Руденко Ю.М.

Методические указания
Для выполнения лабораторных работ по курсу
“Вычислительные системы”

2012

Введение

Параллельные вычислительные системы (ВС) являются одними из самых перспективных направлений увеличения производительности вычислительных средств. При решении задач распараллеливания существует два подхода:

1. Имеется параллельная система, для которой необходимо подготовить план и схему решения поставленной задачи, т.е. ответить на следующие вопросы о том, в какой последовательности будут выполняться программные модули, на каких процессорах, как происходит обмен данными между процессорами, каким образом минимизировать время выполнения поставленной задачи.
2. Имеется класс задач, для решения которых необходимо спроектировать параллельную вычислительную систему, минимизирующую время решения поставленной задачи, при минимальных затратах на её проектирование.

При создании параллельных вычислительных систем учитываются различные аспекты их эксплуатации, такие как множественность решаемых задач, частоту их решения, требования к времени решения и т.д., что приводит к различным структурным схемам построения таких систем. Перечислим их в порядке возрастания сложности:

- однородные многомашинные вычислительные комплексы (ОМВК), которые представляют собой сеть однотипных ЭВМ;
- неоднородные многомашинные вычислительные комплексы (НМВК), которые представляют собой сеть разнотипных ЭВМ;
- однородные многопроцессорные вычислительные системы (ОМВС), которые представляют собой ЭВМ с однотипными процессорами и общим полем оперативной памяти или без него;
- неоднородные многопроцессорные вычислительные системы (НМВС), которые представляют собой системы с разнотипными процессорами и общим полем оперативной памяти или без него.

В лабораторном практикуме ставится задача познакомить студентов с основными алгоритмами, используемыми при проектировании параллельных вычислительных комплексов и систем. В частности, в первой лабораторной работе рассматриваются коммутационные структуры типа циркулянт, обобщенных гиперкубов и n-мерных торов. Вторая лабораторная работа посвящена преобразованию последовательных алгоритмов в параллельные. Третья и четвёртая – получению всех необходимых для решения задач распараллеливания матриц, таких как матрица следования, матриц логической

несовместимости и независимости и т. д. В пятой лабораторной работе определяются множества взаимно независимых операторов. В шестой – определяются ранние и поздние сроки выполнения операторов, а также оценка требуемого количества процессоров и времени решения поставленной задачи на вычислительной системе. В седьмой лабораторной работе изучаются методы подключения к ВС типа кластер из локальной сети кафедры с прогоном двух простых примеров, В восьмой – решается на кластере программа средней сложности. Все основные определения и большинство алгоритмов имеются в литературе [1, 2, 5]. Для знакомства с реально работающими вычислительными системами можно обратиться к литературе [5].

При выполнении лабораторных работ (1–6) рекомендуется использовать среду программирования Delphi. Это позволит избежать многих проблем при реализации интерфейса взаимодействия с пользователем. В частности, алгоритмы и графы удобнее отображать с помощью компонента TImage. Используя TImage.Canvas, можно непосредственно «рисовать» на этом элементе граф, а при помощи обработчиков событий onMouseClick, onMouseOver и т.п. можно запрограммировать возможность интерактивного создания и редактирования графа на «холсте». При реализации программы в средах, подобных Delphi, наилучшим способом удобного отображения матриц со скроллингом является компонент TStringGrid. Используя свойство TStringGrid.Cells[y,x], можно записывать различные значения в ячейки матрицы. Для придания матрице «правильного» вида необходимо подкорректировать значения TStringGrid.DefaultColWidth и TStringGrid.DefaultRowHeight. При выполнении седьмой, восьмой работы следует внимательно изучить операционную систему LENUX соответствующей модификации и библиотеку MPI – стандарта.

Порядок выполнения и защиты лабораторных работ

Для проведения лабораторной работы в классе ЭВМ студент должен:

- ознакомиться с теоретическими сведениями и алгоритмами, связанными с параллельными системами,
- изучить среду программирования, используемую для решения задач построения параллельных систем,
- получить свой вариант задания к лабораторной работе у преподавателя, ответственного за проведение лабораторных работ.

Во время выполнения лабораторной работы на ЭВМ необходимо:

- в соответствии с предлагаемым алгоритмом написать и отладить программу,

- для этой программы подготовить 2 – 3 тестовых примера, иллюстрирующих работу программы,
- обеспечить режим пошагового выполнения алгоритмов с выдачей на дисплей получаемых промежуточных результатов.
- во всех случаях, когда результатом работы программы являются графы или временные диаграммы, необходимо обеспечить их выдачу на дисплей, используя графический метод доступа.

Самостоятельная подготовка студентов к каждой лабораторной работе требует около 4 академических часов. Каждая работа выполняется в учебном классе ЭВМ университета в присутствии преподавателя в течение 4-х академических часов. После выполнения очередной лабораторной работы необходимо получить у преподавателя отметку об её выполнении. Отчет в соответствии с установленной формой представляется на очередном занятии. Отсутствие отчета может служить причиной недопуска студента к следующей лабораторной работе. Залогом успешного выполнения лабораторных работ является самостоятельная подготовка студента. С целью облегчения этой подготовки в описании каждой работы имеется теоретическое введение, в котором приводятся основные понятия, используемые в данной лабораторной работе, а также описаны требуемые алгоритмы. Имеются контрольные вопросы, с помощью которых можно оценить степень готовности студента к выполнению лабораторной работы. Работа считается выполненной и зачтённой, если она была защищена. Защита лабораторной работы заключается в демонстрации работы программы на ЭВМ, ответах на вопросы, связанные с её теоретическими аспектами и получаемыми в ней численными характеристиками.

1. Лабораторная работа №1. Определение параметров n-мерных коммутационных структур ВС типа гиперкуб, тор и циркулянт

Цель работы – ознакомление с коммутационными структурами типа циркулянтных, методов оценок их параметров и вычислений таблиц значений этих параметров для различных конфигураций этих циркулянт. Получение таблиц значений диаметра и среднего диаметра для КС типа гиперкуб, тор и циркулянт при определенных наборах числа вершин в КС. Построение графиков зависимостей диаметра и среднего диаметра от количества вершин в коммутационной структуре.

Теоретическая часть

В настоящее время в индустрии ВС получили широкое распространение коммутационные структуры (КС) типа циркулянтных. Эти структуры традиционно представляются D_n-графами.

Определение 1.1. *Циркулянтой называется D_n-граф, представляемый в виде множества $\{N, q_1, \ldots, q_n\}$, где N – число вершин в графе, вершины нумеруются от 0 до N-1, $q_1, \ldots, q_n$ – множество образующих чисел таких, что $0 < q_1 < \ldots < q_n < \frac{N+1}{2}$, а для чисел $N, q_1, \ldots, q_n$ наибольший общий делитель, равен 1, n – число образующих чисел. Вершина i соединяется ребрами с вершинами $(i \pm q_1)(\mathrm{mod}\ N), \ldots, (i \pm q_n)(\mathrm{mod}\ N)$.*

Определение 1.2. *Если граф коммутационной структуры имеет равные степени вершин [1], то КС называется симметричной, и несимметричной – в противном случае.*

Циркулянта относится к классу симметричных КС. Пример циркулянты, представленной в виде двумерной матрицы или хордового кольца [3] показан на рисунке 1.1 для {7,1,3}.

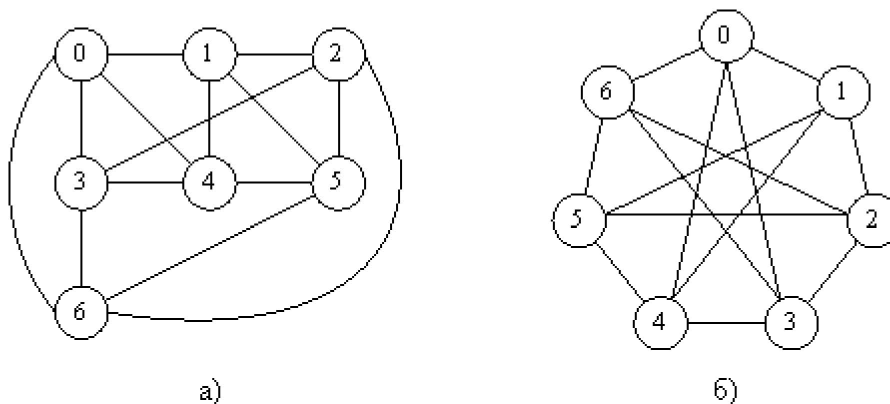


Рисунок 1.1. Пример циркулянты {7,1,3}, изображенной в виде двумерной матрицы (а) и хордового кольца (б)

Определение 1.3. *КС типа n-мерный двоичный гиперкуб описывается следующими соотношениями:*

- *вершины имеют номера* $N_i = 2^p$ *, p=0,1,..., n-1, где n – размерность гиперкуба;*
- *каждая вершина* V_i *задана двоичным числом* $q(V_i) = p_0^{V_i} p_1^{V_i} \dots p_{n-1}^{V_i}$*;*
- *между вершинами* V_i *и* V_j *проводится ребро, если их двоичные номера* $q(V_i)$ *и* $q(V_j)$ *различаются только одним разрядом.* На рисунке 1.2 представлены булевы кубы размерностей 2, 3, 4.

Определение 1.4. *Обобщенный гиперкуб размерности n – это КС, которая удовлетворяет следующим требованиям:*

- *по каждой координате k, k=1,..., n откладываются точки (вершины), с номерами 0,1,...,* N_k*-1, где* N_k *– размерность куба по координате k;*

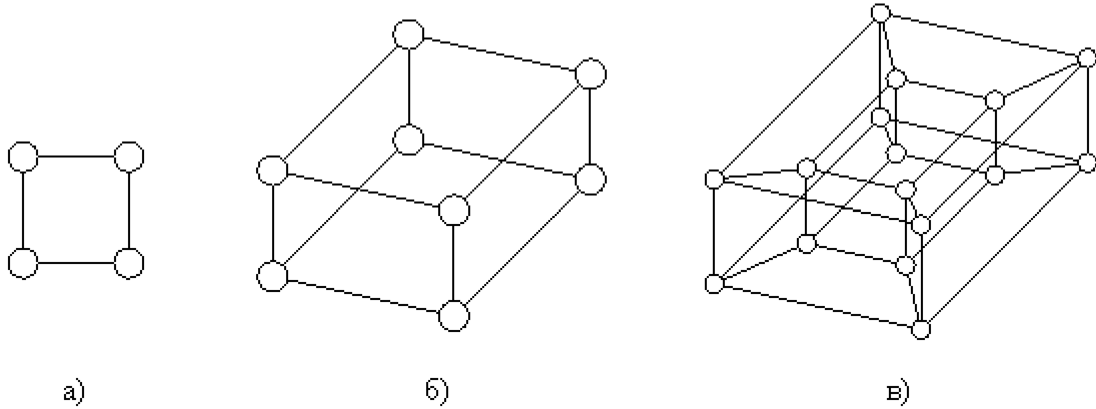


Рисунок 1.2 Пример представления схем булевых кубов размерности 2 (а), 3 (б), 4 (в)

- *множество вершин графа КС задается декартовым произведением* $[0,1,\dots,N_1-1]\times[0,1,\dots,N_2-1]\times\dots\times[0,1,\dots,N_n-1]$*;*
- *две вершины соединяются ребром, если декартовы произведения отличаются друг от друга для рассматриваемой точки и текущей на 1.*

Для 3-х мерного куба 4×3×2 образуются точки по координатам (0,1,2,3), (0,1,2), (0,1) и соответствующие декартовы произведения:

(0,0,0), (0,0,1), (0,1,0), (0,1,1), (0,2,0), (0,2,1),

(1,0,0), (1,0,1), (1,1,0), (1,1,1), (1,2,0), (1,2,1),

(2,0,0), (2,0,1), (2,1,0), (2,1,1), (2,2,0), (2,2,1),

(3,0,0), (3,0,1), (3,1,0), (3,1,1), (3,2,0), (3,2,1).

Ребра проводятся между вершинами (0,0,0) и (0,0,1), (0,1,0), (1,0,0). Вершина (0,0,1) соединяется с вершинами (1,0,1), (0,1,1), (0,0,2) и т.д.

Пример представления этой структуры показан на рисунке 1.3.

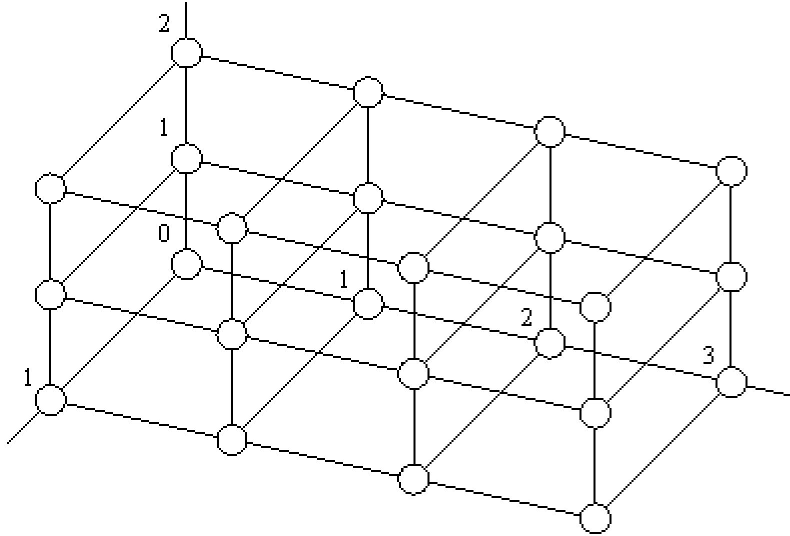


Рисунок 1.3 Схема представления обобщенного 3-х мерного гиперкуба 4х3х2

На основе кубических структур КС введены торы (для одномерного случая – кольцевые структуры).

Определение 1.5. *Структура вычислительной сети типа «двумерный тор» описывается графом* $GS = (M, S^*)$, *где* M *– множество вычислителей,* $M = \{0,\dots,N-1\}$, *$N \geq 6$, а* S^* *– состоит из множества рёбер* s_{kj}, $k \in \{0,1,\dots,L-1\}$ *– множество столбцов,* $j \in \{0,1,\dots,Y-1\}$ *– множество строк и* $L \cdot Y = N$. *Ребро проводится между вершинами, определяемыми декартовым произведением* $[j] \times [k]$. *Две вершины соединяются ребром, если их декартовы произведения отличаются друг от друга на 1 по любой координате или на L - 1 по координате k или на Y - 1 по координате j соответственно.*

Примечание: К обобщённому двумерному тору относятся также и КС, содержащие не все рёбра, отстоящие на расстоянии *Y* - 1 по координате *j* или на *L* - 1 по координате *k*.

Другими словами можно сказать, что двумерный тор это КС типа 2D решётка, противоположные грани которой соединены, обеспечивая обмен данными между первым и последним элементами строки/столбца.

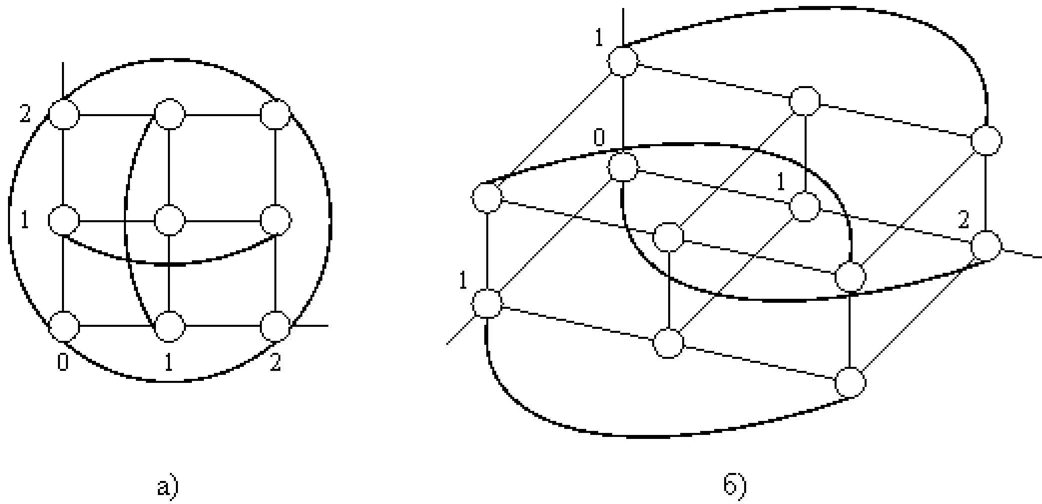


Рисунок 1.4. КС типа двумерный тор (а) и трёхмерный тор (б)

Так, например, из двумерной решётки, показанной на рисунке 1.4, а тор получается введением дополнительных связей, обозначенных на данном рисунке жирными линиями. Степень вершины равна 4. На рисунке 1.4, б показано построение тора со степенью вершины равной 4 из обобщённого 3D-куба (3x2x2).

Одним из важных параметров коммутационных структур является диаметр d и средний диаметр КС, определяющие временные задержки при обмене информацией между процессорами в КС.

Определение 1.3. *Диаметр d – это максимальное расстояние, определяемое как*

$$d = \max_{i,j} \{d_{ij}\}; \quad i, j \in \{0, ..., N-1\},$$

где d_{ij} – расстояние между вершинами i, j рассматриваемой КС.

Расстояние d_{ij} есть минимальная длина простой цепи [1] между вершинами i, j, где длина измеряется в количестве ребер между вершинами i, j. Например, на рисунке 1.1а длины между вершинами (0, 5) есть 3 ((0,1),(1,2),(2,5)), 2 ((0,1),(1,5)), 2 ((0,4),(4,5)), 3 ((0,3),(3,4),(4,5)), 3 ((0,3),(3,6),(6,5)), 2 ((0,6),(6,5)) и др., длина которых больше 3-х. Следовательно, расстояние d_{05} равно 2.

Определение 1.4. *Средний диаметр для симметричной КС относительно выделенной вершины d_i определяется как*

$$\overline{d}_i = \frac{\left(\sum_{p_i}^{d} p_i n_{p_i} \right)}{N-1}, \tag{1.1}$$

где p_i – расстояние от текущей вершины до выделенной (i-ой), n_{p_i} – число вершин, находящихся на расстоянии p_i от выделенной.

Формула (1.1) справедлива для определения среднего диаметра относительно любой вершины, принятой в качестве выделенной, для симметричной КС.

Определение 1.5. *Для несимметричной КС средний диаметр определяется как усреднение по всем d_i, вычисленным по формуле (1.1), рассматриваемого графа КС:*

$$\overline{d} = \frac{\sum_{i=0}^{N-1} d_i}{N}.$$

Вопросы для самопроверки

1. Каким образом структура КС влияет на скорость работы ВС?
2. Чем отличается расстояние между вершинами от длины простой цепи?
3. Что такое циркулянта и каким образом она определяется?
4. Дайте определение диаметра КС и среднего диаметра.
5. Чем отличается изображение циркулянты в виде двумерной матрицы от хордового кольца?
6. Дайте аналитическое описание структуры типа двоичный n-мерный гиперкуб.
7. Дайте аналитическое описание двумерного тора.
8. Какие преимущества имеют КС типа n-мерные двоичные торы перед n-мерными двоичными кубами?
9. Дайте определение декартового произведения над множествами.
10. Дайте аналитическое описание КС типа обобщённый n-мерный гиперкуб.

Задание на лабораторную работу

1. Для заданных КС с помощью программы, разработанной в данной лабораторной работе, вычислить таблицы значений диаметра и среднего диаметра.

Примечание: при разработке программ целесообразно предусмотреть возможность включения в неё новых алгоритмов.

2. Построить графики зависимостей диаметров и средних диаметров от числа вершин в графе КС.

Примечание: при разработке программ целесообразно предусмотреть возможность включения в неё новых алгоритмов.

3. Продемонстрировать работающую программу преподавателю и получить отметку о её выполнении.

4. Сохранить копию программы для выполнения последующих лабораторных работ на дискете.
5. Провести анализ полученных зависимостей.
6. Оформить отчет о проделанной работе.

Содержание отчета

1. Цель работы.
2. Ответы на вопросы для самопроверки.
3. Схема обрабатывающего алгоритма и описание его работы.
4. Распечатки экранных форм, полученных в результате работы программы.
5. Анализ полученных результатов.

2. Лабораторная работа №2. Преобразование последовательного алгоритма в параллельный

Цель работы – ознакомление с принципами преобразования последовательного алгоритма в параллельный. Составление программы этого преобразования для соответствующего варианта задания.

Теоретическая часть

При организации вычислений на ВС на первый план выходит проблема создания параллельных алгоритмов.

Определение 2.1. *Параллельный алгоритм [1] – это описание процесса обработки информации, ориентированного на реализацию его с помощью вычислительных систем.*

Определение 2.2. *Представление параллельного алгоритма на языке программирования, доступном данной ВС, называется параллельной программой.*

Приведение схем алгоритмов к виду, удобному для организации параллельных вычислений.

При создании параллельных программ будем использовать схемы программ в соответствии с ГОСТ 19.003-80 ЕСПД, ГОСТ 19.701-90 ЕСПД, хотя следует отметить, что создание параллельных процессов в решаемых задачах и отображение их в таких схемах – достаточно трудоёмкая задача. В связи с этим, предложена следующая процедура создания параллельных схем алгоритмов или программ.

Вначале создаётся схема алгоритма, как это делалось для традиционных вычислительных средств без учёта параллельных вычислений. Будем называть их последовательными алгоритмами. Затем с помощью предлагаемого ниже алгоритма на основе анализа зависимости участков процесса по обрабатываемым переменным в вычислительном процессе выделяются параллельные ветви вычислений. Алгоритмы с выделенными параллельными ветвями соответствуют понятию параллельные алгоритмы. Введём несколько ограничений при изображении схем алгоритмов с параллельными ветвями.

1. При параллельном выполнении программ окончание алгоритма зависит от составленного плана решения задачи, поэтому символ «терминатор» конца алгоритма исключим.

2. Не ограничивая общности рассуждений, можно считать, что при изображении параллельных алгоритмов можно ограничиться обозначениями логических выходов операторов типа “IF”, “CASE” в виде “№.n”, где № – номер рассматриваемого логического

оператора, n – номер выхода из логического оператора. Такое обозначение позволяет упорядочить существующие обозначения: «истина», «ложь», «FALSE», «TRUE», «>», «<» ,«<>» и т.д., что создает определённые удобства при дальнейшем анализе схем алгоритмов в виде граф–схем.

3. Традиционное изображение вводимой информации в указанных ГОСТах более или менее приемлемо для ВС с общей памятью и совсем не приемлемо для ВС с разделяемой памятью, так как вводится достаточно искусственная зависимость программных модулей по данным. Эта зависимость существенно сужает возможности распараллеливания решаемой задачи. При рассмотрении ВС с разделяемой памятью ввод-вывод информации включается в процесс обмена информацией между процессорами и таким образом учитывается в планировании вычислительного процесса. В связи с этим при рассмотрении ВС с общим полем памяти считается, что вся исходная информация введена в поле общей памяти и на схеме не показывается. Аналогично решаем проблему вывода. Выводимая информация также находится в поле общей памяти. Отсутствие символа вывода (например, параллелограмма) можно объяснить тем, что преобразование и вывод информации не включается в план решения параллельных задач. В связи с этим при преобразовании исходного алгоритма в параллельный опускаются символы ввода-вывода информации.

Сущность алгоритма преобразования схемы последовательного алгоритма в схему параллельного алгоритма заключается в следующем. Разобьем последовательный алгоритм на линейные участки, заключенные между логическими операторами. Каждый логический оператор порождает не менее двух линейных участков. Линейный участок, образованный входом в алгоритм и логическим оператором назовем начальным. Начальный участок может содержать только один оператор. Следующий за начальным участок начинается и заканчивается логическим оператором, т. е., если участок U_i состоит из множества операторов $\{L^i_j, c^i_1, \ldots, c^i_{n_i}, L_{j+1}\}$, то следующий участок $U_{i+1} = \{L^{i+1}_{j+1}, c^{i+1}_1, \ldots c^{i+1}_{n_{i+1}}, L^{i+1}_{j+2}\}$ и т. д., где $L^i_j, L^i_{j+1}, L^{i+1}_{j+1}, L^{i+1}_{j+2}$ – логические операторы, причем $L^i_{j+1} = L^{i+1}_{j+1}$, а $c^i_1, \ldots, c^i_{n_i}, c^{i+1}_1, \ldots, c^{i+1}_{n_{i+1}}$ – некоторые операторы. Таким образом, последний логический оператор L_{j+1} участка U_i является первым оператором для участка U_{i+1}. На каждом участке операторы перенумерованы: 1, 2, …, u_k. Последние участки – это линейные участки, не имеющие в качестве последнего оператора логический оператор. Пусть в результате такого разбиения образовалось N участков $k = 1, \ldots, N$, в каждом из которых оказалось u_k операторов.

Назовём связи, входящие в вычислительный или логический блоки, входными; выходящие из этих блоков – выходными. Будем полагать, что в каждый блок может входить и выходить несколько связей. Для упрощения анализа зависимости рассматриваемого программного модуля от предыдущих в анализируемом алгоритме принята следующая схема: анализируемый алгоритм представляет собой последовательность программных модулей (процедур и/или функций). Обмен данными между ними происходит только через параметры, указанные в списке при вызове модулей. Результаты работы модуля передаются через параметры, формируемые как <имя параметра> ::= <префикс> <имя модуля>. Так, например, на рисунке 2.1 модуль с именем AB передает результаты своих вычислений через параметр *SAB* и т.д. Модуль DE использует данные, формируемые модулем AB, что означает, что модуль DE зависит по данным от модуля AB и т.д. Нетрудно заметить, что предлагаемое упрощение не является принципиальным и в случае необходимости легко можно учесть зависимости программных модулей по данным, осуществляемых с помощью понятий глобальных переменных различных уровней.

Алгоритм 2.1. Преобразование схемы последовательного алгоритма в параллельную.

1. *Вычислим* $k:=1$, $M_V:= \varnothing$ *– множество входов в алгоритм,* $Flag:=TRUE$, $u_k^* = u_k$.
2. *Если* $u_k^* \geq 2$, *то выполнить шаг 3, иначе – шаг 13.*
3. *Вычислим* $u_k = 1$.
4. *Вычислим* $v:=u_k$, $u_k:=u_k+1$.
5. *Если* $u_k \leq u_k^*$, *то выполнить шаг 6, иначе – шаг 16.*
6. *Если* u_k *зависит от* v, *то выполнить шаг 7, иначе – шаг 11.*
7. *Проводим связь из блока* v *в блок* u_k.
8. *Вычислим* $Flag:=False$.
9. *Вычислим* $v:=v-1$.
10. *Если* $v < 1$, *то переходим к шагу 4, иначе – шаг 6.*
11. *Если* $Flag:=TRUE$, *то переходим к шагу 12, иначе – шаг 9.*
12. *Вычислим* $M_V:= M_V \cup \{u_k\}$ *и переходим к шагу 9,*
13. *Если* $u_k^*=2$, *то переходим к шагу 14, иначе – шаг 15.*
14. *Вычислим* $M_V:= M_V +\{1\}$.
15. *Все ли блоки имеют выходные связи? Если – «да», то выполнить шаг 16, иначе – шаг 17.*
16. *Вычислим* $k:=k+1$ *и затем выполним шаг 18.*
17. *Проводим связи из блоков* u_k *в блок* u_k^* *и переходим к шагу 16.*

18. *Если $k \le N$, то переходим к шагу 20, иначе – шаг 19.*
19. *Конец алгоритма.*
20. *Очередной участок является внутренним? Если – «да», то выполнить шаг 21, иначе – шаг 34.*
21. *Если $u_k^*=3$, то переходим к шагу 22, иначе – шаг 23.*
22. *Проводим связи из блоков $u_k=1$ в блок $u_k=2$ и $u_k=2$ в блок $u_k^*=3$ и переходим к шагу 16.*
23. *Вычислим $u_k:=2$.*
24. *Вычислим $v:=u_k$, $u_k:=u_k+1$.*
25. *Если $u_k < u_k^*$, то выполнить шаг 26, иначе – шаг 30.*
26. *Если u_k зависит от v, то выполнить шаг 27, иначе – шаг 28.*
27. *Проводим связь из блока v в блок u_k.*
28. *Вычислим $v:=v-1$ и затем выполним шаг 29.*
29. *Если $v < 2$, то переходим к шагу 24, иначе – шаг 26.*
30. *Все ли блоки имеют входные связи? Если – «да», то выполнить шаг 32, иначе – шаг 35.*
31. *Проводим связи из блока $u_k=1$ в блоки, не имеющих входных связей.*
32. *Все ли блоки имеют выходные связи? Если – «да», то выполнить шаг 16, иначе – шаг 33.*
33. *Провести связи из блоков, не имеющих выходных связей, в блок u_k^*, и затем выполним шаг 16.*
34. *Если $u_k^*=2$, то переходим к шагу 35, иначе – шаг 36.*
35. *Проводим связи из блоков $u_k=1$ в блок u_k^* и переходим к шагу 16.*
36. *Вычислим $u_k:=2$.*
37. *Вычислим $v:=u_k$, $u_k:=u_k+1$.*
38. *Если $u_k < u_k^*$, то выполнить шаг 39, иначе – шаг 43.*
39. *Если u_k зависит от v, то выполнить шаг 40, иначе – шаг 41.*
40. *Проводим связь из блока v в блок u_k и переходим к шагу 41.*
41. *Вычислим $v:=v-1$.*
42. *Если $v < 2$, то переходим к шагу 37, иначе – шаг 39.*
43. *Все ли блоки имеют входные связи? Если – «да», то выполнить шаг 16, иначе – шаг 44.*
44. *Проводим связи из блока $u_k=1$ в блоки, не имеющих входных связей, и переходим к шагу 16.*

Результат работы данного алгоритма показан на рисунке 2.2 (исходным является алгоритм, изображённый на рисунке 2.1).

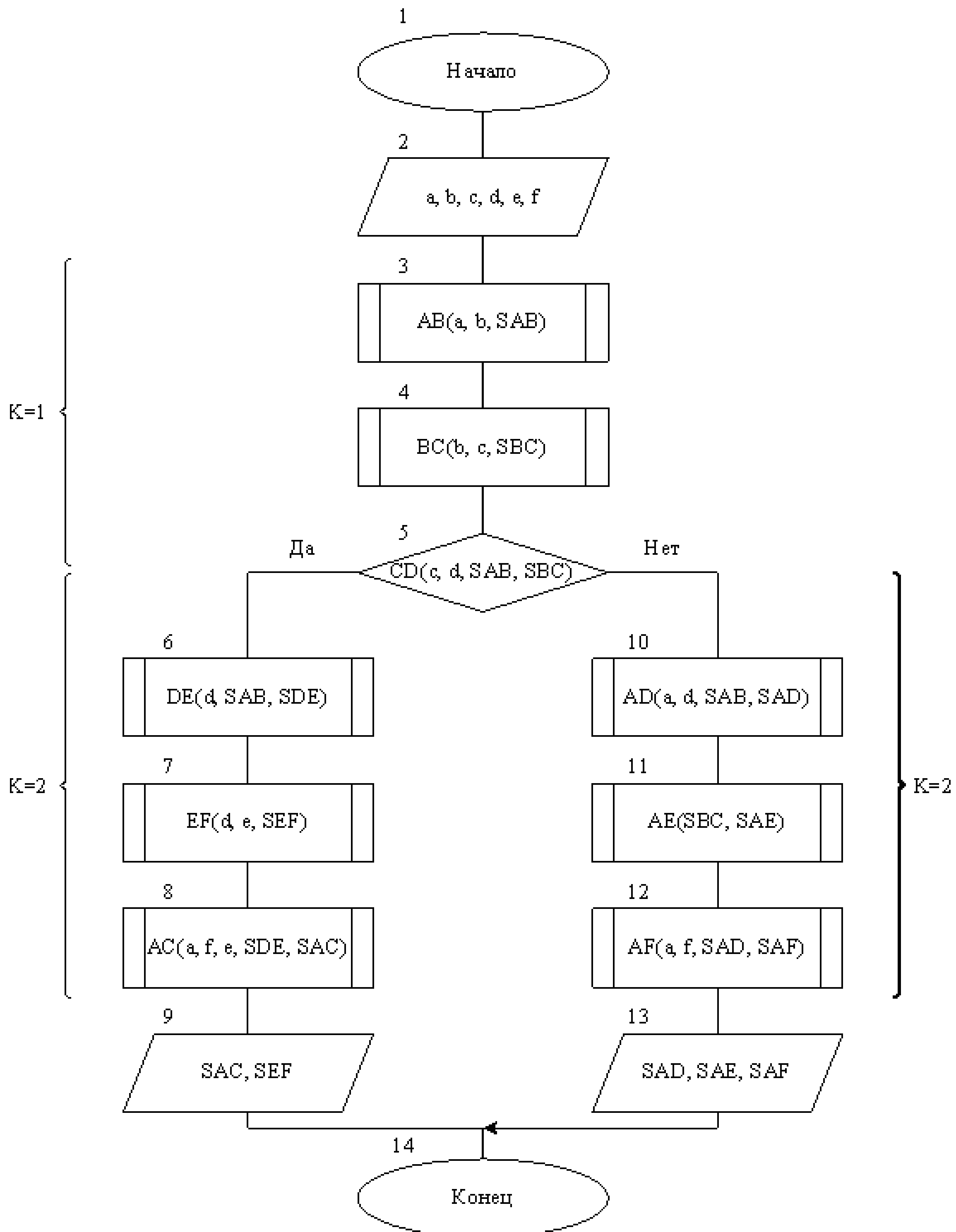


Рисунок 2.1. Классическая схема алгоритма

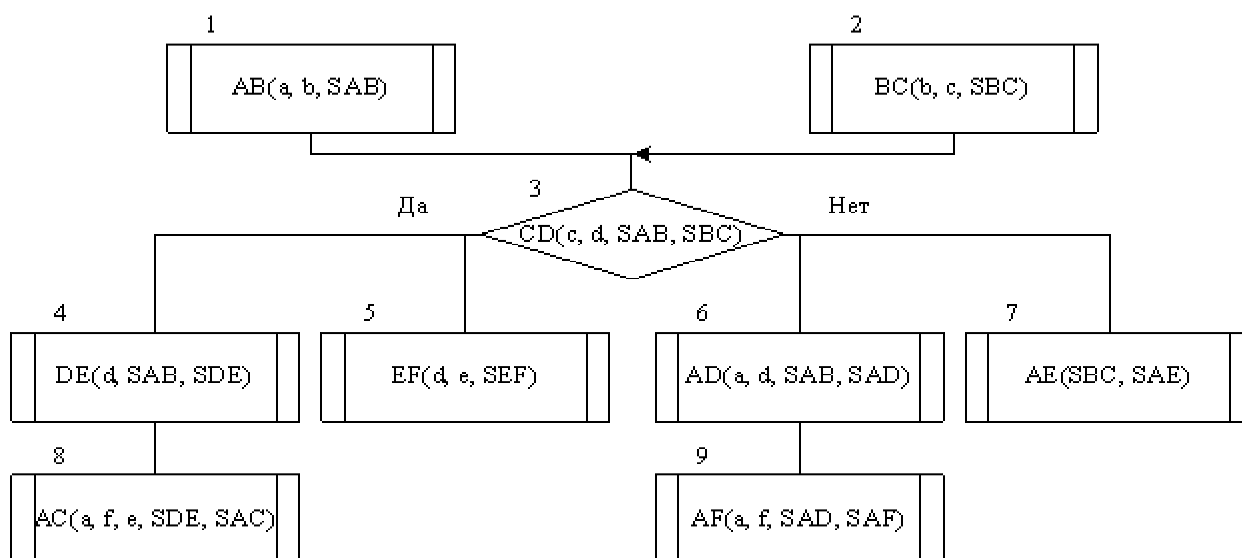


Рисунок 2.2. Схема модифицированного алгоритма

Вопросы для самопроверки

1. Чем отличается последовательный алгоритм от параллельного?
2. Почему символы ввода-вывода данных, а также «терминатор» исключены из схемы параллельного алгоритма?
3. Какое правило положено в основу определения зависимости блоков алгоритма по переменным?
4. Какие ещё зависимости по переменным между блоками алгоритма следует учесть при реализации алгоритма 2.1?
5. Как образуются дополнительные входы в алгоритм?
6. Как производится разбиение алгоритма на ветви?

Задание на лабораторную работу

1. Создать и отладить программу для преобразования последовательного алгоритма в параллельный и проиллюстрировать её работоспособность на заданном варианте алгоритма, предусмотрев возможность редактирования информации на один или два шага назад («откат»). После написания программы необходимо ввести в неё последовательный алгоритм, выданный преподавателем, предпочтительно в виде, показанном на рисунке 1. Результирующий алгоритм представить также в соответствии с вышеуказанными ГОСТами. Сохранить программу для выполнения последующих лабораторных работ.

Примечание: при разработке программ целесообразно предусмотреть возможность включения в неё новых алгоритмов.

2. Продемонстрировать работающую программу преподавателю и получить отметку о её выполнении.

3. Сохранить копию программы для выполнения последующих лабораторных работ на дискете.

4. Оформить отчет о проделанной работе.

Содержание отчета

1. Цель работы.
2. Ответы на вопросы для самопроверки.
3. Схемы исходного и полученного алгоритмов, а также схему обрабатывающего алгоритма и его описание.
4. Распечатки экранных форм, полученных в результате работы программы.
5. Анализ полученных результатов.

3. Лабораторная работа №3. Представление алгоритмов в виде граф–схем.

Цель работы – ознакомление с принципами организации параллельных вычислений с помощью информационного (ИГ) или информационно-логического графа (ИЛГ), представляющего алгоритм решения поставленной задачи как показано на рисунках 3.1, 3.2.

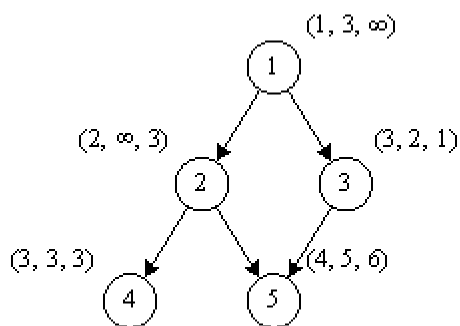


Рисунок 3.1. Информационная граф-схема алгоритма

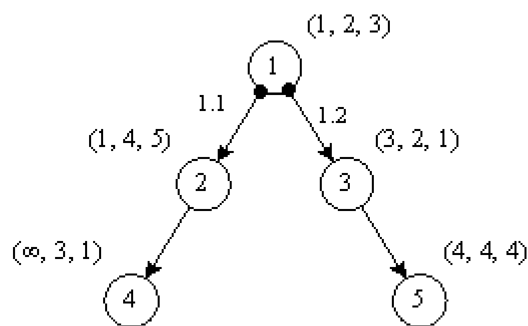


Рисунок 3.2. Информационно-логическая граф-схема алгоритма

По заданной схеме алгоритма, соответствующей варианту задания студента, построить требуемый граф–схему и создать соответствующую программную среду, позволяющую отображать графы на экране дисплея с возможностью редактирования, вычислять матрицы следования, матрицы следования с транзитивными связями и расширенные матрицы следования.

Теоретическая часть

Информационный или информационно-логический граф задается с помощью выражения:

$$G = (X, P, D),$$

где $X = \{i\} = \{1, \ldots, m\}$ – множество вершин графа, соответствующее множеству операторов параллельного алгоритма, $P = \{p_i\}$, $i = 1, \ldots, m$, p_i – множество весов, определяющих время выполнения каждого i-го оператора. В общем случае p_i – вектор. Размерность вектора равна количеству типов процессоров, используемых в неоднородной ВС. Для однородной ВС p_i – скаляр. D – множество дуг графа.

Определение 3.1. *Ориентированный граф, представляющий некоторую схему алгоритма, не содержащую циклов, таким образом, что каждому блоку алгоритма соот-*

ветствует вершина, а связям между блоками – дуги, называется граф-схемой алгоритма.

В общем случае граф–схема алгоритма – это сеть. Как следует из определения 3.1., циклы исключены из граф-схем. Это сделано по нескольким причинам. Во-первых, распараллеливание осуществляется для сложных программ, где каждый блок схемы алгоритмов – это программный модуль, в который всегда можно включить небольшие по времени циклы и тем самым упростить составление граф-схем алгоритмов. Во-вторых, циклы по параметру [1] не поддаются распараллеливанию и их бессмысленно изображать в граф-схеме, а лучше включить в программный модуль. Циклы по счётчику циклов [1] распараллеливается, как будет показано ниже, введением дополнительных вершин в граф – схему. Для дальнейшего рассмотрения сеть можно представить в виде совокупности свёрток и развёрток графа.

Определение 3.2. *Свёрткой k-й вершины граф-схемы называется наличие у k-й вершины граф-схемы n входящих дуг, где n ≥ 1 и является конечным числом.*

Определение 3.3. Р*азвёрткой k-й вершины графа называется наличие у k-й вершины графа n выходящих дуг, где n ≥ 1 и является конечным числом.*

Свёртка и развёртка граф-схемы показаны на рисунке 3.3.

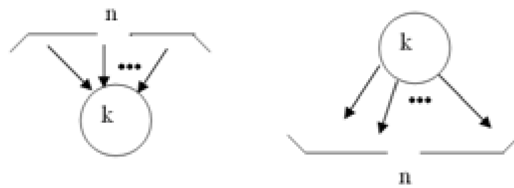


Рисунок 3.3. Свёртка и развёртка граф - схемы

Определение 3.4. Изолированной k-й вершиной граф-схемы называется вершина, у которой отсутствуют входящие и выходящие дуги.

Определение 3.5. Элементарной свёрткой или развёрткой k-й вершины граф-схемы называется наличие у k-й вершины граф-схемы одной входящей или выходящей дуги соответственно.

Вопрос о получении или передаче значений параметров для этих случаев представляет несомненный интерес при создании граф-схем алгоритмов, предназначенных для выполнения на вычислительных системах. Рассмотрим случай свёртки *k-й* вершины граф-схемы. Будем полагать, что каждый вход в *k-ю* вершину рассматривается, как логическая переменная [1], которая принимает значение «истина», если на рассматриваемый

вход приходит информация, полученная на предшествующей вершине, в заданный интервал времени в соответствии с заданными логическими уравнениями пользователя. В противном случае логическая переменная принимает значение «ложь», если логические уравнения определяют значение «ложь». Интервал времени вычисляется соответствующим `способом. Учитывая, что возможны некоторые непредвиденные отклонения, не влияющие существенно на результаты вычислений, при обработке и передаче информации с предшествующей *i-й* вершины в *k-ю*, введём интервал времени $[-\nabla t^*_{ik}, \nabla t^*_{ik}]$. Таким образом, время прихода информации с *i-й* вершины определится как

$$T_{ik} = t_i + t_{ik} + \nabla_{ik}, \quad \nabla_{ik} \in [-\nabla t^*_{ik}, \nabla t^*_{ik}], \qquad (3.1)$$

где t_i – время выполнения программного модуля, представленного *i-й* вершиной, t_{ik} – время передачи информации с *i-й* вершины в *k-ю* вершину, ∇_{ik} - отклонение времени выполнения модуля, представленного *i-й* вершиной, и передачи информации по *k-й* связи от его среднего значения.

Перенумеруем все дуги, входящие в *k-ю* вершиной, получим множество дуг *n*. Тогда образуется множество времён $\{T_{ik}\}$, $i \in n$ и свёртку в вершине *k* можно трактовать как логическую функцию *n* переменных. В качестве примера использования данной концепции рассмотрим две логические функции, реализованные на свёртках или развёртках *k-х* вершин, которые, как будет показано далее, полностью перекрывают возможности, предоставляемые соответствующими ЕСПД по изображению последовательных алгоритмов. Функция «ИСКЛЮЧАЮЩЕЕ ИЛИ» вызывает срабатывание *к-й* вершины при появлении информации на k-й дуге свёртки или развёртки и реализует наиболее быстрый проход k-ой вершины, так как отсутствует ожидание прихода информации на другие дуги рассматриваемой свёртки или развёртки. Кроме того, эта функция обеспечивает наиболее надёжный узел для срабатывания, так как вероятность не прихода информации на рассматриваемую вершину, в общем случае, минимальна. И наоборот, функция «И» реализует наиболее медленный проход *k-й* вершины и наименее надёжный узел для срабатывания, так как в общем случае, должна прийти информация на все *n* дуг свёртки или развёртки. Все другие логические функции занимают промежуточное положение по надёжности и времени срабатывания.

Рассмотрим времена срабатывания для развёртки с *k-й* вершиной для функции «ИСКЛЮЧАЮЩЕЕ ИЛИ» и функции «И». Функция «ИСКЛЮЧАЮЩЕЕ ИЛИ» реализуется за время для случая срабатывания *i-ого* выхода

$$T_{ki}=t_{ki}+t_{ki}^{*}+\nabla_{ki},\quad \nabla_{ki}\in[-\nabla t^{*}_{ki},\ \nabla t^{*}_{ki}], \qquad (3.2)$$

где t_{ki} – время выполнения *k-ого* модуля при формировании *i-ого* выхода, t_{ki}^{*} - время передачи информации, сформированной *k-й* вершины, к *i - й* вершине при формировании *i-ого* выхода, ∇_{ki} – отклонение времени выполнения модуля, представленного *k-й* вершиной, и передаче информации по *i-му* каналу от его среднего значения.

Время срабатывания для функции «И» также определяются по формуле *(3.2),* так как одновременность срабатывания всех выходов не требуется.

Время срабатывания для свёртки с *k-й* вершиной для функции «ИСКЛЮЧАЮЩЕЕ ИЛИ» также определяется по формуле *(3.2),* а для функции «И» для обеспечения одновременного прихода сигнала в *k-ю* вершину время составит

$$T_{ik}^{*} = max\ \{ T_{ik}\},\ i\in n,$$

где T_{ik} – времена, вычисленные по формуле (3.2), n – число входов в *k–ю* развёртку.

Очевидно, что любая граф-схема алгоритма представляет собой некоторую последовательность свёрток – развёрток, образующих композицию из логических функций, зависящих от n переменных, где $n\ \in\{1,\ldots,\Psi\}$, где Ψ – максимальное количество дуг в свёртках и развёртках. Следует отметить особую роль времени t_{ki}^{*}, при формировании граф-схем алгоритмов. При рассмотрении ВС с общей памятью это время, как правило, не учитывается и методы анализа и решения таких алгоритмов значительно проще. Учёт времени t_{ki}^{*} порождает класс ВС с разделяемой памятью, что приводит к усложнению методов анализа и решения таких алгоритмов. Предлагаемая читателю работа, также построена по принципу – «от простого к сложному». В начале представлен материал, связанный с с методами обработки информации на ВС с общей памятью, имея дело с информационными (ИГ) и информационно-логическими (ИЛГ) граф-схемами. Затем исследуем ВС с разделяемой памятью и изучим информационные граф-схемами с разделяемой памятью (ИГР) и информационно-логические граф-схемами с разделяемой памятью (ИЛГП).

Рассмотрим примеры часто используемых конструкций в граф–схемах алгоритмов, которые могут быть реализованы с помощью предлагаемых логических функций. На развёртках могут быть реализованы все типы основных конструкций языков программирования, используемых для создания последовательных программ.

На функции «И» *k-й* развёртки может быть реализован цикл по счётчику циклов. Так, если количество дуг, исходящей из *k-й* вершины, ограничено по некоторым причи-

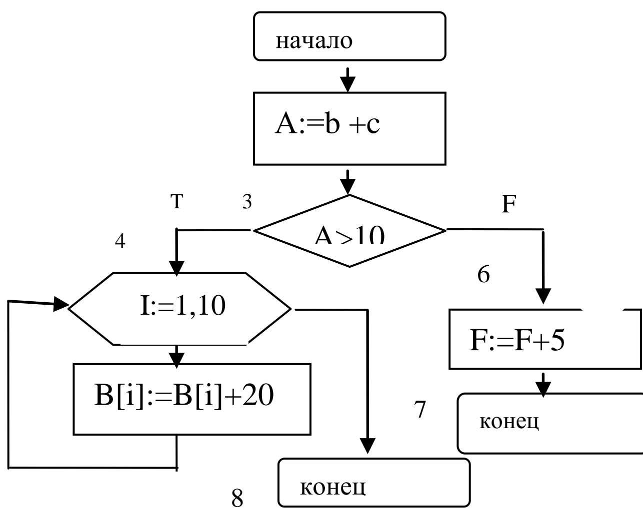
нам есть n, а количество повторений в цикле – F, то количество обращений *k-ого* блока к нижестоящим, связанным с ним блокам определится соотношением $]F/n[$, где $][$ – функция выделения целой части положительного числа, большей или равной этому числу. Время прохождения информации по этому каналу определится по формуле

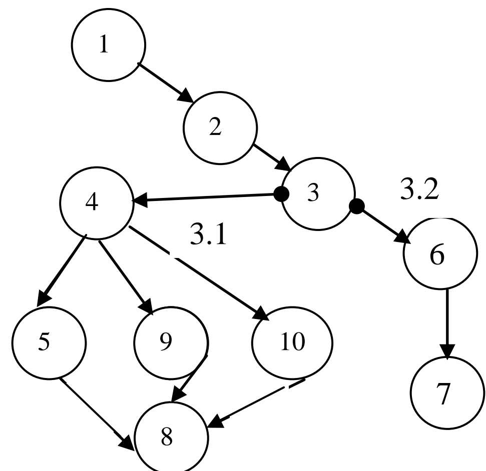
$$T_{ki}^{m} =]F/n[* T_{ki} \qquad (3.3)$$

Время выполнения цикла может быть уменьшено до $T_{ki}^{m} = \mathrm{T_{ki}}$, если увеличить число каналов (число процессоров) до F. Реализация функции «Исключающее ИЛИ» на два или несколько выходов может представлять логическую функцию на два выхода (true, false), три выхода (> 0, < 0, = 0), а также функцию переключения на один из n возможных каналов. Кроме того, задавая ту или иную логическую функцию, можно получить любую комбинацию передающих каналов, ориентированных на использование операторов – переключателей, используемых в различных языках программирования или для каких либо других целей.

Определение 3.6. Дуги, исходящие из вершин, содержащих логические блоки, переключатели каналов, подлежащие распараллеливанию, называются логически зависимыми.

На рисунке 3.4 приведена схема алгоритма, ориентированного на представление его в виде графа-схемы. Для простоты изложения будем полагать, что развёртка содержит две логически зависимые дуги и осуществлена развёртка цикла с помощью введения двух дополнительных вершин под номерами 5,9,10. В общем случае требуется, ввод дополнительных вершин, согласно соотношению (3.3) –]F/n[-1 или F, если используется число процессоров, равное числу итераций в цикле.





а) **б)**

Рисунок 3.4. На рисунке изображена схема алгоритма (а) и соответствующая ей граф-схема (б).

Описание алгоритма, представленного на рисунке 3.4.

1. *Начало алгоритма.*
2. *Вычисляется значение переменной А.*
3. *Проверяется условие А>10.*
4. *Объявляется цикл по переменной i=1, …, 10.*
5. *Вычисляется массив значений: B[i]= B[i]+20,*
6. *Вычисляется значение переменной F:=F+1.*
7. *Конец алгоритма.*
8. *Конец алгоритма.*

На рисунке 3.4, б приведён фрагмент граф-схемы, соответствующий схеме алгоритма, изображённого в соответствии с выше указанными ЕСПД. Условный блок 3 заменён вершиной 3 с исходящими связями, образующими логическую функцию «Исключающее ИЛИ» на два выхода. Связи, выходящие из вершины 3 в граф-схемах будем обозначать в виде стрелок с точками в начале. Для обозначения связей при построении матриц, описывающих граф-схемы, будет использована комбинация символов в виде n.m, где n – номер вершины, из которой выходит дуга, m – порядковый номер связи. Блок цикла по счётчику циклов 4 заменён логической функцией «И» на три входа, образуя функцию (4,5) & (4,9) & (4,10). Цикл 4 на рисунке 3.4, б заканчивается свёрткой по схеме «И». Два выхода из программы 7,8 позволяют использовать результаты вычислений на процессорах, не дожидаясь окончательного завершения одной из ветвей программы, так как они определяют конец параллельной ветви алгоритма. Элементарные свёртки или развёртки вершин 1,2,3,6.7 не имеют идентификатора логических функций, хотя в соответствии с определением алгоритма при его запуске, они должны всегда срабатывать, будем считать, что они реализуют схему «И». Дуги, образующие схему «И» в блок-схеме изображаются в виде стрелок, используемых как для обозначения дуг свёрток и развёрток.

Таким образом, предлагаемая граф-схема может служить удобным средством для изображения параллельного алгоритма.

Для более полного представления о возможностях использования граф-схем приведём изображения основных операторов наиболее часто используемых языков программирования.

На рисунке 3.5, а представлена вершина граф-схемы, с помощью которой реализуется процесс в соответствии с терминологией, используемой в ЕСПД. Рисунок 3.5, б отображает реализацию с помощью функции «Исключающее ИЛИ» логического оператора с двумя выходами. Аналогично, рисунок 3.5, в – реализацию с помощью функции «Исключающее ИЛИ» логического оператора с тремя выходами. На рисунке 3.5, г показана реализация оператора-переключателя с n выходами типа «CASE» языка «Паскаль» [1]. Рисунок 3.5, д отображает реализацию фрагмента программы, в которой начало передачи информации по всем m каналам начинает передаваться одновременно. Используется логическая функция «И»: (13,14) & (13,15) & (13,16). На этой же схеме реализуется цикл по счётчику циклов. Время выполнения зависит от количества итераций в цикле и количества выделенных вычислителей для вычислений цикла ВМ. Время выполнения цикла, определяется согласно формуле (3.3). Идентификаторы процессов в виде чисел записываются внутри окружностей, которыми изображаются вершины графа.

Затем составляется описание процессов:

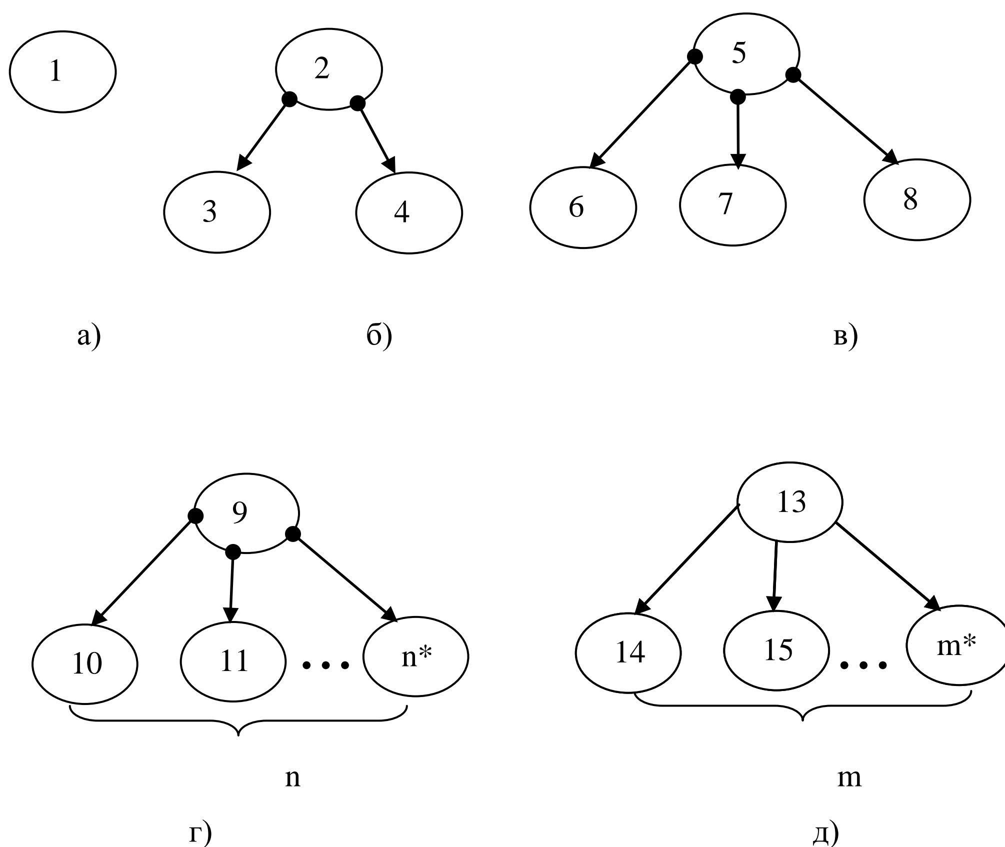


Рисунок 3.5. На рисунке изображены фрагменты граф-схемы алгоритма

Представим в виде граф-схем рассмотренные ранее алгоритмы (см. рисунки 2.1 и 2.2).

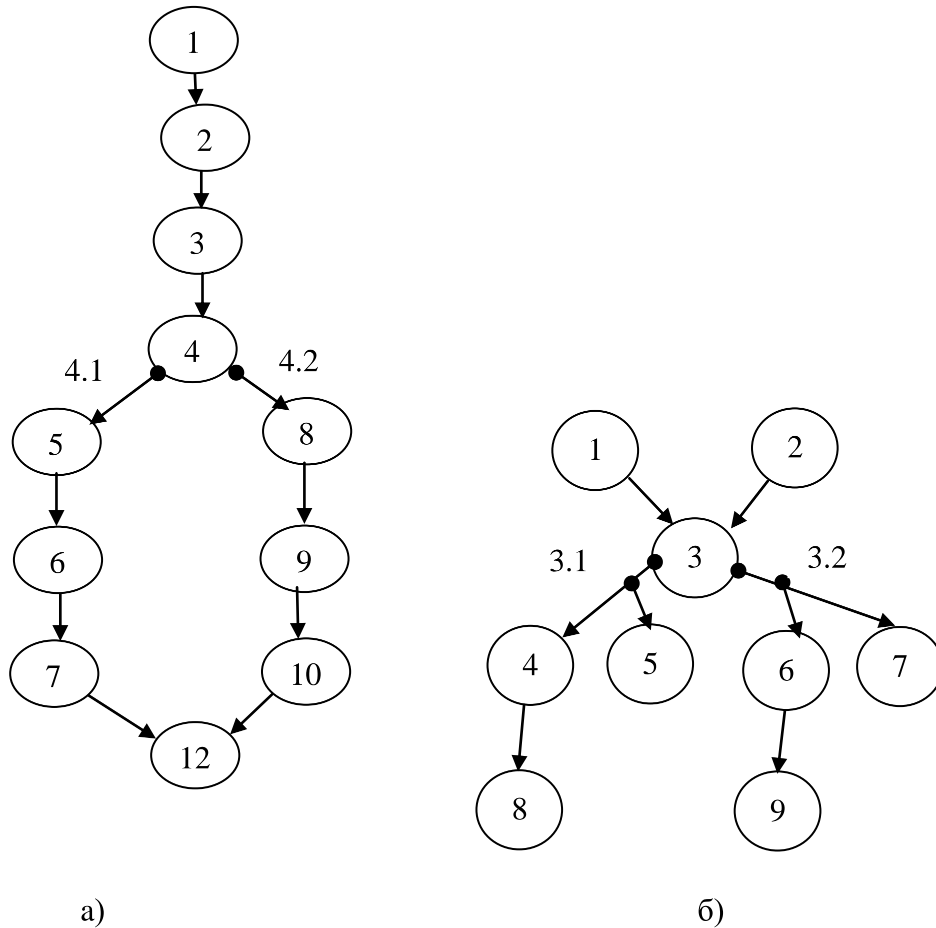


Рисунок 3.6. Граф-схемы алгоритмов, представленных на рисунках 2.1 и 2.2.

Рисунок 3.6, б наглядно показывает, что после обработки схемы алгоритма, представленного на рисунке 3.1, алгоритмом 2.1, модули 1,2, 4 или 5 и 6 или 7 могут выполняться параллельно. Анализируя нумерацию блоков на рисунке 3.6. можно заметить, что она не совпадает. Это связано с двумя причинами. Во-первых, блоки начала и конца алгоритма исключены, во-вторых, при нумерации вершин граф-схемы (б) соблюдалось правило нумерации по ярусам. Это правило, и преимущество, которое даёт его соблюдение, будет рассмотрено ниже.

Использование граф-схем для представления параллельных алгоритмов

Граф- схемы алгоритмов представляются с помощью выражения $G = (X, P, D)$, X – множество вершин граф- схемы, $X \in \{1, ..., m\}$. Множество вершин графа X соответствует множеству операторов параллельного алгоритма. $P = \{1,...,P_m\}$ – множество весов вершин графа. P_i может быть скалярной величиной или вектором, $i \in \{1, ..., m\}$. Если P_i -

скаляр, то рассматривается решение этой задачи на однородной ВС (вычислительная система имеет одинаковые процессоры). Тогда, как правило, P_i – время решения *i-ого* программного модуля. Если $\vec{P}_i$ – вектор, то предполагается решение этой задачи на неоднородной ВС (вычислительная система имеет разные типы процессоров). И, тогда, если система содержит *S* разнотипных процессоров, то вектор $\vec{P}_i = \{ P_{i1}, \dots , P_{is}\}$, где $P_{i1}, \dots , P_{is}$ – набор времен решения *i-ой* процедуры на различных типах процессоров из множества *S*. *D* – множество дуг графов. Дуги бывают трёх типов: $d_i \in D0$, $d_j \in D2$, $d_k \in D3$. $D = D0 \cup D2 \cup D3$. *D0*- множество одиночных дуг графа, соответствующих элементарным свёрткам или развёрткам *k-х* вершин граф-схемы. D2 – множество логических дуг графа, реализующих функции «Исключающее ИЛИ» граф-схемы, D3 – множество дуг графа, реализующих функции «И» граф-схемы,. Обозначим $D1 = D0 \cup D3$.

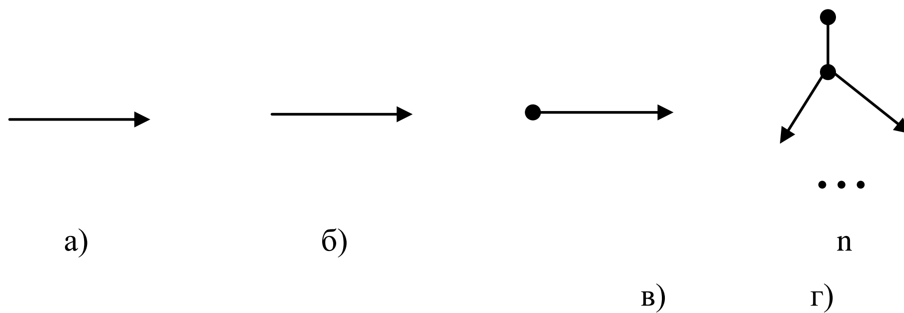


Рисунок 3.7. Типы дуг, используемых в граф-схемах алгоритмов

На рисунке 3.7 показаны типы дуг, используемых граф-схемах алгоритмов. Тип а) формирует множество *D0*, тип б) формирует множество *D3*, тип в) и г) формирует множество D2. Коплексная стрелка г) используется в тех случаях, когда один выход функции «Исключающее ИЛИ» соединяется с несколькими вершинами граф-схемы одновременно. Граф-схемы бывают двух типов согласно определениям 3.7 и 3.8.

Определение 3.7. Граф-схема, содержащая только дуги $d_i \in D1$,называется информационной граф-схемой (ИГ).

Определение 3.8. Граф-схема, содержащая дуги $d_j \in D2$, реализующие функции «Исключающее ИЛИ» или «Исключающее ИЛИ», *«И», называется информационно- логической граф-схемой (ИЛГ).*

Например, информационная граф-схема, представленная на рисунке 3.7а, предназначена для изображения алгоритма, который будет выполняться на неоднородных вычислительных системах, имеющих три типа ВМ. Последовательность чисел, заключён-

ных в круглые скобки, расположенных возле вершин граф-схемы, представляет собой составляющие трёхмерного вектора.

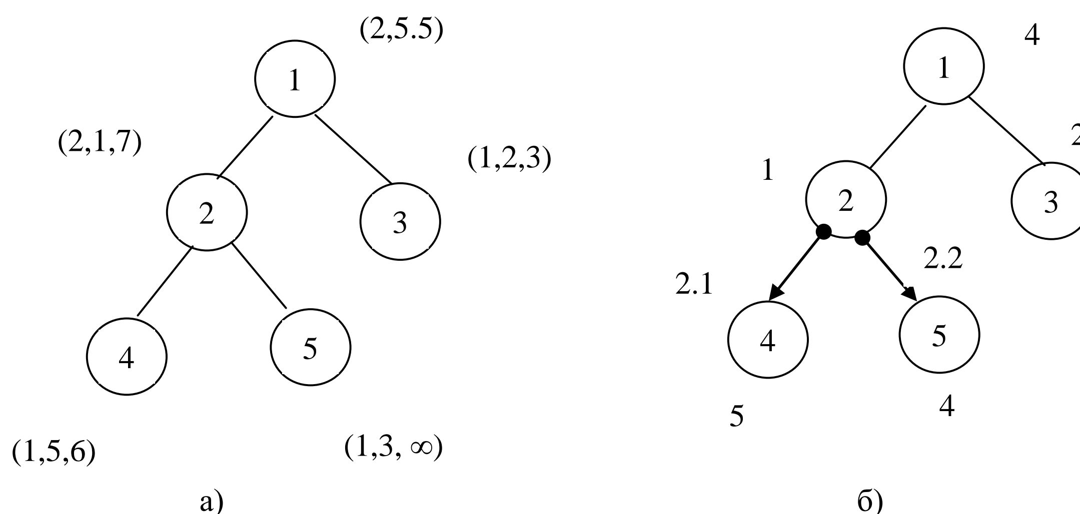


Рисунок 3.7. Типы граф-схем, используемых для изображения алгоритмов
а) информационная граф-схема, используемая для решения задач на неоднородных системах; б) информационно-логическая граф-схема, используемая для решения задач на однородных системах

Принято, что первая составляющая вектора определяет время решения соответствующего программного модуля на первом типе ВМ, вторая – на втором типе ВМ, третья – на третьем и т. д. Символ «∞» обозначает, что данный программный модуль не может выполняться на рассматриваемом типе вычислительного модуля. Информационно-логическая граф-схема, представленная на рисунке 3.7б, предназначена для изображения алгоритма, который будет выполняться на однородных вычислительных системах. Вместо векторных весов используются скалярные величины, так как будет использован одинаковый тип ВМ.

Дуги бывают двух типов: $d_i^1 \in D_1 \subset D$ и $d_j^2 \in D_2 \subset D$, $i, j \in X$, $i \neq j$. Дуги d_i^1 назовем информационными. Эти дуги соответствуют связям, исходящим из исполнительных блоков параллельного алгоритма. Информационно-логические дуги d_j^2 соответствуют связям, исходящим из логических блоков. Дуги d_j^2 нагружены меткой «j.n» для связей, j – это номер оператора в граф-схеме, n – номер дуги, выходящей из j-ого оператора. Нумерация дуг будем осуществлять слева – направо.

Граф, содержащий только дуги из множества D_1, называется информационной граф-схемой алгоритма. Граф, содержащий некоторые дуги $d_j^2 \in D_2$ (в частном случае – все), называется информационно-логической граф-схемой алгоритма. Примеры различных графов приведены на рисунке 3.7 , а и б.

На этом рисунке номера вершин соответствуют номерам блоков в параллельном алгоритме, веса в виде составляющих вектора записываются рядом с соответствующей вершиной. На рис 3.7, б дуги, соответствующие логическим связям или связям по управлению, помечены составными номерами «2.1» и «2.2».

В качестве весов используются трёхмерные векторы. Это означает, что для решения этой задачи будет использована неоднородная ВС с тремя типами процессоров. Например, значение (1, 3, ∞) у первого оператора означает, что время выполнения на первом типе процессора есть 1 условный эквивалент времени выполнения, на втором – 3, на третьем этот оператор не может быть выполнен.

Введем несколько определений, уточняющих построение граф-схемы алгоритма на основе схемы параллельного алгоритма.

Определение 3.9. *Если в параллельном алгоритме существует связь между операторами α, β и α – исполнительный блок, то в граф-схеме G существует дуга* $d_i^1 \in D_1$*, исходящая из вершины α и входящая в вершину β. Эту связь будем обозначать* $\alpha \to \beta$.

Определение 3.10. *Если в параллельном алгоритме существует связь между операторами α, β и α – логический блок, то в графе G существует дуга* $d_j^2 \in D_2$*, исходящая из вершины α и входящая в вершину β. Эту связь будем обозначать* $\alpha \xrightarrow{\alpha.n} \beta$ *и называть связью по управлению. Здесь n – номер логической дуги. Нумерацию удобно осуществлять против движения часовой стрелки, беря за основу начала отсчёта положение стрелки, указывающей на девять часов.*

Связи $\alpha \to \beta$, $\alpha \xrightarrow{\alpha.n} \beta$ назовём задающими связями.

Определение 3.11. *Множество выходных вершин граф-схемы G называется мажорантой граф-схемы G.*

Определение 3.12. *Путями в граф-схеме G назовём последовательности вершин* $\alpha_1, \ldots, \alpha_n$ *, такие, что для любой пары вершин* α_i, α_{i+1} *существует дуга* $d \in D$*, исходящая из вершины* α_i *и входящая в вершину* α_{i+1}*.*

Определение 3.13. *Множество путей в информационно-логической граф-схеме, начинающихся в i-й вершине, соответствующей логическому оператору, и содержащие дугу с меткой* $\alpha \xrightarrow{\alpha.n} \beta$ *, и заканчивающихся вершиной, принадлежащей мажоранте, назовём n-ветвью* α *логического оператора.*

В граф-схеме *G* нет циклов, поэтому все пути имеют конечную длину. Кроме того, будем считать, что значения логических переменных для различных логических операторов не связаны друг с другом, поэтому в процессе реализации алгоритма возможен любой из допустимых путей.

Определение 3.14. *Длиной пути в граф-схеме G назовём количество вершин, входящих в этот путь.*

Определение 3.15. *Характеристикой пути в граф-схеме G со скалярными весами вершин назовём сумму весов вершин, составляющих этот путь.*

Определение 3.16. *Путь с максимальной характеристикой* $T_{кр}$ *в граф-схеме G со скалярными весами вершин назовем критическим.*

В одной *граф-схем*е может быть несколько критических путей.

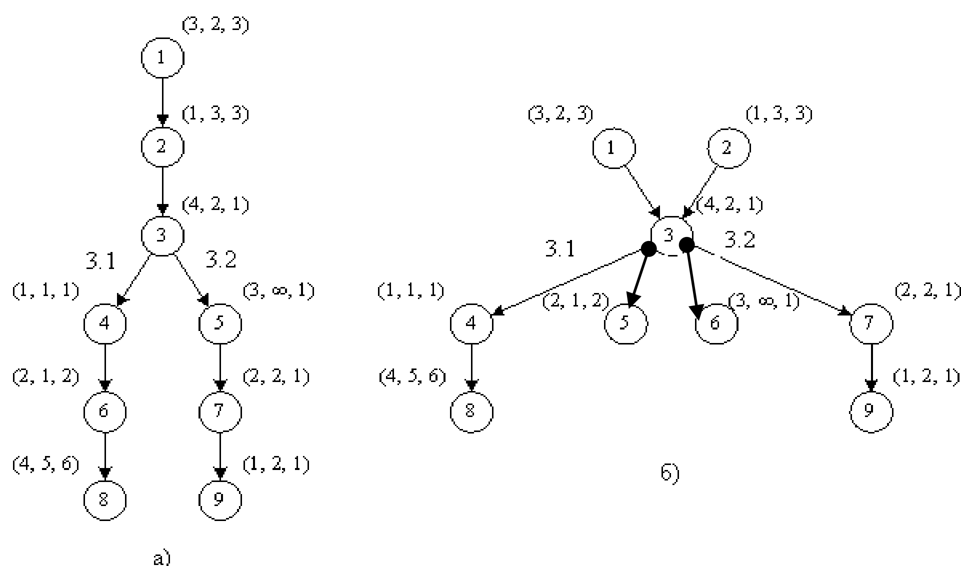


Рисунок 3.8. Граф-схема последовательного (а) и последовательного (б) алгоритмов

В качестве примера на рисунка 3.8. представлены схемы последовательного и параллельного алгоритмов с некоторыми трехмерными весами вершин.

Нумерация блоков в последовательном алгоритме дана в соответствии с ярусностью. В качестве формального средства обработки графов введем матрицу следования *S*. В матрице следования для удобства использования в столбцах помечены не нулевым значением все выходящие из данной вершины связи, а в строках – все входящие в данную вершину связи.

Более точное определение матрицы следования заключается в следующем: *i*-ой вершине графа *G* ставятся в соответствие *i*-ые столбец и строка матрицы *S*; если существует связь по управлению $\alpha \xrightarrow{\alpha.n} \beta$, то элемент матрицы (*i*, *j*) равен *α.n;* при $j \to i$ образуется значение (*i*, *j*) = 1. Остальные элементы матрицы равны 0.

Для отражения весов вершин вводится понятие расширенной матрицы следования S_R: к матрице *S* прибавляется дополнительно *k* столбцов с номерами *m*+1, …, *m*+*k*, где *k* – размерность вектора весов вершин граф-схемы.

Построим расширенные матрицы следования для граф-схем, изображенных на рисунке 3.8.

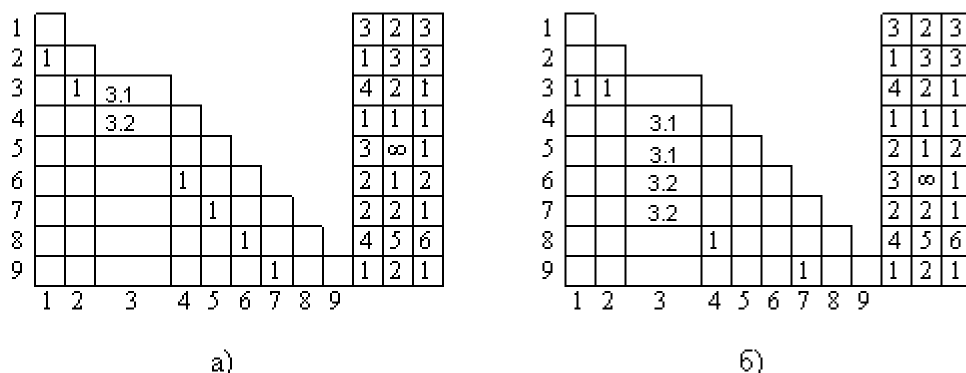


Рисунок 3.9. Расширенные матрицы следования S_R для последовательного алгоритма (а) и параллельного алгоритма (б)

Как видим из рисунка 3.9 матрицы следования получаются треугольными. Рассмотрим условие получения матриц следования в треугольном виде. По условию граф-схемы не должны содержать циклов (контуров). Это означает, что главная диагональ всегда должна содержать нулевые элементы.

Найдем условие построения треугольной матрицы следования для граф-схемы без цикла. Введем в граф-схемы понятие яруса. Возьмем произвольную вершину *α* в графе *G*. Найдем все длины путей, ведущих в *α*. Среди этих длин найдем максимальную. Пусть это будет число h_α. Аналогичные вычисления выполним для некоторой вершины *β*, получим h_β.

Определение 3.17. *Если $h_\alpha = h_\beta = h$, то вершины α и β принадлежат одному ярусу (ярусу h).*

Для обеспечения получения треугольной матрицы следования для графа G необходимо при нумерации вершин придерживаться следующего правила: вершины, принадлежащие d+1 ярусу, должны иметь номера большие, чем номера вершин d-ого яруса. Внутри одного яруса вершины могут нумероваться произвольно. Такую нумерацию назовем нумерацией по ярусам.

При решении задачи распараллеливания важную роль играют не только задающие связи, но и так называемые транзитивные.

Определение 3.18. *Если $\alpha \to \beta$, а $\beta \to \gamma$ связаны задающими связями, то существует транзитивная связь $\alpha --\to \gamma$.*

Определение 3.19. *Множество связей, которые введены направленно внутри всех пар элементов, принадлежащих одному пути в графе G и не связанных задающими связями, назовем множеством транзитивных связей для заданного пути.*

Определение 3.20. *Множество транзитивных связей графа G есть объединение множеств транзитивных связей по всем путям графа G.*

Множество транзитивных связей, очевидно, полностью определяется множеством задающих связей. При формировании множества транзитивных связей следует учитывать, что, если $\alpha \to \beta$ и $\gamma \to \alpha$, где $\gamma = \{\gamma_\mu\}$ – множество всех операторов, связанных с оператором α, то все операторы $\{\gamma_\mu\}$ связаны транзитивно с оператором β.

Рассмотрим построение матрицы следования с транзитивными связями S_T. Возьмем 3 произвольные вершины i, j, k такие, что между ними определены следующие связи: связь вершины i с вершиной j, вершины j с вершиной k, вершины i с вершиной k, как показано на рисунке 3.10.

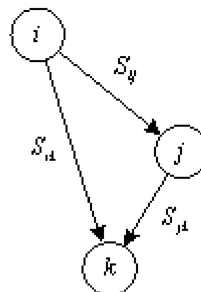


Рисунок 3.10. Рассматриваемая часть информационно-логического графа

…	…	i	…	j	…
…	…	…	…	…	…
j	…	iS'_{ij}	…	…	…
…	…	…	…	…	…
k	…	iS'_{ik}	…	jS'_{jk}	…
…	…	…	…	…	…

Рисунок 3.11. Матрица следования для фрагмента графа, представленного на рис. 3.10. Многоточием обозначены связи, которые в данном случае не представляют интереса

В матрице следования, изображённой на рисунке 3.11, многоточием обозначены другие связи, которые для данного случая не представляют интереса. При построении матрицы *S* элементы матрицы, соответствующие логическим связям, выписываются по формуле (3.1), а информационным – по формуле (3. 2):

$$S'_{ij} = iS_{ij} \quad (3.\ 1)$$

$$S'_{ij} = 1 \quad (3.\ 2)$$

Рассмотрим пример построения матрицы *S* для графа, приведенного на рисунке 3.7, б. Для логических связей в этом графе произведем преобразование по формуле (3.1): $S'_{24} = 2.1$ и $S'_{25} = 2.2$.

Остальные связи в графе не являются логическими, поэтому соответствующие элементы матрицы *S* будут равны единице согласно формуле (3. 2). В результате этого преобразования получим матрицу *S*, изображенную на рисунке 3.12.

1	0	0	0	0	0
2	1	0	0	0	0
3	1	0	0	0	0
4	0	2.1	0	0	0
5	0	2.2	0	0	0
	1	2	3	4	5

Рисунок 3.12. Матрица следования S для графа, изображенного на рисунке 3.7, б

Теперь необходимо произвести анализ значений типов связей между этими вершинами и определить, какую связь между вершинами i и k выбрать: непосредственную или через вершину j.

Первое, что влияет на этот выбор, – наличие транзитивной связи из вершины i в вершину k через вершину j. Обозначим эту связь S_T. Для существования такой связи, как было замечено выше, необходимо, чтобы обе связи S_{ij} и S_{jk} были отличны от нуля. Для проверки существования этой связи введем операцию «⊗», которая аналогична операции конъюнкции в булевой алгебре и в дальнейшем будет называться транзитивной конъюнкцией. В таблице 3.1 приведены истинные значения операции «⊗» применительно к информационно-логическим графам.

Таблица 3.1

Таблица истинности операции «⊗»		
S_{ij}	S_{jk}	$S_{ik} = S_{ij} \otimes S_{jk}$
0	0	0
0	1	0
0	L	0
1	1	1
1	L	L
L_1	L_2	L_1_L_2

Здесь L_1, L_2, L обозначают некоторые кортежи [4] из логических связей $S'_{\alpha\beta}$, где $S'_{\alpha\beta}$ – либо некоторая логическая связь из вершины α в вершину β, либо ранее вычисленная транзитивная связь. Символ «_» – оператор конкатенации [4].

Как нетрудно убедиться операция «⊗» коммутативна, поэтому в таблице приведены значения без учета перестановки операндов.

Рассмотрим построение этой таблицы более подробно. Очевидно, что транзитивная связь есть, если обе связи S_{ij} и S_{jk} отличны от нуля. Соответственно, результат операции на наборах, где хотя бы одна из связей S_{ij} или S_{jk} равна нулю, будет нулевым, т.е. транзитивная связь отсутствует. Далее, в связи с тем, что ход решения алгоритма отражается последовательностью выполненных логических операторов, то в ситуации, когда один операнд равен единице, а второй содержит логический тип связи, необходимо, чтобы результат операции отражал логическую связь. Поэтому на наборе (1, L) результа-

том выполнения операции «⊗» будет L. Исходя из тех же рассуждений, можно сказать, что в случае, когда обе связи содержат логический тип связи, необходимо их объединить и результатом операции на наборе (L_1, L_2) будет выражение $L_1_L_2$. Таким образом, данная операция дает нам транзитивную связь $S_T = S_{ij} \otimes S_{jk}$.

Следующим шагом будет определение, какая связь нам более важна: непосредственная из вершины i в k (S_{ik}) или новая, вычисленная S_T.

Первое, на что следует обратить внимание, как уже говорилось, это на типы связей. Более важной связью будем по-прежнему считать логическую. Такой выбор делается из тех же соображений, что и раньше.

Второе, что определяет результат операции, – непосредственно наличие хотя бы одной из связей между вершинами i и k транзитивной или задающей [1], т.е. необходимо выбрать ненулевую связь, если она есть, при нулевом значении другой связи. Обозначим операцию, осуществляющую такой выбор, «⊕». Из изложенного выше можно сказать, что ее характер подобен операции дизъюнкции булевой алгебры. В дальнейшем эта операция будет называться транзитивной дизъюнкцией. Приведём таблицу истинности для операции «⊕» (таблица 3.2) применительно к информационно-логическим графам, обозначив ранее вычисленную связь $S_{ij} \otimes S_{jk}$ как S_T.

Таблица 3.2

Таблица истинности операции «⊕»		
S_{ik}	S_T	$S_{ik} \oplus S_T$
0	0	0
0	1	1
0	L	L
1	L	L
1	1	1
L_1	L_2	$L_1_L_2$

Таким образом, для трех рассматриваемых вершин можно определить новую связь, используя две введённые операции, т.е. связь S_{ik} можно вычислить по следующей формуле:

$$S_{ik} = S_{ik} \oplus \left(S_{ij} \otimes S_{jk}\right) \qquad (3.\ 3)$$

или применительно к матрице следования:

$$(k,i) = (k,i) \oplus ((j,i) \otimes (k,j)). \qquad (3.\ 4)$$

После последовательного преобразования всей матрицы S мы получим матрицу S_T.

Теперь необходимо привести алгоритм, осуществляющий такой перебор элементов матрицы S для ее преобразования в S_T. Отправной точкой для разработки такого алгоритма является принцип, иллюстрируемый рисунком 3.8. Если при просмотре некоторой k-ой строки матрицы следования, определяющей все входящие в данную вершину связи, обнаруживается в некотором j-ом столбце ненулевой элемент, то, как показано на рисунке 3.10, определяется некоторая вершина j, из которой исходит связь в вершину k. Далее необходимо проверить все входящие в вершину j связи, т.е. проверить все вершины i, а затем к каждой такой тройке применить формулу (3. 4).

Используя соотношения (3.3) и (3.4), построим алгоритм, осуществляющий преобразование матрицы S в матрицу S_T. При описании алгоритма были использованы следующие обозначения:

RS – порядок матрицы следования;

(i, j) – операция по чтению или записи значения в S_T. Первый индекс определяет строку, второй – столбец матрицы S_T.

То, какая операция будет выполнена (считывание или запись), определяется положением выражения (i, j) относительно символа «=». Если она находится слева, то производится запись, если справа – чтение.

Матрица следования является квадратной, поэтому оба цикла определены для множества значений [1, …, RS].

Алгоритм 3.1. Построение матрицы следования с транзитивными связями.

1. Вычислим $S_T := S$.

2. В матрице следования S_T размера RS просматриваются строки, начиная с первой.

3. Если в очередной i-й строке матрицы S_T отыскивается элемент $(i, j) <> 0$, то вычисляются значения элементов $(i, 1)$, …, $(i, j\text{-}1)$ матрицы S_T, используя соотношение (4 – 4):

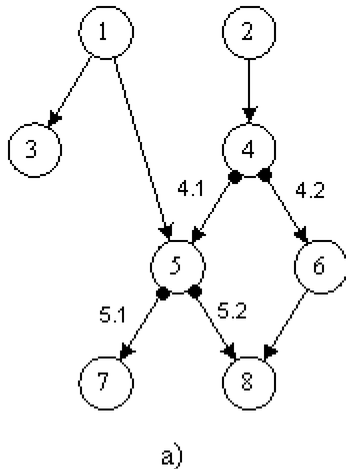
$$(i,k) := ((j,k) \otimes (i,j)) \oplus (i,k)$$

для $k = 1, \ldots, j\text{-}1$.

4. Вычислим $j := j+1$. Если $j \le RS$, то переход на шаг 3, иначе – работа алгоритма заканчивается (просмотрены все строки).

Конец алгоритма.

Рассмотрим пример построения матрицы следования с транзитивными связями, используя данные операции, для графа, изображённого на рисунке 3.13а и соответствующей ему матрицы следования, показанной на рисунке 3.13б.



а)

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0
5	1	0	0	4.1	0	0	0	0
6	0	0	0	4.2	0	0	0	0
7	0	0	0	0	5.1	0	0	0
8	0	0	0	0	5.2	0	0	0

б)

Рисунок 3.13. Пример графа (а) и соответствующей ему матрицы следования (б)

Выполним алгоритм построения матрицы следования с транзитивными связями, используя соотношение (3.4). Первая, вторая и третья строки не удовлетворяют условию применения формулы (3.3) согласно шагу 3 алгоритма 3.1.

Для четвертой вершины:

$$(4,1) := \big((2,1) \otimes (4,2)\big) \oplus (4,1) = \big(0 \otimes 1\big) \oplus 0 = 0 \oplus 0 = 0$$

Для пятой вершины:

$$(5,1) := \big((4,1) \otimes (5,4)\big) \oplus (5,1) = \big(0 \otimes 4.1\big) \oplus 1 = 0 \oplus 1 = 1,$$

$$(5,2) := \big((4,2) \otimes (5,4)\big) \oplus (5,2) = \big(1 \otimes 4.1\big) \oplus 0 = 4.1 \oplus 0 = 4.1,$$

$$(5,3) := \big((4,3) \otimes (5,4)\big) \oplus (5,3) = \big(0 \otimes 4.1\big) \oplus 0 = 0 \oplus 0 = 0.$$

Для шестой вершины:

$$(6,1) := \big((4,1) \otimes (6,4)\big) \oplus (6,1) = \big(0 \otimes 4.2\big) \oplus 0 = 0 \oplus 0 = 0,$$

$$(6,2) := \big((4,2) \otimes (6,4)\big) \oplus (6,2) = \big(1 \otimes 4.2\big) \oplus 0 = 4.2 \oplus 0 = 4.2,$$

$$(6,3) := \big((4,3) \otimes (6,4)\big) \oplus (6,3) = \big(0 \otimes 4.2\big) \oplus 0 = 0 \oplus 0 = 0.$$

Для седьмой вершины:

$(7,1) := \big((5,1) \otimes (7,5)\big) \oplus (7,1) = \big(1 \otimes 5.1\big) \oplus 0 = 5.1 \oplus 0 = 5.1,$

$(7,2) := \big((5,2) \otimes (7,5)\big) \oplus (7,2) = \big(4.1 \otimes 5.1\big) \oplus 0 = 4.1{,}5.1 \oplus 0 = 4.1{,}5.1,$

$(7,3) := \big((5,3) \otimes (7,5)\big) \oplus (7,3) = \big(0 \otimes 5.1\big) \oplus 0 = 0 \oplus 0 = 0,$

$(7,4) := \big((5,4) \otimes (7,5)\big) \oplus (7,4) = \big(4.1 \otimes 5.1\big) \oplus 0 = 4.1{,}5.1 \oplus 0 = 4.1{,}5.1.$

Для восьмой вершины:

$(8,1) := \big((5,1) \otimes (8,5)\big) \oplus (8,1) = \big(1 \otimes 5.2\big) \oplus 0 = 5.2 \oplus 0 = 5.2,$

$(8,2) := \big((5,2) \otimes (8,5)\big) \oplus (8,2) = \big(4.1 \otimes 5.2\big) \oplus 0 = 4.1{,}5.2 \oplus 0 = 4.1{,}5.2,$

$(8,3) := \big((5,3) \otimes (8,5)\big) \oplus (8,3) = \big(0 \otimes 5.2\big) \oplus 0 = 0 \oplus 0 = 0,$

$(8,4) := \big((5,4) \otimes (8,5)\big) \oplus (8,4) = \big(4.1 \otimes 5.2\big) \oplus 0 = 4.1{,}5.2 \oplus 0 = 4.1{,}5.2.$

В результате получим матрицу следования с транзитивными связями (рис. 3.14).

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0
5	1	4.1	0	4.1	0	0	0	0
6	0	4.2	0	4.2	0	0	0	0
7	5.1	4.1,5.1	0	4.1,5.1	5.1	0	0	0
8	5.2	4.1,5.2	0	4.1,5.2	5.2	0	0	0

Рисунок 3.14. Матрица следования с транзитивными связями для графа, изображенного на рисунке 3.13 а

Введение операций «⊗» и «⊕» позволяет построить матрицу следования с транзитивными связями, в которой сохраняется информация о проходимых в процессе выполнения алгоритма логических операторах, что позволяет исключить из программы большое число промежуточных поисковых операций. Так при определении внешних и внутренних замыканий эффективно используется информация из S_T. Кроме того, полученные в этой матрице кортежи из логических связей могу быть использованы для получения вероятностей прохождения по тем или иным путям в граф-схеме алгоритма, что может быть использовано для построения эффективных планов параллельных вычислений.

Определение контуров в граф-схеме алгоритма

Алгоритм использует свойство появления ненулевого элемента в главной диагонали матрицы S_T. В качестве исходной берется не треугольная матрица S. Поэтому при получении транзитивных связей предыдущий алгоритм вызывается несколько раз до получения неизменяемой матрицы S_T.

Алгоритм 3.2. Определение контуров в граф-схеме алгоритма.

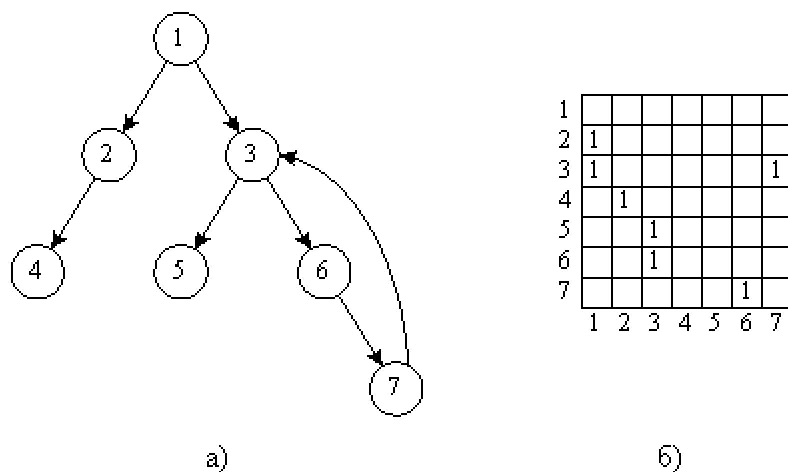
1. Вычисление матрицы $S_{T_i} := S$, $i := 0$.

2. С помощью алгоритма 3.1, используя матрицу S_{T_i} вычислить матрицу $S_{T_{i+1}}$.

3. На главной диагонали матрицы $S_{T_{i+1}}$ определяется, есть ли ненулевые элементы? Если есть, то исследуемый граф имеет цикл – работа алгоритма завершена. В противном случае проверяем, изменилась ли матрица $S_{T_{i+1}}$. Если $S_{T_{i+1}} = S_{T_i}$, то исследуемый граф не имеет контуров. Алгоритм заканчивает работу. Иначе определяется $S_{T_i} := S_{T_{i+1}}$, $i := i+1$ и осуществляется переход на шаг 2.

Конец алгоритма.

Пример работы данного алгоритма проиллюстрирован на рисунке 3.15 и рисунке 3.16.



	1	2	3	4	5	6	7
1							
2	1						
3	1						1
4		1					
5			1				
6			1				
7						1	

Рисунок 3.15. Иллюстрация работы алгоритма 3.2: граф (а) и матрица следования S (б)

	1	2	3	4	5	6	7
1							
2	1						
3	1		1			1	1
4	1	1					
5	1		1			1	1
6	1		1			1	1
7	1		1			1	1

Рисунок 3.16. Матрица S_T для графа, изображенного на рисунке 3.15 а

Вопросы для самопроверки

1. В чем отличие матрицы следования от расширенной матрицы следования и матрицы следования с транзитивными связями?

2. В каких случаях используется треугольная матрица следования?

3. Что является признаком наличия цикла (контура) в информационно-логическом графе в матрице следования?

4. С какой целью вычисляется матрица следования с транзитивными связями?

5. В чём смысл операций «⊗» и «⊕»?

Задание на лабораторную работу

Используя программу, разработанную в предыдущей лабораторной работе, необходимо дополнить ее следующими функциями:

1. Построение матрицы следования S по заданному графу.
2. Построение матрицы следования S_R (с указанием весов).
3. Построение матрицы следования S_T с транзитивными связями.
4. Определение наличия или отсутствия контура в исходном графе.
5. Продемонстрировать работающую программу преподавателю и получить отметку о её выполнении.
6. Сохранить копию программы для выполнения последующих лабораторных работ на дискете.
7. Оформить отчет о проделанной работе.

Содержание отчета

1. Цель работы.
2. Ответы на вопросы для самопроверки.
3. Схемы алгоритмов и их описание.
4. Распечатки экранных форм, полученных в результате работы программы.
5. Анализ полученных результатов.

3. Лабораторная работа №4. Построение матрицы логической несовместимости операторов

Цель работы – ознакомление с понятием логической несовместимости операторов, практическая реализация алгоритмов работы с матрицами логической несовместимости L в виде программных модулей.

Теоретическая часть

Для оценки возможности выполнения программных модулей параллельно, важную роль играют логически несовместимые операторы.

Рассмотрим множество вершин, принадлежащих n-ветви i-го логического оператора. Это множество назовем $Mi.n$. Аналогично построим множество вершин для дуги k – это множество вершин, принадлежащих k-ветви i-го логического оператора – множество MK_i. В множества $Mi.n$ и $Mi.k$ сама вершина i не входит.

Определение 4.1. *Если вершина $p \in Mi.n$, а $q \in Mi.k$ и соответствующие им операторы могут выполняться либо один, либо другой при однократном выполнении алгоритма, то эти операторы называются логически несовместимыми.*

При реализации алгоритма в логическом операторе i выполняется либо ветвь $i.n$, либо $i.k$. Следовательно, при планировании параллельных вычислений следует исключать планирование параллельного выполнения операторов, принадлежащих разным ветвям, т.е. попросту исключить их из планирования. Однако встречаются ситуации, когда ветви $i.n$ и $i.k$ пересекаются, т.е. $Mi.n \cap Mi.k = Mi.nk \neq \emptyset$.

Определение 4.2. *Если $Mi.nk \neq \emptyset$, то существует внутреннее замыкание i-го логического оператора.*

В этом случае операторы $t \in Mi.nk$ могут планироваться для параллельного выполнения. На рисунке 4.1 приведён граф, в котором на вершине 6 произошло пересечение логических ветвей оператора 1. Операторы 6, 7, 9, 10 могут быть запланированы для параллельного выполнения.

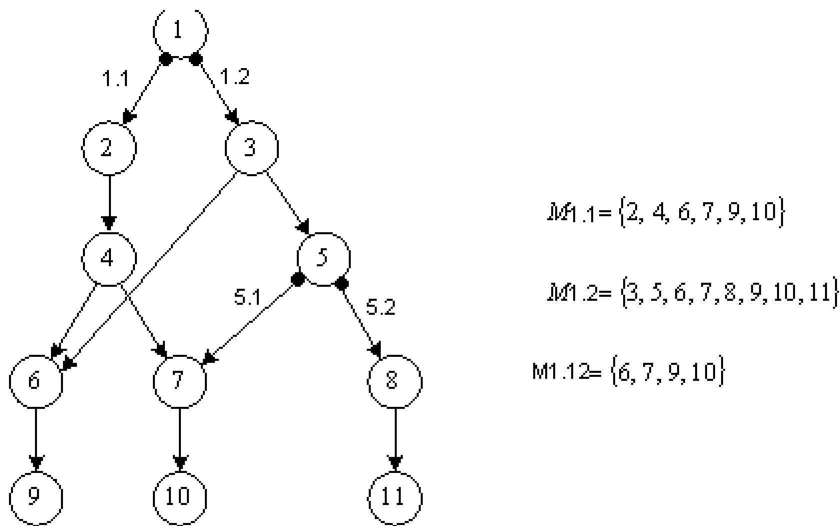


Рисунок 4.1. Граф-схема ИЛГ с пересечением логических путей оператора 1

Определение 4.3. *Вершина* $z \in Mi.nk$ *и имеющая наименьший номер, называется минимальной внутренне замкнутой вершиной i-го логического оператора и обозначается* inz_i.

Так, для примера, представленного на рисунке 4.2, $inz_1 = 6$.

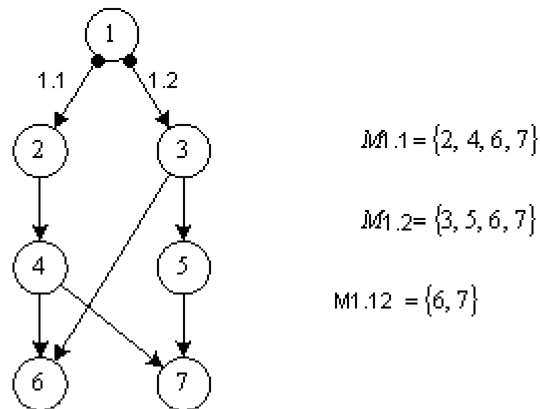


Рисунок 4.2. Граф-схема алгоритма, имеющая замыкающие дуги и со стороны 1.1, и со стороны 1.2

Следует отметить, что замыкание логических путей может осуществляться за счет внешних информационных связей. Как показано на рисунке 4.3, замыкание может произойти за счет информационных связей путями, идущими от операторов 3 или 4. При этом должен существовать информационный путь к вершинам 3 или 4 от входа в алгоритм (вершина 1).

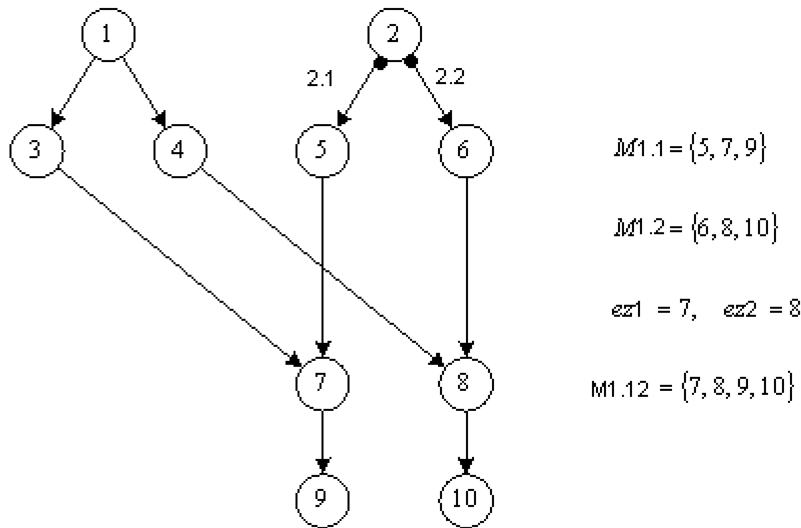


Рисунок 4.3. Граф-схема алгоритма с замыканием логических ветвей за счёт вершин, не принадлежащих путям оператора 2

Целесообразно рассматривать внешние замыкания для ветвей *i.n* и *i.k* отдельно. Это связано с тем, что при рассмотрении возможности параллельного выполнения операторов, включенных в логические ветви, необходимо учитывать результаты как внешнего, так и внутреннего замыканий совместно, имея в виду при этом, что возможно внешнее замыкание только одной ветви.

Определение 4.4. *Если существует информационный путь в вершину* $Z \in Mi.nk$ *от начальной вершины граф-схемы, то вершина Z называется внешне замкнутой в n-ветви для i-го логического оператора. Если таких вершин несколько, то вершину Z с минимальным номером называют минимальной внешне замкнутой вершиной в n-ветви для i-го логического оператора.*

Обозначим эту вершину $ezi.n = x_j$.

Определение 4.5. *Если множество* $Mi.n = \{x_1, \dots, x_j, \dots, x_{j'}\}$ *содержит внешнее замыкание* $ezi.n = x_j$*, то подмножество* $Vi.n = \{x_j, \dots, x_{j'}\}$ *называется внешне замкнутым для n-ветви логического оператора* L_i.

При рассмотрении влияния внешних и внутренних замыканий для оценки возможности распараллеливания операторов, принадлежащих ветвям логического оператора, необходимо учесть, что:

1. внутреннее замыкание, как правило, порождает операторы, которые можно выполнять параллельно;

2. для возможности распараллеливания операторов, принадлежащих путям логического оператора, достаточно одного внешнего замыкания при наличии внутреннего;

3. определение множества MZ_i всех замкнутых операторов i-го логического оператора требуется вычислить объединение множеств: $MZ_i := \bigcup_{r=1,t=1}^{n,k} Mi.rt \cup Vi.r$.

Ситуации, соответствующие пунктам 1 и 2, проиллюстрированы на рисунках 4.1, 4.2 и 4.3. На рисунке 4.4 рассмотрена ситуация, соответствующая пункту 3.

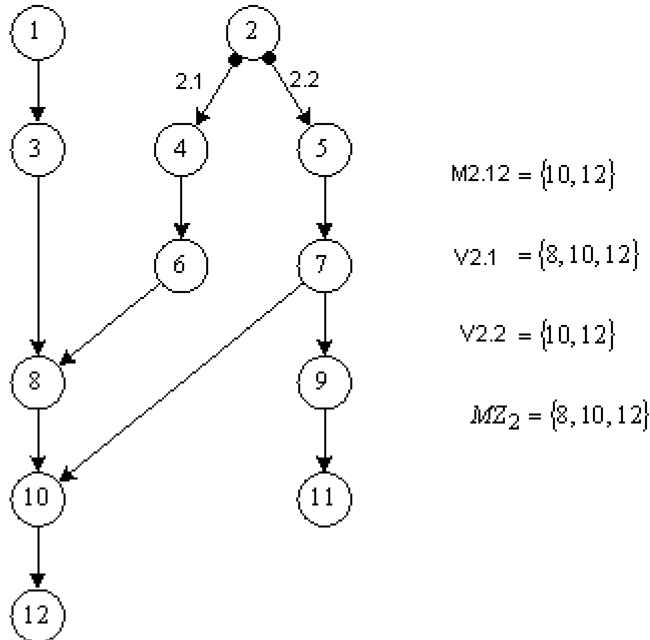


Рисунок 4.4. Пример граф-схемы, в которой внешнее замыкание ветви 2.1 уточняет множество операторов, подлежащих распараллеливанию

На рисунке 4.5 приведён алгоритм построения матрицы логической несовместимости операторов, в котором использованы следующие обозначения:

S – матрица следования;

RS – размерность матрицы S;

S_T – матрица с транзитивными связями;

RST – размерность матрицы S_T;

MLO – множество логических операторов;

$RMLO$ – размерность множества логических операторов;

M – множество вершин операторов;

$Mk.n$ – множество вершин операторов, принадлежащих путям, включающим дугу n k-го логического оператора;

$Vk.g$ – множество вершин операторов, внешне замкнутых для k-го логического оператора связи G;

$Mk.g$ – множество вершин операторов, принадлежащих путям, включающим дугу g k-го логического оператора;

RMk.n – размерность множества логических операторов ветви *k.n*;

RMk.g – размерность множества логических операторов ветви *k.g*;

Nk.g – множество вершин операторов, внутренне замкнутых для *k*-го логического оператора;

Vk.n – множество вершин операторов, внешне замкнутых для *k*-го логического оператора связи *n*;

MZ_k – объединенное множество внешних и внутренних замыканий для *k*-го логического оператора.

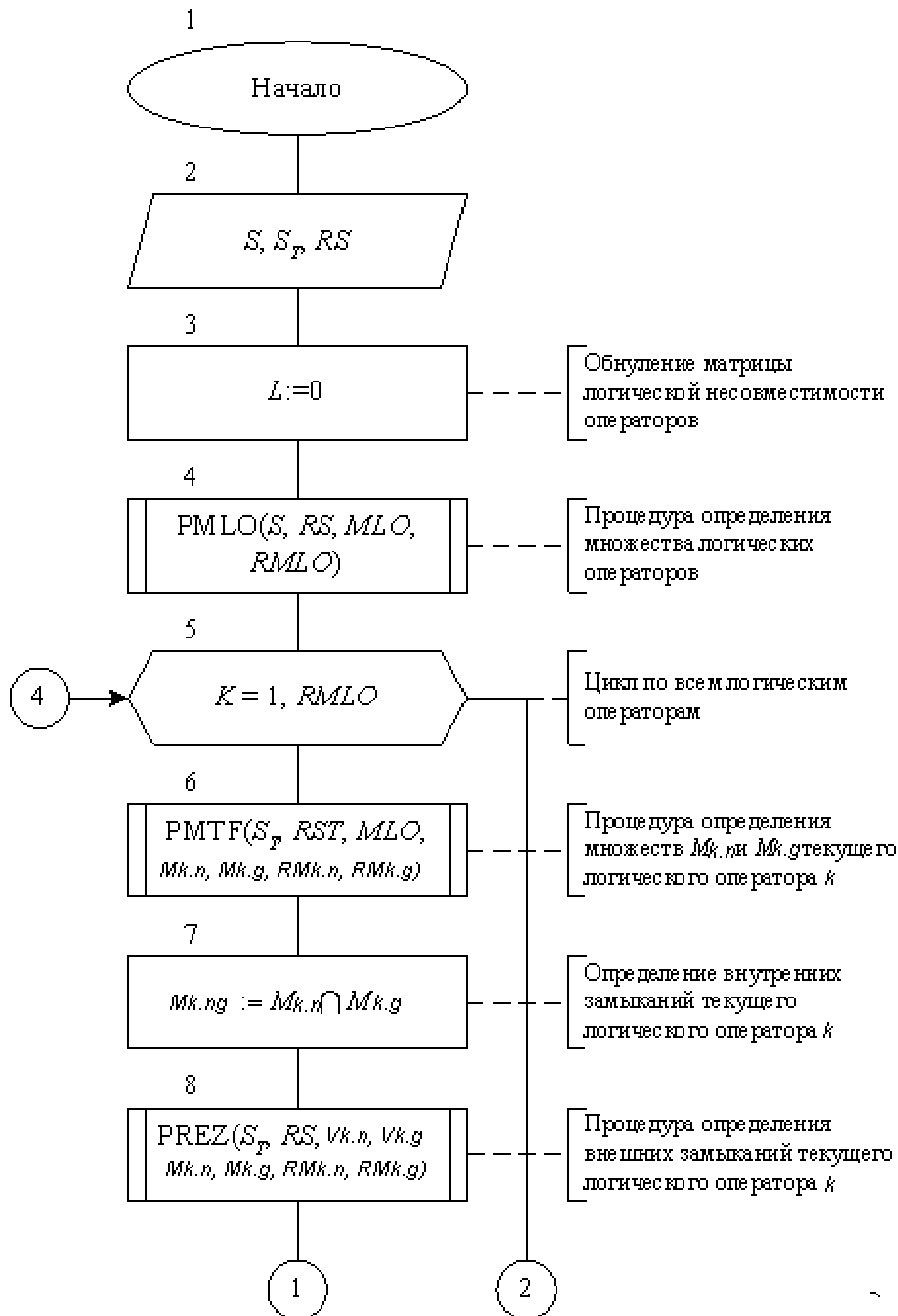


Рисунок 4.5. Схема алгоритма построения матрицы логической несовместимости операторов. Начало

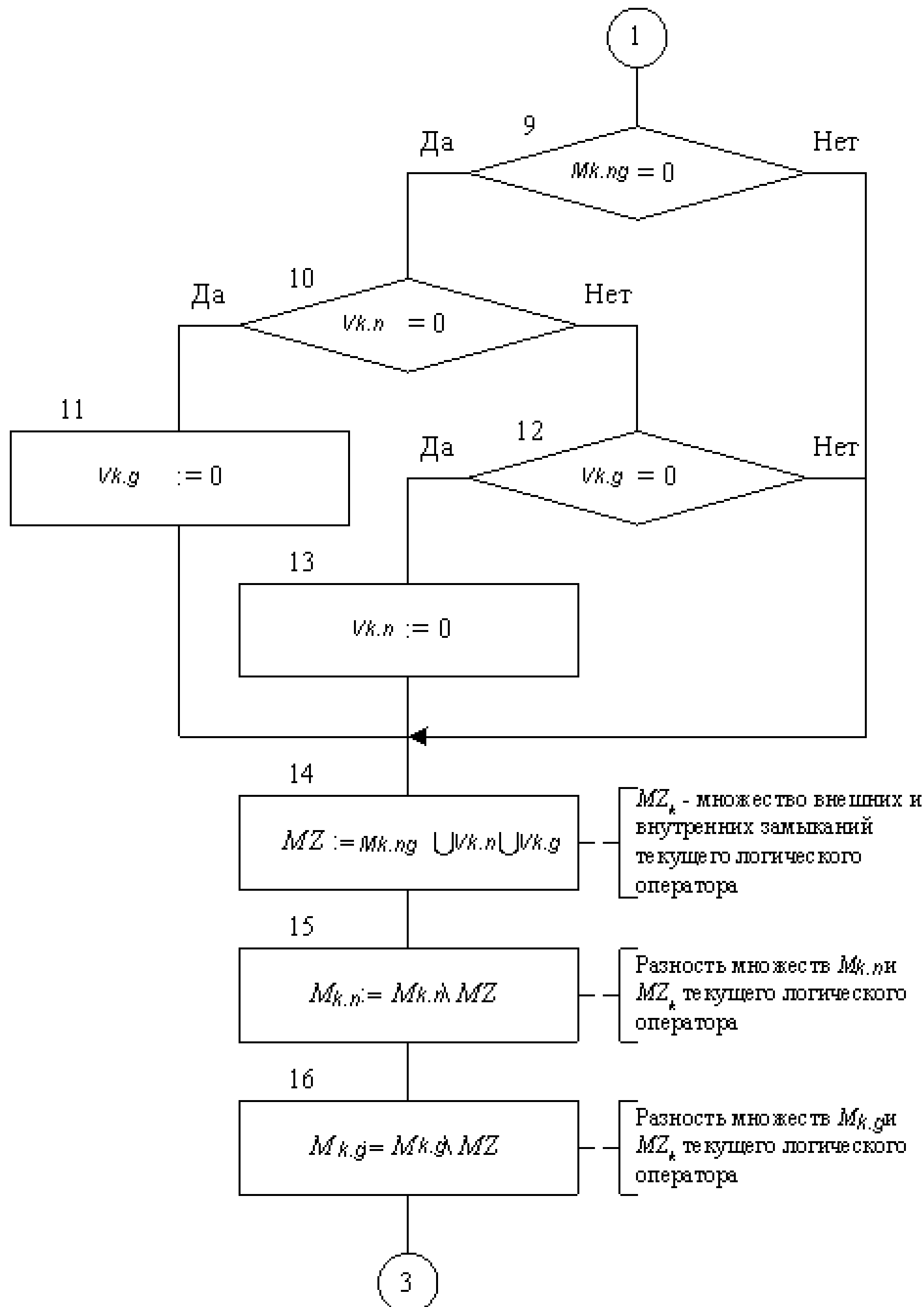


Рисунок 4.5. Продолжение

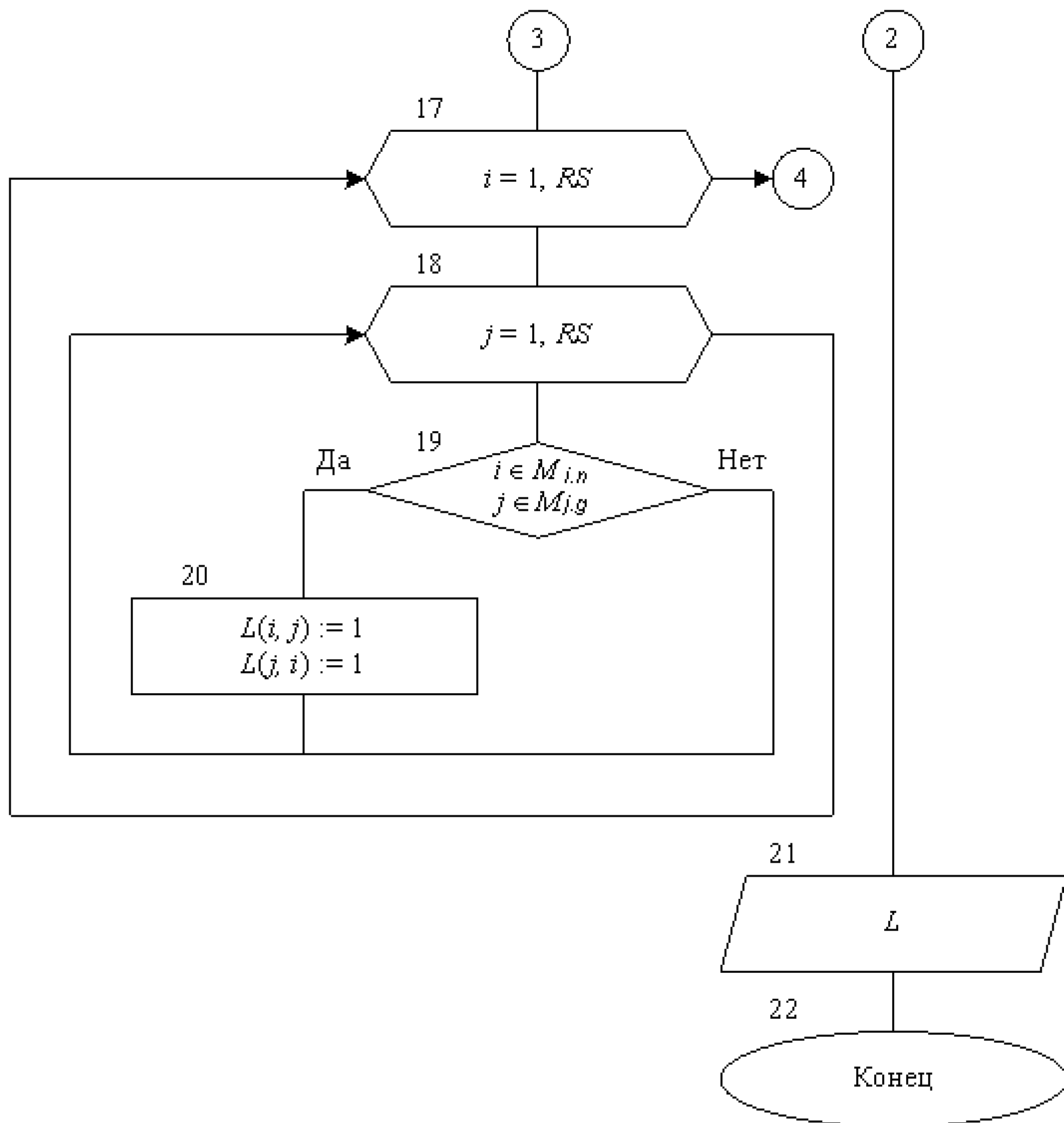


Рисунок 4.5. Окончание

Процедура PMLO.

Признаком логического оператора матрицы S является появление в соответствующем столбце значений *j.n* или *j.g*.

Алгоритм 4.1. Получение множества логических операторов.

1. В матрице *S*, размера *RS* выбираем первый столбец (j := 1), $RMLO := 0$, $MLO := \varnothing$.

2. Просматриваем *j*-й столбец по строкам и определяем равенство текущего элемента матрицы *j.n* или *j.g*.

3. Если найден такой элемент, то полагаем $MLO := MLO \cup \{j\}$, $RMLO := RMLO + 1$, $j := j+1$.

4. Если $j \le$ RS,то переходим к шагу 2, иначе – конец алгоритма.

Конец алгоритма.

Процедура PMNG.

Для получения множеств $Mk.n$ и $Mk.g$ k-го логического оператора необходимо просмотреть k-й столбец и включить во множество MN номера операторов, которые в соответствующих строках имеют значение $k.n$, а во множество MG – номера операторов, которые в соответствующих строках имеют значение $k.g$.

Алгоритм 4.2. Получение множеств $Mk.n$ и $Mk.g$ k-го логического оператора.

1. В соответствии со значением k, выбираем элемент множества $q_k \in MLO$. В q_k столбце матрицы S_T просматриваем i-е строки, $i := 1$ (номер строки), $l := 1$ (номер позиции в множестве $Mk.n$), $m := 1$ (номер позиции в множестве $Mk.g$).

2. Если элемент матрицы $S_N(i, k) = j.n$, то $Mk.n[l] :=$ i, $l := l+1$ и осуществляется переход к шагу 5.

3. Если $S_N(i, k) = j.g$, то $Mk.g[m] := i$; $m := m+1$ и выполняется шаг 5.

4. Если условия пунктов 2 и 3 не выполняются, то осуществляется переход на шаг 5.

5. Вычислим $i := i+1$; если $i > RST$, то $RMk.n :=$ l, $RMk.g := m$ и выполнение алгоритма заканчивается, иначе осуществляется переход на шаг 2.

Конец алгоритма.

Процедура PREZ.

Процедура PREZ формирует множество внешних замыканий для рассматриваемого логического оператора, используя свойства матрицы S_T. Берётся i-ая строка матрицы S_T, содержащая все нули (вход ИЛГ). Затем – для i-го столбца номера всех элементов, равных 1, фиксируются в множестве WZ. Если для рассматриваемого k-го логического оператора пересечения множеств $WZ \cap Mk.n$ и $WZ \cap Mk.g$ не пусты, то обнаружены внешние замыкания для ветвей k-го логического оператора (номера внешне замкнутых операторов входят в эти пересечения).

Алгоритм 4.3. Формирование множества внешних замыканий.

1. Формируем множество из номеров нулевых строк матрицы S_T: $ZS := \{i_1, ..., i_q\}$. Полагаем $Vk.n := \varnothing$, $Vk.g := \varnothing$.

2. Для всех элементов $i_p \in ZS$ строим множество ED_p номеров строк, содержащих единичные элементы в i_p столбцах.

3. Исключим из множества ED_p, $p = 1, \ldots, q$ элементы, равные номеру рассматриваемого логического оператора k. Перенумеруем множества ED_p, учитывая удаленные

элементы. Получим множество ED_u, $u = 1, \ldots, f$, $f < q$. Если все $ED_u = \varnothing$, то $Vk.n := \varnothing$, $Vk.g := \varnothing$.

4. Вычислим множества $Vk.n := Vk.n \cup (Mk.n \cap ED_u)$, $Vk.g := Vk.g \cup (Mk.g \cap ED_u)$, $u = 1, \ldots, f$, где $Mk.g$ и $Mk.n$ – множества операторов для текущего вложенного оператора k.

Конец алгоритма.

Вопросы для самопроверки

1. Дайте определение логически несовместимых операторов.
2. Дайте определения внешних и внутренних замыканий.
3. Когда внешнее замыкание одной ветви сказывается на определенном множестве логически несовместимых операторов?
4. С какой целью вычисляется матрица логически несовместимых операторов?
5. Почему в алгоритме 4.3 анализируются только те столбцы матрицы S_T, номера которых совпадают с номерами нулевых строк этой матрицы?
6. Почему учёт воздействия внешних замыканий на логическую несовместимость зависит от наличия или отсутствия внешних замыканий?

Задание на лабораторную работу

Используя программу, разработанную в предыдущей лабораторной работе, необходимо дополнить её следующими функциями:

1. Для заданного информационно-логического графа построить матрицу логической несовместимости операторов.
2. Выделить на исходном графе логически несовместимые вершины.
3. Продемонстрировать работающую программу преподавателю и получить отметку о её выполнении.
4. Сохранить копию программы для выполнения последующих лабораторных работ на дискете.
5. Оформить отчет о проделанной работе.

Содержание отчета

1. Цель работы.
2. Ответы на вопросы для самопроверки.
3. Схемы алгоритмов и их описание.
4. Распечатки экранных форм, полученных в результате работы программы.
5. Анализ полученных результатов.

4. Лабораторная работа №5. Построение множеств взаимно независимых операторов

Цель работы – ознакомление с понятием множеств взаимно независимых операторов (ВНО). Вычисление матрицы ВНО. Определение множества ВНО и упорядочение его в порядке убывания.

Теоретическая часть

Для определения возможности распараллеливания операторов необходимо произвести анализ независимости операторов по данным и по управлению. Для этих целей вводится матрица независимости операторов *M*.

Определение 5.1. *Симметричная матрица* $M(i, j) = S'(i, j) V\ L(i, j)$, *где V – операция дизъюнкции булевой алгебры,* $S'(i, j) = S_T(i, j)$, *если* $S_T(i, j) = 0$ *и* $S'(i, j) = 1$, *если* $S_T(i, j) \neq 0$ *для i = 1, …, RST и j = 1, …, RST, а L(i, j) – матрица логической несовместимости, называется матрицей независимости операторов.*

Матрица *M* отражает информационно-логические связи между операторами без учета их ориентации с учетом транзитивных связей и логическую несовместимость операторов. Например, для графа, изображённого на рисунке 5.1, матрица независимости показана на рисунке 5.2.

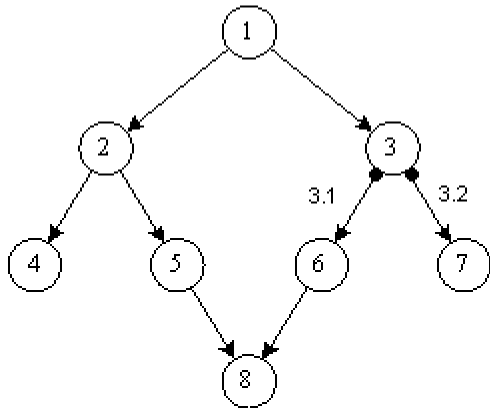


Рисунок 5.1. Граф *G* с информационно-логическими связями

	1	2	3	4	5	6	7	8
1		1	1	1	1	1	1	1
2	1			1	1			1
3	1					1	1	1
4	1	1						
5	1	1						1
6	1		1				1	1
7	1		1			1		
8	1	1	1		1	1		

Рисунок 5.2. Матрица независимости *M*

Следует отметить, что в соответствии с определением 5.1 для информационного графа матрица *M* совпадает с матрицей S'.

Определение 5.2. *Операторы* α *и* β *– взаимно независимые (ВНО), если в матрице независимости* $M(\alpha, \beta) = M(\beta, \alpha) = 0$.

Определение 5.3. *Операторы* $\{\alpha_i\}, i = 1, \dots, s$ *образуют полное множество ВНО, если для любого оператора* $j \notin \{\alpha_i\}$ *существует пара элементов матрицы независимости* $M \quad (\alpha_i, j) = (j, \alpha_i) = 1, i \in \{1, \dots, s\}$.

Определение 5.4. *Множество, содержащее наибольшее число элементов для данного графа, называется максимально полным.*

Пусть некоторый алгоритм представлен информационно-логической граф-схемой (см. рисунок 5.1). По нулевым элементам матрицы независимости *M* в строке каждого оператора можно указать множество тех операторов, каждый из которых при выполнении некоторых условий может быть выполнен одновременно с данным, т.е. он информационно или по управлению не зависит от данного и не является с ним логически несовместимым.

Работа алгоритма поиска полного множество ВНО основана на использовании стека. В стек поочерёдно заносятся строки матрицы *M*, а также строки, получаемые в результате сложения строк матрицы *M* по правилу дизъюнкции: $E = \bigvee_{v=1}^{d} \{(m_v, 1), \dots, (m_v, n)\}, d \in \{1, \dots, n\}$, где n – размер матрицы *M*, а m_v – складываемые строки.

В стеке также хранится информация о том, на каком элементе закончился просмотр строки, и какое множество ВНО при этом сформировалось. Эта информация нужна для того, чтобы возобновить просмотр строки с того места, где была сделана остановка (очередной виток рекурсии) и с тем же набором операторов во множестве ВНО. Структура стека показана на рисунке 5.3.

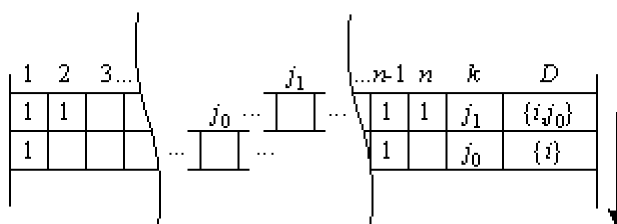


Рисунок 5.3. Структура стека для хранения нуль-единичных строк

Алгоритм 5.1. Нахождение полных множеств ВНО.

1. Пусть *W* – массив полных множеств ВНО. Максимальное полное множество ВНО обозначим через *A*, а *l* – число элементов в нём. Очередное формируемое множество ВНО обозначим через *D* (см. рисунок 5.3), *d* – количество элементов в нём. Номер

очередного найденного нулевого элемента в строке обозначим через k. Изначально полагаем, что стек пуст, $W=\varnothing$, $A=\varnothing$, $l=0$, $D=\varnothing$, $d=0$, $k=0$.

2. Загружаем очередную i-ю строку в стек, $i=1, \dots, n$, где n – размер матрицы M. Полагаем $D = \{i\}, d = 1, k = i$. Если все строки обработаны, то выполнение алгоритма заканчивается. Найдены все полные множества ВНО (W) и определено максимальное (A).

3. В строке-вершине стека находим очередной нуль, занимающий позицию $j > k$. Если нуль найден, то переходим к выполнению шага 6, иначе выполняется следующий шаг.

4. Если такого нуля нет или все нули найдены, выполняем проверку на полноту найденного множества D. Если в строке-вершине стека все нули соответствуют всем операторам из D, то найденное множество полное. Производим сохранение $W_m = D$ и переходим к шагу 7. Если в строке-вершине есть хотя бы один нуль, не соответствующий операторам из D, то найденное множество не является полным. Переходим к следующему шагу.

5. Исключаем из стека строку-вершину (не будем забывать, что, исключая строку, мы уничтожаем и текущее значение k и D и возвращаемся к их предыдущим значениям) Если после этого стек исчерпан, выполняем шаг 2. В противном случае выполняем шаг 3.

6. В текущей вершине стека присваиваем $k = j$. Складываем логически (поэлементная дизъюнкция) строку, исключая поля k и D (см. рисунок 5.3), из вершины стека со строкой с номером j – формируем новую вершину стека. В новой вершине стека формируем множество $D = D \cup \{j\}, d = d + 1, k = j$. Переходим к шагу 3.

7. Сравниваем значения d и l. Если $d > l$, то $A=D$, $l=d$. Независимо от результата сравнения переходим к шагу 5.

Конец алгоритма.

Рассмотрим работу приведённого выше алгоритма (точнее его основную часть – работу по складыванию строк в стеке) на примере графа, изображённого на рисунке 5.1 и его матрицы независимости, представленной на рисунке 5.2.

Записываем в стек первую строку ($i = 1$) матрицы M (см. рисунок 5.4а).

Начинаем просматривать эту строку на наличие нулей, начиная со второй позиции ($i + 1$). Выясняем, что в первой строке нет нулей в старших позициях. Это означает, что первое найденное нами полное множество ВНО будет состоять из единственного

оператора: {1}. Исключаем первую строку матрицы M из стека и заносим вторую строку (рисунок 5.4б).

Осуществляем просмотр этой строки, начиная с третьей позиции. Находим первый нулевой элемент, который стоит в позиции 3. Это нуль указывает, что операторы 2 и 3 взаимно независимы (могут выполняться параллельно). Складываем логически строки, соответствующие операторам 2 и 3. Получаем новую строку $s_1 = "2" \vee "3"$ в вершине стека и весь стек в виде, представленном на рисунке 5.4в.

Строка в вершине стека содержит нули только в тех позициях, которые соответствуют образующим её операторам 2 и 3. Это означает, что мы нашли второе полное множество ВНО {2, 3}.

Убираем из стека строку s_1 и начинаем просматривать строку 2 со следующего места после остановки (когда стали формировать строку s_1), т.е. с позиции 4. Находим следующий нуль, который оказывается в позиции 6. Формируем новую вершину стека, складывая логически строки 2 и 6 (рисунок 5.4г)

Новая строка содержит нули только в позициях 2 и 6. Значит, найдено ещё одно полное множество ВНО {2, 6}.

Исключаем строку s_2 из стека и во второй строке находим следующий нуль, находящийся в позиции 7. Формируем новую вершину стека – строку $s_3 = "2" \vee "7"$ (рисунок 5.4д).

В этой строке нули так же соответствуют только операторам, «участвующим» в её формировании. Значит, найдено четвёртое полное множество ВНО {2, 7}.

Исключаем строку s_3 из стека. Все нули строки 2 найдены, поэтому исключаем и её из стека. Помещаем в стек третью строку матрицы независимости M (рисунок 5.4е).

Находим первый нуль в строке 3 правее третьей позиции. Этот нуль стоит в позиции 4. Складываем логически строки 3 и 4. Формируем новую строку в вершине стека и весь стек в виде, представленном на рисунке 5.4ж.

Строка s_4 содержит нули не только в позициях 3 и 4. Первый такой нуль, правее позиции 4, находится в позиции 5. Формируем новую вершину стека $s_5 = s_4 \vee "5"$. Стек принимает вид, показанный на рисунке 5.4и.

Строка s_5 содержит нули только в позициях 3, 4, 5. Это значит, что найдено очередное полное множество ВНО {3, 4, 5}.

Исключаем строку s_5 из стека. Строка s_4 также исчерпана, поэтому исключаем и её. Просматриваем дальше строку 3. Следующий нуль будет найден в позиции 5. Формируем новую вершину стека, складывая логически строки 3 и 5 (рисунок 5.4к).

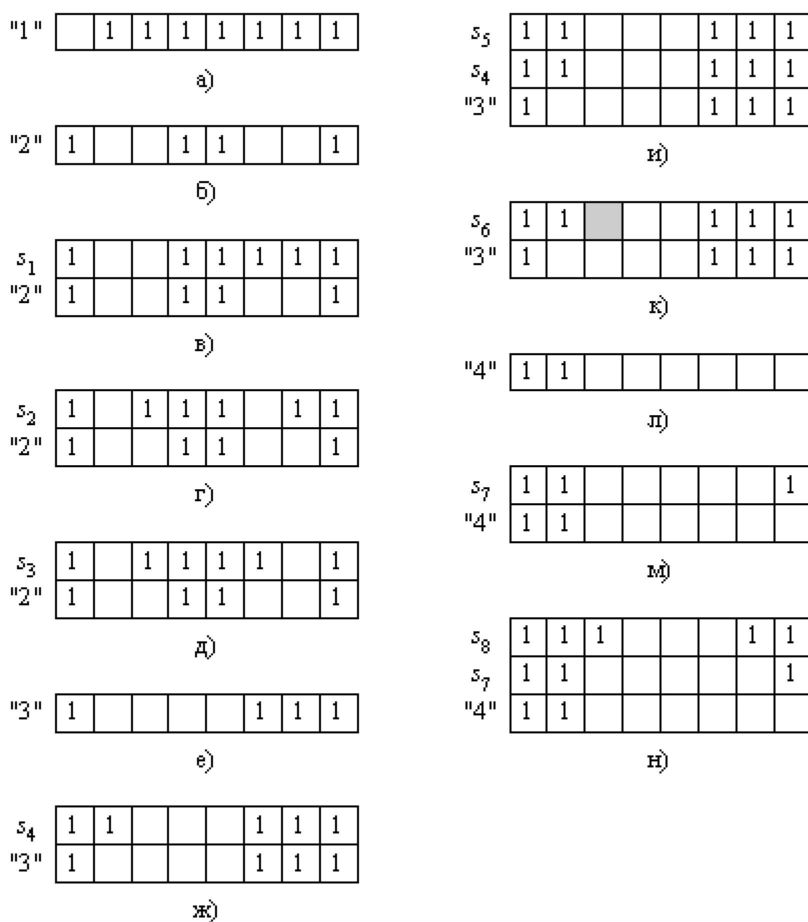


Рисунок 5.4. Пример работы алгоритма нахождения полных множеств ВНО. Начало

Проверяем полученную строку на наличие нулей правее позиции 5. Таких нулей нет. Казалось бы, можно формировать очередное полное множество ВНО {3, 5}, но не будем торопиться. Проверим множество {3, 5} на полноту. Для этого достаточно проверить строку s_6 на наличие «дополнительных» нулей (т.е. нулей, позиции которых не совпадают с номерами из найденного множества ВНО), левее позиции 5. В строке s_6 есть такой «дополнительный» нуль – в позиции 4. Это означает, что множество ВНО {3, 5} не является полным и его необходимо отбросить.

Следует обратить внимание на то, что проверку на полноту необходимо проводить лишь после получения «полного» множества ВНО, т.е. когда все нули правее последнего оператора из множества ВНО найдены. Если такую проверку проводить не после получения множества ВНО, а во время, то, к сожалению, будут полностью потеряны все возможные полные множества ВНО, которые могли бы получиться в результате дальнейших действий с данной строкой.

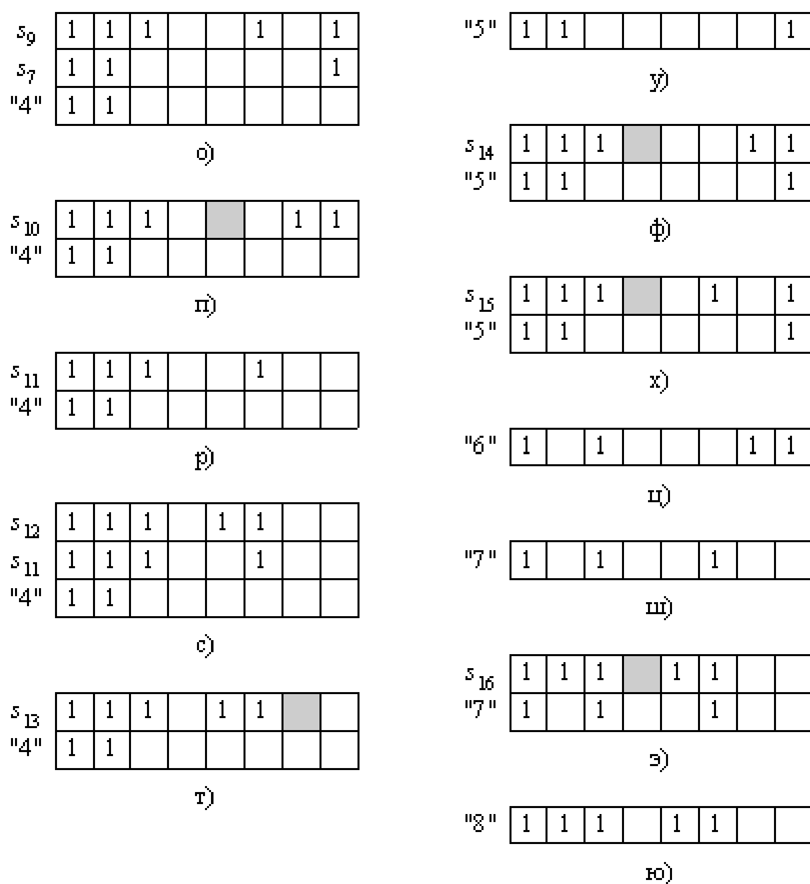


Рисунок 5.4. Окончание

Исключаем строку s_6 из стека. Дальнейший просмотр строки 3 не выявил новых нулей, поэтому исключаем и её из стека. Поскольку стек пуст, то помещаем туда следующую строку – 4 (рисунок 5.4л).

Начинаем просмотр строки 4 с позиции 5. Первый нуль как раз находится в позиции 5. Складываем логически строки 4 и 5 (рисунок 5.4м).

В получившейся строке s_7 имеются нули правее позиции 5. Найденный нуль занимает позицию 6. Формируем новую строку $s_8 = s_7 \vee "6"$. Тогда стек будет иметь вид, изображённый на рисунке 5.4н.

Строка в вершине стека содержит нули только в позициях, соответствующих операторам, «участвующим» в её формировании. Значит, найдено ещё одно (шестое по счёту) полное множество ВНО {4, 5, 6}.

Убираем из стека строку s_8. Следующий найденный нуль в строке s_7 находится в позиции 7. Складываем соответствующие строки (s_7 и 7). Получаем новую строку в вершине стека и весь стек в виде, показанном на рисунке 5.4о.

Строка s_9 содержит нули только в позициях 4, 5 и 7. Это значит, что найдено очередное полное множество ВНО {4, 5, 7}.

Выгружаем строку s_9 из стека. Строка s_7 так же исчерпана, поэтому исключаем и её. Следующий найденный нуль в строке 4 занимает позицию 6. Формируем новую вершину стека, складывая строки 4 и 6 (рисунок 5.4п).

Получившаяся строка не содержит нулей правее позиции 6. Однако экзамен на полноту тоже не выдерживает: в позиции 5 появился «дополнительный нуль». Это значит, что найденное множество ВНО {4, 6} не является полным (т.е. комбинация операторов 4 и 6 уже встречалась с какими-то другими операторами, образующими полное множество ВНО) и его необходимо отбросить.

Исключаем строку s_{10} из стека. Следующий найденный нуль в строке 4 находится в позиции 7. Формируем новую строку $s_{11} = "4" \vee "7"$ и помещаем её в вершину стека (рисунок 5.4р).

Ищем нули во вновь образованной строке правее позиции 7. Такой нуль есть – он находится в позиции 8. Формируем строку $s_{12} = s_{11} \vee "8"$ и помещаем её в вершину стека. Стек будет выглядеть так, как показано на рисунке 5.4с.

Строка s_{12} содержит нули только в позициях 4, 7 и 8. Это означает, что найдено восьмое полное множество ВНО {4, 7, 8}.

Убираем строку s_{12} из стека. Строка s_{11} так же исчерпана, поэтому исключаем её из стека. В стеке остаётся единственная 4-ая строка. Последний найденный нуль находится в позиции 8. Формируем новую вершину стека, складывая строки 4 и 8 (рисунок 5.4т).

В получившейся строке обнаруживаем «дополнительный» нуль левее 8-ой позиции, но не совпадающий с позицией 4. Это значит, что найденное множество ВНО {4, 8} не является полным и его необходимо исключить из рассмотрения.

Исключаем строку s_{13} из стека. Строка 4 пройдена до конца, следовательно, её так же нужно исключить из стека. Заносим в стек следующую – 5-ую строку матрицы независимости M (рисунок 5.4у).

Строка 5 содержит нуль в позиции 6. Формируем новую вершину стека $s_{14} = "5" \vee "6"$. Стек будет выглядеть так, как представлено на рисунке 5.4ф.

В строке s_{14} нет нулевых элементов, правее позиции 6. Проверим найденное множество ВНО на полноту. Выясняем, что в позиции 4 строки s_{14} есть «дополнительный» нуль – найденное множество ВНО не полное, поэтому отбрасываем его.

Выгружаем строку s_{14} из стека. Следующий нуль в строке 5 занимает позицию 7. Складываем логически строки 5 и 7 и формируем новую вершину стека – строку s_{15} (рисунок 5.4х).

Анализируя строку s_{15}, замечаем, что эта строка содержит «дополнительный» нуль левее позиции 7, который не совпадает с позицией 5. Делаем вывод, что множество ВНО {5, 7} не является полным и отбрасываем его.

Исключаем строку s_{15} из стека. Строка 5 оказывается пройдена до конца, поэтому исключаем её. Помещаем в стек строку 6 (рисунок 5.4ц).

В этой строке нет нулевых элементов правее позиции 6. Возникает ситуация, похожая на ту, что складывалась на шаге 1. Проверим, является ли множество ВНО {6} полным. В отличие от шага 1, данное множество не является полным т.к. в строке 6 существуют другие нули, помимо позиции 6. Отбрасываем это множество.

Убираем строку 6 из стека и загружаем туда строку 7 (рисунок 5.4ш).

Строка 7 содержит нуль в позиции 8, следовательно, формируем новую строку $s_{16} = "7" \vee "8"$ и помещаем её в вершину стека (рисунок 5.4э).

Обнаруживаем «дополнительный» нуль в строке s_{16} в позиции 4. Это говорит о том, что множество ВНО {7, 8} не является полным и его необходимо отбросить.

Выгружаем из стека строку s_{16}. Все комбинации операторов с участием оператора 7 исчерпаны. Исключаем строку 7 из стека. Загружаем последнюю 8-ю строку в стек (рисунок 5.4ю).

В восьмой строке (как и в шестой) имеются нули левее позиции 8. Это значит, что множество ВНО {8} не является полным и его необходимо отбросить.

Таким образом, найдено 8 полных множеств ВНО: {1}, {2,3}, {2,6}, {2,7}, {3,4,5}, {4,5,6}, {4,5,7}, {4,7,8}.

Вопросы для самопроверки

1. Дайте определение полных множеств ВНО.
2. С какой целью строится максимально полное множество?
3. Что называется взаимной независимостью операторов?
4. Как строится матрица независимости?

Задание на лабораторную работу

1. Необходимо дополнить разработанные ранее программы следующими функциями: расчет и отображение отраженной матрицы S'. Расчет и отображение матрицы независимости *M*. Нахождение максимально полного множества ВНО.

2. Продемонстрировать работающую программу преподавателю и получить отметку о её выполнении.

3. Сохранить копию программы для выполнения последующих лабораторных работ на дискете.

4. Оформить отчет о проделанной работе.

Содержание отчета

1. Цель работы.
2. Ответы на вопросы для самопроверки.
3. Схемы алгоритмов и их описание.
4. Распечатки экранных форм, полученных в результате работы программы.
5. Анализ полученных результатов.

5. Лабораторная работа №6. Определение ранних и поздних сроков окончания выполнения операторов и оценка снизу требуемого количества процессоров и времени решения задачи на ВС

Цель работы – ознакомление с понятиями ранних и поздних сроков выполнения операторов. Практическая реализация алгоритмов нахождения этих сроков и оценки минимального числа процессоров и времени выполнения задачи, представленной заданным графом алгоритма. Вычисление оценок снизу требуемого количества процессоров и времени решения задачи, представленной заданным графом.

Теоретическая часть

Рассмотрим алгоритм, представляемый информационным графом (без связей по управлению), не имеющим контуров. Тогда очевидно, что момент окончания выполнения любого из операторов не может быть меньше максимальной из длин всех путей, заканчивающихся вершиной, соответствующей этому оператору. Таким образом, для каждого оператора, $j = 1, \ldots, m$ алгоритма можно найти ранний срок t_j^1 окончания его выполнения.

Если окончание выполнения алгоритма ограничено временем $T \geq T_{кр}$, то для каждого оператора можно найти и поздний срок окончания его выполнения $t_j^2(T)$. Здесь $T_{кр}$ (критическое) – максимальная характеристика пути в графе со скалярными весами, и определяет минимальное время, за которое может быть решена данная задача.

Окончание выполнения любого оператора позже этого позднего срока приводит к тому, что все последующие за ним операторы не смогут быть выполнены в заданный срок T, поэтому без задания T определение поздних сроков не имеет смысла. При $T = T_{кр}$ ранние и поздние сроки выполнения операторов, входящие в критический путь, совпадают.

Далее приводятся алгоритмы нахождения ранних t_j^1 и поздних $t_j^2(T)$ сроков окончания выполнения операторов алгоритма, заданного матрицей следования S, где $j = 1, \ldots, RS$.

Алгоритм 6.1. Нахождение ранних сроков окончания выполнения операторов.

1. Положим $t_j^1 := 0$, где $j = 1, \ldots, RS$.

2. Просматриваются строки матрицы S сверху вниз, выбирается первая необработанная строка матрицы и осуществляется переход к следующему шагу. Если обработаны все строки, то – конец алгоритма.

3. Пусть выбрана j-я строка, не содержащая единичных элементов, далее вычисляется $t_j^1 = P_j$, где P_j – вес j-го оператора, затем выполняется переход на шаг 5.

4. Если j-я строка содержит единичные элементы, то вычисляется $t_j^1 = \max\left\{t_{j_v}^1\right\} + P_j$, где $\left\{t_{j_v}^1\right\}$ есть множество времен, которым соответствует единица в данной строке, и выполняется переход на шаг 5. Если во множестве $\left\{t_{j_v}^1\right\}$ есть нулевые элементы, то выполняется шаг 6.

5. Обработанная j-я строка исключается из рассмотрения и осуществляется переход на шаг 2.

6. Если найдена строка j_v, для которой $t_{j_v}^1 = 0$, то вычисляется строка $j = j_v$, и осуществляется переход на шаг 3.

Конец алгоритма.

Примечание: пункт 6 используется для нетреугольной матрицы S.

Алгоритм 6.2. Получение поздних сроков окончания выполнения операторов.

1. Положим $t_j^2 := 0$, где $j = 1, \ldots, RS$.

2. Просматриваются столбцы матрицы S справа налево, выбирается первый необработанный столбец матрицы и производится переход к следующему шагу. Если обработаны все столбцы, то – конец алгоритма.

3. Пусть j – номер очередного необработанного столбца, если он не содержит единичных элементов, то вычислим $t_j^2(T) = T$, где T – время решения задачи, и переходим на шаг 5.

4. Если столбец j содержит единичные элементы, то вычисляется $t_j^2(T) = \min\left\{t_{j_v}^2 - P_{j_v}\right\}$, т.е. минимум определяется по всем j_v единичным элементам j-го столбца. Если $t_{j_v}^2(T) = 0$, то выполняется шаг 6.

5. Обработанный j-й столбец исключаем из рассмотрения, затем выполняется шаг 2.

6. Если найден столбец j_v, для которого $t_{j_v}^2(T) = 0$, то производится поиск необработанного столбца j_v, вычисляется $j = j_v$ и выполняется переход на шаг 3.

Конец алгоритма.

Примечание: пункт 6 используется для не треугольной матрицы S.

Диаграммы выполнения операторов для ранних и поздних сроков – удобный способ наглядно представить многопроцессорную обработку. Всего в диаграмме имеется n строк, соответствующих числу процессоров в системе. Выполнение того или иного оператора отмечается прямоугольниками, имеющими длину, равную весам операторов, и правую границу, соответствующую крайнему сроку окончания выполнения операторов.

Рассмотрим пример получения ранних и поздних сроков окончания выполнения операторов для графа, представленного на рисунке 6.1.

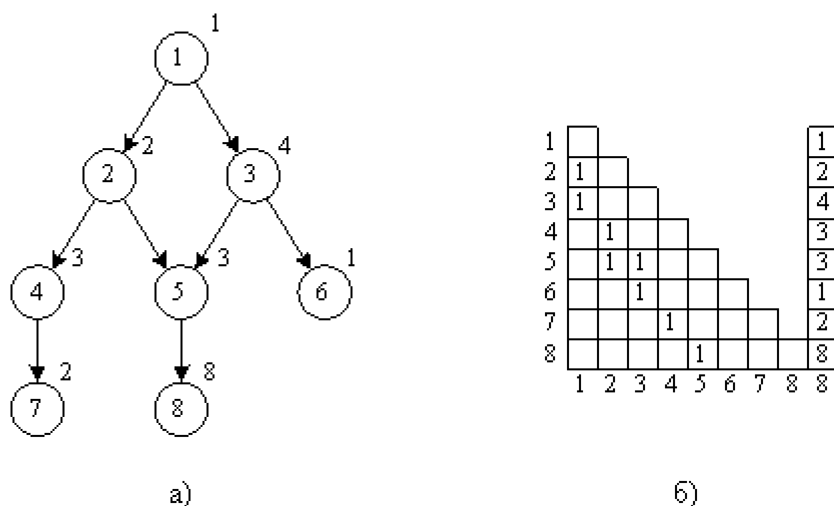


Рисунок 6.1. Пример информационного графа (а) и его матрицы следования (б), используемого для иллюстрации вычисления ранних и поздних сроков окончания выполнения операторов

Ранние сроки будут выглядеть следующим образом:

$t_1^1 = 1,\ t_2^1 = 1 + 2 = 3,\ t_3^1 = 1 + 4 = 5,\ t_4^1 = 3 + 3 = 6,\ t_5^1 = \max\{3{,}5\} + 3 = 8,$

$t_6^1 = 5 + 1 = 6,\ t_7^1 = 6 + 2 = 8,\ t_8^1 = 8 + 8 = 16.$

Поздние сроки (при $T = 18$):

$t_8^2(18) = 18,\ t_7^2(18) = 18,\ t_6^2(18) = 18,\ t_5^2(18) = 18 - 8 = 10,$

$t_4^2(18) = 18 - 2 = 16,\ t_3^2(18) = \min\{18 - 1{,}10 - 3\} = 7,$

$t_2^2(18) = \min\{10 - 3{,}16 - 3\} = 7,\ t_1^2(18) = \min\{7 - 4{,}7 - 2\} = 3.$

Диаграммы выполнения операторов для вычисленных ранних и поздних сроков окончания выполнения операторов показаны на рисунке 6.2.

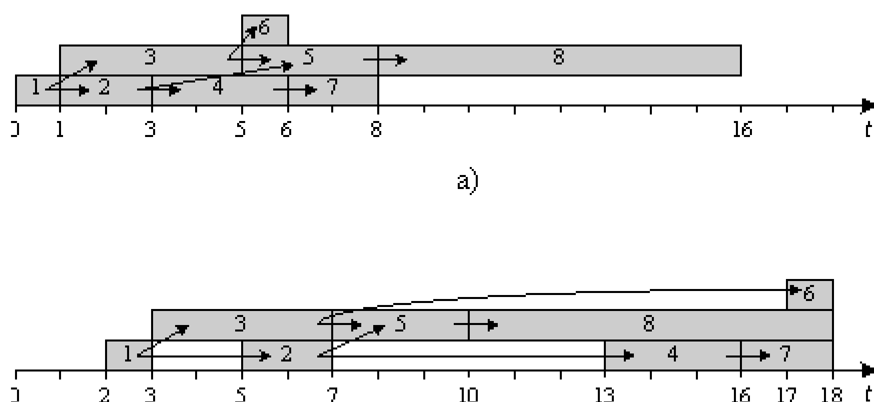


Рисунок 6.2. Временные диаграммы ранних (а) и поздних (б) сроков окончания выполнения операторов

Определение 6.1. *Множество входных вершин графа G называется минорантой графа G.*

Определение 6.2. *Множество выходных вершин графа G называется мажорантой графа G.*

Пусть A есть миноранта , B – мажоранта графа G, а p_j – вес j-го оператора, тогда множество значений сроков окончания выполнения операторов определяется следующими неравенствами:

$$t_j - p_j \geq 0, \text{ если } \quad j \in A; \tag{6. 1}$$

$$t_j - p_j \geq t_i, \text{ если существует } \quad \text{связь } i \to j, \; i \in X \setminus B, \;\; j \in X \setminus A; \tag{6. 2}$$

$$t_j \leq T, \text{ если } \quad j \in B. \tag{6. 3}$$

Множество значений, определяемых неравенствами (8 – 1) – (8 – 3), задает многоугольник M_T в RS-мерном пространстве: $(t_1, ..., t_{RS}) \in M_T$, тогда справедливо следующее определение.

Определение 6.3. *Функция* $PZ(t_1, ..., t_{RS}, \tau) = \sum_{j=1}^{RS} OP(t_j, \tau)$, *где*

$OP(t_j, \tau) = \begin{cases} 1, & \tau \in (t_j - p_j, t_j) \\ 0, & \tau \notin (t_j - p_j, t_j) \end{cases}$ *называется плотностью загрузки ВС в точке* τ *для значения* $t_1, ..., t_{RS}$.

Значение функции *PZ* в каждый момент времени формируется операторами множества ВНО, т.е. в каждый момент времени значение функции *PZ* совпадает с числом одновременно выполняемых операторов.

Определение 6.4. *Функция* $Z(t_1,..., t_{RS}, a, b) = \int_a^b PZ(t_1,..., t_{RS}, \tau) d\tau$ *называется загрузкой отрезка* $[a,b] \subset [0,T]$ *для* $(t_1,..., t_{RS}) \in M_T$.

Функция *Z* определяет количество выполненных на этом отрезке операторов (с учётом частично выполненных операторов).

Определение 6.5. *Функция* $Z^{(T)}(a,b) = \min\ Z(t_1,..., t_{RS}, a, b)$ *называется минимальной загрузкой отрезка* $[a,b] \subset [0,T]$ *для* $(t_1,..., t_{RS}) \in D_T$.

Смысл этого определения заключается в том, что при любом планировании операторов на выполнение при решении задачи за время T загрузка отрезка $[a,b] \subset [0,T]$ не может быть меньше вычисленной, согласно определению 6.5, величины.

Для составления алгоритма вычисления данной функции введем функцию $\xi(x)$:

$$\xi(x) = \begin{cases} x & \text{при} \quad x \ge 0, \\ 0 & \text{при} \quad x < 0. \end{cases}$$

Алгоритм 6.3. Вычисление функции $Z^{(T)}(a,b)$.

1. С помощью алгоритмов 6.1 и 6.2 вычисляются ранние t_j^1 и поздние $t_j^2(T)$ сроки окончания выполнения операторов.

2. Полагаем $Z^{(T)}(a,b) = 0$.

3. Анализируем последовательность оператора $j = 1,..., RS$. Если просмотрены все операторы, то конец алгоритма.

4. Вычислим

$$Z^{(T)}(a,b) = Z^{(T)}(a,b) + \min\left\{\xi(t_j^1 - a), \xi(b - t_j^2(T) + \tau_j), \tau_j, b - a\right\}. \qquad (6.4)$$

5. После перебора всех операторов получаем значение $Z^{(T)}(a,b)$

Конец алгоритма.

Лемма 6.1. «Об оценке сверху требуемого количества процессоров для решения задачи за время *T*».

Минимальное количество однородных процессоров N, способных выполнить данный алгоритм за время $T \geq T_{кр}$*, не превышает* $\mathrm{E} = \max_i \{C_i\}$, $i = 1,..., l$*, где* C_i *– число операторов, входящих в i-ое полное множество ВНО, полученное для информационного графа G, соответствующего исследуемому алгоритму.*

Следствие. При $N = E$ время решения данного алгоритма $T = T_{кр}$.

Примечание. Получаемое количество процессоров N на основании этой леммы является верхней оценкой требуемого количества процессоров (т.е. для решения данной задачи требуется не более N процессоров).

Теорема 6.1. «Об оценке снизу числа процессов, необходимых для решения задачи за время T».

Для того чтобы N процессоров было достаточно для выполнения заданного алгоритма, представленного информационным графом со скалярными весами вершин за время T, необходимо, чтобы для отрезка $[a,b] \subset [0,T]$ *выполнялось соотношение:*

$$N \geq \frac{Z^{(T)}(a,b)}{b-a},$$

где $Z^{(T)}(a,b)$ *– минимальная загрузка отрезка [a,b].*

Теорема 6.2. «Об оценке снизу времени выполнения задачи при заданном количестве процессоров».

Для того, чтобы T было наименьшим временем выполнения алгоритма, представленного информационным графом со скалярными весами вершин вычислительной системой, состоящей из N процессоров, необходимо, чтобы для отрезка $[a,b] \subset [0,T]$ *выполнялось соотношение:*

$$Z^{(T)}(a,b) \leq N(b-a),$$

где $Z^{(T)}(a,b)$ *– минимальная загрузка отрезка [a,b].*

Теорема 6.3. «Об уточнении оценки снизу времени выполнения задачи на N процессорах».

Если T_1 *– оценка снизу времени выполнения алгоритма, представленного информационным графом со скалярными весами вершин на ВС, имеющей N процессоров, и на отрезке* $[a,b] \subset [0,T]$ *выполняется соотношение* $Z^{(T)}(a,b) - N(b-a) = d > 0$*, тогда наименьшее время T реализации алгоритма удовлетворяет соотношению*

$$T \geq T_1 + \frac{d}{N}.$$

Алгоритм 6.4. Оценка минимального числа процессоров, необходимого для выполнения алгоритма за время T.

1. Положим $N := 0$.
2. Последовательно перебираем интервалы $[a,b] \subset [0,T]$ в порядке:

$$[0,1];$$
$$[0,2];\ [1,2];$$
$$[0,3];\ [1,3];\ [2,3];$$
$$\ldots$$
$$[0,T];\ [1,T];\ \ldots\ [T-1,T].$$

Всего отрезков: $\frac{T(T+1)}{2}$.

3. Для очередного интервала $[a,b]$ вычислим $N_1 = \frac{Z^T(a,b)}{(b-a)}$, где $Z^{(T)}(a,b)$ определяется по алгоритму 6.3.
4. Если $N_1 > N$, то $N := N_1$.
5. После обработки всех интервалов, получается требуемое N.

Конец алгоритма.

Алгоритм 6.5. Оценка минимального времени T выполнения заданного алгоритма на ВС, содержащей N процессоров.

1. Вычислим $T := \max\left\{ \left] \frac{1}{N} \sum_{i=1}^{RS} p_i \right[, T_{кр} \right\}$, где $]x[$ – ближайшее к x целое, не меньшее x, p_i – вес i-го оператора.
2. Просматриваются интервалы $[a,b] \subset [0,T]$, как в алгоритме 6.4 пункте 2.

Примечание. При таком выборе последовательности отрезков значение T можно увеличивать, не пересчитывая при этом ранее вычисленные значения d.

3. Для очередного интервала $[a,b]$ вычислим значение $d = Z^{(T)}(a,b) - N(b-a)$, величина $Z^{(T)}(a,b)$ вычисляется по алгоритму 6.3.
4. Если $d > 0$, вычислим $T := T + \left] \frac{d}{N} \right[$.

5. Вычислим $t_j^2(T) := t_j^2(T) + \left]\frac{d}{N}\right[$, $j = 1,..., RS$.

6. После обработки всех интервалов вычисляем значение T – нижнюю оценку минимального времени выполнения данного алгоритма на данной ВС.

Конец алгоритма.

Вопросы для самопроверки

1. Что характеризует ранние и поздние сроки выполнения операторов?
2. Почему для нахождения поздних сроков требуется задать время выполнения задачи?
3. При совпадении $T_{кр}$ с временем T ранние или поздние сроки выполнения каких операторов совпадают?
4. Почему при вычислении функции $Z(T)$ нужно выбирать минимальное значение из разности соответствующих времен в формуле (6.4)?
5. Какой смысл имеют величины a и b при вычислении функции $Z(T)$?
6. На каком временном интервале функция $OP\left(t_j, \tau\right)$ принимает значение 1?
7. Почему количество требуемых процессоров не может быть больше числа операторов, входящих во множество ВНО в рассматриваемый момент времени?
8. В чём смысл леммы 1, в каких целях его можно использовать?
9. Почему минимальная загрузка на рассматриваемом интервале $[a,b]$ может сравниваться с выражением $n \cdot (a, b)$.

Задание на лабораторную работу

1. Необходимо дополнить имеющуюся программу следующими функциями:
   - нахождения ранних и поздних сроков окончания выполнения операторов, отображения их в виде таблицы-дополнения одновременно с матрицей S и весами операторов;
   - отображения полученных результатов на диаграммах выполнения работ;
   - вычисления оценки снизу требуемого числа процессоров и оценки снизу времени решения поставленной задачи.
2. Продемонстрировать работающую программу преподавателю и получить отметку о ее выполнении.
3. Оформить отчет о проделанной работе.

Содержание отчета

1. Цель работы.
2. Ответы на вопросы для самопроверки.
3. Схемы алгоритмов и их описание.
4. Распечатки экранных форм, полученных в результате работы программы.
5. Анализ полученных результатов.

7. Лабораторная работа №7. Запуск параллельных программ на кластере

Цель работы – практическая реализация и отладка простейших параллельных программ "Hello world" и вычисления числа π на кластере. Работа с очередью заданий на кластере с помощью планировщика IBM Tivoli LoadLeveler.

Теоретическая часть

Кластером обычно называют параллельную или распределенную систему, состоящую из набора взаимосвязанных целостных компьютеров, которые используются как единый, унифицированный вычислительный ресурс. Кластеры, имеющие преимущества в сравнении с более традиционными системами, применяют для решения задач повышенной сложности.

На кластере в МГТУ им. Н.Э.Баумана установлена ОС Linux – RHEL. Удаленное управление ОС можно выполнять с помощью протокола SSH (англ. Secure Shell — «безопасная оболочка») — сетевой протокол сеансового уровня, позволяющий производить удалённое управление операционной системой и туннелирование TCP-соединений (например, для передачи файлов). SSH допускает выбор различных алгоритмов шифрования, что позволяет не только реализовать удаленную работу, но и передавать файлы по шифрованному каналу. SSH-клиенты и SSH-серверы доступны для большинства сетевых операционных систем. Рассмотрим возможные способы подключения по SSH:

Linux

- SSH-клиент и используемая им библиотека SSL входят в комплект поставки большинства Linux-дистрибутивов. При необходимости установка из исходных кодов возможна без прав суперпользователя (аналог пользователя с административными правами в Windows) в домашнюю директорию /home/username. Исчерпывающую информацию по настройке и применению команд ssh и scp можно получить, обратившись к встроенной справке:

```
$> man ssh
$> man scp
```

Windows

- Существует несколько реализаций SSH для Windows. Одним из ssh-клиентов является PuTTY (см. тж. PuTTY Portable). При его настройке (раздел «Session») в поле «Host Name» нужно указать интернет-адрес системы, к которой вы хотите подключиться, а переключатель «Connection type» выставить в положение «SSH».

Чтобы не вводить при каждом подключении имя пользователя, с которым вы зарегистрированы в системе (например, myname), его можно указать через символ '@' в том же поле «Host Name» (например, myname@<интернет-адрес>).

- Для копирования файлов с локальной машины на удаленную (в данном случае кластер) можно использовать программу WinSCP (см. тж. WinSCP Portable), которая предоставляет удобный графический интерфейс. При работе в терминале будет полезна утилита pscp, входящая в пакет PuTTY. Синтаксис аргументов ее командной строки аналогичен таковому у Linux-команды scp.
- Кроме того, можно установить Linux-подобное окружение Cygwin, в котором уже имеются программы ssh, scp
- Если вы применяете Linux из виртуальной машины типа VirtualBox, то часто можете непосредственно воспользоваться всеми преимуществами Linux-окружения: встроенными ssh, scp, и X-сервером. (Однако этот механизм может не сработать, если ваш провайдер предоставляет интернет-доступ через VPN-подключение; в таком случае необходимо отредактировать таблицы маршрутизации и/или указать в конфигурационных файлах адреса локальных DNS-серверов, но подобная настройка выходит за пределы рассматриваемых здесь вопросов.)

На компьютерах в лаборатории ИУ-6 установлена ОС Ubuntu 8. Далее рассмотрим подключение к кластеру с использованием данной ОС. Вход в систему произвести для пользователя student (пароль: student). Терминал для ввода команд можно открыть из графической оболочки в меню *Applications (Приложения)*, раздел *Accessories* (*Стандартные* в русифицированной версии) или сочетанием клавиш Ctrl+Alt+F1 (выход из консоли — Ctrl+Alt+F7). В данном случае проще работать через графическую оболочку. В ОС доступны несколько терминалов, необходимо будет открыть два. Один будет использоваться для удаленной работы с Linux RHEL, другой – с локальной ОС (понадобится при копировании файлов на кластер).

1. Подключение к кластеру по ssh

```
student@805-21:~$ ssh username@195.19.33.110
```

где 195.19.33.110 ip адрес кластера.

Логин и пароль для доступа к кластеру необходимо получить у преподавателя перед началом лабораторной. *Username здесь и далее следует заменить на Ваше имя пользователя!* Если логин введен верно, то через несколько секунд последует запрос пароля:

```
username@195.19.33.110's password:
```

Если авторизация успешна, то появится строка с указанием текущей директории и времени и именем пользователя:

```
[11:55 username@mgmt ~]$
```

где ~ означает домашнюю директорию home.

2. Для запуска программ нужно, чтобы пользователь находился в группе loadl. Это можно проверить командой :

```
[11:55 username@mgmt ~]$groups username
```

Отклик системы содержит имя пользователя и имя группы(должно быть loadl).

С точки зрения конечного пользователя файловая система вычислительного кластера состоит из двух частей:

- /home — домашние директории, предназначенные для хранения пользовательских данных, разработки и компиляции программ, постобработки результатов расчетов; узлы ввода-вывода не имеют доступа к этой части файловой системы
- /gpfs — высокопроизводительная часть файловой системы, к которой имеют непосредственный доступ узлы ввода-вывода; служит для хранения временных файлов, необходимых для счета (исполняемых файлов и данных)

В ОС Linux для переходов по каталогу используется команда cd. Для обозначения корня ОС используется символ '/', текущая директория - './', предыдущая директория - '../'.

3. Создание папки для проекта в домашней директории.

```
cd $HOME
mkdir projectname
```

В папке projectname будут хранится все файлы, используемые в лабораторной работе. Для просмотра содержимого папки используется команда ls.

```
[11:55 username@mgmt ~]$ls
```

ответ:

```
laba7  code1   Dragovich  hello  mbox   PRJ7
bin    common  EXAMPLES   laba7  Popov  test
```

Справку по команде всегда можно получить командой man.

```
man ls
```

Просмотр текстовых файлов можно осуществить встроенным текстовым редактором nano. Нихжняя строка содержит описание доступных команд.

```
nano filename
```

4. Копирование файлов

Файлы hello.c и hello.job создать на локальной машине с помощью текстового редактора, скопировав текст файлов из данного методического указания (см. ниже). Необ-

ходимо сохранить в созданной папке (см. п.3) файлы с кодом программ и заданий для IBM LoadLeveler. Для этого **на локальной машине** необходимо выполнить команду scp.

```
            student@805-21:~$scp hello.c username@195.19.33.110:projectname/
            student@805-21:~$scp hello.job
username@195.19.33.110:projectname/
```

Файл задания *hello.job* выглядит следующим образом:

#!/bin/bash

#Название задания (может включать в себя любую комбинацию букв и цифр

#@job_name = hello

#Тип задания: может быть последовательный или параллельный.

#Выберем параллельный тип задания

#@job_type = MPICH

#@class = small_mpi

#@group = loadl

#STDIN для задания – имя файла или /dev/null по умолчанию

#@ input = /dev/null

#STDOUT для задания (выходные данные)

#По умолчанию: /dev/null

*#@ output = /gpfs/home/iu6/**username/projectname**/hello.$(user).$(jobid).stdout*

#STDERR для задания (вывод ошибок)

*#@ error = /gpfs/home/iu6/**username/projectname**/hello.$(user).$(jobid).stderr*

*#@ initialdir = /gpfs/home/iu6/**username/projectname/***

#notification - Specifies when the user specified in the notify_user keyword is sent mail.

#Syntax: notification = always|error|start|never|complete

#@ notification = complete

#node – Минимальное и максимальное число узлов, необходимое шагу задания.

#Syntax: node = [min][,max]

#@ node = 32

#@ tasks_per_node = 2

#node_usage – Показывает разделяет ли данный шаг задания узлы с другими шагами.

#Syntax: node_usage = shared | not_shared

#@ node_usage = shared

#Необходимо выражение qieue, помещающее шаг задания в очередь

```
#@ queue
echo "-----------------------------------------------------"
echo LOADL_STEP_ID = $LOADL_STEP_ID
echo HOSTNAME: `hostname`
echo "-----------------------------------------------------"
mpdboot -r ssh -n `cat $LOADL_HOSTFILE|sort|uniq|wc -l` -f $LOADL_HOSTFILE
mpirun -r ssh -machinefile $LOADL_HOSTFILE -n $LOADL_TOTAL_TASKS ./hello.c.exe
100000
mpdallexit
```

Замените в файле задания пути для выходного файла и файла ошибок для Вашего пользователя и названия папки проекта! Файл должен быть сохранен в кодировке UTF-8 (Обратить на это внимание при редактировании файла в текстовом редакторе ОС Windows). Можно изменить файл уже после копирования в ОС кластера с помощью текстового редактора nano.

Ниже следует текст MPI-программы *hello.c*, которая выводит на экран приветствия от каждого процесса.

```
/* hello.c */
#include <stdio.h>
/* Необходимо подключить для MPI_* функций */
#include "mpi.h"
int main(int argc, char **argv) {
  int rank;
  char host[150];
  int namelen;
  /* Инициализация MPI. Тут передаются специфические для командной строки
mpich аргументы */
  MPI_Init(&argc, &argv);
  /* Получение номера процесса. Номер rank сохраняется в переменную 'rank' */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  /* Узнаем на котором компьютере запущено. Сохраним имя в 'host' */
  MPI_Get_processor_name(host,&namelen);
  printf("Hello world (Rank: %d / Host: %s)\n", rank, host);
  fflush(stdout);
  /* Заключение: Закрываем соединения к остальным дочерним процессам,
очистка памяти
   * где была расположена библиотека MPI и т д */
  MPI_Finalize();
  return 0;
}
```

5. Компиляция программы

```
mpicc -o hello.c.exe hello.c
```

или

```
mpicc -o hello.c.exe hello6.c -lm
```

6. Постановка программы в очередь

```
/gpfs/LoadL/bin/llsubmit ./hello.job
```

или

```
llsubmit hello.job
```

При успешной постановке в очередь последует отклик с номером Вашего задания:

```
llsubmit: The job "mgmt.nodes.489" has been submitted.
```

Для запуска задач на кластере служит команда *mpirun*. Но в интерактивном режиме она используется очень редко и только привилегированными группами пользователей. На многопользовательских системах одновременно несколько пользователей, обладающих разными приоритетами (студенты, аспиранты, сотрудники, …), отправляют на счет задачи, предъявляющие различные требования к вычислительным ресурсам (число процессоров, максимальное время счета, …). Т.е., как правило, проходит некоторое время между отправкой задачи и ее постановкой на счет. Именно поэтому использование *mpirun* в интерактивном режиме с frontend-серверов пользователями системы невозможно. Для управления заданиями применяется специальная программа, называемая планировщиком задач, или диспетчером очереди, которая внутри себя и вызывает *mpirun*. На кластере установлен планировщих задач IBM Tivoli LoadLeveler. Чтобы поставить задание в очередь, необходимо подготовить специальный командный файл, в котором будет указан путь к исполняемому файлу и запрашиваемые программой ресурсы.

Полный список и подробное описание команд планировщика можно найти в официальной документации, здесь же мы приведем только самые необходимые:

llsubmit — постановка задания в очередь

llq — просмотр очереди заданий

llcancel — удаление задания из очереди

llhold — приостановка/продолжение продвижения задания в очереди

llmodify — изменение параметров задания

llprio — изменение приоритета задания

llclass — информация о классах заданий

llstatus — информация о машине

Краткую справку о ключах командной строки можно получить, использовав параметр -H, например:

```
llsubmit -H
```

7. Просмотр результатов работы программы

Во-первых, проверим очередь заданий, чтобы убедиться, что задание запущено:

```
llq
```

Отклик:

```
Id              Owner      Submitted   ST PRI Class        Running On
------------ ---------- ----------- -- --- ------------ -----------
mgmt.490.0    barsic     10/1  11:57 R  50  small_mpi    n1107
mgmt.456.0    barsic      9/21 17:04 H  50  small
2 job step(s) in queue, 0 waiting, 0 pending, 1 running, 1 held, 0 preempted
```

В выведенной таблице выводится следующая информация:

Id	Идентификатор задания: имя_машины.номер_задания.номер_шага
Owner	Идентификатор пользователя, от имени которого запускается задание
Submitted	Дата и время постановки задания в очередь, причем дата записана в североамериканском формате (т.е. 10/6 соответствует 6-му октября)
ST	Состояние задания (здесь приведены не все возможные значения): • R — задание выполняется в данный момент («Running») • ST — задание запускается в данный момент («Starting») • P — задание переведено в процессе запуска («Pending») • I — задание ожидает своей очереди на запуск («Idle») • H — прохождение задания приостановлено пользователем («User Hold») • NQ — задание не готово к запуску («Not Queued») • C — задание завершено («Completed»)
PRI	Приоритет задания
Class	Класс задания
Running On	Имя машины, на которой работает задание; пусто, если задание еще не запущено

Если необходимо удалить задание из череди, воспользуйтесь командой

```
llcancel mgmt.490.0
```

или

```
llcancel 490
```

После удаления задания из очереди все задания пользователя будут заблокированы (статус H=held). Для продолжения работы использовать команду llhold для всех заданий пользователя:

```
llhold -r -u username
```

или для конкретного задания:

```
llhold -r mgmt.490.0
```

Далее необходимо проверить содержимое файлов *.stdout и *.stderr с помощью редактора nano. Содержимое при успешном выполнении будет следующим:

```
-------------------------------------------------------
LOADL_STEP_ID = mgmt.nodes.497.0
HOSTNAME: n2313.nodes
-------------------------------------------------------
Hello world (Rank: 0 / Host: n2313)
Hello world (Rank: 1 / Host: n2313)
Hello world (Rank: 20 / Host: n1112)
Hello world (Rank: 21 / Host: n1112)
Hello world (Rank: 30 / Host: n3207)
Hello world (Rank: 31 / Host: n3207)
```

и т. д.

Файл ошибок не будет содержать записей.

Используя навыки работы с кластером, полученные при работе с программой hello необходимо запустить программу вычисления числа π.

Программа вычисляет число π с помощью нахождения значений интеграла:

$$\int_{-1/2}^{1/2} 4/(1+x\cdot x)\,dx$$

Интеграл представляется как сумма n интервалов. Апроксимация интеграла на каждом интервале:

(1/n)*4/(1+x*x)

Главный процесс (порядок 0) запрашивает у пользователя число интервалов и затем должен передать это число остальным процессам. Каждый процесс затем вычисляет n-ый интервал (x = -1/2+rank/n, -1/2+rank/n+size/n,...). В результате, суммы вычисленные каждым процессом суммируются вместе.

Код программы (*.c файл):

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"

double Func1(double a) {
  return (4.0 / (1.0 + a*a));
}

int main(int argc, char *argv[]) {
  int done = 0, n, myid, numprocs, i;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x;
  double startwtime, endwtime;
  int  namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc,&argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);

    fprintf(stderr,"Process %d on %s\n", myid, processor_name);
    fflush(stderr);
      n = 0;
    while (!done)
    {
      if (myid == 0)
      {
        printf("Enter the number of intervals: (0 quits) ");fflush(stdout);
        n = 6;
        startwtime = MPI_Wtime();
      }
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      if (n == 0)
        done = 1;
      else
      {
        h   = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs)
        {
          x = h * ((double)i - 0.5);
          sum += Func1(x);
        }
        mypi = h * sum;

        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
  MPI_COMM_WORLD);
         if (myid == 0)
        {
          printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
          fflush(stdout);
          endwtime = MPI_Wtime();
          printf("wall clock time = %f\n", endwtime-startwtime);
          fflush(stdout);
        }
      }
    }
    MPI_Finalize();
          return 0;
  }
```

Для запуска программы на кластере необходимо создать фал задания. Взяв за основу файл задания hello.job нужно ввести в него необходимые изменения:

1) название проекта: `#@job_name = pi`

2) выходные файлы *.stdout и *.stderr должны иметь другое название (заменить hello на pi)

3) для выполнения программы будет достаточно 6 узлов, по одному заданию на каждый:

```
#@node = 6
#@tasks_per_node = 1
```

4) количество повторений запуска программы равно 1 вместо 100000, как было для hello (исправить параметр для функции mpirun).

Успешное выполнение файла записывает в *.stderr номера участвующих процессов, а в *.stdout — время выполнения и значение числа π.

```
Process 0 on n1102
Process 62 on n3312
```

и т. д.

```
-------------------------------------------------------
LOADL_STEP_ID = mgmt.nodes.503.0
HOSTNAME: n1102.nodes
-------------------------------------------------------
pi is approximately 3.1416009869231232, Error is 0.0000083333333301
wall clock time = 3.091967
pi is approximately 3.1416009869231232, Error is 0.0000083333333301
wall clock time = 0.002079
```

и т. д.

В этом алгоритме нет большого количества передач сообщений. Поэтому время соединений незначительно и у нас практически линейное увеличение скорости (при 6 процессах на 6 разных машинах увеличение в среднем около 5.8).

Иногда кластер выдаёт следующие ошибки в stdout:

```
-------------------------------------------------------
LOADL_STEP_ID = mgmt.nodes.128.0
HOSTNAME: n1103.nodes
-------------------------------------------------------
mpdboot_n1103.nodes (handle_mpd_output 703): Failed to establish a socket
connection with n2307.nodes:40557 : (111, 'Connection refused')
mpdboot_n1103.nodes (handle_mpd_output 720): failed to connect to mpd on
n2307.nodes
WARNING: Can't read mpd.hosts for list of hosts, start only on current
mpiexec: unable to start all procs; may have invalid machine names
remaining specified hosts:
172.22.12.12 (n1212.nodes)
172.22.22.13 (n2213.nodes)
…
mpdallexit: cannot connect to local mpd (/tmp/mpd2.console_stud101); possi-
ble causes:
1. no mpd is running on this host
2. an mpd is running but was started without a "console" (-n option)
```

Чаще всего это означает, что необходимо уменьшить число используемых узлов #@node и заданий, выполняемых каждым узлом #@tasks_per_node.

Попробуйте изменить значения в файле задания *.job слудующим образом:

```
#@node = 1
#@tasks_per_node = 4
```

Вопросы для самопроверки

1. Какой командой можно получить исполняемый код параллельной программы?
2. Как запустить параллельную программу на кластере?
3. Для чего необходим файл задания IBM LoadLeverer?
4. Где находятся результаты работы программы?
5. Как посмотреть статус запущенного задания?

Задание на лабораторную работу

1. Необходимо скомпилировать и запустить на кластере две параллельных программы "Hello world" и вычисления числа π. Продемонстрировать результат работы программ преподавателю и получить отметку о выполнении работы.
2. Оформить отчет о проделанной работе.

Содержание отчета

1. Цель работы.
2. Ответы на вопросы для самопроверки.
3. Схемы алгоритмов запуска заданий, самих задач и их описание.
4. Распечатки файлов вывода, полученных в результате работы программы.
5. Анализ полученных результатов.

8. Лабораторная работа №8. Написание программ с использованием библиотеки MPI

Цель работы – изучение структуры MPI-программ и практическая реализация программ.

Теоретическая часть

Основная схема программы MPI подчиняется следующим общим шагам:

1. Инициализация для коммуникаций
2. Коммуникации распределения данных по процессам
3. Выход "чистым" способом из системы передачи сообщений по завершении коммуникаций

MPI имеет свыше 125 **функций**. Тем не менее, начинающий программист обычно может иметь дело только с шестью функциями, которые перечислены ниже:

- Инициализация для коммуникаций

  MPI_Init инициализирует окружение MPI

  MPI_Comm_size возвращает число процессов

  MPI_Comm_rank возвращает номер текущего процесса (ранг = номер по-порядку)

- Коммуникации распределения данных по процессами

  MPI_Send отправляет сообщение

  MPI_Recv получает сообщение

- Выход из системы передачи сообщений

  MPI_Finalize

Сообщения MPI состоят из двух основных частей: отправляемые/получаемые данные, и сопроводительная информация (записи на конверте /оболочке/), которая помогает отправить данные по определенному маршруту. Обычно существуют три вызываемых параметра в вызовах передачи сообщений MPI, которые описывают данные и три других параметра, которые определяют маршрут:

Сообщение = данные (3 параметра) + оболочка (3 параметра)

Данные определяют информацию, которая будет отослана или получена. Оболочка (конверт) используется в маршрутизации сообщения к получателю и связывает вызовы отправки с вызовами получения.

Коммуникаторы гарантируют уникальные пространства сообщений. В соединении с группами процессов их можно использовать, чтобы ограничить коммуникацию к подмножеству процессов.

Для С, общий **формат фактических вызовов**, используемых MPI, имеет вид

rc = MPI_Xxxxx(parameter, ...)

Заметим, что регистр здесь важен. Например, MPI должно быть заглавным, так же как и первая буква после подчеркивания. Все последующие символы должны быть в нижнем регистре. Переменная rc --- есть некий код возврата, имеющий целый тип. В случае успеха, он устанавливается в MPI_SUCCESS.

Программа на С должна включать файл *"mpi.h"*. Он содержит определения для констант и функций MPI. В приложении 8.5 приводятся тексты программ на языке Си с использованием библиотеки MPI. Получите у преподавателя номер образца задачи, которую Вам надо выполнить на кластере и выполнить требуемое задание.

Вопросы для самопроверки

1. Какую структуру имеет простая MPI-программа?
2. Какие основные функции реализует библотека MPI?
3. Как организовать передачу сообщений между процессами?
4. Для чего используются коммуникаторы?

Задание на лабораторную работу

1. Необходимо разработать MPI-программы в соответствии с таблицей вариантов заданий (Таблица 8.1):

- Передачи сообщений с буферизацией;
- С использованием таймера;
- С определением структуры проходящего сообщения;
- С анализом тупиковых ситуаций при обмене;
- Определяющую базовые характеристики коммуникационной сети: латентность;
- Определяющую базовые характеристики коммуникационной сети: максимально достижимую пропускную способность и длину сообщений, на которой она достигается;
- Приёма-передачи сообщений без блокировки;
- Реализацию отложенных запросов на взаимодействие;
- Или другую на свой выбор, использующую функциональные возможности библиотеки MPI.

Таблица 8.1

№ вари-анта	*№ примеров, необходимых к реализации*	*№ варианта*	*№ примеров, необходимых к реализации*
1	*13, 18*	9	*15*
2	*1, 20*	10	*9*
3	*2, 16*	11	*7*
4	*3, 11*	12	*4, 6*
5	*12*	13	*5, 17*
6	*14*	14	*6, 11*
7	*19*	15	*1, 8*
8	*4,17*	16	*2, 10*

2. Продемонстрировать работающую программу преподавателю и получить отметку о ее выполнении.
3. Оформить отчет о проделанной работе.

Содержание отчета

1. Цель работы.
2. Ответы на вопросы для самопроверки.
3. Схемы алгоритмов и их описание.
4. Распечатки файлов вывода, полученных в результате работы программы.
5. Анализ полученных результатов.

Приложение. Список вариантов образцов решаемых задач.

Пример простейшей *MPI-программы* на языке Си выглядит следующим образом:

Пример 1:

```
#include  <stdio.h>
#include   "mpi.h"
void main(int argc,char * argv[])
{
int ierr;
printf("Before MPI_Init ...\n");
ierr=MPI_Init(&argc,&argv) ;
printf("Parallel section\n");
ierr=MPI_Finalize ();
```

```
printf("After MPI_Finalize\n") ; }
```

В зависимости от реализации MPI строчки *«Before Mpi_Init»* и *«After Mpi_Finalize»* может печатать либо один выделенный процесс, либо все запущенные процессы приложения. Строчку *«Parallel section»* должны напечатать все процессы. Порядок вывода строк с разных процессов может быть произвольным.

В следующем примере каждый запущенный процесс печатает свой уникальный номер в коммуникаторе *mpi_comm_world* и число процессов в данном коммуникаторе.

Пример 2:

```
#include <stdio.h>
#include "mpi.h"
void main(int argc,char * argv[])
{
int ierr,size,rank;
MPI_Init(&argc,&argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD, &size);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Process %i size %i\n",rank,size);
ierr=MPI_Finalize();
}
```

Строка, соответствующая вызову процедуры printf, будет выведена столько раз, сколько процессов было порождено при запуске программы. Порядок появления строк заранее не определен и может быть, вообще говоря, любым. Гарантируется только то, что содержимое отдельных строк не будет перемешано друг с другом.

Определение характеристик таймера

В этой программе на каждом процессе определяются две характеристики системного таймера: его разрешение и время, требуемое на замер времени (для усреднения получаемого значения выполняется *NTIMES* замеров). Также в данном примере показано использование процедуры MPI_*GET_PROCESSOR_NAME*.

Пример 3:

```
#include  <stdio.h>
#include   "mpi.h"
void main(int argc,char * argv[])
{
int   n,ierr,rank,len,i,NTIMES=100;
char   name[MPI_MAX_PROCESSOR_NAME];
double  time_start,time_finish,tick;
```

```
MPI_Init(&argc,&argv);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
ierr=MPI_Get_processor_name(name, &len);
name[len]=0;
tick=MPI_Wtick();
time_start=MPI_Wtime();
for (n = 0; n<NTIMES; n++) time_finish=MPI_Wtime();
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Processor %s, process %i: tick = %d, time = %d\n",name,rank,tick,(time_finish-
time_start)/(double)NTIMES);
ierr=MPI_Finalize(); }
```

Приём и передача сообщений

Передача сообщения использована буферизация. Для буферизации выделяется массив *buf*, после завершения пересылки он освобождается. Размер необходимого буфера определяется размером сообщения (одно целое число – 4 байта) плюс значение константы *MPI_BSEND_ OVERHEAD.*

Пример 4:

```
#include <stdio.h>
#include "mpi.h"
void main(int argc,char * argv[])
{
const int BUFSIZE=MPI_BSEND_OVERHEAD+4;
unsigned char buf[BUFSIZE];
int rank,ierr,ibufsize,rbuf;
struct MPI_Status status;
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) {
        ierr=MPI_Buffer_attach(buf,BUFSIZE);
        ierr=MPI_Bsend(&rank,1,MPI_INT,1,5,MPI_COMM_WORLD);
        // sending variable rank
        ierr=MPI_Buffer_detach(&buf, &BUFSIZE);
        }
if (rank==1)
{  ierr=MPI_Recv(&rbuf,1,MPI_INT,0,5,MPI_COMM_WORLD,&status);
printf("Process 1 received %i from process %i\n",rbuf, status.MPI_SOURCE);
}
ierr=MPI_Finalize();
}
```

Ниже приведен пример программы, в которой нулевой процесс посылает сообщение процессу с номером один и ждет от него ответа. Если программа будет запущена с большим числом процессов, то реально выполнять пересылки все равно станут только нулевой и первый процессы. Остальные процессы после их инициализации процедурой

MPI__Init напечатают начальные значения переменных *a* и *b*, после чего завершатся, выполнив процедуру *MPI_FINALIZE*.

Пример 5:

```
#include <stdio.h>
#include "mpi.h"
void main(int argc,char * argv[])
{
int ierr,size,rank;
float a,b;
MPI_Status status;
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD, &size);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
a=b=0;
if (rank==0)
{ ierr=MPI_Send(&b,1,MPI_FLOAT,1,5,MPI_COMM_WORLD);
ierr=MPI_Recv(&a,1,MPI_FLOAT,1,5,MPI_COMM_WORLD,&status);
} else
if (rank==1)
{
a=2;
ierr=MPI_Recv(&b,1,MPI_FLOAT,0,5,MPI_COMM_WORLD,&status);
ierr=MPI_Send(&a,1,MPI_FLOAT,0,5,MPI_COMM_WORLD);
printf("Process %i a = %lf b = %lf\n",rank,a,b); }
ierr=MPI_Finalize();
}
```

В следующем примере каждый процесс с четным номером посылает сообщение своему соседу с номером на единицу большим. Дополнительно поставлена проверка для процесса с максимальным номером, чтобы он не послал сообщение несуществующему процессу. Значения переменной b изменятся только на процессах с нечетными номерами.

Пример 6:

```
#include <stdio.h>
#include "mpi.h"
void main(int argc,char * argv[])
{
int ierr,size,rank,a,b;
struct MPI_Status status;
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD, &size);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
a=rank;
b=-1;
```

```
if (rank%2==0)
{
if (rank+1<size)
ierr=MPI_Send(&a, 1, MPI_INT, rank+1, 5, MPI_COMM_WORLD);
}
else ierr=MPI_Recv(&b,1,MPI_INT,rank-1,5,MPI_COMM_WORLD,&status);
printf ("Process %i a = %i b = %i\n",rank,a,b) ;
ierr=MPI_Finalize (); }
```

Задание: Измените данную программу так, чтобы нулевой процесс не посылал сообщений и запустите её с числом процессов, равным номеру вашего варианта.

Процессы приложения в приводимой ниже программе разбиваются на две непересекающиеся примерно равные группы *groupl* и *group2*. При нечетном числе процессов в группе *group2* может оказаться на один процесс больше, тогда последний процесс из данной группы не должен обмениваться данными ни с одним процессом из группы *groupl*. С помощью вызовов процедуры *MPI_Group_translate_ranks* каждый процесс находит процесс с тем же номером в другой группе и обменивается с ним сообщением через коммуникатор *MPl_COMM_WORLD* при помощи вызова процедуры *MPI_Sendrecv*. В конце программы не нужные далее группы уничтожаются с помощью вызовов процедур *MPI_GROUP_FREE*.

Пример 7:

```
#include <stdio.h>
#include "mpi.h"
void main(int argc,char * argv[])
{
int ierr,rank,i,size,size1;
const int n=1000;
int a[4],b[4];
int ranks[128], rank1, rank2, rank3;
MPI_Status status;
MPI_Group group,group1,group2;
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD,&size);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank) ;
ierr=MPI_Comm_group(MPI_COMM_WORLD, &group);
size1=size/2;
for (i=0;i<size1;i++)
ranks[i]=i;
ierr=MPI_Group_incl(group,size1,ranks, &group1);
ierr=MPI_Group_excl(group,size1,ranks, &group2);
ierr=MPI_Group_rank (group1, &rank1);
ierr=MPI_Group_rank(group2,&rank2);
if (rank1==MPI_UNDEFINED) // we are in second half
{ if (rank2<size1)
```

```
ierr=MPI_Group_translate_ranks(group1,1,&rank2,group,&rank3); else
rank3=MPI_UNDEFINED;
}
else  // in first half
ierr=MPI_Group_translate_ranks(group2,1,&rank1,group,&rank3);
a[0]=rank;
a[1]=rank1;
a[2]=rank2;
a[3]=rank3;
if (rank3!=MPI_UNDEFINED)
ierr=MPI_Sendrecv(a,4,MPI_INT,rank3,1,b,4,MPI_INT,rank3,1,MPI_COMM_WORL
D,&status);
ierr=MPI_Group_free(&group) ;
ierr=MPI_Group_free(&group1);
ierr=MPI_Group_free(&group2);
printf ( "Process %i a=[",rank);
for ( i=0;i<4;i++) printf("%i ",a[i]);
 printf ("] b=[");
for ( i=0;i<4;i++) printf ("%i ",b[i]);
 printf("]\n");
 ierr=MPI_Finalize();
}
```

Задание: Измените программу так, чтобы передаваемые сообщения содержали номер передающего процесса и ваше имя.

Определения структуры приходящего сообщения

Иллюстрация применения процедуры *MPI_Probe* ДЛЯ определения структуры приходящего сообщения. Процесс *0* ждет сообщения от любого из процессов *1* и *2* с одним и тем же тегом. Однако посылаемые этими процессами данные имеют разный тип. Для того чтобы определить, в какую переменную помещать приходящее сообщение, процесс сначала при помощи вызова *MPI_Probe* определяет, от кого же именно поступило это сообщение. Следующий непосредственно после *MPI_Probe* вызов *MPI_Recv* гарантированно примет нужное сообщение, после чего принимается сообщение от другого процесса.

Пример 8:

```
#include <stdio.h>
#include "mpi.h"
void main(int argc,char * argv[])
{
int rank,ierr,ibuf;
struct MPI_Status status;
float rbuf;
```

```
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
ibuf=rank;
rbuf=1.0F*rank;
switch (rank)
{
case 1:
{
ierr=MPI_Send(&ibuf,1,MPI_INT,0,5,MPI_COMM_WORLD);
break;
}
case 2:
{
ierr=MPI_Send(&rbuf,1,MPI_FLOAT,0,5,MPI_COMM_WORLD); break;
}
case 0:
{
ierr=MPI_Probe(MPI_ANY_SOURCE,5,MPI_COMM_WORLD,&status);
if (status.MPI_SOURCE==1)
// message from process 1 received first
{ ierr=MPI_Recv(&ibuf,1,MPI_INT,1,5, MPI_COMM_WORLD,&status);
ierr=MPI_Recv(&rbuf,1,MPI_FLOAT,2,5, MPI_COMM_WORLD,&status);
} else // message from process 2 received first
 if (status.MPI_SOURCE==2) // ignore messages from other processes
{ ierr=MPI_Recv(&rbuf,1,MPI_FLOAT, 2,5,MPI_COMM_WORLD,&status);
  ierr=MPI_Recv(&ibuf,1,MPI_INT, 1,5,MPI_COMM_WORLD,&status);}
printf("Process 0 recv %i from process 1 %1f from process 2\n",ibuf,rbuf);
break;
}
}
ierr=MPI_Finalize();
}
```

Задание: Добавьте в программу процесс 3, передающий в качестве сообщения процессу 0 ваше имя.

Определение базовых характеристик коммуникационной сети

Приведена программа, в которой моделируется последовательный обмен сообщениями между двумя процессами, замеряется время на одну итерацию обмена, определяется зависимость времени обмена от длины сообщения. Таким образом, определяются базовые характеристики коммуникационной сети параллельного компьютера: латентность (время на передачу сообщения нулевой длины) и максимально достижимая пропускная способность (количество мегабайт в секунду) коммуникационной сети, а также длина сообщений, на которой она достигается. Константа *NMAX* задает ограничение на максимальную длину посылаемого сообщения, а константа *NTIMES* определяет количе-

ство повторений для усреднения результата. Сначала посылается сообщение нулевой длины для определения латентности, затем длина сообщений удваивается, начиная с посылки одного элемента типа float[8].

Пример 9:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc,char * argv[])
{
	int ierr,rank,size,i,n,nmax,lmax,NTIMES=10;
	//const int NMAX=1000000;
	const int NMAX=1000;
	double time_start,time,bandwidth,max;
	float a[NMAX*8]; // equal to real*8  a[NTIMES] at fortran
	struct MPI_Status status;
	ierr=MPI_Init(&argc,&argv);
	//ierr=MPI_Comm_size(MPI_COMM_WORLD, &size);
	ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	//time_start=MPI_Wtime();
	n=0;
	nmax=lmax=0;
	while (n<=NMAX)
	{
		time_start=MPI_Wtime();
		for (i=0;i<NTIMES;i++)
			{
			if (rank==0)
				{
				ierr=MPI_Send(&a,8*n,MPI_FLOAT, 1, 1,
MPI_COMM_WORLD) ;
				ierr=MPI_Recv(&a, 8*n, MPI_FLOAT, 1, 1,
MPI_COMM_WORLD, &status) ;
				} else
			if (rank==1)
				{

	ierr=MPI_Recv(&a,8*n,MPI_FLOAT,0,1,MPI_COMM_WORLD,&status) ;

	ierr=MPI_Send(&a,8*n,MPI_FLOAT,0,1,MPI_COMM_WORLD);
				}
			}
		time=(MPI_Wtime()-time_start)/(2*NTIMES); // this is time for one way trans-
action
		bandwidth=((double)8*n*sizeof(float))/ (1024*1024)/time;
		if (max<bandwidth)
			{max=bandwidth;
			lmax=8*n*sizeof(float); }
		if (rank==0)
			{
```

```
                        if (!n) printf("Latency = %10.4d seconds\n", time);
                              else printf("%i bytes sent, bandwidth = %10.4d MB/s\n",
8*n*sizeof (float) ,bandwidth);
                        }
                  if (n==0) n=1; else n*=2;
      }
      // Finally print maximum bandwidth
      if (rank==0) printf("Max bandwidth = %10.4d MB/s, length = %i bytes\n",max,lmax);
      ierr=MPI_Finalize();
      return 0;
}
```

Задание: Определите, насколько изменяется результат измерений, если количество количество повторений NTIMES увеличивать. Измените программу так, чтобы процесс 1 выводил NTIMES, при котором были получены результаты, выводимые процессом 0.

Анализ тупиковых ситуаций при обмене

Процессы обмена сообщениями с ближайшими соседями в соответствии с топологией кольца при помощи неблокирующих операций показаны в фрагменте нижеприведенной программы. Заметим, что использование для этих целей блокирующих операций может привести к возникновению тупиковой ситуации.

Пример 10:

```
#include  <stdio.h>
#include   "mpi.h"

void main(int argc,char * argv[])
{
int i,ierr,rank,size,prev,next,buf[2];
MPI_Request reqs[4];
MPI_Status status[4];
double a[5];
int N=5;
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD, &size);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) prev=size-1; else prev=rank-1;
if (rank==size-1) next=0; else next=rank+1;
ierr=MPI_Irecv(&buf[0],1,MPI_INT,prev,5,MPI_COMM_WORLD,&reqs[0]);
ierr=MPI_Irecv(&buf[1],1,MPI_INT,next,6,MPI_COMM_WORLD, &reqs[1]);
ierr=MPI_Isend(&rank,1,MPI_INT,prev,6,MPI_COMM_WORLD,&reqs[2]);
ierr=MPI_Isend(&rank,1,MPI_INT, next,5, MPI_COMM_WORLD, &reqs[3]);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
ierr=MPI_Waitall (4 , reqs, status) ;
if (rank<size-1){
printf ( "process %i prev=%i next=%i\n",rank,buf[0],buf[1]);
i++;
}
```

ierr=MPI_Finalize();
}

Задание: Измените программу так, чтобы в передаваемых сообщениях содержалась строка с номером процесса (например, “one”, “two” и т.д.).

Схема использования процедуры *MPI_Waitsome* демонстрируется в следующем примере.

Пример 11:

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
   int rank, size;
   int i, index[4], count, remaining;
   int buffer[400];
   MPI_Request request[4];
   MPI_Status status[4];

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (size > 4)
   {
      printf("Please run with 4 processes.\n");fflush(stdout);
      MPI_Finalize();
      return 1;
   }
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if (rank == 0)
   {
      for (i=0; i<size * 100; i++)
          buffer[i] = i/100;
      for (i=0; i<size-1; i++)
      {
          MPI_Isend(&buffer[i*100], 100, MPI_INT, i+1, 123, MPI_COMM_WORLD,
&request[i]);
      }
      remaining = size-1;
      while (remaining > 0)
      {
          MPI_Waitsome(size-1, request, &count, index, status);
          if (count > 0)
          {
             printf("%d sends completed\n", count);fflush(stdout);
             remaining = remaining - count;
          }
```

```
    }
  }
  else
  {
    MPI_Recv(buffer, 100, MPI_INT, 0, 123, MPI_COMM_WORLD, &status[0]);
    printf("%d: buffer[0] = %d\n", rank, buffer[0]);fflush(stdout);
  }

  MPI_Finalize();
  return 0;
}
```

Задание: Измените программу так, чтобы при успешном завершении всех передач сообщений процесс 0 посылал остальным процессам сообщение “SENDS COMPLETED” и они, получив получив это сообщение, выводили его.

Организация передачи-приёма сообщений без блокировки

Демонстрация применениея неблокирующих операций для реализации транспонирования квадратной матрицы, распределенной между процессами по строкам, приведена ниже. Сначала каждый процесс локально определяет *nl* строк массива *a.* Затем при помощи неблокирующих операций *MPI_Isend* и *MPI_Irecv* инициализируются все необходимые для транспонирования обмены данными. На фоне начинающихся обменов каждый процесс транспонирует свою локальную часть массива *a*. После этого процесс при помощи вызова процедуры *MPI_Waitany* дожидается прихода сообщения от любого другого процесса и транспонирует полученную от данного процесса часть массива *a*. Обработка продолжается до тех пор, пока не будут получены сообщения от всех процессов. В конце исходный массив *a* и транспонированный массив *b* распечатываются.

Пример 12:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define N 9
double a[N][N],b[N][N];
void work(int n,int nl,int size,int rank)
{
int ierr,i,j,ii,jj,ir,irr;
const int MAXPROC=64;
char *a,*b,c;
MPI_Status status;
MPI_Request req1[MAXPROC] ;
MPI_Request req2[MAXPROC] ;
```

```
// creating arrays
a=(char*) malloc(nl*n);
b=(char*) malloc(nl*n);
//ok
for (i=1;i<nl;i++)
        for (j=1;j<n;j++)
        {
        ii=i+rank*nl;
        if (ii<=n) a[i*n+j]=100*ii+j;
        }
for (ir=0;ir<size-1;ir++)
        if (ir!=rank)
                ierr=MPI_Irecv(&b[i*n+ir*nl+1],nl*nl,MPI_DOUBLE,
ir,MPI_ANY_TAG,MPI_COMM_WORLD,&req1[ir] ) ;
req1[rank]=MPI_REQUEST_NULL;
for (ir=0;ir<size-1;ir++)
        if (ir!=rank)
                ierr=MPI_Isend(&a[i*n+ir*nl+1],nl*nl,MPI_DOUBLE,ir,
6,MPI_COMM_WORLD, &req2[ir] ) ;
ir=rank;
for (i=1;i<nl;i++)
        {
        ii=i+ir*nl;
        for (j=i+1;j<nl;j++)
                { jj=j+ir*nl;
                b[i*n+jj]=a[j*n+ii];
                b[j*n+ii]=a[i*n+jj]; }
        b[i*n+ii] =a[i*n+ii];
        }
for (irr=1;irr<size-1;irr++)
        {
        ierr=MPI_Waitany(size-1,req1,&ir, &status) ;
if (rank==0) printf("Process ir %i nl %i\n",ir,nl); fflush(stdout);
        //ir=ir-1;
        for (i=1;i<nl;i++)
                {
                ii=i+ir*nl;
                for (j=i + 1;j<nl;j++)
                        {
                        jj=j+ir*nl;
if (rank==0) printf("i,j %i,%i \n",i*n+jj,j*n+ii);fflush(stdout);
                        c=b[i*n+jj];
                        b[i*n+jj]=b[j*n+ii];
                        b[j*n+ii]=c;}
                }
        }
for (i=1;i<nl;i++)
for (j=1;j<N;j++)
        { ii=i+rank*nl;
        if (ii<=n)
```

```
                printf("Process %i : a (%i,%i)= %4.4d,
b(%i,%i)=%4.4d\n",rank,ii,j,a[i*n+j],ii,j , b[i*n+j]);
        fflush(stdout);
        }
// deleting arrays
free(a);
free(b);
}

int main(int argc,char * argv[])
{
int ierr,rank,size,nl,i, j;
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD, &size) ;
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank) ;
nl=(N-1)/size+1;
work(N,nl,size,rank);
printf("OK\n");
        fflush(stdout);
//ierr=MPI_Finalize();
return 0;
}
```

Задание: Измените программу так, чтобы она проверяла, является ли введённая матрица симметрической.

В следующем примере операции двунаправленного обмена с соседними процессами в кольцевой топологии производятся при помощи двух вызовов процедуры *MPI_Sendrecv*. При этом гарантированно не возникает тупиковой ситуации.

Пример 13:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc,char * argv[]) {
int ierr,rank,size,prev,next,buf[2];
MPI_Status status1,status2;
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD, &size);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
prev=rank==0?size-1:rank-1;
next=rank==size-1?0:rank+1;
ierr=MPI_Sendrecv(&rank,1,MPI_INT,prev,6,&buf[1],1,MPI_INT,next,
6,MPI_COMM_WORLD,&status2);
ierr=MPI_Sendrecv(&rank,1,MPI_INT,next,5,&buf[0],1,MPI_INT,prev,
5,MPI_COMM_WORLD,&status1);
printf("Process %i prev=%i next = %i\n",rank,buf[0],buf[1]);
ierr=MPI_Finalize();
return 0;}
```

Реализация отложенных запросов на взаимодействие

В следующем примере функциональность процедуры *MPI_Barrier* моделируется при помощи отложенных запросов на взаимодействие. Для усреднения результатов производится *NTIMES* операций обмена, в рамках каждой из них все процессы должны послать сообщение процессу с номером *0*, после чего получить от него ответный сигнал, означающий, что все процессы дошли до этой точки в программе. Использование отложенных запросов позволяет инициализировать посылку данных только один раз, а затем использовать на каждой итерации цикла. Далее время на моделирование сравнивается со временем на синхронизацию при помощи самой стандартной процедуры *MPI_Barrier*.

Пример 14:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc,char * argv[])
{
int ierr,rank,size,i,it;
const int MAXPROC=128, NTIMES=400;
int ibuf[MAXPROC];
double time_start,time_finish;
MPI_Request req[2*MAXPROC];
MPI_Status statuses[MAXPROC];
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD,&size);
ierr=MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank==0) // process number 0
{ for (i=1;i<size;i++)
        { ierr=MPI_Recv_init(&ibuf[i],1,MPI_INT,i,5,MPI_COMM_WORLD, &req[i-
1]) ;
        ierr=MPI_Send_init(&rank,1,MPI_INT,i,6,MPI_COMM_WORLD,&req[size +
i-1] ) ;
        }
time_start=MPI_Wtime();
for (it=0;it<NTIMES;it++)
        {
        // waiting receive
        ierr=MPI_Startall(size-1, req) ;
                ierr=MPI_Waitall(size-1, req,statuses);
                // waiting send
        ierr=MPI_Startall(size-1, &req[size]) ;
                ierr=MPI_Waitall(size-1, &req[size],statuses);
        }
        }
else // not 0 process
{
        ierr=MPI_Recv_init(&ibuf[1],1,MPI_INT,0,6,MPI_COMM_WORLD,&req[1]);
```

```
ierr=MPI_Send_init(&rank,1,MPI_INT,0,5,MPI_COMM_WORLD, &req[2] ) ;
time_start=MPI_Wtime();
          for (it=0;it<NTIMES;it++)
          {
                    // waiting receive
          ierr=MPI_Start(&req[2]);
          ierr=MPI_Wait(&req[2],statuses);
          // remember, statuses==&statuses[0]
          // waiting send
          ierr=MPI_Start(&req[1]);
          ierr=MPI_Wait(&req[1],statuses);
                    }
}
 time_finish = MPI_Wtime()-time_start ;
 printf("Rank = %i barrier time = %d\n",rank, (double)(time_finish)/NTIMES) ;
 ierr=MPI_Finalize();
return 0;}
```

Задание: Измените программу так, чтобы ответные сообщение процесса 0, содержали в себе количество времени, затраченного на получение сообщения от процесса.

Сравнительная оценка различных способов обмена данными

Ниже производится сравнение операции глобального суммирования при помощи схемы сдваивания с использованием пересылок данных типа точка-точка. Эффективность такого моделирования сравнивается с использованием коллективной операции *MPI_Reduce.*

Пример 15:

```
#include <stdio.h>
#include "mpi.h"
#define n 1000000
int main(int argc,char * argv[])
{
int ierr,rank,size,nproc,i;
double time_start,time_finish;
double a[n],b[n],c[n];

MPI_Status status;
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD,&size);
ierr=MPI_Comm_rank(MPI_COMM_WORLD,&rank);
nproc=size;
for (i=1;i<=n;i++) a[i-1]=i/(double)size; // in c indexing starts from 0;
ierr=MPI_Barrier(MPI_COMM_WORLD) ;
time_start=MPI_Wtime();
```

```
for   (i=0;i<n;i++) c[i]=a[i] ;
while   (nproc>1)
{
if (rank<nproc/2)
{
ierr=MPI_Recv(b,n,MPI_DOUBLE,nproc-rank-1,1,MPI_COMM_WORLD,&status);
for   (i=0;i<n;i++) c[i]+=b[i];
}
else
if (rank<nproc)
ierr=MPI_Send(c,n,MPI_DOUBLE,nproc-rank-1,1,MPI_COMM_WORLD);
nproc/=2;
}
for (i=0;i<n;i++) b[i]=c[i];
time_finish=MPI_Wtime() - time_start;
if (rank==0)
printf("model b[1]=%d rank=%i model time=%d  \n", b[1],rank,time_finish);
for (i=1;i<=n;i++) a[i-1]=i/(double)size;
ierr=MPI_Barrier(MPI_COMM_WORLD);
time_start=MPI_Wtime();
ierr=MPI_Reduce(a,b,n,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
time_finish=MPI_Wtime() - time_start;
if (rank==0)
printf("reduce b[1]=%d rank=%i reduce time=%d \n", b[1],rank,time_finish);
fflush(stdout);
ierr=MPI_Finalize ();
return 0;}
```

Задание: Проверьте насколько изменится время выполнения операции глобального суммирования, если изменить тип массивов на float.

Использование глобальных операций в MPI

Следующий пример демонстрирует задание пользовательской функции для использования в качестве глобальной операции. Задается функция *smod5*, вычисляющая поэлементную сумму по модулю *5* векторов целочисленных аргументов. Данная функция объявляется в качестве глобальной операции *op* в вызове процедуры *MPI_Op_create*, затем используется в процедуре *MPI_Reduce*, после чего удаляется с помощью вызова процедуры *MPI_OP_FREE*.

Пример 16:

```
#include <stdio.h>
#include "mpi.h"
int smod5(int * a,int * b,int cnt,int type)
{
int i;
```

```
for ( i=0;i<cnt;i++)
b[i] = (b[i]+a[i] )%5; return  MPI_SUCCESS;
}
int main(int argc,char * argv[])
{
int ierr,rank,i,op;
const int n=1000;
int a[n] , b[n] ;
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank) ;
for (i=0;i<n;i++) a[i]=i+rank+1; // in fortran first elementh will contain 1 // addi-tion
of 1 makes similar result in C
printf ("Process %i, a[0]=%i\n" , rank, a[0] ) ;
ierr=MPI_Op_create((MPI_User_function *)smod5,1, &op) ;
ierr=MPI_Reduce(a,b,n,MPI_INT,op,0,MPI_COMM_WORLD);
ierr=MPI_Op_free (&op) ;
if (rank==0) printf("b[0]=%i\n",b[0]);
ierr=MPI_Finalize () ;
return 0;
}
```

Задание: Проверьте насколько дольше выполняется операция op над векторами a и b, чем операция глобального суммирования MPI_SUM.

Взаимодействие процессов в MPI

С помощью следующей программы создается один новый коммуникатор *comm_revs*, в который входят все процессы приложения, пронумерованные в обратном порядке. Когда коммуникатор становится ненужным, он удаляется при помощи вызова процедуры *MPl_COMM_FREE*. Так можно использовать процедуру *MPI_COMM_SPLIT* для перенумерации процессов.

Пример 17:

```
#include <stdio.h>
#include "mpi.h"
void main(int argc,char * argv[])
{
int ierr,rank,size;
int rankl;
MPI_Comm comm_revs;
ierr=MPI_Init (&argc, &argv) ;
ierr=MPI_Comm_size(MPI_COMM_WORLD,&size) ;
ierr=MPI_Comm_rank(MPI_COMM_WORLD,&rank);
ierr=MPI_Comm_split(MPI_COMM_WORLD, 1, size-rank, &comm_revs) ;
ierr=MPI_Comm_rank(comm_revs,&rankl);
printf("Rank = %i, rankl = %i\n",rank,rankl);
```

ierr=MPI_Finalize(); }

Задание: Изменить программу так, чтобы все процессы передали сообщение процессу 0 со своим новым значением rankl.

Следующая программа иллюстрирует создание графовой топологии *comm_graph* для общения процессов по коммуникационной схеме *master-slave*. Все процессы в рамках данной топологии могут общаться только с нулевым процессом. После создания топологии с помощью вызова процедуры *MPI_GRAPH_CREATE* каждый процесс определяет количество своих непосредственных соседей в рамках данной топологии (с помощью вызова процедуры *MPI_Graph_neighbors_count)* и ранги процессов-соседей (с помощью вызова процедуры *MPI_Graph_neighbors*). После этого каждый процесс может в рамках данной топологии обмениваться данными со своими непосредственными соседями, например, при помощи вызова процедуры *MPI_Sendrecv*.

Пример 18:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc,char * argv[])
{
int ierr, rank, rankl, i, size;
const int MAXPROC =128, MAXEDGES = 512;
int a,b;
MPI_Status status;
MPI_Comm comm_graph;
int index[MAXPROC];
int edges[MAXPROC];
int num,neighbors[MAXPROC];
ierr=MPI_Init (&argc, &argv) ;
ierr=MPI_Comm_size(MPI_COMM_WORLD,&size);
ierr=MPI_Comm_rank(MPI_COMM_WORLD,&rank);
for (i=0; i<size; i++)  index[i]=size+i-2;
//index[0]--;
for (i=0;i<size-1;i++)
{
edges[i]=i;
edges[size+i-1]=0;
}
ierr=MPI_Graph_create(MPI_COMM_WORLD,size,index,&edges[1],1,&comm_grap
h);
ierr=MPI_Graph_neighbors_count(comm_graph,rank,&num);
ierr=MPI_Graph_neighbors(comm_graph,rank,num,neighbors);
if (rank==size-1) {ierr=MPI_Finalize (); return 0;}
for (i=0;i<num;i++) {
```

```
ierr=MPI_Sendrecv(&rank,1,MPI_INT,neighbors[i],1,&rankl,1,
MPI_INT,neighbors[i],1,comm_graph,&status);
printf ( "Process %i communicate with process %i\n" , rank, rankl) ; }
ierr=MPI_Finalize ();
return 0;
}
```

Задание: Измените программу так, чтобы все процессы могли общаться с нулевым и последним процессом и обменивались с ними сообщениями.

Использование производного типа данных показан в следующем примере. Производный тип данных применён для перестановки столбцов матрицы в обратном порядке. Тип данных *matr_rev*, создаваемый процедурой *MPl_TYPE_VECTOR*, описывает локальную часть матрицы данного процесса в переставленными в обратном порядке столбцами. После регистрации этот тип данных может использоваться при пересылке.

Программа работает правильно, если размер матрицы *N* делится нацело на число процессов приложения.

Пример 19:

```
#include <stdio.h>
#include "mpi.h"
#define N 8
void work(double  a[N][N], double b[N][N], int n, const int nl,int size,int rank)
{
int ierr, ii;
int i,j;
MPI_Datatype matr_rev;
MPI_Status status;
for (j=0;j<nl;j++)
for (i=0;i<nl;i++)
{
b[i][j]=0;
ii=j+rank*nl;
a[i][j]=100*ii+i;
}
ierr=MPI_Type_vector(nl,n,-n,MPI_DOUBLE,&matr_rev);
ierr=MPI_Type_commit(&matr_rev);
ierr=MPI_Sendrecv (&a[1][nl],1,matr_rev,size-rank-1, 1,b,nl*n,MPI_DOUBLE,size-
rank-1,1,MPI_COMM_WORLD,&status);
for (j=0;j<nl;j++)
for (i=0;i<n;i++)
printf ("Process %i : %i %i %d %d\n",rank, j+rank*nl,i,a[i][j] ,b[i][j] ) ;
}

void main(int argc,char * argv[])
{
int ierr, rank, nl, size;
```

```
double a[N][N],b[N][N];
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD, &size) ;
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &rank) ;
nl = (N-1)/size+1;
work(a,b,N,nl,size,rank);
ierr=MPI_Finalize () ;
}
```

Задание: Изменить программу так, чтобы в матрицу b сохранялась транспонированная матрица a.

Следующий пример иллюстрирует трансляционный способ обмена информацией. Массив *buf* используется в качестве буфера для упаковки 10 элементов массива *a* типа *float* и *10* элементов массива *b* типа *character*. Полученное сообщение пересылается процедурой *MPI_BCAST* от процесса 0 всем остальным процессам, полученное сообщение распаковывается при помощи вызовов процедур *MPI_UNPACK*.

Пример 20:

```
#include <stdio.h>
#include "mpi.h"
void main(int argc,char * argv[])
{
int i,ierr, rank, position;
float a[10];
char b[10],buf[100];
ierr=MPI_Init(&argc,&argv);
ierr=MPI_Comm_rank(MPI_COMM_WORLD,&rank);
for (i=0;i<10;i++)
{
a[i]=rank+1;
if (rank==0) b[i]='a'; else b[i]='b';
}
position=0;
if (rank==0)
{
ierr=MPI_Pack(a,10,MPI_FLOAT,buf,100,&position,MPI_COMM_WORLD);
ierr=MPI_Pack(b,10,MPI_CHAR,buf,100,&position,MPI_COMM_WORLD);
ierr=MPI_Bcast(buf,100,MPI_PACKED,0,MPI_COMM_WORLD); }
else {
ierr=MPI_Bcast(buf,100,MPI_PACKED,0,MPI_COMM_WORLD);
position=0;
ierr=MPI_Unpack(buf,100,&position,a,10,MPI_FLOAT,MPI_COMM_WORLD);
ierr=MPI_Unpack(buf, 100, &position, b, 10 , MPI_CHAR, MPI_COMM_WORLD) ; }
printf("Process %i a=[",rank);
for (i=0;i<10;i++) printf ("%i ",a[i]);
printf("] b=[");
```

```
for (i=0;i<10;i++) printf("%i ",b[i]);
printf ("]\n");
fflush(stdout);
ierr=MPI_Finalize();
}
```

Задание: Изменить программу так, чтобы процесс 0 передавал всем остальным процессам номер вашего индекса в 7-элементном массиве int, а последний процесс передавал остальным ваше имя в 10-элементном массиве character.

Список рекомендуемой литературы

1. Руденко Ю.М., Волкова Е.А. “Вычислительные системы”.-М.: НИИ РЛ МГТУ им. Н.Э.Баумана, 2010.-212 с.

2. Антонов А.С. "Параллельное программирование с использованием технологии MPI".-М.: Изд-во МГУ, 2004.-71 с.

3. Кузнецов О.П., Адельсон-Вельский Г.М. Дискретная математика для инженера. – М.: Энергоатомиздат. 1988. – 478 с.

4. Водяхо А.И., Горнец Н.Н., Пузанков Д.В. Высокопроизводительные системы обработки данных. – М.: Высшая школа. 1997. – 150 с.

5. Корнеев В.В. Параллельные вычислительные системы. – М.: Наука. 1999. – 312 с.

6. Белоусов А.И., Ткачев С.Б. Дискретная математика: Учеб. для вузов / Под ред. В.С.Зарубина, А.П. Крищенко. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2001. – 774 с. (Сер. Математика в техническом университете; Вып. XIX).

Оглавление

Введение 3
1. Лабораторная работа №1. Определение параметров n-мерных коммутационных структур ВС типа гиперкуб, тор и циркулянт 7
2. Лабораторная работа №2. Преобразование последовательного алгоритма в параллельный 13
3. Лабораторная работа №3. Представление алгоритмов в виде граф–схем........ 20
3. Лабораторная работа №4. Построение матрицы логической несовместимости операторов 41
4. Лабораторная работа №5. Построение множеств взаимно независимых операторов........ 50
5. Лабораторная работа №6. Определение ранних и поздних сроков окончания выполнения операторов и оценка снизу требуемого количества процессоров и времени решения задачи на ВС 59
7. Лабораторная работа №7. Запуск параллельных программ на кластере 68
8. Лабораторная работа №8. Написание программ с использованием библиотеки MPI........ 79
Список рекомендуемой литературы........ 102