное данное (результат), у которого в поле *РЗ* помещается управляющее данное, а в поле *АДР* — значение поля *АДР 2* команды.

Бинарные и унарные операции используют форматы *а* и *б* рис. 26. Команды сравнения используют формат *б* рис. 26. При выполнении команд сравнения осуществляется сравнение первого операнда со вторым и формирование данного, где в поле результата заносится значение первого операнда, а в поле *АДР* — значение поля *АДР 1* команды при выполнении условия или значение поля *АДР 2* при невыполнении условия.

§ 10. Трансляция и настройка программ на конфигурацию вычислительной системы

При описании системы команд мы исходили из того, что программа, которую предстоит выполнять, уже представлена в виде графа потока данных. Такое представление может быть выполнено человеком аналогично программированию на обычных языках. Но при этом программисту придется мыслить в терминах языка потока данных и иметь дело со множеством одновременно протекающих процессов, что связано в свою очередь с трудностями психологического характера, неспособностью человека эффективно выполнять такую работу. Поэтому работа, связанная с переводом программы, написанной на некотором входном языке, во внутреннее представление в виде графа потока данных, должна осуществляться автоматически самой вычислительной системой.

В обычной последовательной программе присутствует практически вся информация, необходимая для представления программы в виде графа потока данных. Однако следует заметить, что использование обычных языков, ориентированных на последовательное выполнение программы, для авто-

матического получения графа потока данных не позволяет достичь максимальной присущей задаче степени параллелизма. В этом случае речь может идти лишь о распараллеливании линейных участков программы. Но при принятии определенных допущений и введении расширений в существующие языки при автоматическом преобразовании программы в представление потока данных может быть получен граф программы, который позволяет добиться при выполнении максимального параллелизма.

Рассмотрим алгоритм преобразования обычной программы в программу, представленную в виде графа потока данных.

а)

```
НАЧАТЬ;
ВВОД a11, a12, b1;
ВВОД a21, a22, b2;
c1 := a11 * a22
c2 := a12 * a21
d := c1 - c2
l1 := a22 * b1
l2 := a12 * b2
n := L1 - L2
x1 := n / d
k1 := a11 * b2
k2 := a21 * b1
m := k1 - k2
x2 := m / d
ВЫВОД x1, x2;
КОНЕЦ
```

б)

N команды				
1	ВВОД	—	—	a11
2	ВВОД	—	—	a12
3	ВВОД	—	—	b1
4	ВВОД	—	—	a21
5	ВВОД	—	—	a22
6	ВВОД	—	—	b2
7	*	a11	a22	c1
8	*	a12	a21	c2
9		c1	c2	d
10	*	a22	b1	L1
11	*	a12	b2	L2
12	-	L1	L2	n
13	/	N	0	x1
14	*	a11	b2	k1
15	*	a21	b1	k2
16	—	k1	k2	m
17	/	M	d	x2
18	ВЫВОД	x1	—	—
19	ВЫВОД	x2	—	—

Рис. 27

В качестве примера возьмем уже знакомую нам задачу определения решения системы двух линейных неоднородных уравнений. Текст этой программы представлен на рис. 27, *а*. На основе этого текста с помощью известных преобразований (обычного транслятора) получим эту же программу, но уже в тетрадном представлении (*КОП*, операнд 1, операнд 2, результат). Тетрадное представление программы показано на рис. 27, *б*. Наша задача заключается в преобразовании этой программы в программу потока данных.

Преобразование осуществляется путем двухкратного просмотра текста программы в тетрадном представлении. При первом просмотре производится заполнение таблицы операндов. При этом по каждому идентификатору операнда осуще-

ствляется запись номера команды, использующей этот операнд, в соответствующее поле таблицы операндов. Заполненная таким образом таблица операндов показана на рис. 28, *а*. При втором просмотре текста программы по идентификаторам результатов осуществляется обращение в таблицу операндов и поля, содержащие номера команд, переписываются в просматриваемую строку программы. Например, в команде *1* стоит идентификатор результата *a11*. Обращаемся к таблице операндов по этому идентификатору. В таблице операндов в строке, содержащей *a11*, находятся номера команд *7*

а)

ОПЕРАНД	N команды	N команды
a11	7	14
a12	8	11
b1	10	15
a21	8	15
a22	10	7
b2	11	14
c1	9	
c2	9	
d	13	17
f1	12	
f2	12	
n	13	
x1	18	
k1	16	
k2	16	
m	17	
x2	19	

б)

N команды	КОП	РЕЗУЛЬТАТ	N команды	N команды
1	ВВОД	a11	7	14
2	ВВОД	a12	8	11
3	ВВОД	b1	10	15
4	ВВОД	a21	8	15
5	ВВОД	a22	10	7
6	ВВОД	b2	11	14
7	*	c1	9	
8	*	c2	9	
9	–	d	13	17
10	*	f1	12	
11	*	f2	12	
12	—	n	13	
13	/	x1	18	
14	*	k1	16	
15	*	k2	16	
16	–	m	17	
17	/	x2	19	
18	ВЫВОД	—	—	
19	ВЫВОД	–	–	

Рис. 28

и *14*. Эти номера переписываем в строку команды *1* после идентификатора *a11*. Таким образом преобразованный текст программы представлен на рис. 28, *б*. Граф программы, полученный на основе этого текста, представлен на рис. 29 (цифрами обозначены номера команд, идентификаторы результатов проставлены над дугами).

Программа, оттранслированная по описанной выше схеме, может быть непосредственно выполнена на ВС с общей памятью и центральным диспетчерским устройством, т. е. на ВС с централизованным управлением. Но для выполнения программы на ВС с разделенной памятью и децентрализованным управлением необходимо осуществить дальнейшие преобразования. Основная цель дальнейших преобразований

заключается в том, чтобы разбить программу на такие блоки, которые удовлетворяют требованиям минимального обмена данными, т. е. при распределении блоков по вычислительным машинам системы обмен данными между блоками в процессе решения задачи должен осуществляться на минимально необходимом уровне.

Первый этап дальнейших преобразований заключается в распределении команд программы по ярусам. Исходным для распределения команд по ярусам является представление программы в виде, показанном на рис. 28, *б*. Все команды ввода данных считаем принадлежащими к нулевому ярусу. Распределение по ярусам осуществляем следующим образом:

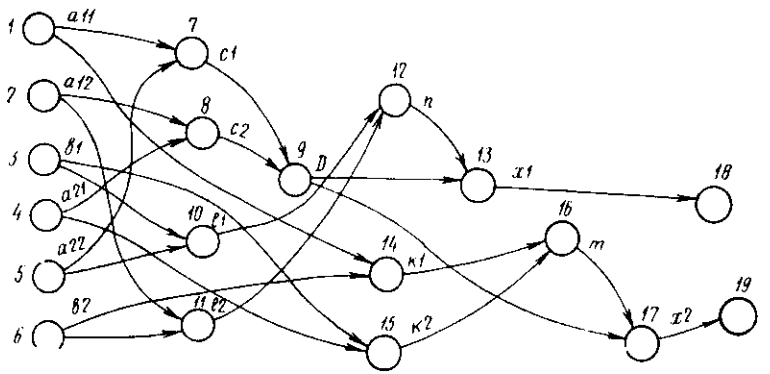


Рис. 29

Просматриваем команды нулевого яруса, все команды, на которые ссылаются команды нулевого яруса, считаем принадлежащими к *I* ярусу. Когда все команды нулевого яруса просмотрены, начинаем просматривать команды *I* яруса. Все команды, на которые ссылаются команды *I* яруса, считаем принадлежащими ко *II* ярусу. И соответственно, просматривая команды i-го яруса, считаем все команды, на которые указывают команды i-го яруса, принадлежащими к $i+1$ ярусу. В случае если команда, на которую указывает ссылка, принадлежит к множеству команд просматриваемого яруса, производится пометка такой команды. После просмотра всех команд данного яруса помеченные команды исключаются из множества команд, принадлежащих к данному ярусу. Распределение команд по ярусам заканчиваем, когда при просмотре очередного яруса оказывается, что все команды, принадлежащие данному ярусу, не имеют ссылок. С помощью изложенного алгоритма осуществим распределение по ярусам вершин графа, представленного на рис. 30. Вершины *1*, *3* принадлежат к нулевому ярусу. На первом шаге после просмотра нулевого яруса к *I* ярусу отнесем вершины *4*, *5*—*8*. После

просмотра *I* яруса ко *II* ярусу отнесем вершины *5, 9* и помеченные вершины *5, 6, 7, 8* исключим из множества команд *I* яруса. На шаге 3 вершины *6, 7, 8, 9* будут отнесены к *III* ярусу и исключены из множества команд *II* яруса. На шаге 4 к *IV* ярусу отнесем вершины *7, 9* и исключим их из множества команд *III* яруса. И, наконец, на шаге 5 к *V* ярусу будет отнесена вершина *9*, и она же будет исключена из множества вершин *IV* яруса. Окончательное распределение команд по ярусам выглядит так: *0* ярус — *1, 2, 3*; *I* ярус — *4*; *II* ярус — *5*; *III* ярус — *6, 8*; *IV* ярус — *7*; *V* ярус — *9*. Чтобы убедиться в правильности распределения по ярусам, достаточно вспомнить, что к *i*-му ярусу могут быть отнесены только те операторы, которые зависят по крайней мере от одного оператора $(i-1)$-го яруса и не зависят ни от одного оператора, не вошедшего в ярусы с номерами, меньшими *i*.

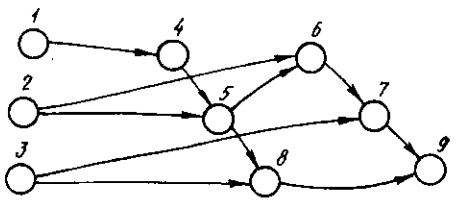


Рис. 30

На рис. 31 показано распределение по ярусам команд знакомой нам программы. Цифрами, как и прежде, обозначены

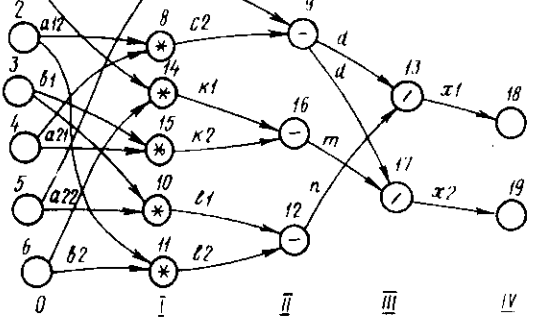


Рис. 31

номера команд, а идентификаторы результатов проставлены над дугами. Из рис. 31 мы видим, что команды, принадлежащие к одному ярусу, независимы, т. е. при поступлении необходимых данных они могут выполняться одновременно (параллельно). А для обеспечения параллелизма при решении задачи команды, принадлежащие к одному ярусу, должны быть помещены на разные вычислительные машины системы.

При этом максимальный параллелизм при решении задач будет получен тогда, когда число ВМ системы, по которым распределяются команды, не меньше числа команд программы, принадлежащих ярусу максимальной ширины. После первого этапа программа может быть разбита на блоки, выполнение которых на разных ВМ системы будет осуществляться параллельно, но обмен данными между блоками будет превышать минимально необходимый, так как при таком разбиении не учитывается связь между командами разных ярусов.

На втором этапе разбиения (на блоки с минимальным обменом) проведем выделение команд разных ярусов, зависимых по данным. Говоря на языке графов, наша задача заключается в том, чтобы определить множество несовпадающих простых путей максимальной длины. Несовпадающими считаются простые пути, которые не содержат ни одной общей вершины.

Будем двигаться по графу программы в соответствии со следующими правилами. Построение пути заключается в переходе из одной непомеченной вершины графа в другую. Первый путь начинаем строить из любой непомеченной вершины нулевого яруса. Переход из вершины в вершину может быть осуществлен только в том случае, если эти вершины соединены дугой. Вершина, на которую осуществлен переход, помечается. Если из вершины выходит несколько дуг, то при переходе по одной из них остальные запоминаются:

— в очереди яруса, к которому принадлежит начальная вершина дуги, если конечная вершина этой дуги не помечена;

— в массиве связей данного пути, если конечная вершина этой дуги помечена.

Построение пути заканчивается, когда на очередном шаге из вершины не выходит ни одной дуги, или все выходящие дуги имеют помеченные конечные вершины. Новый путь начинаем строить с любой непомеченной вершины старшего (нулевого) яруса, а если таких вершин нет, то обращаемся в очередь яруса. Если очередь яруса не пуста, начинаем новый путь с любой непомеченной вершины, на которую указывает дуга из очереди яруса. Дуга, указывающая на вершину, после пометки вершины удаляется из очереди яруса. Если очередь яруса пуста, обращаемся к очереди яруса $i + 1$. Построение путей заканчиваем, когда на всех ярусах окажутся пустые очереди. Проведем построение несовпадающих простых путей на графе рис. 32, *а*. При построении будем использовать:

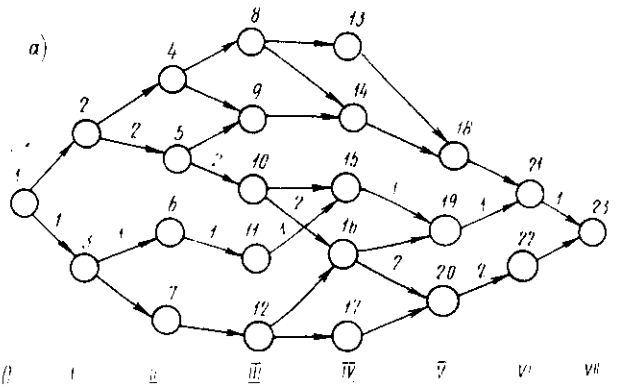
— таблицу принадлежности команд к ярусам и путям (рис. 32, *б*);

— таблицу очередей ярусов (рис. 32, *в*);

— массивы связей путей (рис. 32, *г*).

В начальный момент все вершины графа не помечены.

На рис. 32, *б*, *в* и *г* строки № пути в таблицах пусты. Построение пути *1* не вызывает никаких трудностей. Предположим, что мы построили путь, проходящий через вершины *1*, *3*, *6*, *11*, *15*, *19*, *21*, *23*. После первого шага эти команды будут помечены. Отметки внесем в таблицу рис. 32, *б* (помеченные вершины при дальнейших шагах не могут быть внесены в другие пути). В очередях нулевого и первого яруса появятся ука-



б)

N команды	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
N яруса	0	I	I	II	II	II	II	III	III	III	III	III	IV	IV	IV	IV	IV	V	V	V	VI	VI	VII
N пути	1	2	1		2	1				2	1				1	2			1	2	1	2	1

в)

N яруса	N команды
0	
I	2, 4
II	9

г)

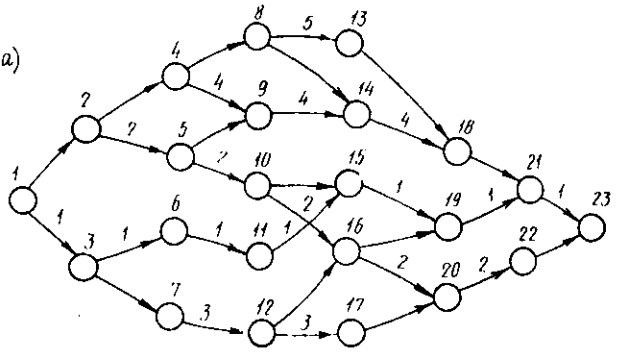
N пути	СВЯЗИ
1	2, 2, 2, 2
2	1, 1, 1, 1
3	
4	
5	

Рис. 32

затели на команды *2* и *7* соответственно. Второй путь начинаем с команды *2*. В массив связей пути *1* заносим признак его связи с путем *2*, а в массив связей пути *2* — признак его связи с путем *1*. Продолжая построение, получим путь *2—2*, *5*, *10*, *16*, *20*, *22*. Помечаем эти команды в таблице рис. 32, *б*. В очереди *I* яруса появится указание на команду *4*, а в очереди *II* яруса — на команду *9*. Очередь нулевого яруса опустеет. В массивах связей путей 1 и 2 появится еще по три метки связей путей. Продолжая описанные действия, полу-

чаем окончательное разбиение команд по путям (рис. 33). Таблицы рис. 32, *б* и *г* заполнены, таблица рис. 33, *в* пуста, цифры, помеченные звездочкой, обозначают прошлое состояние очередей. Номер пути указан цифрами над дугами, принадлежащими соответствующему пути.

Построение несовпадающих простых путей по рассмотренному алгоритму приводит к тому, что каждый построенный



б)

	0	I	II	III	IV	V	VI	VII
N команды	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
N яруса	0	I	I	II	II	II	II	III	III	III	III	IV	IV	IV	IV	IV	V	V	V	V	VI	VI	VII
N пути	1	2	1	4	2	1	3	5	4	2	1	3	5	4	1	2	3	4	1	2	1	2	1

в)

N яруса	N команды
0	1*
I	7*, 4*
II	9*, 8*

г)

N пути	Связи
1	1, 2, 2, 2, 3, 4
2	1, 1, 1, 1, 3, 3, 4, 4
3	1, 2, 2
4	2, 2, 1, 5, 5, 5
5	4, 4, 4

Рис. 33

путь содержит только те команды, которые имеют непосредственную связь по данным. Команды, принадлежащие одному пути, при распределении программы по ВМ системы с целью получения минимума передач должны быть помещены на одну ВМ системы. Правило распределения по ВМ системы в этом случае выглядит так:

1. Если число ВМ системы не меньше максимальной ширины графа программы, то команды, принадлежащие разным путям, помещаются на разные ВМ системы.

2. Если число выделенных для решения задачи ВМ меньше максимальной ширины графа, то на разные n ВМ помещаются команды, принадлежащие n самым длинным путям программы.

Команды, принадлежащие оставшимся путям, распределяются на эти же n ВМ с учетом их связей с уже распределенными командами.

Например, при распределении программы, граф которой показан на рис. 33, по 3 ВМ, команды, принадлежащие пути *1*, будут помещены на 1-ю машину; пути *2* — на 2-ю машину; пути *4* — на 3-ю машину. А команды оставшихся путей *3* и *5* будут помещены соответственно на 2 и 3-ю машины.

Структура вычислительных систем, управляемых потоком данных

Рассмотрение различных вариантов реализации принципа управления потоком данных в аппаратуре начнем с однопроцессорной системы рис. 34. Несмотря на то, что представление программы в виде графа потока данных предполагает ее выполнение на многопроцессорной системе, эффект ускорения

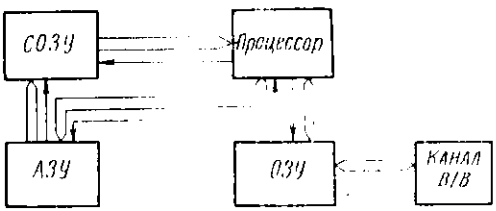


Рис. 34

решения задачи может быть получен и на однопроцессорной системе за счет ликвидации лишних пересылок промежуточных результатов. Кроме того, однопроцессорная модель окажется полезной при дальнейшем рассмотрении.

Выполнение программы, записанной в виде графа потока данных в *ОЗУ* (рис. 34), осуществляется следующим образом: после вычисления результата очередной команды процессор осуществляет обращение к АЗУ, где хранятся активизированные команды вместе с вычисленными ранее операндами. В качестве ключа поиска в *АЗУ* выступает значение поля адреса команды, находящейся на выполнении в процессоре. В случае, если искомая команда находится в *АЗУ*, осуществляется передача этой команды вместе с необходимыми для ее выполнения операндами в сверхоперативную память *СОЗУ* (первый из операндов был вычислен ранее и хранился в *АЗУ* вместе с искомой командой, а второй операнд

поступил в *АЗУ* вместе с ключом поиска (с адресом искомой команды)). В *СОЗУ* находится очередь активных команд, т. е. команд, готовых к выполнению. Процессор переходит к выполнению следующей команды, т. е. осуществляет выборку очередной команды из *СОЗУ*. В случае, если искомая команда отсутствует в АЗУ, т. е. команда не активизирована, осуществляется обращение к *ОЗУ*, выборка искомой команды и засылка вместе с вычисленным операндом в *АЗУ*. После этого процессор переходит к выполнению очередной команды, т. е. выбору ее из очереди активных команд (*СОЗУ*), вычислению результата и повторению описанных выше действий.

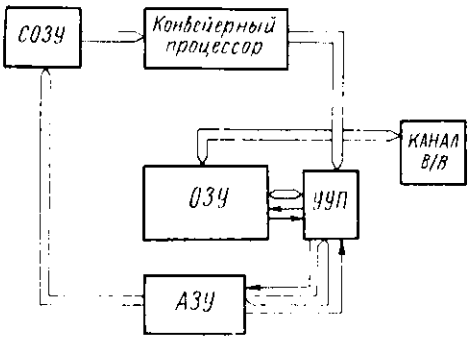


Рис. 35

Рассмотренная однопроцессорная система позволяет получить определенный выигрыш в скорости решения задач по сравнению с однопроцессорными системами, использующими традиционное управление, но в отличие от многопроцессорных систем не позволяет реализовать потенциальный параллелизм, присущий задачам. Перейдем к рассмотрению многопроцессорных систем, использующих управление от данных.

На рис. 35 показана конвейерная система обработки потока данных. В отличие от структуры рис. 35 в этом случае введено два новых блока — конвейерный процессор и устройство управления памятью (*УУП*). Назначение остальных элементов такое же, как и на рис. 34. В *СОЗУ* находится очередь активных команд, питающая конвейерный процессор, в *АЗУ* хранятся активизированные команды, а *ОЗУ* содержит программу. Выполнение команды разбито на два этапа. Первый этап — вычисление результата на конвейерном процессоре (число одновременно выполняемых команд равно числу слоев процессора) заканчивается передачей результата вместе с адресом следующей команды в *УУП*. На втором этапе *УУП* активизирует следующую команду, т. е. производит ее засылку из *ОЗУ* в *АЗУ* или производит засылку команды из *АЗУ* в очередь активных команд в *СОЗУ*. Выполнение первого или второго действия на этом этапе зависит от наличия искомой команды в АЗУ.

В отличие от традиционных конвейерных систем, система, показанная на рис. 35, позволяет эффективно использовать

мощность конвейерного процессора при решении одной задачи, так как в этом случае в конвейере отсутствуют пропуски, связанные с зависимостью команд программы. В *СОЗУ* находятся команды, которые всегда могут выполняться параллельно. Но и в этом случае приходится мириться с недостатком, присущим всем конвейерным системам — с постоянством времени выполнения простых и сложных команд, связанным с трудностями реализации асинхронного конвейерного режима.

На рис. 36 показана система, которая позволяет осуществить параллельное (асинхронное) выполнение команд

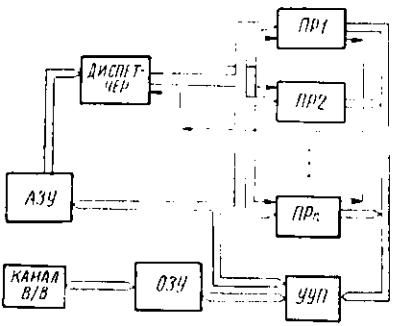


Рис. 36

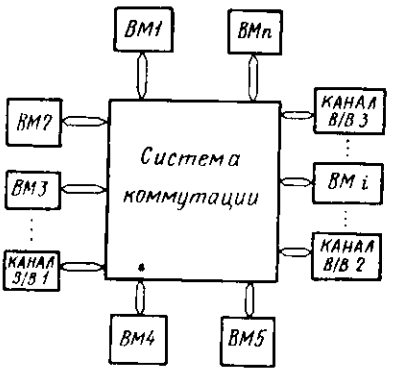


Рис. 37

одной программы. В этом случае очередь активных команд находится в диспетчере (*ДИСП*), который осуществляет распределение команд по процессорам (*ПР1*, ..., *ПРn*). Любой из процессоров, закончивший вычисление результата, посылает его в *УУП* и сообщает диспетчеру о том, что он освободился. Диспетчер посылает на этот процессор команду из очереди активных команд. Работа остальных блоков аналогична работе этих блоков в ранее рассмотренной системе рис. 35.

Производительность систем рис. 35, 36 может быть увеличена только до определенного предела. Предел производительности определяется пропускной способностью *УУП* и *СОЗУ* в системе на рис. 35 и *ДИСП* и *УПП* в системе на рис. 36. Дальнейшее увеличение производительности может быть получено либо улучшением характеристик элементной базы, либо в системах с децентрализованным управлением. Структура одной из таких систем показана на рис. 37. Структура вычислительной машины (ВМ), используемой в такой системе, показана на рис. 38.

Выполнение программы на такой системе осуществляется в два этапа. Первый этап — этап настройки. Оттранслированная и разбитая на блоки программа размещается по вычислительным машинам системы (алгоритм трансляции и разбиения на блоки был рассмотрен ранее). Каждая ВМ кроме блоков, назначение которых уже известно (процессор обработки, *ОЗУ*, *АЗУ*), содержит процессор связи и память таблиц связи. На этапе настройки математическим адресам связи между блоками в памяти таблиц связи ставится в соответствие физический адрес ВМ, где находится данный логический блок. На втором этапе — этапе выполнения программы — команды, принадлежащие одному логическому блоку, выполняются на одной ВМ. Когда возникает необходимость передать результат выполнения команд данного блока другому логическому

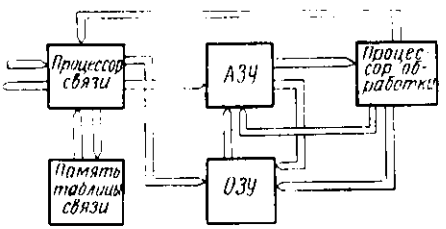


Рис. 38

блоку, размещенному на другой ВМ, процессор обработки передает результат вместе с логическим адресом искомого блока и адресом команды в этом блоке в процессор связи, который при помощи таблицы связей заменяет логический адрес искомого блока на физический адрес ВМ, где этот блок расположен, и передает результат в сопровождении физического адреса системе коммутации. Если на вход процессора связи из системы коммутации поступает результат выполнения от другой ВМ, то по адресу команды он осуществляет ее поиск в *АЗУ*, а при отсутствии команды в *АЗУ* осуществляет ее активацию, т. е. засылку из *ОЗУ* в *АЗУ*. На процессоре обработки осуществляется выполнение активных команд из *АЗУ*. Активацию команд, находящихся в своем *ОЗУ*, также осуществляет процессор обработки. Активизацию команд, находящихся в *ОЗУ* других ВМ, осуществляют процессоры связи. Взаимодействие между процессорами связи осуществляется через систему коммутации. Такая многомашинная система позволяет осуществлять асинхронное (параллельное) выполнение команд одной программы. Производительность системы ограничивается пропускной способностью системы коммутации, и может быть в десятки раз выше, чем у систем, представленных на рис. 35 и 36.

Глава 6. КЛАССИФИКАЦИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

§ 11. Классы вычислительных систем

Все многообразие исполнительных структур можно представить в следующем виде. Введем несколько определений. Потоком команд называется последовательный ряд команд, выполняемый машиной (системой). Потоком данных называется последовательный ряд данных, полученный в результате выполнения соответствующих операций, содержащихся

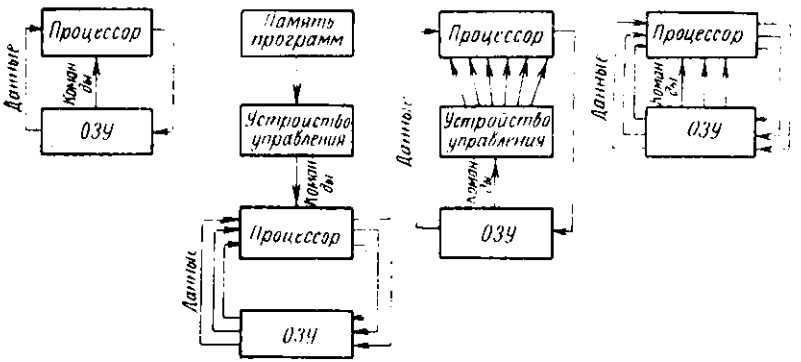


Рис. 39

в командах. Вводится термин *множественность* как синоним параллелизма, который означает максимально возможное в системе число команд или операторов, которое может одновременно находиться на одинаковой стадии обработки. По признаку множественности вычислительные машины (системы) подразделяются на 4 класса:

I — ОКОД: одиночный поток команд — одиночный поток данных (обычные ЭВМ);

II — ОКМД: одиночный поток команд — множественный поток данных (матричные ЭВМ);

III — МКОД: множественный поток команд — одиночный поток данных (ЭВМ, управляемые потоком данных);

IV — МКМД: множественный поток команд — множественный поток данных (многопроцессорные и многомашинные системы).

На рис. 39 показаны укрупненные структурные схемы четырех классов систем.

Рассмотрим кратко общие принципы параллельной обработки потоков команд и данных в системах указанных классов.

ОКОД

В системах данного класса используется так называемый локальный параллелизм, т. е. совмещение во времени работы *ОЗУ* и процессора. Те промежутки времени, когда процессор занят выполнением операций, могут быть использованы для извлечения из *ОЗУ* очередной команды. Управляющее устройство, осуществляющее «просмотр вперед» команд, должно иметь дополнительные регистры для хранения подготовленных команд. Операторы условной передачи управления снижают эффективность просмотра вперед, поскольку наперед неясно, какую из двух ветвей команд необходимо готовить. В некоторых системах даже готовят обе ветви с выбором в дальнейшем той, на которую указывает оператор перехода. Другим средством увеличения производительности систем ОКОД является расслоение *ОЗУ* на память команд и память данных. В этом случае совмещается во времени выборка на *ОЗУ* команд с выборкой операндов (записью результатов). Дальнейшее увеличение производительности может быть получено организацией многоблочной памяти с попеременным обращением к каждому блоку со сдвигом во времени, равным $t_{эфф} = 1/l\, t_{ОЗУ}$, что и определяет среднее время обращения к многоблочной памяти. Здесь $t_{ОЗУ}$ — время обращения к одному блоку, l — число блоков. Системы с частичным перекрытием циклов выполнения команд были названы конфлюентными, а условие равномерного перекрытия циклов — условием конфлюентности. В этих системах пропускная способность системы ограничивается временем дешифрации команды Δt.

Кроме расслоения *ОЗУ* в конфлюентных системах большое внимание уделяется увеличению производительности исполнительного устройства. Исполнительное устройство представляет собой конструкцию, состоящую из последовательности специализированных блоков, организованных по принципу конвейера. Существенным недостатком конфлюентных систем является снижение производительности при нарушении условия конфлюентности (условие конфлюентности нарушается, когда данная команда использует результат предыдущей, и в случае команд условного перехода). Исследования показывают, что в таких системах максимально достижимое увеличение пропускной способности равно двум при условии, что система еще остается в классе ОКОД (просмотр вперед до десяти команд).

ОКМД

В системах этого класса обычный поток команд воздействует на несколько исполнительных блоков, каждый из ко-

торых обрабатывает различные операнды. Различают три типа систем класса ОКМД.

1. Системы матричного типа. В них одно устройство управления связано с n операционными элементами (ОЭ). Каждый ОЭ имеет свою регистровую часть и локальную память данных, обладает универсальным набором операций, но не имеет памяти команд, а выполняет команды центрального управляющего устройства.

2. Системы конвейерного типа. В такой системе имеется несколько операционных блоков, каждый из которых специализирован на выполнении определенной операции. Каждый блок состоит из некоторого числа ступеней (слоев). Пары операндов последовательно принимаются на первый слой блока, производящего необходимую операцию, а затем синхронно передаются от слоя к слою. Каждый слой осуществляет некоторые элементарные преобразования над каждой парой операндов (сдвиг мантисс, нормализация и т. п.). На последнем слое получается окончательный результат. Прием операндов на первый слой может производиться с частотой синхронизации слоев. С той же частотой результаты появляются на выходе блока. Конвейерный процессор требует организации «конвейерной» памяти, которая обеспечивает высокоскоростную засылку операндов на операционные блоки.

Недостатки конвейерных систем подобны недостаткам конфлюентных — потеря производительности (дыра в конвейере) в случае исполнения команд, зависящих по данным, и команд условного перехода. Единственным выходом из такого положения является использование конвейерных принципов в классе МКМД.

3. Системы ассоциативного типа содержат матрицу операционных элементов (ОЭ), каждый из которых оборудован специальным регистром признака. В центральном блоке управления имеется регистр опроса. Состояние каждого регистра признака устанавливается заранее, до формирования текущей команды, а состояние регистра опроса устанавливается одновременно с формированием команды. В тех ОЭ, где результат сравнения положителен, выполняется операция под контролем центрального блока управления. Остальные ОЭ в это время могут простаивать. В матричном и ассоциативном процессорах возможно наделить каждый ОЭ аппаратурой для независимой индексации данных.

Для всех систем ОКМД характерна деградация производительности при появлении команд условного перехода, при несоответствии размеров логического вектора, который должен быть вычислен, и размеров физического вектора (массива) операционных элементов, с помощью которых вычисляется этот вектор. Для систем ОКМД из-за деградации общая

производительность в зависимости от числа ОЭ растет не по линейному закону, а по закону логарифма: $P_{\text{окмд}} = \alpha \cdot \log_2 m$, где α — коэффициент пропорциональности, m — число ОЭ.

МКОД

В системах данного типа выборка команд для исполнения определяется готовностью данных, поэтому эти системы называют системами, управляемыми потоком данных. Каждый исходный или промежуточный результат (данное) может сделать готовыми для исполнения одну или несколько команд, зависящих от этого данного. Выполняемые команды могут группироваться в потоки, например, по коду операции (как в конвейерном процессоре), по формату команд и т. п. Выполненные команды поставляют результаты в поток данных, и он управляет выборкой тех команд, которые могут выполниться. Если в поток данных ввести результаты работы команд условного перехода в виде значений некоторых управляющих переменных, а сами команды кроме полей операндов снабдить еще одним полем — полем условия, то полученная система будет конкурентноспособна с другими системами. Единственно необходимым условием этого является использование ассоциативного ЗУ, поскольку из принципа управления потоком данных следует, что команды из памяти выбираются не по адресу команды, а по некоторым признакам, связанным с командой (готовность данных, выполнение условия перехода).

МКМД

В системах этого типа взаимодействие между потоками может осуществляться на разных уровнях. На самом низком уровне взаимодействия каждый процессор обрабатывает независимое от остальных задание. Память также индивидуальна. Взаимодействие между процессорами происходит на уровне общих файлов.

На следующем уровне взаимодействия каждый процессор обрабатывает часть общего задания (ветвь). Память каждой ветви индивидуальна. Ветви взаимодействуют через общую память, обмениваясь через нее сообщениями. Наличие общей операционной системы необязательно. Этот уровень характерен для многомашинных систем.

На более тесном уровне взаимодействия имеются общие операционная система, память и устройства ввода-вывода. Процессоры универсальны. Операционная система распределяет задания между процессорами. На следующий уровень можно поместить систему, в которой процессоры специализированы на выполнение задач определенного класса и имеются

все ресурсы предыдущего уровня. Эти системы называются системами, основанными на принципе «разделения труда».

Самое тесное взаимодействие обеспечивают так называемые живучие системы, которые функционируют при выходе из строя части ресурсов, реконфигурируя оставшуюся часть для продолжения решения задачи. Такие системы иногда называют системами с постепенной деградацией.

§ 12. Мультимикропроцессорные вычислительные системы

Ограничиваясь в рамках введения знакомством с основными принципами организации параллелизма, необходимо отметить появление нового строительного блока для МВС — микропроцессора — и его влияние на структуру систем. Основное достоинство микропроцессоров по сравнению с другими интегральными схемами заключается в том, что выполняемые ими функции определяются программой, находящейся в ЗУ. Это достоинство в сочетании с низкой стоимостью, малыми размерами и потребляемой мощностью является основной предпосылкой к массовому использованию микропроцессоров при разработке различного рода вычислительных средств. Технологичность и экономичность логических структур микропроцессоров в значительной мере определяется степенью их однородности и числом необходимых связей с внешней средой. При этом ограничения на число внешних связей являются настолько существенными, что уже в настоящее время очевидно противоречие между скоростью обработки информации внутри микропроцессора и скоростью обмена информацией с внешней средой.

Развитие интегральной технологии приводит к тому, что стоимость центральной части (ядра) микро-ЭВМ — микропроцессора уменьшается. При этом использование в системе нескольких микропроцессоров вместо одного будет незначительно увеличивать общую стоимость системы, которая, в основном, определяется суммарной стоимостью внешних устройств, средств сопряжения, программного обеспечения. Эффективность многопроцессорных систем на базе микро-ЭВМ и микропроцессоров в значительной мере определяется топологией и способом работы сети, предназначенной для соединения различных блоков системы между собой. Характеристики этой сети (интерфейса) в конечном счете определяют характеристики вычислительной системы в целом.

Наиболее экономичной системной организацией является объединение функциональных блоков на основе общей шины. В данной организации все функциональные блоки физически подсоединены к общей шине, по которой происходит обмен информацией между ними на основе методов разделения времени и мультиплексирования.

Общая шина может быть активной и пассивной. Активная шина управляется специальным аппаратным оборудованием, в частности, отдельным микропроцессором. Управление пассивной шиной осуществляет любая из микро-ЭВМ системы.

Использование общей шины создает широкие возможности для наращивания вычислительной мощности, повышения надежности и гибкости системы. Но общая шина является ресурсом критическим, начинающим с некоторого момента тормозить увеличение производительности при подключении дополнительных блоков, поскольку для всех потоков информации выделяется один путь. Надежность систем на основе общей шины также не может быть выше надежности элек-

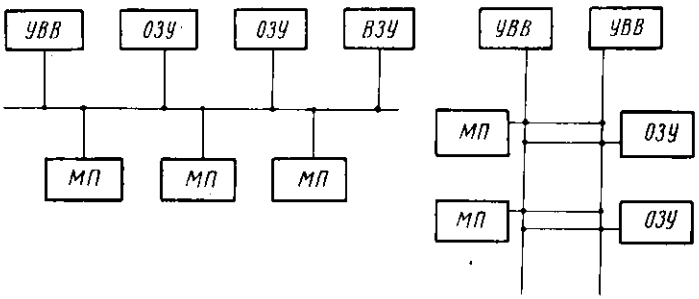


Рис. 40

тронных устройств, управляющих общей шиной. Полностью пассивная шина дает существенное увеличение надежности, но уменьшает производительность.

Для увеличения производительности системы может применяться многошинная организация, содержащая две или более общих шин.

Дальнейшим развитием данной системной организации являются системы с перекрестной коммутацией (рис. 40). Достоинство перекрестной коммутации заключается в том, что любой модуль памяти может быть подключен к любому процессору или устройству ввода-вывода. При этом коммутационная матрица физически связывает два блока на весь период времени, требуемый для обмена информацией. Системы с перекрестной коммутацией уступают по гибкости системам с временным разделением общей шины. Однако наращивание вычислительной мощности ограничивается лишь возможностями коммутатора. Коммутационная матрица позволяет установить несколько путей передачи, по которым информация может передаваться параллельно и независимо друг от друга. Надежность и стоимость системы в значительной мере определяются надежностью и стоимостью коммутатора.

СПИСОК ЛИТЕРАТУРЫ

Ершов А. Г. Введение в теоретическое программирование. — М.: Наука, 1977, 288 с.

Халилов А. И. Алгоритмический язык для описания параллельных процессов. — В сб.: Автоматизация программирования. — Киев: ИК АН УССР, Вып. 3, 1968.

Котов В. Е. Введение в теорию схем программирования. — Новосибирск: Наука, 1978, 256 с.

Цикритзис Д., Бернстайн Ф. Операционные системы. — М.: Мир, 1977, 336 с.

Per. Brinch Hansen. The Programming Language Concurrent Pascal .—IEE Trans. on software engineering, vol. se-1, No 2, June 1975, pp. 199—206.

Rumbaugh J. A data flow multiprocessor.—IEE Trans. on computer selence, No 2, 1979, pp. 138—145.

Котов В. Е. Теория параллельного программирования. — Кибернетика, 1974, № 1, 2.

Головкин Б. А. Параллельная обработка информации. Программирование, вычислительные методы, вычислительные системы. — Техническая кибернетика, 1979, № 2.

ОГЛАВЛЕНИЕ