



**МИНИСТЕРСТВО ОБРАЗОВАНИЯ
РЕСПУБЛИКИ БЕЛАРУСЬ**

**Белорусский национальный
технический университет**

**Кафедра «Программное обеспечение вычислительной техники
и автоматизированных систем»**

А. А. Прихожий

**РАСПРЕДЕЛЕННАЯ
И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА
ДАННЫХ**

Учебно-методическое пособие

**Минск
БНТУ
2016**

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Белорусский национальный технический университет

Кафедра «Программное обеспечение вычислительной техники
и автоматизированных систем»

А. А. Прихожий

РАСПРЕДЕЛЕННАЯ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

Учебно-методическое пособие
для студентов специальности 1-40 01 01
«Программное обеспечение информационных технологий»
и направления специальности 1-40 05 01 04
«Информационные системы и технологии
(в обработке и представлении данных)»

*Рекомендовано учебно-методическим объединением по образованию
в области информатики и радиоэлектроники*

Минск
БНТУ
2016

УДК 004.272.2 (075.8)

ББК 32.97я7

П77

Рецензенты:

В. С. Муха, В. А. Вишняков

Прихожий, А. А.

П77 Распределенная и параллельная обработка данных : учебно-методическое пособие для студентов специальности 1-40 01 01 «Программное обеспечение информационных технологий» и направления специальности 1-40 05 01 04 «Информационные системы и технологии (в обработке и представлении данных)» / А. А. Прихожий. – Минск : БНТУ, 2016. – 91 с.

ISBN 978-985-550-639-4.

Учебно-методическое пособие составлено в соответствии с учебной программой по дисциплине «Распределенная обработка данных на ЭВМ» для студентов специальности 1-40 01 01 «Программное обеспечение информационных технологий» и направления специальности 1-40 05 01 04 «Информационные системы и технологии (в обработке и представлении данных)».

Рассматриваются распределенные и параллельные системы, задачи, модели и алгоритмы планирования процессов обработки данных, языки и инструменты программирования распределенных и параллельных систем.

УДК 004.272.2 (075.8)

ББК 32.97я7

ISBN 978-985-550-639-4

© Прихожий А. А., 2016

© Белорусский национальный
технический университет, 2016

Оглавление

Введение	5
1. Системы распределенной и параллельной обработки данных	6
1.1. Классификация распределенных систем	6
1.2. Архитектуры распределенных систем	6
1.3. Свойства распределенных систем	7
1.4. Классификация параллелизма	8
1.5. Закон Амдала	10
1.6. Задачи распределенных и параллельных систем	13
2. Планирование синхронных параллельных процессов обработки данных	14
2.1. Постановка задачи	14
2.2. Классификация задач и стратегий планирования	16
2.3. Обыкновенное планирование	17
2.4. Многошаговое планирование	23
2.5. Цепочечное планирование	29
2.6. Сравнение стратегий планирования	31
2.7. Свертывание графа распараллеленности операций	32
2.8. Сведение планирования к задаче целочисленного линейного программирования	39
3. Планирование асинхронных распределенных параллельных процессов с учетом обмена данными	44
3.1. Граф задач	44
3.2. Асинхронный параллельный план	46
3.3. Задача минимизации временной длины плана	47
3.4. Стратегия планирования «Ранняя задача первая»	49
3.5. Стратегия планирования «Зануление дуг»	54

3.6. Стратегия планирования «Группировка доминирующей последовательности»	57
3.7. Планирование с использованием мобильности задач	61
4. Планирование решения задач в разнородной распределенной системе	64
4.1. Модель разнородной системы	64
4.2. Назначение задач на узлы	68
4.3. Оценка общего времени решения задач	69
4.4. Алгоритм оптимального назначения задач на процессоры	70
5. Языки и инструменты программирования распределенной и параллельной обработки данных	76
5.1. Многопоточные приложения	76
5.2. Интерфейс MPI.....	79
5.3. Открытый стандарт OpenMP	82
5.4. Технологический стандарт CORBA	84
5.5. Модель компонентных объектов COM.....	85
Список использованных источников.....	91

ВВЕДЕНИЕ

Целями создания и повсеместного использования распределенных и параллельных систем являются:

- обеспечение быстрого и легкого доступа к удаленным ресурсам и разделение их между пользователями прозрачным и эффективным способом;
- обеспечение эффективного взаимодействия пользователей и программ и обмена информацией между ними;
- обеспечение логической прозрачности системы и скрытие от пользователей физической архитектуры распределенных и параллельно работающих ресурсов;
- обеспечение открытости распределенной системы посредством предоставления сервисов, доступных пользователям по стандартным правилам, определяющим общие синтаксис и семантику сервисов;
- организация параллельной эффективной работы вычислительных и информационных ресурсов при решении задач большой вычислительной сложности;
- обеспечение масштабируемости системы в плане: размера, измеряемого числом пользователей и ресурсов; географического расположения, проявляющегося в отдаленном размещении пользователей и ресурсов; административного управления, когда система объединяет много административных организаций.

Виртуализация, управление и планирование процессов являются наиболее значимыми задачами в распределенных и параллельных системах. Важнейшими параметрами систем являются высокая производительность, высокая загрузка оборудования, устойчивость и надежность в работе, низкое энергопотребление и др. Структурная и параметрическая оптимизация систем позволяет значительно улучшить параметры, как в процессе разработки, так и в процессе эксплуатации систем.

Пособие, в силу своего ограниченного объема, делает акцент на проблемах моделирования и оптимизации распределенных и параллельных систем, менее всего освещенных в учебной литературе. Оно уделяет также внимание эффективным и востребованным средствам разработки параллельных и распределенных приложений.

1. СИСТЕМЫ РАСПРЕДЕЛЕННОЙ И ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ ДАННЫХ

1.1. Классификация распределенных систем

Различают три класса распределенных систем [1, 2]: *вычислительные, информационные* и *встроенные*. Системы из первого класса предназначены для высокопроизводительных параллельных вычислений. Этот класс делится на два подкласса: *кластерные* системы и *grid-системы*. Кластерные системы состоят из однотипных рабочих станций или персональных компьютеров, тесно соединенных высокопроизводительной локальной сетью. На каждом узле кластера установлена одна и та же операционная система. *Grid-системы* состоят из объединенных, но административно разделенных компьютерных систем, которые могут сильно различаться как по установленной аппаратуре, так и используемому программному обеспечению.

Второй класс составляют распределенные информационные системы, состоящие, как правило, из *серверных* и *клиентских* частей. Этот класс делится на два подкласса: системы обработки *транзакций* баз данных и системы *интеграции* приложений. И те и другие активно используют механизм удаленного вызова процедур. Разработка систем интеграции приложений основана на использовании различных моделей коммуникации компонентов приложения.

Третий класс составляют *всеохватывающие* встроенные распределенные системы, окружающие человека. В первую очередь к ним относятся малые мобильные системы с беспроводным соединением. Подкласс домашних распределенных систем строится из компьютеров и множества устройств потребительской электроники. Подкласс медицинских распределенных систем выполняет мониторинг состояния пациента и при необходимости взаимодействует с врачом. Сенсорные распределенные системы состоят из десятков и сотен тысяч сенсорных устройств, которые могут рассматриваться как распределенная база данных.

1.2. Архитектуры распределенных систем

Различают следующие архитектурные стили распределенных систем:

1) структурированные по уровням (последующий уровень вызывает компоненты нижестоящего уровня);

2) объектно-ориентированные (объекты взаимодействуют посредством механизма удаленного вызова процедур);

3) центрированные вокруг данных (процессы взаимодействуют через общее хранилище данных, пример – сервисы, базирующиеся на *web*);

4) структурированные на множестве событий (процессы взаимодействуют через распространение событий).

Архитектуры распределенных систем делятся на:

1) централизованные (множество клиентов обслуживается одним сервером);

2) децентрализованные (многосвязные клиент-серверные приложения, в которых выделяются уровни данных, обрабатывающие компоненты и интерфейс пользователей); в пиринговых (*per-to-peer*) децентрализованных системах все процессы наделены равными правами;

3) гибридные (клиент-серверные части комбинируются с децентрализованными частями);

1.3. Свойства распределенных систем

Распределенные системы характеризуются следующими свойствами:

1) скрытие от пользователей различий между компьютерами, включенными в распределенную систему, и между способами взаимодействия компьютеров;

2) использование единого согласованного и унифицированного способа взаимодействия всех пользователей в системе;

3) обеспечение легкой расширяемости и неограниченной масштабируемости системы, не оказывающих влияние на работу пользователей;

4) поддержка разнородных компьютеров и сетей посредством организации системы в виде специального *промежуточного* слоя программного обеспечения (*middleware*), интегрирующего все ресурсы таким образом, что они доступны всем потребителям.

1.4. Классификация параллелизма

В зависимости от организации данных и взаимодействия операций в распределенном алгоритме, выполняется пространственно-временная классификация параллелизма [3]. Различают два основных вида параллельного поведения распределенной системы:

- 1) параллелизм в пространстве;
- 2) параллелизм во времени.

Параллелизм в пространстве. Он свойственен системам, одновременно обрабатывающим один входной набор данных таким образом, что все операции алгоритма принимают на входе элементы этого набора, либо значения, являющиеся производными от этих элементов, полученные операциями-предшественниками. В конечном счете, это приводит к вычислению результирующих значений выходного набора. Операции, выполняющиеся параллельно в пространстве, взаимно не предшествуют друг другу и являются информационно независимыми.

Пример 1.1. Иллюстрация параллелизма в пространстве дана на рис. 1.1. Алгоритм, включающий операции «+» и «*», обрабатывает входной набор, состоящий из переменных x , y , z . Операция «+» не использует переменную b , являющуюся результатом выполнения операции «*», и наоборот, операция «*» не использует результирующую переменную a операции «+». Операции являются взаимно независимыми и выполняются параллельно.

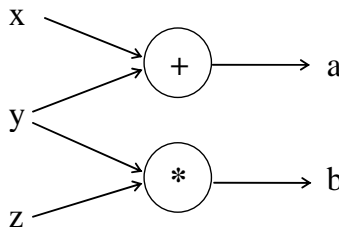


Рис. 1.1. Пример параллелизма в пространстве

Параллелизм во времени. Он свойственен системам, обрабатывающим поток данных, состоящий из последовательности входных наборов данных. Параллелизм во времени называется также конвейерным параллелизмом. Конвейер состоит из ступеней, выпол-

няющих преобразование информации и взаимодействующих таким образом, что выходные данные одной ступени являются входными данными последующей ступени. Ступени соединены последовательно, однако все они работают параллельно на различных наборах данных, которые шаг за шагом проталкиваются через конвейер. Число одновременно обрабатываемых наборов равно числу ступеней конвейера. Более того, порядок обработки последующего набора может зависеть от результатов обработки предыдущих наборов.

Пример 1.2. Иллюстрация параллелизма во времени дана на рис. 1.2. Алгоритм, включающий последовательно выполняющиеся операции «+» и «*», функционирует как двухступенчатый конвейер. Первая ступень конвейера построена из операции «+», вторая ступень – из операции «*». На вход конвейера подается поток данных, состоящий из наборов $x_1, y_1, z_1, \dots, x_n, y_n, z_n$, на выходе первой ступени генерируется поток переменных b_1, \dots, b_n , на выходе второй ступени появляется поток переменных a_1, \dots, a_n , являющийся выходным потоком. В момент вычисления второй ступенью и операцией «*» значения переменной a_i первая ступень и операция + вычисляют значение переменной b_{i+1} . Операции «+» и «*» выполняются одновременно (параллельно) во времени, но над различными наборами данных.

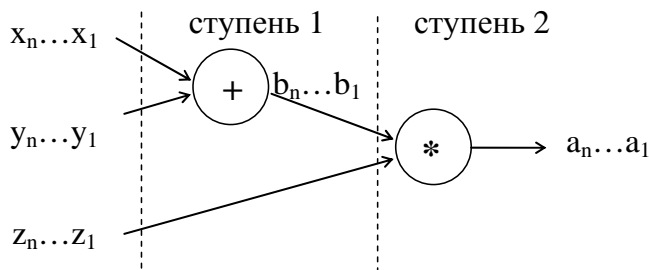


Рис. 1.2. Пример параллелизма во времени (конвейер)

Смешанный параллелизм. В системе со смешанным параллелизмом одновременно присутствует параллелизм в пространстве и параллелизм во времени. Параллелизм в пространстве реализуется внутри каждой ступени, параллелизм во времени проявляется в одновременном выполнении операций на разных ступенях.

Пример 1.3. Иллюстрация смешанного параллелизма дана на рис. 1.3. Алгоритм, в котором две параллельно выполняющиеся операции «+» выполняются последовательно с операцией «*», использован для построения двухступенчатого конвейера. На первой ступени конвейера две операции «+» выполняются над одним набором данных и реализуют параллелизм в пространстве. Две ступени конвейера реализуют параллелизм во времени.

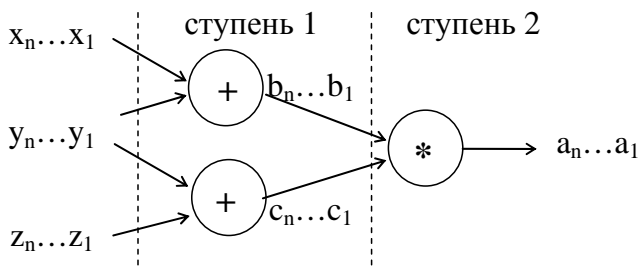


Рис. 1.3. Пример смешанного параллелизма

1.5. Закон Амдала

Закон Амдала устанавливает увеличение производительности многопроцессорной системы по сравнению с однопроцессорной системой в зависимости от свойств решаемой задачи, алгоритма ее решения и объема ресурсов, предоставляемых системой. Рассмотрим закон Амдала в двух вариантах: без учета сети связи (базовый закон) и с учетом сети связи (сетевой закон).

Базовый закон Амдала. Одной из главных характеристик параллельных систем является ускорение R решения задачи параллельной многопроцессорной системой по сравнению с последовательной однопроцессорной системой:

$$R = \frac{T_1}{T_n}, \quad (1.1)$$

где T_1 – время решения задачи на однопроцессорной системе;

T_n – время решения той же задачи на n -процессорной системе.

Выразим T_1 и T_n через основные параметры последовательного и параллельного алгоритмов.

Пусть W – общее число операций в алгоритме решения задачи. Условно все множество операций можно разделить на два подмножества: подмножество последовательно выполняемых операций и подмножество параллельно выполняемых операций. Пусть $W_{\text{посл}}$ – число операций в первом подмножестве, $W_{\text{пар}}$ – число операций во втором подмножестве. Очевидно, что $W = W_{\text{посл}} + W_{\text{пар}}$.

Закон Амдала не учитывает специфику и типы операций, посредством которых описывается алгоритм, но использует среднее время t выполнения одной операции, как последовательной, так и параллельной. Благодаря этому соотношение (1.1) можно переписать в виде

$$R = \frac{Wt}{(W_{\text{посл}} + \frac{W_{\text{пар}}}{n})t} = \frac{1}{a + \frac{1-a}{n}}, \quad (1.2)$$

где $a = W_{\text{посл}} / W$ – доля последовательных операций в общем числе операций алгоритма. При числе процессоров n , стремящемся к бесконечности, ускорение R стремится к величине $1/a$.

Закон Амдала определяет два основных положения, которые являются принципиально важными для параллельных систем:

1. Ускорение R зависит от свойств решаемой задачи, в частности, от потенциального параллелизма, заложенного в алгоритме решения и описываемого долей a последовательно выполняемых операций и долей $1-a$ операций, которые могут выполняться параллельно.

2. Ускорение зависит от параметров многопроцессорной системы, главным образом от числа входящих в систему процессоров n .

Согласно (1.2), для чисто последовательного алгоритма с нулевым уровнем потенциального параллелизма, когда $a = 1$, для ускорения всегда выполняется $R = 1$, что означает отсутствие ускорения не зависимо от числа процессоров n . Обратное, для однопроцессорной системы, когда $n = 1$, для ускорения также всегда выполняется $R = 1$, что означает отсутствие ускорения не зависимо от свойств распараллеливаемого алгоритма. Для всех остальных случаев, когда $a < 1$ и $n > 1$, коэффициент ускорения превышает значение единицы: $R > 1$.

Сетевой закон Амдала. Основной вариант (1.2) закона Амдала не отражает потерь времени на межпроцессорный обмен данными

в многопроцессорной системе. Очевидно, что в однопроцессорной системе такого обмена нет. Потери на обмен данными могут не только снизить ускорение вычислений, но даже замедлить вычисления по сравнению с однопроцессорным вариантом.

Выполним корректировку выражения (1.2), введя дополнительные величины, характеризующие функционирование сети связи. Пусть W_c – количество операций передачи данных между процессорами; t_c – среднее время выполнения одной операции. Тогда закон Амдала переписывается в виде

$$R_c = \frac{Wt}{(W_{\text{посл}} + \frac{W_{\text{нар}}}{n})t + W_c t_c} = \frac{1}{a + \frac{1-a}{n} + c},$$

где c – коэффициент сетевой деградации вычислений в распределенной параллельной системе, определяемый выражением:

$$c = \frac{W_c t_c}{Wt} = c_a c_T.$$

Коэффициент c определяет отношение общего времени выполнения всех операций передачи данных в сети к общему времени выполнения всех операций алгоритма на процессорах. Первая составляющая $c_a = W_c/W$ определяет алгоритмическую часть коэффициента деградации, обусловленную свойствами параллельного алгоритма, а именно отношением числа операций обмена к числу операций на процессорах. При качественной разработке параллельной программы число операций обмена должно быть значительно меньше числа операций на узлах. Вторая составляющая $c_T = t_c/t$ определяет техническую часть коэффициента деградации, обусловленную свойствами коммутационной сети, а именно, отношением среднего времени выполнения операции обмена к среднему времени выполнения операции на узле. Для реальных многопроцессорных систем время операции обмена значительно больше времени операции на процессоре. В целом, коэффициент деградации c может принимать значения, как меньшие, так и большие 1. Благодаря этому, коэффициент ускорения R_c , в отличие от коэффициента R , может принимать значения, меньшие 1, что означает замедление вычислений по сравнению с однопроцессорным вариантом.

Закон Амдала полезен тем, что он дает быструю и простую оценку возможного ускорения при переходе от последовательного алгоритма к его параллельной версии. Недостатком является невысокая точность оценки и невозможность учета конкретных параметрических и структурных особенностей задачи и алгоритма.

1.6. Задачи распределенных и параллельных систем

Существует множество теоретических и практических задач, которые решаются при разработке и эксплуатации распределенных и параллельных систем:

1) математическое и имитационное моделирование с целью исследования характеристик системы и принятия обоснованных решений в период проектирования или эксплуатации;

2) разработка архитектуры и выбор базового аппаратного обеспечения;

3) выбор программного обеспечения для узлов системы и программного обеспечения среднего слоя;

4) разработка архитектуры параллельной распределенной системы в случае необходимости реализации высокопроизводительных вычислений;

5) создание адекватных математических и компьютерных моделей распределенных и параллельных систем;

6) разработка методов, стратегий и алгоритмов планирования информационно-вычислительных процессов в распределенных и параллельных системах;

7) создание и использование языков и инструментальных средств параллельного и распределенного программирования;

8) разработка параллельных и распределенных алгоритмов для решения важнейших задач науки и техники;

9) автоматическое распараллеливание последовательного программного кода, создание и использование распараллеливающих компиляторов;

10) экстракция параллелизма и оптимизация параллельных и распределенных программ.

Важнейшие параметры параллельной распределенной системы – время решения задач (производительность системы) и объем используемых вычислительных ресурсов.

2. ПЛАНИРОВАНИЕ СИНХРОННЫХ ПАРАЛЛЕЛЬНО-ПОСЛЕДОВАТЕЛЬНЫХ ПРОЦЕССОВ ОБРАБОТКИ ДАННЫХ

2.1. Постановка задачи

Система планирования процессов обработки данных отображает заданное описание вычислительного процесса в синхронный последовательно-параллельный план в соответствии с поставленной задачей оптимизации [3], как показано на рис. 2.1.

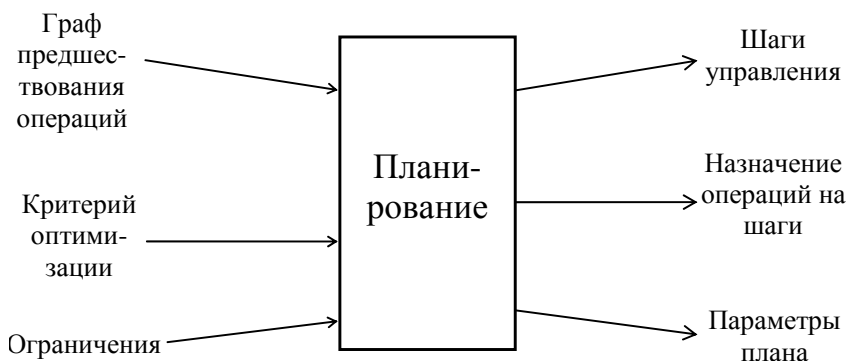


Рис. 2.1. Задача планирования процессов обработки данных

Вычислительный процесс представляется графом $G_H = (N, H)$ предшествования операций, в котором $N = \{1, \dots, n\}$ – множество вершин-операций; H – множество ребер, описывающих отношение предшествования операций. Каждая операция $i \in N$ характеризуется параметрами, такими как тип операции $type_i$, время выполнения t_i и др. Информационные зависимости между операциями являются главным источником построения отношения предшествования H .

Задача оптимизации описывается критерием оптимизации и системой ограничений. Важнейшими оптимизируемыми параметрами синтезируемых планов являются время $Time$ реализации плана и стоимость $Cost$ потребляемых планом ресурсов. Время реализации оценивается выражением $Time = T_{step} \times Steps$, где T_{step} – время шага; $Steps$ – число шагов управления, введенных в план. Стои-

мость $Cost$ оценивается либо общим количеством P процессоров, необходимых для выполнения плана, либо общей стоимостью процессорных элементов

$$Cost = \sum_{j=1}^{Types} c_j P_j,$$

где $Types$ – число типов процессорных элементов;

c_i – стоимость процессора i -го типа;

p_i – число процессоров i -го типа, обрабатывающих соответствующий тип операций.

Критериями оптимизации могут быть $\min\{Time\}$ при заданном ограничении на $Cost$ или $\min\{Cost\}$ при заданном ограничении на $Time$.

Пример 2.1. Граф G_H , построенный на 8 вершинах и 7 ребрах, приведен на рис. 2.2. Он содержит 5 вершин-операций типа «+» и три вершины-операции типа «*». Структура графа описывается следующей матрицей смежности, которая в силу ацикличности графа имеет верхний треугольный вид:

$$H = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ & & 0 & 1 & 0 & 0 & 0 & 0 \\ & & & 0 & 1 & 0 & 0 & 0 \\ & & & & 0 & 0 & 0 & 1 \\ & & & & & 0 & 1 & 0 \\ & & & & & & 0 & 1 \\ & & & & & & & 0 \end{pmatrix}.$$

Синхронный последовательно-параллельный план строится путем введения шагов управления и распределения операций по шагам. Параметры плана зависят от числа шагов. Обычно увеличение числа шагов делает план более последовательным, увеличивает время реализации плана и сокращает объем используемых

ресурсов. Сокращение числа шагов делает план более распараллеленным, сокращает время реализации и увеличивает объем используемых ресурсов.

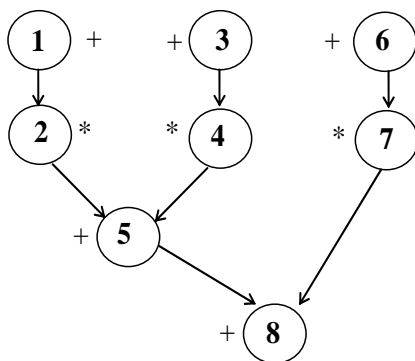


Рис. 2.2. Пример графа G_N предшествования операций

2.2. Классификация задач и стратегий планирования

Планирование обработки данных классифицируется по следующим признакам:

1. По структуре синтезируемого плана:

- *обыкновенное (ordinary)*, в котором каждая операция выполняется на одном шаге управления, операции из одного шага выполняются параллельно, а операции из различных шагов выполняются последовательно;

- *многошаговое (multi cycling)*, в котором одна операция может выполняться на нескольких соседних шагах управления, а в пределах одного шага все операции выполняются параллельно;

- *цепочное (chaining)*, в котором операции могут образовывать цепочки и выполняться последовательно в пределах одного шага, операции из разных цепочек выполняются в пределах одного шага параллельно;

- *конвейерное (pipelining)*, в котором план разбивается на ступени конвейера, обрабатывающие различные наборы данных параллельно во времени;

2. По решаемой оптимизационной задаче:

- с ограничением на время реализации плана (*time-constrained*) и минимизацией объема потребляемых вычислительных ресурсов;
- с ограничением на объем потребляемых вычислительных ресурсов (*resource-constrained*) и минимизацией времени реализации плана;
- с ограничениями на время реализации плана и объем потребляемых вычислительных ресурсов – задача на достижимость (*feasible-constrained*);

3. По стратегии планирования:

- назначение операции на наиболее ранний шаг управления (*As Soon As Possible – ASAP*);
- назначение операции на наиболее поздний шаг управления (*As Late As Possible – ALAP*);
- списковое планирование (*List Scheduling – LS*);
- сведение задачи планирования к задаче целочисленного линейного программирования (*Integer Linear Programming Formulation – ILPF*);
- сведение задачи планирования к свертыванию графа распараллеленности операций и другие стратегии.

2.3. Обыкновенное планирование

Все основные вышеперечисленные стратегии планирования позволяют синтезировать обыкновенный план, в котором каждая операция выполняется ровно на одном шаге управления.

2.3.1. Стратегия планирования ASAP

Алгоритм планирования ASAP стремится назначать операции на как можно более ранние шаги управления. Алгоритм решает оптимизационную задачу на достижимость путем минимизации времени выполнения плана при отсутствии ограничений на ресурсы.

Исходные данные:

1. Граф G_H предшествования операций.

Результирующие данные:

1. Шаги управления;
2. Отображение операций на шаги управления;
3. Число процессоров каждого типа.

Описание алгоритма:

1. Планирование выполняется в цикле, начиная с первого шага и кончая последним шагом;

2. Алгоритм начинает работу с введения первого шага управления и назначения на него операций, не имеющих согласно графу G_H операций-предшественников;

3. Алгоритм завершает работу в случае назначения всех операций на шаги управления;

4. Для каждого шага s управления выполняются следующие действия по планированию:

– из множества всех операций N формируется подмножество $N_s \subseteq N$ операций, не спланированных к моменту формирования текущего шага управления;

– для каждой операции $r \in N_s$ проверяется, все ли операции-предшественники $pred(r)$ уже спланированы на предшествующих шагах управления, другими словами проверяется включение $pred(r) \subseteq N \setminus N_s$;

– если включение выполняется, то операция r назначается на текущий шаг s , в противном случае операция остается не спланированной.

Пример 2.2. Синхронный план ASAP, синтезированный для графа предшествования операций, показанного на рис. 2.2, изображен на рис. 2.3. План построен на минимальном числе шагов управления, равном 4. Время одного шага определяется максимальным временем выполнения одной операции. Два правых столбца показывают число процессоров типа «+» и типа «*», необходимых для параллельного выполнения операций на каждом шаге управления. Взятие максимума по числу процессоров каждого типа на каждом из шагов дает общее количество 6 процессоров, необходимых для реализации плана.

Шаг управления	Операции	Тип +	Тип *
1		3	0
2		0	3
3		1	0
4		1	0
Максимум =		3	3

Рис. 2.3. План ASAP, построенный на четырех шагах управления, требует три процессора типа «+» и три процессора типа «*», всего шесть процессоров

2.3.2. Стратегия планирования ALAP

Алгоритм планирования ALAP стремится назначать операции на как можно более поздние шаги управления. Как и ASAP, алгоритм решает оптимизационную задачу на достижимость путем минимизации времени выполнения плана при отсутствии ограничений на ресурсы.

Исходные данные:

1. Граф G_H предшествования операций.

Результирующие данные:

1. Шаги управления;
2. Отображение операций на шаги управления;
3. Число процессоров каждого типа.

Описание алгоритма:

1. Планирование выполняется в цикле, начиная с последнего шага и кончая первым шагом;
2. Алгоритм начинает работу с введения последнего шага управления и размещения на нем операций, не имеющих операций-последователей;
3. Алгоритм завершает работу в случае назначения всех операций на шаги управления;
4. Для каждого шага управления выполняются следующие действия по планированию:

– из множества всех операций N формируется подмножество $N_s \subseteq N$ операций, не спланированных к моменту формирования текущего шага управления;

– для каждой операции $r \in N_s$ проверяется, все ли операции-последователи $succ(r)$ уже спланированы на последующих шагах управления, другими словами проверяется включение $succ(r) \subseteq M \setminus N_s$;

– если включение выполняется, то операция r назначается на текущий шаг s , в противном случае операция остается не спланированной.

Пример 2.3. На рис. 2.4 изображен синхронный план ALAP, синтезированный для графа G_H , показанного на рис. 2.2. План построен на минимальном числе 4 шагов управления. В отличие от плана ASAP (рис. 2.3), в котором все операции прижаты к первому шагу, в плане ALAP все операции прижаты к последнему шагу. Два правых столбца показывают число процессоров типа «+» и типа «*», необходимых для параллельного выполнения операций на каждом шаге управления, а общее количество процессоров, необходимых для реализации всего плана, равно 4. План ALAP сокращает объем используемых ресурсов на 33 % по сравнению с планом ASAP при одном и том же общем времени выполнения.

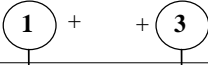
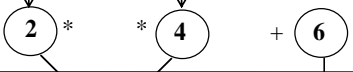
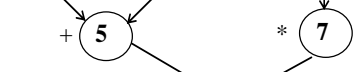
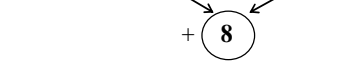
Шаг управления	Операции	Тип +	Тип *
1		2	0
2		1	2
3		1	1
4		1	0
	Максимум =	2	2

Рис. 2.4. План ALAP, построенный на четырех шагах управления требует два процессора типа «+» и два процессора типа «*», всего четыре процессора

2.3.3. Стратегия спискового планирования *LS*

Алгоритм спискового планирования *LS* минимизирует число шагов управления и время выполнения плана при заданных ограничениях на объем используемых вычислительных ресурсов (*resource-constrained scheduling*). Ограничения представляются числом доступных процессоров каждого типа. Название алгоритма подчеркивает тот факт, что в процессе своей работы алгоритм активно использует список *List* готовых к планированию операций. Алгоритм *LS* может быть построен в двух вариантах: на базе стратегии *ASAP* и на базе стратегии *ALAP*. Ниже дается описание алгоритма *LS*, построенного на базе *ASAP*.

Исходные данные:

1. Граф G_H предшествования операций.
2. Число p_i доступных процессоров типа $i = 1, \dots, Types$.

Результирующие данные:

1. Шаги управления.
2. Отображение операций на шаги управления.

Описание алгоритма:

1. Планирование выполняется в цикле, начиная с первого шага и кончая последним шагом управления. В текущий шаг, который формируется на очередной итерации цикла, могут быть включены только те операции, которые находятся в списке *List*;

2. Список *List* готовых к планированию операций изменяется динамически. Он состоит из двух частей. Первая часть включает операции, оставшиеся не спланированными с предыдущей итерации цикла планирования и перенесенные на текущий шаг планирования. Вторая часть включает операции, которые стали готовы к планированию благодаря тому, что все их операции предшественники стали спланированными в результате формирования предыдущего шага управления на предыдущей итерации цикла;

3. Начальное состояние списка *List* формируется перед запуском цикла планирования. Первая часть списка является пустой, во вторую часть включаются операции, не имеющие операций предшественников в графе G_H ;

4. Когда первая и вторая части списка *List* становятся пустыми, то это значит, что все операции назначены на шаги управления. Алгоритм *LS* завершает работу;

5. Для каждого шага управления выполняются следующие действия по планированию операций:

- из списка $List$, равно как из первой, так и из второй части, должны быть выбраны операции, назначаемые на текущий шаг управления;

- если найдется хотя бы один тип $i \in \{1, \dots, Types\}$ такой, что для подмножества $L_i \subseteq List$ операций $r \in L_i$, $type_r = i$ выполняется соотношение $|L_i| > p_i$, то между операциями возникает конкуренция на включение в текущий шаг управления из-за нехватки процессоров типа i ;

- для предпочтительного выбора операций используются критерии выбора. Наиболее часто используемым является критерий принадлежности операции критическому пути на графе G_H предшествования операций;

- на текущий шаг управления назначаются операции из списка $List$ в количестве, не превышающем число имеющихся процессоров каждого типа;

- оставшиеся в списке $List$ операции остаются не спланированными, перемещаются в первую часть списка и становятся претендентами на включение в следующий шаг плана;

- во вторую часть списка $List$ включаются новые ранее не спланированные операции, для которых все операции-предшественники оказываются спланированными на текущем шаге.

Пример 2.4. Проиллюстрируем работу алгоритма LS на примере графа G_H , изображенного на рис. 2.2, при условии, что в наличии имеется по одному процессору каждого типа: $p_+ = 1$ и $p_* = 1$. Генерируемый LS-план показан на рис. 2.5. Он построен на минимальном числе 5 шагов управления. Дополнительный шаг с номером 0 определяет начальное состояние списка $List$, включающего операции 1, 3, 6, не имеющие предшественников в графе G_H . Поскольку все операции 1, 3, 6 относятся к одному типу $+$, а в наличии имеется один процессор этого типа, на первый шаг назначается только одна из трех операций. Это операция с номером 1, лежащая на наиболее длинном пути на графе G_H . Операции 3, 6 перемещаются из второй в первую часть списка $List$. Во вторую часть списка включается операция 2, имеющая одного предшественника, операцию 1, кото-

рая уже назначена на первый шаг. На второй шаг управления назначаются две операции 2 и 3, относящиеся к разным типам. На третий шаг назначаются операции 4 и 6, на четвертый – операции 5 и 7, на пятый – операция 8. LS-план оказался намного эффективнее по сравнению с ASAP и ALAP, так как при увеличении числа шагов на всего 1 он сократил число процессоров на 4 и 2 соответственно.






Шаги управления	Операции	Список <i>List</i>
0		1, 3, 6
1		3, 6 2
2		6 4
3		5, 7
4		8
5		

Рис. 2.5. План LS, построенный на пяти шагах управления при ограничениях $p_+ = 1$ и $p_* = 1$

2.4. Многошаговое планирование

Принципы многошагового планирования:

1. Разрешается выполнение одной операции на нескольких соседних шагах управления;
2. Учитывается время шага управления t_{step} , которое может быть меньше максимального времени выполнения операций;
3. Если время t_i выполнения операции i меньше времени шага t_{step} , операция выполняется на одном шаге;
4. Если время t_i выполнения операции i больше времени шага t_{step} , операция выполняется на $k = \lceil t_i / t_{step} \rceil$ шагах, где $\lceil x \rceil$ – ближайшее целое, не меньшее x .

Преимущества многошагового планирования по сравнению с обыкновенным планированием:

1. Сокращение общего времени выполнения плана;
2. Увеличение загрузки оборудования; в отличие от многошагового плана, в обыкновенном плане оборудование, назначаемое коротким операциям, простаивает до завершения выполнения длинных по времени операций.

Алгоритмы многошагового планирования строятся путем модификации алгоритмов ASAP, ALAP и LS.

2.4.1. Многошаговое планирование на базе ASAP

Алгоритм многошагового планирования MC-ASAP (*Multi Cycling ASAP*) учитывает соотношение между временем шага и временем выполнения операции и стремится назначать каждую операцию на наиболее ранние шаги управления. Алгоритм решает оптимизационную задачу на достижимость минимального числа шагов без учета ограничений на ресурсы.

Исходные данные:

1. Граф G_H предшествования операций;
2. Времена t_i , $i \in N$ выполнения операций;
3. Время t_{step} шага управления.

Результирующие данные:

1. Шаги управления;
2. Отображение операций на шаги управления;
3. Число процессоров каждого типа.

Описание алгоритма:

1. Планирование выполняется в цикле, начиная с первого и кончая последним шагом управления;

2. На каждом шаге s множество всех операций разбивается на три подмножества: N_s не спланированных операций, N'_s готовых к планированию или уже частично спланированных операций, N''_s полностью спланированных операций;

3. С каждой операцией $r \in N'_s$ ассоциируется время выполнения τ_r , не покрытое шагами управления, на которые назначена часть операции r . В момент начального включения r в подмножество N'_s время $\tau_r = t_r$;

4. Алгоритм начинает работу до запуска цикла с поиска в подмножестве $r \in N_s$ операций, не имеющих согласно графу G_H операций-предшественников ($pred(r) = \emptyset$), и перемещения их из N_s в подмножество N'_s с начальной установкой значений $\tau_r = t_r$.

5. Для каждого шага s управления выполняются следующие действия по планированию:

– каждая операция $r \in N'_s$ назначается на шаг s управления; далее, если выполняется неравенство $\tau_r \leq T_{step}$, то операция r перемещается из подмножества N'_s в подмножество N''_s , в противном случае осуществляется редуцирование $\tau_r = \tau_r - T_{step}$ с сохранением r в N'_s ;

– для каждой операции $r \in N_s$ проверяется, все ли операции-предшественники $pred(r)$ уже спланированы на предшествующих шагах управления, другими словами проверяется включение $pred(r) \subseteq N''_s$; если предшественники спланированы, операция r перемещается из подмножества N_s в подмножество N'_s с начальной установкой значений $\tau_r = t_r$;

6. Алгоритм завершает работу в случае появления пустого подмножества N'_s .

Пример 2.5. Проиллюстрируем работу алгоритма MC-ASAP на графе G_H , изображенном на рис. 2.2. Примем время шага $T_{step} = 1$ и время выполнения операций $t_+ = 1$ и $t_* = 2$. Очевидно, что время выполнения операций типа «*» в два раза превышает время шага, а операции 2, 4, 7 должны назначаться на два соседних шага управления. Сгенерированный MC-ASAP-план показан на рис. 2.6. В отличие от ASAP-плана (см. рис. 2.3), построенного на четырех шагах управления, MC-ASAP-план построен на пяти шагах управления. В то время как в ASAP-плане время шага равно 2, в MC-ASAP-плане время шага равно 1. Время выполнения всего плана сократилось с восьми до пяти единиц. Число используемых процессоров равно 6 в обоих случаях.

Алгоритм MC-ASAP легко трансформируется к алгоритму MC-ALAP. Второй отличается от первого тем, что планирование начинается с последнего шага и заканчивается первым шагом управления.

Шаг управления	Операции			Тип +	Тип *
1	①	③	⑥	3	0
2	②	④	⑦	0	3
3				0	3
4		⑤		1	0
5			⑧	1	0
Максимум =				3	3

Рис. 2.6. Многошаговый план MC-ASAP, построенный на пяти шагах управления, требует три процессора типа «+» и три процессора типа «*», всего шесть процессоров

2.4.2. Многошаговое планирование на базе LS

Алгоритм многошагового спискового планирования MC-LS (*Multi Cycling LS*) минимизирует время выполнения плана при заданных ограничениях на объем используемых вычислительных ресурсов (*resource-constrained scheduling*), учитывая при этом соотношение между временем шага и временами выполнения операций, назначая при необходимости операцию на несколько соседних шагов управления. Ограничения представляются числом доступных процессоров каждого типа. Алгоритм MC-LS может быть построен как на базе стратегии ASAP, так и на базе стратегии ALAP. Остановимся на алгоритме, построенном на базе ASAP.

Исходные данные:

1. Граф G_H предшествования операций;
2. Времена t_i , $i \in N$ выполнения операций;
3. Время t_{step} шага управления;
4. Число p_i доступных процессоров типа $i = 1, \dots, Types$.

Результирующие данные:

1. Шаги управления;
2. Отображение операций на шаги управления.

Описание алгоритма:

1. Планирование выполняется в цикле, начиная с первого и кончая последним шагом управления. Алгоритм использует список *List* готовых к планированию операций. На текущий шаг могут назначаться только операции из *List*;

2. Список *List* состоит из трех частей. Первая часть *List*₁ включает операции, планирование которых начато, но не завершено на предыдущих шагах управления. Вторая часть *List*₂ включает операции, ставшие готовыми к планированию на предыдущих шагах управления, но планирование которых еще не начато. Третья часть *List*₃ включает операции, которые стали готовы к планированию на текущем шаге управления;

3. С каждой операцией $r \in List$ ассоциируется время выполнения τ_r , не покрытое предыдущими шагами управления, на которые операция r частично назначена. В момент начального включения r в *List* время $\tau_r = t_r$.

4. Алгоритм начинает работу до запуска цикла с поиска операций, не имеющих согласно графу G_H операций-предшественников ($pred(r) = \emptyset$), и включения их в часть *List*₃ с начальной установкой значений $\tau_r = t_r$. Части *List*₁ и *List*₂ списка зануляются.

5. Для каждого шага s управления выполняются следующие действия по планированию:

– каждая операция $r \in List_1$ назначается на текущий шаг s управления; далее, если выполняется неравенство $\tau_r \leq T_{step}$, то операция r полностью спланирована и удаляется из списка *List*, в противном случае осуществляется редуцирование времени $\tau_r = \tau_r - T_{step}$ с сохранением r в *List*₁;

– если на планирование операций из *List*₁ потрачена только часть процессоров, то оставшиеся процессоры могут быть использованы для конкурентного планирования операций из *List*₂ и *List*₃;

– на текущий шаг управления s назначаются операции из *List*₂ и *List*₃, выбираемые по критерию предпочтения в количестве, не превышающем число оставшихся процессоров каждого типа; операция r , для которой $\tau_r \leq T_{step}$, назначается на шаг s и исключается из

List, а операция r , для которой $\tau_r > T_{step}$, назначается на шаг s и перемещается в $List_1$ с пересчетом времени $\tau_r = \tau_r - T_{step}$;

- оставшиеся в $List_3$ операции остаются не спланированными и перемещаются в $List_2$;

- в $List_3$ включаются новые еще не спланированные операции, для которых все операции-предшественники оказываются спланированными на предыдущих шагах управления, включая текущий шаг.

6. Алгоритм завершает работу в случае зануления списка *List* и назначения всех операций на шаги управления.

Пример 2.6. Проиллюстрируем работу алгоритма MC-LS на графе G_H , изображенном на рис. 2.2. Примем время шага $T_{step} = 1$ и время выполнения операций $t_+ = 1$ и $t_* = 2$. Допустим также, что для реализации плана предоставлено по одному процессору каждого типа: $pe_+ = 1$ и $pe_* = 1$. Сгенерированный MC-LS-план показан на рис. 2.7.















Шаг управления	Операции	Список <i>List</i>
0		1,3,6
1		3,6 2
2	 	2 6 4
3	  	4 7
4	 	4 7
5		7 5
6	 	7
7	 	8
8		

Рис. 2.7. Многошаговый план MC-LS, построенный на восьми шагах управления при ограничениях $p_+ = 1$ и $p_* = 1$

При выборе одной операции из множества конкурирующих операций использован критерий принадлежности критическому пути на графе G_H . План состоит из восьми шагов управления. Вспомогательный шаг 0 инициализирует список $List$. Две вертикальные черты разделяют список на три части, как описано выше. Не смотря на то, что MC-LS-план содержит восемь шагов, а LS-план содержит только пять шагов, общее время выполнения первого плана с учетом $T_{step} = 1$ составляет восемь единиц, в то время как время выполнения второго плана с учетом $T_{step} = 2$ (по времени самой длинной операции) составляет десять единиц. В результате многошаговое списковое планирование дало выигрыш 20 % по сравнению обыкновенным списковым планированием.

2.5. Цепочечное планирование

Принципы цепочечного планирования:

1. Разрешается выполнение цепочек операций на одном шаге управления;

2. Под цепочкой понимается последовательность операций, в которой соседние операции соединены дугой в графе G_H , описывающей отношение непосредственного предшествования;

3. Время шага управления t_{step} должно быть больше либо равно суммарному времени выполнения операций, входящих в цепочку;

4. Входящие в цепочку операции должны быть назначены на различные процессоры;

Преимущества цепочечного планирования по сравнению с обыкновенным планированием:

1. Сокращение числа шагов управления и общего времени выполнения синтезируемого плана;

2. Увеличение загрузки оборудования при совместном использовании с многошаговым планированием.

Алгоритмы цепочечного планирования строятся путем модификации алгоритмов ASAP, ALAP и LS. Смешанные алгоритмы цепочечного и многошагового планирования строятся путем модификации алгоритмов MC-ASAP, MC-ALAP и MC-LS.

Рассмотрим алгоритм C-ASAP цепочечного планирования, построенный на базе обыкновенного ASAP. Он стремится назначать операции на как можно более ранние шаги управления, однако от-

личается от ASAP тем, что учитывает времена выполнения операций и время шага управления. Алгоритм решает оптимизационную задачу на достижимость минимального числа шагов управления при отсутствии ограничений на ресурсы.

Исходные данные:

1. Граф G_H предшествования операций;
2. Времена $t_i, i \in N$ выполнения операций;
3. Время t_{step} шага управления.

Результирующие данные:

1. Шаги управления;
2. Отображение операций на шаги управления;
3. Число процессоров каждого типа.

Описание алгоритма:

1. Планирование выполняется в цикле, начиная с первого шага и кончая последним шагом управления.

2. На каждом шаге s множество всех операций разбивается на три подмножества: N_s не спланированных операций; N'_s операций, спланированных на всех предыдущих шагах, включая текущий; $N''_s \subseteq N'_s$ операций, назначенных на текущий шаг.

3. С каждой операцией $r \in N''_s$, включенной в текущий шаг управления, ассоциируется время τ_r завершения ее выполнения с момента текущего шага управления.

4. Для каждого шага s управления выполняются следующие действия по планированию:

– организуется цикл по назначению еще не спланированных операций на текущий шаг управления, цикл завершается, если ни одна операция не может быть назначена на текущий шаг управления;

– на каждой итерации цикла из подмножества N_s выбирается операция r такая, что $pred(r) \subseteq N'_s$;

– для операции r вычисляется время завершения выполнения $\tau_r = \rho + t_r$, где $\rho = \max\{\tau_w\}$ по всем $w \in W, W = pred(r) \cap N''_s$; если $W = \emptyset$, то $\rho = 0$;

– если $\tau_r \leq t_{step}$, то операция r назначается на шаг s и перемещается из подмножества N_s в подмножества N'_s и N''_s , в противном случае она остается в подмножестве N_s ;

5. Алгоритм завершает работу в случае назначения всех операций на шаги управления.

Пример 2.7. На рис. 2.8 показан цепочечный план C-ASAP, синтезированный для графа G_H (из рис. 2.2) и заданных времени шага $T_{step} = 2$ и времени выполнения операций $t_+ = 1$ и $t_* = 2$. Для построения плана хватило трех шагов управления. Для его выполнения требуется три процессора типа «+» и три процессора типа «*», всего шесть процессоров. На третий шаг назначена цепочка операторов пять и восемь, относящихся к типу «+». Время цепочки составило две единицы, что соответствует времени шага $T_{step} = 2$. Общее время выполнения плана составляет 6 единиц, что на 25 % меньше времени выполнения обыкновенного плана ASAP.

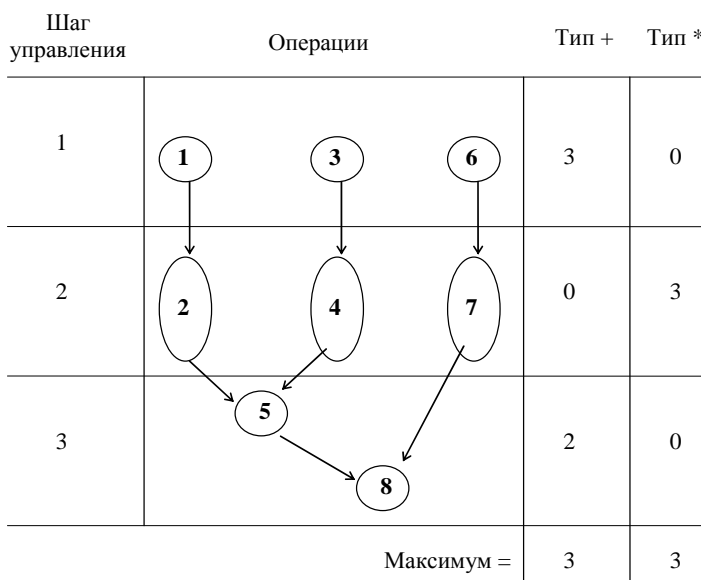


Рис. 2.8. Цепочечный план C-ASAP, построенный на трех шагах управления, требует три процессора типа «+» и три процессора типа «*», всего шесть процессоров

2.6. Сравнение стратегий планирования

Сравним стратегии планирования по двум параметрам планов, синтезированных для рассмотренного примера: времени выполнения и числу используемых процессоров (рис. 2.9).

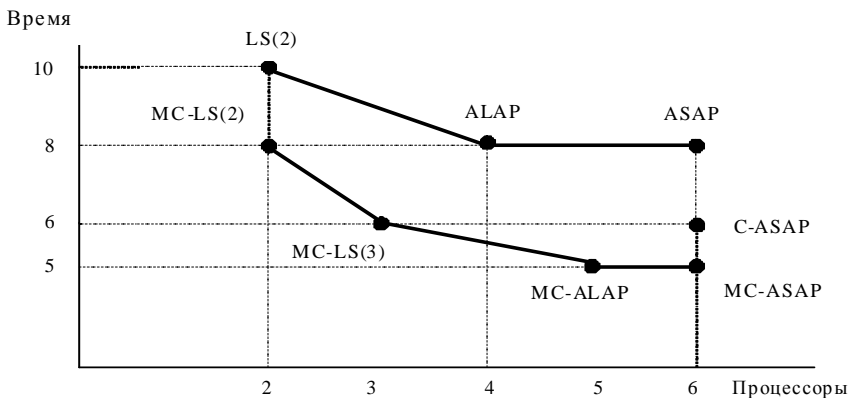


Рис. 2.9. Сравнение стратегий планирования по времени и ресурсам

Каждый план представлен одной точкой в пространстве «процессоры–время», помеченной именем стратегии. В круглых скобках указано число использованных процессоров. Соединение точек линиями дало два графика: первый для стратегий обыкновенного планирования, второй для стратегий многошагового планирования. Стратегии цепочечного планирования представлены одной точкой. Первая закономерность, имеющая место для обоих графиков, состоит в том, что сокращение времени выполнения плана приводит к увеличению числа используемых процессоров. Вторая закономерность состоит в том, что для одинакового числа процессоров стратегии многошагового и цепочечного планирования дали лучшие результаты по сравнению со стратегиями обыкновенного планирования.

Стратегии спискового планирования, будь то обыкновенные, многошаговые или цепочечные, ориентированы на практическое применение, так как они учитывают объем имеющихся у пользователей ресурсов. Планы с минимальным или близким к минимальному числом шагов могут быть синтезированы ими для любого числа процессоров, что приводит к построению области Парето.

2.7. Свертывание графа распараллеленности операций

Пусть $N = \{1, \dots, n\}$ – множество операций. Определим последовательно-параллельный план выполнения операций из множества N рекурсивно, используя суперпозицию двух функций $seq(s_1, \dots, s_k)$

и $par(s_1, \dots, s_k)$, где s_i при $i = 1, \dots, k$ – последовательно-параллельные частичные планы. Функция seq (от *sequential*) называется «последовательный», функция par (от *parallel*) называется «параллельный». Последовательно-параллельный план есть:

1. отдельная операция $i \in N$;
2. план, описываемый функцией $seq(s_1, \dots, s_k)$ последовательного выполнения частичных планов s_1, \dots, s_k ;
3. план, описываемый функцией $par(s_1, \dots, s_k)$ параллельного выполнения частичных планов s_1, \dots, s_k .

Графическое изображение функции seq последовательного плана представлено на рис. 2.10. Графическое изображение функции par параллельного плана представлено на рис. 2.11.

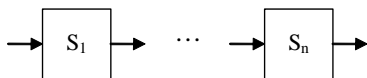


Рис. 2.10. Графическое изображение функции seq

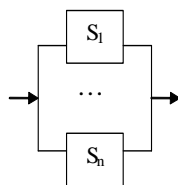


Рис. 2.11. Графическое изображение функции par

Граф распараллеленности операций определяется как $G_R = (V, R)$ [3], где V – множество вершин, являющихся частично-последовательно-параллельными планами и, в частном случае, операциями; R – множество ребер, описывающих отношение распараллеленности частичных планов (операций). Граф является неориентированным. Две вершины $i, k \in V$ соединены ребром, если соответствующие частичные планы выполняются параллельно, если вершины не соединены ребром, соответствующие планы выполняются последовательно. Граф является взвешенным. Каждой вершине $i \in N$ ставится в соответствие два параметра: время T_i выполнения частичного плана (в частном случае, это время t_i выполнения операции) и вектор $b^i = (b^i_1, \dots, b^i_m)$ чисел b^i_j процессоров типов $j = 1, \dots, m$, необходимых для выполнения плана.

Исходный граф G_R^0 на множестве вершин-операций строится по графу G_H . В граф G_R^0 вводится ребро (i, k) , если на ориентированном

графе G_H не существует путь, соединяющий операции i и k , в противном случае ребро не вводится. Для вершины i определяются вес $T_i = t_i$ и вектор $b^i = (0, \dots, 1, \dots, 0)$, где 1 находится в позиции $type_i$.

Пример 2.8. На рис. 2.12 приведен исходный граф G_R^0 распараллеленности операций, построенный на восьми вершинах по графу G_H предшествования операций, показанному на рис. 2.2. Он содержит пять вершин-операций 1, 3, 5, 6 и 8 типа «+» с временем выполнения $t_+ = 1$ и три вершины-операции 2, 4, и 7 типа «*» с временем выполнения $t_* = 2$. Вершины типа «+» метятся векторами $(1, 0)$, а вершины типа «*» метятся векторами $(0, 1)$.

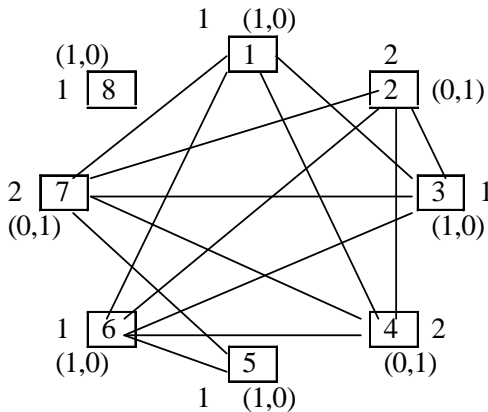


Рис. 2.12. Пример графа G_R распараллеленности операций

Граф распараллеленности G_R^0 является исходными данными для синтеза последовательно-параллельного плана методом свертывания. Свертывание графа G_R^0 выполняется посредством пошагового склеивания пар вершин, при этом в графе появляются более сложные вершины, которым соответствуют частичные последовательно-параллельные планы и которые также метятся двумя интегрированными величинами:

1. временем T_i выполнения частичного плана;
2. вектором b^i чисел процессоров различных типов, необходимых для реализации частичного плана.

Рассмотрим последовательно-параллельное планирование с целью минимизации времени решения задачи при заданных ограничениях на ресурсы. Ограничения представим вектором $b^{\max} = (b_1^{\max}, \dots, b_m^{\max})$ максимального числа используемых процессоров каждого типа.

Свертывание графа G_R посредством склеивания пар вершин. Пусть на шаге s планирования в графе G_R^s для склеивания выбраны вершины x и y . В результате склеивания x и y образуется новая вершина с именем xy . Граф G_R^s преобразуется путем удаления вершин x и y , удаления множества ребер, инцидентных этим вершинам, а также путем добавления вершины xy и добавления ребер, инцидентных этой вершине и вершинам из множества $\chi(x) \cap \chi(y)$, где $\chi(x)$ – множество вершин, смежных с вершиной x в графе G_R^s . Если вершины x и y соединены ребром, они склеиваются параллельно, при этом новой вершине соответствует параллельный план $par(x, y)$. Если вершины не соединены ребром, они склеиваются последовательно, при этом новой вершине соответствует последовательный план $seq(x, y)$. Время выполнения T_{xy} и вектор b^{xy} для новой вершины xy вычисляются следующим образом:

1) если вершины склеиваются параллельно, то $T_{xy} = \max(T_x, T_y)$ и $b^{xy} = b^x + b^y$;

2) если вершины склеиваются последовательно, то $T_{xy} = T_x + T_y$ и $b^{xy} = \max(b^x, b^y) = (\max(b^x_1, b^y_1), \dots, \max(b^x_m, b^y_m))$.

Параллельное склеивание вершин увеличивает число используемых процессоров, последовательное склеивание увеличивает время выполнения плана. Параллельное склеивание вершин не должно приводить к нарушению ограничений на максимальное число процессоров, описываемых вектором b^{\max} . Если нарушение ограничений имеет место, ребро может быть заранее удалено из графа G_R^s .

Различный порядок склеивания пар вершин приводит к построению различных последовательно-параллельных планов. Существенное влияние на качество синтезируемого плана оказывает потеря ребер, обусловленная с одной стороны, особенностями механизма склеиванием вершин и, с другой стороны, учетом ограничения на число процессоров b^{\max} . Потеря ребра при склеивании вершин x и y происходит тогда, когда существует вершина z , смежная с вершиной x и не смежная с вершиной y , т. е. $z \in \chi(x)$ и $z \notin \chi(y)$, либо смежная с вершиной y и не смежная с вершиной x , т. е. $z \notin \chi(x)$

и $z \in \chi(y)$. В результате такого склеивания новая вершина xu и вершина z не соединены ребром. Это означает потерю потенциального параллелизма между вершинами xu и z , что может сказаться отрицательно на минимизации времени выполнения синтезируемого плана. В силу сказанного, предпочтительным является выбор пар вершин, склеивание которых минимизирует потерю ребер.

Склеивание пар вершин и свертывание графа распараллеленности завершается тогда, когда граф включает одну вершину. Этой вершине соответствует суперпозиция функций *seq* и *par*, описывающая синтезированный последовательно-параллельный план.

Пример 2.9. Выполним свертывание графа G_R^0 распараллеленности операторов, изображенного на рис. 2.12 при ограничении на число процессоров $b^{\max} = (2,2)$. Процесс склеивания вершин графа представлен на рис. 2.13. Он состоит из шести шагов. На каждом из первых пяти шагов склеиваются две вершины, на шестом шаге сразу склеиваются три вершины. Именование вновь образуемых вершин показывает характер склеивания, например, получаемая на шаге а) новая вершина именуется 34 и с ней ассоциируется план, заключающийся в последовательном выполнении операторов 3 и 4. В то же время, на шаге в) образуется вершина, именуемая 12/34. С ней ассоциируется план, состоящий в параллельном выполнении (символ «/») последовательных планов 12 и 34.

Опишем каждый шаг свертывания графа детально (рис. 2.13).

Шаг 1. Вершины 3, 4 не соединены ребром и имеют одинаковое множество $\{1, 2, 6, 7\}$ смежных вершин. Следовательно, последовательное склеивание вершин не приводит к потере ребер. Результат склеивания представлен на рис. 2.13, а. Новая вершина 34 метится временем $T_{34} = 1 + 2 = 3$ и вектором $b^{34} = \max((1,0), (0,1)) = (\max(1,0), \max(0,1)) = (1,1)$. Ей соответствует частичный план *seq*(3, 4).

Шаг 2. Вершины 1, 2 не соединены ребром и имеют одинаковое множество $\{34, 6, 7\}$ смежных вершин. Следовательно, как и на предыдущем шаге, последовательное склеивание вершин не приводит к потере ребер. Результат склеивания представлен на рис. 2.13, б. Новая вершина 12 метится временем $T_{12} = 1 + 2 = 3$ и вектором $b^{12} = \max((1,0), (0,1)) = (\max(1,0), \max(0,1)) = (1,1)$. Ей соответствует частичный план *seq*(1, 2).

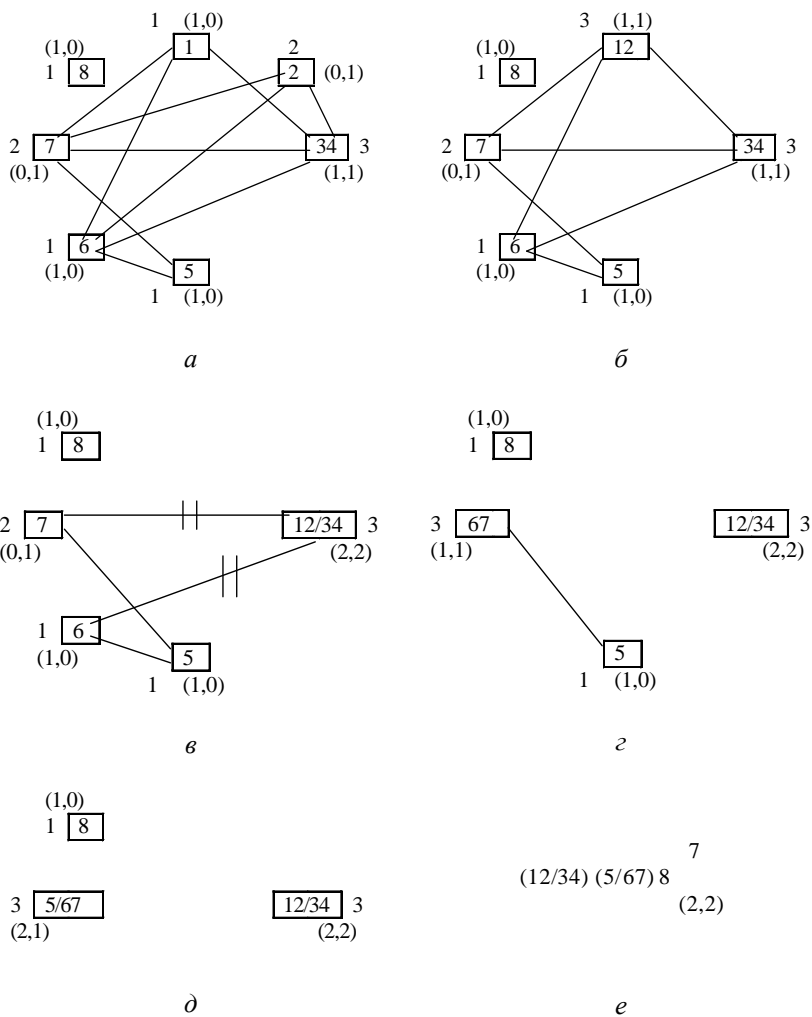


Рис. 2.13. Пошаговый процесс свертывания графа распараллеленности операторов G_R^0

Шаг 3. В графе на рис. 2.13, б вершина 12 имеет окрестность $\{34, 6, 7\}$, а вершина 34 имеет окрестность $\{12, 6, 7\}$. Первые вершины окрестностей представляют ребро (12, 34), а вершины 6, 7 представляют одинаковые оставшиеся части окрестностей. Следо-

вательно, вершины 12 и 34 могут быть склеены параллельно без потери ребер. Результат склеивания показан на рис. 2.13, в. Новая вершина 12/34 метится временем $T_{12/34} = \max(3,3) = 3$ и вектором $b^{12/34} = (1,1) + (1,1) = (2,2)$. Ей соответствует частичный план $par(seq(1, 2), seq(3, 4))$. Заметим, что новая вершина не нарушает ограничение на $b^{\max} = (2, 2)$. Проверим, не нарушает ли это ограничение реализация новых ребер. Реализация ребра (12/34, 6) привела бы к склеиванию соответствующих вершин параллельно и дала бы вектор $b = (3, 2)$, что нарушает ограничение на b^{\max} . Точно также, реализация ребра (12/34, 7) привела бы к появлению вектора $b = (2, 3)$, что также нарушает ограничение на b^{\max} . Как следствие, два ребра удаляются из графа, что показано на рис. 2.13, в двойными перечеркивающими линиями.

Шаг 4. В графе распараллеленности, показанном на рис. 2.13, в, осталось два ребра (5, 7) и (5, 6). Потенциальный параллелизм этих ребер реализуется последовательным склеиванием вершин 6 и 7. Результат склеивания показан на рис. 2.13, г. Новая вершина метится временем $T_{67} = 1 + 2 = 3$ и вектором $b^{67} = \max((1,0), (0,1)) = (\max(1,0), \max(0,1)) = (1,1)$. Ей соответствует частичный план $seq(6, 7)$.

Шаг 5. С целью реализации потенциального параллелизма ребра (5, 67) склеим вершины 5 и 67 параллельно. Результат склеивания представлен на рис. 2.13, д. Новая вершина 5/67 метится временем $T_{5/67} = \max(1,3) = 3$ и вектором $b^{5/67} = (1,0) + (1,1) = (2,1)$. Ей соответствует частичный план $par(5, seq(6, 7))$.

Шаг 6. Полученный на предыдущем шаге граф содержит три вершины и не содержит ребер. Следовательно, все три вершины могут быть склеены последовательно. Результат склеивания представлен на рис. 2.13, е. Новая вершина (12/34) (5/67) 8 метится временем $T_{(12/34)(5/67)8} = 3 + 3 + 1 = 7$ и вектором $b^{(12/34)(5/67)8} = \max((2,2), (2,1), (1,0)) = (2,2)$. Ей соответствует частичный план $seq(par(seq(1, 2), seq(3, 4)), par(5, seq(6, 7)), 8)$. Графический образ этого плана показан на рис. 2.14. Для заданного ограничения на число процессоров $b^{\max} = (2, 2)$ он выполняется за минимальное время $T = 7$.

Последовательно-параллельное планирование вычислительных процессов имеет широкое практическое применение как при проектировании программного обеспечения так и при проектировании цифровой аппаратуры. Оно позволяет синтезировать оптимизированные планы функционирования многопоточных приложений для

многопроцессорных и многоядерных систем. Такого рода планирование позволяет оптимизировать синхронные цифровые системы.

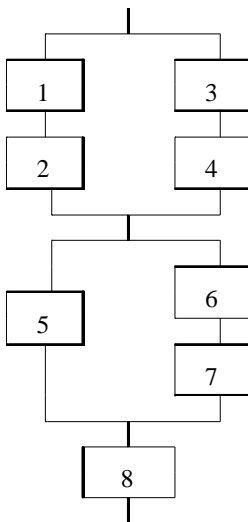


Рис. 2.14. Синтезированный последовательно-параллельный план

2.8. Сведение планирования к задаче целочисленного линейного программирования

Представление планирования в виде задачи целочисленного линейного программирования ILPF является альтернативой стратегии спискового планирования LS. Однако, если LS является эвристическим алгоритмом и находит в общем случае субоптимальное решение, то ILPF является точным методом, гарантирующим нахождение глобального оптимума. Метод может быть сформулирован для обыкновенного, многошагового и цепочечного планирования. Он применим также ко всем постановкам оптимизационных задач. Остановимся на обыкновенном планировании и рассмотрим задачу минимизации стоимости используемых для реализации плана процессоров при заданном числе T шагов управления (*time-constrained scheduling*).

Пусть, как и раньше, G_H – граф предшествования операций, где $N = \{1, \dots, n\}$ – множество операций и $H \subseteq N \times N$ – бинарное отношение предшествования операций, получаемое в результате анализа информационных зависимостей между операциями.

Через S_i обозначим наиболее ранний шаг выполнения операции i в соответствии с графом G_H . Он удовлетворяет неравенству $1 \leq S_i \leq T$ и вычисляется посредством алгоритма ASAP. Через L_i обозначим наиболее поздний шаг выполнения операции i в соответствии с графом G_H и требуемым числом шагов T . Он также удовлетворяет неравенству $1 \leq L_i \leq T$, однако вычисляется посредством алгоритма ALAP. Очевидно что $S_i \leq L_i$. Пара величин S_i и L_i описывает время жизни операции i . Через b_k обозначим число реально используемых в плане процессоров типа $k = 1, \dots, Types$. Стоимость процессора типа k описывается величиной c_k .

Для описания плана введем матрицу X , элемент x_{ij} которой при $i \in N$ и $j = 1, \dots, T$ принимает значение 1 в случае, если операция i выполняется на шаге j , и принимает значение 0 в противном случае. Введенных обозначений достаточно чтобы сформулировать постановку задачи ILPF, описываемую целевой функцией и системой ограничений.

Целевая функция:

$$\min \sum_{k=1}^{Types} (c_k b_k). \quad (2.1)$$

Система ограничений:

$$\left(\sum_{\substack{i \in N, \\ type(i)=k}} x_{ij} \right) - b_k \leq 0 \quad \text{при } 1 \leq j \leq T, \quad 1 \leq k \leq Types, \quad (2.2)$$

$$\sum_{j=S_i}^{L_i} x_{ij} = 1 \quad \text{при } 1 \leq i \leq n, \quad (2.3)$$

$$\sum_{r=S_i}^{L_i} (rx_{ir}) - \sum_{r=S_j}^{L_j} (rx_{jr}) \leq -1 \quad \text{при } (i, j) \in H. \quad (2.4)$$

Целевая функция (2.1) является линейной и зависит от числа и стоимости процессоров типа k . Ограничение (2.2) описывает требование такое, что на каждом шаге j число используемых процессоров

типа k не должно превышать число b_k имеющихся процессоров этого типа. Поскольку значения свободных переменных j и k принадлежат диапазонам $1 \leq j \leq T$ и $1 \leq k \leq Types$, число линейных неравенств в ограничении (2.2) равно $T \times Types$.

Ограничение (2.3) выражает требование выполнения любой операции ровно на одном шаге управления, что справедливо для обыкновенного планирования. Поскольку значение свободной переменной i принадлежит диапазону $1 \leq i \leq n$, число линейных равенств в ограничении (2.3) равно n .

Ограничение (2.4) выражает требование такое, что операция j должна выполняться на шаге с номером большим, чем номер шага, на котором выполняется операция i , в случае, если операция i предшествует операции j в графе G_H . Поскольку число пар операций, для которых имеет место предшествование, равно числу ребер в графе G_H , то число линейных неравенств в ограничении (2.4) равно $|H|$.

Пример 2.10. Применим метод ILPF к графу G_H , показанному на рис. 2.2, с целью минимизации суммарной стоимости процессоров типов «+» и «*» в синтезируемом плане при условии, что стоимости $c_+ = 1$ и $c_* = 5$, а число шагов в плане $T = 5$. Начнем с вычисления времен жизни операторов на шагах управления. Для этого построим планы ASAP и ALAP на заданном числе 5 шагов управления. Они показаны на рис. 2.15. План ASAP прижат к первому шагу, план ALAP прижат к 5 шагу.

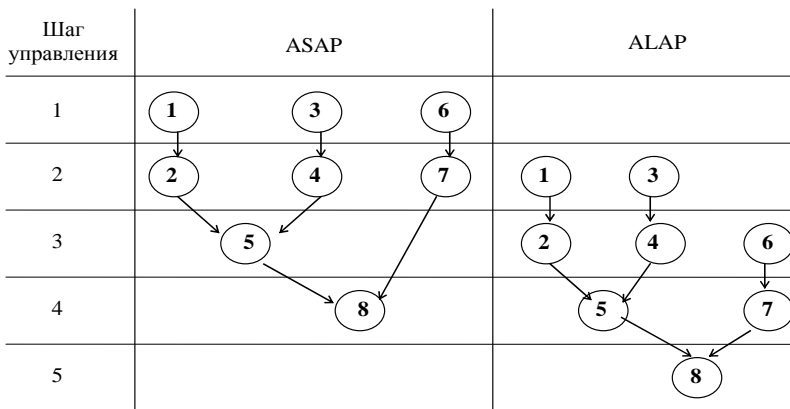


Рис. 2.15. Планы ASAP и ALAP, построенные на 5 шагах управления

Времена жизни операций каждого из двух типов, определенные по планам ASAP и ALAP, описаны в табл. 2.1. Так для операции 1 наиболее раннее время $S_1 = 1$ возьмем из плана ASAP, а наиболее позднее время $L_1 = 2$ – из плана ALAP. Времена жизни других операций вычисляются аналогично. Они влияют на построение матрицы X .

Таблица 2.1

Времена жизни операций

Тип операции	+					*		
Операция	1	3	5	6	8	2	4	7
S_i	1	1	3	1	4	2	2	2
L_i	2	2	4	3	5	3	3	4

Число строк матрицы X равно числу операций и равно 8. Число столбцов равно числу шагов управления и равно 5. Для каждой операции i в матрицу вводится строка, включающая T переменных x_{ij} . Для шагов управления j , лежащих за пределами времени жизни операции, заранее известно, что переменные x_{ij} имеют значение 0, следовательно, соответствующие элементы матрицы X могут быть занулены. В результате получаем матрицу:

$$X = \begin{bmatrix} x_{11} & x_{12} & 0 & 0 & 0 \\ 0 & x_{22} & x_{23} & 0 & 0 \\ x_{31} & x_{32} & 0 & 0 & 0 \\ 0 & x_{42} & x_{43} & 0 & 0 \\ 0 & 0 & x_{53} & x_{54} & 0 \\ x_{61} & x_{62} & x_{63} & 0 & 0 \\ 0 & x_{72} & x_{73} & x_{74} & 0 \\ 0 & 0 & 0 & x_{84} & x_{85} \end{bmatrix}.$$

Обозначая через b_1 число процессоров типа «+», через b_2 – число процессоров типа «*». В соответствии с (2.1) целевую функцию задачи оптимизации запишем в виде

$$\min_{b_1, b_2} (b_1 + 5b_2).$$

Система ограничений строится по соотношениям (2.2)–(2.4). Для данного примера общее число неравенств в ограничении (2.2) на число процессоров типа «+» и «*» равно $5 \cdot 2 = 10$. Однако, с учетом того, что операции типа «*» не входят в шаги управления 1 и 5, число неравенств сокращается до 8:

$$\begin{aligned}x_{11} + x_{31} + x_{61} - b_1 &\leq 0; \\x_{12} + x_{32} + x_{62} - b_1 &\leq 0; \\x_{53} + x_{63} - b_1 &\leq 0; \\x_{54} + x_{84} - b_1 &\leq 0; \\x_{85} - b_1 &\leq 0; \\x_{22} + x_{42} + x_{72} - b_2 &\leq 0; \\x_{23} + x_{43} + x_{73} - b_2 &\leq 0; \\x_{74} - b_2 &\leq 0.\end{aligned}$$

Общее число равенств в ограничении (2.3), описывающем вхождение каждой операции ровно в один шаг управления, равно числу операций и равно 8, а общее число равенств в ограничении (2.4), учитывающем предшествование операций при их назначении на шаги управления, равно числу ребер в графе G_H и равно 7:

$$\begin{aligned}x_{11} + x_{12} &= 1; & x_{11} + 2x_{12} - 2x_{22} - 3x_{23} &\leq -1; \\x_{31} + x_{32} &= 1; & 2x_{22} + 3x_{23} - 3x_{53} - 4x_{54} &\leq -1; \\x_{53} + x_{54} &= 1; & x_{31} + 2x_{32} - 2x_{42} - 3x_{43} &\leq -1; \\x_{61} + x_{62} + x_{63} &= 1; & 2x_{42} + 3x_{43} - 3x_{53} - 4x_{54} &\leq -1; \\x_{84} + x_{85} &= 1; & 3x_{53} + 4x_{54} - 4x_{84} - 5x_{85} &\leq -1; \\x_{22} + x_{23} &= 1; & x_{61} + 2x_{62} + 3x_{63} - 2x_{72} - 3x_{73} - 4x_{74} &\leq -1; \\x_{42} + x_{43} &= 1; & 2x_{72} + 3x_{73} + 4x_{74} - 4x_{84} - 5x_{85} &\leq -1. \\x_{72} + x_{73} + x_{74} &= 1.\end{aligned}$$

Таким образом, сформулированная задача целочисленного линейного программирования состоит из целевой функции и системы ограничений, включающей 23 равенства и неравенства. Задача решается одним из известных методов, разработанных в теории математического программирования. В частности, для решения задачи могут быть использованы метод ветвей и границ, генетический алгоритм. Быстрое решение в вещественной области может быть получено путем использования симплекс-метода с последующим отображением вещественных значений переменных x_{ij} на целочисленные значения.

3. ПЛАНИРОВАНИЕ АСИНХРОННЫХ РАСПРЕДЕЛЕННЫХ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ С УЧЕТОМ ОБМЕНА ДАННЫМИ

Рассмотренные выше стратегии планирования синтезируют и оптимизируют синхронные планы, учитывая, главным образом, характеристики операций, выполняемых параллельно или последовательно на узлах распределенной параллельной системы. Они не учитывают асинхронный обмен данными между операциями и между процессорами, на которых операции выполняются.

В настоящей главе рассматриваются модели и алгоритмы планирования, учитывающие параметры операций, выполняемых на узлах системы, и параметры операций, выполняемых в каналах передачи данных, которые соединяют узлы [5]. Сначала предполагается, что время выполнения операции не зависит от параметров процессора, на который эта операция назначается, а время выполнения операции обмена данными не зависит от параметров канала передачи данных, посредством которого выполняется обмен. Эти допущения справедливы для однородных распределенных систем. В неоднородных распределенных системах времена выполнения всех операций могут изменяться в зависимости от параметров оборудования, на котором эти операции выполняются.

3.1. Граф задач

Граф задач (task graph) – это ориентированный ациклический взвешенный граф $G = (V, E)$, в котором V – множество узлов, представляющих задачи, E – множество дуг, представляющих передачу данных и отношение предшествования между задачами. Задача определяется как множество инструкций (операторов), выполняемых последовательно на одном процессоре. Соответствующая вершина n_i графа метится числом $w(n_i)$, описывающим время решения задачи. Дуга (n_i, n_j) графа метится числом $c(i, j)$, описывающим время передачи данных от задачи n_i к задаче n_j .

Входным называется узел, не имеющий входящих дуг. Выходным называется узел, не имеющий исходящих дуг. Остальные узлы называются промежуточными. Задача не может начать выполнение, не получив данные от родительских задач. Следовательно, условием запуска задачи на выполнение является завершение выполнения

всех задач-предшественников на графе задач. При выполнении этого условия задача немедленно запускается на выполнение, следовательно, поведение графа задач в период выполнения является асинхронным. Решение одних задач может происходить параллельно с передачами данных между другими задачами.

Важнейшей характеристикой графа задач является коэффициент «коммуникация/вычисление» (*communication-to-computation ratio*), определяемый как отношение среднего времени передачи данных от одной задачи к другой к среднему времени решения одной задачи. Время реализации графа задач определяется суммарным весом всех узлов и дуг, входящих в наиболее длинный путь на графе.

Пример 3.1. Показанный на рис. 3.1 граф задач включает четыре задачи n_1, n_2, n_3, n_4 , имеющие время решения 5, 20, 10 и 8 единиц соответственно. Время передачи данных между задачами n_1, n_2 составляет 1 единицу; между задачами n_1, n_3 – 20 единиц; n_2, n_4 – 1 единицу; между задачами n_3, n_4 – 10 единиц. Наиболее длинный (критический) путь на графе включает три задачи n_1, n_3, n_4 и две дуги $(n_1, n_3), (n_3, n_4)$. Отсюда время реализации графа задач равно $5 + 20 + 10 + 10 + 8 = 53$. Суммарное время решения всех задач равно 43, суммарное время всех передач данных равно 32, коэффициент «коммуникация/вычисление» равен 0,73.

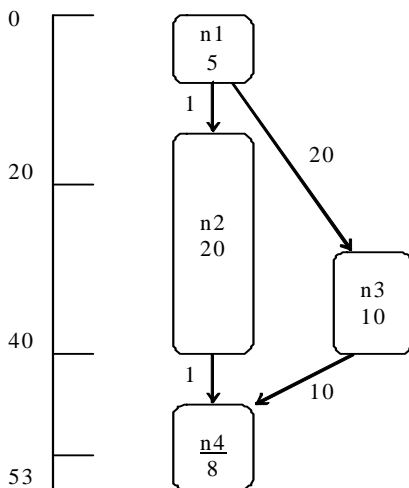


Рис. 3.1. Пример графа задач

3.2. Асинхронный параллельный план

Асинхронный параллельный план генерируется в результате назначения задач на процессоры и назначения передач данных между задачами на каналы передачи данных, соединяющие процессоры. Структура плана учитывает множество $P = \{1, \dots, k\}$, где k – количество процессоров в распределенной параллельной системе, и топологию $Y = \{(v, u) \mid v, u \in P\}$ сети связи, описываемую множеством пар процессоров, соединенных каналами передачи данных. Для каждого процессора $p \in P$ план определяет подмножество задач $V_p \subseteq V$, решаемых на этом процессоре. Дуги графа задач, соединяющие задачи, назначаемые планом на один процессор, не показываются в плане, так как время передачи данных между двумя задачами, равное c_{ij} , принимается равным нулю при назначении задач на один процессор. Задачи $i, j \in V$, соединенные в графе G ребром $(i, j) \in E$, могут быть назначены на разные процессоры $v, u \in P$ такие, что они соединены каналом передачи данных: $(v, u) \in Y$. Такое ребро показывается в плане и метится временем c_{ij} передачи данных. Все задачи V_p , назначаемые на один процессор p , упорядочиваются во времени, и для каждой задачи $i \in V_p$ определяется момент t_i ее запуска на выполнение, при этом учитываются моменты завершения всех задач-предшественников и времена передачи данных от задач, назначенных на другие процессоры.

Время функционирования каждого процессора делится на слоты полезного времени решения задач и бесполезные слоты времени простаивания в ожидании завершения задач-предшественников или ожидания передачи данных от задач-предшественников. Первые слоты определяют полезную загрузку каждого из процессоров.

Пример 3.2. На рис. 3.2 изображен асинхронный параллельный план, построенный по графу задач, показанному на рис. 3.1.

План построен для двух процессоров PE0 и PE1. На процессор PE0 назначены задачи n_1, n_3 и n_4 , на процессор PE1 назначена задача n_2 . Дуги (n_1, n_3) и (n_3, n_4) , присутствующие в графе задач, в плане не показаны, а их веса занулены, так как задачи n_1, n_3, n_4 назначены на один процессор. Две другие дуги (n_1, n_2) и (n_2, n_4) каждая с весом 1 в плане показаны, так как задачи n_1, n_4 с одной стороны и задача n_2 с другой стороны назначены на разные процессоры PE0 и PE1. Для

каждой задачи определяется время запуска. Время запуска задачи n_1 равно 0. Задача n_3 запускается по завершении задачи n_1 в момент времени 5. Задача n_2 запускается по завершении задачи n_1 и завершении передачи данных от n_1 с задержкой 1, в сумме в момент времени 6. Задача n_4 запускается после завершения задачи n_3 в момент времени 15 и завершения передачи данных от n_2 в момент времени 27, в итоге по максимальному времени 27. Общее время выполнения плана определяется максимальным временем работы каждого из двух процессоров. С учетом собственного времени решения задачи n_4 оно равно 35 единицам. Процессор PE0 тратит 23 единицы времени из 35 на решение трех задач, его полезная загрузка составляет 65,7%. Процессор PE1 тратит 20 единицы времени из 35 на решение задач, его полезная загрузка составляет 57,1%.

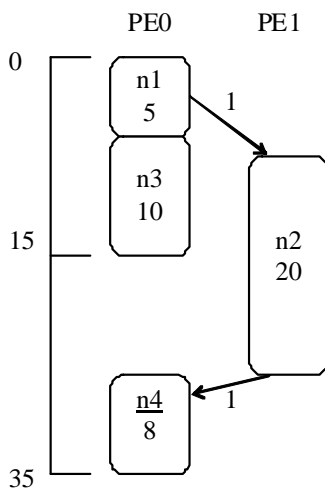


Рис. 3.2. Пример асинхронного параллельного плана

3.3. Задача минимизации временной длины плана

Назначение задач на процессоры трансформирует граф задач в асинхронный план. Планирование задач может выполняться как при наличии, так и отсутствии ограничений на число процессоров. В отличие от стратегий планирования, рассмотренных выше и применимых к графу предшествования операций, стратегии планиро-

вания, применимые к графу задач, не обязательно используют все имеющиеся процессоры для генерации самого быстрого плана. Это происходит в силу наличия накладных расходов на обмен данными между задачами. В предельном случае самым быстрым может оказаться план, построенный только на одном процессоре.

Целью планирования является назначение задач на процессоры, минимизирующее длину плана при заданных ограничениях на число процессоров. Длина плана на многопроцессорной системе определяется как максимальная длина планов по всем процессорам. При этом предполагается, что каждая из задач может решаться на каждом из процессоров. План является эффективным, если его временная длина мала, а использование каждого процессора является обоснованным.

Известно несколько подходов к эффективному решению задачи планирования. Стратегия спискового планирования на графе задач базируется на построении упорядоченного списка задач путем присвоения задачам приоритета. Упорядочение задач в списке может быть статическим и динамическим. Статическое упорядочение выполняется один раз перед началом планирования. Динамическое переупорядочение повторяется после каждого шага планирования. Динамическое планирование выполняется в цикле, на каждой итерации которого:

- 1) определяются приоритеты и новый порядок задач в списке;
- 2) выбирается задача с наивысшим приоритетом;
- 3) выбирается процессор, на который задача назначается.

Основным критерием выбора процессора, на который назначается задача, является наиболее раннее время начала решения задачи. Эта локальная эвристика хорошо коррелирует со стремлением синтезировать план минимальной длины.

Критический путь – важное понятие, используемое стратегиями планирования. Критический путь на графе задач есть последовательность вершин и дуг, формирующих путь от входной вершины до выходной вершины, для которого сумма времен решения задач и времен передачи данных является максимальной. Сумма времен решения задач, лежащих на критическом пути, дает нижнюю границу временной длины плана.

К известным стратегиям планирования относятся:

- 1) наиболее ранняя задача первая (*Earliest Task First* – ETF);
- 2) зануление дуг (*Edge Zeroing* – EZ);

- 3) группировка доминирующей последовательности» (*Dominant Sequence Clustering – DSC*);
 - 4) управление мобильностью (*Mobility Directed – MD*);
 - 5) модифицированный критический путь (*Modified Critical Path – MCP*);
 - 6) планирование на основе динамических уровней (*Dynamic Level Scheduling – DLS*);
 - 7) и другие.
- Рассмотрим в деталях четыре первые стратегии.

3.4. Стратегия планирования «Ранняя задача первая»

Стратегия планирования ETF «наиболее ранняя задача первая» использует статические приоритеты задач и учитывает ограничение на число процессоров. Тем не менее, задача с более высоким приоритетом не обязательно планируется перед задачей с менее высоким приоритетом. Причиной является то, что стратегия ETF на каждом шаге вычисляет наиболее раннее время запуска всех готовых к планированию задач и выбирает задачу с минимальным временем запуска. Задача готова к планированию, если все предшествующие задачи уже спланированы. Наиболее раннее время запуска задачи вычисляется путем оценивания времен запуска задачи на всех возможных процессорах, включая ранее не использовавшийся процессор. Если две задачи имеют одинаковое наиболее раннее время запуска, выбирается задача с более высоким статическим приоритетом.

Статические приоритеты вычисляются на основе применения к графу задач стратегии ALAP и оценивания наиболее поздних времен запуска задач. Приоритеты представляются целыми числами, начиная с 1. Задача с меньшим временем запуска получает больший приоритет, представленный меньшим целым числом. Стратегия ALAP вычисляет наиболее позднее время запуска и завершения решения каждой задачи, начиная с выходных вершин и кончая входными вершинами графа, а также с учетом определения времен запуска и моментов начала передач данных задачам-последователям.

Пример 3.3. Рассмотрим стратегию ETF на примере графа задач, изображенного на рис. 3.1. На рис. 3.3 показаны результаты применения алгоритма ALAP к графу задач. На шкале слева отображено

наиболее позднее время запуска каждой из четырех задач. В табл. 3.1 задачи упорядочены по возрастанию времени, и каждой из них присвоен статический приоритет в возрастании от n_1 до n_4 .

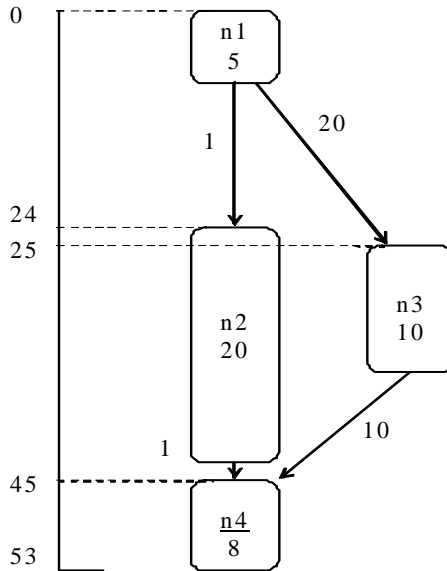


Рис. 3.3. Результаты применение алгоритма ALAP к графу задач

Таблица 3.1

Результаты использования ALAP

Задача	n_1	n_2	n_3	n_4
Наиболее позднее время начала выполнения задачи	0	24	25	45
Приоритет	1	2	3	4

Построение плана происходит на четырех шагах по числу вершин в графе задач.

Шаг 1. Задача n_1 готова к планированию, имеет наивысший статический приоритет 1 и соответственно минимальное время начала запуска на выполнение. Поскольку ни одного процессора еще не

задействовано, вводим процессор PE0 и назначаем на него вершину n_1 (рис. 3.4).

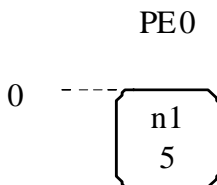


Рис. 3.4. Шаг 1 стратегии ETF

Шаг 2. Задачи n_2 и n_3 готовы к планированию. Наиболее раннее время запуска обеих задач равно 5 при условии, что каждая из них назначается на процессор PE0, при этом дуги с весами 1 и 20 зануляются. Задача n_2 имеет более высокий статический приоритет 2 по сравнению с задачей n_3 , имеющей приоритет 3, поэтому для планирования на шаге 2 выбирается задача n_2 . Поскольку один процессор PE0 уже задействован, рассматриваем также попытку введения второго процессора PE1 и исследуем два варианта I и II назначения узла n_2 на процессор PE0 и на новый процессор PE1 (рис. 3.5). В варианте I время запуска задачи n_2 равно 5. В варианте II время ее запуска равно 6. Следовательно, предпочтительным является вариант I, его и выбираем для дальнейшего рассмотрения.

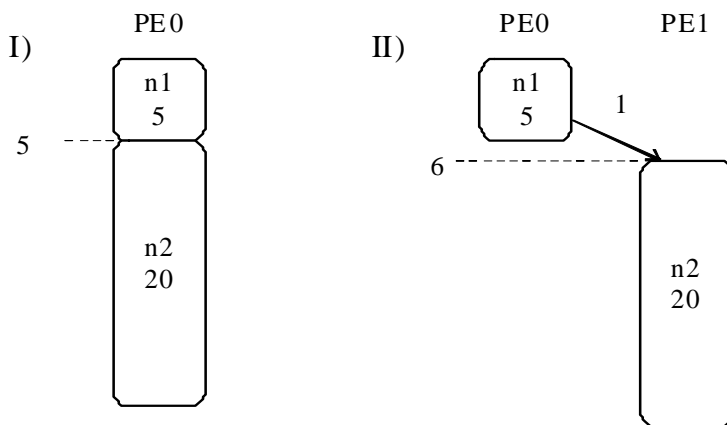


Рис. 3.5. Шаг 2 стратегии ETF

Шаг 3. Задача n_3 готова к планированию, имеет статический приоритет 3 и наиболее позднее время запуска 25. Поскольку задачи n_1 и n_2 назначены на шаге 2 на процессор PE0, рассматриваем попытку введения второго процессора и снова исследуем два варианта I и II назначения узла n_3 на процессор PE0 и на новый процессор PE1 (рис. 3.6). В варианте I, так же как и в варианте II, время запуска задачи n_3 равно 25. Для дальнейшего рассмотрения выбираем вариант I, поскольку он использует только один процессор.

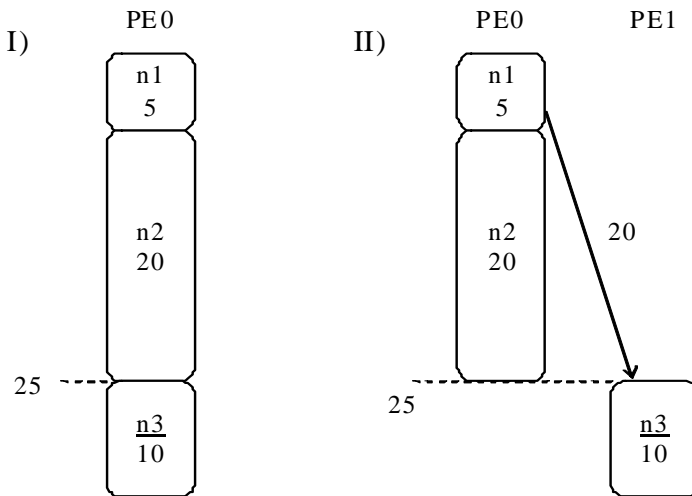


Рис. 3.6. Шаг 3 стратегии ETF

Шаг 4. Задача n_4 готова к планированию, имеет наиболее раннее время запуска 35, статический приоритет 4 и наиболее позднее время начала выполнения 45. Поскольку задачи n_1 , n_2 и n_3 назначены на процессор PE0, рассматриваем попытку введения второго процессора PE1 и исследуем два варианта I и II назначения узла n_4 на процессор PE0 и на новый процессор PE1 (рис. 3.7). В варианте I время запуска задачи n_4 равно 35. В варианте II время запуска равно 45.

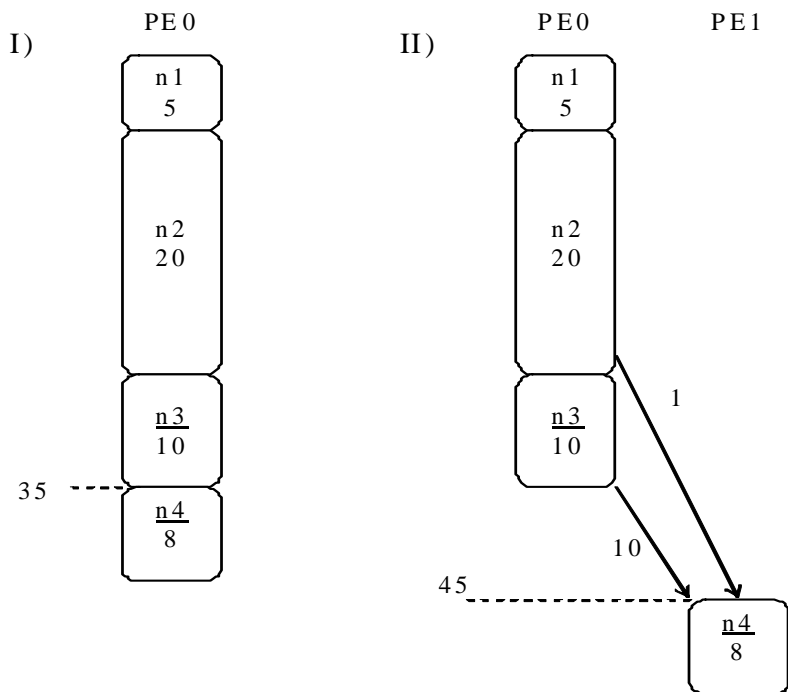


Рис. 3.7. Шаг 4 стратегии ETF

В качестве искомого асинхронного плана выбираем вариант I, поскольку он имеет меньшее время запуска задачи n_4 . Как показано на рис. 3.8, время реализации результирующего плана равно 43 единицам при 100 % загрузке процессора PE0. Достоинством плана является исключение затрат времени на обмен данными между задачами и между процессорами.

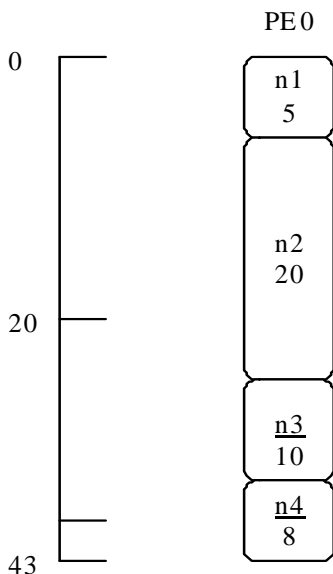


Рис. 3.8. План EFTF, построенный на 1 процессоре

3.5. Стратегия планирования «Зануление дуг»

Стратегия планирования EZ «зануление дуг» стремится сократить длину частично построенного асинхронного плана на каждом шаге планирования путем рассмотрения дуги с максимальным временем передачи данных. Стратегия назначает две задачи, соединенные наиболее «тяжелой» дугой, на один и тот же процессор при условии, что время частичного плана не увеличивается по сравнению с назначением задач на разные процессоры. Если время увеличивается, задачи назначаются на разные подходящие процессоры.

Стратегия EZ сначала строит список дуг, упорядочивая их в невозрастающем (убывающем) порядке весов, описывающих времена передачи данных. Первая дуга удаляется из списка, а инцидентные узлы-задачи назначаются на один и тот же либо на разные процессоры. Если задачи назначаются на один процессор, дуга зануляется, что интерпретируется как немедленный запуск последующей задачи после завершения предыдущей задачи с нулевым временем обмена данных на одном процессоре. Задачи, назначенные на один процес-

сор, упорядочиваются в соответствии с отношением предшествования и по возрастанию их уровня в графе задач. Процесс планирования заканчивается, когда все задачи назначены на процессоры.

Число шагов работы стратегии меньше числа задач в графе, поскольку рассматриваемое на каждом шаге зануление одной дуги приводит к назначению на процессоры одной или сразу двух задач. Для выбора процессора, на который назначается задача, используется критерий наиболее раннего времени запуска задачи или критерий наиболее короткого во времени частичного плана.

Пример 3.4. Продемонстрируем работу стратегии зануления дуг на примере графа задач, показанного на рис. 3.1. Дуги (n_1, n_3) , (n_3, n_4) , (n_1, n_2) , (n_2, n_4) упорядочиваются в списке согласно убыванию их весов: 20, 10, 1, 1. Процесс планирования состоит из следующих шагов, на каждом из которых выбирается первая из оставшихся в списке дуг, над которой выполняются следующие действия.

Шаг 1. Первой в списке стоит дуга (n_1, n_3) с весом 20. Возможны два варианта назначения задач n_1 и n_3 на процессоры (рис. 3.9). В варианте I обе задачи назначаются на процессор PE0. В варианте II задача n_1 назначается на процессор PE0, задача n_3 – на процессор PE1. Вариант I является предпочтительным, поскольку длина плана равна 15, что значительно меньше длины плана 35 в варианте II.

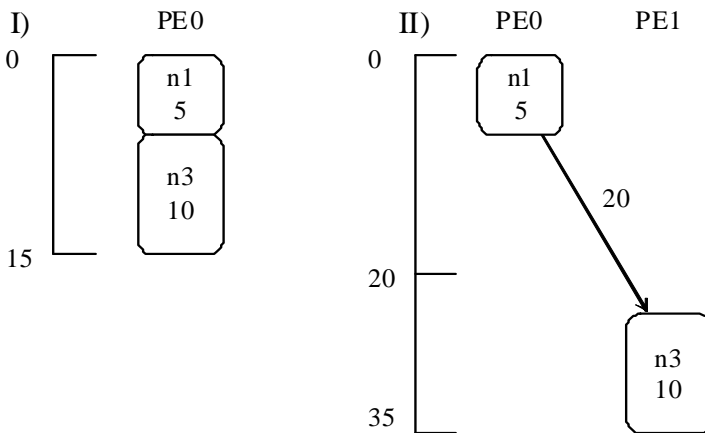


Рис. 3.9. Шаг 1 стратегии EZ

Шаг 2. Дуга (n_3, n_4) с весом 10 стоит в списке на втором месте. Она выбирается для планирования задач на шаге 2. Возможны два варианта назначения задач n_3 и n_4 на процессоры (рис. 3.10). В варианте I обе задачи назначаются на один процессор, в данном случае на процессор PE0, на который уже назначена задача n_3 . В варианте II задача n_3 назначается на процессор PE0, задача n_4 – на новый процессор PE1. Длина плана в варианте I равна 23, длина плана в варианте II равна 33. Для дальнейшего планирования выбираем вариант I.

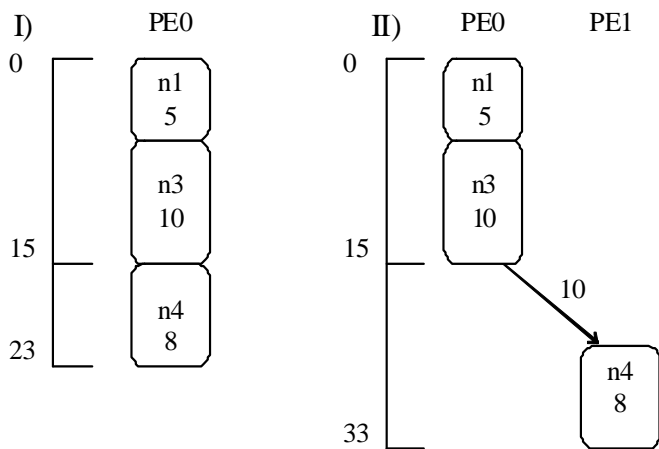


Рис. 3.10. Шаг 2 стратегии EZ

Шаг 3. Дуга (n_1, n_2) с весом 1 стоит в списке на третьем месте. Она выбирается для планирования на шаге 3. Возможны два варианта назначения задач n_1 и n_2 на процессоры (рис. 3.11). В варианте I обе задачи назначаются на один процессор, в данном случае на процессор PE0, на который задача n_1 была назначена ранее. В варианте II задача n_1 остается назначенной на процессор PE0, задача n_2 назначается на новый процессор PE1. Вариант I дает длину плана 43. Вариант II дает длину плана 35. Выбираем вариант II. Этот вариант является искомым асинхронным планом, реализуемым на двух процессорах, поскольку все задачи уже назначены на процессоры, не смотря на то, что в списке осталась одна дуга (n_2, n_4) .

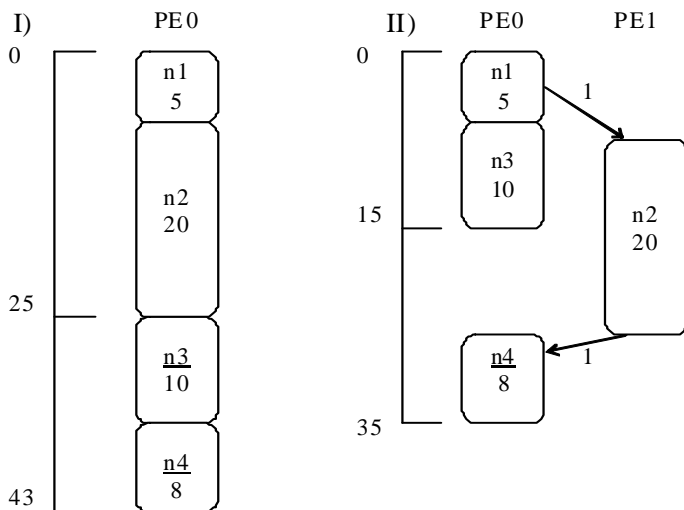


Рис. 3.11. Шаг 3 стратегии EZ

Сравнивая стратегии ETF и EZ, видим, что вторая стратегия дает план меньшей длины: 35 единиц вместо 43 единиц. При этом вторая стратегия использует два процессора вместо одного, используемого первой стратегией. Стратегия ETF загружает единственный процессор PE0 на 100 %, в то время как стратегия EZ загружает процессор PE0 на 65,7 %, процессор PE1 на 57,1 %.

3.6. Стратегия планирования «Группировка доминирующей последовательности»

Стратегия DSC «группировка доминирующей последовательности» не устанавливает ограничений на число процессоров и базируется на использовании понятия доминирующей последовательности, которое по сути является критическим путем в частично спланированном графе задач. На каждом шаге планирования DSC стратегия проверяет, готов ли к планированию первый (верхний) узел критического пути. Если да, стратегия DSC назначает соответствующую задачу на процессор, обеспечивающий наиболее раннее время запуска. Минимальное время запуска может быть достигнуто перепланированием некоторых родительских задач на тот же процессор. С другой стороны, если верхний узел критического пути не

готов к планированию, стратегия DSC не выбирает его для планирования на текущем шаге. Вместо него стратегия выбирает верхний узел, предшествующий первому узлу критического пути. Узел назначается на процессор, дающий минимальное время запуска при условии, что такой выбор процессора не задержит время запуска еще не спланированного верхнего узла критического пути. Отложенное планирование узлов критического пути позволяет стратегии DSC готовить к планированию в убывающем порядке следующий верхний узел критического пути.

Пример 3.5. Продемонстрируем работу стратегии DSC на примере графа задач, показанного на рис. 3.1. Построение асинхронного плана происходит на четырех шагах, поскольку у DSC число шагов планирования равно числу узлов в исходном графе задач.

Шаг 1. Как показано на рис. 3.1, на критическом пути графа лежат задачи n_1, n_3, n_4 . Задача n_1 является верхней в критическом пути, и она готова к планированию. Поскольку ни одного процессора еще не задействовано, назначаем задачу n_1 на процессор PE0 (рис. 3.12).

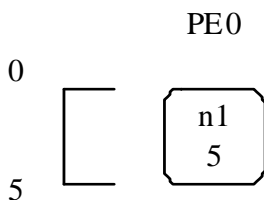


Рис. 3.12. Шаг 1 стратегии DSC

Шаг 2. Следующей не спланированной задачей, лежащей на критическом пути, является задача n_3 . Она готова к планированию, поскольку имеет одного предшественника – задачу n_1 , которая уже назначена на процессор PE0. Поскольку один процессор уже задействован задачей n_1 , рассматриваем попытку введения второго процессора и исследуем два варианта I и II назначения узла n_3 на процессор PE0 и на новый процессор PE1 (рис. 3.13). В варианте I длина частичного плана равна 15. В варианте II длина плана равна 35 из-за длительного обмена данными между задачами. Следовательно, предпочтительным является вариант I, его и выбираем для дальнейшего рассмотрения.

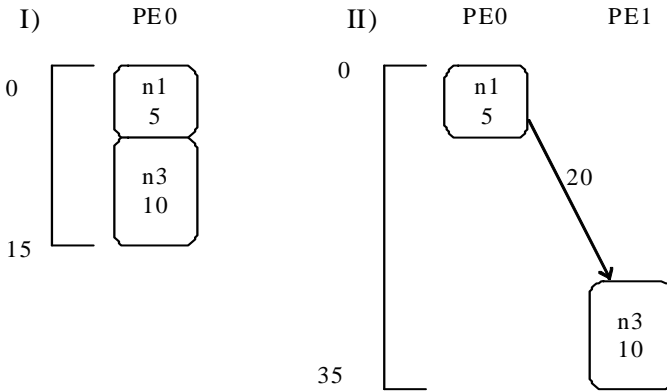


Рис. 3.13. Шаг 2 стратегии DSC

Шаг 3. Следующей не спланированной задачей, лежащей на критическом пути, является задача n_4 . Соответствующая вершина графа имеет две входящие дуги. Одна исходит из задачи n_3 , которая уже спланирована. Другая исходит из задачи n_2 , которая еще не спланирована. Значит, задача n_4 не готова к планированию. Для того, чтобы сделать задачу n_4 готовой к планированию, для планирования выбираем задачу n_2 , которая готова к планированию. Поскольку один процессор уже задействован задачами n_1, n_3 , рассматриваем попытку введения второго процессора и исследуем два варианта I и II назначения узла n_2 на процессор PE0 и на новый процессор PE1 (рис. 3.14).

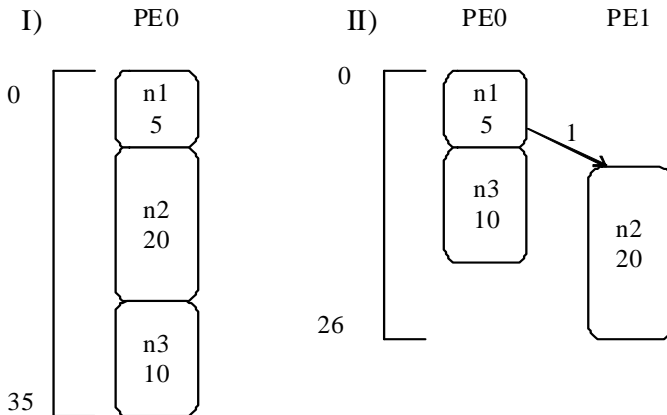


Рис. 3.14. Шаг 3 стратегии DSC

В варианте I задача n_2 вставляется между задачами n_1 и n_3 , уже назначенными на процессор PE0. При этом длина получаемого частичного плана равна 35. В варианте II длина частичного плана равна 26. Следовательно, предпочтительным является вариант II, его и выбираем для дальнейшего рассмотрения.

Шаг 4. Теперь задача n_4 готова к планированию. Поскольку на шаге 3 процессор PE0 использован задачами n_1 и n_3 , а процессор PE1 использован задачей n_2 , рассматриваем попытку введения третьего процессора и исследуем три варианта I, II и III назначения узла n_4 на процессор PE0, процессор PE1 и на новый процессор PE2 (рис. 3.15). В вариантах I и III длина частичного плана равна 35. В варианте II длина частичного плана равна 34. Следовательно, предпочтительным является вариант II, его и выбираем в качестве искомого асинхронного плана, реализуемого на двух процессорах.

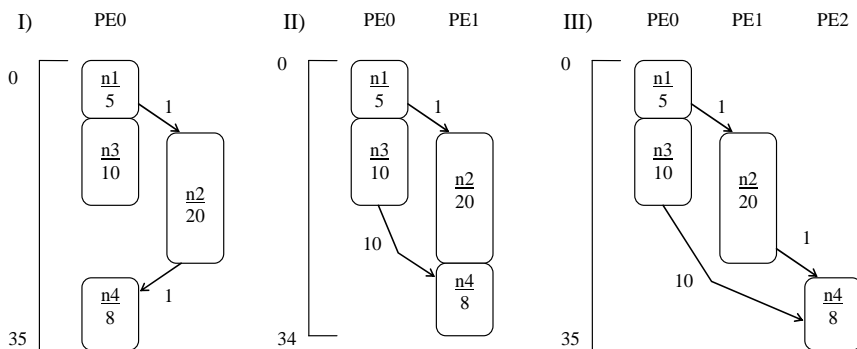


Рис. 3.15. Шаг 4 стратегии DSC

Сопоставляя стратегию DSC со стратегией EZ, заключаем, что стратегия DSC сократила длину плана на 1 за счет более тщательного анализа вершин, принадлежащих критическому пути, и стремления начать выполнение задач критического пути как можно раньше при разумном использовании процессоров. Сокращения длины плана удалось достичь при одном и том же числе используемых процессоров.

Стратегия DSC загружает процессор PE0 на 44,1 %, процессор PE1 на 82,4 %, в сумме на 126,5 %. В то же время стратегия EZ загружает процессоры PE0 и PE1 суммарно на 122,8 %. Стратегия DSC предпочтительнее стратегии EZ и по этому показателю.

3.7. Планирование с использованием мобильности задач

Стратегия MD «управляемое мобильностью планирование» выполняет планирование задач в порядке возрастания их относительной мобильности. Обоснование такого подхода базируется на сопоставлении выбора варианта плана для задачи с низкой мобильностью и задачи с высокой мобильностью. Очевидно, что задача с низкой мобильностью имеет меньше вариаций в выборе плана по сравнению с задачей с высокой мобильностью. Более раннее планирование задачи с низкой мобильностью повышает ее шансы быть эффективно спланированной.

Абсолютная мобильность задачи есть разность времени наиболее позднего и времени наиболее раннего начала выполнения задачи. Время наиболее раннего запуска задачи на выполнение определяется стратегией ASAP, время наиболее позднего запуска операции определяется стратегией ALAP. Обе стратегии должны быть адаптированы к графу задач. Относительная мобильность определяется делением абсолютной мобильности на время решения задачи. Задача с наименьшей относительной мобильностью назначается на первый по порядку процессор, имеющий достаточный временной слот. После назначения выбранной задачи на процессор относительная мобильность оставшихся задач пересчитывается с учетом частично синтезированного плана.

Пример 3.6. Рассмотрим стратегию MD на примере графа задач, показанного на рис. 3.1. К графу применяется стратегия ASAP с целью определения наиболее ранних моментов времени начала выполнения задач (рис. 3.16). Наиболее поздние моменты времени начала выполнения задач определяются посредством стратегии ALAP (см. рис. 3.3). Результаты применения стратегий помещены в строки 2 и 3 (табл. 3.2). Они позволяют рассчитать абсолютную и относительную статическую мобильность всех задач (строка 4 табл. 3.2). Деление абсолютной мобильности на время решения задачи дает относительную мобильность задачи (строка 6 табл. 3.2). Упорядочение задач (n_1, n_3, n_4, n_2) выполняется с учетом возрастания относительной мобильности.

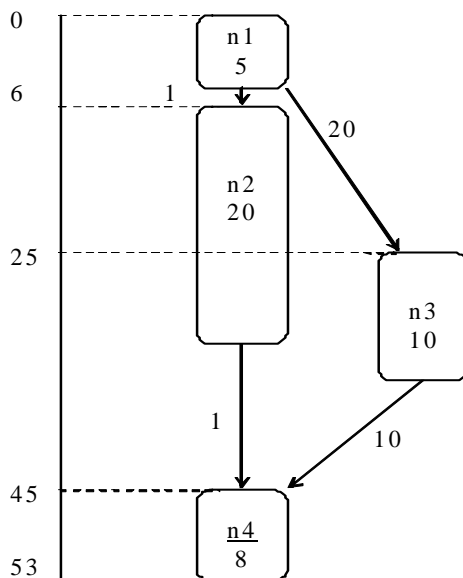


Рис. 3.16. Применение стратегии ASAP к графу задач

Таблица 3.2

Мобильность задач

Задача	n_1	n_2	n_3	n_4
Наиболее раннее время запуска	0	6	25	45
Наиболее позднее время запуска	0	24	25	45
Абсолютная мобильность	0	18	0	0
Время решения задачи	5	20	10	8
Относительная мобильность	0	0.9	0	0

Интересно заметить, что задачи, лежащие на критическом пути, имеют нулевую абсолютную и нулевую относительную мобильность.

Построение плана MD происходит на четырех шагах. После шага 2 не спланированные задачи n_2 , n_4 имеют относительную мобильность 0, следовательно, на шаге 3 планируется операция n_2 , на шаге 4 – операция n_4 . Результирующий план показан на рис. 3.17.

Для задачи n_2 не находится свободного слота времени на процессоре PE0, поэтому она назначается на процессор PE1. Напротив, для задачи n_4 имеется свободный слот времени на процессоре PE0, на который она и назначается.

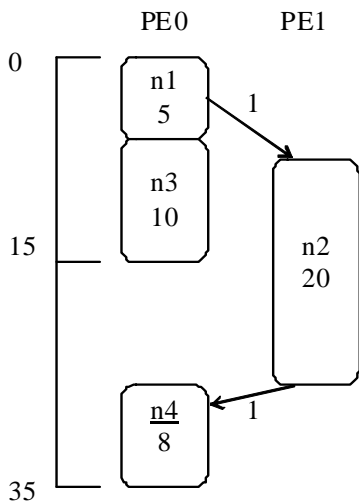


Рис. 3.17. План стратегии MD

4. ПЛАНИРОВАНИЕ РЕШЕНИЯ ЗАДАЧ В РАЗНОРОДНОЙ РАСПРЕДЕЛЕННОЙ СИСТЕМЕ

Реальные распределенные системы являются типично разнородными [5–7]. Разнородность системы обусловлена двумя главными причинами:

1) разнородностью процессоров, находящихся в узлах, и разнородностью программного обеспечения, установленного на этих процессорах;

2) разнородностью каналов передачи данных, соединяющих процессоры.

Разнородность процессоров и каналов передачи данных проявляется в вариациях значений их главных параметров в широких диапазонах. Так тактовые частоты процессоров и скорости в каналах передачи данных могут изменяться в разы.

Как следствие, предположение о константном времени решения задач и константном времени передачи данных от одной задачи к другой не совсем соответствует свойствам реальной разнородной системы. В настоящем разделе мы рассматриваем задачу планирования вычислений в распределенной системе в предположении, что время решения задачи изменяется при переходе от одного процессора к другому.

Высокая вычислительная сложность поиска оптимального плана обусловлена тем, что всего возможно n^m вариантов назначения m задач на n процессоров. Полный перебор всех вариантов для реальных графов задач практически невозможен. Так, если мы имеем 50 задач и 5 процессоров, число вариантов равно 5^{50} , а это является необозримо большим числом. Следовательно, необходимы «разумные» стратегии сокращения перебора вариантов, позволяющие найти за реальное время оптимальный или близкий к оптимальному план решения задач.

4.1. Модель разнородной системы

Целью построения модели является, в конечном счете, поиск отображения графа задач на распределенную систему процессоров, минимизирующего суммарное время решения всех задач с учетом их параллельного асинхронного выполнения в многопроцессорной системе. Предполагается, что процессоры соединены сетью произвольной топологии.

Вместо модели графа задач, рассмотренной ранее, воспользуемся моделью графа $G_T = (V_T, E_T)$ взаимодействия задач, где $V_T = \{t_1, \dots, t_m\}$ – множество задач-вершин; E_T – множество неориентированных ребер, помеченных временами передачи данных между взаимодействующими задачами. Граф G_T взаимодействия задач отличается от графа задач G , рассмотренного в предыдущей главе, тем, что в первом обмен данными выполняется в двух направлениях, в то время как во втором обмен данными выполняется в одном направлении.

Топологию сети процессоров представим неориентированным графом $G_L = (P, L)$, в котором $P = \{p_1, \dots, p_n\}$ – множество вершин-процессоров; L – множество ребер, представляющих каналы передачи данных. Граф G_L описывается матрицей L , строки и столбцы которой соответствуют процессорам:

$$L = \begin{pmatrix} L_{11} & \dots & L_{1p} & \dots & L_{1n} \\ & & \dots & & \\ L_{q1} & \dots & L_{qp} & \dots & L_{qn} \\ & & \dots & & \\ L_{n1} & \dots & L_{np} & \dots & L_{nn} \end{pmatrix}.$$

Элемент L_{qp} равен 1, если процессоры q и p соединены каналом передачи данных, и равен 0, если процессоры каналом не соединены. Задача t_i из множества V_T может быть решена на любом из n процессоров. Матрица A размерностью $m \times n$ определяет время решения каждой из m задач на каждом из n процессоров:

$$A = \begin{pmatrix} A_{11} & \dots & A_{1p} & \dots & A_{1n} \\ & & \dots & & \\ A_{i1} & \dots & A_{ip} & \dots & A_{in} \\ & & \dots & & \\ A_{m1} & \dots & A_{mp} & \dots & A_{mn} \end{pmatrix}.$$

Элемент A_{ip} матрицы есть время решения задачи t_i на процессоре p . С задачами t_i и t_j , решаемыми на различных процессорах, ассоцииру-

ются дополнительные временные затраты на обмен данными, если задачи взаимно информационно зависимы. Матрица C описывает временные затраты на обмен данными между парами задач:

$$C = \begin{pmatrix} C_{11} & \dots & C_{1j} & \dots & C_{1m} \\ & & \dots & & \\ C_{i1} & \dots & C_{ij} & \dots & C_{im} \\ & & \dots & & \\ C_{m1} & \dots & C_{mj} & \dots & C_{mm} \end{pmatrix}.$$

Элемент C_{ij} матрицы есть время передачи данных в период взаимодействия двух задач t_i и t_j , назначенных и решаемых на различных процессорах. Если задачи назначаются на один процессор, время C_{ij} зануляется.

Загрузка одного процессора определяется суммой времен решения всех задач, назначенных на данный процессор, и времен обмена данными с другими задачами, назначенными на другие процессоры. Время, затрачиваемое наиболее загруженным процессором, рассматривается как время параллельного распределенного решения всех задач. Оптимальное назначение m задач на n процессоров ставит целью минимизировать время работы наиболее загруженного процессора.

Пример 4.1. Пусть дано множество из пяти задач $V_T = \{t_1, t_2, t_3, t_4, t_5\}$, связи между которыми описываются графом взаимодействия задач, показанным на рис. 4.1.

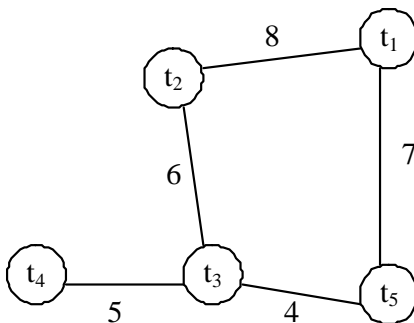


Рис. 4.1. Граф взаимодействия задач

Ребра графа помечены числами, интерпретируемыми как длительности интервалов времени, на которых задачи попарно обмениваются данными. Ребра не являются ориентированными, значит, данные могут передаваться в обоих направлениях. Следует отметить, что время передачи данных, указанное на каждом ребре, характеризует сам граф, оно не характеризует канал передачи данных и его параметры. Коммуникации между задачами представляются матрицей

$$C = \begin{array}{c} t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \\ \begin{array}{c} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{array} \left| \begin{array}{ccccc} 0 & 8 & - & - & 7 \\ 8 & 0 & 6 & - & - \\ - & 6 & 0 & 5 & 4 \\ - & - & 5 & 0 & - \\ 7 & - & 4 & - & 0 \end{array} \right. \end{array}.$$

Символ «—» означает отсутствие коммуникаций между парами задач. Нули на главной диагонали матрицы означают нулевые затраты времени на обмен данными внутри задачи.

Пусть распределенная система строится на множестве $P = \{p_1, p_2, p_3\}$ из трех процессоров. Сеть процессоров представлена графом, изображенным на рис. 4.2. В ней каждая из трех пар процессоров (p_1, p_2) , (p_1, p_3) , (p_2, p_3) соединена каналом передачи данных. Все каналы имеют одинаковую пропускную способность, следовательно, время передачи данных не зависит от канала, по которому данные передаются.

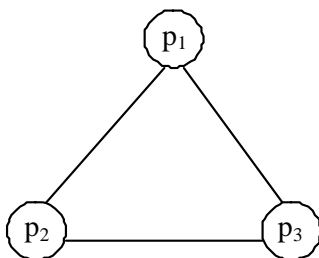


Рис. 4.2. Топология сети

Сеть процессоров можно также описать единичной матрицей

$$L = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \end{array} \begin{array}{ccc} p_1 & p_2 & p_3 \\ \left| \begin{array}{ccc} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{array} \right| \end{array} .$$

Поскольку распределенная система является неоднородной, время решения каждой задачи меняется при переходе от одного процессора к другому. Времена решения задач описываются матрицей A размерности 5×3 :

$$A = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{array} \begin{array}{ccc} p_1 & p_2 & p_3 \\ \left| \begin{array}{ccc} 15 & 11 & 9 \\ 14 & 12 & 8 \\ 16 & 13 & 6 \\ 5 & 4 & 3 \\ 10 & 9 & 7 \end{array} \right| \end{array} .$$

В среднем, время решения задачи на процессоре p_3 меньше, чем на процессоре p_1 .

4.2. Назначение задач на узлы

Назначение задач на процессоры определяется матрицей X размерности $m \times n$:

$$X = \begin{array}{c} \\ \\ \\ \\ X_{m1} \end{array} \begin{array}{ccccc} X_{11} & \dots & X_{1p} & \dots & X_{1n} \\ & & \dots & & \\ X_{i1} & \dots & X_{ip} & \dots & X_{in} \\ & & \dots & & \\ X_{m1} & \dots & X_{mp} & \dots & X_{mn} \end{array} .$$

Элемент X_{ip} равен 1, если задача i назначается на процессор p , и равен 0 в противном случае. С учетом того, что каждая задача назначается только на один процессор, каждая строка матрицы содержит ровно одну единицу. Столбец матрицы может содержать произвольное число единиц, но не превышающее число задач. Множество всех значений матрицы X представляет пространство поиска, в котором ищется оптимальное решение.

4.3. Оценка общего времени решения задач

Общее время решения всех задач определяется временем функционирования наиболее загруженного процессора. Загрузка процессора $p \in P$ определяется следующим выражением

$$load_X(p) = \sum_{i=1}^m (A_{ip} \times X_{ip}) + \sum_{\substack{q=1, \\ q \neq p}}^n \sum_{i=1}^m \sum_{j=1}^m (C_{ij} X_{ip} X_{jq} L_{pq}). \quad (4.1)$$

Первая часть суммы (4.1) есть полное время решения всех задач, назначенных на процессор p . Задача i назначена на процессор p , если $X_{ip} = 1$, при этом время решения задачи определяется значением A_{ip} . Вторая часть есть дополнительное время, затрачиваемое процессором p на обмен данными с другими процессорами. Единичные значения элементов X_{ip} и X_{jq} указывают на то, что задача i назначена на процессор p , а задача j назначена на процессор q . Единичное значение элемента L_{pq} указывает на то, что процессоры p и q соединены каналом передачи данных. Для того чтобы найти процессор с максимальной загрузкой, определяющий общее время $Time_X$ решения всех задач, необходимо рассчитать загрузку каждого из n процессоров для определенной матрицы X

$$Time_X = \max_{p \in P} (load_X(p)). \quad (4.2)$$

Оптимальное назначение задач на процессоры есть то, которое дает минимум загрузки $MinTime$ по всем возможным значениям матрицы X наиболее нагруженного процессора $p \in P$:

$$\min Time = \min_{X \in \Omega} (Time_X). \quad (4.3)$$

Задача (4.3) относится к классу дискретных задач комбинаторной оптимизации.

4.4. Алгоритм оптимального назначения задач на процессоры

Алгоритм A^* [5] реализует стратегию поиска первого наилучшего при решении широкого круга проблем, включая проблемы искусственного интеллекта. Для построения алгоритма используются древовидные графы, называемые деревьями поиска. При решении проблемы назначения задач на процессоры множество вершин дерева поиска разбивается на ярусы. Корень дерева, находящийся на нулевом ярусе, соответствует нулевому назначению задач на процессоры. Листья дерева поиска (терминальные вершины), находящиеся на m -м ярусе, соответствуют полным назначениям задач на процессоры. Нетерминальные вершины соответствуют частичным назначениям задач на процессоры. Каждая нетерминальная вершина d_i , находящаяся на ярусе i дерева поиска, является родительской для n дочерних вершин, получаемых в результате назначения задачи t_i на процессоры $1, \dots, n$ соответственно. Как следствие, вершина d_i имеет n исходящих дуг.

С каждой нетерминальной вершиной d_i дерева поиска ассоциируется функция стоимости $\minTime(d_i)$, описывающая оценку минимального времени решения всех задач при их частичном назначении на процессоры. Если вершина d_i является терминальной, то $i = m$ и функция $\minTime(d_m)$ описывает точное значение $Time_x$ времени решения всех задач при их полном назначении на процессоры, вычисляемое по формуле (4.2). При этом значение матрицы X определяется путем от корня дерева поиска до терминальной вершины d_m .

Для нетерминальной вершины d_i оценка $\minTime(d_i)$ минимального времени решения всех задач записывается в виде суммы:

$$\minTime(d_i) = g(d_i) + b(d_i), \quad (4.4)$$

где $g(d_i)$ – время, ассоциируемое с путем от корня дерева до вершины d_i и вычисляемое по формулам (4.1)–(4.2) с учетом только задач $1, \dots, i - 1$, которые уже назначены на процессоры;

$b(d_i)$ – нижняя граница времени, ассоциируемого с путем от вершины d_i до одного из листьев дерева, оцениваемая для еще не назначенных на процессоры задач с номерами i, \dots, m .

Если слагаемое $g(d_i)$ точно рассчитывается по уже пройденному на дереве поиску пути, то слагаемое $b(d_i)$ требует разработки метода оценки нижней границы времени для еще не пройденного пути. Более точная оценка нижней границы сокращает комбинаторный перебор вариантов на дереве поиска.

Для расчета нижней границы времени используется следующий метод. С целью оценки загрузки процессора p рассматривается два подмножества задач. Подмножество T_p включает задачи, уже назначенные на процессор p . Подмножество U_p включает задачи, еще не назначенные на процессоры и имеющие одну или более связей с задачами из множества T_p . Для каждой задачи $t_j \in U_p$ существует два альтернативных варианта: либо она будет назначена на процессор p , и тогда время обмена данными с задачами из T_p зануляется; либо она назначается на другой процессор q , соединенный с процессором p каналом передачи данных, и тогда время передачи не зануляется. Как следствие, с каждым назначением задачи t_j ассоциируются два вида стоимости: либо время A_{jp} решения t_j на p , либо сумма H_{pj} времен обмена данными всех задач из T_p с задачей t_j . С использованием минимального значения стоимостей двух видов

$$\text{cost}(t_j) = \min(A_{jp}, H_{pj}) \quad (4.5)$$

принимается решение о назначении или не назначении t_j на p . Тогда нижняя граница $b(d_i)$ стоимости назначения оставшихся вершин на процессоры оценивается выражением:

$$b(d_i) = \sum_{t_j \in U_p} \text{cost}(t_j). \quad (4.6)$$

Каждой вершине дерева поиска поставим в соответствие две метки. Первой является строка символов $s = s_1 \dots s_i \dots s_n$, где s_i есть цифра, описывающая номер процессора, на который назначена задача t_i . Если задача еще не назначена на процессор, значением s_i является символ «X». Второй меткой является значение времени \minTime , которое оценивается уравнениями (4.1), (4.4)–(4.6).

Каждый уровень дерева поиска соответствует одной задаче. Нумерация уровней соответствует нумерации задач: i -й задаче соответствует i -й уровень дерева поиска. Корень дерева находится на нулевом уровне, ему соответствуют метки «X...X» и 0. Все полные назначения задач находятся на уровне m . Вершины этого уровня метятся строками без символа «X» и значениями времени, не равными 0. Частичные назначения описываются вершинами, находящимися на уровнях от 1 до $m - 1$. Им соответствуют строки, состоящие как из цифр, так и символа «X». Переход с уровня i на уровень $i + 1$ связан с заменой символа «X» в позиции $i + 1$ строки вершины на один из номеров процессоров, взятых из диапазона 1, ..., n . При этом вершина раскрывается, в результате генерируются n дочерних вершин. Для каждой из дочерних вершин оценивается значение времени \minTime .

Алгоритм A^* использует упорядоченный список не раскрытых вершин дерева поиска, называемый OPEN. Включаемые в список новые вершины упорядочиваются по не убыванию (возрастанию) значения функции стоимости \minTime . Обход дерева поиска начинается с корня, следовательно, вершина с метками «X...X» и 0 помещается в список OPEN первой. В процессе поиска список OPEN изменяется динамически. Для этого из списка выбирается первая вершина с наименьшей стоимостью, к ней применяется оператор раскрытия, в результате генерируются дочерние вершины. Первая вершина удаляется из списка, а новые вершины добавляются в него, при этом список перепорядочивается. Такой порядок раскрытия вершин гарантирует нахождение оптимального решения в момент появления в списке вершины со строковой меткой, в которой отсутствует символ «X».

Пример 4.2. Применим алгоритм A^* к графу взаимодействия задач и графу топологии сети, которые специфицированы в примере 4.1. Поскольку первый граф состоит из пяти задач, вершины дерева поиска метятся строками из пяти символов. Так как граф топологии сети построен на трех процессорах, нетерминальные вершины дерева поиска имеют каждая три исходящие дуги. Соответственно в строках, метящих вершины, могут появиться только цифры 1, 2, 3.

Продемонстрируем процесс динамического построения дерева поиска алгоритмом A^* . Результатом этого процесса является граф, показанный на рис. 4.3. Его вершины представлены прямоугольниками, в которых строка символов показывает состояние назначения

задач на процессоры, число описывает значение функции стоимости $\min Time$. Число, стоящее рядом с прямоугольником, есть порядковый номер раскрытия вершины. Отсутствие такого числа указывает на то, что вершина осталась не раскрытой. Одна терминальная вершина помечена как «цель». Эта вершина представляет оптимальное решение задачи. Более подробно процесс поиска решения описывается следующими шагами.

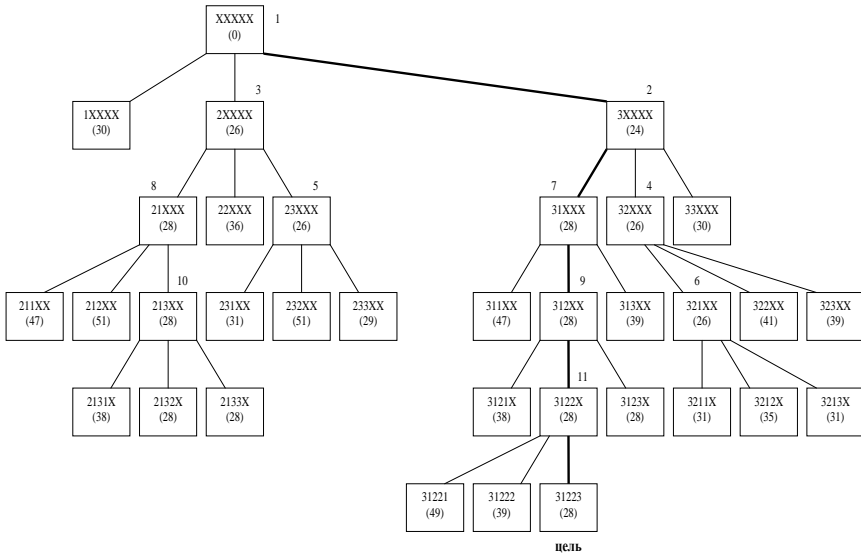


Рис. 4.3. Результирующее дерево поиска

Шаг 0. OPEN = {«XXXXX»-0}.

Шаг 1. Выбираем вершину «XXXXX». Заменяем «X» на 1 в первой позиции. Используем формулу (4.1) для расчета загрузки процессоров. Значение g рассчитываем по формуле (4.4). Процессоры p_2, p_3 вообще не загружены, поэтому $load(p_2) = load(p_3) = 0$. На процессор p_1 назначена только задача t_1 . Отсюда, $g = load(p_1) = 15$. Для вычисления b строим множество $U = \{t_2, t_5\}$. Рассчитываем $cost(t_2) = \min(14, 8) = 8$ и $cost(t_5) = \min(10, 7) = 7$. Величина $b = cost(t_2) + cost(t_5) = 15$. В результате $\min Time = 30$. Аналогичным образом рассчитываются значения $\min Time$ для вершин «2XXXX» и «3XXXX», они равны 26 и 24 соответственно. Исключая из списка OPEN вер-

шину «XXXXX», включая вместо нее три новые вершины «1XXXX», «2XXXX», «3XXXX», упорядочивая их по возрастанию значения $\min Time$, получаем новое состояние списка OPEN = {«3XXXX» – 24, «2XXXX» – 26, «1XXXX» – 30}.

Шаг 2. Выбираем вершину «3XXXX». Раскрываем ее в три новые вершины «31XXX», «32XXX», «33XXX» со значениями функции $\min Time$, равными 28, 26, 30 соответственно. Новое состояние списка OPEN = {«2XXXX» – 26, «32XXX» – 26, «31XXX» – 28, «1XXXX» – 30, «33XXX» – 30}.

Шаг 3. Выбираем вершину «2XXXX» и раскрываем ее в три новые вершины «21XXX», «22XXX», «23XXX» с весами 28, 36, 26. Список OPEN = {«32XXX» – 26, «23XXX» – 26, «31XXX» – 28, «21XXX» – 28, «1XXXX» – 30}.

Шаг 4. Выбираем вершину «32XXX» и раскрываем ее в три новые вершины «321XX», «322XX», «323XX» с весами 26, 41, 39. Список OPEN = {«23XXX» – 26, «321XX» – 26, «31XXX» – 28, «21XXX» – 28, «1XXXX» – 30}.

Шаг 5. Выбираем вершину «23XXX» и раскрываем ее в три новые вершины «231XX», «232XX», «233XX» с весами 31, 51, 29. Список OPEN = {«321XX» – 26, «31XXX» – 28, «21XXX» – 28, «233XX» – 29, «1XXXX» – 30}.

Шаг 6. Выбираем вершину «321XX» и раскрываем ее в три новые вершины «3211X», «3212X», «3213X» с весами 31, 35, 31. Список OPEN = {«31XXX» – 28, «21XXX» – 28, «233XX» – 29, «1XXXX» – 30, «33XXX» – 30}.

Шаг 7. Выбираем вершину «31XXX» и раскрываем ее в три новые вершины «311XX», «312XX», «313XX» с весами 47, 28, 39. Список OPEN = {«21XXX» – 28, «312XX» – 28, «233XX» – 29, «1XXXX» – 30, «33XXX» – 30}.

Шаг 8. Выбираем вершину «21XXX» и раскрываем ее в три новые вершины «211XX», «212XX», «213XX» с весами 47, 51, 28. Список OPEN = {«312XX» – 28, «213XX» – 28, «233XX» – 29, «1XXXX» – 30, «33XXX» – 30}.

Шаг 9. Выбираем вершину «312XX» и раскрываем ее в три новые вершины «3121X», «3122X», «3123X» с весами 38, 28, 28. Список OPEN = {«213XX» – 28, «3122X» – 28, «3123X» – 28, «233XX» – 29, «1XXXX» – 30}.

Шаг 10. Выбираем вершину «213XX» и раскрываем ее в три новые вершины «2131X», «2132X», «2133X» с весами 38, 28, 28. Список OPEN = {«3122X» – 28, «3123X» – 28, «2132X» – 28, «2133X» – 28, «233XX» – 29}.

Шаг 11. Выбираем вершину «3122X». Раскрываем ее в три новые вершины «31221», «31222», «31223» со значениями 49, 39, 28 функции стоимости. Поскольку вершина «31223» описывает полное назначение задач на процессоры и имеет значение 28 стоимости (все другие вершины не имеют меньшего значения, а их раскрытие может только увеличить значение стоимости), эта вершина является целевой оптимальной.

Таким образом, оптимальное назначение задач на процессоры определяется матрицей A :

$$A = \begin{array}{c} \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \\ \\ \\ \\ \\ \end{array} \begin{array}{ccc} p_1 & p_2 & p_3 \\ \left| \begin{array}{ccc} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right. \end{array} .$$

Для поиска оптимального назначения потребовалось 11 раскрытий вершин, при этом были сгенерированы и включены в список OPEN 33 вершины (три вершины на одно раскрытие).

5. ЯЗЫКИ И ИНСТРУМЕНТЫ ПРОГРАММИРОВАНИЯ РАСПРЕДЕЛЕННОЙ И ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ ДАННЫХ

Для распределенного параллельного программирования существует ряд инструментов, включающих:

- *Multithreading* – модель многопоточных приложений;
- MPI – интерфейс передачи сообщений;
- OpenMP – открытый стандарт для распараллеливания программ;
- CORBA – технологический стандарт для написания распределенных приложений;
- COM – модель компонентных объектов и др.

5.1. Многопоточные приложения

Операционная система (ОС) является *многозадачной*, если она способна одновременно выполнять несколько программ. Одна операционная система способна реализовать многозадачность в двух вариантах: путем разделения между задачами процессорного времени одного процессора; путем назначения задач на разные процессоры. Уже в первом варианте многозадачности, ОС настолько быстро переключает вычислительные ресурсы между задачами, что создается впечатление их одновременного выполнения.

Многозадачность может быть *кооперативной* и *вытесняющей*. В случае кооперативной многозадачности ОС не занимается решением проблемы распределения процессорного времени между задачами. Распределяют его сами задачи. Активная задача самостоятельно решает, отдавать ли процессор другой задаче или не отдавать.

В случае вытесняющей многозадачности распределением процессорного времени занимается ОС. Она выделяет каждой задаче фиксированный квант процессорного времени. По истечении этого кванта система вновь получает управление, чтобы выбрать другую задачу для активизации. Если задача обращается к ОС до истечения кванта, это также служит причиной переключения задач. В операционных системах корпорации Microsoft, начиная с Windows 95, реализована вытесняющая многозадачность.

Процессом называется экземпляр программного приложения, загруженного в оперативную память для выполнения. Многопоточность ОС [8] означает то, что процессы могут разделяться на части – потоки, самостоятельно претендующие на процессорное время. Использование потоков обеспечивает одновременное выполнение нескольких ветвей программы. При выполнении процесс создает не менее одного потока (*thread*). Первый поток называется *главным* потоком приложения, создаваемым ОС автоматически. Он порождает другие потоки, те в свою очередь третьи и т. д.

Потоки выполняются на процессорах или ядрах процессоров. Так как обычно потоков больше, чем процессоров, часть потоков выполняются не параллельно, а последовательно по очереди. Система выделяет потокам кванты времени по принципу карусели (рис. 5.1).

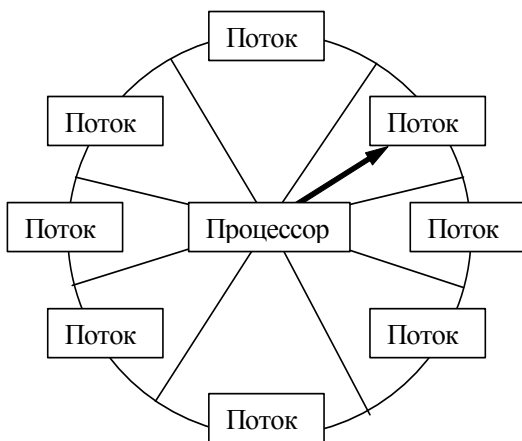


Рис. 5.1. Карусельная модель выполнения потоков

В зависимости от ситуации потоки могут находиться в трех состояниях: *активном*, *готовности/ожидания*, *блокировки*. Перевод потока из состояния блокировки в состояние готовности выполняется по событиям.

Первичный поток приложения вызывает входную функцию *main* или ее аналоги. В библиотеке классов MFC основной поток приложения создается с помощью класса *CWinApp*, производного от класса *CWinThread*. Пространство кода и данных процесса доступно

всем его потокам. Разные потоки имеют доступ и могут обращаться к общим глобальным переменным процесса. Каждый поток имеет свой собственный стек, поддерживающий его выполнение. Windows поддерживает два вида потоков: *рабочие* потоки; потоки *пользовательского интерфейса*. Поток пользовательского интерфейса может иметь окна и свой цикл выборки сообщений.

В Win32 API поток создается функцией *CreateThread*. Поток завершает выполнение при выходе из функции потока посредством оператора *return*. Функция *WaitForSingleObject* ждет, пока конкретный поток закончит свою работу. Выполнение потока можно приостановить вызовом функции *SuspendThread*. Поток возобновляется вызовом функции *ResumeThread*.

Приоритеты потоков. Каждому потоку присваивается приоритет от 0 до 31. Windows поддерживает следующие семь классов приоритета потоков: CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE. Для установки и получения приоритета служат функции *SetThreadPriority* и *GetThreadPriority*.

Потоки взаимодействуют друг с другом в двух основных случаях: при совместном использовании разделяемого ресурса с целью избегания его разрушения; при уведомлении других потоков в момент завершения каких-либо операций. Для синхронизации потоков программисту предоставляются следующие средства.

Критическая секция синхронизирует потоки посредством обеспечения монопольного захвата ресурса и атомарного доступа одним потоком в случае, когда доступ к общим ресурсам (файлу, глобальной переменной и др.) выполняется разными потоками одновременно. Критическая секция представляет собой участок кода, позволяющий сделать так, чтобы одновременно только один поток получал доступ к ресурсу.

Семафор есть объект, ограничивающий количество потоков, которые могут войти в заданный участок кода. Семафоры используются при передаче данных через разделяемую память. Они реализуются посредством счетчика, значение которого уменьшается, когда семафор выделяется задаче, и увеличивается, когда семафор освобождается задачей. Семафор позволяет параллельно работающим потокам обращаться к общему ресурсу избегая конфликтов.

Взаимно исключающий семафор называется *мьютексом*. Являясь разновидностью семафора, использующего счетчик числа поль-

зователей, мьютекс гарантируют потокам взаимоисключающий безконфликтный доступ к общему ресурсу. Идентификатор потока определяет, какой поток захватил мьютекс, а счетчик — сколько раз. Как правило, мьютексы защищают блок памяти, к которому обращается множество потоков.

5.2. Интерфейс MPI

MPI расшифровывается как Message Passing Interface – интерфейс передачи сообщений [9]. MPI является международным стандартом интерфейса обмена данными в параллельном программировании. Он реализован на большом числе компьютерных платформ, а также используется при параллельном программировании для кластеров и суперкомпьютеров. Существуют реализации MPI для языков Фортран 77/90, Java, C и C++. В первую очередь MPI ориентирован на системы с распределенной памятью, в которых затраты на передачу данных велики. Одной из реализаций MPI является пакет MPICH.

MPI поддерживает три основных режима передачи данных:

1. Синхронный режим, когда посылающий процесс ждет начала приема сообщения принимающим процессом; при этом не требуются промежуточные буферы для передаваемых данных; обеспечивается надежная передача данных большого размера;

2. Асинхронный режим, когда посылающий процесс не ждет начала приема сообщения принимающим процессом, используя для этого промежуточные буферы; обеспечивается эффективная передача коротких сообщений с меньшей надежностью;

3. Широковещательный режим, когда процесс посылает сообщения всем другим процессам.

MPI поддерживает операции процесс-процесс и коллективные операции, такие как разборка-сборка и редукция сообщений. Он обеспечивает создание приложений в модели «одна программа – множественные данные», позволяющей исполнять один программный код в разных процессах, каждый со своими данными. При этом программному коду доступна информация о том, в каком именно процессе он выполняется. Это позволяет адаптировать исполнение программы к конкретному процессу и учесть особенности этого процесса,

что позволяет рассматривать MPI как реализацию модели «множественная программа – множественные данные».

В MPI все операции передачи сообщений и синхронизации локализуются внутри *коммуникатора*. С коммуникатором связывается группа процессов, а все процессы группы взаимодействуют через коммуникатор. В частности, все операции процесс-процесс и коллективные операции выполняются процессами, входящими в группу и взаимодействующими через коммуникатор. Процессы внутри группы нумеруются целыми числами в диапазоне 0...groupsize-1. Коммуникаторы идентифицируются именами. Стандартное имя коммуникатора – MPI_COMM_WORLD.

MPI-программа представляет собой набор независимых параллельно выполняемых процессов, каждый из которых выполняет общий программный код, специфицированный к конкретному процессу. Если параллельное приложение строится на параллелизме данных, то программный код процессов имеет большое общее ядро. Если параллельное приложение строится на параллелизме ветвей программы, то программный код одного процесса может сильно отличаться от программного кода другого процесса. При параллельном выполнении процессы взаимодействуют друг с другом посредством вызова коммуникационных процедур. Каждый процесс выполняется в своем собственном адресном пространстве, однако допускается и режим разделения памяти между процессами. MPI не специфицирует модель выполнения одного процесса – это может быть как последовательный, так и многопоточный процесс.

MPI реализуется библиотекой функций. Все множество функций (их около 130) разбито на классы: операции точка-точка, операции коллективного обмена, топологические операции, системные и вспомогательные операции. Простейшая параллельная программа может быть написана с использованием всего 6 MPI-функций, а достаточно полную и удобную среду программирования составляет набор из 24 функций. Определения всех именованных констант, типов данных и прототипов функций содержатся в подключаемом файле *mpi.h*.

Пример MPI программы решения системы линейных алгебраических уравнений (СЛАУ) блочно-параллельным методом Гаусса дан на рис. 5.2.

```

#define M 400
#define N 50
#define EL(x) (sizeof(x) / sizeof(x[0]))
double MA[N][M+1], V[M+1], MAD, R;
int main(int args, char **argv) {
    int size, MyP, i, j, v, k, d, p;
    MPI_Comm comm_gr;
    MPI_Status status;
    MPI_Init(&args, &argv); // Инициализация библиотеки
    MPI_Comm_size(MPI_COMM_WORLD, &size); // Размер системы
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP); // Свой номер
    for(i = 0; i < N; i++) { // Инициализация СЛАУ
        for(j = 0; j < M; j++) {
            if((N*MyP+i) == j) MA[i][j] = 2.0; else MA[i][j] = 1.0;
        } MA[i][M] = 1.0*(M)+1.0;
    }
    for(p = 0; p < size; p++) { // Прямой ход метода Гаусса
        for(k = 0; k < N; k++) {
            if(MyP == p) { // Работа активного передающего процесса
                MAD = 1.0/MA[k][N*p+k];
                for(j = M; j >= N*p+k; j--) MA[k][j] = MA[k][j] * MAD;
                for(d = p+1; d < size; d++)
                    MPI_Send(&MA[k][0], M+1, MPI_DOUBLE, d, 1, comm_gr);
                for(i = k+1; i < N; i++) {
                    for(j = M; j >= N*p+k; j--) MA[i][j] = MA[i][j]-MA[i][N*p+k]*MA[k][j];
                }
            } else if(MyP > p) { // Работа принимающих процессов
                MPI_Recv(V, EL(V), MPI_DOUBLE, p, 1, comm_gr, &status);
                for(i = 0; i < N; i++)
                    { for(j = M; j >= N*p+k; j--) MA[i][j] = MA[i][j]-MA[i][N*p+k]*V[j];
                    } }
        }
    }
    for(p = size-1; p >= 0; p--) { // Обратный ход метода Гаусса
        for(k = N-1; k >= 0; k--) {
            if(MyP == p) { // Работа активного передающего процесса
                for(d = p-1; d >= 0; d--)
                    MPI_Send(&MA[k][M], 1, MPI_DOUBLE, d, 1, comm_gr);
                for(i = k-1; i >= 0; i--) MA[i][M] -= MA[k][M]*MA[i][N*p+k];
            } else if(MyP < p) { // Работа принимающих процессов
                MPI_Recv(&R, 1, MPI_DOUBLE, p, 1, comm_gr, &status);
                for(i = N-1; i >= 0; i--) MA[i][M] -= R*MA[i][N*p+k];
            }
        }
    }
    MPI_Finalize(); return(0);
}

```

Рис. 5.2. MPI программа решения СЛАУ методом Гаусса

Программа представлена функцией *main* языка C. Макро-переменная *M* со значением 400 определяет число переменных в СЛАУ. Макро-переменная *N* со значением 50 определяет число блоков, на которые разбивается СЛАУ. Тип данных *MPI_Comm* описывает структуру коммуникатора. Граф топологии коммуникатора определяется отдельным кодом, который здесь не приводится. Функция *MPI_Init* инициализирует библиотеку MPI. Функция *MPI_Comm_size* возвращает число *size* процессов в коммуникаторе *MPI_COMM_WORLD*. Функция *MPI_Comm_rank* возвращает номер *MyP* вызывающего ее процесса. Следующие два вложенных цикла, инициализирующих СЛАУ, выполняются в каждом процессе. Прямой ход метода Гаусса реализуется двумя циклами *for*: первый проходит по процессам *p*, второй – по блокам *k* СЛАУ. Если номер процесса *p* равен номеру *MyP* исполняемого процесса, то этот процесс становится активным, начинает пересчет строк соответствующего блока матрицы СЛАУ с посылкой результатов другим процессам, пересчитывающим строки других блоков матрицы. Для посылки результатов используется функция *MPI_Send*, для приема – функция *MPI_Recv*. Обратный ход метода Гаусса также реализуется двумя циклами *for*, однако они проходят процессы и блоки в обратном порядке.

5.3. Открытый стандарт OpenMP

OpenMP реализует параллельные вычисления на машинах с несколькими процессорами с помощью многопоточности, которая строится автоматически из последовательного кода по директивам препроцессора, называемым *pragma*. Количество создаваемых потоков, регулируемое посредством директив, может превышать количество доступных процессоров. OpenMP ориентирован на системы с общей памятью, к которым относятся многоядерные системы с общим кэшем. В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды. Ключевыми элементами OpenMP являются: конструкции для создания потоков (директива *parallel*); конструкции распределения работы между потоками (директивы *do/for* и *section*); конструкции для управления работой с данными (выражения *shared* и *private* для определения класса памяти переменных); конструкции для синхронизации потоков (директивы *critical*, *atomic* и *barrier*) и др.

OpenMP можно рассматривать как высокоуровневую надстройку над многопоточностью. Программная модель OpenMP представляет собой *fork-join* параллелизм, в котором главный поток по необходимости порождает вспомогательные потоки при вхождении программы в параллельные области. OpenMP позволяет быстро распараллелить программы с циклами, выполняющими большой объем вычислений. Одна OpenMP-программа выполняется параллельно на многопроцессорной систем и выполняется последовательно на однопроцессорной системе. Для языка программирования C типы данных и функции OpenMP определены в подключаемом файле *omp.h*.

Пример программы на языке C, использующей директивы OpenMP [10], приведен на рис. 5.3. Внешняя функция *average* с тремя входными аргументами возвращает их среднее значение. Функция *master_example* имеет три аргумента: массив *x*; массив *xold*; число *n* элементов в массивах *x* и *xold*; пороговое значение *tol*.

За объявлениями локальных переменных *c*, *i*, *toobig*, *error*, у следует директива *#pragma omp parallel*. Эта важнейшая директива *parallel* указывает на необходимость автоматического распараллеливания нижеследующего блока, заключенного в *{ }*. Блок состоит из одного оператора цикла *do*, завершающего вычисления в случае, когда ни одно среднее значение трех соседних элементов массива *x* не превышает порогового значения *tol*.

Следующая директива *#pragma omp for private(i)* относится к последующему циклу *for*. Благодаря присутствию *private(i)*, для каждого вспомогательного потока, создаваемого с целью параллельной реализации различных итераций цикла, вводится своя копия переменной *i*. Директива *#pragma omp single* указывает на то, что только один поток (не обязательно ведущий) выполняет нижеследующий блок, а именно, присваивание переменной *toobig* значения 0.

Директива *#pragma omp for private(i,y,error) reduction(+:toobig)* относится к последующему циклу *for*. Вспомогательные потоки, реализующие различные итерации цикла, имеют свои копии переменных *i*, *y*, *error*. Слово *reduction* указывает в момент завершения цикла на суммирование (+) значений всех копий переменной *toobig*, вычисленных различными вспомогательными потоками. Директива *#pragma omp master* специфицирует нижеследующий блок, исполняемый только ведущим потоком.

```

#include <omp.h>
#include <stdio.h>
extern float average(float, float, float);
void master_example( float* x, float* xold, int n, float tol ) {
    int c, i, toobig;
    float error, y;
    c = 0;
#pragma omp parallel
    {
        do{
#pragma omp for private(i)
            for( i = 1; i < n-1; ++i ) {
                xold[i] = x[i];
            }
#pragma omp single
            {
                toobig = 0;
            }
#pragma omp for private(i,y,error) reduction(+:toobig)
            for( i = 1; i < n-1; ++i ) {
                y = x[i];
                x[i] = average( xold[i-1], x[i], xold[i+1] );
                error = y - x[i];
                if( error > tol || error < -tol ) ++toobig;
            }
#pragma omp master
            {
                ++c;
                printf( "iteration %d, toobig=%d\n", c, toobig );
            }
        } while(toobig > 0);
    }
}

```

Рис. 5.3. Программа на языке C с директивами OpenMP

5.4. Технологический стандарт CORBA

CORBA – одна из ведущих технологий создания распределённых приложений [2]. CORBA – аббревиатура от *Common Object Request Broker Architecture* (общая архитектура брокера объектных запросов). Разработана эта технология с целью обеспечения объектно-ориентированной коммуникации между частями одного распределённого приложения. CORBA – технология кросс-платформенная.

Суть CORBA состоит в следующем: каждый компонент распределённого приложения имеет доступ к открытым методам других компонентов, которые он может вызывать на выполнение. О том, какие есть методы, он узнаёт с помощью интерфейсов, описание которых осуществляется на специальном языке IDL – *Interface Definition Language*.

Брокер объектных запросов (*Object Request Broker*) архитектуры CORBA – это один из самых важных компонентов распределенной системы, отвечающий за то, чтобы запросы от одних объектов пришли к другим, причем именно к тем, которым они были посланы. Запрашивающий компонент называется клиентом, запрашиваемый компонент называется сервером. В рамках другого запроса клиент и сервер могут поменяться местами. Таким образом, каждый компонент в распределенном CORBA-приложении имеет одновременно свойства и клиента, и сервера – и, таким образом, достигает максимальной самостоятельности и независимости в своей реализации от других компонентов приложения. Брокеры могут быть самостоятельными приложениями или могут встраиваться в другие приложения.

Еще одно важное понятие CORBA – *Interface Repository*. Это хранилище всех зарегистрированных в системе интерфейсов, их методов и параметров этих методов. Другое хранилище *Implementation Repository* содержит информацию о доступных серверах.

5.5. Модель компонентных объектов COM

После компиляции приложение состоит из одного монолитного двоичного файла, который в соответствии с традиционными технологиями остается неизменным пока не будет скомпилирована новая версия. Модель компонентных объектов Microsoft COM (*Component Object Model*) позволяет разбить монолитное приложение на отдельные части, называемые *компонентами* [11–13]. В процессе работы приложения одни версии компонентов могут заменяться другими версиями. COM является стандартной спецификацией общего метода создания компонентов и построения из них приложений. Преимущества компонентной модели:

1. Способность приложения эволюционировать с течением времени путем замены устаревших версий компонентов более современными версиями;

2. Адаптация приложения к различным пользователям путем использования компонентов, наиболее адекватных потребностям пользователя;

3. Возможность быстрой сборки приложения из компонентов библиотеки;

4. Повышение эффективности разработки распределенных клиент-серверных приложений.

Общие требования к компонентам:

1. Подключение компонентов во время выполнения приложения требует применения динамической компоновки;

2. Применение принципа инкапсуляции к компонентам; компоненты должны разбиваться на две основные части: интерфейс с внешним миром и внутреннюю реализацию;

3. Обладание способностью реализации внутри одного процесса, в разных процессах и на разных машинах, должно обеспечиваться перемещение компонентов в компьютерной сети;

4. Поддерживание клиент-серверной архитектуры приложений, в которой сервер реализуется компонентом, а клиент общается с сервером посредством соответствующих интерфейсов;

5. Разработка отдельных компонентов и целых многокомпонентных приложений должна обеспечиваться на разных языках программирования;

6. Поддерживание библиотечного сервиса управления компонентами.

5.5.1. Реализация компонентов

Основные принципы реализации *компонентов* COM опишем с использованием языка C++ объектно-ориентированного программирования. Структура компонента, построенная с использованием *поллиморфизма*, представлена на рис. 5.4. Класс объектов *Component* наследует в открытом режиме *чисто абстрактные базовые классы Interface1 ... InterfaceN*, описывающие *интерфейсы*, реализуемые объектами. Основное назначение класса *Component* – реализация интерфейсов, прежде всего стандартного для COM интерфейса *IUnknown*, который включает *чисто виртуальные функции QueryInterface, AddRef и Release*. Другие интерфейсы также построены на базе чисто виртуальных функций. Счетчик числа ссылок *m_cRef* используется для подсчета указателей на интерфейсы, установленных в процессе совме-

стной работы клиента и сервера, при этом клиент использует для реализации своей функциональности интерфейсы, адреса которых он получает от компонента. Нулевое значение счетчика вызывает самоликвидацию компонента.

```
class Component : public Interface1,..., public InterfaceN {
    <Реализация IUnknown:>
        <Реализация QueryInterface>
        <Реализация AddRef>
        <Реализация Release>
    <Реализация Interface1>
    ...
    <Реализация InterfaceN>
    <Реализация конструктора и деструктора>
private: long m_cRef;
};
```

Рис. 5.4. Структура компонента на языке C++

Архитектура многокомпонентного СОМ приложения определяет систему интерфейсов между компонентами таким образом, что отдельные компоненты могут модернизироваться или заменяться другими компонентами без изменения интерфейсов. Для идентификации интерфейсов и компонентов используются *глобально уникальные идентификаторы GUID*, позволяющие находить и выполнять доступ к компонентам и интерфейса через *реестр* операционной системы. Идентификаторы GUID могут быть сгенерированы на компьютере программиста программой *guidgen.exe*. Для создания и динамической загрузки компонентов используется специальный компонент, называемый *фабрикой класса*. Фабрика класса реализует специальный интерфейс *IClassFactory*, удовлетворяющий всем требованиям интерфейсов СОМ.

Клиент и сервер-компонент могут быть реализованы тремя способами. Реализация в одном процессе делает доступным одно адресное пространство и клиенту и серверу. Все интерфейсы, реализуемые компонентами, доступны клиенту посредством указателей. Реализация в разных процессах на одном компьютере делает необходимым использование технологии передачи данных, называемой *маршалингом*. Реализация в разных процессах на разных компьютерах делает необходимым использование сетевого программного обеспечения в дополнение к маршалингу.

Составные компоненты строятся из базовых компонентов посредством *включения* и *агрегирования*. Включение компонентов в СОМ является аналогом *композиции*, а агрегирование является аналогом *наследования* в объектно-ориентированном программировании. Отличие состоит в том, в компоненты взаимодействуют не напрямую, а через интерфейсы.

5.5.2. Определение интерфейсов

Интерфейс СОМ представляет собой чисто абстрактный базовый класс, определяющий набор функций-методов, реализуемых компонентами и используемых клиентами для взаимодействия с компонентами. Для клиента компонент представляет собой набор интерфейсов, реализуемых компонентами. По-существу, интерфейс есть структура данных в памяти, содержащая массив указателей на функции-методы интерфейса. Поскольку каждый компонент СОМ может поддерживать сколь угодно много интерфейсов, для реализации компонента с несколькими интерфейсами используется множественное наследование чисто абстрактных базовых классов. Одно из самых больших преимуществ компонентной модели — возможность повторного использования интерфейсной архитектуры приложения. В случае сохранения интерфейсной архитектуры изменения в компоненте не вызывают изменений в клиенте и наоборот. При изменении интерфейсной архитектуры не следует изменять существующие интерфейсы, достаточно добавить новые интерфейсы.

На рис. 5.4 представлена структура интерфейса как абстрактного базового класса (ключевое слово *interface* есть переопределение слова *struct*), состоящего из чисто виртуальных функций *Method1...Methodr*. Согласно требованиям СОМ, этот класс наследует в закрытом режиме стандартный интерфейс *IUnknown*, подключаемый из заголовочного файла *unknwn.h*. Объявление интерфейса *IUnknown* представлено на рис. 5.5.

```
interface Interface1: IUnknown {
    virtual type1 __stdcall Method1(type1l,..., type1kl) = 0;
    ...
    virtual typep __stdcall Methodr(typep1,..., typepkr) = 0;
};
```

Рис. 5.4. Объявление интерфейса

```

interface IUnknown {
    virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv) = 0;
    virtual ULONG __stdcall AddRef() = 0;
    virtual ULONG __stdcall Release() = 0;
};

```

Рис. 5.5. Интерфейс IUnknown

Функция *QueryInterface* предназначена для запроса адреса *ppv* интерфейса, идентифицируемого глобальным идентификатором *iid*. Функция *AddRef* предназначена для инкрементации, а функция *Release* для декрементации числа ссылок на интерфейсы с целью определения времени жизни компонента. Поскольку все интерфейсы COM наследуют *IUnknown*, в каждом интерфейсе есть функции *QueryInterface*, *AddRef* и *Release* — первые три функции в таблице виртуальных функций класса.

5.5.3. Библиотека COM

Все клиенты и компоненты COM выполняют много типовых операций, которые стандартизированы в библиотеке функций COM, доступной посредством заголовочного файла *objbase.h*. Для инициализации библиотеки процесс клиента вызывает функцию *CoInitialize*, для завершения работы с библиотекой процесс вызывает функцию *CoUninitialize*. Создание компонентов выполняется функцией *CoCreateInstance*, которая, получив от клиента глобальный идентификатор компонента *CLSID*, создает экземпляр компонента и возвращает адрес интерфейса, запрашиваемого посредством идентификатора *IID*. Функция *CoCreateInstance* создает и использует фабрику класса для экспортирования запрашиваемого компонента из библиотеки *dll*. Фабрика класса — это компонент, задачей которого является создание других компонентов. Экспортирование компонента и управление подключением библиотеки выполняется посредством функций *DllGetClassObject* и *DllCanUnloadNow*, которые не принадлежат библиотеке COM, но пишутся разработчиком приложения и подключаются посредством *.def* файла. Полные имена файлов компонентов, индексированные *CLSID*, помещаются в реестр операционной системы.

5.5.4. Сервер в процессе, локальный и удаленный сервер

В ряде случаев предпочтительнее реализовать компонент в грузочном модуле *.exe*, а не в библиотеке *.dll*. Тогда клиент и сервер находятся в разных адресных пространствах, что в корне меняет механизм передачи данных между клиентом и сервером. Компонент в *.dll* называется сервером *внутри процесса*, а компонент в *.exe* называется сервером *вне процесса*. Компонент в *.exe* называется *локальным сервером*, если он расположен на той же машине, что и клиент. *Удаленный сервер* — это компонент в *.exe*, работающий на другой машине. Средством коммуникации между различными процессами, работающими на одной машине, является *локальный вызов процедуры (Local Procedure Call – LPC)*. Средством коммуникации между различными процессами, работающими на разных машинах, является *удаленный вызов процедуры (Remote Procedure Call – RPC)*, построенный на использовании разнообразных сетевых протоколов. Процесс передачи параметров функции в случае ее вызова в одном процессе и реализации в другом процессе называется *маршалингом (marshaling)*. Если процессы находятся на одной машине, маршалинг копирует данные с учетом различий адресов, если на разных машинах, данные преобразуются в стандартный формат, учитывающий межмашинные различия. В процессе клиента компонент представлен *.dll*, называемой *заместителем (proxy)*. В процессе сервера клиент представлен *.dll*, называемой *заглушкой (stub)*. Маршалинг выполняется в двух направлениях: от клиента к серверу и обратно. Заместитель и заглушка генерируются автоматически по описанию интерфейса на языке IDL (*Interface Definition Language*).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Лорин, Г. Распределенные вычислительные системы / Г. Лорин. – М.: Радио и связь, 1984.
2. Tanenbaum, A. and Van Steen, M., Distributed Systems / A. Tanenbaum and M. Van Steen // Pearson Prentice Hall, NJ. – 2007. – 705 p.
3. Прихожий, А. А. Распараллеливание и планирование вычислительных и информационных процессов / А. А. Прихожий // Доклады БГУИР. – 2003. – № 4. – С. 104–114.
4. Прихожий, А. А. Модель и алгоритм оптимизации назначения объектов на узлы распределенной информационно-вычислительной системы / А. А. Прихожий // Информатика. – 2010. – № 4.
5. Kafil, M. Optimal Task Assignment in Heterogeneous Distributed Computing Systems / M. Kafil, I. Ahmad // IEEE Concurrency, July-September. – 1998. – pp. 42–51.
6. Prihozhy, A. Net Scheduling in High-Level Synthesis / A. Prihozhy // IEEE Design & Test of Computers". – Spring, 1996. – pp. 26–35.
7. Prihozhy, A. Evaluation of Parallelization Potential for Efficient Multimedia Implementations: Dynamic Evaluation of Algorithm Critical Path / A. Prihozhy, M. Mattavelli, D. Mlynek // IEEE Trans. on Circuits and Systems for Video Technology. – Vol. 15. – No. 5. – May 2005. – pp. 593–608.
8. Эндрюс, Г. Р. Основы многопоточного, параллельного и распределенного программирования / Г. Р. Эндрюс // М. : – Вильямс, 2003.
9. Шпаковский, Г. И. Программирование для многопроцессорных систем в стандарте MPI / Г. И. Шпаковский, Н. В. Серикова // Минск : БГУ, 2002. – 323 с.
10. OpenMP [Электронный ресурс]: – Режим доступа: <http://openmp.org/mp-documents/>. – Дата доступа: 07.09.2014.
11. Эммерих, В. Конструирование распределенных объектов / В. Эммерих. – М. : МИР, 2002.
12. Роджерсон, Д. Основы СОМ / Д. Роджерсон. – М. : Издат.-торговый дом «Русская редакция». – 2000.
13. Мюллер, Д. Технология СОМ+ / Д. Мюллер. – СПб. : Питер Бук, 2002.

Учебное издание

ПРИХОЖИЙ Анатолий Алексеевич

РАСПРЕДЕЛЕННАЯ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

Учебно-методическое пособие
для студентов специальности 1-40 01 01
«Программное обеспечение информационных технологий»
и направления специальности 1-40 05 01 04
«Информационные системы и технологии
(в обработке и представлении данных)»

Редактор *О. В. Ткачук*
Компьютерная верстка *Н. А. Школьниковой*

Подписано в печать 09.12.2016. Формат 60×84 ¹/₁₆. Бумага офсетная. Ризография.
Усл. печ. л. 5,35. Уч.-изд. л. 4,18. Тираж 100. Заказ 885.

Издатель и полиграфическое исполнение: Белорусский национальный технический университет.
Свидетельство о государственной регистрации издателя, изготовителя, распространителя
печатных изданий № 1/173 от 12.02.2014. Пр. Независимости, 65. 220013, г. Минск.