#### 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

1

УДК 004.272.2 (075.8)

ББК 32.97я7

П77

Р е ц е н з е н т ы:

*В. С. Муха*, *В. А. Вишняков*

П77

**Прихожий, А. А.**

Распределенная и параллельная обработка данных : учебно-мето- дическое пособие для студентов специальности 1-40 01 01 «Про- граммное обеспечение информационных технологий» и направления специальности 1-40 05 01 04 «Информационные системы и техноло- гии (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 специ- альности 1-40 01 01 «Программное обеспечение информационных технологий»

и направления специальности 1-40 05 01 04 «Информационные системы и техноло- гии (в обработке и представлении данных)».

Рассматриваются распределенные и параллельные системы, задачи, модели и ал- горитмы планирования процессов обработки данных, языки и инс трументы програм- мирования 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

2

Оглавление Введение 5

1. Системы распределенной и параллеьной обработки данных 6
   1. [Классификация распределенных систем 6](#_TOC_250006)
   2. [Архитектуры распределенных систем 6](#_TOC_250005)
   3. [Свойства распределенных систем 7](#_TOC_250004)
   4. [Классификация параллелизма  8](#_TOC_250003)
   5. [Закон Амдала 10](#_TOC_250002)
   6. [Задачи распределенных и параллельных систем 13](#_TOC_250001)
2. Планирование синхронных параллельных процессов

[обработки данных 14](#_TOC_250000)

* 1. Постановка задачи 14
  2. Классификация задач и стратегий планирования 16
  3. Обыкновенное планирование 17
  4. Многошаговое планирование 23
  5. Цепочечное планирование 29
  6. Сравнение стратегий планирования 31
  7. Свертывание графа распараллеленности операций 32
  8. Сведение планирования к задаче целочисленного

линейного программирования 39

1. Планирование асинхронных распределенных параллельных процессов с учетом обмена данными 44
   1. Граф задач 44
   2. Асинхронный параллельный план 46
   3. Задача минимизации временной длины плана 47
   4. Стратегия планирования «Ранняя задача первая» 49
   5. Стратегия планирования «Зануление дуг» 54

3

* 1. Стратегия планирования «Группировка доминирующей последовательности» 57
  2. Планирование с использованием мобильности задач 61

1. Планирование решения задач в разнородной

распределенной системе 64

* 1. Модель разнородной системы 64
  2. Назначение задач на узлы 68
  3. Оценка общего времени решения задач 69
  4. Алгоритм оптимального назначения задач

на процессоры 70

1. Языки и инструменты программирования распределенной

и параллельной обработки данных 76

* 1. Многопоточные приложения 76
  2. Интерфейс MPI 79
  3. Открытый стандарт OpenMP 82
  4. Технологический стандарт CORBA 84
  5. Модель компонентных объектов COM 85

Список использованных источников 91

4

#### ВВЕДЕНИЕ

Целями создания и повсеместного использования распределен- ных и параллельных систем являются:

* + - обеспечение быстрого и легкого доступа к удаленным ресур- сам и разделение их между пользователями прозрачным и эффек- тивным способом;
    - обеспечение эффективного взаимодействия пользователей и программ и обмена информацией между ними;
    - обеспечение логической прозрачности системы и скрытие от пользователей физической архитектуры распределенных и парал- лельно работающих ресурсов;
    - обеспечение открытости распределенной системы посредст- вом предоставления сервисов, доступных пользователям по стан- дартным правилам, определяющим общие синтаксис и семантику сервисов;
    - организация параллельной эффективной работы вычислитель- ных и информационных ресурсов при решении задач большой вы- числительной сложности;
    - обеспечение масштабируемости системы в плане: размера, из- меряемого числом пользователей и ресурсов; географического рас- положения, проявляющегося в отдаленном размещении пользовате- лей и ресурсов; административного управления, когда система объ- единяет много административных организаций.

Виртуализация, управление и планирование процессов являются наиболее значимыми задачами в распределенных и параллельных системах. Важнейшими параметрами систем являются высокая про- изводительность, высокая загрузка оборудования, устойчивость

и надежность в работе, низкое энергопотребление и др. Структур- ная 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 парал- лельных систем, менее всего освещенных в учебной литературе.

Оно уделяет также внимание эффективным и востребованным сред- ствам разработки параллельных и распределенных приложений.

5

#### СИСТЕМЫ РАСПРЕДЕЛЕННОЙ И ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ ДАННЫХ

#### Классификация распределенных систем

Различают три класса распределенных систем [1, 2]: *вычислитель- ные*, *информационные* и *встроенные*. Системы из первого класса предназначены для высокопроизводительных параллельных вычисле- ний. Этот класс делится на два подкласса: *кластерные* системы и *grid*- системы. Кластерные системы состоят из однотипных рабочих стан- ций или персональных компьютеров, тесно соединенных высокопро- изводительной локальной сетью. На каждом узле кластера установлена одна и та же операционная система. *Grid*-системы состоят из объеди- ненных, но административно разделенных компьютерных систем, ко- торые могут сильно различаться как по установленной аппаратуре, так и используемому программному обеспечению.

Второй класс составляют распределенные информационные сис- темы, состоящие, как правило, из *серверных* и *клиентских* частей.

Этот класс делится на два подкласса: системы обработки *транзак- ций* баз данных и системы *интеграции* приложений. И те и другие активно используют механизм удаленного вызова процедур. Разра- ботка систем интеграции приложений основана на использовании различных моделей коммуникации компонентов приложения.

Третий класс составляют *всеохватывающие* встроенные распре- деленные системы, окружающие человека. В первую очередь к ним относятся малые мобильные системы с беспроводным соединением. Подкласс домашних распределенных систем строится из компьюте- ров и множества устройств потребительской электроники. Подкласс медицинских распределенных систем выполняет мониторинг состо- яния пациента и при необходимости взаимодействует с врачом.

Сенсорные распределенные си стемы состоят из десятков и сотен тысяч сенсорных устройств, которые могут рассматриваться как распределенная база данных.

#### Архитектуры распределенных систем

Различают следующие архитектурные стили распределенных систем:

6

1. структурированные по уровням (последующий уровень вызы- вает компоненты нижестоящего уровня);
2. объектно-ориентированные (объекты взаимодействую посред- ством 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 взаимо- действуют через распространение событий).

Архитектуры распределенных систем делятся на:

1. централизованные (множество клиентов обслуживается од- ним сервером);
2. децентрализованные (многосвязные клиент-серверные при- ложения, 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 децен- трализованными частями);

#### Свойства распределенных систем

Распределенные системы характеризуются следующими свой- ствами:

1. скрытие от пользователей различий между компьютерами, включенными в распределенную систему, и между способами вза- имодействия компьютеров;
2. использование единого согласованного и унифицированого способа взаимодействия всех пользователей в системе;
3. обеспечение легкой расширяемости и неограниченной мас- штабируемости системы, не оказывающих влияние на работу поль- зователей;
4. поддержка разнородных компьютеров и сетей посредством организации системы в виде специального *промежуточного* слоя программного обеспечения (*middleware*), интегрирующего все ре- сурсы таким образом, что они доступны всем потребителям.

7

#### Классификация параллелизма

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

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

***Параллелизм в пространстве*.** Он свойственен системам, одно- временно обрабатывающим один входной набор данных таки м об- разом, что все операции алгоритма принимают на входе элементы этого набора, либо значения, являющиеся производными от этих элементов, полученные операциями-предшественниками. В конеч- ном счете, это приводит к вычислению результирующих значений выходного набора. Операции, выполняющиеся параллельно в про- странстве, взаимно не предшествуют друг другу и являются инфор- мационно 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 результиру- ющую переменную *a* операции «+». Операции являются взаимно независимыми и выполняются параллельно.

x

**+**

**\***

a

y

b

z

Рис. 1.1. Пример параллелизма в пространстве

*Параллелизм во времени*. Он свойственен системам, обрабаты- вающим поток данных, состоящий из последовательности входных наборов данных. Параллелизм во времени называется также кон- вейерным параллелизмом. Конвейер состоит из ступеней, выпол-

8

няющих преобразование информации и взаимодействующих таким образом, что выходные данные одной ступени являются входными данными последующей ступени. Ступени соединены последова- тельно, однако все они работают параллельно на различных набо- рах данных, которые шаг за шагом проталкиваются через конвейер. Число одновременно обрабатываемых наборов равно числу ступе- ней конвейера. Более того, порядок обработки последующего набо- ра 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 сту- пень  из операции «\*». На вход конвейера подается поток данных, состоящий из наборов *x* 1, *y*1, *z*1,  , *x n*, *yn*, *zn*, на выходе первой сту- пени генерируется поток переменных *b* 1, , *b n*, на выходе второй ступени появляется поток переменных *a* 1, , *a n*, являющийся вы- ходным потоком. В момент вычисления второй ступенью и опера- цией «\*» значения переменной *a* i первая ступень и операция + вы- числяют значение переменной *bi*+1. Операции «+» и «\*» выполняют- ся одновременно (параллельно) во времени, но над различными наборами данных.

xnx1

ступень 1ступень 2

**+**

bnb1

**\***

yny1

znz1

ana1

Рис. 1.2. Пример параллелизма во времени (конвейер)

***Смешанный параллелизм*.** В системе со смешанным паралле- лизмом одновременно присутствует параллелизм в пространстве

и параллелизм во времени. Параллелизм в пространстве реализуется внутри каждой ступени, параллелизм во времени проявляется в од- новременном выполнении операций на разных ступенях.

9

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

xnx1

ступень 1ступень 2

**+**

bnb1

**\***

**+**

cnc1

yny1

znz1

ana1

Рис. 1.3. Пример смешанного параллелизма

#### 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. Рассмот- рим 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 парал- лельных систем является ускорение *R* решения задачи параллельной многопроцессорной системой по сравнению с последовательной однопроцессорной системой:

(1.1)

*R* 

*T*

1

*T*

,

*n*

где *T*1  время решения задачи на однопроцессорной системе;

*Tn*  время решения той же задачи на *n*-процессорной системе.

Выразим *T* 1 и *T n* через основные параметры последовательного и параллельного алгоритмов.

10

Пусть *W*  общее число операций в алгоритме решения задачи. Условно все множество операций можно разделить на два подмно- жества: подмножество последовательно выполняемых операций

и подмножество параллельно выполняемых операций. Пусть *W* посл  число операций в первом подмножестве, *W* пар  число операций во втором подмножестве. Очевидно, что *W = W*посл *+ W*пар.

Закон Амдала не учитывает специфику и типы операций, по- средством которых описывается алгоритм, но использует среднее время *t* выполнения одной операции, как последовательной, так

и параллельной. Благодаря этому соотношение (1.1) можно перепи- сать в виде

*R*  *Wt*

(*W*  *W* пар )*t*

 1 ,

*a*  1 *a*

(1.2)

посл *n n*

где *a = W* посл / *W*  доля последовательных операций в общем числе операций алгоритма. При числе процессор*о*в *n*, стремящемся к бес- конечности, ускорение *R* стремится к величине 1/*a*.

Закон Амдала определяет два основных положения, которые яв- ляются 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 *а* последовательно выполняемых операций

и долей 1*-а* операций, которые могут выполняться параллельно.

1. Ускорение зависит от параметров многопроцессорной систе- мы, главным образом от числа входящих в систему процессоров *n*. Согласно (1.2), для чисто последовательного алгоритма с нуле- вым уровнем потенциального параллелизма, когда *a* = 1, для уско-

рения всегда выполняется *R* = 1, что означает отсутствие ускоре- ния не зависимо от числа процессоров *n*. Обратно, для однопроцес- сорной системы, когда *n* = 1, для ускорения также всегда выполня- ется *R* = 1, что означает отсутствие ускорения не зависимо от свойств распараллеливаемого алгоритма. Для всех остальных слу- чаев, когда *a* < 1 и *n* > 1, коэффициент ускорения превышает значе- ние единица: *R* > 1.

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

11

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

Выполним корректировку выражения (1.2), введя дополнитель- ные величины, характеризующие функционирование сети связи.

Пусть *W*c  количество операций передачи данных между процессо- рами; *t* c  среднее время выполнения одной операции. Тогда закон Амдала переписывается в виде

*R*с 

(*W* посл 

*Wt W* пар *n*

)*t*  *W* с*t*с

 1 ,

*a*  1  *a*  *c n*

где *c*  коэффициент сетевой деградации вычислений в распреде- ленной параллельной системе, определяемый выражением:

 *W* с*t*с  *c*а*c*т.*c*

*Wt*

Коэффициент *c* определяет отношение общего времени выпол- нения всех операций передачи данных в сети к общему времени выполнения всех операций алгоритма на процессорах. Первая со- ставляющая *с* а = *W* c/*W* определяет алгоритмическую часть коэффи- циента деградации, обусловленную свойствами параллельного ал- горитма, а именно отношением числа операций обмена к числу опе- раций на процессорах. При качественной разработке параллельной программы число операций обмена должно быть значительно мень- ше числа операций на узлах. Вторая составляющая *с* т = *t*c/*t* опреде- ляет техническую часть коэффициента деградации, обусловленную свойствами коммутационной сети, а именно, отношением среднего времени выполнения операции обмена к среднему времени выпол- нения операции на узле. Для реальных многопроцессорных систем время операции обмена значительно больше времени операции на процессоре. В целом, коэффициент деградации *с* может принимать значения, как меньшие, так и больш ие 1. Благодаря этому, коэффи- циент ускорения *R* c, в отличие от коэффициента *R*, может прини- мать значения, меньшие 1, что означает замедление вычислений по сравнению с однопроцессорным вариантом.

12

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

#### Задачи распределенных и параллельных систем

Существует множество теоретических и практических задач, ко- торые решаются при разработке и эксплуатации распределенных

и параллельных систем:

1. математическое и имитационное моделирование с целью ис- следования характеристик системы и принятия обоснованных ре- шений в период проектирования или эксплуатации;
2. разработка архитектуры и выбор базового аппаратного обес- печения;
3. выбор программного обеспечения для уз лов системы и про- граммного обеспечения среднего слоя;
4. разработка архитектуры параллельной распределенной систе- мы в случае необходимости реализации высокопроизводительных вычислений;
5. создание адекватных математических и компьютерных моде- лей распределенных и параллельных систем;
6. разработка методов, стратегий и алгоритмов планирования информационно-вычислительных процессов в распределенных и па- раллельных системах;
7. создание и использование языков и инст рументальных средств параллельного и распределенного программирования;
8. разработка параллельных и распределенных алгоритмов для решения важнейших задач науки и техники;
9. автоматическое распараллеливание последовательного про- граммного кода, создание и использование распараллеливающих компиляторов;
10. экстракция параллелизма и оптимизация параллельных и рас- пределенных программ.

Важнейшие параметры параллельной распределенной системы  время решения задач (производительность си стемы) и объем ис- пользуемых вычислительных ресурсов.

13

#### ПЛАНИРОВАНИЕ СИНХРОННЫХ ПАРАЛЛЕЛЬНО- ПОСЛЕДОВАТЕЛЬНЫХ ПРОЦЕССОВ ОБРАБОТКИ

**ДАННЫХ**

#### Постановка задачи

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

Граф предшес- твования операций

Критерий оптими- зации

Ограничения

Шаги управления

Назначение операций на шаги

Плани- рование

Параметры плана

Рис. 2.1. Задача планирования процессов обработки данных

Вычислительный процесс представляется графом *G* H = (*N*, *H*) предшествования операций, в котором *N* = {1, , *n*}  множество вершин-операций; *H*  множество ребер, описывающих отношение предшествования операций. Каждая операция *i*  *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 си- стемой ограничений. Важнейшими оптимизируемыми параметрами синтезируемых планов являются время *Time* реализации плана

и стоимость *Cost* потребляемых планом ресурсов. Время реализа- ции оценивается выражением *Time* = *T* step  *Steps*, где *T step*  время шага; *Steps*  число шагов управления, введенных в план. Стои-

14

мость *Cost* оценивается либо общим количеством *P* процессоров, необходимых для выполнения плана, либо общей стоимостью про- цессорных элементов

*Cost*  *Types ci pi* ,

*j*1

где *Types*  число типов процессорных элементов;

*ci*  стоимость процессора *i*-го типа;

*pi*  число процессоров *i*-го типа, обрабатывающих соответст- вующий 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* .

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 |

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

15

ресурсов. Сокращение числа шагов делает план более распаралле- ленным, сокращает время реализации и увеличивает объем исполь- зуемых ресурсов.

Рис. 2.2. Пример графа *G*H предшествования операций

**1** +

+ **3**

+ **6**

**2** \*

\* **4**

\* **7**

+ **5**

+ **8**

#### Классификация задач и стратегий планирования

Планирование обработки данных классифицируется по следую- щим признакам:

1. По структуре синтезируемого плана:
   * + *обыкновенное* (*ordinary*), в котором каждая операция выпол- няется на одном шаге управления, операции из одного шага выпол- няются параллельно, а операции из различных шагов выполняются последовательно;
     + *многошаговое* (*multi cycling*), в котором одна операция может выполняться на нескольких соседних шагах управления, а в преде- лах одного шага все операции выполняются параллельно;
     + *цепочечное* (*chaining*), в котором операции могут образовы- вать цепочки и выполняться последовательно в переделах одного шага, операции из разных цепочек выполняются в пределах одного шага параллельно;
     + *конвейерное* (*pipelining*), в котором план разбивается на ступе- ни конвейера, обрабатывающие различные наборы данных парал- лельно во времени;

16

1. По решаемой оптимизационной задаче:
   * + с ограничением на время реализации плана (*time-constrained*)

и минимизацией объема потребляемых вычислительных ресурсов;

* + - с ограничением на объем потребляемых вычислительных ре- сурсов (*resource-constrained*) и минимизацией времени реализации плана;
    - с ограничениями на время реализации плана и объем потреб- ляемых вычислительных ресурсов  задача на достижимость (*feasi- ble-constrained*);

1. 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 ли- нейного программирования (*Integer Linear Programming Formu- lation  ILPF*);
    - сведение задачи планирования к свертыванию графа распа- раллеленности операций и другие стратегии.

#### Обыкновенное планирование

Все основные вышеперечисленные стратегии планирования поз- воляют синтезировать обыкновенный план, в котором каждая опе- рация выполняется ровно на одном шаге управления.

##### *Стратегия планирования ASAP*

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

Исходные данные:

1. Граф *G*H предшествования операций.

17

Результирующие данные:

1. Шаги управления;
2. Отображение операций на шаги управления;
3. Число процессоров каждого типа. Описание алгоритма:
4. Планирование выполняется в цикле, начиная с первого шага и кончая последним шагом;
5. Алгоритм начинает работу с введения первого шага управ- ления и назначения на него операций, не имеющих согласно графу *GH* операций-предшественников;
6. Алгоритм завершает работу в случае назначения всех опера- ций на шаги управления;
7. Для каждого шага *s* управления выполняются следующие действия по планированию:

* из множества всех операций *N* формируется подмножество *N* s

 *N* операций, не спланированных к моменту формирования теку- щего шага управления;

* для каждой операции *r*  *N* s проверяется, все ли операции- предшественники *pred*(*r*) уже спланированы на предшествую- щих шагах управления, другими словами проверяется включение *pred*(*r*)  *N\N*s;
* если включение выполняется, то операция *r* назначается на те- кущий шаг *s*, в противном случае операция остается не сплани- рованной.

*Пример* 2.2. Синхронный план ASAP, синтезированный для гра- фа предшествования операций, показанного на рис. 2.2, изображен на рис. 2.3. План построен на минимальном числе шагов управле- ния, равном 4. Время одного шага определяется максимальным временем выполнения одной операции. Два правых столбца пока- зывают 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.

18

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | **1** | **3** + |  | + |  |  | + | **6** | 3 | 0 |
| 2 | **2** | \* |  | \* | **4** |  | \* | **7** | 0 | 3 |
| 3 |  | + | **5** |  |  |  |  |  | 1 | 0 |
| 4 |  |  |  |  | + | **8** |  |  | 1 | 0 |

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



Шаг управления

Операции

Тип \* Т

Максимум = 3

3

##### *Стратегия планирования ALAP*

Алгоритм планирования ALAP стремится назначать операции на как можно более поздние шаги управления. Как и ASAP, алгоритм ре- шает 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:
4. Планирование выполняется в цикле, начиная с последнего шага и кончая первым шагом;
5. Алгоритм начинает работу с введения последнего шага управ- ления и размещения на нем операций, не имеющих операций-по- следователей;
6. Алгоритм завершает работу в случае назначения всех опера- ций на шаги управления;
7. Для каждого шага управления выполняются следующие дей- ствия по планированию:

19

* из множества всех операций *N* формируется подмножество *N*s  *N* операций, не спланированных к моменту формирования те- кущего шага управления;
* для каждой операции *r*  *N* s проверяется, все ли операции-по- следователи *succ*(*r*) уже спланированы на последующих шагах управ- ления, другими словами проверяется включение *succ*(*r*)  *N\N*s;
* если включение выполняется, то операция *r* назначается на текущий шаг *s*, в противном случае операция остается не сплани- рованной.

*Пример* 2.3. На рис. 2.4 изображен синхронный план ALAP, син- тезированный для графа *GH*, 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 пра- вых столбца показывают число процессоров типа «+» и типа «\*», необходимых для параллельного выполнения операций на каждом шаге управления, а общее колич ество 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.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | **1** | **3** + |  | + |  |  |  |  | 2 | 0 |
| 2 | **2** | \* |  | \* | **4** |  | + | **6** | 1 | 2 |
| 3 |  | + | **5** |  |  |  | \* | **7** | 1 | 1 |
| 4 |  |  |  |  | + | **8** |  |  | 1 | 0 |

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



Шаг управления

Операции

Тип \* Т

Максимум = 2

2

20

##### *Стратегия спискового планирования LS*

Алгоритм спискового планирования LS минимизирует число ша- гов управления и время выполнения плана при заданных ограниче- ниях на объем используемых вычислительных ресурсов (*resource- constrained scheduling*). Ограничения представляются числом до- ступных процессоров каждого типа. Название алгоритма подчерки- вает тот факт, что в процессе своей работы алгоритм активно ис- пользует 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, , *Types*.

Результирующие данные:

1. Шаги управления.
2. Отображение операций на шаги управления. Описание алгоритма:
3. 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*;
4. 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 планиро- ванию благодаря тому, что все их операции предшественники стали спланированными в результате формирования предыдущего шага управления на предыдущей итерации цикла;
5. Начальное состояние списка *List* формируется перед запуском цикла планирования. Первая часть списка является пустой, во вто- рую часть включаются операции, не имеющие операций предше- ственников в графе *GH*;
6. Когда первая и вторая части списка *List* становятся пустыми, то это значит, что все операции назначены на шаги управления. Ал- горитм LS завершает работу;

21

1. Для каждого шага управления выполняются следующие дей- ствия 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*  {1, , *Types*} такой, что для подмножества *L i*  *List* операций *r*  *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 процессо- ров типа *i*;
* для предпочтительного выбора операций используются кри- терии выбора. Наиболее часто используемым является критерий принадлежности операции критическому пути на графе *G H* предше- ствования 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 спланирован- ными, перемещаются в первую часть списка и становятся претен- дентами на включение в следующий шаг плана;
* во вторую часть списка *List* включаются новые ранее не спла- нированные 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. Гене- рируемый LS-план показан на рис. 2.5. Он построен на минималь- ном числе 5 шагов управления. Дополнительный шаг с номером 0 определяет начальное состояние списка *List*, включающего опера- ции 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-

22

рая уже назначена на первый шаг. На второй шаг управления назна- чаются 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 |  | Опе | раци | и |  | Список *List* |
| 0 |  |  |  |  |  | | 1, 3, 6 |
| 1 | **1** |  |  |  |  | 3, 6 | 2 |
| 2 | **2** |  | **3** |  |  | 6 | 4 |
| 3 |  |  | **4** |  | **6** | | 5, 7 |
| 4 |  | **5** |  |  | **7** | | 8 |
| 5 |  |  |  | **8** |  | | |

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

#### Многошаговое планирование

Принципы многошагового планирования:

1. Разрешается выполнение одной операции на нескольких со- седних 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

*tstep*, операция выполняется на одном шаге;

1. Если время *t i* выполнения операции *i* больше времени шага

*tstep*, операция выполняется на *k*  *ti*/ *tstep*  шагах, где    бли- жайшее целое, не меньшее *x*.

23

Преимущества многошагового планирования по сравнению с обык- новенным планированием:

1. Сокращение общего времени выполнения плана;
2. Увеличение загрузки оборудования; в отличие от многошаго- вого плана, в обыкновенном плане оборудование, назначаемое ко- ротким операциям, простаивает до завершения выполнения длин- ных по времени операций.

Алгоритмы многошагового планирования строятся путем моди- фикации алгоритмов ASAP, ALAP и LS.

##### *Многошаговое планирование на базе ASAP*

Алгоритм многошагового планирования MC-ASAP (*Multi Cy- cling* ASAP) учитывает соотношение между временем шага и вре- менем выполнения операции и стремится назначать каждую опера- цию на наиболее ранние шаги управления. Алгоритм решает опти- мизационную задачу на достижимость минимального числа шагов без учета ограничений на ресурсы.

Исходные данные:

* 1. Граф *GH* предшествования операций;
  2. Времена *ti*, *i*  *N* выполнения операций;
  3. Время *tstep* шага управления. Результирующие данные:

1. Шаги управления;
2. Отображение операций на шаги управления;
3. Число процессоров каждого типа. Описание алгоритма:
4. Планирование выполняется в цикле, начиная с первого и кон- чая последним шагом управления;
5. 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;
6. С каждой операцией*r*  *N s* ассоциируется время выполнения

*r*, не покрытое шагами управления, на которые назначена часть операции *r*. В момент начального включения *r* в подмножество *N s* время *r* = *tr*;

24

1. Алгоритм начинает работу до запуска цикла с поиска в под- множестве *r*  *N s* операций, не имеющих согласно графу *G H* опера- ций-предшественников (*pred*(*r*) = ), и перемещения их из *N s* в подмножество *Ns* с начальной установкой значений *r* = *tr*.
2. Для каждого шага *s* управления выполняются следующие дей- ствия по планированию:

* каждая операция *r*  *N s* назначается на шаг *s* управления; да- лее, если выполняется неравенство  *r*  *Tstep*, то операция *r* перемеща- ется из подмножества *N s* в подмножество *N s*, в противном случае осуществляется редуцирование *r* = *r*  *Tstep* с сохранением *r* в *Ns*;
* для каждой операции *r*  *N* s проверяется, все ли операции-пред- шественники *pred*(*r*) уже спланированы на предшествующих шагах управления, другими словами проверяется включение *pred*(*r*)  *N s*; если предшественники спланированы, операция *r* перемещается из подмножества *Ns* в подмножество *N s* с начальной установкой зна- чений *r* = *tr*;

1. Алгоритм завершает работу в случае появления пустого под- множества *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 управ- ления. Сгенерированный MC-ASAP-план показан на рис. 2.6. В от- личие 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 равн о 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.

25

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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



Шаг управления

Операции

Тип \*Т

##### *Многошаговое планирование на базе LS*

Алгоритм многошагового спискового планирования MC-LS (*Multi Cycling* LS) минимизирует время выполнения плана при за- данных ограничениях на объем используемых вычислительных ре- сурсов (*resource-constrained scheduling*), учитывая при этом соот- ношение между временем шага и временами выполнения операций, назначая при необходимости операцию на несколько соседних ша- гов управления. Ограничения представляются числом доступных процессоров каждого типа. Алгоритм MC-LS может быт ь построен как на базе стратегии ASAP, так и на базе стратегии ALAP. Остано- вимся на алгоритме, построенном на базе ASAP.

Исходные данные:

* 1. Граф *GH* предшествования операций;
  2. Времена *ti*, *i*  *N* выполнения операций;
  3. Время *tstep* шага управления;
  4. Число *pi* доступных процессоров типа *i* = 1, , *Types*.

26

Результирующие данные:

1. Шаги управления;
2. Отображение операций на шаги управления. Описание алгоритма:
3. Планирование выполняется в цикле, начиная с первого и кон- чая последним шагом управления. Алгоритм использует список *List* готовых к планированию операций. На текущий шаг могут назна- чаться только операции из *List*;
4. Список *List* состоит из трех частей. Первая часть *List* 1 вклю- чает операции, планирование которых начато, но не завершено на предыдущих шагах управления. Вторая часть *List* 2 включает опера- ции, 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;
5. С каждой операцией *r*  *List* ассоциируется время выполне- ния  *r*, не покрытое предыдущими шагами управления, на которые операция *r* частично назначена. В момент начального включения *r* в *List* время *r* = *tr*.
6. Алгоритм начинает работу до запуска цикла с поиска опера- ций, не имеющих согласно графу *G H* операций-предшественников (*pred*(*r*) = ), и включения их в часть *List* 3 с начальной установкой значений *r* = *tr*. Части *List*1 и *List*2 списка зануляются.
7. Для каждого шага *s* управления выполняются следующие действия по планированию:

* каждая операция *r*  *List* 1 назначается на текущий шаг *s* управления; далее, если выполняется неравенство  *r*  *T step*, то опе- рация *r* полностью спланирована и удаляется из списка *List*, в про- тивном случае осуществляется редуцирование времени  *r* = *r*  *Tstep* с сохранением *r* в *List*1;
* если на планирование операций из *List* 1 потрачена только часть процессоров, то оставшиеся процессоры могут быть исполь- зованы 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; опе- рация *r*, для которой *r*  *Tstep*, назначается на шаг *s* и исключается из

27

*List*, а операция *r*, для которой  *r* > *Tstep*, назначается на шаг *s* и пе- ремещается в *List*1 с пересчетом времени *r* = *r**Tstep*;

* 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 спланиро- ванными на предыдущих шагах управления, включая текущий шаг.

1. Алгоритм завершает работу в случае зануления списка *List*

и назначения всех операций на шаги управления.

*Пример* 2.6. Проиллюстрируем работу алгоритма MC-LS на гра- фе *GH*, изображенном на рис. 2.2. Примем время шага *T step* = 1 и вре- мя выполнения операций *t* + = 1 и *t*\* = 2. Допустим также, что для реа- лизации плана предоставлено по одному процессору каждого типа: pe+ = 1 и pe\* = 1. Сгенерированный MC-LS-план показан на рис. 2.7.

Шаг управления



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

| |

**8**

**7**

**5**

**4**

**6**

**2**

**3**

**1**

Операции

Список *List*

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

28

При выборе одной операции из множества конкурирующих опе- раций использован критерий принадлежности критическому пути на графе *GH*. План состоит из восьми шагов управления. Вспомога- тельный шаг 0 инициализирует список *List*. Две вертикальные чер- ты 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 уче- том *Tstep* = 1 составляет восемь единиц, в то время как время выпол- нения 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 обык- новенным списковым планированием.

#### Цепочечное планирование

Принципы цепочечного планирования:

1. Разрешается выполнение цепочек операций на одном шаге управления;
2. Под цепочкой понимается последовательность операций, в ко- торой соседние операции соединены дугой в графе *G H*, описываю- щей 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 обык- новенным планированием:

1. Сокращение числа шагов управления и общего времени вы- полнения синтезируемого плана;
2. Увеличение загрузки оборудования при совместном исполь- зовании с многошаговым планированием.

Алгоритмы цепочечного планирования строятся путем модифи- кации алгоритмов ASAP, ALAP и LS. Смешанные алгоритмы цепо- чечного и многошагового планирования строятся путем модифика- ции алгоритмов MC-ASAP, MC-ALAP и MC-LS.

Рассмотрим алгоритм C-ASAP цепочечного планирования, по- строенный на базе обыкновенного ASAP. Он стремится назначать операции на как можно более ранние шаги у правления, однако от-

29

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

Исходные данные:

* 1. Граф *GH* предшествования операций;
  2. Времена *ti*, *i*  *N* выполнения операций;
  3. Время *tstep* шага управления. Результирующие данные:

1. Шаги управления;
2. Отображение операций на шаги управления;
3. Число процессоров каждого типа. Описание алгоритма:
4. Планирование выполняется в цикле, начиная с первого шага и кончая последним шагом управления.
5. 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*  *Ns* операций, назначенных на текущий шаг.

1. С каждой операцией *r*  *Ns*, включенной в текущий шаг управ- ления, ассоциируется время *r* завершения ее выполнения с момента текущего шага управления.
2. Для каждого шага *s* управления выполняются следующие дей- ствия по планированию:

* организуется цикл по назначению еще не спланированных опе- раций 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*)  *Ns*;
* для операции *r* вычисляется время завершения выполнения

*r* =  + *t r*, где  = max{ *w*} по всем *w*  *W*, *W* = *pred*(*r*)  *N s*; если

*W* = , то  = 0;

* если *r*  *t step*, то операция *r* назначается на шаг *s* и перемеща- ется из подмножества *N s* в подмножества *N s* и *N s*, в противном случае она остается в подмножестве *Ns*;

1. Алгоритм завершает работу в случае назначения всех опера- ций на шаги управления.

30

*Пример* 2.7. На рис. 2.8 показан цепочечный план C-ASAP, син- тезированный для графа *G H* (из рис. 2.2) и заданных времени шага *Tstep* = 2 и времени выполнения операций *t* + = 1 и *t*\* = 2. Для постро- ения 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 време- ни выполнения обыкновенного плана ASAP.

Шаг управления



1

0

2

3

3

0

Максимум = 3

3

2

**8**

**5**

0

**7**

**4**

**2**

3

**6**

**3**

**1**

Операции

Тип \*Т

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

#### Сравнение стратегий планирования

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

31

Время

10

LS(2)

8

6

5

2 3 4 5 6 Процессоры



M C -LS(2)

A LA P

ASAP

C-ASAP

M C-ALAP

M C-ASAP

M C-LS(3 )

Рис. 2.9. Сравнение стратегий планирования по времени и ресурсам

Каждый план представлен одной точкой в пространстве «процес- сорывремя», помеченной именем стратегии. В круглых скобках ука- зано число использованных процессоров. Соединение точек линиями дало два графика: первый для стратегий обыкновенного планирова- ния, второй для стратегий многошагового планирования. Стратегии цепочечного планирования представлены одной точкой. Первая за- кономерность, имеющая место для обоих граф иков, 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 многошаго- вого и цепочечного планирования дали лучшие результаты по срав- нению со стратегиями обыкновенного планирования.

Стратегии спискового планирования, будь то обыкновенные, мно- гошаговые или цепочечные, ориентированы на практическо е при- менение, так как они учитывают объем имеющихся у пользователей ресурсов. Планы с минимальным или близким к минимальному чис- лом шагов могут быть синтезированы ими для любого числа про- цессоров, что приводит к построению области Парето.

#### Свертывание графа распараллеленности операций

Пусть *N* = {1, , *n*}  множество операций. Определим последо- вательно-параллельный план выполнения операций из множества *N* рекурсивно, используя суперпозицию двух функций *seq*(*s* 1, , *s k*)

32

и *par*(*s*1, , *s k*), где *s i* при *i* = 1, , *k*  последовательно-параллель- ные частичные планы. Функция *seq* (от *sequential*) называется «по- следовательный», функция *par* (от *parallel*) называется «параллель- ный». Последовательно-параллельный план есть:

1. отдельная операция *i*  *N*;
2. план, описываемый функцией *seq*(*s*1, , *sk*) последовательного выполнения частичных планов *s*1, , *sk*;
3. план, описываемый функцией *par*(*s*1, , *sk*) параллельного вы- полнения частичных планов *s*1, , *sk*.

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

S1

Sn

S1

Sn

Рис. 2.10. Графическое изображение функции *seq*

Рис. 2.11. Графическое изображение функции par

*Граф распараллеленности операций* определяется как *GR* = (*V*, *R*) [3], где *V*  множество вершин, являющихся частичными последо- вательно-параллельными планами и, в частном случае, операциями; *R*  множество ребер, описывающих отношение распараллеленно- сти частичных планов (операций). Граф является неориентирован- ным. Две вершины *i*, *k*  *V* соединены ребром, если соответствую- щие частичные планы выполняются параллельно, если вершины не соединены ребром, соответствующие планы выполняются последо- вательно. Граф является взвешенным. Каждой вершине *i*  *N* ста- вится в соответствие два параметра: время *T i* выполнения частично- го плана (в частном случае, это время *t* i выполнения операции)

и вектор *b i* = *(bi* , , *b i )* чисел *b i* процессоров типов *j* = 1, , *m*,

1 *m j*

необходимых для выполнения плана.

Исходный граф *G* 0*R* на множестве вершин-операций строится по

графу *GH*. В граф *G* 0 вводится ребро (*i*, *k*), если на ориентированном

*R*

33

графе *G*H не существует путь, соединяющий операции *i* и *k*, в против- ном случае ребро не вводится. Для вершины *i* определяются вес *T i* = *ti* и вектор *bi* = (0, , 1, , 0), где 1 находится в позиции *type*i.

*Пример* 2.8. На рис. 2.12 приведен исходный граф *G* 0*R* распарал- леленности операций, построенный на восьми вершинах по графу *GH* предшествования операций, показанному на рис. 2.2. Он содер- жит пять вершин-операций 1, 3, 5, 6 и 8 типа «+» с временем вы- полнения *t*+ = 1 и три вершины-операции 2, 4, и 7 типа «\*» с време- нем выполнения *t*\* = 2. Вершины типа «+» метятся векторами (1, 0), а вершины типа «\*» метятся векторами (0, 1).

1 (1,0)

(1,0) 1 2

1 8 2 (0,1)

2 7 3 1

(0,1) (1,0)

1 6 4 2

(1,0)

5

1 (1,0)

(0,1)

Рис. 2.12. Пример графа *GR* распараллеленности операций

Граф распараллеленности *G* 0*R* является исходными данными для синтеза последовательно-параллельного плана методом свертыва- ния. Свертывание графа *G* 0*R* выполняется посредством пошагового склеивания пар вершин, при этом в графе появляются более слож- ные вершины, которым соответствуют частичные последовательно- параллельные планы и которые также метятся двумя интегрирован- ными величинами:

1. временем *Ti* выполнения частичного плана;
2. вектором *b i* чисел процессоров различных типов, необходи- мых для реализации частичного плана.

34

Рассмотрим последовательно-параллельное планирование с целью минимизации времени решения задачи при заданных ограничениях

)

на ресурсы. Ограничения представим вектором *b* max = (*b* max, , *b*

1

максимального числа используемых процессоров каждого типа.

max

*m*

*Свертывание графа G R посредством склеивания пар вершин*.

Пусть на шаге *s* планирования в графе *G sR* для склеивания выбраны вершины *x* и *y*. В результате склеивания *x* и *y* образуется новая вер- шина с именем *xy*. Граф *G sR* преобразуется путем удаления вершин *x* и *y*, удаления множества ребер, инцидентных этим вершинам, а так- же путем добавления вершины *xy* и добавления ребер, инцидентных этой вершине и вершинам из множества *x*  *y*, где (*x*)  мно- жество вершин, смежных с вершиной *x* в графе *GsR*. Если вершины *x* и *y* соединены ребром, они склеиваются параллельно, при этом но- вой вершине соответствует параллельный план *par*(*x*, *y*). Если вер- шины 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* вычисля- ются следующим образом:

1. если вершины склеиваются параллельно, то *T xy* = max(*Tx*, *Ty*)

и *bxy* = *bx* + *by*;

1. если вершины склеиваются последовательно, то *T xy* = *T x* + *Ty*

и *bxy* = max(*bx*, *by*) = (max(*bx*1, *by* ), , max(*bxm*, *bym*)).

1

Параллельное склеивание вершин увеличивает число используе- мых процессоров, последовательное склеивание увеличивает время выполнения плана. Параллельное склеивание вершин не должно при- водить к нарушению ограничений на максимальное число процес- соров, описываемых вектором *b* max. Если нарушение ограничений имеет место, ребро может быть заранее удалено из графа *GsR*.

Различный порядок склеивания пар вершин приводит к построе- нию различных последовательно-параллельных планов. Существен- ное 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 верши- ной *x* и не смежная с вершиной *y*, т. е. *z*  *x* и *z*  *y*, либо смежная с вершиной *y* и не смежная с вершиной *x*, т. е. *z*  *x*

35

и *z*  *y*. В результате такого склеивания новая вершина *xy* и вер- шина *z* не соединены ребром. Это означает потерю потенциального параллелизма между вершинами *xy* и *z*, что может сказаться отрица- тельно на минимизации времени выполнения синтезируемого пла- на. В силу сказанного, предпочтительным является выбор пар вер- шин, склеивание которых минимизирует потерю ребер.

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

*Пример* 2.9. Выполним свертывание графа *G* 0*R* распараллеленно- сти 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 сра- зу склеиваются три вершины. Именование вновь образуемых вер- шин показывает характер склеивания, например, получаемая на ша- ге a) новая вершина именуется 34 и с ней ассоциируется план, за- ключающийся 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, последова- тельное 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 пре- дыдущем 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).

36

1 (1,0)

(1,0) 1 2

1 8 2 (0,1)

(1,0)

1 8

3 (1,1)

12

2 7 34 3 2 7 34 3

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

1 6 1 6

(1,0)

5

1 (1,0)

(1,0)

5

1 (1,0)

*а б*

(1,0) (1,0)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 8 |  |  |  |  |  | 1 |  | 8 |  |  |  | |
| 2 | 7 |  |  |  |  | 3 | 3 | 67 |  |  |  |  | 12/34 | 3 |

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

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | 12/34 |

1 6

(1,0)

5

1 (1,0)

*в*

5

1 (1,0)

*г*

(1,0)

1

8

5/67

3

(12/34) (5/67) 8

3

12/34

7

(2,2)

(2,1) (2,2)

*д е*

Рис. 2.13. Пошаговый процесс свертывания графа распараллеленности операторов *G*0

*R*

**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 вер-

шины окрестностей представляют ребро (12, 34), а вершины 6, 7

представляют одинаковые оставшиеся части окрестностей. Следо-

37

вательно, вершины 12 и 34 могут быть склеены параллельно без потери ребер. Результат склеивания показан на рис. 2.13, *в*. Но-

вая вершина 12/34 метится временем *T* 12/34 = max(3,3) = 3 и векто- ром *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 ограни- чение 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 уда- ляются из графа, что показано на рис. 2.13, *в* двойными перечеркива- ющими 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. Ре- зультат 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 b5/67 = (1,0) + (1,1) = (2,1). Ей соответ- ствует 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 мо- гут быть склеены последовательно. Результат склеивания представ- лен 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 пока- зан 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 проек- тировании программного обеспечения так и при проектировании цифровой аппаратуры. Оно позволяет синтезировать оптимизиро- ванные планы функционирования многопоточных приложений для

38

многопроцессорных и многоядерных систем. Такого рода планиро- вание позволяет оптимизировать синхронные цифровые системы.

8

7

5

6

4

2

3

1

Рис. 2.14. Синтезированный последовательно-параллельный план

#### Сведение планирования к задаче целочисленного линейного программирования

Представление планирования в виде задачи целочисленного ли- нейного программирования ILPF является альтернативой стратегии спискового планирования LS. Однако, если LS является эвристиче- ским алгоритмом и находит в общем случае субоптимальное реше- ние, то ILPF является точным методом, гарантирующим нахождение глобального оптимума. Метод может быть сформулирован для обык- новенного, многошагового и цепочечного планирования. Он приме- ним также ко всем постановкам оптимизационных задач. Остановим- ся на обыкновенном планировании и рассмотрим задачу минимиза- ции 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, , *n*}  множество операций и *H*  *N*  *N*  бинарное отно- шение предшествования операций, получаемое в результате анализа информационных зависимостей между операциями.

39

Через *Si* обозначим наиболее ранний шаг выполнения операции *i* в соответствии с графом *GH*. Он удовлетворяет неравенству 1  *Si*  *T* и вычисляется посредством алгоритма ASAP. Через *L i* обозначим наиболее поздний шаг выполнения операции *i* в соответствии с гра- фом *GH* и требуемым числом шагов *T*. Он также удовлетворяет не- равенству 1  *L i*  *T*, однако вычисляется посредством алгоритма ALAP. Очевидно что *S i*  *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, , *Types*. Стоимость процессора типа *k* описывается величиной *сk*.

Для описания плана введем матрицу *X*, элемент *x ij* которой при *i*  *N* и *j* = 1, , *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 поста- новку задачи ILPF, описываемую целевой функцией и системой ограничений.

Целевая функция:

Система ограничений:

 

min

*Types*

# 

*k* 1

(*ckbk* ).

(2.1)

  *x* 

*b*  0

при 1  *j* *T*, 1  *k*  *Types*, (2.2)



 *i**N* ,

 *type*(*i*)*k*

*Li*

*ij*  *k*





# 

*j**Si*

*xij* 1

при 1  *i*  *n*, (2.3)

*Li*

 (*rxir* )

*r* *Si*

*Lj*

# 

*r* *S j*

(*rx jr* )  1

при (*i*, *j*)  *H*. (2.4)

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

40

типа *k* не должно превышать число *b k* имеющихся процессоров это- го типа. Поскольку значения свободных переменных *j* и *k* принад- лежат диапазонам 1  *j*  *T* и 1  *k*  *Types*, число линейных нера- венств в ограничении (2.2) равно *T*  *Types*.

Ограничение (2.3) выражает требование выполнения любой опе- рации ровно на одном шаге управления, что справедливо для обык- новенного планирования. Поскольку значение свободной перемен- ной *i* принадлежит диапазону 1  *i*  *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* пред- шествует операции *j* в гр афе *GH*. Поскольку число пар операций, для которых имеет место предшествование, равно числу ребер в графе *GH*, 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.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Шаг управления |  |  | ASAP |  |  |  |  | ALAP | |  |
| 1 | **1 3** |  | **6** |  |  |  |  |  |  |  |
| 2 | **2** |  | **4** |  | **7** | **1** |  | **3** |  |  |
| 3 |  | **5** |  |  |  | **2** |  | **4** |  | **6** |
| 4 |  |  |  | **8** |  |  | **5** |  |  | **7** |
| 5 |  |  |  |  |  |  |  |  | **8** |  |

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



41

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 опера- ций 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 |
| *Si* 1 | 1 |  | 3 |  |  | 1 | 4 | 2 | 2 | 2 |
| *Li* 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 опе- рации *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*11



*x*12

0 0 0 

0 *x*22

*x*23

0 0 

*x*31

*x*32

0 0 0 

*X*  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

84 *x*85 

Обозначая через *b*1 число процессоров типа «+», через *b* 2  число процессоров типа «\*». В соответствии с (2.1) целевую функцию за- дачи оптимизации запишем в виде

min *b*1

*b*1,*b*2

42

5*b*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  2 = 10. Однако, с учетом того, что операции типа «\*» не входят в шаги управления 1 и 5, число неравенств сокращается до 8:

*x*11 + *x*31 + *x*61  *b*1  0; *x*12 + *x*32 + *x*62  b1  0; *x*53 + *x*63  *b*1  0;

*x*54 + *x*84  *b*1  0;

*x*85  *b*1  0;

*x*22 + *x*42 + *x*72  *b*2  0; *x*23 + *x*43 + *x*73  *b*2  0; *x*74  *b*2  0.

Общее число равенств в ограничении (2.3), описывающем вхож- дение 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:

*x*11 + *x*12 = 1; *x*31 + *x*32 = 1; *x*53 + *x*54 = 1;

*x*61 + *x*62 + *x*63 = 1;

*x*84 + *x*85 = 1; *x*22 + *x*23 = 1; *x*42 + *x*43 = 1;

*x*72 + *x*73 + *x*74 = 1.

*x*11 + 2*x*12  2*x*22  3*x*23  1; 2*x*22 + 3*x*23  3*x*53  4*x*54  1; *x*31 + 2*x*32  2*x*42  3*x*43  1; 2*x*42 + 3*x*43  3*x*53  4*x*54  1;

3*x*53 + 4*x*54  4*x*84  5*x*85  1;

*x*61 + 2*x*62 + 3*x*63  2*x*72  3*x*73  4*x*74  1; 2*x*72 + 3*x*73 + 4*x*74  4*x*84  5*x*85  1.

Таким образом, сформулированная задача целочисленного линей- ного программирования состоит из целевой функции и системы огра- ничений, включающей 23 равенства и неравенства. Задача решается одним из известных методов, разработанных в теории математическо- го программирования. В частности, для решения задачи могут быть использованы метод ветвей и границ, генетический алгоритм. Быстрое решение в вещественной области может быть получено путем исполь- зования симплекс-метода с последующим отображением веществен- ных значений переменных *xij* на целочисленные значения.

43

#### ПЛАНИРОВАНИЕ АСИНХРОННЫХ РАСПРЕДЕЛЕННЫХ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ

**С УЧЕТОМ ОБМЕНА ДАННЫМИ**

Рассмотренные выше стратегии планирования синтезируют и оп- тимизируют синхронные планы, учитывая, главным образом, харак- теристики операций, выполняемых параллельно или последователь- но на узлах распределенной параллельной системы. Они не учиты- вают асинхронный обмен данными между операциями и между процессорами, на которых операции выполняются.

В настоящей главе рассматриваются модели и алгоритмы плани- рования, 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 справед- ливы для однородных распределенных систем. В неоднородных распределенных системах времена выполнения всех операций мо- гут изменяться в зависимости от параметров оборудования, на ко- тором эти операции выполняются.

#### Граф задач

*Граф задач* (*task graph*)  это ориентированный ациклический взвешенный граф *G* = (*V*, *E*), в котором *V*  множество узлов, пред- ставляющих задачи, *E*  множество дуг, представляющих передачу данных и отношение предшествования между задачами. Задача оп- ределяется как множество инструкций (операторов), выполняемых последовательно на одном процессоре. Соответствующая вершина *ni* графа метится числом *w*(*n i*), описывающим время решения зада- чи. Дуга (*n i*, *nj*) графа метится числом *c*(*i*, *j*), описывающим время передачи данных от задачи *ni* к задаче *nj*.

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

44

всех задач-предшественников на графе задач. При выполнении это- го условия задача немедленно запускается на выполнение, следова- тельно, поведение графа задач в период выполнения является асин- хронным. 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 зада- чи. 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 со- ставляет 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 ду- ги (*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 рав- но 43, суммарное время всех передач данных равно 32, коэффици- ент «коммуникация/вычисление» равен 0,73.

0



n 1

5

1

20

n 2

20

n3

10

1

10

n 4 8

20

40

53

Рис. 3.1. Пример графа задач

45

#### Асинхронный параллельный план

Асинхронный параллельный план генерируется в результате на- значения задач на процессоры и назначения передач данных между задачами на каналы передачи данных, соединяющие процессоры. Структура плана учитывает множество *P* = {1, , *k*}, где *k*  коли- чество процессоров в распределенной параллельной системе, и то- пологию *Y* = {(*v*, *u*) | *v*, *u*  *P*} сети связи, описываемую множеством пар процессоров, соединенных каналами передачи данных. Для каж- дого процессора *p*  *P* план определяет подмножество задач *V p*  *V*, решаемых на этом процессоре. Дуги графа задач, соединяющие за- дачи, назначаемые планом на один процессор, не показываются в пла- не, так как время передачи данных между двумя задачами, равное *c ij*, принимается равным нулю при назначении задач на один процес- сор. Задачи *i*, *j*  *V*, соединенные в графе *G* ребром (*i*, *j*)  *E*, могут быть назначены на разные процессоры *v*, *u*  *P* такие, что они со- единены каналом передачи данных: (*v*, *u*)  *Y*. Такое ребро пока- зывается в плане и метится временем *c ij* передачи данных. Все зада- чи *Vp*, назначаемые на один процессор *p*, упорядочиваются во вре- мени, и для каждой задачи *i*  *V p* определяется момент *t i* ее запуска на выполнение, при этом учитываются моменты завершения всех задач-предшественников и времена передачи данных от задач, на- значенных 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 ожидании передачи данных от задач-предшественников. Пе рвые 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

46

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 завер- шении передачи данных от *n* 1 с задержкой 1, в сумме в момент вре- мени 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 выполне- ния плана определяется максимальным временем работы каждого из двух процессоров. С учетом собственного времени решения за- дачи *n* 4 оно равно 35 единицам. Процессор PE0 тратит 23 единицы времени из 35 на решение трех задач, его полезная загрузка состав- ляет 65,7 %. Процессор PE1 тратит 20 единицы времени из 35 на решение задач, его полезная загрузка составляет 57,1 %.

PE0 PE1

0

n1 5

1

n3 10

n2 20

n4 8

1

15

35

Рис. 3.2. Пример асинхронного параллельного плана

#### 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 при- менимых к графу предшествования операций, стратегии планиро-

47

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 ока- заться 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 определяет- ся 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 при- своения задачам приоритета. Упорядочение задач в списке может быть статическим и динамическим. Статическое упорядочение вы- полняется один раз перед началом планирования. Динамическое переупорядочение повторяется после каждого шага планирования. Динамическое планирование выполняется в цикле, на каждой ите- рации 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 назначает-

ся задача, является наиболее раннее время начала решения задачи. Эта локальная эвристика хорошо коррелирует со стремлением син- тезировать план минимальной длины.

Критический путь  важное понятие, используемое стратегиями планирования. Критический путь на графе задач есть последова- тельность 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 гра- ницу временной длины плана.

К известным стратегиям планирования относятся:

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

48

1. группировка доминирующей последовательности» (*Dominant Sequence Clustering*  DSC);
2. управление мобильностью (*Mobility Directed*  MD);
3. модифицированный критический путь (*Modified Critical Path* 

MCP);

1. планирование на основе динамических уровней (*Dynamic Level Scheduling*  DLS);
2. и другие.

Рассмотрим в деталях четыре первые стратегии.

#### 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 приори- тетом 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 пла- нированию задач и выбирает задачу с минимальным временем запус- ка. Задача готова к планированию, если все предшествующие задачи уже спланированы. Наиболее раннее время запуска задачи вычисля- ется путем оценивания времен запуска задачи на всех возможных процессорах, включая ранее не использовавшийся процессор. Если две задачи имеют одинаковое наиболее раннее время запуска, выби- рается задача с более высоким статическим приоритетом.

Статические приоритеты вычисляются на основе применения

к графу задач стратегии ALAP и оценивания наиболее поздних вре- мен 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 приме- нения алгоритма ALAP к графу задач. На шкале слева отображено

49

наиболее позднее время запуска каждой из четырех задач. В табл. 3.1 задачи упорядочены по возрастанию времени, и каждой из них при- своен статический приоритет в возрастании от *n*1 до *n*4.

0



n 1

5

1

20

n 2

20

n 3

10

1

10

n 4 8

24

25

45

53

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 вер- шин в графе задач.

**Шаг 1.** Задача *n* 1 готова к планированию, имеет наивысший ста- тический приоритет 1 и соответственно минимальное время начала запуска на выполнение. Поскольку ни одного процессора еще не

50

задействовано, вводим процессор PE0 и назначаем на него вершину

*n*1 (рис. 3.4).

PE 0

0

n1 5

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 зану- ляются. Задача *n* 2 имеет более высокий статический приоритет 2 по сравнению с задачей *n* 3, имеющей приоритет 3, поэтому для плани- рования на шаге 2 выбирается задача n 2. Поскольку один процессор PE0 уже задействован, рассматриваем также попытку введения вто- рого 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 вари- ант I, его и выбираем для дальнейшего рассмотрения.

PE 0

I)

n1 5

n2 20

II)

PE0 PE1

5

n1

5

1

n2

20

6

Рис. 3.5. Шаг 2 стратегии ETF

51

**Шаг 3.** Задача *n* 3 готова к планированию, имеет статический приоритет 3 и наиболее позднее время запуска 25. Поскольку зада- чи *n* 1 и *n* 2 назначены на шаге 2 на процессор PE0, рассматриваем попытку введения второго процессора и снова исследуем два вари- анта I и II назначения узла n 3 на процессор PE0 и на новый процес- сор PE1 (рис. 3.6). В варианте I, так же как и в варианте II, время запуска задачи *n*3 равно 25. Для дальнейшего рассмотрения выбира- ем вариант I, поскольку он использует только один процессор.

PE 0

I)

n1

5

n2

20

n3 10

II)

PE0 PE1

25

n1

5

n2

20

20

n3 10

25

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 вре- мя начала выполнения 45. Поскольку задачи *n*1, *n*2 и *n*3 назначены на процессор PE0, рассматриваем попытку введения второго процес- сора PE1 и исследуем два варианта I и II назначения узла *n* 4 на про- цессор 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.

52

PE 0

1. )

n 1

5

n 2

2 0

n 3 1 0

n 4 8

1. )

P E 0 P E 1

3 5

n 1

5

n 2

2 0

n 3 1 0

1

1 0

n 4 8

4 5

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 за- дачами и между процессорами.

53

PE 0

0

n1

5

n2

20

n3 10

n4 8

20

43

Рис. 3.8. План ETF, построенный на 1 процессоре

#### Стратегия планирования «Зануление дуг»

Стратегия планирования EZ «зануление дуг» стремится сокра- тить длину частично построенного асинхронного плана на каждом шаге планирования путем рассмотрения дуги с максимальным вре- менем передачи данных. Стратегия назначает две задачи, соединен- ные наиболее «тяжелой» дугой, на один и тот же процессор при условии, что время частичного плана не увеличивается по сравне- нию с назначением зад ач на разные процессоры. Если время увели- чивается, задачи назначаются на разные подходящие процессоры.

Стратегия EZ сначала строит список дуг, упорядочивая их в не- возрастающем (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 процессо- ры. Если задачи назначаются на один процессор, дуга зануляется, что интерпретируется как немедленный запуск последующей задачи по- сле завершения предыдущей задачи с нулевым временем обмена данных на одном процессоре. Задачи, назначенные на один процес-

54

сор, упорядочиваются в соответствии с отношением предшествова- ния и по возрастанию их уровня в графе задач. Процесс планирова- ния заканчивается, когда все задачи назначены на процессоры.

Число шагов работы стратегии меньше числа задач в графе, по- скольку рассматриваемое на каждом шаге зануление одной дуги приводит к назначению на процессоры одной или сразу двух задач. Для выбора процессора, на который назначается задача, использу- ется критерий наиболее раннего времени запуска задачи или крите- рий 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 ва- рианте 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.

* 1. PE 0

0

n1 5

n3 10

15

0

20

35

PE0 PE1

1

5

20

n3 10

Рис. 3.9. Шаг 1 стратегии EZ

55

**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 вари- анте 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 процес- сор PE1. Длина плана в варианте I равна 23, длина плана в варианте II равна 33. Для дальнейшего планирования выбираем вариант I.

I) PE0

0

n1 5

n3 10

n4 8

15

II)

0

15

PE0 PE1

23

n1 5

n3 10

10

n4 8

33

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 вари- анта 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 про- цессор PE0, на который задача *n* 1 была назначена ранее. В варианте II задача n 1 остается назначенной на процессор PE0, задача *n* 2 на- значается 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 про- цессорах, поскольку все задачи уже назначены на процессоры, не смотря на то, что в списке осталась одна дуга (*n*2, *n*4).

56

I) PE 0

0

n1

5

n2

20

n3 10

n4 8

II)

0

15

PE0 PE1

25

n1 5

1

n3 10

n2

20

n4 8

1

35

43

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 процес- сор PE0 на 100 %, в то время как стратегия EZ загружает процессор PE0 на 65,7 %, процессор PE1 на 57,1 %.

#### Стратегия планирования «Группировка доминирующей последовательности»

Стратегия DSC «группировка доминирующей последовательно- сти» не устанавливает ограничений на число процессоров и базиру- ется на использовании понятия доминирующей последовательно- сти, которое по сути является критическим путем в частично спла- нированном графе задач. На каждом шаге планирования DSC стра- тегия проверяет, готов ли к планированию первый (верхний) узел критического пути. Если да, стратегия DSC назначает соответству- ющую задачу на процессор, обеспечивающий наиболее раннее вре- мя запуска. Минимальное время запуска может быть достигнуто перепланированием некоторых родительских задач на тот же про- цессор. С другой стороны, если верхний узел критического пути не

57

готов к планированию, стратегия DSC не выбирает его для плани- рования на текущем шаге. Вместо него стратегия выбирает верхний узел, предшествующий первому узлу критического пути. Узел на- значается на процессор, дающий минимальное время запуска при условии, что такой выбор процессора не задержит время запуска еще не спланированного верхнего узла критического пути. Отло- женное 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 приме- ре 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 ле- жат 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).

### PE 0

0

n1 5

### 5

Рис. 3.12. Шаг 1 стратегии DSC

**Шаг 2.** Следующей не спланированной задачей, лежащей на кри- тическом пути, является задача n 3. Она готова к планированию, по- скольку имеет одного предшественника  задачу *n* 1, которая уже назначена на процессор PE0. Поскольку один процессор уже задей- ствован задачей *n* 1, рассматриваем попытку введения второго про- цессора и исследуем два варианта I и II назначения узла *n* 3 на про- цессор PE0 и на новый процессор PE1 (рис. 3.13). В варианте I дли- на частичного плана равна 15. В варианте II длина плана равна 35 из-за длительного обмена данными между задачами. Следователь- но, предпочтительным является вариант I, его и выбираем для даль- нейшего рассмотрения.

58

* + 1. PE 0

0

n1

5

n3

10

0

PE0 PE1

15

n1

5

20

n3

10

35

Рис. 3.13. Шаг 2 стратегии DSC

**Шаг 3.** Следующей не спланированной задачей, лежащей на кри- тическом 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 сплани- рована. Значит, задача *n* 4 не готова к планированию. Для того, чтобы сделать задачу *n*4 готовой к планированию, для планирования выби- раем 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 IIназна- чения узла *n*2 на процессор PE0 и на новый процессор PE1 (рис. 3.14).

I) PE 0

0

n1 5

n2 20

n3 10

II)

0

26

PE0 PE1

n1 5

1

n3 10

n2 20

35

Рис. 3.14. Шаг 3 стратегии DSC

59

В варианте I задача *n* 2 вставляется между задачами *n* 1 и *n* 3, уже назначенными на процессор PE0. При этом длина получаемого ча- стичного плана равна 35. В варианте II длина частичного плана рав- на 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 тре- тьего процессора и исследуем три варианта I, II и III назначения уз- ла *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, пред- почтительным является вариант II, его и выбираем в качестве иско- мого асинхронного плана, реализуемого на двух процессорах.

1. PE0

0



n1 5

1

n3 10

n2 20

n4 1

8

II)

0

PE0 PE1

III)

0

PE0 PE1

PE2

35 34 35



n1 5

1

n3 10

n2 20

10

n4 8



n1 5

1

n3 10

n2 20

1

10

n4 8

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 уда- лось 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 за- гружает процессоры PE0 и PE1 суммарно на 122,8 %. Стратегия DSC предпочтительнее стратегии EZ и по этому показателю.

60

#### Планирование с использованием мобильности задач

Стратегия MD «управляемое мобильностью планирование» вы- полняет планирование задач в порядке возрастания их относитель- ной мобильности. Обоснование такого подхода базируется на сопо- ставлении 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 сравне- нию с задачей с высокой мобильностью. Более раннее планирова- ние задачи с низкой мобильностью повышает ее шансы быть эффек- тивно спланированной.

Абсолютная мобильность задачи есть разность времени наибо- лее позднего и времени наиболее раннего начала выполнения зада- чи. Время наиболее раннего запуска задачи на выполнение опреде- ляется стратегией ASAP, время наиболее позднего запуска опера- ции определяется стратегией ALAP. Обе стратегии должны быть адаптированы к графу задач. Относительная мобильность определя- ется 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 относитель- ная мобильность оставшихся задач пересчитывается с уч етом час- тично 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 выполне- ния 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 аб солютную и относи- тельную 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). Упорядо- чение задач (*n* 1, *n* 3, *n* 4, *n* 2) выполняется с учетом возрастания отно- сительной мобильности.

61

0



n 1

5

1

20

n 2

2 0

n 3

1 0

1

1 0

n 4 8

6

2 5

4 5

5 3

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 ша- га 2 не спланированные задачи *n* 2, *n* 4 имеют относительную мо- бильность 0, следовательно, на шаге 3 планируется операция *n* 2, на шаге 4  операция *n* 4. Результирующий план показан на рис. 3.17.

62

Для задачи *n* 2 не находится свободного слота времени на процессо- ре PE0, поэтому она назначается на процессор PE1. Напротив, для задачи *n* 4 имеется свободный слот времени на процессоре PE0, на который она и назначается.

PE 0 PE1

0



n1 5

1

n3 10

n2 20

n4 8

1

15

35

Рис. 3.17. План стратегии MD

63

1. **ПЛАНИРОВАНИЕ РЕШЕНИЯ ЗАДАЧ**

**В РАЗНОРОДНОЙ РАСПРЕДЕЛЕННОЙ СИСТЕМЕ**

Реальные распределенные системы являются типично разно- родными [57]. Разнородность системы обусловлена двумя главны- ми причинами:

* 1. разнородностью процессоров, находящихся в узлах, и раз- нородностью программного обеспечения, установленного на этих процессорах;
  2. разнородностью каналов передачи данных, соединяющих процессоры.

Разнородность процессоров и каналов передачи данных проявля- ется в вариациях значений их главных параметров в широких диа- пазонах. Так тактовые частоты процессоров и скорости в каналах передачи данных могут изменяться в разы.

Как следствие, предположение о константном времени решения за- дач и константном времени передачи данных от одной задачи к другой не совсем соответствует свойствам реальной разнородной системы.

В настоящем разделе мы рассматриваем задачу планирования вычис- лений 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 реаль- ных 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 «разум- ные» стратегии сокращения перебора вариантов, позволяющие най- ти за реальное время оптимальный или близкий к оптимальному план решения задач.

#### 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 произ- вольной топологии.

64

Вместо модели графа задач, рассмотренной ранее, воспользуемся моделью графа *GT* = (*VT*, *ET*) взаимодействия задач, где *VT* = {*t*1, , *tm*}  множество задач-вершин; *E T*  множество неориентированных ре- бер, помеченных временами передачи данных между взаимодейст- вующими задачами. Граф *G T* взаимодействия задач отличается от графа задач *G*, рассмотренного в предыдущей главе, тем, что в пер- вом обмен данными выполняется в двух направлениях, в то время как во втором обмен данными выполняется в одном направлении.

Топологию сети процессоров представим неориентированным графом *GL* = (*P*, *L*), в котором *P* = {*p* 1, , *pn*}  множество вершин- процессоров; *L*  множество ребер, представляющих каналы пере- дачи данных. Граф *G L* описывается матрицей *L*, строки и столбцы которой соответствуют процессорам:

*L*11

*L*  *Lq*1 *Ln*1

...

...

...

*L*1 *p*

...

*Lqp*

...

*Lnp*

...

...

...

*L*1*n Lqn* .

*Lnn*

Элемент *Lqp* равен 1, если процессоры *q* и *p* соединены каналом передачи данных, и равен 0, если процессоры каналом не соедине- ны. Задача *t i* из множества *V T* может быть решена на любом из *n* процессоров. Матрица *A* размерностью *m*  *n* определяет время ре- шения каждой из *m* задач на каждом из *n* процессоров:

*A*11

*A*  *Ai*1 *Am*1

...

...

...

*A*1 *p*

...

*Aip*

...

*Amp*

...

...

...

*A*1*n Ain* .

*Amn*

Элемент *Aip* матрицы есть время решения задачи *ti* на процессоре *p*.

С задачами *t i* и *t j*, решаемыми на различных процессорах, ассоцииру-

65

ются дополнительные временные затраты на обмен данными, если за- дачи взаимно информационно зависимы. Матрица *C* описывает вре- менные затраты на обмен данными между парами задач:

*C*11

*C*  *Ci*1 *Cm*1

...

...

...

*C*1 *j*

...

*Cij*

...

*Cmj*

...

...

...

*C*1*m Cim* .

*Cmm*

Элемент *Cij* матрицы есть время передачи данных в период взаи- модействия двух задач *t i* и *t j*, назначенных и решаемых на различ- ных процессорах. Если задачи назначаются на один процессор, вре- мя *Cij* зануляется.

Загрузка одного процессора определяется суммой времен решения всех задач, назначенных на данный процессор, и времен обмена дан- ных с другими задачами, назначенными на другие процессоры. Вре- мя, затрачиваемое наиболее загруженным процессором, рассматри- вается как время параллельного распределенного решения всех задач. Оптимальное назначение *m* задач на *n* процессоров ставит целью ми- нимизировать время работы наиболее загруженного процессора.

*Пример* 4.1. Пусть дано множество из пяти задач *VT* = {*t*1, *t*2, *t*3, *t*4, *t*5}, связи между которыми описываются графом взаимодействия задач, показанным на рис. 4.1.

8

t1

t2

6

7

t4

t3

5

4

t5

Рис. 4.1. Граф взаимодействия задач

66

Ребра графа помечены числами, интерпретируемыми как дли- тельности интервалов времени, на которых задачи попарно обмени- ваются данными. Ребра не являются ориентированными, значит, дан- ные 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 пара- метры. Коммуникации между задачами представляются матрицей

*t*5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *t*1 | *t*2 | *t*3 | *t*4 |
| *t*1 | 0 | 8 |  |  |
| *t*2 | 8 | 0 | 6 |  |
| *t*3 |  | 6 | 0 | 5 |
| *t*4 |  |  | 5 | 0 |
| *t*5 | 7 |  | 4 |  |

7

*C*   .

4



0

Символ «» означает отсутствие коммуникаций между парами задач. Нули на главной диагонали матрицы означают нулевые за- траты 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 ка- налы имеют одинаковую пропускную способность, следовательно, время передачи данных не зависит от канала, по которому данные передаются.

p1

p2 p3

Рис. 4.2. Топология сети

67

Сеть процессоров можно также описать единичной матрицей

*p*1 *p*2 *p*3

*L*  *p*1 *p*2

*p*3

1 1 1 .

1 1 1

1 1 1

Поскольку распределенная система является неоднородной, вре- мя решения каждой задачи меняется при переходе от одного про- цессора к другому. Времена решения задач описываются матрицей *A* размерности 5  3:

*p*1 *p*2 *p*3

*t*1

*A*  *t*2 *t*3

*t*4 *t*5

15 11 9

14 12 8 .

16 13 6

5 4 3

10 9 7

В среднем, время решения задачи на процессоре *p*3 меньше, чем на процессоре *p*1.

#### Назначение задач на узлы

Назначение задач на процессоры определяется матрицей *X* раз- мерности *m*  *n*:

*X*11

*X*  *Xi*1

*Xm*1

...

...

...

*X*1 *p*

...

*Xip*

...

*Xmp*

...

...

...

*X*1*n*

*Xin* .

*Xmn*

68

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 со- держит ровно одну единицу. Столбец матрицы может содержать произвольное число единиц, но не превышающее число задач. Мно-

жество всех значений матрицы *X* представляет пр остранство поис- ка, в котором ищется оптимальное решение.

#### Оценка общего времени решения задач

Общее время решения всех задач определяется временем функ- ционирования наиболее загруженного процессора. Загрузка процес- сора *p*  *P* определяется следующим выражением

*m n m m*

 

*load X* ( *p*)  ( *Aip*  *Xip* )     *Cij Xip X jq Lpq* . (4.1)

*i*1

*q*1, *i*1 *j*1

*q* *p*

Первая часть суммы (4.1) есть полное время решения всех задач, назначенных на процессор *p*. Задача *i* назначена на процессор *p*, ес- ли *Xip* = 1, при этом время решения задачи определяется значением *Aip*. Вторая часть есть дополнительное время, затрачиваемое про- цессором *p* на обмен данными с другими процессорами. Единичные значения элементов *Xip* и *Xjq* указывают на то, что задача *i* назначена на процессор *p*, а задача *j* назначена на процессор *q*. Единичное зна- чение элемента *L pq* указывает на то, что процессоры *p* и *q* соедине- ны каналом передачи данных. Для того чтобы найти процессор

с максимальной загрузкой, определяющий общее время *Time X* ре- шения всех задач, необходимо рассчитать загрузку каждого из *n* процессоров для определенной матрицы *X*

*TimeX* .max *load X* ( *p*)

*p**P*

(4.2)

Оптимальное назначение задач на процессоры есть то, которое дает минимум загрузки *MinTime* по всем возможным значениям мат- рицы *X* наиболее нагруженного процессора *p*  *P*:

min*Time* .min *TimeX* 

*X* 

(4.3)

69

Задача (4.3) относится к классу дискретных задач комбинатор- ной оптимизации.

#### Алгоритм оптимального назначения задач на процессоры

Алгоритм А\* [5] реализует стратегию поиска первого наилучше- го при решении широкого круга проблем, включая проблемы ис- кусственного интеллекта. Для построения алгоритма используются древовидные графы, называемые деревьями поиска. При решении проблемы назначения задач на процессоры множество вершин де- рева поиска разбивается на ярусы. Корень дерева, находящийся на нулевом ярусе, соответствует нулевому назначению зада ч на про- цессоры. Листья дерева поиска (терминальные вершины), находя- щиеся на *m*-м ярусе, соответствуют полным назначениям задач на процессоры. Нетерминальные вершины соответствуют частичным назначениям задач на процессоры. Каждая нетерминальная верши- на *di*, находящаяся на ярусе *i* дерева поиска, является родительской для *n* дочерних вершин, получаемых в результате назначения зада- чи *ti* на процессоры 1, , *n* соответственно. Как следствие, верши- на *di* имеет *n* исходящих дуг.

С каждой нетерминальной вершиной *d i* дерева поиска ассоции- руется функция стоимости min*Time*(*d i*), описывающая оценку ми- нимального времени решения всех задач при их частичном назна- чении на процессоры. Если вершина *d* i является терминальной, то *i* = *m* и функция min*Time*(*d m*) описывает точное значение *Time X* вре- мени решения всех задач при их полном назначении на процессоры, вычисляемое по формуле (4.2). При этом значение матрицы *X* опре- деляется путем от корня дерева поиска до терминальной вершины *dm*.

Для нетерминальной вершины *d* i оценка min*Time*(*d* i) минималь- ного времени решения всех задач записывается в виде суммы:

min*Time*(*di* )

*g*(*di* )  *b*(*di* ),

(4.4)

где *g*(*di*)  время, ассоциируемое с путем от корня дерева до верши- ны *d*i и вычисляемое по формулам (4.1)(4.2) с учетом только задач 1, , *i * 1, которые уже назначены на процессоры;

70

*b*(*di*)  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*, , *m*.

Если слагаемое g(*d i*) точно рассчитывается по уже пройденному на дереве поиска пути, то слагаемое *b*(*di*) требует разработки метода оценки нижней границы времени для еще не пройденного пути. Бо- лее точная оценка нижней границы сокращает комбинаторный пе- ребор вариантов на дереве поиска.

Для расчета нижней границы времени используется следующий метод. С целью оценки загрузки процессора *p* рассматривается два подмножества задач. Подмножество *Tp* включает задачи, уже назна- ченные 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*  *U p* существует два альтернативных варианта: либо она будет назначена на процес- сор *p*, и тогда время обмена данными с задачами из *T p* зануляется; либо она назначается на другой процессор *q*, соединенный с про- цессором *p* каналом передачи данных, и тогда время передачи не зануляется. Как следствие, с каждым назначением задачи *t* j ассоци- ируются 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

cost(*tj*) = min(*Ajp*, *Hpj*) (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*(*di* )



*t j**U p*

cos *t*(*t j* ).

(4.6)

Каждой вершине дерева поиска поставим в соответствие две мет- ки. Первой является строка символов *s* = *s* 1*si**sn*, где *s i* есть циф- ра, 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 min*Time*, которое оценивается уравнениями (4.1), (4.4)(4.6).

71

Каждый уровень дерева поиска соответствует одной задаче. Ну- мерация уровней соответствует нумерации задач: *i*-й задаче соответ- ствует *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 сим- вола «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 ре- зультате генерируются *n* дочерних вершин. Для каждой из дочерних вершин оценивается значение времени min*Time*.

Алгоритм A\* использует упорядоченный список не раскрытых вершин дерева поиска, называемый OPEN. Включаемые в список но- вые вершины упорядочиваются по не убыванию (возрастанию) значе- ния функции стоимости min*Time*. 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 наи- меньшей стоимостью, к ней применяется оператор раскрытия, в ре- зультате генерируются дочерние вершины. Первая вершина удаляется из списка, а новые вершины добавляются в него, при этом список пе- реупорядочивается. Такой порядок раскрытия вершин гарантирует нахождение оптимального решения в момент появления в списке вер- шины со строковой меткой, в которой отсутствует символ «X».

*Пример* 4.2. Применим алгоритм A\* к графу взаимодействия за- дач и графу топологии сети, которые специфицированы в примере 4.1. Поскольку первый граф состоит из пяти задач, вершины дерева по- иска метятся строками из пяти символов. Так ка к граф топологии сети построен на трех процессорах, нетерминальные вершины дере- ва поиска имеют каждая три исходящие дуги. Соответственно в стро- ках, 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 прямоугольни- ками, в которых строка символов показывает состояние назначения

72

задач на процессоры, число описывает значение функции стоимо- сти min*Time*. Число, стоящее рядом с прямоугольником, есть поряд- ковый номер раскрытия вершины. Отсутствие такого числа указы- вает на то, что вершина осталась не раскрытой. Одна терминальная вершина помечена как «цель». Эта вершина представляет опти- мальное решение задачи. Более подробно процесс поиска решения описывается следующими шагами.

**цель**

XXXXX 1

(0)

3

1XXXX (30)

2XXXX (26)

2

3XXXX (24)

8 5

21XXX 22XXX 23XXX (28) (36) (26)

7 4

31XXX 32XXX 33XXX (28) (26) (30)

10

211XX 212XX 213XX 231XX 232XX 233XX (47) (51) (28) (31) (51) (29)

9 6

311XX 312XX 313XX 321XX 322XX 323XX (47) (28) (39) (26) (41) (39)

2131X 2132X 2133X

(38) (28) (28)

11

3121X 3122X 3123X 3211X 3212X 3213X (38) (28) (28) (31) (35) (31)

31221 31222 31223

(49) (39) (28)

Рис. 4.3. Результирующее дерево поиска

**Шаг 0.** OPEN = {«XXXXX»-0}.

**Шаг 1.** Выбираем вершину «XXXXX». Заменяем «X» на 1 в пер- вой позиции. Используем формулу (4.1) для расчета загрузки про- цессоров. Значение *g* рассчитываем по формуле (4.4). Процессоры *p*2, *p*3 вообще не загружены, поэтому *load*(*p* 2) = *load*(*p*3) = 0. На про- цессор p1 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 рас- считываются значения min*Time* для вершин «2XXXX» и «3XXXX», они равны 26 и 24 соответственно. Исключая из списка OPEN вер-

73

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 но- вые вершины «31XXX», «32XXX», «33XXX» со значениями функ- ции 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 но- вые 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 но- вые 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 но- вые 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 но- вые вершины «3211X», «3212X», «3213X» с весами 31, 35, 31. Спи- сок OPEN = {«31XXX»  28, «21XXX»  28, «233XX»  29,

«1XXXX»  30, «33XXX»  30}.

**Шаг 7.** Выбираем вершину «31XXX» и раскрываем ее в три но- вые 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 но- вые 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}.

74

**Шаг 10.** Выбираем вершину «213XX» и раскрываем ее в три но- вые вершины «2131X», «2132X», «2133X» с весами 38, 28, 28. Спи- сок OPEN = {«3122X»  28, «3123X»  28, «2132X»  28, «2133X»  28, «233XX»  29}.

**Шаг 11.** Выбираем вершину «3122X». Раскрываем ее в три но- вые 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*  .

|  |  |  |  |
| --- | --- | --- | --- |
|  | *p*1 | *p*2 | *p*3 |
| *t*1 | 0 | 0 | 1 |
| *t*2 | 1 | 0 | 0 |
| *t*3 | 0 | 1 | 0 |
| *t*4 | 0 | 1 | 0 |
| *t*5 | 0 | 0 | 1 |

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

75

#### ЯЗЫКИ И ИНСТРУМЕНТЫ ПРОГРАММИРОВАНИЯ РАСПРЕДЕЛЕННОЙ И ПАРАЛЛЕЛЬНОЙ

#### ОБРАБОТКИ ДАННЫХ

Для распределенного параллельного программирования суще- ствует ряд инструментов, включающих:

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

#### Многопоточные приложения

Операционная система (ОС) является *многозадачной*, если она способна одновременно выполнять несколько программ. Одна опе- рационная система способна реализовать многозадачность в двух вариантах: путем разделения между задачами процессорного вре- мени одного процессора; путем назначения задач на разные процес- соры. Уже в первом варианте многозадачности, ОС настолько быстро переключает вычислительные ресурсы между задачами, что созда- ется впечатление их одновременного выполнения.

Многозадачность может быть *кооперативной* и *вытесняющей*.

В случае кооперативной многозадачности ОС не занимается решени- ем проблемы распределения процессорного времени между задача- ми. Распределяют его сами задачи. Активная задача самостоятельно решает, отдавать ли процессор другой задаче или не отдавать.

В случае вытесняющей многозадачности распределением про- цессорного 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 опера- ционных системах корпорации Microsoft, начиная с Windows 95, реализована вытесняющая многозадачность.

76

*Процессом*называется экземпляр программного приложения, за- груженного в оперативную память для выполнения. *Многопоточ- ность* ОС [8] означает то, что процессы могут разделяться на части  потоки, самостоятельно претендующие на процессорное время. Ис- пользование потоков обеспечивает одновременное выполнение не- скольких ветвей программы. При выполнении процесс создает не менее одного потока (*thread*). Первый поток называется *главным* потоком приложения, создаваемым ОС автоматически. Он по- рождает 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 выпол- няются не параллельно, а последовательно по очереди. Система вы- деляет потокам кванты времени по принципу карусели (рис. 5.1).

Поток

Поток

Поток

Поток

Поток

Поток

Процессор

Поток

Поток

Рис. 5.1. Карусельная модель выполнения потоков

В зависимости от ситуации потоки могут находиться в трех со- стояниях: *активном*, *готовности/ожидания*, *блокировки*. Перевод потока из состояния блокировки в состояние готовности выполня- ется по событиям.

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

77

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 *пользова- тельского интерфейса*. Поток пользовательского интерфейса мо- жет иметь окна и свой цикл выборки сообщений.

В Win32 API поток создается функцией *CreateThread*. Поток за- вершает выполнение при выходе из функции потока посредством оператора *return*. Функция *WaitForSingleObject* ждет, пока конкрет- ный поток закончит свою работу. Выполнение потока можно при- остановить вызовом функции *SuspendThread*. Поток возобновляется вызовом функции *ResumeThread*.

*Приоритеты потоков*. Каждому потоку присваивается приоритет от 0 до 31. Windows поддерживает следующие семь классов приорите- та потоков: CRITICAL, HIGHEST, ABOVE\_NORMAL, NORMAL,

BELOW\_NORMAL, LOWEST, IDLE. Для установки и получения при- оритета служат функции *SetThreadPriority* и *GetThreadPriority*.

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

*Критическая секция* синхронизирует потоки посредством обес- печения монопольного захвата ресурса и атомарного доступа одним потоком в случае, когда доступ к общим ресурсам (файлу, глобаль- ной переменной и др.) выполняется разными потоками одновре- менно. Критическая секция представляет собой участок кода, поз- воляющий сделать так, чтобы одновременно только один поток по- лучал доступ к ресурсу.

*Семафор* есть объект, ограничивающий количество потоков, ко- торые могут войти в заданный участок кода. Семафоры использу- ются при передаче данных через разделяемую память. Они реали- зуются посредством счетчика, значение которого уменьшается, ко- гда семафор выделяется задаче, и увеличивается, когда семафор освобождается задачей. Семафор позволяет параллельно работаю- щим потокам обращаться к общему ресурсу избегая конфликтов.

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

78

зователей, мьютекс гарантируют потокам взаимоисключающий без- конфликтный 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.

#### Интерфейс MPI

MPI расшифровывается как Message Passing Interface  интер- фейс передачи сообщений [9]. MPI является международным стан- дартом интерфейса обмена данными в параллельном программиро- вании. 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, С и С++. В первую очередь MPI ори- ентирован на системы с распределенной памятью, в которых затра- ты 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 требуют- ся промежуточные буферы для передаваемых данных; обеспечива- ется 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 пере- дача коротких сообщений с меньшей надежностью;
3. Широковещательный режим, когда процесс посылает сооб- щения всем другим процессам.

MPI поддерживает операции процесс-процесс и коллективные операции, такие как разборка-сборка и редукция сообщений. Он обес- печивает создание приложений в модели «одна программа  множе- ственные данные», позволяющей исполнять один программный код в разных процессах, каждый со своими данными. При этом програм- мному коду доступна информация о том, в каком именно процессе он исполняется. Это позволяет адаптировать исполнение програм- мы к конкретному процессу и учесть особенности этого процесса,

79

что позволяет рассматривать MPI как реализацию модели «множе-

ственная программа  множественные данные».

В MPI все операции передачи сообщений и синхронизации лока- лизуются внутри *коммуникатора*. С коммуникатором связывается группа процессов, а все процессы группы взаимодействуют через коммуникатор. В частности, все операции процесс-процесс и кол- лективные операции выполняются процессами, входящими в груп- пу и взаимодействующими через коммуникатор. Процессы внутри группы нумеруются целыми числами в диапазоне 0...groupsize-1.

Коммуникаторы идентифицируются именами. Стандартное имя ком- муникатора  MPI\_COMM\_WORLD.

MPI-программа представляет собой набор независимых парал- лельно выполняемых процессов, каждый из которых выполняет об- щий программный код, специфицированный к конкретному процес- су. Если параллельное приложение строится на параллелизме дан- ных, 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 парал- лельном выполнении процессы взаимодействуют друг с другом по- средством 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 вспо- могательные 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 алгебраи- ческих уравнений (СЛАУ) блочно-параллельным методом Гаусса дан на рис. 5.2.

80

#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

81

Программа представлена функцией *main* языка C. Макро-пере- менная *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 струк- туру коммуникатора. Граф топологии коммуникатора определяется отдельным кодом, который здесь не приводится. Функция *MPI\_Init* инициализирует библиотеку MPI. Функция *MPI\_Comm\_size* возвраща- ет 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 стро- ки 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.

#### Открытый стандарт OpenMP

OpenMP реализует параллельные вычисления на машинах с не- сколькими процессорами с помощью многопоточности, которая строится автоматически из последовательного кода по директивам препроцессора, называемым *pragma*. Количество создаваемых по- токов, регулируемое посредством директив, может превышать ко- личество доступных процессоров. OpenMP ориентирован на систе- мы 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 по- токами (директивы*do*/*for*и*section*); конструкции для управления работой с данными (выражения *shared* и *private* для определения класса памяти переменных); конструкции для синхронизации пото- ков (директивы *critical*, *atomic* и *barrier*) и др.

82

OpenMP можно рассматривать как высокоуровневую надстройку над многопоточностью. Программная модель OpenMP представляет собой *fork*-*join* параллелизм, в котором главный поток по необходи- мости порождает вспомогательные потоки при вхождении програм- мы в параллельные области. OpenMP позволяет быстро распаралле- лить программы с циклами, выполняющими большой объем вычис- лений. Одна OpenMP-программа выполняется параллельно на много- процессорной систем и выполняется последовательно на однопро- цессорной 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 тре- мя входными аргументами возвращает их среднее значение. Функ- ция *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*, *y* сле- дует директива #*pragma omp parallel*. Эта важнейшая директива *parallel* указывает на необходимость автоматического распаралле- ливания 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 по- следующему циклу *for*. Благодаря присутствию *private*(*i*), для каж- дого вспомогательного потока, создаваемого с целью параллельной реализации различных итераций цикла, вводится своя копия пере- менной *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 пере- менных *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, испол- няемый только ведущим потоком.

83

#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

#### Технологический стандарт CORBA

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

84

Суть CORBA состоит в следующем: каждый компонент распре- делённого 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 Defi- nition 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 при- шли 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 компо- нент в распределенном CORBA-приложении имеет одновременно свойства и клиента, и сервера  и, таким образом, достигает макси- мальной самостоятельности и независимости в своей реализации от других компонентов приложения. Брокеры могут быть самостоятель- ными приложениями или могут встраиваться в другие приложения.

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

#### Модель компонентных объектов COM

После компиляции приложение состоит из одного монолитного *двоичного* файла, который в соответствии с традиционными техно- логиями остается неизменным пока не будет скомпилирована новая версия. *Модель компонентных объектов* Microsoft COM (*Compo- nent 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 об- щего метода создания компонентов и построения из них приложе- ний. Преимущества компонентной модели:

1. Способность приложения эволюционировать с течением вре- мени путем замены устаревших версий компонентов более совре- менными версиями;

85

1. Адаптация приложения к различным пользователям путем использования компонентов, наиболее адекватных потребностям пользователя;
2. Возможность быстрой сборки приложения из компонентов библиотеки;
3. Повышение эффективности разработки распределенных кли- ент-серверных приложений.

Общие требования к компонентам:

1. Подключение компонентов во время выполнения приложения требует применения динамической компоновки;
2. Применение принципа инкапсуляции к компонентам; компо- ненты должны разбиваться на две основные части: интерфейс с внеш- ним миром и внутреннюю реализацию;
3. Обладание способностью реализации внутри одного процес- са, в разных процессах и на разных машинах, должно обеспечивать- ся перемещение компонентов в компьютерной сети;
4. Поддерживание клиент-серверной архитектуры приложений, в которой сервер реализуется компонентом, а клиент общается с сер- вером посредством соответствующих интерфейсов;
5. Разработка отдельных компонентов и целых многокомпонент- ных приложений должна обеспечиваться на разных языках програм- мирования;
6. Поддерживание библиотечного сервиса управления компо- нентами.

#### Реализация компонентов

Основные принципы реализации *компонентов* COM опишем с ис- пользованием языка С++ объектно-ориентированного программи- рования. Структура компонента, построенная с использованием *по- лиморфизма*, представлена на рис. 5.4. Класс объектов *Component* наследует в открытом режиме *чисто абстрактные базовые классы Interface1*  *InterfaceN*, описывающие *интерфейсы*, реализуемые объ- ектами. Основное назначение класса *Component*  реализация интер- фейсов, прежде всего стандартного для COM интерфейса *IUnknown*, который включает *чисто виртуальные функции QueryInterface*, *AddRef* и *Release*. Другие интерфейсы также построены на базе чисто вир- туальных функций. Счетчик числа ссылок *m\_cRef* используется для посчета указателей на интерфейсы, установленных в процессе совме-

86

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 С++

Архитектура многокомпонентного COM приложения определяет систему интерфейсов между компонентами таким образом, что от- дельные компоненты могут модернизироваться или заменяться дру- гими компонентами без изменения интерфейсов. Для идентифика- ции интерфейсов и компонентов используются *глобально уникаль- ные идентификаторы* GUID, позволяющие находить и выполнять доступ к компонентам и интерфейса через *реестр* операционной системы. Идентификаторы GUIDмогут быть сгенерированы на ком- пьютере программиста программой *guidgen*.*exe*. Для создания и ди- намической загрузки компонентов используется специальный ком- понент, называемый *фабрикой класса*. Фабрика класса реализует специальный интерфейс *IClassFactory*, удовлетворяющий всем тре- бованиям интерфейсов COM.

Клиент и сервер-компонент могут быть реализованы тремя спо- собами. Реализация в одном процессе делает доступным одно ад- ресное пространство и клиенту и серверу. Все интерфейсы, реали- зуемые компонентами, доступны клиенту посредством указателей. Реализация в разных процессах на одном компьютере делает необ- ходимым использование технологии передачи данных, называемой *маршалингом*. Реализация в разных процессах на разных компьюте- рах делает необходимым использование сетевого программного обес- печения в дополнение к маршалингу.

87

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

#### Определение интерфейсов

Интерфейс СОМ представляет собой чисто абстрактный базовый класс, определяющий набор функций-методов, реализуемых ком- понентами и используемых клиентами для взаимодействия с ком- понентами. Для клиента компонент представляет собой набор интер- фейсов, реализуемых компонентами. По-существу, интерфейс есть структура данных в памяти, содержащая массив указателей на функ- ции-методы интерфейса. Поскольку каждый компонент СОМ может поддерживать сколь угодно много интерфейсов, для реализации ком- понента с несколькими интерфейсами используется множественное наследование чисто абстрактных базовых классов. Одно из самых больших преимуществ компонентной модели  возможность по- вторного использования интерфейсной архитектуры приложения.

В случае сохранения интерфейсной архитектуры изменения в компо- ненте не вызывают изменений в клиенте и наоборот. При изменении интерфейсной архитектуры не следует изменять существующие ин- терфейсы, 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 ре- жиме стандартный интерфейс *IUnknown*, подключаемый из заголо- вочного файла *unknwn*.*h*. Объявление интерфейса *IUnknown* пред- ставлено на рис. 5.5.

interface Interface1: IUnknown {

virtual *type1* stdcall *Method1*(*type11*,, *type1k1*) = 0;

virtual *typer* stdcall *Methodr*(*typer1*,, *typerkr*) = 0;

};

Рис. 5.4. Объявление интерфейса

88

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 *Re- lease* для декрементации числа ссылок на интерфейсы с целью оп- ределения времени жизни компонента. Поскольку все интерфейсы СОМ наследуют *IUnknown*, в каждом интерфейсе есть функции *QueryInterface*, *AddRef* и *Release*  первые три функции в таблице виртуальных функций класса.

#### Библиотека COM

Все клиенты и компоненты СОМ выполняют много типовых опе- раций, которые стандартизированы в библиотеке функций СОМ, до- ступной посредством заголовочного файла *objbase*.*h.* Для инициали- зации библиотеки процесс клиента вызывает функцию *CoInitialize*, для завершения работы с библиотекой процесс вызывает функ- цию *CoUninitialize*. Создание компонентов выполняется функцией *CoCreateInstance*, которая, получив от клиента глобальный иденти- фикатор компонента CLSID, создает экземпляр компонента и воз- вращает адрес интерфейса, запрашиваемого посредством иденти- фикатора IID. Функция *CoCreateInstance* создает и использует фаб- рику класса для экспортирования запрашиваемого компонента из библиотеки *dll*. Фабрика класса  это компонент, задачей которого является создание других компонентов. Экспортирование компо- нента и управление подключением библиотеки выполняется по- средством функций *DllGetClassObject* и *DllCanUnloadNow*, которые не принадлежат библиотеке СОМ, но пишутся разработчиком при- ложения и подключаются посредством .*def* файла. Полные имена файлов компонентов, индексированные CLSID, помещаются в ре- естр операционной системы.

89

#### Сервер в процессе, локальный и удаленный сервер

В ряде случаев предпочтительнее реализовать компонент в за- 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* на- зывается сервером *вне процесса*. Компонент в .*exe* называется *локаль- ным 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 проце- дуры* (*Local Procedure Call*  *LPC*). Средством коммуникации меж- ду различными процессами, работающими на разных машинах, яв- ляется *удаленный вызов процедуры* (*Remote Procedure Cal*l  *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, марша- линг копирует данные с учетом различий адресов, если на разных машинах, данные преобразуются в стандартный формат, учитываю- щий межмашинные различия. В процессе клиента компонент пред- ставлен .*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*).

90

#### СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Лорин, Г. Распределенные вычислительные системы / Г. Ло- рин.  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.

1. Прихожий, А. А. Распараллеливание и планирование вычис- лительных и информационных процессов / А. А. Прихожий // Док- лады БГУИР.  2003.  № 4.  С. 104114.
2. Прихожий, А. А. Модель и алгоритм оптимизации назначе- ния объектов на узлы распределенной информационно-вычисли- тельной системы / А. А. Прихожий // Информатика.  2010.  № 4.
3. Kafil, М. Optimal Task Assignment in Heterogeneous Distributed Computing Systems / M. Kafil, I. Ahmad // IEEE Concurrency, July- September.  1998.  pp. 4251.
4. Prihozhy, А. Net Scheduling in High-Level Synthesis / A. Prihozhy // IEEE Design & Test of Computers".  Spring, 1996.  pp. 2635.
5. Prihozhy, А. 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.
6. Эндрюс, Г. Р. Основы многопоточного, параллельного и рас- пределенного программирования / Г. Р. Эндрюс // М. :  Вильямс, 2003.
7. Шпаковский, Г. И. Программирование для многопроцессор- ных систем в стандарте MPI / Г. И. Шпаковский, Н. В. Серикова // Минск : БГУ, 2002.  323 c.
8. OpenMP [Электронный ресурс]:  Режим доступа: [http://openmp.org/mp-documents/.](http://openmp.org/mp-documents/)  Дата доступа: 07.09.2014.
9. Эммерих, В. Конструирование распределенных объектов /

В. Эммерих.  М. : МИР, 2002.

1. Роджерсон, Д. Основы COM / Д. Роджерсон.  М. : Издат.- торговый дом «Русская редакция».  2000.
2. Мюллер, Д. Технология COM+ / Д. Мюллер.  СПб. : Питер Бук, 2002.

91

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.

92

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.