

Параллельная обработка данных

**Пособие по успокоению нервов перед экзаменом.
Ни в коем случае не является официальным пособием**

**Издание 6-е, исправленное и дополненное
Дополнения и исправления приветствуются**

ВНИМАНИЕ!

**МАТЕРИАЛЫ СОБРАНЫ ПО ВСЕМУ ИНТЕРНЕТУ. К СОЖАЛЕНИЮ, ЕСТЬ НЕ ВСЕ.
НАСКОЛЬКО СООТВЕТСТВУЕТ ТОЧКЕ ЗРЕНИЯ ЛЕКТОРА – НЕ ЗНАЮ.
ИСПОЛЬЗОВАТЬ НА СВОЙ СТРАХ И РИСК, ПРЕДВАРИТЕЛЬНО ПОМОЛИВШИСЬ...**

А. Ельцин, составитель и редактор.

Москва, 2007

Основы информатики.	4
Информация. Меры Хартли, Шеннона.	4
Знания и ЭВМ.	5
Эволюционная классификация вычислительных систем на примере развития отечественной техники.	10
Критерий, положенный в основу эволюционной классификации ЭВМ.	10
Основоположники отечественной вычислительной техники.	14
Вычислители фон-Нейманновской архитектуры. Конвейерная обработка данных и команд. Архитектура памяти.	21
Принципы фон-Нейманновской архитектуры ЭВМ.	22
Конвейерная обработка данных.	22
Зацепление конвейеров.	25
Векторно-конвейерные вычислители.	27
CISC и RISC архитектуры ЭВМ.	28
Внеочередное и спекулятивное выполнения команд.	29
Механизмы предсказания переходов.	30
Управление виртуальной памятью.	32
Полностью ассоциативная кэш-память.	46
Кэш-память с прямым отображением.	48
Частично ассоциативная кэш-память.	49
Дисциплина обновления кэш-памяти.	51
Стратегии записи в кэш-память.	52
Расслоение памяти.	54
Принципы VLIW архитектуры.	56
Суперскалярные и мультитредовые архитектуры микропроцессоров.	57
Стандарт IA-64.	61
Оптимизация программ под архитектуру микропроцессора.	63
Архитектура параллельных вычислительных систем	65
Архитектура параллельных вычислительных систем	65
Метакомпьютинг.	69
Кластерные архитектуры.	72
Симметричные мультипроцессорные системы.	73
Матричные мультипроцессорные системы.	75
Классификации вычислителей по Флинну.	76
Масштабируемость мультипроцессорных вычислителей.	79
Управление памятью в мультипроцессорных системах.	79
Когерентность данных.	84
Топологии мультипроцессорных коммутационных сетей.	85
Типы внутренних связей.	85
Статические и динамические коммутаторы.	87
Параметры статических коммутационных сетей.	89
Топологии линейки, решетки, пирамиды.	91
Топология гиперкуба.	93
Согласование сеточных топологий со структурой гиперкуба.	95
Перекрестный коммутатор.	96
Многокаскадные коммутационные сети.	99
Производительность вычислительных систем.	101
Пиковая производительность.	101
Методы оценки производительности.	102

Закон Амдала. _____	110
Нетрадиционные архитектуры: потоковые и нейронные вычислители. _____	111
Принципы потоковой обработки информации. _____	111
Схемы потоковых вычислителей. _____	112
Нейронные сети. _____	114
Области применения нейронных сетей. _____	117
Параллельное программирование _____	120
Модели программирования для систем с разделяемой, распределенной памятью. _____	120
Разделение последовательных программ на параллельные нити. _____	124
Ограничения на распараллеливание циклов. _____	128
Синхронизация параллельных процессов. Барьеры. _____	129
Критические секции. Двоичные и общие семафоры. _____	130
Упорядоченные секции. Распараллелить цикл, используя упорядоченные секции и семафоры: _____	132
Системы передачи сообщений. Фортран-GNS. _____	134
Статический и динамический способы образования параллельных процессов. _____	134
Требования к системам программирования методом передачи сообщений. _____	134
Система программирования MPI. _____	136
Средства описания и создания процессов в языке Фортран-GNS. _____	138
Средства передачи и приема сообщений в языке Фортран-GNS. _____	143
Протоколы передачи и приема сообщений в языке Фортран-GNS. _____	144
Идентификация абонентов при передаче сообщений в языке Фортран-GNS. _____	148
Адаптация последовательных программ к параллельным архитектурам.	
Векторизация и распараллеливание циклов. Методы координат, гиперплоскостей. _____	149
Автоматическое распараллеливание последовательных программ. _____	149
Семантика циклов, выполняемых параллельно на ОКМД системах. _____	153
Алгоритмы преобразования программ методом координат. _____	155
Схема преобразования программ методом гиперплоскостей. _____	162
Метод параллелепипедов. _____	163
Оценить возможность параллельного выполнения цикла: _____	164
Языки параллельного программирования _____	164
Стандарты OpenMP. _____	164
Язык Фортран-DVM. _____	169
Язык Sisal. _____	171
Система программирования Норма. _____	173
Особенности выполнения арифметических выражения на ЭВМ. Распараллеливание вычислительных алгоритмов. Метод распараллеливания алгоритма общей рекурсии 1-го порядка. _____	174
Распараллеливание алгоритмов сложения методом редукции _____	174
Метод распараллеливания алгоритма общей рекурсии 1-го порядка. _____	176
Представление машинных чисел. _____	177
Арифметика машинных чисел. _____	182
Погрешности при вычислениях чисел на параллельных системах. Оценить полную ошибку суммирования положительных чисел. _____	189
Точность плавающей арифметики. Машинный эпсилон. _____	190
Перечислить алгоритмы оптимизации объектных программ, которые могут повлиять на точность вычислений. _____	191

Основы информатики.

Информация. Меры Хартли, Шеннона.

Термин "информация" происходит от латинского слова "informatio", что означает сведения, разъяснения, изложение. Несмотря на широкое распространение этого термина, понятие информации является одним из самых дискуссионных в науке. В настоящее время наука пытается найти общие свойства и закономерности, присущие многогранному понятию информация, но пока это понятие во многом остается интуитивным и получает различные смысловые наполнения в различных отраслях человеческой деятельности:

в обиходе информацией называют любые данные или сведения, которые кого-либо интересуют. Например, сообщение о каких-либо событиях, о чьей-либо деятельности и т.п.

"Информировать" в этом смысле означает "сообщить нечто, неизвестное раньше";

в технике под информацией понимают сообщения, передаваемые в форме знаков или сигналов;

в кибернетике под информацией понимает ту часть знаний, которая используется для ориентирования, активного действия, управления, т.е. в целях сохранения, совершенствования, развития системы (Н. Винер).

Клод Шеннон, американский учёный, заложивший основы теории информации — науки, изучающей процессы, связанные с передачей, приёмом, преобразованием и хранением информации, — рассматривает информацию как снятую неопределенность наших знаний о чем-то.

Приведем еще несколько определений:

Информация — это сведения об объектах и явлениях окружающей среды, их параметрах, свойствах и состоянии, которые уменьшают имеющуюся о них степень неопределенности, неполноты знаний (Н.В. Макарова);

Информация — это отрицание энтропии (Леон Бриллюэн);

Информация — это мера сложности структур (Моль);

Информация — это отраженное разнообразие (Урсул);

Информация — это содержание процесса отражения (Тузов);

Информация — это вероятность выбора (Яглом).

Современное научное представление об информации очень точно сформулировал Норберт Винер, "отец" кибернетики. А именно:

Информация — это обозначение содержания, полученного из внешнего мира в процессе нашего приспособления к нему и приспособления к нему наших чувств.

Подходы к определению количества информации. Формулы Хартли и Шеннона.

Американский инженер Р. Хартли в 1928 г. процесс получения информации рассматривал как выбор одного сообщения из конечного наперед заданного множества из N равновероятных сообщений, а количество информации I , содержащееся в выбранном сообщении, определял как двоичный логарифм N .

Формула Хартли: $I = \log_2 N$

Допустим, нужно угадать одно число из набора чисел от единицы до ста. По формуле Хартли можно вычислить, какое количество информации для этого требуется: $I = \log_2 100 > 6,644$. Таким образом, сообщение о верно угаданном числе содержит количество информации, приблизительно равное 6,644 единицы информации.

Для задач такого рода американский учёный Клод Шеннон предложил в 1948 г. другую формулу определения количества информации, учитывающую возможную неодинаковую вероятность сообщений в наборе.

Формула Шеннона: $I = - (p_1 \log_2 p_1 + p_2 \log_2 p_2 + \dots + p_N \log_2 p_N)$,
где p_i — вероятность того, что именно i -е сообщение выделено в наборе из N сообщений.

Легко заметить, что если вероятности $p_1, \dots, p_N$ равны, то каждая из них равна $1 / N$, и формула Шеннона превращается в формулу Хартли.

Помимо двух рассмотренных подходов к определению количества информации, существуют и другие. Важно помнить, что любые теоретические результаты применимы лишь к определённым кругу случаев, очерченному первоначальными допущениями.

В качестве единицы информации Клод Шеннон предложил принять один бит (англ. bit — binary digit — двоичная цифра).

Бит в теории информации — количество информации, необходимое для различения двух равновероятных сообщений (типа "орел"—"решка", "чет"—"нечет" и т.п.).

В вычислительной технике битом называют наименьшую "порцию" памяти компьютера, необходимую для хранения одного из двух знаков "0" и "1", используемых для внутримашинного представления данных и команд.

Знания и ЭВМ.

Что-то типа...

Числа в ЭВМ

Рассмотрим теперь более подробно процесс кодирования чисел в компьютере. Среди чисел, которые мы используем, встречаются натуральные, целые, рациональные, иррациональные.

В вычислительных машинах применяются две формы представления чисел:

- естественная форма или форма с фиксированной запятой (точкой);
- нормализованная форма или форма с плавающей запятой (точкой);

С фиксированной запятой числа изображаются в виде последовательности цифр с постоянным для всех чисел положением запятой, отделяющей целую часть от дробной. Например, 32,54; 0,0036; -108,2. Эта форма проста, естественна, но имеет небольшой диапазон представления чисел и поэтому не всегда приемлема при вычислениях. Если в результате операции получится число, выходящее за допустимый диапазон, происходит переполнение разрядной сетки и дальнейшие вычисления теряют смысл. В современных компьютерах форма представления чисел с фиксированной запятой используется только для целых чисел.

С плавающей запятой числа изображаются в виде $X = \pm M \times P^{\pm r}$, где M - мантисса числа (правильная дробь в пределах $0,1 \leq M < 1$), r - порядок числа (целое), P - основание системы счисления. Например, приведенные выше числа с фиксированной запятой можно преобразовать в числа с плавающей запятой так: $0,3254 \times 10^2$, $0,36 \times 10^{-2}$, $-0,1082 \times 10^3$. Нормализованная форма представления имеет огромный диапазон чисел и является основной в современных ЭВМ.

Всякое десятичное число, прежде чем оно попадает в память компьютера, преобразуется по схеме: $X_{10} \rightarrow X_2 \rightarrow X_2 = M_2 \times 10^{2r}$

После этого осуществляется ещё одна важная процедура:

- мантисса с её знаком заменяется кодом мантиссы с её знаком;

- порядок числа с его знаком заменяется кодом порядка с его знаком.

Указанные коды двоичных чисел - это образы чисел, которые и воспринимают вычислительные устройства.

Каждому двоичному числу можно поставить в соответствие несколько видов кодов. Существуют следующие коды двоичных чисел:

Прямой код. Прямой код двоичного числа (а это либо мантисса, либо порядок) образуется по такому алгоритму:

1. Определить данное двоичное число - оно либо целое (порядок), либо правильная дробь (мантисса).

2. Если это дробь, то цифры после запятой можно рассматривать как целое число.

3. Если это целое и положительное двоичное число, то вместе с добавлением 0 в старший разряд число превращается в код. Для отрицательного двоичного числа перед ним ставится единица. Например:

число $X_2 = -0,1011012 \rightarrow$ код числа $X_{пр} = 1101101$; число $Y_2 = +0,11011012 \rightarrow$ код числа $Y_{пр} = 01101101$.

Красным цветом выделены знаковые разряды и, кроме того, у кодов отсутствует индекс "2".

Обратный код. Обратный код положительного двоичного числа совпадает с прямым кодом, а для отрицательного числа нужно, исключая знаковый разряд, во всех остальных разрядах нули заменить на единицы и наоборот. Например: число $X_2 = -0,101012 \rightarrow X_{пр} = 1\ 10101 \rightarrow X_{обр} = 101010$;

число $Y_2 = +0,11012 \rightarrow Y_{пр} = 01101 = Y_{обр}$.

Дополнительный код. Дополнительный код положительного числа совпадает с его прямым кодом. Дополнительный код отрицательного числа образуется путём прибавления 1 к обратному коду. Например:

число $X_2 = -0,100102 \rightarrow X_{пр} = 110010 \rightarrow X_{обр} = 101101 \rightarrow X_{доп} = 101110$;

число $Y_2 = +0,1011 \rightarrow Y_{пр} = 01011 = Y_{обр} = Y_{доп}$.

Сложение и вычитание двоичных чисел. Сложение чисел, а также вычитание чисел в обратном или дополнительном кодах выполняется с использованием обычного правила арифметического сложения многоразрядных чисел. Это правило распространяется и на знаковые разряды чисел. Различие же обратного и дополнительного кодов связано с тем, что потом делают с единицей переноса из старшего разряда, изображающего знак числа. При сложении чисел в обратном коде эту единицу надо прибавить к младшему разряду результата, а в дополнительном коде единица переноса из старшего разряда игнорируется. Это очевидно, если вспомнить, что дополнительный код из обратного получается как раз прибавлением единицы. Рассмотрим пример: сложить числа $+18$ и -7 .

Итак, как в обратном, так и в дополнительном кодах результаты сложения совпали и равны $10112 = 1110$.

Умножение и деление двоичных чисел в ЭВМ производится в прямом коде, а их знаки используются лишь для определения знака результата. Также как и в математике, умножение и деление сводится к операциям сдвигов и сложений (с учётом знака числа).

Полученные коды мантиссы и порядка для каждого числа помещаются в ячейки памяти ЭВМ. Для каждой цифры, входящей в код, в ячейке памяти отводится своё отдельное место. Одна ячейка памяти состоит из 8 бит, т.е. 1 байта. В современных компьютерах 2 байта выделяются для одного машинного слова. В последних моделях ПК обработка информации ведётся двойными словами, содержащими 4 байта. Числа с фиксированной запятой имеют формат одного слова, а числа с плавающей запятой - формат двойного слова..

Пример: Дано число с плавающей запятой величиной $-0,625 \times 10^8$. Нужно преобразовать его в машинный код и заполнить 32-разрядную ячейку памяти. Мантисса числа $0,62510 = 0,1012$. Поскольку порядок чисел может быть как положительным, так и отрицательным, то

машинный порядок смещается относительно естественного таким образом, чтобы весь диапазон машинных порядков изменялся от 0 до максимума, определяемого количеством разрядов, выделяемых для размещения чисел порядка. Обычно в 32-разрядной ячейке цифры порядка занимают 7 разрядов старшего байта, а восьмой разряд используется для фиксации знака числа. Семь двоичных разрядов позволяют разместить диапазон десятичных разрядов от -64 до +63. Если обозначить машинный порядок через R, а естественный через r, то связь между ними будет такая: $R_{10} = r_{10} + 6410$. Для двоичной системы счисления $R_2 = r_2 + 10000002$. В нашем примере порядок r равен $810 = 10002$, следовательно $R_2 = 1001000$. В двоичной системе исходное число выглядит так: - 0,101x101000. Запись этого числа в 32-разрядной ячейке представлена в таблице: Знак

числа	Порядок		Мантисса									
№ разряда	31	30	29	28	27	26	25	24	23	22	21	
20	19	18	...	1	0							
Число 1	1	0	0	1	0	0	0	1	0	1	0	0
0	0	0	0									

Хранение нечисловых видов информации

ЭВМ первых двух поколений могли обрабатывать только числовую информацию, полностью оправдывая свое название вычислительных машин. Лишь переход к третьему поколению принес изменения: к этому времени уже назрела настоятельная необходимость использования текстов. Следуя ходу исторического развития, мы тоже начнем наше рассмотрение с этого вида информации.

С точки зрения ЭВМ текст состоит из отдельных символов. К числу символов принадлежат не только буквы (заглавные или строчные, латинские или русские), но и цифры, знаки препинания, спецсимволы типа "=", "(", "&" и т.п. и даже (обратите особое внимание!) пробелы между словами. Да, не удивляйтесь: пустое место в тексте тоже должно иметь свое обозначение.

Каждый символ хранится в виде двоичного кода, который является номером символа. Можно сказать, что компьютер имеет собственный алфавит, где весь набор символов строго упорядочен. Количество символов в алфавите также тесно связано с двоичным представлением и у всех ЭВМ равняется 256. Иными словами, каждый символ всегда кодируется 8 битами, т.е. занимает ровно один байт.

Как видите, хранится не начертание буквы, а ее номер. Именно по этому номеру воспроизводится вид символа на экране дисплея или на бумаге. Поскольку алфавиты в различных типах ЭВМ не полностью совпадают, при переносе с одной модели на другую может произойти превращение разумного текста в "абракадабру". Такой эффект иногда получается даже на одной машине в различных программных средах: например, русский текст, набранный в MS DOS, нельзя без специального преобразования прочитать в Windows. Остается утешать себя тем, что задача перекодировки текста из одной кодовой таблицы в другую довольно проста и при наличии программ машина сама великолепно с ней справляется.

Наиболее стабильное положение в алфавитах всех ЭВМ занимают латинские буквы, цифры и некоторые специальные знаки. Это связано с существованием международного стандарта ASCII (American Standard Code for Information Interchange - Американский стандартный код для обмена информацией). Русские же буквы не стандартизированы и могут иметь различную кодировку.

Ниже в качестве примера приводится таблица стандартной части алфавита ЭВМ - символы с шестнадцатиричными кодами с 20 до 7F.

	0	1	2	3	4	5	6						
	7	8	9	A	B	C	D	E	F				
2		!	"	#	\$	%	&	'	(	)	*	+	,
	-	.	/										
3	0	1	2	3	4	5	6	7	8	9	:	;	<
	=	>	?										
4	@	A	B	C	D	E	F	G	H	I	J	K	L
	M	N	O										
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\
	]	^	_										
6	`	a	b	c	d	e	f	g	h	i	j	k	l
	m	n	o										
7	p	q	r	s	t	u	v	w	x	y	z	{	
	}	~											

Таблица стандартных ASCII символов.

Пример: код 52 соответствует "R".

Нельзя также пройти мимо еще одного интересного факта: каждый символ текста имеет свой числовой код, но не каждому коду соответствует отображаемый на экране символ. Речь идет о существовании так называемых **УПРАВЛЯЮЩИХ КОДОВ**, величина которых меньше шестнадцатиричного числа 20 (т.е. 32 в десятичной системе счисления). При получении этих кодов внешние устройства не изображают какого-либо символа, а выполняют те или иные управляющие действия. Так, код 07 вызывает подачу стандартного звукового сигнала, а код 0C - очистку экрана. Особую роль играют коды 0A (перевод строки, обозначаемый часто LF) и 0D (возврат каретки - CR). Первый вызывает перемещение в следующую строку без изменения позиции, а второй - на начало текущей строки. Таким образом, для перехода на начало новой строки требуются оба кода и в любом тексте эта "неразлучная пара" кодов хранится после каждой строки.

Обратим внимание читателя на то, что названия возврат каретки и перевод строки имеют историческое происхождение и связаны с устройством пишущей машинки.

Теперь рассмотрим очень кратко, не вдаваясь в технические подробности, основные принципы хранения в памяти ЭВМ графической информации. В отличие от только что рассмотренного текстового режима дисплея, когда минимальной единицей изображения является символ, при отображении графики картинка строится из отдельных элементов - **ПИКСЕЛОВ** (от английских слов PICTURE ELEMENt, означающих "элемент картинки"). Очень часто пиксел совпадает с точкой дисплея, но это совсем необязательно: например, в некоторых видеорежимах 1 пиксел может состоять из 2 или 4 точек экрана.

Каждый пиксел характеризуется цветом. Как и вся остальная информация в ЭВМ, цвет кодируется числом. В зависимости от количества допустимых цветов, число двоичных разрядов на один пиксел будет различным. Так, для черно-белой картинки закодировать цвет точки можно одним битом: 0 - черный, 1 - белый. Для случая 16 цветов требуется уже по 4 разряда на каждую точку, а для 256 цветов - 8, т.е. 1 байт.

Для того, чтобы наглядно представить себе, как хранится в памяти ЭВМ простейшее изображение, рассмотрим для примера белый квадратик на черном фоне размером 4x4. В черно-белом режиме это будет выглядеть наиболее компактно (сначала для наглядности приведен двоичный, а затем шестнадцатиричный вид):

```
1111    F
1001    9
```

1001 9
1111 F

В режиме 16-цветной графики это же самое изображение потребует памяти в 4 раза больше: 1111 1111 1111 1111 F F F F

1111 0000 0000 1111 F 0 0 F
1111 0000 0000 1111 F 0 0 F
1111 1111 1111 1111 F F F F

Наконец, при 256 цветах на каждую точку требуется уже по байту и наш квадратик разрастется еще вдвое: 11111111 11111111 11111111 11111111 FF FF FF FF

11111111 00000000 00000000 11111111 FF 00 00 FF
11111111 00000000 00000000 11111111 FF 00 00 FF
11111111 11111111 11111111 11111111 FF FF FF FF

Обратите внимание на то, что белый цвет, как самый яркий, обычно имеет максимально возможный номер. Поэтому для черно-белого режима он равен 1, для 16-цветного - 15, а для 256 цветов - 255.

Осталось обсудить вопрос, как кодируются промежуточные цвета. Например, вполне естественно со стороны читателя спросить: какой номер имеет, например, красный цвет? К сожалению, методы кодирования цвета даже для одной и той же ЭВМ могут довольно существенно различаться. Причем не только в зависимости от конструкции дисплея, но даже от графического режима, в котором тот в данный момент работает! Более того, соответствие между номерами цветов и их представлением на экране можно переопределять по усмотрению пользователя (это называется изменением палитры). Поэтому давайте ограничимся в качестве примера стандартным 16-цветным набором для наиболее распространенного компьютера IBM PC (по данным [2]): 0 - черный 8 - темно-серый

1 - синий 9 - ярко-синий
2 - зеленый A - ярко-зеленый
3 - голубой B - ярко-голубой
4 - красный C - ярко-красный
5 - розовый D - ярко-розовый
6 - коричневый E - ярко-желтый
7 - серый F - ярко-белый

Таким образом, графическая информация, также как числовая и текстовая, в конечном счете заносится в память в виде двоичных чисел.

Несколько забегаая вперед, к следующему поколению, рассмотрим дополнительно кодирование звуковой информации (ЭВМ третьего поколения, конечно, еще не умела обрабатывать этот вид информации). Принцип преобразования звукового сигнала в цифровую форму и его последующее воспроизведение показаны на рис.2.

Запись звука происходит следующим образом. Выбирается система равноотстоящих друг от друга уровней напряжения сигнала и каждому из них ставится в соответствие определенный номер. Затем через равные очень небольшие промежутки времени измеряется уровень входного сигнала и определяется, к какому из стандартных уровней он ближе всего подходит; номер найденного уровня и записывается в память в качестве громкости звука в данный момент.

Для высококачественной стереофонической записи используется частота 44000 Гц, т.е. измерение происходит десятки тысяч раз в секунду.

При воспроизведении данные считываются, и с такой же самой, как и при записи, высокой частотой компьютер изменяет интенсивность звука в зависимости от прочитанных номеров уровней. Интересно, что регулировка громкости при таком методе воспроизведения в самом прямом смысле осуществляется с помощью умножения: например, чтобы увеличить громкость вдвое, перед воспроизведением номер уровня необходимо также удвоить.

Таким образом, рассмотрев принципы хранения в ЭВМ различных видов информации, можно сделать важный вывод о том, что все они так или иначе преобразуются в числовую форму и кодируются набором нулей и единиц. Благодаря такой универсальности представления данных, если из памяти наудачу извлечь содержимое какой-нибудь ячейки, то принципиально невозможно определить, какая именно информация там закодирована: текст, число или картинка.

Эволюционная классификация вычислительных систем на примере развития отечественной техники.

Критерий, положенный в основу эволюционной классификации ЭВМ.

Поколение первое. Компьютеры на электронных лампах.

Появление электронно-вакуумной лампы позволило учёным реализовать в жизнь идею создания вычислительной машины.

Компьютеры на основе электронных ламп появились в 40-х годах XX века. Первая электронная лампа - вакуумный диод - была построена Флемингом лишь в 1904 году, хотя эффект прохождения электрического тока через вакуум был открыт Эдисоном в 1883 году. Вскоре Ли де Форрест изобретает вакуумный триод - лампу с тремя электродами, затем появляется газонаполненная электронная лампа - тиратрон, пятиэлектродная лампа - пентод и т. д. До 30-х годов электронные вакуумные и газонаполненные лампы использовались главным образом в радиотехнике. Но в 1931 году англичанин Винни-Вильямс построил (для нужд экспериментальной физики) тиратронный счетчик электрических импульсов, открыв тем самым новую область применения электронных ламп. Электронный счетчик состоит из ряда триггеров. Триггер, изобретенный М. А. Бонч-Бруевичем (1918) и - независимо - американцами У. Икклзом и Ф. Джорданом (1919), содержит 2 лампы и в каждый момент может находиться в одном из двух устойчивых состояний; он представляет собой электронное реле. Подобно электромеханическому, оно может быть использовано для хранения одной двоичной цифры.

Использование электронной лампы в качестве основного элемента ЭВМ создавало множество проблем. Из-за того, что высота стеклянной лампы - 7см, машины были огромных размеров. Каждые 7-8 мин. одна из ламп выходила из строя, а так как в компьютере их было 15 - 20 тысяч, то для поиска и замены поврежденной лампы требовалось очень много времени. Кроме того, они выделяли огромное количество тепла, и для эксплуатации "современного" компьютера того времени требовались специальные системы охлаждения.

Чтобы разобраться в запутанных схемах огромного компьютера, нужны были целые бригады инженеров. Устройств ввода в этих компьютерах не было, поэтому данные заносились в память при помощи соединения нужного штеккера с нужным гнездом.

Примерами машин I-го поколения могут служить Mark 1, ENIAC, EDSAC (Electronic Delay Storage Automatic Calculator), - первая машина с хранимой программой. UNIVAC (Universal Automatic Computer). Первый экземпляр Юнивака был передан в Бюро переписи населения США. Позднее было создано много разных моделей Юнивака, которые нашли применение в различных сферах деятельности. Таким образом, Юнивак стал первым серийным компьютером. Кроме того, это был первый компьютер, где вместо перфокарт использовалась магнитная лента.

ЭВМ первого поколения обладали небольшим быстродействием в несколько десятков тыс. оп./сек. В качестве внутренней памяти применялись ферритовые сердечники.

Основной недостаток этих ЭВМ – рассогласование быстродействия внутренней памяти и АЛУ и УУ за счет различной элементной базы. Общее быстродействие определялось более медленным компонентом – внутренней памятью и снижало общий эффект. Уже в ЭВМ первого поколения делались попытки ликвидировать этот недостаток путем асинхронизации работы устройств и введения буферизации вывода, когда передаваемая информация «сбрасывается» в буфер, освобождая устройство для дальнейшей работы (принцип автономии). Таким образом, для работы устройств ввода-вывода использовалась собственная память.

Существенным функциональным ограничением ЭВМ первого поколения являлась ориентация на выполнение арифметических операций. При попытках приспособления для задач анализа они оказывались неэффективными.

Языков программирования как таковых еще не было, и для кодирования своих алгоритмов программисты использовали машинные команды или ассемблеры. Это усложняло и затягивало процесс программирования. К концу 50-х годов средства программирования претерпевают принципиальные изменения: осуществляется переход к автоматизации программирования с помощью универсальных языков и библиотек стандартных программ. Использование универсальных языков повлекло возникновение трансляторов.

Программы выполнялись позадачно, т.е. оператору надо было следить за ходом решения задачи и при достижении конца самому инициировать выполнение следующей задачи.

Начало современной эры использования ЭВМ в нашей стране относят к 1950 году, когда в институте электротехники АН УССР под руководством С.А. Лебедева была создана первая отечественная ЭВМ под названием МЭСМ – Малая Электронная Счетная Машина. В течение первого этапа развития средств вычислительной техники в нашей стране создан ряд ЭВМ: БЭСМ, Стрела, Урал, М-2.

Поколение второе. Транзисторные компьютеры.

1 июля 1948 года на одной из страниц "Нью-Йорк Таймс", посвященной радио и телевидению, было помещено скромное сообщение о том, что фирма "Белл телефон лабораториз" разработала электронный прибор, способный заменить электронную лампу. Физик-теоретик Джон Бардин и ведущий экспериментатор фирмы Уолтер Брайттен создали первый действующий транзистор. Это был точечно-контактный прибор, в котором три металлических "усика" контактировали с брусом из поликристаллического германия. Первые компьютеры на основе транзисторов появились в конце 50-х годов, а к середине 60-х годов были созданы более компактные внешние устройства, что позволило фирме Digital

Equipment выпустить в 1965 г. первый мини-компьютер PDP-8 размером с холодильник (!!) и стоимостью всего 20 тыс. долларов (!!) .

Созданию транзистора предшествовала упорная, почти 10-летняя работа, которую еще в 1938 году начал физик теоретик Уильям Шокли . Применение транзисторов в качестве основного элемента в ЭВМ привело к уменьшению размеров компьютеров в сотни раз и к повышению их надежности.

И все-таки самой удивительной способностью транзистора является то, что он один способен трудиться за 40 электронных ламп и при этом работать с большей скоростью, выделять очень мало тепла и почти не потреблять электроэнергию. Одновременно с процессом замены электронных ламп транзисторами совершенствовались методы хранения информации. Увеличился объем памяти, а магнитную ленту , впервые примененную в ЭВМ Юнивак , начали использовать как для ввода, так и для вывода информации. А в середине 60-х годов получило распространение хранение информации на дисках. Большие достижения в архитектуре компьютеров позволило достичь быстродействия в миллион операций в секунду! Примерами транзисторных компьютеров могут послужить "Стретч" (Англия), "Атлас" (США). В то время СССР шел в ногу со временем и выпускал ЭВМ мирового уровня (например БЭСМ-6 ").

Получает дальнейшее развитие принцип автономии – он реализуется уже на уровне отдельных устройств, что выражается в их модульной структуре. Устройства ввода-вывода снабжаются собственными УУ (называемыми контроллерами), что позволило освободить центральное УУ от управления операциями ввода-вывода.

Совершенствование и удешевление ЭВМ привели к снижению удельной стоимости машинного времени и вычислительных ресурсов в общей стоимости автоматизированного решения задачи обработки данных, в то же время расходы на разработку программ (т.е. программирование) почти не снижались, а в ряде случаев имели тенденции к росту. Таким образом, намечалась тенденция к эффективному программированию, которая начала реализовываться во втором поколении ЭВМ и получает развитие до настоящего времени.

Начинается разработка на базе библиотек стандартных программ интегрированных систем, обладающих свойством переносимости, т.е. функционирования на ЭВМ разных марок. Наиболее часто используемые программные средства выделяются в ППП для решения задач определенного класса.

Совершенствуется технология выполнения программ на ЭВМ: создаются специальные программные средства - системное ПО.

Цель создания системного ПО – ускорение и упрощение перехода процессором от одной задачи к другой. Появились первые системы пакетной обработки, которые просто автоматизировали запуск одной программ за другой и тем самым увеличивали коэффициент загрузки процессора. Системы пакетной обработки явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными для управления вычислительным процессом. В ходе реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Совокупность нескольких заданий, как правило, в виде колоды перфокарт, получила название пакета заданий. Этот элемент жив до сих пор: так называемые пакетные (или командные) файлы MS DOS есть не что иное, как пакеты

заданий (расширение в их имени bat является сокращением от английского слова batch, что означает пакет).

К отечественным ЭВМ второго поколения относятся Проминь, Минск, Раздан, Мир.

Поколение третье. Интегральные схемы.

Подобно тому, как появление транзисторов привело к созданию второго поколения компьютеров, появление интегральных схем ознаменовало собой новый этап в развитии вычислительной техники - рождение машин третьего поколения. Интегральная схема, которую также называют кристаллом, представляет собой миниатюрную электронную схему, вытравленную на поверхности кремниевого кристалла площадью около 10 мм².

Первые интегральные схемы (ИС) появились в 1964 году. Сначала они использовались только в космической и военной технике. Сейчас же их можно обнаружить где угодно, включая автомобили и бытовые приборы. Что же касается компьютеров, то без интегральных схем они просто немыслимы!

Появление ИС означало подлинную революцию в вычислительной технике. Ведь она одна способна заменить тысячи транзисторов, каждый из которых в свою очередь уже заменил 40 электронных ламп. Другими словами, один крошечный кристалл обладает такими же вычислительными возможностями, как и 30-тонный Эниак ! Быстродействие ЭВМ третьего поколения возросло в 100 раз, а габариты значительно уменьшились.

Ко всем достоинствам ЭВМ третьего поколения добавилось еще и то, что их производство оказалось дешевле, чем производство машин второго поколения. Благодаря этому, многие организации смогли приобрести и освоить такие машины. А это, в свою очередь, привело к росту спроса на универсальные ЭВМ, предназначенные для решения самых различных задач.

Большинство созданных до этого ЭВМ являлись специализированными машинами, на которых можно было решать задачи какого-то одного типа.

Расширение функциональных возможностей ЭВМ увеличило сферу их применения, что вызвало рост объема обрабатываемой информации и поставило задачу хранения данных в специальных базах данных и их ведения. Так появились первые системы управления базами данных – СУБД.

Изменились формы использования ЭВМ: введение удаленных терминалов (дисплеев) позволило широко и эффективно внедрить режим разделения времени и за счет этого приблизить ЭВМ к пользователю и расширить круг решаемых задач.

Обеспечить режим разделения времени позволил новый вид ОС, поддерживающих мультипрограммирование. Мультипрограммирование - это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ. Пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при последовательном выполнении программ (однопрограммный режим), а выполняет другую программу (многопрограммный режим). При этом каждая программа загружается в свой участок внутренней памяти, называемый разделом. Мультипрограммирование нацелено на создание для каждого отдельного пользователя иллюзии единоличного использования вычислительной машины, поэтому эти ОС носили интерактивный характер, когда в процессе диалога с ЭВМ пользователь решал свои задачи.

Поколение четвертое. Большие интегральные схемы.

Вы уже знаете, что электромеханические детали счетных машин уступили место электронным лампам, которые в свою очередь уступили место транзисторам, а последние - интегральным схемам. Могло создаться впечатление, что технические возможности ЭВМ исчерпаны. В самом деле, что же можно еще придумать?

Чтобы получить ответ на этот вопрос, давайте вернемся к началу 70-х годов. Именно в это время была предпринята попытка выяснить, можно ли на одном кристалле разместить больше одной интегральной схемы. Оказалось, можно! Развитие микроэлектроники привело к созданию возможности размещать на одном-единственном кристалле тысячи интегральных схем. Так, уже в 1980 году, центральный процессор небольшого компьютера оказался возможным разместить на кристалле, площадью всего в четверть квадратного дюйма (1,61 см²). Началась эпоха микрокомпьютеров.

Наиболее значительным стало появление персональных ЭВМ, что позволило приблизить ЭВМ к своему конечному пользователю. Компьютеры стали широко использоваться неспециалистами, что потребовало разработки "дружественного" программного обеспечения. Возникают ОС, поддерживающие графический интерфейс, интеллектуальные ППП, операционные оболочки. В связи с возросшим спросом на ПО совершенствуются технологии его разработки – появляются развитые системы программирования, инструментальные среды пользователя.

В середине 80-х стали бурно развиваться сети персональных компьютеров, работающие под управлением сетевых или распределенных ОС. В сетевых ОС хорошо развиты средства защиты информации от несанкционированного доступа. Распределенные ОС обладают схожими с сетевыми системами функциями работы с файлами и другими ресурсами удаленных компьютеров, но там слабее выражены средства защиты.

Пятое поколение ЭВМ

На ЭВМ пятого поколения ставятся совершенно другие задачи, нежели при разработке всех прежних ЭВМ. Если перед разработчиками ЭВМ с I по IV поколений стояли такие задачи, как увеличение производительности в области числовых расчётов, достижение большой ёмкости памяти, то основной задачей разработчиков ЭВМ V поколения является создание искусственного интеллекта машины (возможность делать логические выводы из представленных фактов), развитие "интеллектуализации" компьютеров - устранения барьера между человеком и компьютером. Компьютер теперь используется и дома, это компьютерные игры, прослушивание высококачественной музыки, просмотр фильмов. Уже сейчас компьютеры способны воспринимать информацию с рукописного или печатного текста, с бланков, с человеческого голоса, узнавать пользователя по голосу, осуществлять перевод с одного языка на другой. Это позволяет общаться с компьютерами всем пользователям, даже тем, кто не имеет специальных знаний в этой области.

Основоположники отечественной вычислительной техники.

1947-1948 г. - начало работ по созданию в Институте электроники Академии наук Украины под руководством академика Сергея Алексеевича Лебедева первой отечественной первой универсальной ламповой ЭВМ - МЭСМ (малой электронной счетной машины).

1948 г. - И. С. Брук получил диплом на изобретение ЭВМ и представил проект создания такой машины, названной М-1. В декабре И. С. Брук и Б. И. Рамеев получили авторское свидетельство на изобретение "Автоматическая цифровая электронная машина". Из-за организационных трудностей работы затянулись.

1950 г. - вступает в действие первая в СССР вычислительная электронная цифровая машина МЭСМ, самая быстродействующая тогда в Европе, а в 1951 году она официально вводится в эксплуатацию.

1952 г. - началась практическая эксплуатация ЭВМ М-1, разработанной под руководством И. С. Брук. За М-1 последовали М-2. Ее разработку выполнила группа выпускников МЭИ, возглавляемая М.А.Карцевым. Затем была выпущена машина М-3. ЭВМ М-3 занимает особое место в развитии вычислительной техники. С некоторыми модификациями она была повторена в Ереване, Минске, а также за рубежом - в Китае и Венгрии, где послужила основой для развития математического машиностроения.

1953 г. - в Академии наук СССР (Москва), вводится в эксплуатацию БЭСМ (большая электронная счетная вычислительная машина), разработанная в Институте точной механики и вычислительной техники АН СССР. под руководством С.А.Лебедева. БЭСМ относится к классу цифровых вычислительных машин общего назначения, ориентированных на решение сложных задач науки и техники.

1953 г. в Москве, в СКБ Министерства машиностроения и приборостроения под руководством Ю. Я. Базилевского и Б. И. Рамеева закончена разработка серийной ЭВМ "Стрела" общего назначения.

1954 г. - начался серийный выпуск ЭВМ "Стрела". Серия оказалась очень маленькой: всего за четыре года было выпущено семь машин. Тем не менее 1954 г. - это год становления отечественной индустрии ЭВМ.

1955 г. - институт точной механики и вычислительной техники АН СССР ввел усовершенствования в Большую ЭВМ "БЭСМ", повысившие её быстродействие до 8000 операций в секунду.

1956 г. - в СССР Госкомиссии представлена ЭВМ М-3, разработанная инициативной группой (И. С. Брук, Н.Я.Матюхин, В.В.Белынский, Г.П.Лопато, Б.М.Каган, В.М.Долкарт, Б.Б.Мелик-Шахназаров).

1956 г. - разработана ЭВМ БЭСМ-2. Руководитель разработки - С.А.Лебедева

1957 г. - завершена разработка одной из наиболее совершенных чисто релейных вычислительных машин РВМ-1. Машина сконструирована и построена под руководством советского инженера И. И. Бессонова (начало постройки относится к 1954 году).

1957 г. - в Пензе под руководством Б. И. Рамеева создана одноадресная ламповая ЭВМ "Урал-1" общего назначения, ориентированных на решение инженерно-технических и планово-экономических задач. Она положила начало целому семейству малых ЭВМ "Урал".

1958 г. - введена в эксплуатацию ЭВМ М-20 (Казань) Разработка выполнена ИТМ и ВТ совместно с СКБ-245. Руководитель: С.А.Лебедев, заместитель главного конструктора М. К. Сулим, М. Р. Шура-Бура. М-20 - цифровая электронная вычислительная машина общего

назначения, ориентированная на решение сложных математических задач. Она послужила исходной моделью семейства совместимых вычислительных машин М-220 и М-222.

1958 г. - начало выпуска в Ульяновске БЭСМ-2 (С.А.Лебедев, В.А.Мельников).

1958 г. - в институте кибернетики АН УССР разработана электронная цифровая вычислительная машина "КИЕВ", предназначенная для решения широкого круга научных и инженерных задач.

1958 г. - в Ереване под руководством Ф.Т. Саркисяна (Б.Б.Мелик-Шахназаров) создана ЭВМ "Раздан".

1958 г. - под руководством Н.П. Брусенцова в вычислительном центре Московского университета была создана и запущена в производство первая и единственная в мире троичная ЭВМ "Сетунь". "Сетунь" - малая цифровая вычислительная машина, предназначенная для решения научно-технических и экономических задач средней сложности. Серийно выпускалась 1962-1964.

1959 — 1967 гг., второе поколение ЭВМ

1959 г. - созданы опытные образцы ЭВМ М-40, М-50 для систем противоракетной обороны (ПРО). Разработчики - С.А.Лебедев и В.С.Бурцев (Ленинская премия 1966 г. за специализированный автоматизированный комплекс обработки информации для системы ПРО на базе этих ЭВМ).

1959 г. - начало выпуска в Минске ЭВМ "Минск-1" применялась в основном для решения инженерных, научных и конструкторских задач математического и логического характера. (Г.П.Лопато).

1959 г. - в СССР была введена в эксплуатацию первая ламповая специализированная стационарная ЭВМ СПЕКТР-4 предназначенная для наведения истребителей-перехватчиков.

1959 г. - под руководством Я.А.Хетагурова (ЦМНИИ-1) создана первая в СССР мобильная полупроводниковая ЭВМ "КУРС" для обработки радиолокационной информации.

1960 г. - в СССР разработана первая полупроводниковая управляющая машина "Днепр" (В.М.Глушков, Б.Н. Малиновский).

1960 г. - создана первая микропрограммная специализированная ЭВМ "Тетива" для системы ПВО. Производство в Минске. Главный конструктор Н.Я.Матюхин.

1961 г. - начат серийный выпуск ЦВМ "Раздан-2", предназначена для решения научно-технических и инженерных задач, малой производительности (скорость вычислений - до 5 тысяч операций в 1 секунд).

1961 г. - в СССР создана первая в стране серийная универсальная полупроводниковая управляющая ЭВМ широкого назначения "Днепр-1" (В.М.Глушков, Б.Н. Малиновский). Выпускалась на протяжении 10 лет.

1961 г. - начало выпуска "Урал-4" (Пенза). Руководитель работ - Б.И.Рамеев.

1962 г. - в ИТМиВТ выпущена ЭВМ БЭСМ-4.

1962 г. - в Северодонецком научно-исследовательском институте управляющих вычислительных машин создана "МППИ-1" - машина первичной переработки информации - информационно-вычислительная машина. Применялась "МППИ-1" в химической, нефтеперерабатывающей, металлургической и других отраслях промышленности.

1962 г. - создан опытный образец ЭВМ "Восток" (А.Н.Мямлин).

1962 г. - в Институте кибернетики АН УССР разработано семейство малых цифровых электронных вычислительных машин "Промінь", предназначенных для автоматизации инженерных расчетов средней сложности.

1962 г. - разработана первая в Украине ЭВМ с асинхронным управлением "Киев" (В.М.Глушков, Е.Л.Ющенко, Л.Н.Дашевский). Запуск ее в ОИЯИ (Дубна).

1962 г. - начало выпуска ЭВМ "Минск-2" с использованием импульсно потенциальной элементной базы и введением представления данных в виде двоично-десятичных чисел и алфавитно-цифровых слов (Минск) (С 1965 г. – "Минск-22"). В.В.Пржиялковский.

1963 г. - начало серийного производства малой ЭВМ для инженерных расчетов "Промінь" на Северодонецком заводе вычислительных машин. В ней использовалось ступенчатое микропрограммное управление (С.Б.Погребинский, В.Д.Лосев).

1963 г. - начало выпуска ЭВМ "Минск-32" (Минск) с внешней памятью на сменных магнитных дисках (В.Я.Пыхтин).

1963 г. - создан многомашинный вычислительный комплекс "Минск-222" (Г.П.Лопато).

1964 г. - в Ереванском научно-исследовательском институте математических машин разработана и запущена в производство ЭВМ с микропрограммным управлением "Наири".

1964 г. - начало выпуска ряда ЭВМ Урал; Урал-11, Урал-14, Урал-16 (с 1969 г.) с операциями над словами переменной длины и структурной адресацией (Б.И.Рамеев, В.И.Бурков, А.Н.Невский, Г.С.Горшков, А.С.Горшков, В.И.Мухин).

1964 г. - начало выпуска электронная цифровая вычислительная машина общего назначения "Весна". Производство в Минске. Главный конструктор В.С.Полин (В.К.Левин, М.Р.Шура-Бура, В.С.Штаркман, В.А.Слепушкин, Ю.А.Котов).

1965 г. - группой инженеров в Институте точной механики и вычислительной техники под руководством С.А.Лебедева была создана мощная полупроводниковая ЭВМ БЭСМ-6 ("Быстродействующая электронно-счетная машина"). БЭСМ-6 занимает особенно важное место в развитии и использовании вычислительной техники в СССР. Это первая в СССР суперЭВМ с производительностью 1 миллион оп/сек.

1965 г. - в Киеве Институте кибернетики АН УССР создана машина МИР-1. Разработчики В.М.Глушков, Ю.В.Благовещенский, А.А.Летичевский, А.А.Летинский, В.Д.Лосев, И.Н.Молчанов, С.Б. Погребинский, А.А.Стогний, З.Л.Рабинович.

1965 г. - начало выпуска в Казани полупроводниковых ЭВМ М-220 и М-222 с производительностью до 200 тыс. оп/сек, продолжающих линию ЭВМ М-20.

Предназначены для решения научно-технических, а также отдельных классов экономических задач. Главный конструктор М.К.Сулим.

1965 г. - в Ереванском научно-исследовательском институте математических машин выпущена модификация ЭВМ "Наири-М".

1965 г. - создан макет ЭВМ с системой счисления в остаточных классах (И.Я.Акушский, Д.И.Юдицкий). Технический проект ЭВМ "Украина" с развитыми системами интерпретации. В.М.Глушков, З.Л.Рабинович, А.А.Стогний.

1966 г. - завершается разработка проекта большой ЭВМ "Украина", предвосхитившего многие идеи американских больших ЭВМ 70-х годов.

1966 г. - начат серийный выпуск ЦВМ "Раздан-3", предназначенной для решения научно-технических, планово-экономических и статистических задач.

1966 г. - для командных пунктов ПВО в СССР была создана мощная по тем временам специализированная ЭВМ ГРАНИТ (А.З.Шостак).

1967 г. - начало выпуска в Киеве заводе ВУМ управляющей ЭВМ "Днепр-2". Разработка Института кибернетики АН Украины (В.М.Глушков, А.Г.Кухарчук).

1967 г. - в Ереванском научно-исследовательском институте математических машин выпущена модификации ЭВМ "Наири-С" и "Наири-2".

1967 г. - ввод в действие электронной счетной машины БЭСМ-6 в Вычислительном центре АН СССР. Начало ее серийного производства на заводе счетно-аналитических машин (САМ) в Москве. За все время (до начала 80-х гг.) было построено около 350 БЭСМ-6.

1968 — 1973 гг., третье поколение ЭВМ

1968 г. - проект полностью параллельной вычислительной системы М-9 с производительностью порядка 10 оп/сек. В М-9 операции задавались над функциями двух переменных. М.А.Карцев.

1968 г. - начало производства ЭВМ МИР-2, созданной под руководством В.М.Глушкова в Киеве.

1969 г. - "РУТА-110" - комплекс устройств обработки, ввода, хранения, вывода, а также дистанционного сбора и выдачи алфавитно-цифровой информации, предназначенный для создания локальных систем обработки данных. Разработан СКВ вычислительных машин (г. Вильнюс).

1969 г. 5Э92Б - двухпроцессорный компьютер на дискретных полупроводниковых схемах, основной компьютер в первой системе ПРО Москвы;

1970 г. - создана многомашинная система коллективного пользования "АИСТ-0" на базе нескольких М-20 под управлением "Минск-32". Разработчики А.П.Ершов, Г.И.Кожухин, Г.П.Макаров, М.И. Нечепуренко, И.В.Поттосин.

1970 г. - в Ереванском научно-исследовательском институте математических машин выпущена модификации ЭВМ "Наири-3" и "Наири-3-1" (на интегральных гибридных микросхемах).

1971 г. - начало выпуска модели ЕС-1020 (20 тыс. оп/сек), Минск. В.В.Пржиялковский.

1973 г. - начало выпуска модели ЕС-1030 (100 тыс. оп/сек), Казань (разработка выполнена в Ереване, М. Семирджан).

1973 г. - с использованием БЭСМ-6 была создана многомашинная система с переменной структурой АС-6 для задач управления космическими полетами в СССР.

1973 г. - начало выпуска ЭВМ ЕС-1050 (Москва, Пенза). В.С.Антонов.

1973 г. - начало выпуска высокопроизводительной ЭВМ с многоформатной векторной RISC-архитектурой для систем предупреждения о ракетном нападении и общего наблюдения за космическим пространством М-10 (Загорск, М.А.Карцев).

1974 — 1982 гг., четвертое поколение ЭВМ

1974 г. - начало выпуска модели ЕС-1022, (80 тыс. оп/сек), А.Ростовцев.

1976 г. - начало выпуска модели ЕС-1033 (200 тыс. оп/сек), Казань. А.Гусев.

1975 г. - результатом совместных разработок специалистами СССР, НРБ, ВНР, ПНР, ЧССР и ГДР явилось создание и выпуск мини-ЭВМ - СМ-1, СМ-2, СМ-3 и СМ-4 с широким диапазоном применений: в научных работах, для управления технологическими процессами, обработки экспериментальных данных в реальном масштабе времени, для автоматизации инженерных и управленческих работ и т.д.

1977 г. - начало выпуска старшей модели "Ряда 1" – ЕС-1060. Москва, Пенза, Ю.Ф.Ломов.

1977 г. - начало выпуска модели ЕС-1035 ("Ряд 2"), Минск (Г.Д.Смирнов).

1977 г. - создание первого симметричного многопроцессорного вычислительного комплекса (МВК) "Эльбрус-1" на ИС средней интеграции со средствами аппаратной поддержки развитой структуризации программ и данных (В.С.Бурцев, Б.А.Бабаян).

1978 г. - начало выпуска модели ЕС-1055.

1978 г. - Начало выпуска СМ-3 и СМ-4, совместимых с М-400 (Москва, Киев). Разработчики Б.Н. Наумов, М.А. Боярченков, А.Н. Кобалевский, В.П. Семик, Ю.Н. Глухов, М.Н. Островский, Б.И. Панферов, В.С. Громов, А.А. Солохин, Б.Я. Фельдман, Е.Н. Филинов, А.В. Филин.

1978 г. - Начало выпуска УВК СМ-1 и СМ-2, совместимых с М-6000/М-7000 (Северодонецк). В.В. Резанов, В.М. Костелянский, Г.М. Лехнова, В.А. Барабанов, Л.А. Сопочкин.

1979 г. - начало выпуска в Ереване и Казане модели ЕС-1045 (800 тыс. оп/сек, "Ряд 2"). А.Т.Кучукян.

1979 г. - начало серийного выпуска высокопроизводительных многопроцессорных УВК с перестраиваемой структурой ПС 2000, реализующих распараллеливание на уровне задач, ветвей, векторных и скалярных операций в задачах геофизики, научных экспериментов и др. областей. Разработчики: ИПУ (Москва), НИИУВМ (Северодонецк). И.В.Прангишвили, В.В.Резанов, Э.А.Трахтенгерц, А.А.Новохатний.

1980 г. начало выпуска ЭВМ ЕС-1061. Главный конструктор: Карпилович Ю. В

1980 г. - в Киеве двухпроцессорного комплекса СМ-1410 на базе процессора СМ 4П и специализированного языкового процессора, обеспечивающего совместимость с ЭВМ серии "Мир". Б.Н.Наумов, А.Н.Кабалевский, В.П.Семик, В.С.Зонис, В.В.Родионов, А.С.Шумей.

1980 г. - в ССР разработан (выпущен в 1982 г.) первый отечественный Электронный словарь-переводчик (ЭСП) "Электроника СП" на трех языках (русский, английский, немецкий).

1981 г. - начало выпуска в Киеве УВК СМ-1420 (Б.Н.Наумов, А.Н.Кабалевский, А.С.Шумей, В.П.Семик, Е.Н.Филинов, Т.Д.Чернина, В.В.Родионов, В.А.Афанасьев, С.С.Забара.

1981 г. - начало выпуска УВК СМ 1800, СМ 1803, СМ 1804 на заводах Минприбора в Киеве и Тбилиси (Н.Д.Кабанов, А.Н.Шкамарда, Н.Л.Прохоров, А.Я.Соколов, В.Г.Каневский).

1981 г. - начало выпуска двухпроцессорного комплекса СМ-1210, реализующего основной набор команд СМ-2 и набор векторных и скалярных операций ПС 3000. Северодонецк. В.В.Резанов, В.М.Костелянский.

1982 г. - выпуск профессиональных персональных ЭВМ (ПЭВМ): в СССР ЕС-1840, в Болгарии "Правец-16", в Польше "Мазовия-1016".

1982 г. - начало выпуска в Вильнюсе двухпроцессорного комплекса СМ-1600 для решения учетных, статистических и планово-экономических задач на базе процессора СМ 1420 и процессора, реализующего полную систему команд М-5000.(А.М.Немейкшис).

1983 г. - начало выпуска ЕС-1036 – 400 тыс. оп/сек, "Ряд 3" (Р.М.Асцатуров).

1983 г. - начало выпуска в Загорске многопроцессорной векторной ЭВМ М-13. Руководитель разработки М.А.Карцев.

1984 г. - на завод ВЭМ в г. Пенза передана ЭВМ ЕС 2701.

1985 г. - начало выпуска электронной вычислительной машины ЕС-1066. Главный конструктор: Ломов Ю.С.

1985 г. - начало выпуска многопроцессорного (10 процессоров) вычислительного комплекса "Эльбрус-2" производительностью 125 млн. оп/сек (MIPS). В.С.Бурцев.

1985 г. - выпущен бытовой компьютер "Электроника БК0010-01".

1986 г. - начало выпуска на заводах Минприбора в Киеве, Тбилиси, Черновцах и Орле. УВК СМ 1810, СМ 1814, СМ 1820 (А.Н.Шкамарда, Н. Д.Кабанов, В.С.Кравченко, Э. М.Пройдаков, Г. Каневский, Н.Л.Прохоров, В.М.Головков).

1986 г. - начало выпуска . в Вильнюсе СМ 1700, совместимой с VAX-11 фирмы Digital Equipment Corp. (Н.Л.Прохоров, Г.А.Егоров, В.И.Фролов, Л.Н.Столяр).

1986 г. - выпуск комплексов КУВТ-86, состоящих из учительской машины ДВК-2 и 12 ученических машин БК-0010Ш, объединенных в сеть.

1986 г. - на заводе ВЭМ в Пензе передана в производство ЭВМ ЕС 1766 (до 256 процессоров).

1986 г. - выпуск 8-разрядной персональной ЭВМ Агат-7 (КУВТ "Агат") с открытой архитектурой. Позднее начат выпуск модели Агат-9, программно совместимой с ПЭВМ Apple II+, Правец-8 и Цзыньцзы.

1987 г.- выпуск комплекта учебной вычислительной техники КУВТ "Корвет", включающего рабочее место учителя, 12 рабочих мест учащихся, объединенных в локальную сеть.

1987 г. - государственная комиссия приняла опытный образец нового школьного комплекса учебной вычислительной техники УКНЦ ("Электроника МС 0202, -05115"), который включал в себя рабочее место преподавателя и 12 рабочих мест учащихся, объединенных в локальную сеть.

1987 г. выпущена ПЭВМ "Москва", являющаяся наиболее полноценным в то время ПК. В качестве прототипа был взят ZX Spectrum. Одновременно с ним выпущена "Балтика", оказавшаяся менее совместимой с ZX Spectrum из-за оригинальной схемотехники и повышенной тактовой частоты процессора (4 МГц).

1990 г. - выпущена модификация бытового компьютера БК0011М.

1991 г. - создан опытный образец ЭВМ "ЛОКОН 9В51" (проект "ЛОКОН" -- локально-связанная машина). ЭВМ построена в архитектуре CLIP/ CAM (клеточно-автоматные машины), в основу проекта были положены принципы параллельной обработки информации с расширяемой архитектурой. Главный конструктор и научный руководитель проекта - Бронников В. А.

1993 г. - в России (Москва) начался выпуск персональных компьютеров PS/1.

1994 г. - "Эльбрус-3" - LSI, ECL БИС, 16 процессоров, быстродействие в два раза выше, чем у CRAY-YMP, был изготовлен, но в серию запущен не был (на конечной стадии в работе участвовало дочернее предприятие Института - Московский центр SPARC-технологий).

1996 г. - Завершено изготовление опытного образца ЭВМ "Эльбрус-90 Микро"

Вычислители фон-Нейманновской архитектуры. Конвейерная обработка данных и команд. Архитектура памяти.

Принципы фон-Нейманновской архитектуры ЭВМ.

В основу построения подавляющего большинства компьютеров положены следующие общие принципы, сформулированные в 1945 г. американским ученым Джоном фон Нейманом.

1. Принцип программного управления. Из него следует, что программа состоит из набора команд, которые выполняются процессором автоматически друг за другом в определенной последовательности.

Выборка программы из памяти осуществляется с помощью счетчика команд. Этот регистр процессора последовательно увеличивает хранимый в нем адрес очередной команды на длину команды.

А так как команды программы расположены в памяти друг за другом, то тем самым организуется выборка цепочки команд из последовательно расположенных ячеек памяти.

Если же нужно после выполнения команды перейти не к следующей, а к какой-то другой, используются команды условного или безусловного переходов, которые заносят в счетчик команд номер ячейки памяти, содержащей следующую команду. Выборка команд из памяти прекращается после достижения и выполнения команды “стоп”.

Таким образом, процессор исполняет программу автоматически, без вмешательства человека.

2. Принцип однородности памяти. Программы и данные хранятся в одной и той же памяти. Поэтому компьютер не различает, что хранится в данной ячейке памяти — число, текст или команда. Над командами можно выполнять такие же действия, как и над данными. Это открывает целый ряд возможностей. Например, программа в процессе своего выполнения также может подвергаться переработке, что позволяет задавать в самой программе правила получения некоторых ее частей (так в программе организуется выполнение циклов и подпрограмм). Более того, команды одной программы могут быть получены как результаты исполнения другой программы. На этом принципе основаны методы трансляции — перевода текста программы с языка программирования высокого уровня на язык конкретной машины.

3. Принцип адресности. Структурно основная память состоит из перенумерованных ячеек; процессору в произвольный момент времени доступна любая ячейка. Отсюда следует возможность давать имена областям памяти, так, чтобы к запомненным в них значениям можно было впоследствии обращаться или менять их в процессе выполнения программ с использованием присвоенных имен.

Компьютеры, построенные на этих принципах, относятся к типу фон-неймановских. Но существуют компьютеры, принципиально отличающиеся от фон-неймановских. Для них, например, может не выполняться принцип программного управления, т.е. они могут работать без “счетчика команд”, указывающего текущую выполняемую команду программы. Для обращения к какой-либо переменной, хранящейся в памяти, этим компьютерам не обязательно давать ей имя. Такие компьютеры называются не-фон-неймановскими.

Конвейерная обработка данных.

Основу конвейерной обработки составляет раздельное выполнение некоторой операции в несколько этапов (за несколько ступеней) с передачей данных одного этапа следующему. Производительность при этом возрастает благодаря тому, что одновременно на различных ступенях конвейера выполняются несколько операций. Конвейеризация эффективна только тогда, когда загрузка конвейера близка к полной, а скорость подачи новых операндов соответствует максимальной производительности конвейера. Если происходит задержка, то параллельно будет выполняться меньше операций и суммарная производительность

снизится. Векторные операции обеспечивают идеальную возможность полной загрузки вычислительного конвейера.

При выполнении векторной команды одна и та же операция применяется ко всем элементам вектора (или чаще всего к соответствующим элементам пары векторов). Для настройки конвейера на выполнение конкретной операции может потребоваться некоторое установочное время, однако затем операнды могут поступать в конвейер с максимальной скоростью, допускаемой возможностями памяти. При этом не возникает пауз ни в связи с выборкой новой команды, ни в связи с определением ветви вычислений при условном переходе. Таким образом, главный принцип вычислений на векторной машине состоит в выполнении некоторой элементарной операции или комбинации из нескольких элементарных операций, которые должны повторно применяться к некоторому блоку данных. Таким операциям в исходной программе соответствуют небольшие компактные циклы.

Принцип конвейерной обработки информации нашел широкое применение в вычислительной технике. В первую очередь это относится к конвейеру команд. Практически все современные ЭВМ используют этот принцип. Вместе с тем во многих вычислительных системах наряду с конвейером команд используется и конвейер данных. Сочетание этих двух конвейеров дает возможность достигнуть очень высокой производительности систем на определенных классах задач, особенно если при этом используется несколько конвейерных процессоров, способных работать одновременно и независимо друг от друга. Именно так и построены самые высокопроизводительные системы. Целесообразнее всего рассмотреть принцип конвейерной обработки на примере некоторых, наиболее представительных систем.

К числу ЭВМ, в которых широкое применение нашел конвейер команд, относится одна из лучших свое время машин БЭСМ-6. Она была в течение многих лет самой быстродействующей в стране благодаря целому ряду интересных решений, в том числе и конвейеру команд. Последний обеспечивался использованием восьми независимых модулей ОЗУ, работающих в системе чередования адресов, и большого числа быстрых регистров, предназначенных также и для буферизации командной информации. Это позволило получить на БЭСМ-6 производительность 1 млн. операций в секунду.

Определенный интерес представляет построение систем IBM 360/91, а также более поздней и более современной IBM 360/195. Пять основных устройств системы: ОЗУ, управления памятью с буферным ОЗУ, процессор команд, операционные устройства для выполнения операций с плавающей запятой, с фиксированной запятой и десятичной арифметики работают одновременно и независимо друг от друга. Оперативное ЗУ построено по многомодульному принципу (до 32 модулей), устройство управления памятью работает по принципу конвейера и обеспечивает 8- или 16-кратное чередование адресов при обращении процессора и каналов ввода – вывода информации к ОЗУ.

Кроме конвейера команд в системах IBM 360/91 и IBM 370/195 в обоих ОУ используется также и конвейерная обработка данных. Однако в системе 360/195 конвейер получается довольно внушительный: в каждом цикле осуществляется выборка до 8 команд, расшифровка 16 команд, до 3 операций над адресами и до 3 процессорных операций. Всего в системе одновременно может обрабатываться до 50 команд.

Существенно более полно используется принцип магистральной обработки в системе STAR-100, разработанной фирмой CDC в 1973 г. Надо сказать, что фирма CDC в большинстве разрабатываемых и выпускаемых машин и систем использует принцип

конвейерном обработке для повышения производительности, однако раньше использовался только командный конвейер, а в системе STAR-100 – оба типа конвейера. Система содержит три конвейерных процессора (рис. 3.1): ППЗ – процессор, содержащий конвейерные устройства сложения и умножения с плавающей запятой; ППФЗ – процессор, содержащий конвейерное устройство сложения с плавающей запятой, конвейерное многоцелевое устройство, выполняющее умножение с фиксированной запятой, деление и извлечение квадратного корня; СП – специальный конвейерный 16-разрядный процессор, выполняющий операции с фиксированной запятой и ряд логических операций.

Конвейерные процессоры оперируют с 64- или 32-разрядными числами и каждые 40 нс выдают результаты в блок управления потоками данных и буферами. Оперативное ЗУ построено по модульному принципу (32-модуля памяти) и работает с чередованием адресов под управлением устройства УП (управление памятью). Каждый малый цикл обращения к памяти – 40 нс. (Полный цикл ОЗУ–1,28 мкс, т. е. 40×32 нс.)

Конвейерный сумматор с плавающей запятой состоит из четырех сегментов – специализированных операционных устройств (см. рис. 1.2, а). Продолжительность цикла каждого сегмента составляет 40 нс; таким образом, время выполнения операции сложения с плавающей запятой равно 160 нс.

Конвейерный умножитель включает в себя 8 сегментов, поэтому время выполнения операции умножения составляет 320 нс. Но при загрузке конвейерных процессоров длинной последовательностью операндов, над которыми производится одна и та же операция, результат выдается каждые 40 нс. Учитывая, что каждый из двух основных процессоров может выдавать по два 32-разрядных результата, нетрудно подсчитать, что система STAR-100 может в пределе выполнять до 100 млн. операций в секунду.

Устройства конвейерной обработки далеко не всегда выполняют с жесткой настройкой на одну определенную операцию. Чаще их делают многоцелевыми, вводя в конвейер сегменты, необходимые для реализации полного набора операций, в процессе выполнения которых весь тракт настраивается соответствующим образом. На рис. 3.2 представлена структура системы ASC фирмы «Техас Инструментс» и показано, какие сегменты универсальной цепочки работают при различных операциях.

Одной из наиболее высокопроизводительных вычислительных систем в мире общепризнанно считается система CRAY, созданная в 1976 г. В этой системе конвейерный принцип обработки используется в максимальной степени: имеется и конвейер команд, и конвейер арифметических и логических операций. Кроме того, в системе широко применяется совмещенная обработка информации несколькими устройствами. Все это позволило при решении научных задач достигнуть чрезвычайно высокой производительности – до 250 млн. операций в секунду

Система CRAY (рис. 2.3) состоит из четырех секций: функциональных устройств, регистров, управления программой, памяти и ввода – вывода. В системе 12 функциональных устройств, работающих в режиме конвейера, разбитых на 4 группы: адресную, скалярную, операций с плавающей запятой и векторную. Число сегментов в каждом функциональном устройстве (указано в скобках на схеме) сравнительно невелико, оно зависит от сложности операций и колеблется в пределах от 1 до 14 (вычисление обратной величины). Такое сравнительно небольшое число сегментов в каждом магистральном устройстве имеет определенные преимущества – они сравнительно быстро заполняются. Длительность цикла каждого сегмента составляет 12,5 нс: это значит, что каждые 12,5 нс любое функциональное устройство может выдавать результаты.

Оперативная память системы, выполненная на интегральных схемах, имеет емкость 1 млн. слов (позже была увеличена до 4 млн.) и организована в виде 16 блоков памяти с независимым управлением емкостью по 64 кслов. Каждый блок включает в себя 72 модуля, причем модуль содержит один разряд всех 64 кслов. Система работает с 64-разрядными словами, 8 разрядов используется для коррекции одиночных и обнаружения двойных ошибок, что обеспечивает высокую надежность хранения информации. Независимые блоки дают возможность организовать 16-кратное чередование адресов. Цикл обращения к памяти – 50 нс.

Существенную роль в достижении столь высокой производительности играют быстрые регистры. Они разделены на 3 группы: адресные – А-регистры, скалярные – S-регистры и векторные – V-регистры. Адресные регистры 24-разрядные, их всего восемь; 64-разрядных S-регистров также восемь и восемь 64-элементных V-регистров, причем каждый элемент вектора содержит 64-разрядное слово. Время обращения к регистру всего лишь 6 нс. В системе имеется еще две группы промежуточных регистров (между ОЗУ и А-, S- и V-регистрами): 24-разрядные B-регистры и 64-разрядные T-регистры, на рисунке не показанные. Все эти регистры позволяют конвейерным устройствам работать с максимальной скоростью без непосредственного обращения к ОЗУ: все операнды получаются из регистров и результаты отправляются также в регистры. Благодаря регистрам конвейерные устройства связываются в цепочки, т. е, поток результатов засылаемых в векторный регистр одним устройством, одновременно служит входным потоком операндов для другого устройства; исключаются промежуточные обращения к памяти. Это является еще одной отличительной особенностью системы CRAY, повышающей ее производительность.

Состав операций универсальный, только вместо деления используется операция вычисления обратной величины. Общее число операций 128. Команды двух форматов – 16 и 32 разряда. Арифметические и логические команды имеют 16-разрядный формат 7 разрядов – код операции и по 3 разряда для адресов регистров операндов и результата, причем 6 разрядов адресов регистров операндов в совокупности с дополнительными 16 разрядами используются для обращения к основной памяти и командам перехода.

Зацепление конвейеров.

По версии Томилина

Сколько здесь можно разогнать? Конечно, в основном разгонка идёт здесь. Частота у Cray была 80 МГц, это приблизительно 80 миллионов операций в секунду с плавающей запятой при выполнении векторных операций. Сколько операций можно запустить параллельно? Одна операция требует 3 векторных регистра, всего их 8, значит 2 операции. Таким образом, до 160 MFlops можно было раскрутить Cray-1. Это, конечно, очень серьёзное было приобретение в мировом вычислительном хозяйстве, и потому и названа была «супермашина».

На самом деле можно было больше. И это «больше» достигалось вот каким способом:

$$Q = \frac{(A+B) \cdot C}{D}$$

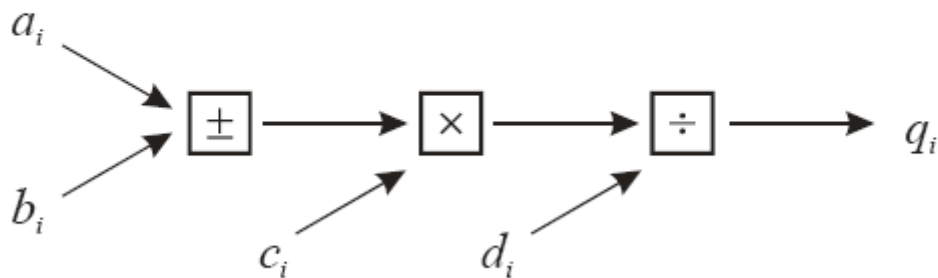


Рис. 3

Результат сложителя попадает не на регистр, а на умножитель. Туда же приходят c_i . Отсюда это идёт на делитель, куда подсовываются d_i . И отсюда выскакивает на векторный регистр Q . Итого, сколько нам нужно векторных регистров: 1 — вектор A , 2 — вектор B , 3 — вектор C , 4 — вектор D и 5 — вектор Q . Даже пяти хватает, чтобы сколько получить операций в секунду? Через 12 тактов появится q_1 и плюс ещё 63. Таким образом, через 75 тактов у вас появится результат скольких операции? $64 \text{ на } 3 = 192$ операций. Таким образом разогнали почти до 240 MFlops (чуть меньше). Но это ещё не предел. На Cray'е можно было в определённых случаях получать существенно больший результат. Вот это называется «зацепление работы векторных функциональных конвейерных устройств» или «супервекторная производительность», за счёт вот такой работы. Чуть сложнее, конечно, оборудование работает. Но вот это очень важный момент векторно-конвейерных машин и он тут реВалоит ээотвиа нд. в а новых уровня параллелизма (вертикальный — количество функциональных устройств, и горизонтальный — конвейер) и привело к высокой производительности машины, которую изобрёл Крэй, и это всё было почтительно показано, мировое достижение, и появилось слово «суперкомпьютер».

Из интернета

Векторная обработка увеличивает скорость и эффективность обработки за счет того, что обработка целого набора (вектора) данных выполняется одной командой. Скорость выполнения операций в векторном режиме приблизительно в 10 раз выше скорости скалярной обработки. Для фрагмента типа

```

Do i = 1, n
  A(i) = B(i)+C(i)
End Do
  
```

в скалярном режиме потребуется сгенерировать целую последовательность команд: прочитать элемент $B(I)$, прочитать элемент $C(I)$, выполнить сложение, записать результат в $A(I)$, увеличить параметр цикла, проверить условие цикла. В векторном режиме этот фрагмент преобразуется в: загрузить порцию массива B , загрузить порцию массива C (эти две операции будут выполняться со сдвигом в один такт, т.е. практически одновременно), векторное сложение, запись порции массива в память, если размер массивов больше длины векторных регистров, то повторить эту последовательность некоторое число раз.

Перед тем, как векторная операция начнет выдавать результаты, проходит некоторое время (startup), связанное с заполнением конвейера и подкачкой аргументов. Чем больше длина

векторов, тем менее заметным оказывается влияние данного начального промежутка времени на все время выполнения программы.

Векторные операции, использующие различные ФУ и регистры, могут выполняться параллельно.

Зацепление функциональных устройств

Архитектура CRAY Y-MP C90 позволяет использовать регистр результатов векторной операции в качестве входного регистра для последующей векторной операции, т.е. выход сразу подается на вход. Это называется зацеплением векторных операций. Вообще говоря, глубина зацепления может быть любой, например, чтение векторов, выполнение операции сложения, выполнение операции умножения, запись векторов.

Векторно-конвейерные вычислители.

Реализация команд организации цикла (счетчик и переход) при регулярной работе с данными - накладные расходы и препятствие опережающему просмотру на обычных, скалярных вычислителях, показывает, что такие вычисления эффективнее выполнять на специализированном векторно-конвейерном вычислителе.

Если вместо цикла:

DO L=1,N A(I) = B(I)+C(I) ENDDO

использовать запись алгоритма в виде векторной команды сложения вида:

VADD(B,C,A,N), то векторный вычислитель, выполняющий такие команды, будет вырабатывать результаты на каждом такте. В таком вычислителе имеется один (или небольшое число) конвейерный процессор, выполняющий векторные команды путем засылки элементов векторов в конвейер с интервалом, равным длительности прохождения одной стадии обработки. Скорость вычислений зависит только от длительности стадии и не зависит от задержек в процессоре в целом.

Так как конвейер для однотипных операций дешевле и быстрее чем для многофункциональных, то выгодно их делать специализированными - однофункциональным: например, только для + или только для *. Для их совместной работы используется принцип зацепления конвейеров. Так, в ЭВМ Крей-1 имеется 12 конвейеров, из них 8 могут быть зацеплены, то есть результаты вычисления конвейера могут входными аргументами для другого. Операнды (результаты) находятся в памяти верхнего уровня или на регистрах. Для операндов задается: базовый адрес вектора, число элементов, тип данных в каждом элементе, схема хранения вектора в памяти. Некоторые векторные машины могут работать с двух-трех мерными массивами.

Производительность конвейерных вычислителей

Время выполнения отдельной скалярной операции на конвейерном вычислителе равно: $T = S + K$, где K - время работы, за которое конвейер выдает очередной результат, а S - время запуска конвейера, время заполнения конвейера, которое без учета времени подготовки операндов, равно: $S = K \cdot (m-1)$, где m - число ступеней конвейера. Производительность конвейерного вычислителя на скалярных операциях (число результатов, выдаваемых за единицу времени) равна: $R = 1/(S + K)$.

Время выполнения векторной операции на конвейерном вычислителе равно: $T = S + K \cdot N$, где N - длина вектора. Производительность конвейерного вычислителя при векторной работе (число результатов, выдаваемых за единицу времени) равна: $R = N/(S + K \cdot N)$, асимптотическая производительность $R_{\infty} = 1/K$.

Например, при $K = 10$ нс, $R_{\infty} = 10^{-8}$ результатов/сек, т.е. 100 мегафлопов. Графики достижения такой производительности для $S = 100$ нс. и $S = 1000$ нс. показывают, что они

имеют различное расстояние до асимптоты. Для оценки этого эффекта используется величина $N1/2$, определяемая как длина вектора, для которой достигается половина асимптотической зависимости. Для приведенного выше примера $N1/2 = 100$ для $S = 1000$ и $N1/2 = 10$ для $S = 100$.

Дополнение

По мере освоения средств сверхскоростной обработки данных разрыв между совершенствованием методов векторизации программ, т.е. автоматического преобразования в процессе компиляции последовательных языковых конструкций в векторную форму, и чрезвычайной сложностью программирования коммутации и распределения данных между процессорными элементами привел к достаточно жесткой реакции пользователей в отношении матричных суперЭВМ - широкому кругу программистов требовалась более простая и "прозрачная" архитектура векторной обработки с возможностью использования стандартных языков высокого уровня типа FORTRAN. Решение было найдено в конце 60-х годов, когда фирма Control Data, с которой в то время сотрудничал Крей, представила машину STAR-100, основанную на векторно-конвейерном принципе обработки данных. Отличие векторно-конвейерной технологии от архитектуры матричных ЭВМ заключается в том, что вместо множества процессорных элементов, выполняющих одну и ту же команду над разными элементами вектора, применяется единственный конвейер операций, принцип действия которого полностью соответствует классическому конвейеру автомобильных заводов Форда. Даже такая архаичная по современным понятиям суперЭВМ, как STAR-100, показала предельную производительность на уровне 50 MFLOPS. При этом существенно, что векторно-конвейерные суперЭВМ значительно дешевле своих матричных "родственников". К примеру, разработка и производство ILLIAC IV обошлись в 40 млн. долл. при расходах на эксплуатацию порядка 2 млн. долл. в год, тогда как рыночная стоимость первых суперкомпьютеров фирм CRAY и Control Data находилась в пределах 10 - 15 млн. долл., в зависимости от объема памяти, состава периферийных устройств и других особенностей конфигурации системы.

Второй существенной особенностью векторно-конвейерной архитектуры является то, что конвейер операций имеет всего один вход, по которому поступают операнды, и один выход результата, тогда как в матричных системах существует множество входов по данным в процессорные элементы и множество выходов из них. Другими словами, в компьютерах с конвейерной обработкой данные всех параллельно исполняемых операций выбираются и записываются в единую память, в связи с чем отпадает необходимость в коммутаторе процессорных элементов, ставшем камнем преткновения при проектировании матричных суперЭВМ.

CISC и RISC архитектуры ЭВМ.

Двумя основными архитектурами набора команд, используемыми компьютерной промышленностью на современном этапе развития вычислительной техники (в соответствии с [2]) являются архитектуры CISC и RISC. Основоположником CISC-архитектуры можно считать компанию IBM с ее базовой архитектурой /360, ядро которой используется с 1964 года и дошло до наших дней, например, в таких современных мейнфреймах как IBM ES/9000. Лидером в разработке микропроцессоров с полным набором команд (CISC – Complete Instruction Set Computer) считается компания Intel со своей серией x86 и Pentium. Эта архитектура является практическим стандартом для рынка микрокомпьютеров. Для CISC-процессоров характерно: сравнительно небольшое число регистров общего назначения; большое количество машинных команд, некоторые

из которых нагружены семантически аналогично операторам высокоуровневых языков программирования и выполняются за много тактов; большое количество методов адресации; большое количество форматов команд различной разрядности; преобладание двухадресного формата команд; наличие команд обработки типа регистр-память.

Основой архитектуры современных рабочих станций и серверов является архитектура компьютера с сокращенным набором команд (RISC – Reduced Instruction Set Computer). Зачатки этой архитектуры уходят своими корнями к компьютерам CDC6600, разработчики которых (Торнтон, Крэй и др.) осознали важность упрощения набора команд для построения быстрых вычислительных машин. Эту традицию упрощения архитектуры С. Крэй с успехом применил при создании широко известной серии суперкомпьютеров компании Cray Research. Однако окончательно понятие RISC в современном его понимании сформировалось на базе трех исследовательских проектов компьютеров: процессора 801 компании IBM, процессора RISC университета Беркли и процессора MIPS Стенфордского университета.

Среди других особенностей RISC-архитектур следует отметить наличие достаточно большого регистрового файла (в типовых RISC-процессорах реализуются 32 или большее число регистров по сравнению с 8 – 16 регистрами в CISC-архитектурах), что позволяет большому объему данных храниться в регистрах на процессорном кристалле большее время и упрощает работу компилятора по распределению регистров под переменные.

Для обработки, как правило, используются трехадресные команды, что помимо упрощения дешифрации дает возможность сохранять большее число переменных в регистрах без их последующей перезагрузки.

Развитие архитектуры RISC в значительной степени определялось прогрессом в области создания оптимизирующих компиляторов. Именно современная техника компиляции позволяет эффективно использовать преимущества большего регистрового файла, конвейерной организации и большей скорости выполнения команд. Современные компиляторы используют также преимущества другой оптимизационной техники для повышения производительности, обычно применяемой в процессорах RISC: реализацию задержанных переходов и суперскалярной обработки, позволяющей в один и тот же момент времени выдавать на выполнение несколько команд.

Следует отметить, что в последних разработках компании Intel (имеются в виду Pentium и Pentium Pro), а также ее последователей-конкурентов (AMD R5, Cyrix M1, NexGen Nx586 и др.) широко используются идеи, реализованные в RISC-микропроцессорах, так что многие различия между CISC и RISC стираются. Однако сложность архитектуры и системы команд x86 остается и является главным фактором, ограничивающим производительность процессоров на ее основе.

Внеочередное и спекулятивное выполнения команд.

Внеочередное выполнение команд

Методы динамической оптимизации: неупорядоченное выполнение “out-of-order execution”, неупорядоченная выдача “out-of-order issue” основаны на изменении порядка вычислений. Так, пример вычислений из предыдущего раздела после его преобразования к виду:

$$A1 = B1 + C1$$

$$A3 = B3 + C3 \quad A3 = B3 + C3$$

$$A4 = B4 + C4 \quad A4 = B4 + C4$$

$$A2 = A1 + C2 \quad A2 = (B1 + C1) + C2$$

позволит проводить вычисления без пропуска рабочих тактов для ожидания вычисления. Оптимизационные преобразования последовательности вычисления могут проводиться динамически, аппаратурой АЛУ, а также, в ряде случаев и статически проводиться системами программирования.

14. Спекулятивное выполнение команд

В конвейерных архитектурах устройство выборки команд является таким же конвейером, как и другие функциональные устройства. Так, для условного оператора: IF (A<B) GO TO L;S1;L:S2 еще до вычисления значения условного выражения A<B необходимо решать задачу о заполнении конвейера командами S1 или S2 – спекулятивного выполнения программы (чтобы не было пропуска тактов конвейера из-за неверно выбранной ветки, коды которой потребуются убирать из конвейера).

Механизмы предсказания переходов.

Тривиальное решение состоит в выборе кода, текстуально следующего за командой условного перехода. Для такого оборудования компиляторы могут формировать объектный код с размещением наиболее вероятно выполняемым фрагменте программы непосредственно за командой условного перехода. Так, для циклических конструкций, вероятность перехода на повторение цикла выше вероятности выхода из него. Некоторые системы программирования дают возможность программисту указывать вероятность перехода по метке в условном переходе.

Аппаратный механизм учета вероятности перехода состоит из блока предсказания переходов. Этот блок, кроме (вместо) статически определенных предпочтений для ветвлений, имеет таблицу переходов, в которой хранится история переходов для каждого (в рамках объема таблицы) перехода программы. Большинство современных микропроцессоров обещают точность предсказаний переходов этим способом выше 90%. Причина повышенного внимания к этому вопросу обусловлена большими задержками, возникающими при неверном предсказании переходов, что грозит существенной потерей производительности. Используемые в микропроцессорах методы предсказания переходов, как уже было сказано, бывают статические и динамические. Как динамический, так и статический подходы имеют свои преимущества и недостатки.

Статические методы предсказания используются реже. Такие предсказания делаются компилятором еще до момента выполнения программы. Соответствующие действия компилятора можно считать оптимизацией программ. Такая оптимизация может основываться на сборе информации, получаемой при тестовом прогоне программы (Profile Based Optimisation, PBO) или на эвристических оценках. Результатом деятельности компилятора являются "советы о направлении перехода", помещаемые непосредственно в коды выполняемой программы. Эти советы использует затем аппаратура во время выполнения. В случае, когда переход происходит, или наоборот - как правило, не происходит, советы компилятора часто бывают весьма точны, что ведет к отличным результатам. Преимущество статического подхода - отсутствие необходимости интегрировать на чипе дополнительную аппаратуру предсказания переходов.

Большинство производителей современных микропроцессоров снабжают их различными средствами динамического предсказания переходов, производимого на базе анализа "предыстории". Тогда аппаратура собирает статистику переходов, которая помещается в таблицу истории переходов ВНТ (Branch History Table). В обоих случаях компилятор не может выработать эффективные рекомендации на этапе трансляции программы. В то же время схемы динамического предсказания переходов легко справляются с такими задачами. В этом смысле динамическое предсказание переходов "мощнее" статического. Однако у

динамического предсказания есть и свои слабые места - это проблемы, возникающие из-за ограниченности ресурсов для сбора статистики .

Буфера прогнозирования условных переходов

Простейшей схемой динамического прогнозирования направления условных переходов является буфер прогнозирования условных переходов (branch-prediction buffer) или таблица "истории" условных переходов (branch history table). Буфер прогнозирования условных переходов представляет собой небольшую память, адресуемую с помощью младших разрядов адреса команды перехода. Каждая ячейка этой памяти содержит один бит, который говорит о том, был ли предыдущий переход выполняемым или нет. Это простейший вид такого рода буфера. В нем отсутствуют теги, и он оказывается полезным только для сокращения задержки перехода в случае, если эта задержка больше, чем время, необходимое для вычисления значения целевого адреса перехода. В действительности мы не знаем, является ли прогноз корректным (этот бит в соответствующую ячейку буфера могла установить совсем другая команда перехода, которая имела то же самое значение младших разрядов адреса). Но это не имеет значения. Прогноз - это только предположение, которое рассматривается как корректное, и выборка команд начинается по прогнозируемому направлению. Если же предположение окажется неверным, бит прогноза инвертируется. Конечно такой буфер можно рассматривать как кэш-память, каждое обращение к которой является попаданием, и производительность буфера зависит от того, насколько часто прогноз применялся и насколько он оказался точным.

Однако простая однобитовая схема прогноза имеет недостаточную производительность. Рассмотрим, например, команду условного перехода в цикле, которая являлась выполняемым переходом последовательно девять раз подряд, а затем однажды невыполняемым. Направление перехода будет неправильно предсказываться при первой и при последней итерации цикла. Неправильный прогноз последней итерации цикла неизбежен, поскольку бит прогноза будет говорить, что переход "выполняемый" (переход был девять раз подряд выполняемым). Неправильный прогноз на первой итерации происходит из-за того, что бит прогноза инвертируется при предыдущем выполнении последней итерации цикла, поскольку в этой итерации переход был невыполняемым. Таким образом, точность прогноза для перехода, который выполнялся в 90% случаев, составила только 80% (2 некорректных прогноза и 8 корректных). В общем случае, для команд условного перехода, используемых для организации циклов, переход является выполняемым много раз подряд, а затем один раз оказывается невыполняемым. Поэтому однобитовая схема прогнозирования будет неправильно предсказывать направление перехода дважды (при первой и при последней итерации).

Для исправления этого положения часто используется схема двухбитового прогноза. В двухбитовой схеме прогноз должен быть сделан неверно дважды, прежде чем он изменится на противоположное значение. На рисунке 5.27 представлена диаграмма состояний двухбитовой схемы прогнозирования направления перехода.

Двухбитовая схема прогнозирования в действительности является частным случаем более общей схемы, которая в каждой строке буфера прогнозирования имеет n -битовый счетчик. Этот счетчик может принимать значения от 0 до $2n - 1$. Тогда схема прогноза будет следующей:

Если значение счетчика больше или равно $2n-1$ (точка на середине интервала), то переход прогнозируется как выполняемый. Если направление перехода предсказано правильно, к значению счетчика добавляется единица (если только оно не достигло максимальной величины); если прогноз был неверным, из значения счетчика вычитается единица.

Если значение счетчика меньше, чем $2n-1$, то переход прогнозируется как невыполняемый. Если направление перехода предсказано правильно, из значения счетчика вычитается единица (если только не достигнуто значение 0); если прогноз был неверным, к значению счетчика добавляется единица.

Исследования n -битовых схем прогнозирования показали, что двухбитовая схема работает почти также хорошо, и поэтому в большинстве систем применяются двухбитовые схемы прогноза, а не n -битовые.

Буфер прогнозирования переходов может быть реализован в виде небольшой специальной кэш-памяти, доступ к которой осуществляется с помощью адреса команды во время стадии выборки команды в конвейере (IF), или как пара битов, связанных с каждым блоком кэш-памяти команд и выбираемых с каждой командой. Если команда декодируется как команда перехода, и если переход спрогнозирован как выполняемый, выборка команд начинается с целевого адреса как только станет известным новое значение счетчика команд. В противном случае продолжается последовательная выборка и выполнение команд. Если прогноз оказался неверным, значение битов прогноза меняется в соответствии с рисунком 5.27. Хотя эта схема полезна для большинства конвейеров, рассмотренный нами простейший конвейер выясняет примерно за одно и то же время оба вопроса: является ли переход выполняемым и каков целевой адрес перехода (предполагается отсутствие конфликта при обращении к регистру, определенному в команде условного перехода. Напомним, что для простейшего конвейера это справедливо, поскольку условный переход выполняет сравнение содержимого регистра с нулем во время стадии ID, во время которой вычисляется также и эффективный адрес). Таким образом, эта схема не помогает в случае простых конвейеров, подобных рассмотренному ранее.

Как уже упоминалось, точность двухбитовой схемы прогнозирования зависит от того, насколько часто прогноз каждого перехода является правильным и насколько часто строка в буфере прогнозирования соответствует выполняемой команде перехода. Если строка не соответствует данной команде перехода, прогноз в любом случае делается, поскольку все равно никакая другая информация не доступна. Даже если эта строка соответствует совсем другой команде перехода, прогноз может быть удачным.

Управление виртуальной памятью.

Виртуальная память: основные концепции

Суть концепции виртуальной памяти заключается в том, что адреса, к которым обращается выполняющийся процесс, отделяются от адресов, реально существующих в первичной памяти.

Те адреса, на которые делает ссылки выполняющийся процесс, называются виртуальными адресами, а те адреса, которые существуют в первичной памяти, называются реальными (или физическими) адресами. Диапазон виртуальных адресов, к которым может обращаться выполняющийся процесс, называется пространством виртуальных адресов V этого процесса. Диапазон реальных адресов, существующих в конкретной вычислительной машине, называется пространством реальных адресов R этого компьютера.

Несмотря на то, что процессы обращаются только к виртуальным адресам, в действительности они должны работать с реальной памятью. Таким образом, во время выполнения процесса виртуальные адреса необходимо преобразовывать в реальные, причем это нужно делать быстро, ибо в противном случае производительность вычислительной машины будет снижаться до неприемлемых уровней и тем самым практически сведется на

нет те преимущества, которые и призвана обеспечить прежде всего концепция виртуальной памяти (рис. 8.2).

Для установления соответствия между виртуальными и реальными адресами разработаны различные способы. Так, механизмы динамического преобразования адресов (DAT) обеспечивают преобразование виртуальных адресов в реальные во время выполнения процесса. Все подобные системы обладают общим свойством: смежные адреса виртуального адресного пространства процесса не обязательно будут смежными в реальной памяти, это свойство называют «искусственной смежностью» (рис. 8.3). Таким образом, пользователь освобождается от необходимости учитывать размещение своих процедур и данных в реальной памяти. Он получает возможность писать программы наиболее естественным образом, прорабатывая только детали алгоритма и структуры программы и игнорируя конкретные особенности структуры аппаратных средств, служащих для выполнения программы. При этом компьютер рассматривается (или может рассматриваться) только как логическое средство, обеспечивающее реализацию необходимых алгоритмов, а не как физическая машина с уникальными характеристиками, часть которых может лишь затруднить процесс проектирования программы.

8.4 Многоуровневая организация памяти

Если мы предусматриваем, что пространство виртуальных адресов пользователя будет больше пространства реальных адресов, и, естественно, если мы ориентируемся на то, что система будет эффективно работать в мультипрограммном режиме с совместным использованием (разделением) ресурсов реальной памяти многими пользователями, то мы должны обеспечить средства хранения программ и данных в большой вспомогательной памяти. Обычно для этого применяется двухуровневая схема построения памяти (рис. 8.4). Первый уровень — это реальная память, в которой находятся выполняемые процессы и в которой должны размещаться данные, чтобы процесс во время работы мог к ним обращаться.

Второй уровень — это внешняя память большой емкости, например диски или барабаны, способные хранить программы и данные, которые не могут все сразу уместиться в реальной памяти ограниченной емкости. Память этого второго уровня, как правило, называется вторичной, или внешней. Чтобы обеспечить возможность выполнения процесса, его код и данные вводятся в основную память. В настоящей и следующей главах будет подробно описано, каким образом это делается.

Поскольку реальная память разделяется между многими процессами и поскольку каждый процесс может иметь гораздо большее пространство виртуальных адресов, чем реальная память, то в текущий момент времени в реальной памяти имеется возможность держать лишь небольшую часть программных кодов и данных каждого процесса. На рис. 8.5 показана двухуровневая система памяти, в которой реальная память содержит лишь определенные элементы из виртуальных памяти различных пользователей.

8.6 Страничная организация: основные концепции

Памятуя о том, насколько сложнее обращаться с блоками переменных размеров при мультипрограммировании переменными разделами, давайте начнем с рассмотрения поблочного отображения

Номер страницы p

Смещение d

Виртуальный адрес $V=(p,d)$

при фиксированном размере блока, т. е. со страничной организацией памяти. В настоящем разделе мы будем рассматривать чисто страничную, а не странично-сегментную организацию.

Виртуальный адрес в страничной системе — это упорядоченная пара (p, d) , где p — номер страницы в виртуальной памяти, а d — смещение в рамках страницы p , где размещается адресуемый элемент (рис. 8.8). Процесс может выполняться, если его текущая страница находится в первичной памяти. Страницы переписываются из внешней памяти в первичную и размещаются в ней в блоках, называемых страничными кадрами и имеющих точно такой же размер, как поступающие страницы. Страничные кадры начинаются с адресов первичной памяти, кратных фиксированному размеру страницы (рис. 8.9). Поступающая страница может быть помещена в любой свободный страничный кадр.

Динамическое преобразование адресов в системе со страничной организацией осуществляется следующим образом. Выполняющийся процесс обращается по адресу виртуальной памяти $V=(p, d)$. Механизм отображения страниц, показанный на рис. 8.10, ищет номер страницы p в таблице отображения страниц и определяет, что эта страница находится в страничном кадре p' . Адрес реальной памяти формируется затем путем конкатенации p' и d .

Рассмотрим теперь этот процесс более подробно. В частности, поскольку обычно не все страницы процесса находятся в первичной памяти одновременно, таблица отображения страниц должна указывать, есть ли адресуемая страница в первичной памяти, и если есть, то где именно, а если нет, то в каком месте внешней памяти ее можно найти. На рис. 8.11 показана типичная строка таблицы отображения страниц. Бит-признак присутствия страницы g устанавливается в 0, если данной страницы в первичной памяти нет, и в 1, если эта страница есть в первичной памяти. Если страницы нет, то s — ее адрес во внешней памяти. Если страница есть в первичной памяти, то p' — номер ее страничного кадра. Отметим, что p' — это не действительный адрес первичной памяти. Адрес первичной памяти a , с которого начинается страничный кадр p' (если размер страницы равняется p), определяется произведением

$a=(p)(p')$ (если страничные кадры имеют номера 0, 1, 2 и т. д.).

В целях экономии емкости первичной памяти в таблицу отображения страниц можно не включать адреса страниц во внешней памяти.

В этом и нескольких следующих разделах мы рассмотрим несколько способов преобразования адресов страниц. Рассмотрим вначале прямое отображение, которое иллюстрирует рис. 8.12.

Выполняющийся процесс делает ссылку по виртуальному адресу $V=(p, d)$. Прежде чем процесс начнет выполняться, операционная система загружает адрес первичной памяти для таблицы отображения страниц в регистр начального адреса этой таблицы. Этот базовый адрес b таблицы прибавляется к номеру страницы p , образуя адрес первичной памяти $b+p$ для строки страницы p в таблице. В этой строке указано, что виртуальной странице p соответствует страничный кадр p' . Затем к значению p' приформировывается (путем конкатенации) смещение d , так что образуется реальный адрес a . Такой подход называется способом прямого отображения потому, что таблица отображения страниц содержит отдельную строку для каждой страницы виртуальной памяти данного процесса. Если процесс имеет в виртуальной памяти n страниц, то таблица при прямом отображении для этого процесса содержит последовательные строки для страницы 0, страницы 1, страницы 2, ..., страницы $n-1$.

Как преобразуемый виртуальный адрес, так и базовый адрес таблицы отображения страниц хранятся в высокоскоростных регистрах управляющего процессора, так что операции с их участием можно выполнять очень быстро внутри командного цикла. Однако таблица прямого отображения страниц, которая может быть довольно большой, обычно находится в первичной памяти. Следовательно, для обращения к этой таблице требуется один полный

цикл выборки из первичной памяти. Поскольку время выборки из первичной памяти составляет, как правило, наибольшую долю цикла выполнения команды и поскольку нам требуется еще один цикл обращения к первичной памяти для выборки соответствующей строки таблицы, это означает, что использование метода прямого отображения для преобразования адресов страниц может вызвать снижение скорости вычислительной машины при выполнении программ почти вдвое! Это, естественно, недопустимо. Поэтому необходимы более высокоскоростные способы преобразования адресов. В то же время это не означает, что способ прямого отображения совершенно неприемлем, например некоторые системы с успехом его реализуют, размещая целиком всю таблицу в сверхвысокоскоростной кэш-памяти.

8.7 Сегментная организация

В главе, посвященной реальной памяти, мы говорили о том, что в системах мультипрограммирования с переменными разделами размещение программ в памяти чаще всего осуществляется в соответствии с выбором «первого подходящего», «наиболее подходящего» или «наименее подходящего» по размеру свободного участка памяти. Однако мы все же были ограничены необходимостью выполнять программы в одном блоке смежных ячеек реальной памяти.

В системах с сегментной организацией (сегментацией) это ограничение снимается, и программа (и ее данные) может занимать много отдельных блоков первичной памяти (рис. 8.16). Сами блоки не обязательно одинакового размера, все же должны состоять из смежных ячеек, однако отдельные блоки могут размещаться и не рядом друг с другом.

При таком подходе возникает несколько интересных проблем. Например, гораздо более сложной теперь становится проблема защиты программы каждого пользователя от несанкционированного доступа со стороны других пользователей. Парой граничных регистров здесь больше не обойтись. Аналогично становится гораздо труднее ограничивать допустимую область доступа для любой конкретной программы. Один из способов реализации защиты памяти в системах с сегментной организацией — это использование ключей защиты, как показано на рис. 8.17.

Рис. 8.17 Защита памяти при помощи ключей в мультипрограммных системах с распределением памяти несмежными блоками. Когда в ЦП для ключа защиты памяти устанавливается значение 2, соответствующее пользователю В, программа этого пользователя может обращаться только к тем блокам памяти, которые имеют именно этот ключ защиты (2). Управление ключами защиты осуществляет только операционная система.

Виртуальный адрес в сегментной системе — это упорядоченная пара $V=(s, d)$, где s — номер сегмента виртуальной памяти, а d — смещение в рамках этого сегмента, где находится адресуемый элемент (рис. 8.18). Процесс может выполняться только в случае, если его текущий сегмент (как минимум) размещается в первичной памяти. Сегменты передаются из внешней памяти в первичную целиком. Все ячейки, относящиеся к сегменту, занимают смежные адреса первичной памяти. Поступающий из внешней памяти сегмент может

Номер сегмента s

Смещение d

Виртуальный адрес $V=(s,d)$

быть помещен в любой достаточно большой по размеру свободный блок смежных ячеек первичной памяти. Стратегии размещения при сегментации идентичны стратегиям,

применяемым при мультипрограммировании с переменными разделами, причем наиболее часто используются принципы «первый подходящий» и «наиболее подходящий».

Динамическое преобразование адресов осуществляется следующим образом (рис. 8.19). Выполняющийся процесс выдает адрес виртуальной памяти $V=(s, d)$. Механизм отображения сегментов ищет сегмент s в таблице сегментов и определяет, что этот сегмент находится в реальной памяти, начиная с ячейки s' . Адрес реальной памяти, соответствующий виртуальному адресу $V=(s, d)$, формируется затем путем добавления s' к d . Более подробно эту процедуру мы рассмотрим в нескольких следующих разделах.

8.7.1 Управление доступом в системах с сегментной организацией

Предоставлять каждому процессу неограниченный доступ к любому сегменту в системе нецелесообразно. В действительности одним из достоинств сегментных систем является именно возможность четкого управления доступом. Для этого каждому процессу предоставляются определенные права доступа к каждому сегменту и фактически полностью запрещается доступ ко многим сегментам.

8.8 Системы с комбинированной странично-сегментной организацией

И сегментная, и страничная организации имеют важные достоинства как способы построения виртуальной памяти. Начиная с систем середины 60-х годов, в частности с информационно-вычислительной системы с разделением времени Multics и системы разделения времени TSS корпорации IBM, во многих вычислительных машинах применяется комбинированная странично-сегментная организация памяти (Da68, De71, Do76). Эти системы обладают достоинствами обоих способов реализации виртуальной памяти. Сегменты обычно содержат целое число страниц, причем не обязательно, чтобы все страницы сегмента находились в первичной памяти одновременно, а смежные страницы виртуальной памяти не обязательно должны оказываться смежными в реальной памяти. В системе со странично-сегментной организацией применяется трехкомпонентная (трехмерная) адресация, т. е. адрес виртуальной памяти V определяется как упорядоченная тройка $V=(s, p, d)$, где s — номер сегмента, p — номер страницы, а d — смещение в рамках страницы, по которому находится нужный элемент (рис. 8.24).

Рассмотрим теперь динамическое преобразование виртуальных адресов в реальные в странично-сегментной системе с применением комбинированного ассоциативно-прямого отображения, как показано на рис. 8.25.

Выполняющийся процесс делает ссылку по виртуальному адресу $V=(s, p, d)$. Самые последние по времени обращения страницы имеют соответствующие строки в ассоциативной таблице. Система производит ассоциативный поиск, пытаясь найти строку с параметрами (s, p) в ассоциативной таблице. Если такая строка здесь обнаруживается, то адрес страничного кадра p' , по которому эта страница размещается в первичной памяти, соединяется со смещением d , образуя реальный адрес r , соответствующий виртуальному адресу V , — и на этом преобразование адреса завершается.

В обычном случае большинство запросов на преобразование адресов удастся удовлетворить подобным ассоциативным поиском. Если же требуемого адреса в ассоциативной памяти нет, то преобразование осуществляется способом полного прямого отображения. Это делается следующим образом: базовый адрес b таблицы сегментов прибавляется к номеру сегмента s , так что образуется адрес $b+s$ строки для сегмента s в таблице сегментов по первичной памяти. В этой строке указывается базовый адрес s' таблицы страниц для сегмента s . Номер страницы p прибавляется к s' , так что образуется адрес $p+s'$ строки в таблице страниц для страницы p сегмента s . Эта таблица позволяет установить, что виртуальной странице p соответствует номер кадра p' . Этот номер кадра соединяется со

смещением d , так что образуется реальный адрес r , соответствующий виртуальному адресу $V=(s, p, d)$.

Эта процедура преобразования адресов описана, конечно, в предположении, что каждый необходимый элемент информации находится именно там, где ему положено быть. Однако в процессе преобразования адресов существует много моментов, когда обстоятельства могут складываться не столь благоприятно. Просмотр таблицы сегментов может показать, что сегмента s в первичной памяти нет; при этом возникает прерывание по отсутствию сегмента, операционная система найдет нужный сегмент во внешней памяти, сформирует для него таблицу страниц и загрузит соответствующую страницу в первичную память, быть может, вместо некоторой существующей страницы этого или какого-либо другого процесса. Когда сегмент находится в первичной памяти, обращение к таблице страниц может показать, что нужной страницы в первичной памяти все же нет. При этом произойдет прерывание по отсутствию страницы, операционная система возьмет управление на себя, найдет данную страницу во внешней памяти и загрузит ее в первичную память (быть может, опять-таки с замещением другой страницы). Как и при чисто сегментной системе, адрес виртуальной памяти может выйти за рамки сегмента и произойдет прерывание по выходу за пределы сегмента. Или контроль по битам-признакам защиты может выявить, что операция, выполнение которой запрашивается по указанному виртуальному адресу, не разрешена, вследствие чего произойдет прерывание по защите сегмента. Операционная система должна предусматривать обработку всех этих ситуаций.

Ассоциативная память (или аналогично высокоскоростная кэш-память) играет решающую роль в обеспечении эффективной работы механизма динамического преобразования адресов. Если преобразование адресов производить с помощью чисто прямого отображения с ведением в первичной памяти полного набора таблиц соответствия виртуальных и реальных адресов, типичное обращение к виртуальной памяти потребовало бы цикла памяти для доступа к таблице сегментов, второго цикла памяти для доступа к таблице страниц и третьего для доступа к нужному элементу в реальной памяти. Таким образом, каждое обращение к адресуемому элементу занимало бы три цикла памяти, т. е. реальное быстродействие вычислительной системы составило бы лишь приблизительно треть от номинального значения, а две трети затрачивалось бы на преобразование адресов! Интересно, что при всего лишь восьми или шестнадцати ассоциативных регистрах разработчики добиваются для многих систем показателей быстродействия, составляющих 90 и более процентов полных скоростных возможностей их управляющих (центральных) процессоров.

На рис. 8.26 показана подробная структура таблиц, необходимых для комбинированной странично-сегментной системы. На самом верхнем уровне этой структуры находится таблица процессов, которая содержит по строке для каждого процесса, известного системе. Строка таблицы процессов, соответствующая конкретному процессу, указывает на таблицу сегментов этого процесса. Каждая строка таблицы сегментов процесса указывает на таблицу страниц соответствующего сегмента, а каждая строка таблицы страниц указывает либо на страничный кадр, в котором размещается данная страница, либо на адрес внешней памяти, где можно найти эту страницу. В системе с большим количеством процессов, сегментов и страниц вся эта табличная структура может занимать значительную долю первичной памяти. Здесь следует учитывать, что преобразование адресов производится во время выполнения быстрее, если все таблицы находятся в первичной памяти. Однако чем больше таблиц в памяти, тем меньшее количество процессов система в состоянии поддерживать в текущий период, а это может приводить к снижению пропускной способности. Разработчики операционных систем должны анализировать многие подобные взаимозависимости и принимать тщательно сбалансированные компромиссные решения —

только в этом случае можно обеспечить и эффективную загрузку системы, и оперативное предоставление ее услуг пользователям.

9.2 Стратегии управления виртуальной памятью

Стратегии вталкивания. Их цель - определить, в какой момент следует переписать страницу или сегмент из вторичной памяти в первичную. Вталкивание по запросу (по требованию) предполагает, что система ждет ссылки на страницу или сегмент от выполняющего процесса и только после появления ссылки начинает переписывать данную страницу или сегмент в первичную память. Вталкивание с упреждением (опережением) предполагает, что система пытается заблаговременно определить, к каким страницам или сегментам будет обращаться процесс. Если вероятность обращения высока и в первичной памяти имеется свободное место, то соответствующие страницы или сегменты будут переписываться в основную память еще до того, как к ним будет явно производиться обращение.

Стратегии размещения. Их цель — определить, в какое место первичной памяти помещать поступающую страницу или сегмент. В системах со страничной организацией решение о размещении принимается достаточно тривиально, поскольку поступающая страница может быть помещена в любой свободный страничный кадр. Системы с сегментной организацией требуют стратегий размещения, аналогичных тем, которые мы обсуждали применительно к системам мультипрограммирования с переменными разделами.

Стратегии выталкивания. Их цель — решить, какую страницу или сегмент следует удалить из первичной памяти, чтобы освободить место для помещения поступающей страницы или сегмента, если первичная память полностью занята.

9.3 Стратегии выталкивания страниц

В системах со страничной организацией все страничные кадры бывают, как правило, заняты. В этом случае программы управления памятью, входящие в операционную систему, должны решать, какую страницу следует удалить из первичной памяти, чтобы освободить место для поступающей страницы. Мы рассмотрим следующие стратегии выталкивания страниц.

- Принцип оптимальности.
- Выталкивание случайной страницы.
- Первой выталкивается первая пришедшая страница (FIFO).
- Первой выталкивается дольше всего не использовавшаяся страница (LRU).
- Первой выталкивается наименее часто использовавшаяся страница (LFU).
- Первой выталкивается не использовавшаяся в последнее время страница (NUR).
- Рабочее множество.

9.3.1 Принцип оптимальности

Принцип оптимальности (De70) говорит о том, что для обеспечения оптимальных скоростных характеристик и эффективного использования ресурсов следует заменять ту страницу, к которой в дальнейшем не будет новых обращений в течение наиболее длительного времени. Можно, конечно, продемонстрировать, что подобная стратегия действительно оптимальна, однако реализовать ее, естественно, нельзя, поскольку мы не умеем предсказывать будущее.

В связи с этим для обеспечения высоких скоростных характеристик и эффективного использования ресурсов мы попытаемся наиболее близко подойти к принципу оптимальности, применяя различные методы выталкивания страниц, приближающиеся к оптимальному.

9.3.2 Выталкивание случайной страницы

Если нам нужно иметь стратегию выталкивания страниц, которая характеризовалась бы малыми издержками и не являлась бы дискриминационной по отношению к каким-либо конкретным пользователям, то можно пойти по очень простому пути - выбирать случайную страницу. В этом случае все страницы, находящиеся в основной памяти, могут быть выбраны для выталкивания с равной вероятностью, в том числе даже следующая страница, к которой будет производиться обращение (и которую, естественно, удалять из памяти наиболее нецелесообразно). Поскольку подобная стратегия по сути как бы рассчитана на «слепое» везение и похожа на подход «пан или пропал», в реальных системах она применяется редко.

9.3.3 Выталкивание первой пришедшей страницы (FIFO)

При выталкивании страниц по принципу FIFO мы присваиваем каждой странице в момент поступления в основную память временную метку. Когда появляется необходимость удалить из основной памяти какую-нибудь страницу, мы выбираем ту, которая находилась в памяти дольше других. Интуитивный аргумент в пользу подобной стратегии кажется весьма весомым, а именно: у данной страницы уже были возможности «использовать свой шанс», и пора дать подобные возможности и другой странице. К сожалению, стратегия FIFO с достаточно большой вероятностью будет приводить к замещению активно используемых страниц, поскольку тот факт, что страница находится в основной памяти в течение длительного времени, вполне может означать, что она постоянно в работе. Например, для крупных систем разделения времени стандартна ситуация, когда многие пользователи во время ввода и отработки

своих программ совместно используют одну копию текстового редактора. Если в подобной системе выталкивать страницы по принципу FIFO, это может привести к удалению из памяти какой-либо интенсивно используемой страницы редактора. А это будет безусловно нецелесообразно, поскольку ее почти немедленно придется снова переписывать в основную память.

Однако, как установили Билейди, Нельсон и Шедлер (Be69b), в стратегии FIFO определенные последовательности обращений к страницам приводят в действительности к увеличению количества прерываний по отсутствию страницы при увеличении количества страничных кадров, выделяемых процессу. Это явление носит название «аномалии FIFO».

9.3.4 Выталкивание дольше всего не использовавшейся страницы (LRU)

Эта стратегия предусматривает, что для выталкивания следует выбирать ту страницу, которая не использовалась дольше других. Здесь мы исходим из эвристического правила, говорящего о том, что недавнее прошлое — хороший ориентир для прогнозирования ближайшего будущего. Стратегия LRU требует, чтобы при каждом обращении к странице ее временная метка обновлялась. Это может быть сопряжено с существенными издержками, и поэтому стратегия LRU, хотя она и кажется весьма привлекательной, в современных системах реализуется редко. Чаще применяются близкие к LRU стратегии, для которых характерны меньшие издержки.

Разработчики операционных систем должны всегда с большой осторожностью применять эвристические правила и рассуждения. Например, при реализации стратегии LRU может быть так, что страница, к которой дольше всего не было обращений, в действительности станет следующей используемой страницей, если программа к этому моменту очередной раз пройдет большой цикл, охватывающий несколько страниц. Таким образом, выталкивая

страницу, к которой дольше всего не было обращений, мы можем оказаться вынужденными почти немедленно возвращать ее обратно.

9.3.5 Выталкивание реже всего используемой страницы (LFU)

Одной из близких к LRU стратегий является стратегия, согласно которой выталкивается наименее часто (наименее интенсивно) использовавшаяся страница (LFU). Здесь мы контролируем интенсивность использования каждой страницы. Выталкивается та страница, которая наименее интенсивно используется или обращения к которой наименее часты. Подобный подход опять-таки кажется интуитивно оправданным, однако в то же время велика вероятность того, что удаляемая страница будет выбрана нерационально. Например, наименее интенсивно используемой может оказаться та страница, которую только что переписали в основную память и к которой успели обратиться только один раз, в то время как к другим страницам могли уже обращаться более одного раза. Теперь работающий по принципу LFU механизм вытолкнет эту страницу, а она скорее всего сразу же будет использоваться.

Таким образом, практически любой метод выталкивания страниц, по-видимому, не исключает опасности принятия нерациональных решений. Это действительно так просто потому, что мы не можем достаточно точно прогнозировать будущее. В связи с этим необходима такая стратегия выталкивания страниц, которая обеспечивала бы принятие рациональных решений в большинстве случаев и в то же время не требовала больших накладных расходов.

9.3.6 Выталкивание не использовавшейся в последнее время страницы (NUR)

Один из распространенных алгоритмов, близких к стратегии LRU и характеризующихся малыми издержками, — это алгоритм выталкивания страницы, не использовавшейся в последнее время (NUR); к страницам, которые в последнее время не использовались, вряд ли будут обращения и в ближайшем будущем, так что их можно заменять на вновь поступающие страницы.

Поскольку желательно заменять ту страницу, которая в период нахождения в основной памяти не изменялась, реализация стратегии NUR предусматривает введение двух аппаратных битов-признаков на страницу. Это

а) бит-признак обращения

= 0, если к странице не было обращений;

= 1, если к странице были обращения;

б) бит-признак модификации

= 0, если страница не изменялась;

= 1, если страница изменялась.

Бит-признак модификации часто называют также «признаком записи» в страницу. Стратегия NUR реализуется следующим образом. Первоначально биты-признаки обращения и модификации для всех страниц устанавливаются в 0. При обращении к какой-либо странице ее бит-признак обращения устанавливается в 1, а в случае изменения содержимого страницы устанавливается в 1 ее бит-признак модификации. Когда нужно выбрать страницу для выталкивания, прежде всего мы пытаемся найти такую страницу, к которой не было обращений (поскольку мы стремимся приблизиться к алгоритму LRU). В противном случае у нас не будет другого выхода, как вытолкнуть страницу, к которой были обращения. Если к странице обращения были, мы проверяем, подверглась ли она изменению или нет. Если нет, мы заменяем ее из тех соображений, что это связано с меньшими затратами, чем в случае замены модифицированной страницы, которую

необходимо будет физически переписывать во внешнюю память. В противном случае нам придется заменять модифицированную страницу.

В многоабонентских системах основная память, естественно, работает активно, так что рано или поздно у большинства страниц бит-признак обращения будет установлен в 1, и мы не сможем отличать те страницы, которые вытолкнуть наиболее целесообразно. Один из широко распространенных способов решения этой проблемы заключается в том, что все биты-признаки обращений периодически сбрасываются в 0, с тем чтобы механизм вталкивания оказался в исходном состоянии, а затем снова разрешается установка этих битов-признаков в 1 обычным образом при обращениях. Правда, в этом случае существует опасность того, что могут быть вытолкнуты даже активные страницы, однако только в течение короткого периода после сброса битов-признаков, поскольку почти немедленно биты-признаки обращений для этих страниц будут снова установлены в 1.

Описанный выше алгоритм NUR предусматривает существование четырех групп страниц:

Группа 1

Обращений не было

Модификаций не было

Группа 2

Обращений не было

Модификация была

Группа 3

Обращения были

Модификаций не было

Группа 4

Обращения были

Модификация была

Страницы групп с меньшими номерами следует выталкивать в первую очередь, а с большими — в последнюю. Отметим, что группа 2 обозначает на первый взгляд нереальную ситуацию, она включает страницы, к которым как бы не было обращений, но они оказались модифицированными. В действительности это просто результат того, что биты-признаки обращения (но не биты-признаки модификации) периодически сбрасываются, т. е. подобная ситуация вполне возможна.

9.4 Локальность

Большинство стратегий управления памятью базируется на концепции локальности, суть которой заключается в том, что распределение запросов процессов на обращение к памяти имеет, как правило, неравномерный характер с высокой степенью локальной концентрации.

Свойство локальности проявляется как во времени, так и в пространстве. Временная локальность — это концентрация во времени. Если, например, в 15 ч наблюдается солнечная погода, то можно с достаточно высокой вероятностью считать (но, естественно, без всяких гарантий), что было солнечно в 14 ч 30 мин и будет солнечно в 15 ч 30 мин. Пространственная локальность означает, что соседние объекты будут, как правило, характеризоваться одинаковыми свойствами. Если опять-таки взять в качестве примера погоду, то в случае, когда в одном городе солнечно, можно с достаточно высокой вероятностью (но без всяких гарантий) считать, что и в близлежащих городах солнечно.

Свойство локальности наблюдается также и в работе операционных систем, в частности при управлении памятью. Свойство это скорее эмпирическое (наблюдаемое), чем теоретически обоснованное. Локальность никак нельзя гарантировать, однако ее вероятность зачастую весьма велика. Например, в системах со страничной организацией мы наблюдаем, что процессы обычно «оказывают предпочтение» определенным подмножествам своих страниц и что эти страницы обычно размещаются по соседству друг с другом в виртуальном адресном пространстве процесса. Это не значит, что процесс не будет иметь ссылок на какую-либо новую страницу — если бы это было так, то процессы вообще не могли бы даже начать выполнение. Это просто означает, что процесс будет, как правило, сосредоточивать свои обращения в течение некоторого временного интервала на некотором конкретном подмножестве своих страниц.

9.6 Подкачка страниц по запросу

Традиционно считается, что наиболее рационально загружать в основную память страницы, необходимые для работы процесса, по его запросу. Не следует переписывать из внешней памяти в основную ни одной страницы до тех пор, пока к ней явно не обратится выполняющийся процесс. В пользу такой стратегии можно привести несколько аргументов:

- Теория вычислимости, в частности проблема останова (Mi67, He77), говорит о том, что путь, который выберет программа при своем выполнении, точно предсказать невозможно. Поэтому любая попытка заранее загрузить страницы в память в предвидении того, что они потребуются в работе, может оказаться неудачной — будут загружены не те страницы.

- Подкачка страниц по запросу гарантирует, что в основную память будут переписываться только те страницы, которые фактически необходимы для работы процессов.

- Накладные расходы на то, чтобы определить, какие страницы следует передавать в основную память, минимальны. Стратегии вталкивания с упреждением могут потребовать значительных дополнительных затрат процессорного времени.

Подкачка страниц по запросу, как показывает рис. 9.7, имеет свои проблемы. Процесс должен накапливать в памяти, требуемые ему страницы по одной. При появлении ссылки на каждую новую страницу процессу приходится ждать, когда эта страница будет передана в основную память. В зависимости от того, сколько страниц данного процесса уже находятся в основной памяти, эти периоды ожидания будут обходиться все более дорого, поскольку ожидающие процессы будут занимать все больший объем памяти. Рис. 9.7 иллюстрирует понятие произведения «пространство— время», которое часто применяется в операционных системах для оценки использования памяти процессом. Произведение «пространство— время» соответствует площади «под кривой» на этом рисунке. Оно является комплексным показателем, отражающим как объем, так и время использования памяти процессом. Уменьшение произведения «пространство— время» за счет периодов ожидания процессом нужных ему страниц является важнейшей целью всех стратегий управления памятью.

9.6 Подкачка страниц с упреждением

Главный критерий эффективности управления ресурсами можно сформулировать следующим образом: интенсивность использования каждого ресурса должна определяться относительной ценностью этого ресурса. Так, в настоящее время стоимость аппаратуры резко снижается, и поэтому значительно снижается относительная ценность машинного времени по сравнению с временем, затрачиваемым человеком. Сейчас разработчики операционных систем активно ищут пути уменьшения количества времени, в течение которого пользователям приходится ждать получения результатов от компьютера. Одним из

весьма перспективных в этом смысле является метод подкачки страниц с упреждением (с опережением).

При упреждающей подкачке операционная система пытается заблаговременно предсказать, какие страницы потребуются процессу, а затем, когда в основной памяти появляется свободное место, загружает в нее эти страницы. Пока процесс работает со своими текущими страницами, система запрашивает новые страницы, которые будут уже готовы к использованию, когда процесс к ним обратится. Если решения о выборе страниц для подкачки принимаются правильно, то удастся значительно сократить общее время выполнения данного процесса.

Метод подкачки страниц с упреждением характеризуется следующими преимуществами:

- Если в большинстве случаев удастся принимать правильные решения о выборе страниц для подкачки, то время выполнения процесса значительно уменьшается. Поэтому имеет смысл пытаться создавать механизмы упреждающей подкачки, даже если от них нельзя ожидать абсолютной точности.

- Во многих случаях вполне можно находить правильные решения. Если такое принятие решений удастся реализовать при относительно малых затратах, то выполнение данного процесса можно значительно ускорить, не замедляя при этом работы других активных процессов.

- Поскольку аппаратура вычислительных машин все более дешевеет, последствия неоптимальных решений становятся менее серьезны. Сейчас мы можем позволить себе приобрести дополнительную основную память, обеспечивающую накопление дополнительных страниц, которые механизм упреждающей подкачки будет передавать в основную память.

9.8 Освобождение страниц

При управлении памятью по рабочему множеству программы говорят нам, какие страницы они хотят использовать, явно обращаясь к этим страницам. Страницы, которые больше не нужны, должны каким-то образом выводиться из рабочих множеств. Существует обычно период времени, в течение которого ненужные страницы все еще остаются в основной памяти.

Когда выясняется, что некоторая страница больше не понадобится, пользователь может по собственной инициативе выдать сигнал об удалении страницы из памяти (отказаться от страницы) с освобождением соответствующего страничного кадра. Благодаря этому будет исключен период задержки, неизбежный в случае, если предоставить системе возможность естественным порядком вывести эту страницу из рабочего множества процесса.

Подобное «добровольное освобождение» страниц может исключить непроизводительное использование памяти и ускорить выполнение программ. Однако большинство пользователей современных вычислительных машин даже не знают, что такое страница, поэтому их вообще нельзя просить принимать решения на системном уровне. Включение команд освобождения страниц в прикладные программы пользователя может значительно замедлить их разработку.

Можно надеяться, что реально эта проблема будет решена при помощи компиляторов и операционных систем, которые будут автоматически обнаруживать ситуации, требующие освобождения страниц, причем, делать это гораздо быстрее, чем позволяют стратегии с использованием рабочих множеств страниц.

9.9 Размер страниц

В системах со страничной организацией реальная память обычно разбивается на страничные кадры фиксированного размера. Какой величины следует выбирать размер этих страничных кадров? Какой величины следует выбирать размер страницы? Должны ли все страницы в системе иметь одинаковый размер либо следует использовать страницы нескольких различных размеров? Если используются различные размеры страниц, то должны ли большие размеры страниц быть целыми кратными меньших размеров?

Для ответа на эти вопросы требуется тщательный анализ и всестороннее понимание возможностей аппаратных и программных средств и области применения конкретной системы. Здесь не может быть универсальных ответов. Не существует никакой особой необходимости в том, чтобы все вычислительные системы имели одинаковый или единственный размер страниц.

Некоторые соображения, которые необходимо учитывать при выборе того или иного размера страницы, приведены ниже.

- Чем меньше размер страницы, тем большее количество страниц и страничных кадров будет в системе и тем больше будут таблицы страниц. Для систем, в которых таблицы страниц занимают первичную память, это указывает на необходимость увеличенных размеров страниц. Такое неэффективное использование памяти из-за чрезмерно больших таблиц называется табличной фрагментацией. Здесь мы отметим, что подобный аргумент в настоящее время не столь актуален, поскольку выпускается недорогая память очень большой емкости.

- При крупных размерах страниц в первичную память при подкачке попадают большие объемы информации, которая, вообще говоря, может и не понадобиться. Это указывает на необходимость уменьшения размеров страниц.

- Поскольку обмены данными с дисковой памятью занимают относительно много времени, мы хотим свести к минимуму число обменов, которые будут производиться для программы во время ее выполнения. Это указывает, по-видимому, на необходимость увеличения размеров страниц.

- Программы, как правило, проявляют свойство локальности обращений, причем размеры подобных локальных участков невелики. Таким образом, уменьшение размера страницы должно способствовать тому, что у программы образуется более компактный набор страниц, т. е. «страницы рабочего множества» размещаемые в реальной памяти, будут содержать более интенсивно используемые объекты.

- Поскольку блоки процедур и данных редко представляют целое число страниц, в системах со страничной организацией наблюдается внутренняя фрагментация, иллюстрируемая рис. 9.8. Сегмент длины s с равной вероятностью может иметь свою последнюю страницу почти полную или почти пустую, и, таким образом, в среднем потери из-за внутренней фрагментации составляют половину страницы (при том ограничительном условии, что страница не может содержать части более чем одного сегмента). Чем меньше размер страницы, тем меньше потери из-за внутренней фрагментации.

Большинство приводящихся в литературе данных (De80), как теоретических, так и эмпирических, указывает на то, что необходимы страницы малых размеров.

Ассоциативная память

Оперативную память (ОП) можно представить в виде двумерной таблицы, строки которой хранят в двоичном коде команды и данные. Обращения за содержанием строки производится заданием номера (адреса) нужной строки. При записи, кроме адреса строки указывается регистр, содержимое которого следует записать в эту строку. Запись занимает больше времени, чем чтение. Пусть в памяти из трех строк хранятся номера телефонов.

1924021
9448304

3336167

Для получения номера телефона второго абонента следует обратиться: load (2) и получить в регистре ответа R = 9448304. Такой вид памяти, при котором необходимая информация определяется номером строки памяти, называется адресной. Другой вид оперативной памяти – ассоциативной можно рассматривать также как двумерную таблицу, но у каждой строки которой есть дополнительное поле, поле ключа. Например:

Ключ	Содержимое
Иванов	1924021
Петров	9448304
Сидоров	3336167

После обращение к ассоциативной памяти с запросом : load (Петров) для данного примера получим ответ: R = 9448304. Здесь задание координаты строки памяти производится не по адресу, а по ключу. Но при состоянии ассоциативной памяти:

Ключ	Содержимое
1	1924021
2	9448304
3	3336167

можно получить номер телефона из второй строки запросом: load (2). Таким образом на ассоциативной памяти можно моделировать работу адресной. Ассоциативная память имеет очевидные преимущества перед адресной, однако, у нее есть большой недостаток - ее аппаратная реализация невозможна для памяти большого объема.

ВОПРОС: Предложите схему реализации модели ассоциативной памяти при помощи адресной.

Ответ: Область памяти делим ровно пополам. Первая половина заполняется ключами, вторая соответствующими ключам значениями. Когда найден ключ, известен его адрес как смещение относительно начала памяти. Тогда адрес содержимого по ключу – это смещение + размер области ключей, то есть адрес ячейки из второй половины памяти, которая соответствует ключу.

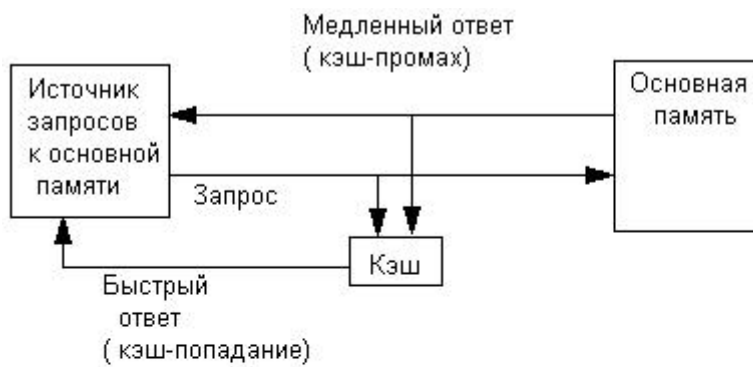
Назначение и структура кэш-памяти.

Cache - это способ совместного функционирования двух типов запоминающих устройств, отличающихся временем доступа и стоимостью хранения данных, который позволяет уменьшить среднее время доступа к данным и экономить более дорогую быстродействующую память за счет динамического копирования в "быстрое" ЗУ (Кэш-память) наиболее часто используемой информации из "медленного" ЗУ.

Кэш - очень быстрая память сравнительно небольшого объема (обычно на статических микросхемах SRAM), логически располагающаяся между ЦП и оперативной памятью.

Кэш состоит из строк (блоков небольшого размера); каждая кэш-строка, кроме блока информации, содержит адрес начала блока в оперативной памяти и тэги (дополнительная информация, которая используется для реализации алгоритма замещения данных в кэше и обычно включает признак модификации и признак действительности данных).

При каждом обращении к основной памяти по физическому адресу просматривается содержимое кэш-памяти с целью определения, не находятся ли там нужные данные. Если данные обнаруживаются в кэш-памяти, то есть произошло кэш-попадание (cache-hit), они считываются из нее и результат передается источнику запроса.



Структура кэш-памяти

Адрес данных в основной памяти	Данные	Управляющая информация

Алгоритм поиска и алгоритм замещения данных в кэше непосредственно зависят от того, каким образом основная память отображается на кэш-память. Существует три основных типа организации кэш-памяти, которые определяются принципом размещения строк в кэше.

Концепция кэш-памяти возникла раньше чем архитектура IBM/360, и сегодня кэш-память имеется практически в любом классе компьютеров, а в некоторых компьютерах - во множественном числе.

Все термины, которые были определены раньше могут быть использованы и для кэш-памяти, хотя слово "строка" (line) часто употребляется вместо слова "блок" (block).

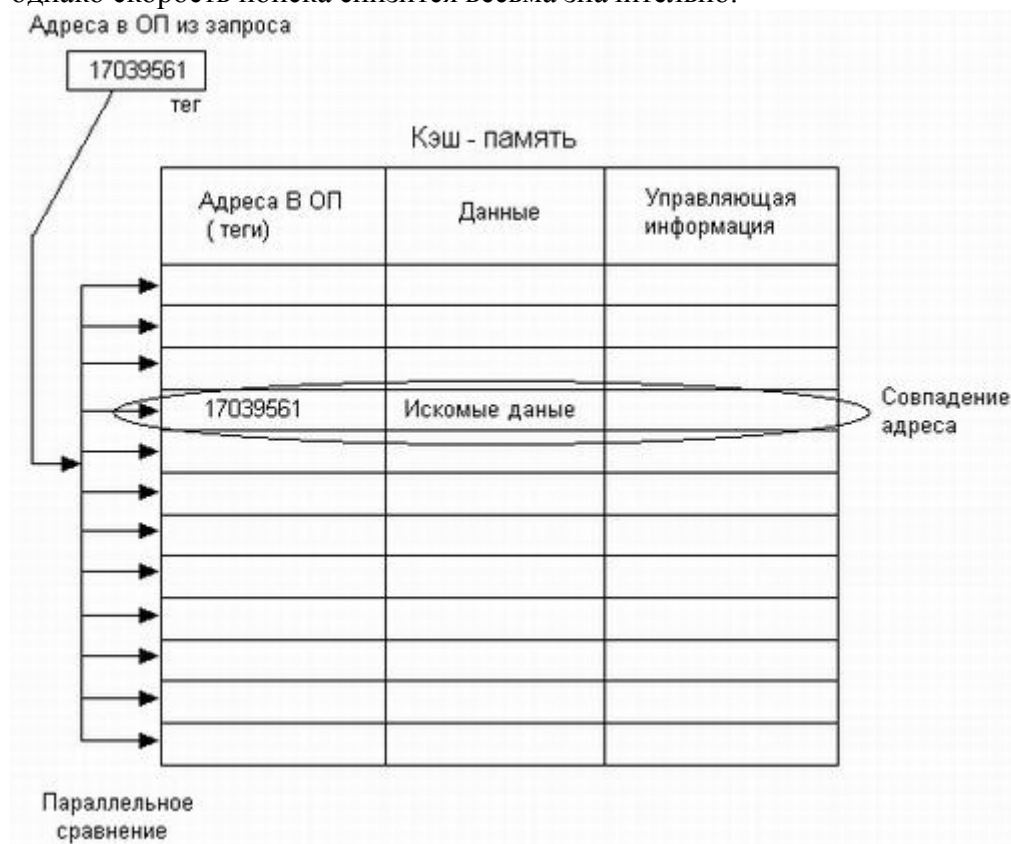
Кэш-строка представляет собой полный набор данных, которым обмениваются кэш и основная память. В кэш-строке адрес двойного слова для 32-битной шины данных задается адресными битами A3-A2. Кэш-строка обычно является либо целиком действительной, либо целиком недействительной; она не может быть действительной частично. Если даже процессору нужно прочитать только один байт, то все равно все 16 байт соответствующей кэш-строки должны находиться в кэш-памяти SRAM; в противном случае происходит кэш-промах. Кэш-строка - это собственно кэш-память; кэш-директория используется только для управления ею. Кэш-строка обычно содержит большее число данных, чем можно передать за один цикл шины. По этой причине в большинстве кэш-контроллеров реализован пакетный режим, при котором предварительное задание адресных последовательностей обеспечивает более быструю передачу данных через шину. Это очень просто реализовать для заполнения кэш-строки и для обратной записи кэш-строк, поскольку в этих режимах всегда используется непрерывная и последовательно расположенная информация.

Полностью ассоциативная кэш-память.

Если некоторая строка основной памяти может располагаться на любом месте кэш-памяти, то такой кэш называется полностью ассоциативным (fully associative). Для того чтобы в дальнейшем можно было найти нужные данные в кэше, они помещаются туда вместе со своим адресом, то есть тем адресом, который данные имеют в оперативной памяти. При каждом запросе к оперативной памяти выполняется поиск в кэше, причем критерием поиска выступает адрес оперативной памяти из запроса. Каждый тег одновременно сравнивается с

этим адресом (ассоциативный поиск - сравнение выполняется не последовательно с каждой записью кэша, а параллельно со всеми его записями, так как простой перебор приводит к недопустимо большим временным затратам). Если тег совпадает с адресом, то обеспечивается доступ к блоку памяти, связанному с этим тегом.

Такая организация позволяет достичь максимально возможного процента попаданий, однако скорость поиска снизится весьма значительно.



Дополнение

Память, адресуемая по содержанию

Понятие ассоциативной памяти часто связывается с кэш-памятью. Она известна также как память, адресуемая по содержанию, или САМ. Название указывает на принцип ее работы. Обычно к данным в памяти обращаются при помощи адреса. Однако в случае ассоциативной памяти для этого используется часть собственно данных, хранящихся в памяти САМ. Чтобы прочесть данные из САМ, туда посылается фрагмент этих данных. Если этот фрагмент совпадает с соответствующим фрагментом информации, которая там хранится, САМ возвращает всю информацию, связанную с этим фрагментом, т.е., иначе говоря, она "дополняет" фрагмент. Поскольку фрагмент, направляемый в САМ, является только частью полного элемента данных, может случиться так, что у других элементов данных тоже окажутся совпадающие с ним фрагменты, которые будут поэтому тоже выданы в ответ на запрос. Результат в этом случае будет неоднозначным; впрочем, при использовании ассоциаций это не удивительно - вы тоже связываете, например, слово "море" со многими вещами - с отпуском, пляжем, купанием, загаром и т.п. Если совпадение с содержащимися в памяти данными полностью отсутствует, САМ не выдаст никакой информации. Так же происходит и в реальной жизни Ц осмысленная ассоциация с "вапролд" является как минимум затруднительной.

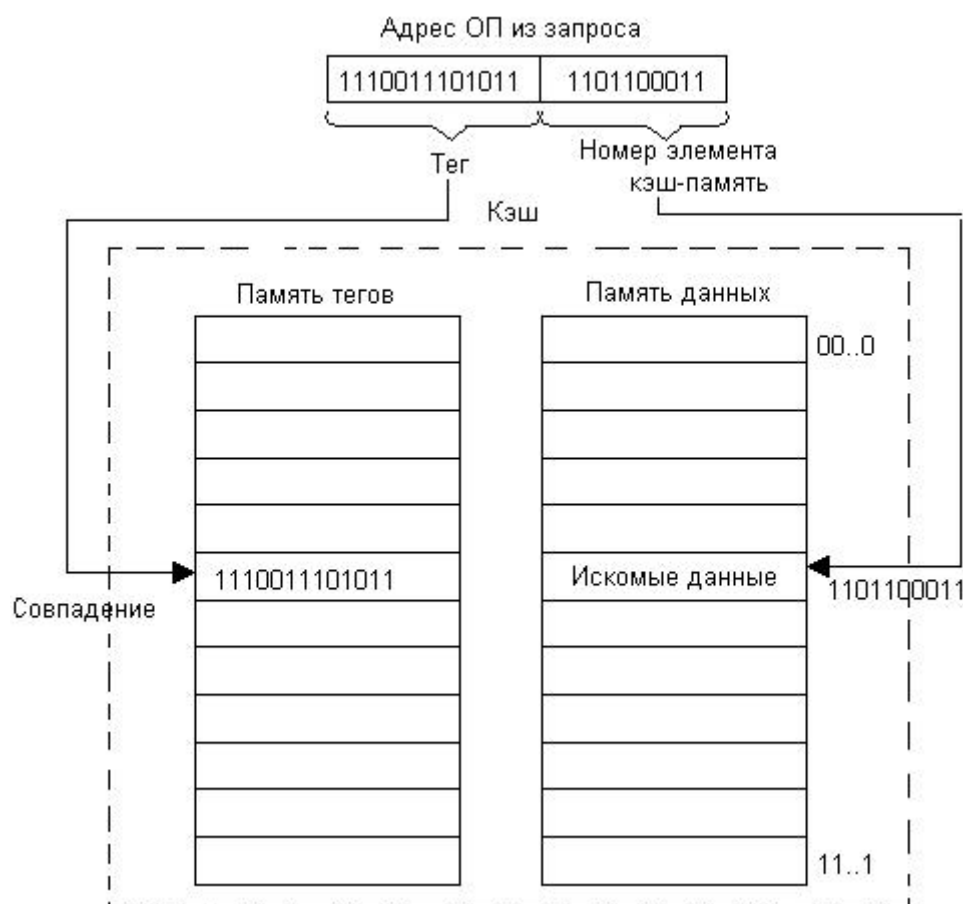
Ассоциативная адресация выполняется набор за набором, поэтому работают оба вида ассоциаций. Адресация набора, однако, выполняется явным образом посредством адреса набора А5-А11. Фрагмент информации для ассоциативной адресации задается адресом

дескриптора внутри набора. Этот адрес дескриптора сравнивается с адресами A12-31 из CPU. Если они совпадают, то кэш-контроллер присоединяет весь соответствующий элемент кэш-памяти, т.е. всю кэш-строку, принадлежащую соответствующей магистрали набора, который имеет нужный адрес дескриптора. Подробнее об этом будет рассказано дальше.

Кэш-память с прямым отображением.

Если каждая строка основной памяти может быть отображена только в одну фиксированную строку кэш-памяти, то такая кэш-память называется кэшем **с прямым отображением** (direct mapped). В каждый блок помещаются данные с определенными (непересекающимися) диапазонами адресов (группами диапазонов). Это наиболее простая организация кэш-памяти, при которой для отображения адресов блоков основной памяти на адреса кэш-памяти просто используются младшие разряды адреса блока (выделяются несколько разрядов из адреса оперативной памяти и интерпретируются как номер строки кэш-памяти).

$(\text{адрес блока кэш-памяти}) = (\text{адрес блока основной памяти}) \bmod (\text{число блоков в кэш-памяти})$



При этом все блоки основной памяти, имеющие одинаковые младшие разряды в своем адресе, попадают в один блок кэш-памяти. Между номерами строк кэш-памяти и адресами оперативной памяти устанавливается соответствие «один ко многим»: одному номеру строки соответствует несколько (обычно достаточно много) адресов оперативной памяти.

При поиске данных в кэше используется быстрый прямой доступ к записи по номеру строки, полученному путем обработки адреса оперативной памяти из запроса. Однако поскольку в найденной строке могут находиться данные из любой ячейки оперативной памяти, младшие разряды адреса которой совпадают с номером строки, необходимо выполнить дополнительную проверку. Для этих целей каждая строка кэш-памяти

дополняется тегом, содержащим старшую часть адреса данных в оперативной памяти. При совпадении тега с соответствующей частью адреса из запроса констатируется кэш-попадание.

Достоинство такой организации заключается в том, что наиболее просты аппаратные решения; поиск по кэшу производится максимально быстро, но при этом процент попаданий (hit rate) относительно низок, из-за того, что порой невозможно одновременно кэшировать две секции часто используемых данных, т.к. они не могут занять одно и то же место в кэше.

По версии лектора

Кэш памяти с прямым отображением.

Архитектура кэш-памяти с прямым отображением (direct-mapped) характеризуется наличием явной зависимости между адресами буферной и оперативной памяти, причем каждому адресу кэша соответствует адреса оперативной памяти, кратные размеру кэш памяти. Память кэша состоит из памяти данных и памяти тэгов. Пусть, длина строки кэша равна 32 байтам, размер кэша – 4 Кбайт, тогда кэш памяти для данных состоит из 128 строк. ОЗУ разделено на блоки, размером в 4Кбайт, каждый содержит 128 строк. В нулевой строке кэша может быть размещены 0 или 128, или строки ОЗУ, в первой – 1 или 129 или.. строки т.д. Для каждой заполненной строки данных кэша известен тэг – номер блока ОЗУ, которому принадлежит строка. Тэги хранятся в специальной памяти - памяти тэгов, размер которой – 128 строк. Длина строки памяти тэгов зависит от размера ОЗУ. Если объем ОЗУ – 4 Гбайт, тогда полный адрес - 32 бита можно представить в виде полей: 20 pp – тэг (T), 7 pp – номер строки таблиц кэша (S), 5 pp – номер байта в строке (N). Поиск запрошенного байта (T-S-N) в кэше с прямым распределением производится так:

1. Из памяти данных и памяти тэгов кэша одновременно считываются S-ные строки.

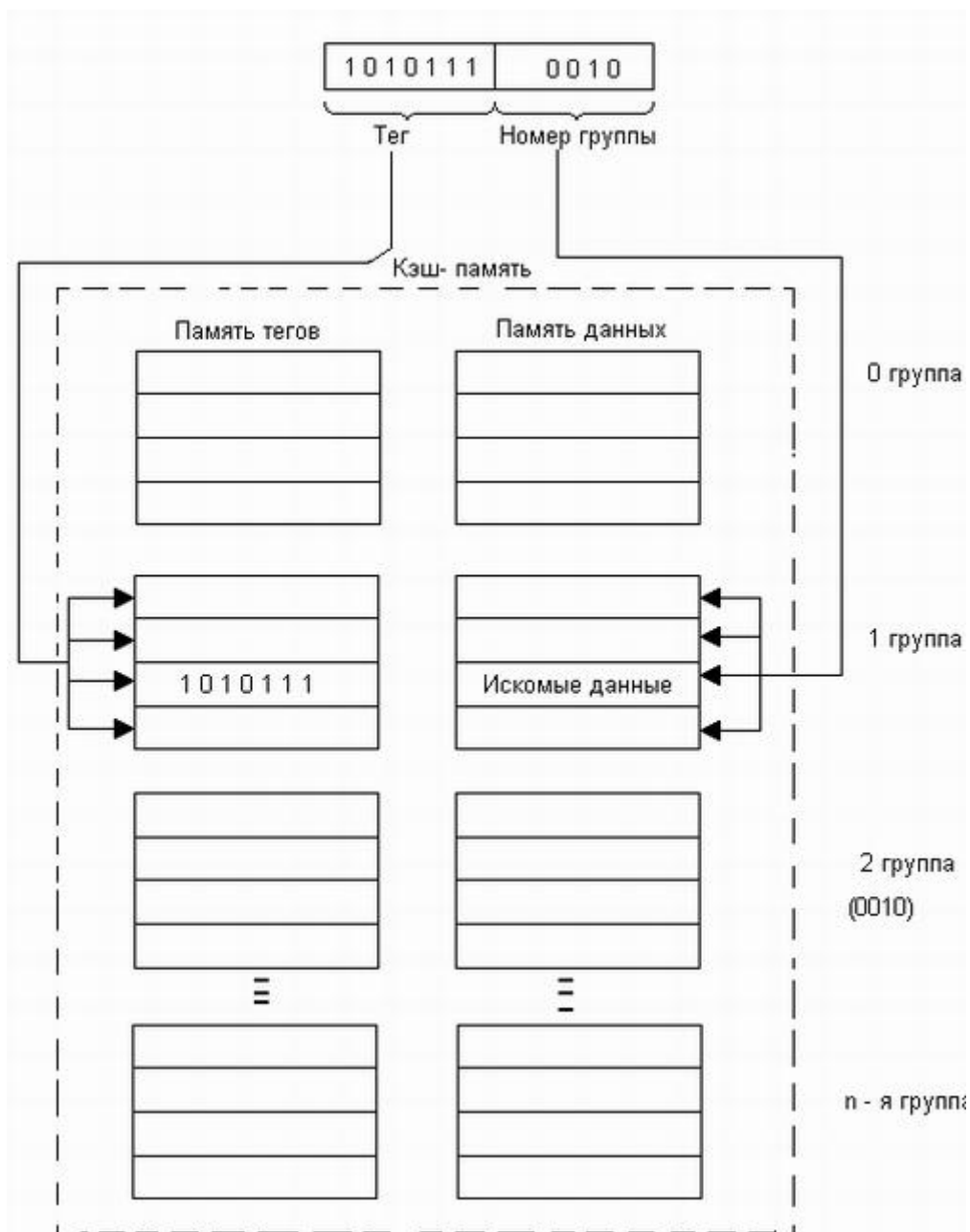
Если содержимое считанной строки памяти тэгов равно T – кэш попадание, это значит, что считанная S строка памяти данных кэша содержит запрашиваемый байт и его номер в строке есть N.

3. Если содержимое считанной строки памяти тэгов не равно T – кэш промах, и тогда T-S строка ОЗУ переписывается в S строку памяти данных кэша, а T записывается в S строку памяти тэгов. Затем, см по п. 1.

Частично ассоциативная кэш-память.

Если некоторая строка основной памяти может располагаться на ограниченном множестве мест в кэш-памяти, то кэш называется **частично-ассоциативным** (set associative). То есть произвольный адрес оперативной памяти отображается на некоторую группу адресов кэш-памяти. Если группа состоит из n строк, то такое размещение называется *частично-ассоциативным с n каналами* (n-way set associative). Группа определяется младшими разрядами адреса строки:

$(\text{адрес группы кэш-памяти}) = (\text{адрес строки основной памяти}) \bmod (\text{число групп в кэш-памяти})$. Далее блок может размещаться на любом месте данной группы.



Все группы пронумерованы. Поиск в кэше осуществляется вначале по номеру группы, полученному из адреса оперативной памяти из запроса, а затем в пределах группы путем ассоциативного просмотра всех записей в группе на предмет совпадения старших частей адресов оперативной памяти. (То есть указанный адрес ищем в любом блоке в пределах данной группы блоков; внешне – кэш с прямой адресацией, внутри – полностью ассоциативный кэш).

Адресация частично-ассоциативной кэш-памяти осуществляется путем деления адреса на три части: поле смещения используется для выбора байта внутри строки кэш-памяти, поле индекса определяет номер группы, а поле тега используется для сравнения. Если общий размер кэш-памяти зафиксировать, то увеличение степени ассоциативности приводит к увеличению количества строк в группе, при этом уменьшается размер индекса и увеличивается размер тега.

Магистраль

Понятие магистрали указывает на ассоциативный характер кэш-системы. Для заданного адреса набора адреса дескрипторов для всех магистралей одновременно сравниваются с дескрипторной частью адреса, выдаваемого процессором, чтобы определить, находятся

данные в кэш-памяти или нет. В примере показана ассоциативная кэш-память, имеющая 4 магистрали. Это означает, что массив данных, соответствующий кэш-строке, может храниться в кэш-памяти в четырех различных местах. Каждая магистраль в примере содержит 256 наборов, каждый из которых имеет кэш-строку длиной 16 байт. Это, в соответствии с формулой: $\text{емкость} = \text{число магистралей} \times \text{число наборов} \times \text{длина кэш-строки}$, обеспечивает указанные 16 кбайт кэш-памяти. Элементу набора для всех четырех магистралей назначается 3-битный элемент LRU; кэш-контроллер использует эти биты для того, чтобы определить, какая из четырех кэш-строк должна обновляться в случае кэш-промаха. Для этого существует ряд алгоритмов.

Заметим, что в данном случае адрес памяти в верхней части рисунка 8.2 не содержит элемента для магистрали. При наличии магистрали в игру вступает ассоциативность. Кэш с непосредственно отображаемой памятью имеет только одну магистраль и поэтому не является ассоциативным; массив данных, хранящийся в кэш-памяти, может занимать в ней только одно положение, определяемое 12 младшими битами. В рассматриваемом примере данные, расположенные по двоичному адресу xxxx xxxx xxxx xxxx ssss sssx xxxb, всегда будут храниться в наборе ssssss. Элемент, который ранее там размещался, должен быть заменен новым. В 4-магистральной ассоциативной кэш-памяти данные с двоичным адресом xxxx xxxx xxxx xxxx ssss sssx xxxb, также будут всегда храниться в наборе ssssss. В этом случае, однако, для каждого набора доступны четыре магистрали, поэтому предыдущий элемент не всегда приходится перезаписывать. Если кэш-строка в другой магистрали свободна, то кэш-контроллер может поместить данные в свободную магистраль. Это увеличивает и число кэш-попаданий, поскольку для одного и того же адреса набора имеется четыре элемента. 8-магистральные ассоциативные модели кэш-памяти имеют соответственно большее число кэш-попаданий, но увеличивается и техническая сложность. В противоположность этому, часто реализуемые 2-магистральные ассоциативные модели кэш-памяти имеют существенно меньшее число попаданий, но и логики для определения кэш-попадания нужно значительно меньше. Это приводит к увеличению скорости работы. Такая организация используется особенно часто для высокоскоростных моделей кэш-памяти (например, для встроенной в чип кэш-памяти в Pentium).

Дисциплина обновления кэш-памяти.

При возникновении промаха происходит обновление содержимого КЭШа – вытеснение.

1) Кэш с прямым отображением. Если произошел промах, то данные считываются из оперативной памяти и копируются в кэш. Если строка кэш-памяти, в которую должен быть скопирован элемент данных из оперативной памяти, содержит другие данные, то последние вытесняются из кэша. Заметим, что процесс замещения данных в кэш-памяти на основе прямого отображения существенно отличается от процесса замещения данных в ассоциативной кэш-памяти. Во-первых, вытеснение данных происходит не только в случае отсутствия свободного места в кэше, во-вторых, никакого выбора данных на замещение не существует.

2) Полностью ассоциативный кэш. Вытеснение старых данных происходит только в том случае, когда вся кэш-память заполнена и нет свободного места. Выбор данных на выгрузку осуществляется среди всех записей кэша. Стратегии вытеснения:

- случайная – с кэшем ассоциируется датчик случайных чисел. Достоинство – этот алгоритм проще реализовать.
- LRU - Least-Recently Used - заменяется та строка, которая не использовалась дольше всех (выбирается блок, частота обращения к которому была наименьшей) - чтобы уменьшить вероятность выбрасывания информации, которая скоро может потребоваться. Достоинство – этот алгоритм учитывает интенсивность обращения к данным и тем самым повышает вероятность попаданий в будущем.

- 3) Частично-ассоциативный кэш. При промахе данные копируются по любому свободному адресу из однозначно заданной группы. Если свободных адресов в группе нет, то выполняется вытеснение данных – одна из стратегий. Таким образом, в данном способе комбинируется прямое отображение на группу и случайное отображение в пределах группы.

Дополнение

3. Какой блок кэш-памяти должен быть замещен при промахе?

При возникновении промаха, контроллер кэш-памяти должен выбрать подлежащий замещению блок. Польза от использования организации с прямым отображением заключается в том, что аппаратные решения здесь наиболее простые. Выбирать просто нечего: на попадание проверяется только один блок и только этот блок может быть замещен. При полностью ассоциативной или множественно-ассоциативной организации кэш-памяти имеются несколько блоков, из которых надо выбрать кандидата в случае промаха. Как правило для замещения блоков применяются две основных стратегии: случайная и LRU.

В первом случае, чтобы иметь равномерное распределение, блоки-кандидаты выбираются случайно. В некоторых системах, чтобы получить воспроизводимое поведение, которое особенно полезно во время отладки аппаратуры, используют псевдослучайный алгоритм замещения.

Во втором случае, чтобы уменьшить вероятность выбрасывания информации, которая скоро может потребоваться, все обращения к блокам фиксируются. Заменяется тот блок, который не использовался дольше всех (LRU - Least-Recently Used).

Достоинство случайного способа заключается в том, что его проще реализовать в аппаратуре. Когда количество блоков для поддержания трассы увеличивается, алгоритм LRU становится все более дорогим и часто только приближенным. На рисунке 3.22 показаны различия в долях промахов при использовании алгоритма замещения LRU и случайного алгоритма. Ассоциативность: 2-канальная 4-канальная 8-канальная

Сравнение долей промахов для алгоритма LRU и случайного алгоритма замещения при нескольких размерах кэша и разных ассоциативностях при размере блока 16 байт

Размер кэш-памяти	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Стратегии записи в кэш-память.

При обращениях к кэш-памяти на реальных программах преобладают обращения по чтению. Все обращения за командами являются обращениями по чтению и большинство команд не пишут в память. Обычно операции записи составляют менее 10% общего трафика памяти. Желание сделать общий случай более быстрым означает оптимизацию кэш-памяти для выполнения операций чтения, однако при реализации высокопроизводительной обработки данных нельзя пренебрегать и скоростью операций записи.

К счастью, общий случай является и более простым. Блок из кэш-памяти может быть прочитан в то же самое время, когда читается и сравнивается его тег. Таким образом, чтение

блока начинается сразу как только становится доступным адрес блока. Если чтение происходит с попаданием, то блок немедленно направляется в процессор. Если же происходит промах, то от заранее считанного блока нет никакой пользы, правда нет и никакого вреда.

Однако при выполнении операции записи ситуация коренным образом меняется. Именно процессор определяет размер записи (обычно от 1 до 8 байтов) и только эта часть блока может быть изменена. В общем случае это подразумевает выполнение над блоком последовательности операций чтение-модификация-запись: чтение оригинала блока, модификацию его части и запись нового значения блока. Более того, модификация блока не может начинаться до тех пор, пока проверяется тег, чтобы убедиться в том, что обращение является попаданием. Поскольку проверка тегов не может выполняться параллельно с другой работой, то операции записи отнимают больше времени, чем операции чтения.

Очень часто организация кэш-памяти в разных машинах отличается именно стратегией выполнения записи. Когда выполняется запись в кэш-память имеются две базовые возможности:

сквозная запись (write through, store through) - информация записывается в два места: в блок кэш-памяти и в блок более низкого уровня памяти.

запись с обратным копированием (write back, copy back, store in) - информация записывается только в блок кэш-памяти. Модифицированный блок кэш-памяти записывается в основную память только когда он замещается. Для сокращения частоты копирования блоков при замещении обычно с каждым блоком кэш-памяти связывается так называемый бит модификации (dirty bit). Этот бит состояния показывает был ли модифицирован блок, находящийся в кэш-памяти. Если он не модифицировался, то обратное копирование отменяется, поскольку более низкий уровень содержит ту же самую информацию, что и кэш-память.

Оба подхода к организации записи имеют свои преимущества и недостатки. При записи с обратным копированием операции записи выполняются со скоростью кэш-памяти, и несколько записей в один и тот же блок требуют только одной записи в память более низкого уровня. Поскольку в этом случае обращения к основной памяти происходят реже, вообще говоря требуется меньшая полоса пропускания памяти, что очень привлекательно для мультипроцессорных систем. При сквозной записи промахи по чтению не влияют на записи в более высокий уровень, и, кроме того, сквозная запись проще для реализации, чем запись с обратным копированием. Сквозная запись имеет также преимущество в том, что основная память имеет наиболее свежую копию данных. Это важно в мультипроцессорных системах, а также для организации ввода/вывода.

Когда процессор ожидает завершения записи при выполнении сквозной записи, то говорят, что он приостанавливается для записи (write stall). Общий прием минимизации остановов по записи связан с использованием буфера записи (write buffer), который позволяет процессору продолжить выполнение команд во время обновления содержимого памяти. Следует отметить, что остановки по записи могут возникать и при наличии буфера записи.

При промахе во время записи имеются две дополнительные возможности:

сквозная запись (write-through) - информация записывается в два места: в строку кэш-памяти и одновременно в основную память (при этом содержимое кэша и памяти когерентны – совпадают). Сквозная запись проще для реализации, чем запись с обратным копированием. Также преимущество в том, что основная память имеет наиболее свежую копию данных. Это важно в мультипроцессорных системах, а также для организации ввода/вывода.

запись с обратным копированием, отложенная запись (write-back) - информация записывается только в строку кэш-памяти. Модифицированная строка кэш-памяти записывается в основную память только, когда она замещается. Для сокращения частоты копирования строк при замещении обычно с каждой строкой кэш-памяти связывается так называемый бит модификации (Modify). Этот бит состояния показывает, была ли модифицирована строка, находящаяся в кэш-памяти. Если он не модифицировался, то обратное копирование отменяется, поскольку более низкий уровень содержит ту же самую информацию, что и кэш-память. При записи с обратным копированием операции записи выполняются со скоростью кэш-памяти, и несколько записей в один и тот же блок требуют только одной записи в память более низкого уровня; в этом случае обращения к основной памяти происходят реже.

Обычно в кэш-памяти, реализующей запись с обратным копированием, используется размещение записи в кэш-памяти (в надежде, что последующая запись в этот блок будет перехвачена), а в кэш-памяти со сквозной записью размещение записи в кэш-памяти часто не используется (поскольку последующая запись в этот блок все равно пойдет в память).

Увеличение производительности кэш-памяти

Формула для среднего времени доступа к памяти в системах с кэш-памятью выглядит следующим образом:

Среднее время доступа = Время обращения при попадании + Доля промахов $\times$ Потери при промахе

Эта формула наглядно показывает пути оптимизации работы кэш-памяти: сокращение доли промахов, сокращение потерь при промахе, а также сокращение времени обращения к кэш-памяти при попадании. Ниже на рисунке 3.23 кратко представлены различные методы, которые используются в настоящее время для увеличения производительности кэш-памяти. Использование тех или иных методов определяется прежде всего целью разработки, при этом конструкторы современных компьютеров заботятся о том, чтобы система оказалась сбалансированной по всем параметрам.

Расслоение памяти.

Наличие в системе множества микросхем памяти позволяет использовать потенциальный параллелизм, заложенный в такой организации. Для этого микросхемы памяти часто объединяются в банки или модули, содержащие фиксированное число слов, причем только к одному из этих слов банка возможно обращение в каждый момент времени. Как уже отмечалось, в реальных системах имеющаяся скорость доступа к таким банкам памяти редко оказывается достаточной. Следовательно, чтобы получить большую скорость доступа, нужно осуществлять одновременный доступ ко многим банкам памяти. Одна из общих методик, используемых для этого, называется расслоением памяти. При расслоении банки памяти обычно упорядочиваются так, чтобы N последовательных адресов памяти i , $i+1$, $i+2$, ..., $i+N-1$ приходились на N различных банков. В i -том банке памяти находятся только слова, адреса которых имеют вид $kN + i$ (где $0 \leq k < M-1$, а M число слов в одном банке). Можно достичь в N раз большей скорости доступа к памяти в целом, чем у отдельного ее банка, если обеспечить при каждом доступе обращение к данным в каждом из банков. Имеются разные способы реализации таких расслоенных структур. Большинство из них напоминают конвейеры, обеспечивающие рассылку адресов в различные банки и мультиплексирующие поступающие из банков данные. Таким образом, степень или коэффициент расслоения определяют распределение адресов по банкам памяти. Такие

системы оптимизируют обращения по последовательным адресам памяти, что является характерным при подкачке информации в кэш-память при чтении, а также при записи, в случае использования кэш-памятью механизмов обратного копирования. Однако, если требуется доступ к непоследовательно расположенным словам памяти, производительность расслоенной памяти может значительно снижаться.

Обобщением идеи расслоения памяти является возможность реализации нескольких независимых обращений, когда несколько контроллеров памяти позволяют банкам памяти (или группам расслоенных банков памяти) работать независимо.

Если система памяти разработана для поддержки множества независимых запросов (как это имеет место при работе с кэш-памятью, при реализации многопроцессорной и векторной обработки), эффективность системы будет в значительной степени зависеть от частоты поступления независимых запросов к разным банкам. Обращения по последовательным адресам, или в более общем случае обращения по адресам, отличающимся на нечетное число, хорошо обрабатываются традиционными схемами расслоенной памяти. Проблемы возникают, если разница в адресах последовательных обращений четная. Одно из решений, используемое в больших компьютерах, заключается в том, чтобы статистически уменьшить вероятность подобных обращений путем значительного увеличения количества банков памяти. Например, в суперкомпьютере NEC SX/3 используются 128 банков памяти.

Подобные проблемы могут быть решены как программными, так и аппаратными средствами.

Дополнение

Параллелизм работы блоков оперативной памяти ("расслоение памяти", Memory Interleaving). Один из аппаратных путей решения проблемы дисбаланса в скорости доступа к данным, размещенным в ОЗУ и производительностью ЦП. Суть расслоения ОЗУ состоит в следующем. Все ОЗУ состоит из k блоков, способных хранить одинаковое количество информации и способных взаимодействовать с процессором независимо друг от друга. Ячейки памяти распределены между блоками таким образом, что у любой ячейки ее соседи размещаются в соседних блоках. Полный адрес каждой ячейки памяти будет состоять из двух частей: номера банка и номера ячейки в банке.

Возможность предварительной буферизации при чтении команд/данных. Программа состоит (в большей степени) из линейных участков. Если использовать этот параллелизм, то можно организовать в процессоре еще один буфер (кэш?), который организован так же, но в котором размещаются машинные команды. За счет того, что есть параллельно работающие устройства, то этот буфер автоматически заполняется вперед. Т.е. за одно обращение можно прочесть k машинных слов и разместить их в этом буфере. Далее, действия с буфером команд похожи на действия с буфером чтения/записи. Когда нужна очередная команда (ее адрес находится в счетчике команд), происходит ее поиск (по адресу) в буфере, и если такая команда есть, то она считывается. Если такой команды нет, то опять-таки работает внутренний алгоритм выталкивания строки, новая строка считывается из памяти и копируется в буфер команд. Расслоение памяти в идеале увеличивает скорость доступа в k раз, плюс буфер команд позволяет сократить обращения к ОЗУ. (Когда идет запрос на определенную ячейку памяти, одновременно считывается по одной ячейке из каждого банка с "номером в банке", равным "номеру в банке" запрошенной ячейки; на выход выдается ячейка с нужным номером банка. Соответственно, чем больше банков, тем быстрее работа с памятью, но только при последовательном чтении. Последовательное чтение - 60% от всех чтений из памяти.); (При обычном однобанковом устройстве модуля памяти, каждое последующее слово можно получать лишь через 50 нс. после предыдущего, а то и реже. Если разбить память на отдельные области (банки), то при последовательном доступе одно

слово будет выдаваться первым банком, а следующим банком - второе и т. д. К тому моменту, когда снова нужно будет обратиться к первому банку, пройдет полный цикл и он будет готов выдать данные без задержки. Теоретически, ускорение работы прямо пропорционально числу банков памяти. На практике это не достижимо: существуют накладные расходы, кроме того, программа может обращаться к памяти не последовательно, а к произвольным ячейкам, что легко сводит все преимущества расслоения на нет. Однако, в большинстве случаев деление на банки работает.)

Принципы VLIW архитектуры.

В ЭВМ с архитектурой VLIW (Very Long Instruction Word) - (очень длинные командные слова), команды могут иметь широкий формат (длину) и команда может содержать несколько содержательных инструкций, выполнение которых детально регламентируется в терминах тактов работы АЛУ (параллельное выполнение нескольких команд в АЛУ). В таких архитектурах имеется возможность программировать вычислительные алгоритмы (включая векторные) с максимальной производительностью для данной аппаратуры. В них вся работа по оптимальному программированию возлагается на системы программирования (или ручное программирование).

Однако упрощения в архитектуре управления приводит к значительному возрастанию сложности задачи планирования выдачи команд, так программными средствами должна быть обеспечена точная синхронизация считывания и записи данных. При этом необходимо так планировать параллельное выполнение операций машины, чтобы выполнялись определенные ограничения на число одновременно считываний и записей в наборы регистров, использование ФУ и т.д. Размер командного слова в машинах данной архитектуры - FPS (AP-120B) - 64 бита, Multilow Tract - 1024.

Определяющие свойства архитектуры VLIW:

- Одно центральное управляющее устройство (УУ), обрабатывающее за один такт одну длинную команду.
- Большое число функциональных устройств (ФУ) - АЛУ.
- Наличие в длинной команде полей, каждое из которых содержит команду управления некоторым функциональным устройством или команду обращения к памяти.
- Статически определенная длительность в тактах исполнения каждой операции. Операции могут быть конвейеризованы.
- Закрепление во время компиляции банков расслоенной памяти за ФУ для получения максимальной ширины доступа для данных, которые можно соединить в одну команду.
- Система передвижения данных между ФУ минуя память. Маршрут передвижения полностью специфицируется во время компиляции.
- Практическая невозможность ручного программирования в силу большой сложности возникающих комбинаторных задач. То есть требуются специальные системы программирования и оптимизации. В отличие от суперскалярных процессоров, здесь требуется статическая (на этапе компиляции) распараллеливание.

Архитектура сверхдлинного командного слова (VLIW — Very Long Instruction Word) берет свое начало от параллельного микрокода, применявшегося еще на заре вычислительной техники, и от суперкомпьютеров Control Data CDC6600 и IBM 360/91. В 1970 году многие вычислительные системы оснащались дополнительными векторными сигнальными процессорами, использующими VLIW-подобные длинные инструкции, прошитые в ПЗУ. Эти процессоры применялись для выполнения быстрого преобразования Фурье и других вычислительных алгоритмов. Первыми настоящими VLIW-компьютерами стали мини-суперкомпьютеры, выпущенные в начале 1980 года компаниями MultiFlow, Culler и Sydrome, но они не имели коммерческого успеха. Планировщик вычислений и программная

конвейеризация были предложены Фишером и Pay (Cydrone). Сегодня это является основой технологии VLIW-компилятора.

Первый VLIW-компьютер компании MultiFlow 7/300 использовал два арифметико-логического устройства для целых чисел, два АЛУ для чисел с плавающей точкой и блок логического ветвления — все это было собрано на нескольких микросхемах. Его 256bit командное слово содержало восемь 32bit кодов операций. Модули для обработки целых чисел могли выполнять две операции за один такт длиной 130ns (т.е. всего четыре при двух АЛУ), что при обработке целых чисел обеспечивало быстродействие около 30 MIPS. Можно было также комбинировать аппаратные решения так, чтобы получать из или 256bit, или 1024bit вычислительные машины. Первый VLIW-компьютер Cydrone Cydra-5 использовал 256bit инструкцию и специальный режим, обеспечивающий выполнение команд как последовательности из шести 40bit операций, поэтому его компиляторы могли генерировать смесь параллельного кода и обычного последовательного. Существует мнение, что в то время, как эти VLIW-БМ использовали несколько микросхем, процессор Intel i860 стал первым VLIW-процессором на одной микросхеме. Однако, i860 можно отнести к VLIW достаточно условно — по сути у него есть всего лишь программно-управляемое спаривание инструкций, в отличие от более позднего программно-неуправляемого, ставшего частью суперскалярных процессоров. В качестве исторической справки также хотелось бы упомянуть компьютеры фирмы FPS (AP-120B, AP-190L и все более поздние под маркой FPS), также основанные на VLIW-архитектуре, которые были в свое время достаточно распространенными и успешными на рынке. Кроме этого, существовали такие «канонические» машины, как M10 и M13 Карцева, а также «Эльбрус-3» — при всем «неуспехе» последнего проекта, он все же явился этапом VLIW. Вообще, быстродействие VLIW-процессора в большей степени зависит от компилятора, нежели от аппаратуры, поскольку здесь эффект от оптимизации последовательности операций превышает результат, возникающий от повышения частоты.

Относительно недавно мы были свидетелями «противостояния» CISC против RISC, а теперь уже намечается новое «сражение» — VLIW против RISC. Строго говоря, VLIW и суперскалярный RISC — никак не антагонисты, ни в коей мере. Справедливости ради необходимо отметить, что последние — это вовсе не «внешнеархитектурное» свойство, а просто некий способ исполнения. Возможно, что в дальнейшем появятся суперскалярные VLIW-процессоры, которые тем самым приобретут, если так можно выразиться, «параллелизм в квадрате» — объединение явного статического параллелизма с неявным динамическим. Но на сегодняшнем этапе развития процессоров нет видимых способов совмещать статическое и динамическое переупорядочивание. Именно поэтому Itanium/Itanium2 сейчас следует рассматривать не столько в контексте сравнения VLIW «против» CISC (и тем более не VLIW «против» O3E), а скорее как «синхронный VLIW» vs «асинхронный (Out-Of-Order) RISC». Ну, и не стоит забывать, что альянс Intel-HP придумали для своей архитектуры отдельное название — EPIC, т.е. явный параллелизм.

Суперскалярные и мультитредовые архитектуры микропроцессоров.

Суперскалярный процессор представляет собой нечто большее, чем обычный последовательный (скалярный) процессор. В отличие от последнего, он может выполнять несколько операций за один такт. Основными компонентами суперскалярного процессора являются устройства для интерпретации команд (УУ), снабженные логикой, позволяющей определить, являются ли команды независимыми, и достаточное число исполняющих устройств (ФУ, АЛУ). В исполняющих устройствах могут быть конвейеры. Суперскалярные процессоры реализуют параллелизм на уровне команд. Примером компьютера с суперскалярным процессором является IBM RISC/6000. Тактовая частота

процессора у ЭВМ была 62.5 МГц, а быстродействие системы на вычислительных тестах достигало 104 Mflop (Mflop - единица измерения быстродействия процессора - миллион операций с плавающей точкой в секунду). Суперскалярный процессор не требует специальных векторизирующих компиляторов, хотя компилятор должен в этом случае учитывать особенности архитектуры. Итак, суперскалярные процессор призван, в отличие от VLIW, динамически определять места распараллеливания.

Другой, по сравнению с организацией кэш-памяти, метод построения внутрикристальной памяти применяется в **мультиредовой** архитектуре, основная особенность которой – использование совокупности регистровых файлов (добавление УУ при одном АЛУ). Эта архитектура решает проблему разрыва между скоростью обработки в процессоре и временем доступа в основную память за счет переключения в каждом такте процессора на работу с очередным регистровым файлом. Каждый регистровый файл обслуживает один вычислительный процесс – тред (поток). Всего в каждом процессоре имеется n регистровых файлов, поэтому запрос, выданный в основную память каждым из потоков, может обслуживаться в течение $n-1$ такта, вплоть до момента, когда процессор снова переключится на тот же регистровый файл. Выбор значения n определяется отношением времени доступа в память ко времени выполнения команды процессором. Конечно, задача формирования потоков из последовательной программы должна, по возможности, решаться компилятором. В противном случае будущее этой архитектуры окажется ограниченным узкой проблемной ориентацией.

Концепция реализации параллелизма на уровне команд (Explicitly Parallel Instruction Computing) определяет новый тип архитектуры, способной конкурировать по масштабам влияния с RISC. Эта идеология направлена на то, чтобы упростить аппаратное обеспечение и, в то же время, извлечь как можно больше «скрытого параллелизма» на уровне команд, используя большую ширину «выдачи» команд (WIW -Wide Issue-Width) и длинные (глубокие) конвейеры с большой задержкой (DPL — Deep Pipeline-Latency), чем это можно сделать при реализации VLIW или суперскалярных стратегий. EPIC упрощает два ключевых момента, реализуемых во время выполнения. Во-первых, его принципы позволяют во время исполнения отказаться от проверки зависимостей между операциями, которые компилятор уже объявил как независимые. Во-вторых, данная архитектура позволяет отказаться от сложной логики внеочередного исполнения операций, полагаясь на порядок выдачи команд, определенный компилятором. Более того, EPIC совершенствует возможность компилятора статически генерировать планы выполнения за счет поддержки разного рода перемещений кода во время компиляции, которые были бы некорректными в последовательной архитектуре. Более ранние решения достигали этой цели главным образом за счет серьезного увеличения сложности аппаратного обеспечения, которое стало настолько значительным, что превратилась в препятствие, не позволяющее отрасли добиваться еще более высокой производительности. EPIC разработан именно для того, чтобы обеспечить более высокую степень параллелизма на уровне команд, поддерживая при этом приемлемую сложность аппаратного обеспечения.

Более высокая производительность достигается как за счет увеличения скорости передачи сигналов, так и благодаря увеличению плотности расположения функциональных устройств на кристалле. Зафиксировав рост этих двух составляющих, дальнейшего увеличения скорости выполнения программ можно добиться в первую очередь благодаря реализации определенного вида параллелизма. Так, параллелизм на уровне команд (ILP — Instruction-Level Parallelism) стал возможен благодаря созданию процессоров и методик компиляции, которые ускоряют работу за счет параллельного выполнения отдельных RISC-операций. Системы на базе ILP используют программы, написанные на традиционных языках высокого уровня, для последовательных процессоров, а обнаружение «скрытого

параллелизма» автоматически выполняется благодаря применению соответствующей компиляторной технологии и аппаратного обеспечения.

Тот факт, что эти методики не требуют от прикладных программистов дополнительных усилий, имеет крайне важное значение, поскольку данное решение резко отличается от традиционного микропроцессорного параллелизма, который предполагает, что программисты должны переписывать свои приложения. Параллельная обработка на уровне команд является единственным надежным подходом, позволяющим добиться увеличения производительности без фундаментальной переработки приложения.

Суперскалярные процессоры — это реализации ILP-процессора для последовательных архитектур, программа для которых не должна передавать и, фактически, не может передавать точную информацию о параллелизме. Поскольку программа не содержит точной информации о наличии ILP, задача обнаружения параллелизма должна решаться аппаратурой, которая, в свою очередь, должна создавать план действий для обнаружения «скрытого параллелизма». Процессоры VLIW представляют собой пример архитектуры, для которой программа предоставляет точную информацию о параллелизме — компилятор выявляет параллелизм в программе и сообщает аппаратному обеспечению какие операции не зависят друг от друга. Эта информация имеет важное значение для физического слоя, поскольку в этом случае он «знает» без дальнейших проверок какие операции можно начинать выполнять в одном и том же такте. Архитектура EPIC — это эволюция архитектуры VLIW, которая абсорбировала в себе многие концепции суперскалярной архитектуры, хотя и в форме, адаптированной к EPIC. По сути — это «идеология», определяющая, как создавать ILP-процессоры, а также набор характеристик архитектуры, которые поддерживают данную основу. В таком смысле EPIC похож на RISC: определяющий класс архитектур, подчиняющихся общим основным принципам. Точно также, как существует множество различных архитектур наборов команд (ISA) для RISC, может существовать и больше одной ISA для EPIC. В зависимости от того, какие из характеристик EPIC использует архитектура EPIC ISA, она может быть оптимизирована для различных приложений — например, для систем общего назначения или встроенных устройств. Первым примером коммерческой EPIC ISA стала архитектура IA-64.

Код для суперскалярных процессоров содержит последовательность команд, которая порождает корректный результат, если выполняется в установленном порядке. Код указывает последовательный алгоритм и, за исключением того, что он использует конкретный набор команд, не представляет себе точно природу аппаратного обеспечения, на котором он будет работать или точный временной порядок, в котором будут выполняться команды. В отличие от программ для суперскалярных процессоров, код VLIW предлагает точный план (POE — Plan Of Execution, схема исполнения создается статически во время компиляции) того, как процессор будет выполнять программу. Код точно указывает когда будет выполнена каждая операция, какие функциональные устройства будут работать и какие регистры будут содержать операнды. Компилятор VLIW создает такой план выполнения, имея полное представление о самом процессоре, чтобы добиться требуемой записи исполнения (ROE — Record Of Execution) — последовательности событий, которые действительно происходят во время работы программы. Компилятор передает ROE (через архитектуру набора команд, которая точно описывает параллелизм) аппаратному обеспечению, которое, в свою очередь, выполняет указанный план. Этот план позволяет VLIW использовать относительно простое аппаратное обеспечение, способное добиться высокого уровня ILP. В отличие от VLIW, суперскалярная аппаратура динамически строит ROE на основе последовательного кода. Хотя такой подход и увеличивает сложность физической реализации, суперскалярный процессор создает план,

используя преимущества тех факторов, которые могут быть определены только во время выполнения.

Одна из целей, которые ставили перед собой при создании EPIC, состояла в том, чтобы сохранить реализованный во VLIW принцип статического создания РОЕ, но и в то же время обогатить его возможностями, аналогичными возможностям суперскалярного процессора, позволяющими новой архитектуре лучше учитывать динамические факторы, традиционно ограничивающие параллелизм, свойственный VLIW. Чтобы добиться этих целей, «идеология» EPIC была построена на некоторых основных принципах.

Первый — это создание плана выполнения во время компиляции. EPIC возлагает нагрузку по созданию РОЕ на компилятор. Хотя, в общем, архитектура и физическая реализация могут препятствовать компилятору в выполнении этой задачи, процессоры EPIC предоставляют функции, которые помогают компилятору создавать план выполнения. Во время исполнения поведение процессора EPIC с точки зрения компилятора должно быть предсказуемым и управляемым. Динамическое внеочередное исполнение команд может «запутать» компилятор так, что он не будет «понимать», как его решения повлияют на реальную запись выполнения, созданную процессором, поэтому ему необходимо уметь предсказывать действия процессора, что еще больше усложняет задачу. В данной ситуации предпочтителен процессор, четко исполняющий то, что ему указывает программа. Суть же создания плана во время компиляции состоит в переупорядочивании исходного последовательного кода так, чтобы использовать все преимущества параллелизма приложения и максимально эффективно тратить аппаратные ресурсы, минимизируя время выполнения. Без соответствующей поддержки архитектуры такое переупорядочивание может нарушить корректность программы. Таким образом, поскольку EPIC возлагает создание РОЕ на компилятор, она должна обеспечивать еще и архитектурные возможности, поддерживающие интенсивное переупорядочивание кода во время компиляции.

Следующим принципом является использование компилятором вероятностных оценок. Компилятор EPIC сталкивается с серьезной проблемой при создании плана выполнения: информация определенного типа, которая существенно влияет на запись исполнения, становится известна только лишь в момент выполнения программы. Например, компилятор не может точно знать какая из ветвей после оператора перехода будет выполняться, когда запланированный код пройдет базовые блоки и какой из путей графа будет выбран. Кроме того, обычно невозможно создать статический план, который одновременно оптимизирует все пути в программе. Неоднозначность также возникает и в тех случаях, когда компилятор не может решить, будут ли ссылки указывать на одно и то же место в памяти. Если да, то обращение к ним должно осуществляться последовательно; если нет, то их можно запланировать в произвольном порядке. При такой неоднозначности часто наиболее вероятен некий конкретный результат. Одним из важнейших принципов EPIC в данной ситуации является возможность разрешения компилятору оперировать вероятностными оценками — он создает и оптимизирует РОЕ для наиболее вероятных случаев. Однако EPIC обеспечивает архитектурную поддержку, такую как спекулятивное выполнение по управлению и по данным (Control and Data Speculation), с тем, чтобы гарантировать корректность программы, даже если исходные предположения были не верны. Когда предположение оказывается неверным, совершенно очевидно падение производительности при выполнении программы. Такой эффект производительности иногда виден на плане программы, к примеру, в тех случаях, когда существует высоко оптимизированная программная область, а код исполняется в менее оптимизированной. Также падение производительности может возникнуть в моменты «остановки» (Stall), которые на плане программы не видны — определенные операции, подпадающие под наиболее вероятный и, следовательно, оптимизированный случай, выполняются при максимальной

производительности, но приостанавливают процессор для того, чтобы гарантировать корректность, если возникнет менее вероятный, не оптимизированный случай.

После того, как создан план, компилятор передает его аппаратному обеспечению. Для этого ISA должен обладать возможностями достаточно богатыми, чтобы сообщить решения компилятора о том, когда инициировать каждую операцию и какие ресурсы использовать (в частности, должен существовать способ указать, какие операции инициируются одновременно). В качестве альтернативного решения компилятор мог бы создавать последовательную программу, которую процессор динамически реорганизует с тем, чтобы получить требуемую запись. Но в таком случае цель, сводимая к освобождению аппаратного обеспечения от динамического планирования, не достигается. При передаче РОЕ аппаратному обеспечению крайне важно своевременно предоставить необходимую информацию. Примером этому может служить операция перехода, которая, в случае ее использования, требует, чтобы по адресу перехода команды выбирались с упреждением, заведомо до того, как будет инициирован сам переход. Вместо того, чтобы решение о том, когда это нужно сделать и какой адрес перехода отдавать на откуп аппаратному обеспечению, такая информация в соответствии с основными принципами EPIC передается аппаратному обеспечению точно и своевременно через код. Микроархитектура принимает и другие решения, не связанные напрямую с выполнением кода, но которые влияют на время выполнения. Один из таких примеров — управление иерархией кэш-памяти и соответствующие решения о том, какие данные нужны для поддержки иерархии, а какие следует заменить. Такие правила обычно предусматриваются алгоритмом функционирования контроллера кэша. EPIC расширяет принцип, утверждающий, что план выполнения компилятор создает так, чтобы тоже иметь возможность управлять этими механизмами микроархитектуры. Для этого обеспечиваются архитектурные возможности, позволяющие осуществлять программный контроль механизмами, которыми обычно управляет микроархитектура.

Стандарт IA-64.

Для большей конкретности рассмотрим кратко IA-64 как один из примеров воплощения VLIW. Со временем эта архитектура способна вытеснить x86 (IA-32) не только на рынке, но вообще как класс, хотя это уже удел далекого будущего. Тем не менее, необходимость разработки для IA-64 весьма сложных компиляторов и трудности с созданием оптимизированных машинных кодов может вызвать дефицит специалистов, работающих на ассемблере IA-64, особенно на начальных этапах, как самых сложных.

Наиболее кардинальным нововведением IA-64 по сравнению с RISC является явный параллелизм команд (EPIC), вносящий некоторые элементы, напоминающие архитектуру сверхдлинного командного слова, которые называли связками (bundle). Так, в обеих архитектурах явный параллелизм представлен уже на уровне команд, управляющих одновременной работой функциональных исполнительных устройств (или функциональных модулей, или просто функциональных устройств, ФУ).

В данном случае связка имеет длину 128bit и включает в себя 3 поля для команд длиной 41bit каждое, и 5-разрядный слот шаблона. Предполагается, что команды связки могут выполняться параллельно разными ФУ. Возможные взаимозависимости, препятствующие параллельному выполнению команд одной связки, отражаются в поле шаблона. Не утверждается, впрочем, что параллельно не могут выполняться и команды разных связок. Хотя, на основании заявленного уровня параллельности исполнения, достигающего шести команд за такт, логично предположить, что одновременно могут выполняться как минимум две связки.

Шаблон указывает какого непосредственно типа команды находятся в слотах связки. В общем случае однотипные команды могут выполняться в более чем одном типе функциональных устройств. Шаблоном задаются так называемые остановки, определяющие слот, после начала выполнения команд которого инструкции последующих полей должны ждать завершения. Порядок слотов в связке (более важные справа) отвечает и порядку байт (Little Endian), однако данные в памяти могут располагаться и в режиме Big Endian (более важные слева), который устанавливается специальным битом в регистре маски пользователя.

Вращение регистров является в некотором роде частным случаем переименования регистров, применяемого во многих современных суперскалярных процессорах с внеочередным спекулятивным («умозрительным») выполнением команд. В отличие от них, вращение регистров в IA-64 управляется программно. Использование этого механизма в IA-64 позволяет избежать накладных расходов, связанных с сохранением/восстановлением большого числа регистров при вызовах подпрограмм и возвратах из них, однако статические регистры при необходимости все-таки приходится сохранять и восстанавливать, явно кодируя соответствующие команды.

К слову, система команд IA-64 довольно уникальна. Среди принципиальных особенностей следует отдельно отметить спекулятивное выполнение команд и применение предикатов — именно это подмножество и определяет исключительность IA-64. Все подобные команды можно подразделить на команды работы со стеком регистров, целочисленные команды, команды сравнения и работы с предикатами, команды доступа в память, команды перехода, мультимедийные команды, команды пересылок между регистрами, «разные» (операции над строками и подсчет числа единиц в слове) и команды работы с плавающей запятой.

Аппаратная реализация VLIW-процессора очень проста: несколько небольших функциональных модулей (сложения, умножения, ветвления и т.д.), подключенных к шине процессора, и несколько регистров и блоков кэш-памяти. VLIW-архитектура представляет интерес для полупроводниковой промышленности по двум причинам. Первая — теперь на кристалле больше места может быть отведено для блоков обработки, а не, скажем, для блока предсказания переходов. Вторая причина — VLIW-процессор может быть высокоскоростным, так как предельная скорость обработки определяется только внутренними особенностями самих функциональных модулей. Привлекает и то, что VLIW при определенных условиях может реализовать старые CISC-инструкции эффективнее RISC. Это потому, что программирование VLIW-процессора очень напоминает написание микрокода (исключительно низкоуровневый язык, позволяющий всесторонне программировать физический слой, синхронизируя работу логических вентилей с шинами обмена данными и управляя передачей информации между функциональными модулями).

В те времена, когда память для ПК была дорогостоящей, программисты экономили ее, прибегая к сложным инструкциям процессора x86 типа STOS и LODS (косвенная запись/чтение в/из памяти). CISC реализует такие инструкции, как микропрограммы, зашитые в постоянную память (ROM) и выполняемые процессором. Архитектура RISC вообще исключает использование микрокода, реализуя инструкции чисто аппаратным путем — фактически, инструкции RISC-процессора почти аналогичны микрокоду, используемому в CISC. VLIW делает по-другому — изымает процедуру генерирования микрокода из процессора (да и вообще стадии исполнения) и переносит его в компилятор, на этап создания исполняемого кода. В результате эмуляция инструкций процессора x86, таких как STOS, осуществляется очень эффективно, поскольку процессор получает для исполнения уже готовые макросы. Но вместе с тем, это порождает и некоторые трудности,

поскольку написание достаточно эффективного микрокода — невероятно трудоемкий процесс. Архитектуре VLIW может обеспечить жизнеспособность только «умный» компилятор, который возьмет эту работу на себя. Именно это обстоятельство ограничивает использование вычислительных машин с архитектурой VLIW: пока они нашли свое применение в основном в векторных (для научных расчетов) и сигнальных процессорах.

Оптимизация программ под архитектуру микропроцессора.

Архитектура VLIW представляет собой одну из реализаций концепции внутреннего параллелизма в микропроцессорах. Их быстродействие можно повысить двумя способами: увеличив либо тактовую частоту, либо количество операций, выполняемых за один такт. В первом случае требуется применение «быстрых» технологий (например, использование арсенида галлия вместо кремния) и таких архитектурных решений, как глубинная конвейеризация (конвейеризация в пределах одного такта, когда в каждый момент времени задействованы все логические блоки кристалла, а не отдельные его части). Для увеличения количества выполняемых за один цикл операций необходимо на одном чипе разместить множество функциональных модулей обработки и обеспечить надежное параллельное исполнение машинных инструкций, что дает возможность включить в работу все модули одновременно. Надежность в таком контексте означает, что результаты вычислений будут правильными. Для примера рассмотрим два выражения, которые связаны друг с другом следующим образом: $A=B+C$ и $B=D+E$. Значение переменной A будет разным в зависимости от порядка, в котором вычисляются эти выражения (сначала A , а потом B , или наоборот), но ведь в программе подразумевается только одно определенное значение. И если теперь вычислить эти выражения параллельно, то на правильный результат можно рассчитывать лишь с определенной вероятностью, а не гарантировано.

Планирование порядка вычислений — довольно трудная задача, которую приходится решать при проектировании современного процессора. В суперскалярных архитектурах для распознавания зависимостей между машинными инструкциями применяется специальное довольно сложное аппаратное решение (например, в P6- и post-P6-архитектуре от Intel для этого используется буфер переупорядочивания инструкций — ReOrder Buffer, ROB). Однако размеры такого аппаратного планировщика при увеличении количества функциональных модулей обработки возрастают в геометрической прогрессии, что, в конце концов, может занять весь кристалл процессора. Поэтому суперскалярные проекты остановились на отметке 5-6 обрабатываемых за цикл инструкций. На самом же деле, текущие реализации VLIW тоже далеко не всегда могут похвастаться 100% заполнением пакетов — реальная загрузка около 6-7 команд в такте — примерно столько же, сколько и лидеры среди RISC-процессоров. При другом подходе можно передать все планирование программному обеспечению, как это делается в конструкциях с VLIW. «Умный» компилятор должен выискать в программе все инструкции, которые являются совершенно независимыми, собрать их вместе в очень длинные строки (длинные инструкции) и затем отправить на одновременное исполнение функциональными модулями, количество которых, как минимум, не меньше, чем количество операций в такой длинной команде. Очень длинные инструкции (VLIW) обычно имеют размер 256-1024 bit, однако, бывает и меньше. Сам же размер полей, кодирующих операции для каждого функционального модуля, в такой метаинструкции намного меньше.

Вновь вспыхнувший в последнее время интерес к VLIW как к архитектуре, которую можно использовать для реализации вычислений общего назначения, дал существенный толчок развитию техники VLIW-компиляции. Такой компилятор упаковывает группы независимых операций в очень длинные слова инструкций таким способом, чтобы обеспечить быстрый запуск и более эффективное их исполнение функциональными модулями. Компилятор

сначала обнаруживает все зависимости между данными, а затем определяет, как их развязать. Чаще всего это делается путем переупорядочивания всей программы — разные ее блоки перемещаются с одного места в другое. Данный подход отличается от применяемого в суперскалярном процессоре, который для определения зависимостей использует специальное аппаратное решение прямо во время выполнения приложения (оптимизирующие компиляторы, безусловно, улучшают работу суперскалярного процессора, но не делают его «привязанным» к ним). Большинство суперскалярных процессоров может обнаружить зависимости и планировать параллельное исполнение только внутри базовых программных блоков (группа последовательных операторов программы, не содержащих внутри себя останова или логического ветвления, допустимых только в конце). Некоторые переупорядочивающие системы положили начало расширению области сканирования, не ограничивая ее базовыми блоками. Для обеспечения большего параллелизма VLIW-компьютеры должны наблюдать за операциями из разных базовых блоков, чтобы поместить эти операции в одну и ту же длинную инструкцию (их «область обзора» должна быть шире, чем у суперскалярных процессоров) — это обеспечивается путем прокладки «маршрута» по всей программе (трассировка). Трассировка — наиболее оптимальный для некоторого набора исходных данных маршрут по программе (для обеспечения правильного результата гарантируется не пересечение этих данных), т.е. маршрут, который «проходит» по участкам, пригодным для параллельного выполнения (эти участки формируются, кроме всего прочего, и путем переноса кода из других мест программы), после чего остается упаковать их в длинные инструкции и передать на выполнение. Планировщик вычислений осуществляет оптимизацию на уровне всей программы, а не ее отдельных базовых блоков. Для VLIW, так же как и для RISC, ветвления в программе являются «врагом», препятствующим эффективному ее выполнению. В то время как RISC для прогнозирования ветвлений использует аппаратное решение, VLIW оставляет это компилятору. Сам компилятор использует информацию, собранную им путем профилирования программы, хотя у будущих VLIW-процессоров предполагается небольшое аппаратное расширение, обеспечивающее сбор для компилятора статистических данных непосредственно во время выполнения программы, что принципиально важно при циклической работе с переменным набором. Компилятор прогнозирует наиболее подходящий маршрут и планирует прохождение, рассматривая его как один большой базовый блок, затем повторяет этот процесс для всех других возникших после этого программных веток, и так до самого конца программы. Он также умеет делать при анализе кода и другие «интеллектуальные шаги», такие как развертывание программного цикла и IF-преобразование, в процессе которого временно удаляются все логические переходы из секции, подвергающейся трассировке. Там, где RISC может только просмотреть код вперед на предмет ветвлений, VLIW-компилятор перемещает его с одного места в другое до обнаруженного ветвления (согласно трассировке), но предусматривает при необходимости возможность отката назад, к предыдущему программному состоянию. Формально ничего не мешает это же сделать и RISC процессору, просто соотношение «цена/эффективность» оказывается слишком высоким. Соответствующее аппаратное обеспечение, добавленное к VLIW-процессору, может оказать определенную поддержку компилятору. Например, операции, имеющие по несколько ветвлений, могут входить в одну длинную инструкцию и, следовательно, выполняться за один машинный такт. Поэтому выполнение условных операций, которые зависят от предыдущих результатов, может быть реализовано программным способом, а не аппаратным. Цена, которую приходится платить за увеличение быстродействия VLIW-процессора, намного меньше стоимости компиляции — именно поэтому основные расходы приходятся на сами компиляторы.

Архитектура параллельных вычислительных систем

Архитектура параллельных вычислительных систем

SMP (symmetric multiprocessing) – симметричная многопроцессорная архитектура. Главной особенностью систем с архитектурой SMP является наличие общей физической памяти, разделяемой всеми процессорами.



Рис. 3.1. Схематический вид SMP-архитектуры

Память служит, в частности, для передачи сообщений между процессорами, при этом все вычислительные устройства при обращении к ней имеют равные права и одну и ту же адресацию для всех ячеек памяти. Поэтому SMP-архитектура называется симметричной. Последнее обстоятельство позволяет очень эффективно обмениваться данными с другими вычислительными устройствами. SMP-система строится на основе высокоскоростной системной шины (SGI PowerPath, Sun Gigaplane, DEC TurboLaser), к слотам которой подключаются функциональные блоки типов: процессоры (ЦП), подсистема ввода/вывода (I/O) и т. п. Для подсоединения к модулям I/O используются уже более медленные шины (PCI, VME64). Наиболее известными SMP-системами являются SMP-серверы и рабочие станции на базе процессоров Intel (IBM, HP, Compaq, Dell, ALR, Unisys, DG, Fujitsu и др.) Вся система работает под управлением единой ОС (обычно UNIX-подобной, но для Intel-платформ поддерживается Windows NT). ОС автоматически (в процессе работы) распределяет процессы по процессорам, но иногда возможна и явная привязка.

Основные преимущества SMP-систем:

простота и универсальность для программирования. Архитектура SMP не накладывает ограничений на модель программирования, используемую при создании приложения: обычно используется модель параллельных ветвей, когда все процессоры работают независимо друг от друга. Однако можно реализовать и модели, использующие межпроцессорный обмен. Использование общей памяти увеличивает скорость такого обмена, пользователь также имеет доступ сразу ко всему объему памяти. Для SMP-систем существуют довольно эффективные средства автоматического распараллеливания; простота эксплуатации. Как правило, SMP-системы используют систему кондиционирования, основанную на воздушном охлаждении, что облегчает их техническое обслуживание; относительно невысокая цена.

Недостатки:

системы с общей памятью плохо масштабируются.

Этот существенный недостаток SMP-систем не позволяет считать их по-настоящему перспективными. Причиной плохой масштабируемости является то, что в данный момент шина способна обрабатывать только одну транзакцию, вследствие чего возникают проблемы разрешения конфликтов при одновременном обращении нескольких процессоров

к одним и тем же областям общей физической памяти. Вычислительные элементы начинают друг другу мешать. Когда произойдет такой конфликт, зависит от скорости связи и от количества вычислительных элементов. В настоящее время конфликты могут происходить при наличии 8-24 процессоров. Кроме того, системная шина имеет ограниченную (хоть и высокую) пропускную способность (ПС) и ограниченное число слотов. Все это очевидно препятствует увеличению производительности при увеличении числа процессоров и числа подключаемых пользователей. В реальных системах можно задействовать не более 32 процессоров. Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры. При работе с SMP-системами используют так называемую парадигму программирования с разделяемой памятью (shared memory paradigm).

MPP (massive parallel processing) – массивно-параллельная архитектура. Главная особенность такой архитектуры состоит в том, что память физически разделена. В этом случае система строится из отдельных модулей, содержащих процессор, локальный банк операционной памяти (ОП), коммуникационные процессоры (рутеры) или сетевые адаптеры, иногда – жесткие диски и/или другие устройства ввода/вывода. По сути, такие модули представляют собой полнофункциональные компьютеры (см. рис.3.2). Доступ к банку ОП из данного модуля имеют только процессоры (ЦП) из этого же модуля. Модули соединяются специальными коммуникационными каналами. Пользователь может определить логический номер процессора, к которому он подключен, и организовать обмен сообщениями с другими процессорами. Используются два варианта работы операционной системы (ОС) на машинах MPP-архитектуры. В одном полноценная операционная система (ОС) работает только на управляющей машине (front-end), на каждом отдельном модуле функционирует сильно урезанный вариант ОС, обеспечивающий работу только расположенной в нем ветви параллельного приложения. Во втором варианте на каждом модуле работает полноценная UNIX-подобная ОС, устанавливаемая отдельно.

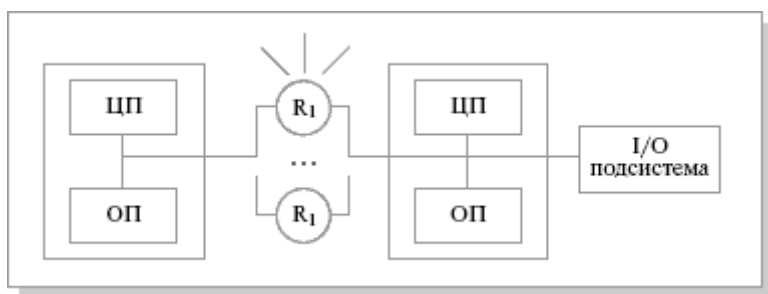


Рис. 3.2. Схематический вид архитектуры с разделяемой памятью

Главным преимуществом систем с разделяемой памятью является хорошая масштабируемость: в отличие от SMP-систем, в машинах с разделяемой памятью каждый процессор имеет доступ только к своей локальной памяти, в связи с чем не возникает необходимости в потактовой синхронизации процессоров. Практически все рекорды по производительности на сегодня устанавливаются на машинах именно такой архитектуры, состоящих из нескольких тысяч процессоров (ASCI Red, ASCI Blue Pacific).

Недостатки:

отсутствие общей памяти заметно снижает скорость межпроцессорного обмена, поскольку нет общей среды для хранения данных, предназначенных для обмена между процессорами. Требуется специальная техника программирования для реализации обмена сообщениями между процессорами;

каждый процессор может использовать только ограниченный объем локального банка памяти;
вследствие указанных архитектурных недостатков требуются значительные усилия для того, чтобы максимально использовать системные ресурсы. Именно этим определяется высокая цена программного обеспечения для массивно-параллельных систем с раздельной памятью.

Системами с раздельной памятью являются суперкомпьютеры MBC-1000, IBM RS/6000 SP, SGI/CRAY T3E, системы ASCI, Hitachi SR8000, системы Parsytec.

Машины последней серии CRAY T3E от SGI, основанные на базе процессоров Dec Alpha 21164 с пиковой производительностью 1200 Мфлопс/с (CRAY T3E-1200), способны масштабироваться до 2048 процессоров.

При работе с MPP-системами используют так называемую Massive Passing Programming Paradigm – парадигму программирования с передачей данных (MPI, PVM, BSPlib).

Гибридная архитектура NUMA. Главная особенность гибридной архитектуры NUMA (nonuniform memory access) – неоднородный доступ к памяти.

Гибридная архитектура совмещает достоинства систем с общей памятью и относительную дешевизну систем с раздельной памятью. Суть этой архитектуры – в особой организации памяти, а именно: память физически распределена по различным частям системы, но логически она является общей, так что пользователь видит единое адресное пространство. Система построена из однородных базовых модулей (плат), состоящих из небольшого числа процессоров и блока памяти. Модули объединены с помощью высокоскоростного коммутатора. Поддерживается единое адресное пространство, аппаратно поддерживается доступ к удаленной памяти, т.е. к памяти других модулей. При этом доступ к локальной памяти осуществляется в несколько раз быстрее, чем к удаленной. По существу, архитектура NUMA является MPP (массивно-параллельной) архитектурой, где в качестве отдельных вычислительных элементов берутся SMP (симметричная многопроцессорная архитектура) узлы. Доступ к памяти и обмен данными внутри одного SMP-узла осуществляется через локальную память узла и происходит очень быстро, а к процессорам другого SMP-узла тоже есть доступ, но более медленный и через более сложную систему адресации.

Структурная схема компьютера с гибридной сетью: четыре процессора связываются между собой при помощи кроссбара в рамках одного SMP-узла. Узлы связаны сетью типа "бабочка" (Butterfly):

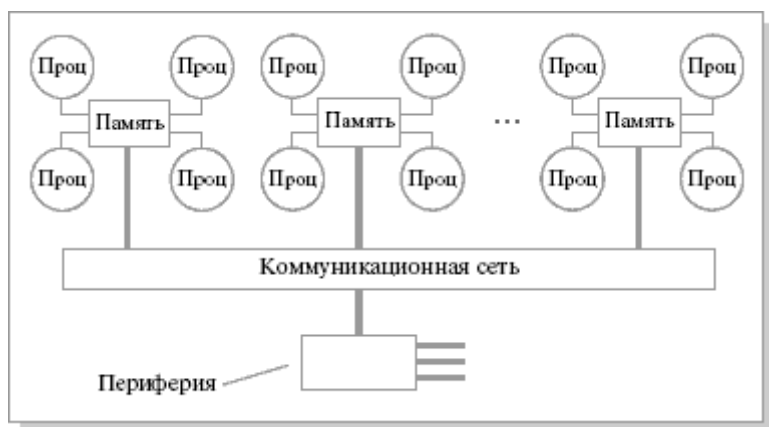


Рис. 3.3. Структурная схема компьютера с гибридной сетью

Впервые идею гибридной архитектуры предложил Стив Воллох, он воплотил ее в системах серии Exemplar. Вариант Воллоха – система, состоящая из восьми SMP-узлов. Фирма HP купила идею и реализовала на суперкомпьютерах серии SPP. Идею подхватил Сеймур Крей (Seymour R.Cray) и добавил новый элемент – когерентный кэш, создав так называемую архитектуру cc-NUMA (Cache Coherent Non-Uniform Memory Access), которая расшифровывается как "неоднородный доступ к памяти с обеспечением когерентности кэшей". Он ее реализовал на системах типа Origin.

Организация когерентности многоуровневой иерархической памяти

Понятие когерентности кэшей описывает тот факт, что все центральные процессоры получают одинаковые значения одних и тех же переменных в любой момент времени. Действительно, поскольку кэш-память принадлежит отдельному компьютеру, а не всей многопроцессорной системе в целом, данные, попадающие в кэш одного компьютера, могут быть недоступны другому. Чтобы этого избежать, следует провести синхронизацию информации, хранящейся в кэш-памяти процессоров.

Для обеспечения когерентности кэшей существует несколько возможностей: использовать механизм отслеживания шинных запросов (snoopy bus protocol), в котором кэши отслеживают переменные, передаваемые к любому из центральных процессоров и при необходимости модифицируют собственные копии таких переменных; выделять специальную часть памяти, отвечающую за отслеживание достоверности всех используемых копий переменных.

Наиболее известными системами архитектуры cc-NUMA являются: HP 9000 V-class в SCA-конфигурациях, SGI Origin3000, Sun HPC 15000, IBM/Sequent NUMA-Q 2000. На сегодня максимальное число процессоров в cc-NUMA-системах может превышать 1000 (серия Origin3000). Обычно вся система работает под управлением единой ОС, как в SMP. Возможны также варианты динамического "подразделения" системы, когда отдельные "разделы" системы работают под управлением разных ОС. При работе с NUMA-системами, так же, как с SMP, используют так называемую парадигму программирования с общей памятью (shared memory paradigm).

Гетерогенные распределенные вычислительные системы.

Наибольшее число существующих в настоящее время распределенных систем построено по схеме гетерогенных мультимикомпьютерных. Это означает, что компьютеры, являющиеся частями этой системы, могут быть крайне разнообразны, например, по типу процессора, размеру памяти и производительности каналов ввода-вывода. На практике роль некоторых из этих компьютеров могут исполнять высокопроизводительные параллельные системы, например мультипроцессорные или гомогенные мультимикомпьютерные.

Соединяющая их сеть также может быть сильно неоднородной. Так, например, авторы этой книги помогали разрабатывать самодельную распределенную компьютерную систему, названную DAS, состоящую из четырех кластеров мультимикомпьютерных систем, соединенных высокопроизводительными АТМ-коммутируемыми каналами. Фотографии этой системы и ссылки на исследования, проводимые на ней, можно найти по адресу <http://Www.cs.vu.nl-balldas.html>. Кластеры также были связаны между собой через стандартные Интернет-соединения. Каждый кластер содержал одинаковые процессоры (Pentium III) и соединяющую их сеть (Myrinet), но различался по числу процессоров (64–128).

Другим примером гетерогенности является создание крупных мультимикомпьютерных систем с использованием существующих сетей и каналов. Так, например, не является чем-то необычным существование кампусных университетских распределенных систем, состоящих из локальных сетей различных факультетов, соединенных между собой высокоскоростными каналами. В глобальных системах различные станции могут, в свою очередь, соединяться общедоступными сетями, например сетевыми службами, предлагаемыми коммерческими операторами связи, например SMDS или Frame relay.

В отличие от систем, обсуждавшихся в предыдущих пунктах, многие крупномасштабные гетерогенные мультимикомпьютерные системы нуждаются в глобальном подходе. Это означает, что приложение не может предполагать, что ему постоянно будет доступна определенная производительность или определенные службы. Так, в проекте I-way [147] несколько высокопроизводительных компьютерных центров были связаны через Интернет. Согласно общей модели системы предполагалось, что приложения будут резервировать и использовать ресурсы любого из центров, но полностью скрыть от приложений разницу между центрами оказалось невозможно.

Переходя к вопросам масштабирования, присущим гетерогенным системам, и учитывая необходимость глобального подхода, присущую большинству из них, заметим, что создание приложений для гетерогенных мультимикомпьютерных систем требует специализированного программного обеспечения. С этой проблемой распределенные системы справляются. Чтобы у разработчиков приложений не возникало необходимости волноваться об используемом аппаратном обеспечении, распределенные системы предоставляют программную оболочку, которая защищает приложения от того, что происходит на аппаратном уровне (то есть они обеспечивают прозрачность).

Метакомпьютинг.

Что такое мета-компьютинг? Этот термин возник вместе с развитием высокоскоростной сетевой инфраструктуры в начале 90-х годов и относился к объединению нескольких разнородных вычислительных ресурсов в локальной сети организации для решения одной задачи. Основная цель построения мета-компьютера в то время заключалась в оптимальном распределении частей работы по вычислительным системам различной архитектуры и различной мощности. Например, предварительная обработка данных и генерация сеток для счета могли производиться на пользовательской рабочей станции, основное моделирование на векторно-конвейерном суперкомпьютере, решение больших систем линейных уравнений - на массивно-параллельной системе, а визуализация результатов - на специальной графической станции.

В дальнейшем, исследования в области технологий мета-компьютинга были развиты в сторону однородного доступа к вычислительным ресурсам большого числа (вплоть до нескольких тысяч) компьютеров в локальной или глобальной сети. Компонентами "мета-компьютера" могут быть как простейшие ПК, так и мощные массивно-параллельные системы. Что важно, мета-компьютер может не иметь постоянной конфигурации - отдельные компоненты могут включаться в его конфигурацию или отключаться от нее; при этом технологии мета-компьютинга обеспечивают непрерывное функционирование системы в целом. Современные исследовательские проекты в этой области направлены на обеспечение прозрачного доступа пользователей через Интернет к необходимым распределенным вычислительным ресурсам, а также прозрачного подключения простаивающих вычислительных систем к мета-компьютерам.

Очевидно, что наилучшим образом для решения на мета-компьютерах подходят задачи переборного и поискового типа, где вычислительные узлы практически не взаимодействуют друг с другом и основную часть работы производят в автономном режиме. Основная схема работы в этом случае примерно такая: специальный агент, расположенный на вычислительном узле (компьютере пользователя), определяет факт простоты этого компьютера, соединяется с управляющим узлом мета-компьютера и получает от него очередную порцию работы (область в пространстве перебора). По окончании счета по данной порции вычислительный узел передает обратно отчет о фактически проделанном переборе или сигнал о достижении цели поиска.

Далее на данной странице будут вкратце описаны и приведены ссылки на основные исследовательские проекты в области мета-компьютинга, разработанные программные технологии, конкретные примеры мета-компьютеров.

Одно из самых больших объединений пользователей Интернет, предоставляющих свои компьютеры для решения крупных переборных задач. Основные проекты связаны с задачами взлома шифров (RSA Challenges). В частности, 19 января 1999 года была решена предложенная RSA Data Security задача расшифровки фразы, закодированной с помощью шифра DES-III. В настоящее время в distributed.net идет работа по расшифровке фразы, закодированной с 64-битным ключом (RC5-64). С момента начала проекта в нем зарегистрировались 191 тыс. человек. Достигнута скорость перебора, равная 75 млрд. ключей в секунду (всего требуется проверить 264 ключей). За решение этой задачи RSA предлагает приз в \$10 тыс.

Информация на русском языке.

GIMPS - Great Internet Mersenne Prime Search

<http://www.mersenne.org/>

Поиск простых чисел Мерсенна (т.е. простых чисел вида 2^p-1). С начала проекта было найдено 4 таких простых числа. Организация Electronic Frontier Foundation предлагает приз в \$100 тыс. за нахождение простого числа Мерсенна с числом цифр 10 млн.

SETI@home

<http://setiathome.ssl.berkeley.edu>

Проект SETI@home (Search for Extraterrestrial Intelligence) - поиск внеземных цивилизаций с помощью распределенной обработки данных, поступающих с радиотелескопа. Присоединится может любой желающий. Доступны клиентские программы для Windows, Mac, UNIX, OS/2 (клиент Windows срабатывает в качестве screen-saver'a). Для участия в проекте зарегистрировались около 920 тыс. человек.

Коммерческий проект TERRA ONE компании Cerentis ставит своей целью объединение множества персональных компьютеров, подключенных (или периодически подключаемых) к Интернет, для решения задач анализа информации, предоставляемой различными заказчиками. Клиентские компьютеры (TerraProcessor), подключенные к TERRA ONE, используются во время простаивания с помощью screen-saver'a. За обработку информации владельцы ПК получают возможность покупки в Интернет-магазинах - им начисляются "кредиты" (TerraPoints) за каждую единицу обработанной информации.

Globus

<http://www.globus.org>.

Разработка ПО для организации распределенных вычислений в Интернет. Проект реализуется в Argonne National Lab. Цель The Globus Project - построение т.н. "computational grids", включающих в себя вычислительные системы, системы визуализации,

экспериментальные установки. В рамках проекта проводятся исследования по построению распределенных алгоритмов, обеспечению безопасности и отказоустойчивости мета-компьютеров.

В рамках проекта Globus разработан ряд программных средств:

Globus Resource Allocation Manager - единообразный интерфейс к различным "локальным" системам распределения нагрузки (LSF, NQE, LoadLeveler) Для описания требований приложения к ресурсам разработан специальный язык RSL (Resource Specification Language)
Globus Security Infrastructure - система аутентификации на базе открытого ключа и X.09-сертификатов

Metacomputing Directory Service (MDS) - репозиторий информации о вычислительных ресурсах, входящих в метакомпьютер

Nexus - коммуникационная библиотека

Heartbeat Monitor (HBM) - средство мониторинга, позволяющее определить сбой некоторых машин и процессов, входящих в метакомпьютер

Globus Access to Secondary Storage (GASS) - средство доступа к удаленным данным через URL

Для тестирования Globus был создан реальный метакомпьютер GUSTO (testbed environment), который включает около 40 компонент с суммарной пиковой производительностью 2.5 TFLOPS.

The Metacomputing Project

Совместный проект Oak Ridge National Labs, Sandia National Labs и Pittsburgh Supercomputer Center В рамках данного проекта объединены 4 системы: три суперкомпьютера Intel XP/S 150 Paragon и суперкомпьютер T3D в PSC. Коммуникации обеспечиваются с помощью PVM. Для связи используются высокоскоростные сети ESNet и vBNS и протоколы ATM/AAL5.

Legion

<http://legion.virginia.edu/>

Проект Legion: A Worldwide Virtual Computer университета Вирджинии. Цель - разработка объектно-ориентированного ПО для построения виртуальных мета-компьютеров, включающих до нескольких миллионов индивидуальных хостов, объединенных высокоскоростными сетями. Пользователь, работающий на своем домашнем компьютере, должен иметь абсолютно прозрачный доступ ко всем ресурсам мета-компьютера.

В рамках Legion возможно исполнение параллельных приложений - поддерживаются библиотеки MPI и PVM, а также язык Mentat. Программное обеспечение проекта доступно бесплатно (поддерживаются платформы SGI IRIX, Linux, Alpha/OSF1, RS/6000).

PACX-MPI

<http://www.hlrs.de/structure/organisation/par/projects/pacx-mpi/>.

Расширение MPI для поддержки распределенных вычислений. Поддерживается объединение в единый мета-компьютер нескольких MPP-систем, возможно с различными реализациями MPI. Передача данных между MPP производится через Интернет с помощью TCP/IP. В настоящее время в рамках PACX-MPI реализовано подмножество стандарта MPI 1.2.

На конференции SC'98 продемонстрировано совместное использование двух 512-процессорных суперкомпьютеров Cray T3E, находящихся в университете Штутгарта (Германия) и в PSC (Питтсбург, США). Для объединения компьютеров использовалась библиотека PACX-MPI.

Condor

<http://www.cs.wisc.edu/condor/>

Система Condor разрабатывается в университете шт. Висконсин (Madison). Condor распределяет независимые подзадачи по существующей в организации сети рабочих станций, заставляя компьютеры работать в свободное время (то есть в то время, когда они простаивали бы без своих пользователей). Программное обеспечение системы Condor доступно бесплатно. В настоящее время поддерживаются платформы SGI, Solaris, Linux, HP-UX, и Digital Unix, однако планируется также поддержка Windows NT.

Кластерные архитектуры.

Кластер представляет собой два или более компьютеров (часто называемых узлами), объединяемые при помощи сетевых технологий на базе шинной архитектуры или коммутатора и предстающие перед пользователями в качестве единого информационно-вычислительного ресурса. В качестве узлов кластера могут быть выбраны серверы, рабочие станции и даже обычные персональные компьютеры. Узел характеризуется тем, что на нем работает единственная копия операционной системы. Преимущество кластеризации для повышения работоспособности становится очевидным в случае сбоя какого-либо узла: при этом другой узел кластера может взять на себя нагрузку неисправного узла, и пользователи не заметят прерывания в доступе. Возможности масштабируемости кластеров позволяют многократно увеличивать производительность приложений для большего числа пользователей технологий (Fast/Gigabit Ethernet, Myrinet) на базе шинной архитектуры или коммутатора. Такие суперкомпьютерные системы являются самыми дешевыми, поскольку собираются на базе стандартных комплектующих элементов ("off the shelf"), процессоров, коммутаторов, дисков и внешних устройств.

Кластеризация может осуществляться на разных уровнях компьютерной системы, включая аппаратное обеспечение, операционные системы, программы-утилиты, системы управления и приложения. Чем больше уровней системы объединены кластерной технологией, тем выше надежность, масштабируемость и управляемость кластера.

Типы кластеров

Условное деление на классы предложено Язеком Радаевским и Дугласом Эдлайном:

Класс I. Класс машин строится целиком из стандартных деталей, которые продают многие поставщики компьютерных компонентов (низкие цены, простое обслуживание, аппаратные компоненты доступны из различных источников).

Класс II. Система имеет эксклюзивные или не слишком широко распространенные детали. Таким образом можно достичь очень хорошей производительности, но при более высокой стоимости.

Как уже отмечалось, кластеры могут существовать в различных конфигурациях. Наиболее распространенными типами кластеров являются:

системы высокой надежности;

системы для высокопроизводительных вычислений;

многопоточные системы.

Отметим, что границы между этими типами кластеров до некоторой степени размыты, и кластер может иметь такие свойства или функции, которые выходят за рамки перечисленных типов. Более того, при конфигурировании большого кластера, используемого как система общего назначения, приходится выделять блоки, выполняющие все перечисленные функции.

Кластеры для высокопроизводительных вычислений предназначены для параллельных расчетов. Эти кластеры обычно собраны из большого числа компьютеров. Разработка таких кластеров является сложным процессом, требующим на каждом шаге согласования таких вопросов как инсталляция, эксплуатация и одновременное управление большим числом компьютеров, технические требования параллельного и высокопроизводительного доступа к одному и тому же системному файлу (или файлам) и межпроцессорная связь между узлами, и координация работы в параллельном режиме. Эти проблемы проще всего решаются при обеспечении единого образа операционной системы для всего кластера. Однако реализовать подобную схему удастся далеко не всегда и обычно она применяется лишь для не слишком больших систем.

Многопоточные системы используются для обеспечения единого интерфейса к ряду ресурсов, которые могут со временем произвольно наращиваться (или сокращаться). Типичным примером может служить группа web-серверов.

В 1994 году Томас Стерлинг (Sterling) и Дон Беккер (Becker) создали 16-узловой кластер из процессоров Intel DX4, соединенных сетью 10 Мбит/с Ethernet с дублированием каналов. Они назвали его "Beowulf" по названию старинной эпической поэмы. Кластер возник в центре NASA Goddard Space Flight Center для поддержки необходимыми вычислительными ресурсами проекта Earth and Space Sciences. Проектно-конструкторские работы быстро превратились в то, что известно сейчас как проект Beowulf. Проект стал основой общего подхода к построению параллельных кластерных компьютеров, он описывает многопроцессорную архитектуру, которая может с успехом использоваться для параллельных вычислений. Beowulf-кластер, как правило, является системой, состоящей из одного серверного узла (который обычно называется головным), а также одного или нескольких подчиненных (вычислительных) узлов, соединенных посредством стандартной компьютерной сети. Система строится с использованием стандартных аппаратных компонентов, таких как ПК, запускаемые под Linux, стандартные сетевые адаптеры (например, Ethernet) и коммутаторы. Нет особого программного пакета, называемого "Beowulf". Вместо этого имеется несколько кусков программного обеспечения, которые многие пользователи нашли пригодными для построения кластеров Beowulf. Beowulf использует такие программные продукты как операционная система Linux, системы передачи сообщений PVM, MPI, системы управления очередями заданий и другие стандартные продукты. Серверный узел контролирует весь кластер и обслуживает файлы, направляемые к клиентским узлам.

Архитектура кластерной системы (способ соединения процессоров друг с другом) в большей степени определяет ее производительность, чем тип используемых в ней процессоров. Критическим параметром, влияющим на величину производительности такой системы, является расстояние между процессорами. Так, соединив вместе 10 персональных компьютеров, мы получим систему для проведения высокопроизводительных вычислений. Проблема, однако, будет состоять в поиске наиболее эффективного способа соединения стандартных средств друг с другом, поскольку при увеличении производительности каждого процессора в 10 раз производительность системы в целом в 10 раз не увеличится.

Симметричные мультипроцессорные системы.

Ключевым понятием многопроцессорных архитектур является узел - вычислительная система, состоящая из одного или нескольких процессоров, имеющая оперативную память и систему ввода-вывода. Узел характеризуется тем, что на нем работает единственная копия операционной системы (ОС).

Симметричный многопроцессорный (SMP) узел содержит два или более одинаковых процессора, используемых равноправно. Все процессоры имеют одинаковый доступ к вычислительным ресурсам узла. Поскольку процессоры одновременно работают с данными, хранящимися в единой памяти узла, в SMP-архитектурах обязательно должен быть механизм, поддержки когерентности данных. Когерентность данных означает, что в любой момент времени для каждого элемента данных во всей памяти узла существует только одно его значение несмотря на то, что одновременно могут существовать несколько копий элемента данных, расположенных в разных видах памяти и обрабатываемых разными процессорами. Механизм когерентности должен следить за тем, чтобы операции с одним и тем же элементом данных выполнялись на разных процессорах последовательно, удаляя, в частности, устаревшие копии. В современных SMP-архитектурах когерентность реализуется аппаратными средствами.

Механизм когерентности является критичным для эффективной параллельной работы узла SMP и должен иметь малое время задержки. До сегодняшнего дня самые крупные SMP-системы содержали максимум 32 процессора на узел, что объяснялось требованием малых задержек когерентных связей, приводящим к архитектуре с одной объединительной платой, а это физически ограничивает возможное число подсоединенных процессоров и плат памяти. Поэтому для дальнейшего увеличения числа процессоров в узле приходится вместо аппаратно реализованной техники когерентности применять более медленную программную реализацию, что очень существенно сказывается на программируемости систем и их производительности.

SMP-узлы очень удобны для разработчиков приложений: операционная система почти автоматически масштабирует приложения, давая им возможность использовать наращиваемые ресурсы. Само приложение не должно меняться при добавлении процессоров и не обязано следить за тем, на каких ЦПУ оно работает. Временная задержка доступа от любого ЦПУ до всех частей памяти и системы ввода-вывода одна и та же. Разработчик оперирует с однородным адресным пространством. Все это приводит к тому, что SMP-архитектуры разных производителей выглядят в основном одинаково: упрощается переносимость программного обеспечения между SMP-системами. Переносимость программ - одно из основных достоинств SMP-платформ.

Типичные SMP-архитектуры в качестве аппаратной реализации механизма поддержки когерентности используют шину слежения (snoopy bus). Каждый процессор имеет свой собственный локальный кэш, где он хранит копию небольшой части основной памяти, доступ к которой наиболее вероятен. Для того чтобы все кэши оставались когерентными, каждый процессор "подглядывает" за шиной, осуществляя поиск тех операций считывания и записи между другими процессорами и основной памятью, которые влияют на содержимое их собственных кэшей. Если процессор "B" запрашивает ту часть памяти, которая обрабатывается процессором "A", то процессор "A" перехватывает этот запрос и помещает свои значения области памяти на шину, где "B" их считывает. Когда процессор "A" записывает измененное значение обратно из своего кэша в память, то все другие процессоры видят, как эта запись проходит по шине и удаляют устаревшие значения из своих кэшей.

Существует несколько вариантов SMP-узла с одной и несколькими системными шинами, однако последний вариант приводит к усложнению архитектуры и повышению ее стоимости. Одним из примеров может служить структура, реализованная компанией NCR, - две шины и общая разделяемая ими память. Когерентность в такой структуре реализуется путем хранения записей о состоянии и местонахождении каждого блока данных из

оперативной памяти. Такой тип работы с кэш-памятью называется кэшированием на основе каталога и обладает тем достоинством, что удваивает пропускную способность шины. Недостатками такого подхода являются необходимость более сложных аппаратных средств и дополнительные задержки при пересылке данных между памятью и обеими шинами.

В архитектуре Cray SuperServer 6400 SMP используются четыре шины. Все ЦПУ подсоединяются к каждой из четырех шин и реализуют протокол следящей шины для поддержки когерентности. Каждое ЦПУ отслеживает только те операции, которые влияют на содержимое своего кэша. Этот подход отличается от протоколов кэширования на основе каталога, так как здесь каталога нет. В системе Cray 6400 пропускная способность шины потенциально увеличивается в 4 раза, однако очевидным недостатком является то, что на каждом ЦПУ устанавливается четыре экземпляра аппаратных средств поддержки когерентности.

Новая система Sun Ultra Enterprise 6000 для подсоединения всех ЦПУ, памяти и систем ввода-вывода использует коммутатор, который заменяет традиционную объединительную плату, но по существу выполняет ту же функцию. Соответственно, сохраняются те же недостатки, поскольку весь трафик между памятью, ЦПУ и системой ввода-вывода должен проходить через коммутатор. Система имеет только 16 слотов для плат ЦПУ/память и ввода-вывода. Хотя такой новый коммутатор несколько увеличивает пропускную способность шины, проблема "большой шины" остается: требование малых задержек ограничивает в этих архитектурах число присоединенных ЦПУ, а увеличение скорости в шинах или коммутаторах не соответствует росту производительности ЦПУ.

Матричные мультипроцессорные системы.

Если не считать машины Унгера (1958г.), которая была узкоспециализированной и предназначалась только для решения задач распознавания образов, то, по-видимому, первой матричной системой следует считать систему SOLOMON. Система содержит 1024 ПЭ, соединенных в виде матрицы 32X32. Каждый ПЭ в матрице соединен с четырьмя соседними и включает в себя процессор, обеспечивающий выполнение последовательных поразрядных арифметических и логических операций, а также оперативное ЗУ емкостью 16 кбит, разбитое на модули по 4 кбит каждый. Длина слова переменная - от 1 до 128 разрядов. Разрядность слов устанавливается программно. По каналам связи от УУ передаются команды и общие константы. В ПЭ используется так называемая многомодальная логика, которая позволяет каждому ПЭ выполнять (т.е. быть активным) или не выполнять (быть пассивным) общую операцию в зависимости от значений обрабатываемых данных.

В каждый момент все активные ПЭ выполняют одну и ту же операцию над данными, хранящимися в собственной памяти и имеющими один и тот же адрес. Идея многомодальности заключается в том, что в каждом ПЭ имеется специальный регистр на четыре состояния - регистр моды.

Мода (или модальность) заносится в этот регистр от УУ. При выполнении последовательности команд модальность передается в коде операции и сравнивается с содержимым регистра моды. Если есть совпадение, то операции выполняется. В других случаях ПЭ не выполняет операцию, но может в зависимости от кода пересылать свои операнды соседнему ПЭ. Такой механизм позволяет, в частности, выделить строку или столбец ПЭ, что может быть полезным при операциях над матрицами. Взаимодействуют ПЭ с периферийным оборудованием через внешние ПЭ.

Система SOLOMON оказалась нежизнеспособной вследствие громоздкости, недостаточной гибкости и эффективности. Однако идеи, заложенные в ней, получили развитие в системе

ILLIAC-IV, разработанной Иллинойским университетом и изготовленной фирмой "Барроуз".

В каждом квадранте 64 ПЭ образуют матрицу размером 8X8. Схема связей между ПЭ похожа на схему системы SOLOMON, но связь с внешней средой имеют все ПЭ без исключения. Реально действующая система ILLIAC-IV состоит, таким образом, из двух частей; центральной с устройством управления и 64 ПЭ, а также подсистемы ввода - вывода, включающей в себя универсальную ЭВМ B-6700, файловые диски и лазерную архивную память большой емкости. Каждый ПЭ состоит из собственно процессора и ОЗУ. Процессор оперирует с 64-разрядными числами и выполняет универсальный набор операций. Быстродействие процессора достаточно высокое: операция сложения 64-разрядных чисел выполняется за 240 нс, а умножения - за 400 нс. Таким образом, процессор выполняет в среднем 3 млн. операций в секунду, а следовательно производительность системы равна $3 \times 64 \times 200$ млн. операций в секунду.

Емкость ОЗУ каждого ПЭ составляет 2048 64-разрядных слов, длительность цикла обращения к памяти 350 нс. Память выполнена на интегральных схемах. Каждый процессор имеет счетчик адресов и индексный регистр, так что конечный адрес в каждом процессоре может формироваться как сумма трех составляющих: адреса, содержащегося в команде для данного ПЭ, кода. Это существенно повышает гибкость системы по сравнению с системой SOLOMON, где все ПЭ выбирают информацию по одному адресу. Каждый процессор кроме индексного регистра имеет в своем составе пять программно-адресуемых регистров: накапливающийся сумматор, регистр для операндов, регистр пересылок, используемый при передачах от одного ПЭ к другому, буферный регистр на одно слово и регистр управления состоянием ПЭ (аналогичный регистру моды в системе SOLOMON). Регистр управления имеет 8 разрядов. В зависимости от содержимого этого регистра ПЭ становится активным или пассивным, а также выполняет ряд пересылочных операций. Если вычисления не требуют полной разрядности, то процессор может быть разбит на два 32-разрядных подпроцессора или даже восемь 8-разрядных. Это позволяет в случае необходимости обрабатывать векторные операнды из 64, $2 \times 64 = 128$ и $8 \times 64 = 512$ элементов.

Система ILLIAC была включена в состав вычислительной сети АКРА. В результате усовершенствования программного обеспечения производительность системы выросла до 300 млн. операций в секунду.

Классификации вычислителей по Флинну.

По-видимому, самой ранней и наиболее известной является классификация архитектур вычислительных систем, предложенная в 1966 году М.Флинном [1,2]. Классификация базируется на понятии потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD, MISD, SIMD, MIMD.

SISD (single instruction stream / single data stream) - одиночный поток команд и одиночный поток данных. К этому классу относятся, прежде всего, классические последовательные машины, или иначе, машины фон-неймановского типа, например, PDP-11 или VAX 11/780. В таких машинах есть только один поток команд, все команды обрабатываются последовательно друг за другом и каждая команда инициирует одну операцию с одним потоком данных. Не имеет значения тот факт, что для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка - как машина CDC 6600 со скалярными функциональными устройствами, так и CDC 7600 с конвейерными попадают в этот класс.

SIMD (single instruction stream / multiple data stream) - одиночный поток команд и множественный поток данных. В архитектурах подобного рода сохраняется один поток команд, включающий, в отличие от предыдущего класса, векторные команды. Это позволяет выполнять одну арифметическую операцию сразу над многими данными - элементами вектора. Способ выполнения векторных операций не оговаривается, поэтому обработка элементов вектора может производиться либо процессорной матрицей, как в ILLIAC IV, либо с помощью конвейера, как, например, в машине CRAY-1.

Машины типа SIMD.

Машины типа SIMD состоят из большого числа идентичных процессорных элементов, имеющих собственную память. Все процессорные элементы в такой машине выполняют одну и ту же программу. Очевидно, что такая машина, составленная из большого числа процессоров, может обеспечить очень высокую производительность только на тех задачах, при решении которых все процессоры могут делать одну и ту же работу. Модель вычислений для машины SIMD очень похожа на модель вычислений для векторного процессора: одиночная операция выполняется над большим блоком данных.

В отличие от ограниченного конвейерного функционирования векторного процессора, матричный процессор (синоним для большинства SIMD-машин) может быть значительно более гибким. Обработываемые элементы таких процессоров - это универсальные программируемые ЭВМ, так что задача, решаемая параллельно, может быть достаточно сложной и содержать ветвления. Обычное проявление этой вычислительной модели в исходной программе примерно такое же, как и в случае векторных операций: циклы на элементах массива, в которых значения, вырабатываемые на одной итерации цикла, не используются на другой итерации цикла.

Модели вычислений на векторных и матричных ЭВМ настолько схожи, что эти ЭВМ часто обсуждаются как эквивалентные.

MISD (multiple instruction stream / single data stream) - множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Однако ни Флинн, ни другие специалисты в области архитектуры компьютеров до сих пор не смогли представить убедительный пример реально существующей вычислительной системы, построенной на данном принципе. Ряд исследователей [3,4,5] относят конвейерные машины к данному классу, однако это не нашло окончательного признания в научном сообществе. Будем считать, что пока данный класс пуст.

MIMD (multiple instruction stream / multiple data stream) - множественный поток команд и множественный поток данных. Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.

Машины типа MIMD.

Термин "мультипроцессор" покрывает большинство машин типа MIMD и (подобно тому, как термин "матричный процессор" применяется к машинам типа SIMD) часто используется в качестве синонима для машин типа MIMD. В мультипроцессорной системе каждый процессорный элемент (ПЭ) выполняет свою программу достаточно независимо от других процессорных элементов. Процессорные элементы, конечно, должны как-то связываться друг с другом, что делает необходимым более подробную классификацию машин типа MIMD. В мультипроцессорах с общей памятью (сильносвязанных мультипроцессорах) имеется память данных и команд, доступная всем ПЭ. С общей памятью ПЭ связываются с

помощью общей шины или сети обмена. В противоположность этому варианту в слабосвязанных многопроцессорных системах (машинах с локальной памятью) вся память делится между процессорными элементами и каждый блок памяти доступен только связанному с ним процессору. Сеть обмена связывает процессорные элементы друг с другом.

Базовой моделью вычислений на MIMD-мультипроцессоре является совокупность независимых процессов, эпизодически обращающихся к разделяемым данным. Существует большое количество вариантов этой модели. На одном конце спектра - модель распределенных вычислений, в которой программа делится на довольно большое число параллельных задач, состоящих из множества подпрограмм. На другом конце спектра - модель потоковых вычислений, в которых каждая операция в программе может рассматриваться как отдельный процесс. Такая операция ждет своих входных данных (операндов), которые должны быть переданы ей другими процессами. По их получении операция выполняется, и полученное значение передается тем процессам, которые в нем нуждаются. В потоковых моделях вычислений с большим и средним уровнем гранулярности, процессы содержат большое число операций и выполняются в потоковой манере.

Итак, что же собой представляет каждый класс? В SISD, как уже говорилось, входят однопроцессорные последовательные компьютеры типа VAX 11/780. Однако, многими критиками подмечено, что в этот класс можно включить и векторно-конвейерные машины, если рассматривать вектор как одно неделимое данное для соответствующей команды. В таком случае в этот класс попадут и такие системы, как CRAY-1, CYBER 205, машины семейства FACOM VP и многие другие.

Бесспорными представителями класса SIMD считаются матрицы процессоров: ILLIAC IV, ICL DAP, Goodyear Aerospace MPP, Connection Machine 1 и т.п. В таких системах единое управляющее устройство контролирует множество процессорных элементов. Каждый процессорный элемент получает от устройства управления в каждый фиксированный момент времени одинаковую команду и выполняет ее над своими локальными данными. Для классических процессорных матриц никаких вопросов не возникает, однако в этот же класс можно включить и векторно-конвейерные машины, например, CRAY-1. В этом случае каждый элемент вектора надо рассматривать как отдельный элемент потока данных.

Класс MIMD чрезвычайно широк, поскольку включает в себя всевозможные мультипроцессорные системы: Cm*, C.mmp, CRAY Y-MP, Denelcor HEP, BBN Butterfly, Intel Paragon, CRAY T3D и многие другие. Интересно то, что если конвейерную обработку рассматривать как выполнение множества команд (операций ступеней конвейера) не над одиночным векторным потоком данных, а над множественным скалярным потоком, то все рассмотренные выше векторно-конвейерные компьютеры можно расположить и в данном классе.

Предложенная схема классификации вплоть до настоящего времени является самой применяемой при начальной характеристике того или иного компьютера. Если говорится, что компьютер принадлежит классу SIMD или MIMD, то сразу становится понятным базовый принцип его работы, и в некоторых случаях этого бывает достаточно. Однако видны и явные недостатки. В частности, некоторые заслуживающие внимания архитектуры, например dataflow и векторно-конвейерные машины, четко не вписываются в данную классификацию. Другой недостаток - это чрезмерная заполненность класса MIMD. Необходимо средство, более избирательно систематизирующее архитектуры, которые по Флинну попадают в один класс, но совершенно различны по числу процессоров, природе и

топологии связи между ними, по способу организации памяти и, конечно же, по технологии программирования.

Наличие пустого класса (MISD) не стоит считать недостатком схемы. Такие классы, по мнению некоторых исследователей в области классификации архитектур [6,7], могут стать чрезвычайно полезными для разработки принципиально новых концепций в теории и практике построения вычислительных систем.

Масштабируемость мультипроцессорных вычислителей.

Масштабируемость представляет собой возможность наращивания числа и мощности процессоров, объемов оперативной и внешней памяти и других ресурсов вычислительной системы. Масштабируемость должна обеспечиваться архитектурой и конструкцией компьютера, а также соответствующими средствами программного обеспечения.

Так, например, возможность масштабирования кластера ограничена значением отношения скорости процессора к скорости связи, которое не должно быть слишком большим (реально это отношение для больших систем не может быть более 3-4, в противном случае не удастся даже реализовать режим единого образа операционной системы). С другой стороны, последние 10 лет истории развития процессоров и коммутаторов показывают, что разрыв в скорости между ними все увеличивается. Добавление каждого нового процессора в действительно масштабируемой системе должно давать прогнозируемое увеличение производительности и пропускной способности при приемлемых затратах. Одной из основных задач при построении масштабируемых систем является минимизация стоимости расширения компьютера и упрощение планирования. В идеале добавление процессоров к системе должно приводить к линейному росту ее производительности. Однако это не всегда так. Потери производительности могут возникать, например, при недостаточной пропускной способности шин из-за возрастания трафика между процессорами и основной памятью, а также между памятью и устройствами ввода/вывода. В действительности реальное увеличение производительности трудно оценить заранее, поскольку оно в значительной степени зависит от динамики поведения прикладных задач.

Возможность масштабирования системы определяется не только архитектурой аппаратных средств, но зависит от свойств программного обеспечения. Масштабируемость программного обеспечения затрагивает все его уровни, от простых механизмов передачи сообщений до работы с такими сложными объектами как мониторы транзакций и вся среда прикладной системы. В частности, программное обеспечение должно минимизировать трафик межпроцессорного обмена, который может препятствовать линейному росту производительности системы. Аппаратные средства (процессоры, шины и устройства ввода/вывода) являются только частью масштабируемой архитектуры, на которой программное обеспечение может обеспечить предсказуемый рост производительности. Важно понимать, что, например, простой переход на более мощный процессор может привести к перегрузке других компонентов системы. Это означает, что действительно масштабируемая система должна быть сбалансирована по всем параметрам.

Управление памятью в мультипроцессорных системах.

Системы класса: SMP (Scalable Parallel Processor) состоят из нескольких однородных процессоров и массива общей памяти (разделяемой памяти – shared memory): любой процессор может обращаться к любому элементу памяти. По этой схеме построены 2,4 процессорные SMP сервера на базе процессоров Intel, HP и т. д., причем процессоры подключены к памяти с помощью общей шины. Системы с большим числом процессоров

(но не более 32) подключаются к общей памяти, разделенной на блоки, через не блокирующийся полный коммутатор: crossbar. Любой процессор системы получает данное по произвольному адресу памяти за одинаковое время, такая структура памяти называется: **UMA** - Uniform Memory Access (Architecture). Пример: HP-9000. Дальнейшее **масштабирование** (увеличение числа процессоров системы) SMP систем обеспечивается переходом к архитектуре памяти: **NUMA** - Non Uniform Memory Access. По схеме, называемой, этой иногда, кластеризацией SMP, соответствующие блоки памяти двух (или более) серверов соединяются кольцевой связью, обычно по GCI интерфейсу. При запросе данного, расположенного вне локального с сервера диапазона адресов, это данное по кольцевой связи переписывается дублируется в соответствующий блок локальной памяти, ту часть его, которая специально отводится для буферизации глобальных данных и из этого буфера поставляется потребителю. Эта буферизация прозрачна (невидима) пользователю, для которого вся память кластера имеет сквозную нумерацию, и время выборки данных, не локальных в сервере, будет равно времени выборки локальных данных при повторных обращениях к глобальному данному, когда оно уже переписано в буфер. Данный аппарат буферизации есть типичная схема кэш памяти. Так как к данным возможно обращение из любого процессора кластера, то буферизация, размножение данных требует обеспечение их когерентности. Когерентность данных состоит в том, что при изменении данного все его потребители должны получать это значение. Проблема когерентности усложняется дублированием данных еще и в процессорных кэшах системы. Системы, в которых обеспечена когерентность данных, буферизуемых в кэшах, называются кэш когерентными (cc-cache coherent), а архитектура памяти описываемого кластера: cc-NUMA (cache coherent Non Uniform Memory Access). Классической архитектурой принято считать систему SPP1000.

Не уверен, что это нужно...

Распределенная общая память (DSM - Distributed Shared Memory)

Традиционно распределенные вычисления базируются на модели передачи сообщений, в которой данные передаются от процессора к процессору в виде сообщений. Удаленный вызов процедур фактически является той же самой моделью (или очень близкой).

DSM - виртуальное адресное пространство, разделяемое всеми узлами (процессорами) распределенной системы. Программы получают доступ к данным в DSM примерно так же, как они работают с данными в виртуальной памяти традиционных ЭВМ. В системах с DSM данные перемещаются между локальными памятьями разных компьютеров аналогично тому, как они перемещаются между оперативной и внешней памятью одного компьютера.

6.1 Достоинства DSM

В модели передачи сообщений программист обеспечивает доступ к разделяемым данным посредством явных операций отправки и приема сообщений. При этом приходится квантовать алгоритм, обеспечивать своевременную смену информации в буферах, преобразовывать индексы массивов. Все это сильно усложняет программирование и отладку. DSM скрывает от программиста пересылку данных и обеспечивает ему абстракцию разделяемой памяти, к использованию которой он уже привык на мультипроцессорах. Программирование и отладка с использованием DSM гораздо проще. В модели передачи сообщений данные перемещаются между двумя различными адресными пространствами. Это делает очень трудным передачу сложных структур данных между процессами. Более того, передача данных по ссылке и передача структур данных, содержащих указатели, является в общем случае делом сложным и дорогостоящим. DSM же позволяет передавать данные по ссылке, что упрощает разработку распределенных приложений.

Объем суммарной физической памяти всех узлов может быть огромным. Эта огромная память становится доступна приложению без издержек, связанных в традиционных системах с дисковыми обменами. Это достоинство становится все весомее в связи с тем, что скорости процессоров растут быстрее скоростей памяти и в то же время появляются очень быстрые коммуникации.

DSM-системы могут наращиваться практически беспредельно в отличие от систем с разделяемой памятью, т.е. являются масштабируемыми.

Программы, написанные для мультипроцессоров с общей памятью, могут в принципе без каких-либо изменений выполняться на DSM-системах (по крайней мере, они могут быть легко перенесены на DSM-системы). По существу, DSM-системы преодолевают архитектурные ограничения мультипроцессоров и сокращают усилия, необходимые для написания программ для распределенных систем. Обычно они реализуются программно-аппаратными средствами, но в последние годы появилось несколько коммерческих MPP с DSM, реализованной аппаратно (Convex SPP, KSR1).

6.2 Алгоритмы реализации DSM

При реализации DSM центральными являются следующие вопросы. как поддерживать информацию о расположении удаленных данных. как снизить при доступе к удаленным данным коммуникационные задержки и большие накладные расходы, связанные с выполнением коммуникационных протоколов. как сделать разделяемые данные доступными одновременно на нескольких узлах для того, чтобы повысить производительность системы.

Рассмотрим четыре основных алгоритма реализации DSM.

6.2.1 Алгоритм с центральным сервером

Все разделяемые данные поддерживает центральный сервер. Он возвращает данные клиентам по их запросам на чтение, по запросам на запись он корректирует данные и посылает клиентам в ответ квитанции. Клиенты могут использовать тайм-аут для отправки повторных запросов при отсутствии ответа сервера. Дубликаты запросов на запись могут распознаваться путем нумерации запросов. Если несколько повторных обращений к серверу остались без ответа, приложение получит отрицательный код ответа (это обеспечит клиент). Алгоритм прост в реализации, но сервер может стать узким местом.

Чтобы избежать этого, разделяемые данные могут быть распределены между несколькими серверами. В этом случае клиент должен уметь определять, к какому серверу надо обращаться при каждом доступе к разделяемой переменной. Посылка запросов сразу всем серверам нежелательна, поскольку не снижает нагрузку на серверы. Лучшее решение - распределить данные в зависимости от их адресов и использовать функцию отображения для определения нужного сервера.

6.2.2 Миграционный алгоритм

В отличие от предыдущего алгоритма, когда запрос к данным направлялся в место их расположения, в этом алгоритме меняется расположение данных - они перемещаются в то место, где потребовались. Это позволяет последовательные обращения к данным осуществлять локально. Миграционный алгоритм позволяет обращаться к одному элементу данных в любой момент времени только одному узлу.

Обычно мигрирует целиком страницы или блоки данных, а не запрашиваемые единицы данных. Это позволяет воспользоваться присущей приложениям локальностью доступа к данным для снижения стоимости миграции. Однако, такой подход приводит к трэшингу, когда страницы очень часто мигрируют между узлами при малом количестве обслуживаемых запросов. Некоторые системы позволяют задать время, в течение которого страница насильно удерживается в узле для того, чтобы успеть выполнить несколько обращений к ней до миграции ее в другой узел.

Миграционный алгоритм позволяет интегрировать DSM с виртуальной памятью, обеспечивающейся операционной системой в отдельных узлах. Если размер страницы DSM совпадает с размером страницы виртуальной памяти (или кратен ей), то можно обращаться к разделяемой памяти обычными машинными командами, воспользовавшись аппаратными средствами проверки наличия в оперативной памяти требуемой страницы и замены виртуального адреса на физический. Конечно, для этого виртуальное адресное пространство процессоров должно быть достаточно, чтобы адресовать всю разделяемую память. При этом, несколько процессов в одном узле могут разделять одну и ту же страницу.

Для определения места расположения блоков данных миграционный алгоритм может использовать сервер, отслеживающий перемещения блоков, либо воспользоваться механизмом подсказок в каждом узле. Возможна и широковещательная рассылка запросов.

6.2.3 Алгоритм размножения для чтения

Предыдущий алгоритм позволял обращаться к разделяемым данным в любой момент времени только процессам в одном узле (в котором эти данные находятся). Данный алгоритм расширяет миграционный алгоритм механизмом размножения блоков данных, позволяя либо многим узлам иметь возможность одновременного доступа по чтению, либо одному узлу иметь возможность читать и писать данные (протокол многих читателей и одного писателя). Производительность повышается за счет возможности одновременного доступа по чтению, но запись требует серьезных затрат для уничтожения всех устаревших копий блока данных или их коррекции.

При использовании такого алгоритма требуется отслеживать расположение всех блоков данных и их копий. Например, каждый собственник блока может отслеживать расположение его копий.

Данный алгоритм может снизить среднюю стоимость доступа по чтению тогда, когда количество чтений значительно превышает количество записей.

6.2.4 Алгоритм полного размножения

Этот алгоритм является расширением предыдущего алгоритма. Он позволяет многим узлам иметь одновременный доступ к разделяемым данным на чтение и запись (протокол многих читателей и многих писателей). Поскольку много узлов могут писать данные параллельно, требуется для поддержания согласованности данных контролировать доступ к ним.

Одним из способов обеспечения консистентности данных является использование специального процесса для упорядочивания модификаций памяти. Все узлы, желающие модифицировать разделяемые данные должны посылать свои модификации этому процессу. Он будет присваивать каждой модификации очередной номер и рассылать его широковещательно вместе с модификацией всем узлам, имеющим копию модифицируемого блока данных. Каждый узел будет осуществлять модификации в порядке возрастания их номеров. Разрыв в номерах полученных модификаций будет означать потерю одной или нескольких модификаций. В этом случае узел может запросить недостающие модификации.

Все перечисленные алгоритмы являются неэффективными. Добиться эффективности можно только изменив семантику обращений к памяти.

6.5.1 Страничная DSM

Общая память разбивается на порции одинаковой длины - страницы или блоки. Если выбрать длину совпадающей (или кратной) длине страницы оперативной памяти

процессоров (если их память страничная), то можно будет воспользоваться механизмом защиты памяти для обнаружения отсутствующих страниц DSM и аппаратным механизмом замены виртуального адреса на физический.

К этому же типу DSM (не знающих заранее ничего о содержимом памяти) можно отнести и аппаратные реализации на базе кэшей (Convex SPP).

6.5.2 DSM на базе разделяемых переменных

В DSM системах с разделяемыми переменными только отдельные структуры данных разделяются между процессорами. Программист должен точно определить какие переменные в программе должны разделяться, а какие не должны.

Знание информации о режиме использования разделяемых переменных позволяет воспользоваться более эффективными протоколами когерентности.

Рассмотрим следующую программу, которая автоматически распараллеливается посредством размножения всех переменных на все процессоры и распределения между ними витков цикла.

```
s = 0;
izer = 0;
for (i = 1; i < L2-1; i++)
{
    r = A[i-1]*A[i+1];
    C[i] = r;
    s = s + r*r;
    if (A[i]==0) { izer = i};
}
A[izer] = 1;
```

Переменные внутри цикла используются в следующих режимах:

A - только на чтение;

r - приватно (фактически не является разделяемой);

C - раздельно используется (любой элемент массива изменяется не более, чем одним процессом, и никакой процесс не читает элемент, изменяемый другими);

s - редукционная переменная, используемая для суммирования;

izer - переменная, хранящая индекс последнего нулевого элемента массива A;

Приведения в консистентное состояние переменных A и r - не требуется.

Для приведения в консистентное состояние массива C необходимо при завершении цикла каждому процессу послать остальным все свои изменения в массиве C.

Для переменной s в конце цикла надо довыполнить операцию редукции - сложить все частичные суммы, полученные разными процессорами в своих копиях переменной s и разослать результат всем процессорам (если бы начальное значение переменной s было отлично от нуля, то это надо было бы учесть).

Переменной izer на всех процессорах при выходе из цикла должно быть присвоено то значение, которое было получено на витке цикла с максимальным номером (требуется фиксировать на каждом процессоре максимальный номер витка, на котором переменной присваивается значение. При распределении витков последовательными блоками между процессорами достаточно фиксировать сам факт изменения переменной каждым процессором).

Вне цикла приведение в консистентное состояние переменной A[izer] не требуется, поскольку на каждом процессоре выполняется один и тот же оператор, который присваивает одно и то же значение всем копиям переменной.

6.5.3 DSM на базе объектов

Последнюю группу образуют многопроцессорные системы с объектной организацией распределенной общей памятью. В отличие от всех остальных рассмотренных систем, программы для объектно-ориентированной DSM системы не могут напрямую использовать общие переменные, а только через специальные функции-методы. Система поддержки выполнения параллельных программ, получив запрос на использование некоторой общей переменной, обрабатывает его, поддерживая при этом консистентное состояние разделяемых данных. Весь контроль осуществляется только программными средствами. В тех случаях, когда для балансировки загрузки процессоров применяется миграция данных, воспользоваться соседством расположения данных в локальной памяти процессора затруднительно.

В таких случаях потери эффективности из-за доступа к данным через функции могут быть вполне приемлемыми, поскольку они могут сполна компенсироваться тем выигрышем, который достигается балансировкой загрузки.

Когерентность данных.

Несколько процессоров могут разделять доступ к одной и той же памяти, если им обеспечена широкая полоса пропускания за счет использования большой многоуровневой кэш-памяти. Поскольку процессоры одновременно работают с данными, хранящимися в единой памяти узла, в подобной архитектуре обязательно должен быть механизм поддержки когерентности данных. Когерентность данных означает, что в любой момент времени для каждого элемента данных во всей памяти узла существует только одно его значение, несмотря на то, что одновременно могут существовать несколько копий элемента данных, расположенных в разных видах памяти и обрабатываемых разными процессорами. Механизм когерентности должен следить за тем, чтобы операции с одним и тем же элементом данных выполнялись на разных процессорах последовательно, удаляя, в частности, устаревшие копии. В современных SMP-архитектурах когерентность реализуется аппаратными средствами.

Известны два основных класса протоколов когерентности кэш-памяти, один из которых строится на основе справочника (directory based). В этом случае информация о состоянии блока физической памяти содержится только в одном месте, называемом справочником (физически справочник может быть распределен по узлам системы). В протоколах наблюдения (snooping) каждая кэш-память, которая содержит копию данных некоторого блока физической памяти, имеет соответствующую копию служебной информации о ее состоянии. Централизованная система записей отсутствует. Обычно кэш-память расположена на общей (разделяемой) шине, и все ее контроллеры наблюдают за шиной (просматривают ее), чтобы определить, не содержат ли они копию соответствующего блока.

В мультипроцессорных системах, где применяются процессоры с кэш-памятью, подсоединенные к централизованной общей памяти, популярны протоколы наблюдения, поскольку для опроса состояния кэш-памяти они могут использовать уже существующее физическое соединение (шину памяти). Кэширование на основе каталога удваивает пропускную способность шины, но требует более сложных аппаратных средств и вызывает дополнительные задержки при пересылке данных между памятью и обеими шинами.

В отличие от классической архитектуры NUMA, при использовании кэш-когерентного доступа к неоднородной памяти - ccNUMA (cache coherent NUMA) все процессоры объединены в один узел (рис. 2), причем первый уровень иерархии памяти образует их кэш-память, а ccNUMA поддерживает когерентность внутри узла аппаратно. В системах ccNUMA распределенная память представляется единым адресным пространством. Не происходит никакого копирования страниц или данных. Нет программной передачи сообщений для синхронизации доступа. Есть просто единый массив памяти (хотя физически он состоит из отдельных частей). Аппаратная когерентность кэш-памяти означает, что не требуется никакого ПО для поддержки актуальности множества копий данных. Все это выполняется на аппаратном уровне так же, как в любом SMP-узле, с одним экземпляром ОС и множеством процессоров. В свою очередь NUMA-Q - это, по сути, реализация архитектуры ccNUMA компанией Sequent.

Элементарным блоком платформы NUMA-Q (рис. 3) служил квод (quad), в котором объединялись четыре процессора, блок разделяемой памяти и шина PCI с семью слотами. Несколько кводов можно было объединить для формирования более крупного одиночного SMP-узла с аппаратно-реализованной кэш-когерентностью. Кэш-когерентное соединение, устанавливаемое между шинами, называлось IQ-Link. Оно было практически полностью прозрачно для программ, подобно обычной кэш-памяти. В традиционном смысле память в каждом кводе не была локальной; скорее, это была одна треть адресного пространства физической памяти, имевшая собственный адресный диапазон. Адресная карта распределялась по памяти равномерно, при этом каждый квод содержал смежную часть адресного пространства. Например, если требуемый адрес находился за пределами диапазона локальной памяти квода, поиск распространялся на кэш-память IQ-Link, которая называлась удаленной. Доступ к ней осуществлялся с такой же скоростью, как и к локальной памяти квода. Если и в кэш-памяти IQ-Link данные не удавалось найти, отсылался запрос на шину IQ-Link. После того как требуемое значение загружалось из другого квода, оно сохранялось в удаленной кэш-памяти IQ-Link запрашивающего квода.

Топологии мультипроцессорных коммутационных сетей.

Типы внутренних связей.

11.9 Системы со слабо и сильно связанными процессорами

Мультипроцессорные системы со слабо связанными процессорами (многомашинные комплексы) предусматривают соединение двух или более независимых вычислительных машин при помощи канала связи (рис. 11.9). При этом каждая машина имеет свою собственную



Рис. 11.9 Мультипроцессорная система со слабо связанными процессорами (многомашинный вычислительный комплекс).

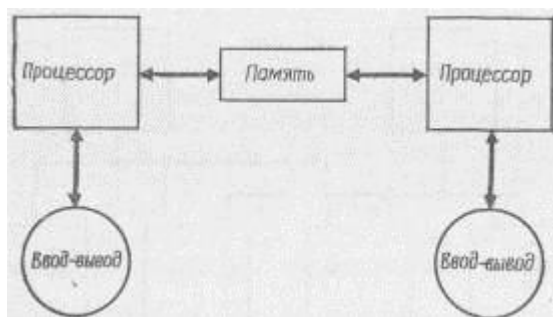


Рис. 11.10 Мультипроцессорная система с сильно связанными процессорами.

операционную систему и память. Машины могут функционировать независимо друг от друга и взаимодействовать при необходимости. Эти самостоятельные машины могут обращаться к файлам друг друга при помощи канала связи, а в некоторых случаях они могут передавать задачи для выполнения на менее загруженных процессорах, что позволяет в определенной степени балансировать нагрузку.

Мультипроцессорные системы с сильно связанными процессорами (многопроцессорные вычислительные комплексы) (рис. 11.10) имеют единую память, коллективно используемую различными процессорами, и единую операционную систему, которая управляет работой всех процессоров и других аппаратных средств системы.

11.10 Организация «главный — подчиненный»

В мультипроцессорной системе с организацией «главный — подчиненный» один из процессоров выполняет функции главного, а другие являются подчиненными. В качестве главного используется процессор общего назначения, который может выполнять как вычисления, так и операции ввода-вывода. Подчиненные процессоры выполняют только вычисления.

Подчиненные процессоры могут эффективно выполнять вычисления, задания же, в которых преобладают операции ввода-вывода, вызывают частое обращение к услугам, которые может предоставлять только главный процессор, что приводит к снижению эффективности работы системы. Оценивая подобную организацию с точки зрения надежности, можно отметить, что в случае выхода из строя одного из подчиненных процессоров система потеряет часть своих вычислительных возможностей, но все же будет продолжать функционировать. Если же выйдет из строя главный процессор, система не сможет выполнять операции ввода-вывода. В некоторых системах эта проблема решается путем переключения устройств ввода-вывода на один из подчиненных процессоров, с тем чтобы при рестарте системы этот процессор выполнял функции главного.

За последние годы скорости работы процессоров удалось повысить гораздо более существенно, чем скорости выполнения операций ввода-вывода. Это привело к тому, что задания, которые ранее считались преимущественно вычислительными, теперь стали требовать относительно больших затрат времени на ввод-вывод, поскольку вычисления, выполняемые между последовательными операциями ввода-вывода, начали производиться намного быстрее. А это в свою очередь привело к тому, что задания, которые ранее могли эффективно выполняться на одной модели мультипроцессорной системы с организацией

«главный — подчиненный», стало теперь менее целесообразно выполнять на новых моделях систем подобной организации с усовершенствованной аппаратурой. Таким образом, основной недостаток мультимикропроцессорных систем с организацией «главный — подчиненный» заключается в асимметричности аппаратных средств. В такой системе процессоры не эквивалентны, поскольку только главный процессор может выполнять как вычисления, так и операции ввода-вывода. Поэтому более удачной можно считать симметричную мультимикропроцессорную систему, в которой все процессоры функционально эквивалентны и могут выполнять и вычисления, и операции ввода-вывода.

Статические и динамические коммутаторы.

С миру по нитке...

Для систем с распределенной памятью обычно используется механизм задач (процессов) и передачи сообщений (обмен данными). Исходная программа разбивается на задачи (подзадачи), которые могут выполняться параллельно на разных процессорах. Связь между задачами осуществляется с помощью передачи сообщений. В основе подхода лежит фундаментальная модель Хоара [12].

Очевидно, что не все программы позволяют эффективно использовать распределенные системы. Так как обмены сообщениями требуют затрат времени для своего выполнения и приостанавливают процессор при ожидании получения сообщения, очень важно, чтобы время, необходимое для обмена сообщениями, не доминировало над временем выполнения программы.

Распределенные системы лучше всего использовать для тех вычислительных задач, которые могут быть разбиты на такие подзадачи, для которых время выполнения существенно больше (долгоживущие процессы), чем время обмена между ними.

Имеются разные подходы как к выбору конкретных средств реализации механизма задач и передачи сообщений, так и отображению этих средств в языке.

Во многих системах (EXPRESS, PVM, P4, PARMACS и др.) механизм задач реализуется с помощью библиотечных вызовов. Накоплен большой опыт использования подобных систем; предпринимаются попытки унификации. Проект стандарта MPI (Message Passing Interface) [13] представляет собой стандартный набор библиотечных интерфейсов для передачи сообщений.

MPI включает большой набор средств, в число которых входят операции передачи сообщений от одного источника (процессора) к другому. Каждому из 6 разных (по способу синхронизации) видов передачи сообщений соответствует своя операция; необходимая для выполнения операции информация задается с помощью параметров процедур. Предусмотрены коллективные операции, включающие как широковещательную передачу сообщений, так и функции редукции.

Поддерживаются все типы данных, имеющиеся в Фортране и Си, имеются и собственные типы данных. Кроме того, для предотвращения коллизии с правилами типов языков верхнего уровня в MPI предусмотрены системные (так называемые "скрытые") объекты, внутреннее представление которых скрыто от пользователя. Предусмотрена возможность конструирования производных типов (структур), которые обеспечивают возможность с помощью одного вызова передать объекты данных разных типов, передать данные, не расположенные в непрерывной области памяти или передать секции массивов.

Для адресации используются группы процессов и коммутаторы. Коммутаторы и контексты обеспечивают возможность делить общее коммуникационное пространство на отдельные замкнутые области.

Хотя MPI предлагает большой набор средств, все же, как уже отмечалось, для прикладных программистов более предпочтительными являются системы, основанные на расширении языков.

Расширения Фортрана для рассматриваемой архитектуры реализованы, например, в языке Fortran Suprenum [14]. В этом языке используется аппарат динамического создания задач и асинхронная модель передачи сообщений. Языковые средства обеспечивают более высокий уровень программирования, так в операторах передачи сообщений можно указывать произвольный список передаваемых данных, аналогичный списку ввода/вывода Фортрана, что гораздо удобнее, чем средства, имеющиеся в системах, основанных на библиотечном подходе, в которых требуется, чтобы либо все передаваемые данные находились в непрерывной области памяти, длина которой часто указывается в байтах, либо (как в MPI) необходимо предварительно объединить эти данные в структуру.

Разработка Fortran Suprenum легла в основу проекта европейского стандарта, который разработан в Германии в рамках общеевропейского проекта ESPRIT [15]. Языковые расширения в данном проекте ориентированы на Фортран 90. Для того, чтобы вводимые расширения были машинно-независимыми, предлагается модель абстрактной машины - GENESIS.

Согласно этой модели, прикладная программа содержит одну начальную задачу и может содержать другие задачи. Несколько задач могут выполняться параллельно на разных процессорах, однако число задач не ограничивается числом процессоров. Каждая задача имеет уникальное имя. Модель предполагает динамическое создание задач; однажды созданная задача размещается в локальной памяти статически. Задача использует только локальную область памяти без деления данных с другими задачами. Выполнение программы начинается с выполнения начальной задачи. Она создает другие задачи, которые, в свою очередь, могут также создавать новые задачи.

Обмен информацией между задачами осуществляется с помощью передачи сообщений. Предусмотрено три способа передачи сообщений: синхронный, асинхронный и передача без ожидания.

В язык введен новый тип данных (имена задач), дополнительный вид программных единиц (программные единицы-задачи), новые операторы, включая операторы передачи сообщений SEND и RECEIVE, расширен набор стандартных процедур.

Группы, контексты, коммутаторы.

Группа - упорядоченное (от 0 до ранга группы) множество идентификаторов процессов (т.е. процессов). Группы служат для указания адресата при посылке сообщений (процесс-адресат специфицируется своим номером в группе), определяют исполнителей коллективных операций. Являются мощным средством функционального распараллеливания - позволяют разделить группу процессов на несколько подгрупп, каждая из которых должна выполнять свою параллельную процедуру. При этом существенно упрощается проблема адресации при использовании параллельных процедур.

Контекст - область “видимости” для сообщений, аналогичное области видимости переменных в случае вложенных вызовов процедур. Сообщения, посланные в некотором контексте, могут быть приняты только в этом же контексте. Контексты - также важные средства поддержки параллельных процедур.

Коммуникаторы - позволяют ограничить область видимости (жизни, определения) сообщений рамками некоторой группы процессов, т.е. могут рассматриваться как пара - группа и контекст. Кроме того, они служат и для целей оптимизации, храня необходимые для этого дополнительные объекты. Имеются предопределенные коммуникаторы (точнее, создаваемые при инициализации MPI-системы):

MPI_COMM_ALL - все процессы

MPI_COMM_PEER - все, кроме главного процесса.

(6) Операции над группами (локальные, без обмена сообщениями).

Для поддержки пользовательских серверов имеется коллективная операция разбиения группы на подгруппы по ключам, которые указывает каждый процесс группы. Для поддержки связывания с серверами, имеются средства построения коммуникатора по некоторому имени, известному и серверу и клиентам.

(7) Точечные коммуникации.

Основные операции - send, receive. Операции могут быть блокирующими и неблокирующими.

Параметры статических коммутационных сетей.

Основные характеристики для анализа топологической структуры

При анализе топологической структуры любой мультипроцессорной системы часто используют ряд характеристик, определяющих количественную меру топологии системы. Целесообразность определения таких характеристик обоснована на необходимости оценки качества системы и ее элементов с позиций общего системного подхода. Ниже приведены некоторые наиболее важные из них.

1. Сложность, определяется как число узлов и связей, составляющих топологию структуры. В случае необходимости выделяются элементы, соответствующие изолированным, висячим и тупиковым вершинам графа. Изолированные вершины неинцидентны ни одному из ребер графа, висячие — соответствуют вершинам, которые нельзя попасть ни из одной другой вершины графа, тупиковые — вершинам, из которых нельзя попасть в другие вершины графа. Наличие в графе изолированных вершин обычно свидетельствует об ошибках, допущенных при формировании или описании структуры, поскольку вся система — всегда целостный объект.

Исследуя граф структуры, следует убедиться в том, что все висячие и тупиковые вершины в нем правильно интерпретированы. Исследование числа и особенностей связей между элементами структуры направлено, прежде всего, на выявление в соответствующем графе контуров и сильно связанных подграфов.

2. Иерархичность — проявляется в структуре связей между элементами. Данная характеристика позволяет распределить элементы топологии в порядке их значимости и дает возможность провести классификацию топологий. Значимость элемента определяется его расположением и связями в графе по отношению к другим элементам, составляющим граф. Для ориентированного графа ранг элемента позволяет высказать такое весьма правдоподобное допущение: чем выше ранг элемента, тем сильнее он влияет на остальные

элементы и тем тяжелее будут последствия его отказа. Возможно узкое понимание ранга как числа связей данного элемента с другими [23].

3. Диаметр структуры — соответствует метрической характеристике, введенной на графе для определения кратчайшего пути между наиболее удаленными вершинами. В ряде случаев разработчики структур оперируют понятием среднего диаметра, имеющим большую системную значимость. По значению диаметра (среднего диаметра) можно косвенно судить о ряде предельных параметров системы, в частности о ее надежности, длительности задержек сообщений, инерционности, количестве разделяющих вершин и ребер.

4. Связность — способность противостоять разбиению, разделению топологии на независимые части. Данная количественная характеристика позволяет выявить наличие обрывов в топологии, «мосты», и т. д. В теории графов существует несколько определений связности, обусловленных различными критериями. Расчет этих характеристик позволяет оценить топологию структуры с разных точек зрения.

5. Плотность — число вершин топологического графа структуры, обладающего заданными диаметром и степенями вершин. На практике эта характеристика применяется только к графам, все узлы которых имеют одинаковую степень.

6. Адаптируемость — приспособленность топологии системы к внешним условиям, позволяющая оптимальным образом достичь цели функционирования.

7. Надежность — способность топологии системы обеспечить функционирование системы в течение заданного промежутка времени. Надежность является одной из важнейших многопараметрических характеристик структуры и определяется как надежностью элементов, так и схемой объединения элементов. Существующие показатели надежности подразделяются на детерминированные и вероятностные.

9. Живучесть — оценивает сохранение частей топологии структуры, обеспечивающих выполнение поставленной задачи. Живучесть имеет тесную функциональную связь с показателями надежности топологии.

10. Параллельность — позволяет обеспечить одновременное функционирование элементов структуры с целью повышения производительности системы.

11. Нарастиваемость — способность топологии некоторой структуры к развитию, достигаемая (без нарушения процесса выполнения основных функций) изменением числа вершин и (или) ребер.

12. Топологическая избыточность — отражает превышение общего числа связей над минимально необходимым. Данная топологическая характеристика используется для косвенной оценки экономичности и надежности исследуемых систем.

13. Стоимость — характеристика, оценивающая стоимость реализации структуры. Наилучшей топологической структурой считается та, которая либо наиболее эффективна при заданной стоимости, либо обладает заданными параметрами при минимальной стоимости. Стоимость топологических элементов предполагается не зависящей от времени, протяженности.

Топологии линейки, решетки, пирамиды.

Линейка (linear array or farm) – представляет собой линейный массив процессоров (рис.1). Система, в которой каждый процессор имеет линии связи только с двумя соседними (с предыдущим и последующим) процессорами; такая схема является, с одной стороны, просто реализуемой, а с другой стороны, соответствует структуре передачи данных при решении многих вычислительных задач (например, при организации конвейерных вычислений);

Шина.

Шинные топологии традиционно используются для организации взаимодействия в вычислительных системах. Одномерная шина применяется во многих SMP-конфигурациях для соединения процессоров, памяти и подсистемы ввода/вывода. Основными ее достоинствами являются простота построения и малая задержка при передаче сообщения (за счет того, что все узлы находятся на расстоянии 1 друг от друга). Основными минусами шин является то, что ее пропускная способность делится между всеми узлами.

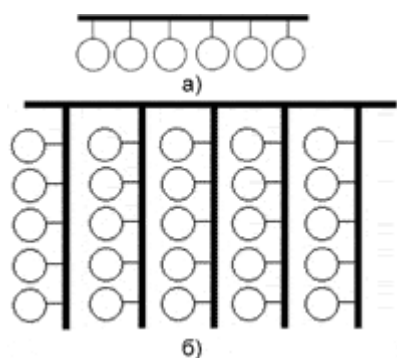


Рис. 15: Шины: а) одномерная; б) иерархическая.

Иерархическая шина является двумерной структурой, в которой множество локальных шин объединены глобальной шиной (или несколькими шинами). Эта схема лучше одномерной шины, если удастся сделать большую часть пересылок локальными. Если же этого добиться невозможно и имеются множественные пересылки между локальными сегментами, то такая организация оказывается даже хуже одномерной шины.

6.2 Полносвязная сеть.

Такая сеть предусматривает соединение узлов по принципу „каждый с каждым“. Такая топология обеспечивает минимальную задержку при обращении к произвольному узлу. К сожалению, число связей равно $[(n-1)n]/2$, где n - количество узлов, то есть наблюдается квадратичный рост числа связей при увеличении числа узлов. Это делает стоимость связей неприемлемо большой, а так как обеспечивать настолько быстрое взаимодействие обычно не надо, то эта схема не используется совсем или используется для систем с числом узлов менее 4.

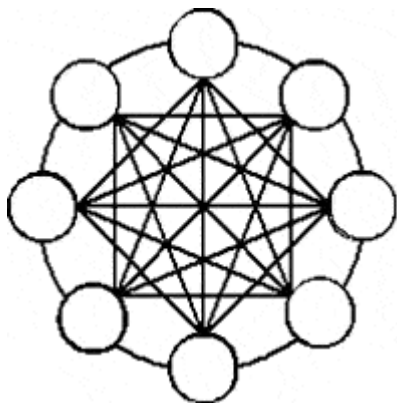


Рис. 16: Полносвязная схема.

6.3 Решетка.

Эта топология представляется наиболее естественной для соединения равноправных узлов. Но при этом максимальное расстояние между узлами будет равно $2\sqrt{n}-2$, что уже для случая с 16-тью узлами даст расстояние 6. Таким образом, подобное объединение однородных узлов не является достаточно компактным.

Тем не менее, подобная организация иногда применяется. Например, так могут быть связаны обрабатывающие элементы векторного процессора, хотя обычно при этом и узлы, и связи получаются неоднородными.

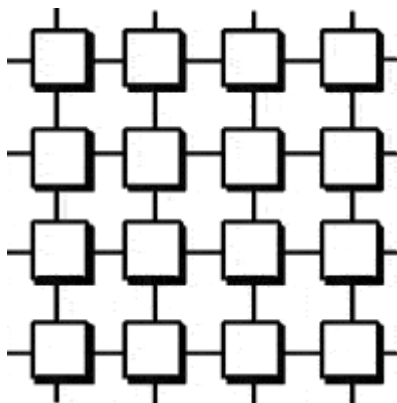


Рис. 17: Решетка.

Иногда решетка образуется двумя наборами шин - горизонтальным и вертикальным. В этом случае каждый процессор имеет два порта. Необходимо обеспечивать схемы маршрутизации для сообщений, которые идут в другие столбец и/или строку.

6.4 Деревья.

Деревья являются довольно популярным способом организации коммуникационной системы из-за своей иерархичности. При этом обеспечивается довольно малый диаметр сети. Наиболее часто применяется бинарное дерево (рис.).

Однако простое дерево имеет узкое место. Им являются два узла - потомки корня. Сам корень должен только пересылать пакеты из левого поддерева в правое, а эти два узла должны осуществлять маршрутизацию для сообщений всего своего поддерева, анализируя адреса получателей данных.

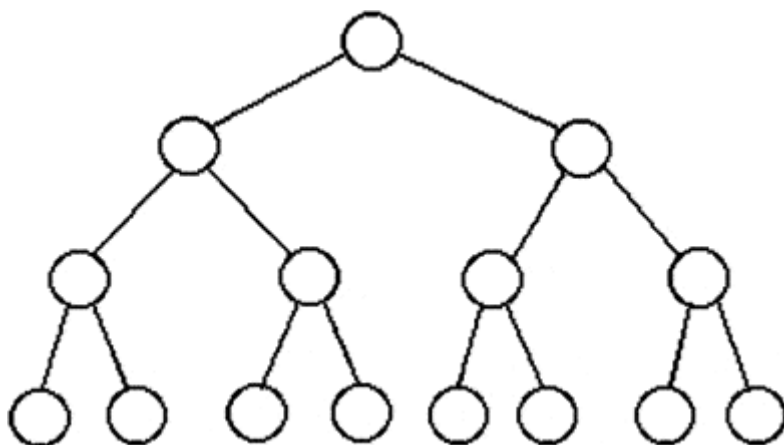


Рис. 18: Бинарное дерево.

Чтобы избавиться от этой проблемы прибегают к схеме Х-дерева (рис. , рис.). В этом дереве вычислительные узлы находятся в листьях дерева, и организованные дополнительные связи, позволяющие листьям организовывать передачу данных внутри поддерева, не затрагивая более высокие уровни дерева.

Возможно, Х-дерево и есть пирамида... (примечание А.Е)

Другой подход к решению той же проблемы предоставляет „толстое дерево“ (рис.). Это прямой способ решения, при котором связи поддеревьев более высокого уровня обладают более высокой производительностью.

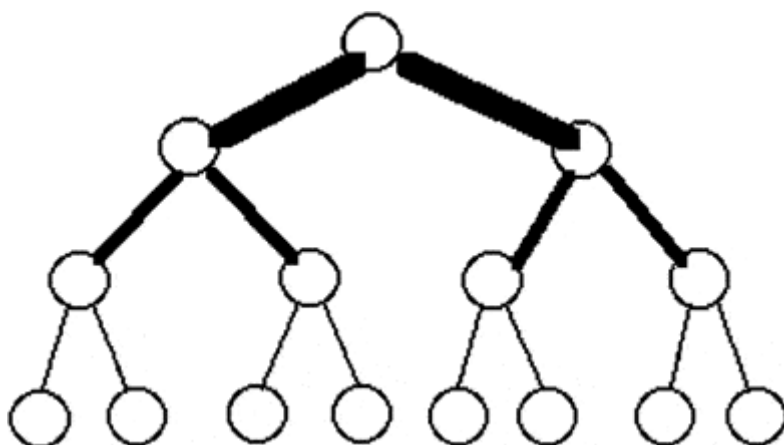


Рис. 21: Толстое дерево (fat-tree).

Топология гиперкуба.

Для получения более компактной конфигурации необходимо решить задачу о нахождении фигуры, имеющей максимальный объем при минимальной площади поверхности. В трехмерном пространстве таким свойством обладает шар. Но поскольку нам необходимо построить узловую систему, вместо шара приходится использовать куб (если число процессоров равно 8) или гиперкуб, если число процессоров больше 8. Размерность гиперкуба будет определяться в зависимости от числа процессоров, которые необходимо соединить. Так, для соединения 16 процессоров потребуется четырехмерный гиперкуб. Для его построения следует взять обычный трехмерный куб, сдвинуть в нужном направлении и, соединив вершины, получить гиперкуб размером 4.

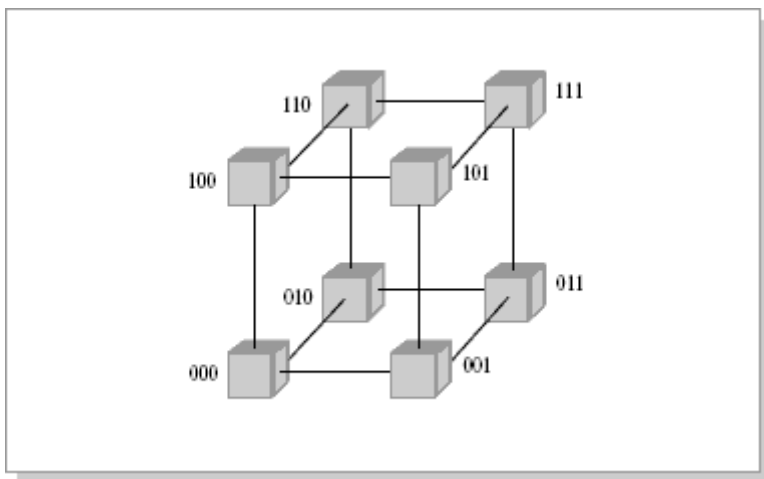


Рис. 4.4. Топология связи, 3-х мерный гиперкуб

Архитектура гиперкуба является второй по эффективности, но самой наглядной. Используются и другие топологии сетей связи: трехмерный тор, "кольцо", "звезда" и другие.

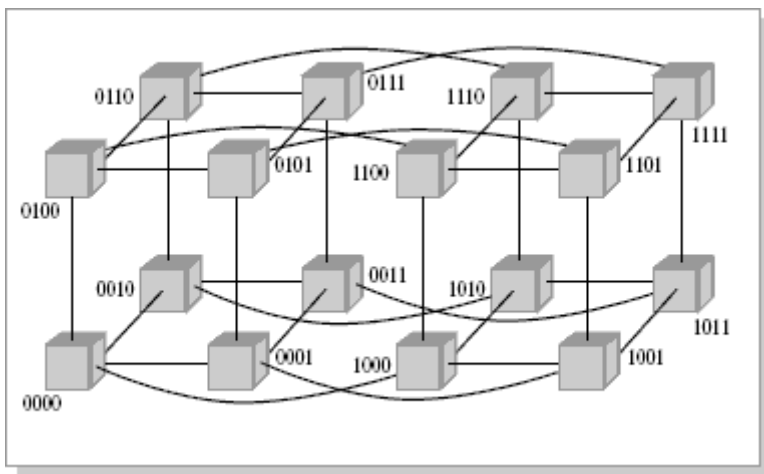


Рис. 4.5. Топология связи, 4-х мерный гиперкуб

Гиперкуб — пространственное обобщение рассмотренной плоской решетки, где адресация процессоров (и соответственно, блоков распределенной памяти) ставится в зависимость от структуры связей между ними. Проводится аналогия с n -мерным пространством.

Возьмем единичный квадрат в двухмерном пространстве с вершиной в начале координат (рис. 9.3,а). Пусть его вершины соответствуют процессорным элементам (ПЭ), а ребра — связям между ними. Пусть код, образованный координатами вершин, — адрес ПЭ. Тогда видно, что связи между ПЭ существуют тогда, когда адреса отличаются не более чем в одном разряде. То же самое можно обнаружить для $n = 3$, $n = 4$ и т.д.

Значит, в общем случае ВС типа "гиперкуб" формируется следующим образом. Ее образуют 2^n ПЭ, каждый ПЭ соединен ровно с n ПЭ. При длине адреса ПЭ, равной n , непосредственно связаны ПЭ, у которых адреса разнятся не более чем в одном разряде.

Структура типа "гиперкуб" обладает важными свойствами. Из структуры "гиперкуб" легко получаются более простые структуры. Например, при $n = 3$ легко получается матричная ВС, на которой, в частности, хорошо решать задачи в конечных разностях. Как мы видели ранее, для этого удобна плоская решетка, где существуют непосредственные (не транзитные) связи между тремя ПЭ (столбец составляют только два ПЭ, поэтому нет необходимости в

связях в двух направлениях), как показано на рис. 9.4. Здесь можно в дополнение к адресам ввести (временно) нумерацию ПЭ и поставить эти номера в соответствие адресам.

Процессоры образовали фрагмент плоской решетки для реализации конечно-разностного метода (метода сеток). Процессоры "прокатываются" по области, на которой строится решение, например, задачи численного интегрирования.

Второе важное свойство "гиперкуба" — возможность выделения задаче необходимых связанных областей вычислительных ресурсов. Эти области соответствуют пространствам меньшей размерности, чем размерность всего "гиперкуба". Это означает, что т.к. процессоры адресуемы внутри некоторого адресного пространства, то каждой решаемой задаче может быть выделен вычислительный ресурс в пределах какого-то массива со сквозной нумерацией процессоров. В n -мерном пространстве этот массив принадлежит подпространству меньшего измерения. Т.е. значительная часть разрядов адреса длины n для всех процессоров, образующих ресурс данной задачи, совпадает, а меняются только несколько последних разрядов.

Например, предположим, что в предыдущем примере мы располагали не 3-мерным, а 6-мерным "гиперкубом". Пусть планирующая система в составе ОС из всего адресного пространства ПЭ $000000 \div 111111$ выделила нам массив $101000 \div 101111$, т.е. определенное 3-хмерное подпространство всего пространства процессоров. В это же время пусть другой задаче выделен массив $000000 \div 001111$, т.е. четырехмерное подпространство и т.д.

Как и для других ОВС, здесь предполагается наличие распределенной памяти. При формировании общего адресного пространства примем, что старшие разряды адреса памяти совпадают с адресом ПЭ.

Согласование сеточных топологий со структурой гиперкуба.

Методы логического представления топологии коммуникационной среды

Как показало рассмотрение основных коммуникационных операций в п. 3.3, ряд алгоритмов передачи данных допускает более простое изложение при использовании вполне определенных топологий сети межпроцессорных соединений. Кроме того, многие методы коммуникации могут быть получены при помощи того или иного логического представления исследуемой топологии. Как результат, важным моментом является при организации параллельных вычислений умение логического представления разнообразных топологий на основе конкретных (физических) межпроцессорных структур.

Способы логического представления (отображения) топологий характеризуются следующими тремя основными характеристиками:

- уплотнение дуг (congestion), выражаемое как максимальное количество дуг логической топологии, отображаемых в одну линию передачи физической топологии;
- удлинение дуг (dilation), определяемое как путь максимальной длины физической топологии, на который отображаемая дуга логической топологии;
- увеличение вершин (expansion), вычисляемое как отношение количества вершин в физической и логической топологиях.

Для рассматриваемых в рамках пособия топологий ограничимся изложением вопросов отображения топологий кольца и решетки на гиперкуб; предлагаемые ниже подходы для логического представления топологий характеризуются единичными показателями уплотнения и удлинения дуг.

Представление кольцевой топологии в виде гиперкуба

Установление соответствия между кольцевой топологией и гиперкубом может быть выполнено при помощи двоичного рефлексивного кода Грея $G(i, N)$ (binary reflected Gray code), Код Грея для $N = 1$ Код Грея для $N = 2$ Код Грея для $N = 3$ Номера процессоров

гиперкуба	кольца			
0	0 0	0 0 0	0	0
1	0 1	0 0 1	1	1
	1 1	0 1 1	3	2
	1 0	0 1 0	2	3
		1 1 0	6	4
		1 1 1	7	5
		1 0 1	5	6
		1 0 0	4	7

Рис. 3.2. Отображение кольцевой топологии на гиперкуб при помощи кода Грея

определяемого в соответствии с выражениями:

$$G(0, 1) = 0, G(1, 1) = 1,$$

где i задает номер значения в коде Грея, а N есть длина этого кода. Для иллюстрации подхода на рис. 3.2 показывается отображение кольцевой топологии на гиперкуб для сети из $p = 8$ процессоров.

Важным свойством кода Грея является тот факт, что соседние значения $G(i, N)$ и $G(i + 1, N)$ имеют только одну различающуюся битовую позицию. Как результат, соседние вершины в кольцевой топологии отображаются на соседние процессоры в гиперкубе .

Отображение топологии решетки на гиперкуб

Отображение топологии решетки на гиперкуб может быть выполнено в рамках подхода, использованного для кольцевой структуры сети. Тогда для отображения решетки $2r \times 2s$ на гиперкуб размерности $N = r + s$ можно принять правило, что элементу решетки с координатами (i, j) , будет соответствовать процессор гиперкуба с номером

$$G(i, r) \parallel G(j, s),$$

где операция $\parallel$ означает конкатенацию кодов Грея

Перекрестный коммутатор.

Полностью лишены недостатков, присущих МПВК с общей шиной, МПВК с перекрестной коммутацией. Идея структурной организации таких ВК заключается в том, что все связи между устройствами осуществляются с помощью специального устройства - коммутационной матрицы. Коммутационная матрица (КМ) позволяет связывать друг с другом любую пару устройств, причем таких пар может быть сколько угодно: связи не зависят друг от друга.

В МПВК с перекрестной коммутацией нет конфликтов из-за связей, остаются только конфликты из-за ресурсов. Возможность одновременной связи нескольких пар устройств позволяет добиваться очень высокой производительности комплекса. Важно отметить и такое обстоятельство, как возможность установления связи между устройствами на любое,

даже на длительное время, так как это совершенно не мешает работе других устройств, зато позволяет передавать любые массивы информации с высокой скоростью, что также способствует повышению производительности комплекса. Заметим, что в МПВК с общей шиной передача информации массивами, т.е, занятие шины одной парой устройств на длительный отрезок времени, обычно допускается лишь в крайних случаях, так как это приводит к длительным простоям остальных устройств.

Кроме того, к достоинствам структуры с перекрестной коммутацией можно отнести простоту и унифицированность интерфейсов всех устройств, а также возможность разрешения всех конфликтов в коммутационной матрице. Важно отметить и то, что нарушение какой-то связи приводит не к выходу из строя всего комплекса, а лишь к отключению какого-либо устройства, т.е. надежность таких комплексов достаточно высока. Однако и организация МПВК с перекрестной коммутацией не свободна от недостатков.

Прежде всего - сложность наращивания ВК. Если в коммутационной матрице заранее не предусмотреть большого числа входов, то введение дополнительных устройств в комплекс потребует установки новой коммутационной матрицы. Существенным недостатком является и то, что коммутационная матрица при большом числе устройств в комплексе становится сложной, громоздкой и достаточно дорогостоящей. (Надо учитывать то обстоятельство, что коммутационные матрицы строятся обычно на схемах, быстродействие которых существенно выше быстродействия схем и элементов основных устройств, - только при этом условии реализуются все преимущества коммутационной матрицы.) Это обстоятельство в значительной степени усложняет и удорожает комплексы.

Для того чтобы упростить и удешевить ВК, коммутацию устройств осуществляют с помощью двух и даже более коммутационных матриц. Перекрестная коммутация довольно широко используется при построении ВК, в частности практически всех МПВК фирмы "Барроуз" (в том числе и упомянутого выше комплекса D-825).

В МПВК с многовходовыми ОЗУ все, что связано с коммутацией устройств, осуществляется в ОЗУ. В этом случае модули ОЗУ имеют число входов, равное числу устройств, которые к ним подключаются, т. е. для каждого устройства предусматривается свой вход в ОЗУ. В отличие от ВК с перекрестной коммутацией, которые имеют централизованное коммутационное устройство, в МПВК с многовходовыми ОЗУ средства коммутации распределены между несколькими устройствами. Такой способ организации МПВК сохраняет все преимущества систем с перекрестной коммутацией, несколько упрощая при этом саму систему коммутации. Для наращивания системы должны быть предусмотрены дополнительные входы в ОЗУ. Правда, введение дополнительных модулей ОЗУ не вызывает затруднений.

Мультипроцессорные системы, построенные по принципу осуществления связей между модулями посредством "прямоугольной решетки" соединительных шин, которые могут контактировать в любой точке их пересечения, называют системами с перекрестной коммутацией (рис 2.1).

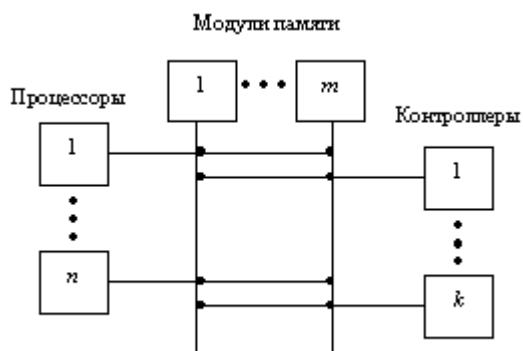


Рис. 7.2.1

Такая организация системы позволяет устанавливать контакт между любыми двумя блоками системы на все время обмена информацией. В отличие от коммутации с временным разделением, реализуемым в системах с общей шиной, рассматриваемый метод переключения связей часто называют коммутацией с пространственным разделением.

Перекрестный коммутатор является "неблокирующимся" в том смысле, что передача через него может быть запрещена из-за отсутствия путей передачи. Существует возможность установить одновременно несколько путей передачи информации в системе. В то же время следует иметь в виду, что коммутатор может быть заблокирован, если одно из соединяемых устройств уже занято.

Одной из ранних структур, в которой реализован принцип перекрестной коммутации, явилась система, получившая название "полиморфная ЭВМ" (рис. 2.2). Модули ЭВМ, включающие блоки процессоров и памяти, могли осуществлять связь с периферийными устройствами через центральный коммутатор.



Рис. 7.2.2

В данной системе была сделана попытка организовать соединения непосредственно между процессорами и перекрестный доступ к памяти путем замыкания соответствующего набора пересечений. Сложность такого способа связи между процессорами и блоками памяти, неэффективность использования оборудования (процессор и память одного единственного модуля, имея единственную шину связи, "мешают" друг другу) выявляют недостатки структуры "полиморфной ЭВМ" по сравнению со структурой системы, приведенной на рис. 2.1

Мультипроцессорные системы с перекрестной коммутацией, обладая несколько меньшей гибкостью, чем системы с общей шиной, позволяют тем не менее сравнительно просто вводить новые модули, если коммутационная матрица обладает достаточной емкостью. Матрица полностью отделена от других функциональных блоков и может быть построена также по модульному принципу, что допускает ее расширение. Однако вследствие сложности функций коммутатора, структура его может существенно усложниться.

Для обеспечения большей гибкости и увеличения возможностей по расширению в системе может быть введена дополнительная коммутационная матрица устройств ввода-вывода. Такой коммутатор связывается с центральным через процессоры управления вводом-выводом (рис. 2.3), при этом устройства ввода-вывода могут подсоединяться к любому каналу. Рассмотренная структура мультипроцессорных систем используется в больших вычислительных системах фирмы "Burroughs" (США).

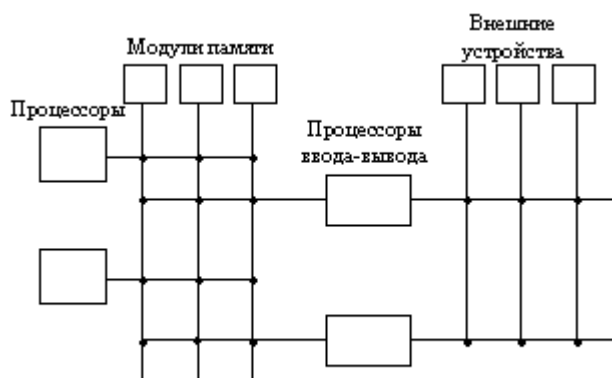


Рис. 2.3

Оригинальный вариант организации мультипроцессорной конфигурации предложен для системы Multi-Interpreter фирмы "Burroughs", в которую введена группа однотипных процессорных блоков с микропрограммным управлением. Путем перезагрузки блоков микропрограммной памяти одни и те же модули используются в качестве центральных процессоров либо контроллеров ввода-вывода. Благодаря этому все процессоры, модули памяти и периферийные устройства подключены к общей коммутационной матрице (рис. 2.4).



Рис. 2.4

Мультипроцессорными системами с перекрестной коммутацией, кроме уже упомянутых зарубежных ЭВМ, являются отечественная вычислительная система высокой производительности "Эльбрус-1" и вычислительный комплекс СМ-2-одна из моделей СМ ЭВМ.

Многокаскадные коммутационные сети.

Составные коммутаторы

Простые коммутаторы имеют ограничения на число входов и выходов, а также могут требовать большого количества оборудования при увеличении этого числа (в случае пространственных коммутаторов). Поэтому для построения коммутаторов с большим

количеством входов и выходов используют совокупность простых коммутаторов, объединенных с помощью линий "точка-точка".

Составные коммутаторы имеют задержку, пропорциональную количеству простых коммутаторов, через которые проходит сигнал от входа до выхода, т.е. числу каскадов. Однако объем оборудования составного коммутатора меньше, чем простого с тем же количеством входов и выходов.

Чаще всего составные коммутаторы строятся из прямоугольных коммутаторов 2×2 с двумя входами и выходами. Они имеют два состояния: прямое пропускание входов на соответствующие выходы и перекрестное пропускание. Коммутатор 2×2 состоит из собственно блока коммутации данных и блока управления. Блок управления в зависимости от поступающих на него управляющих сигналов определяет, какой тип соединения следует осуществить в блоке коммутации - прямой или перекрестный. При этом если оба входа хотят соединиться с одним выходом, то коммутатор разрешает конфликт и связывает с данным выходом только один вход, а запрос на соединение со стороны второго блокируется или отвергается.

Коммутатор Клоза

Коммутатор Клоза может быть построен в качестве альтернативы для прямоугольного коммутатора с $(m \times d)$ входами и $(m \times d)$ выходами. Он формируется из трех каскадов коммутаторов: m коммутаторов $(d \times d)$ во входном каскаде, m коммутаторов $(d \times d)$ в выходном и d промежуточных коммутаторов $(m \times m)$.

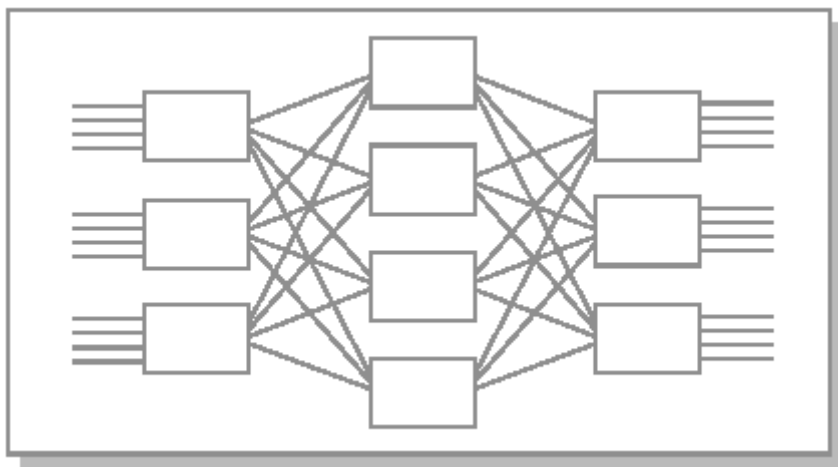


Рис. 10.1. Коммутатор Клоза 3×4

Соединения внутри коммутатора устроены следующим образом:

j -й выход i -ого коммутатора входного каскада соединен с i -ым входом j -ого промежуточного коммутатора;

j -й вход k -ого коммутатора выходного каскада соединен с k -ым выходом j -ого промежуточного коммутатора.

Данный тип составных коммутаторов позволяет соединять любой вход с любым выходом, однако при установленных соединениях добавление нового соединения может потребовать разрыва и переустановки всех соединений.

Баньян-сети

Коммутаторы этого типа строятся на базе прямоугольных коммутаторов таким образом, что существует только один путь от каждого входа к каждому выходу.

Наиболее важной разновидностью баньян-сетей является дельта-сеть. Она формируется из прямоугольных коммутаторов ($a \times b$) и представляет собой n -каскадный коммутатор с an входами и bn выходами. Составляющие коммутаторы соединены так, что для соединения любого входа и выхода образуется единственный путь одинаковой для всех пар входов и выходов длины.

Распределенные составные коммутаторы

В распределенных вычислительных системах ресурсы разделяются между задачами, каждая из которых выполняется на своем подмножестве процессоров. В связи с этим возникает понятие близости процессоров, которая является важной для активно взаимодействующих процессоров. Обычно близость процессоров выражается в различной каскадности соединений, различных расстояниях между ними.

Один из вариантов создания составных коммутаторов заключается в объединении прямоугольных коммутаторов $(v+1 \times v+1)$, $v > 1$ таким образом, что один вход и один выход каждого составляющего коммутатора служат входом и выходом составного коммутатора. К каждому внутреннему коммутатору подсоединяются процессор и память, образуя вычислительный модуль с v -каналами для соединения с другими вычислительными модулями. Свободные v -входов и v -выходов каждого вычислительного модуля соединяются линиями "точка-точка" с входами и выходами других коммутаторов, образуя граф межмодульных связей.

Наиболее эффективным графом межмодульных связей с точки зрения организации обмена данными между вычислительными модулями является полный граф. В этом случае между каждой парой вычислительных модулей существует прямое соединение. При этом возможны одновременные соединения между произвольными вычислительными модулями.

Однако обычно создать полный граф межмодульных связей невозможно по ряду причин. Обмен данными приходится производить через цепочки транзитных модулей. Из-за этого увеличиваются задержки, и ограничивается возможность установления одновременных соединений. Таким образом, эффективный граф межмодульных связей должен минимизировать время межмодульных обменов и максимально увеличить количество одновременно активизированных соединений. Кроме того, на выбор графа межмодульных связей влияет учет отказов и восстановлений вычислительных модулей и линий связи.

Производительность вычислительных систем.

Пиковая производительность.

Главной отличительной особенностью многопроцессорной вычислительной системы является ее производительность, т.е. количество операций, производимых системой за единицу времени. Различают пиковую и реальную производительность. Под пиковой понимают величину, равную произведению пиковой производительности одного процессора на число таких процессоров в данной машине. При этом предполагается, что все устройства компьютера работают в максимально производительном режиме. Пиковая производительность компьютера вычисляется однозначно, и эта характеристика является базовой, по которой производят сравнение высокопроизводительных вычислительных систем. Чем больше пиковая производительность, тем (теоретически) быстрее пользователь сможет решить свою задачу. Пиковая производительность есть величина теоретическая и, вообще говоря, недостижимая при запуске конкретного приложения. Реальная же производительность, достигаемая на данном приложении, зависит от взаимодействия

программной модели, в которой реализовано приложение, с архитектурными особенностями машины, на которой приложение запускается.

Существует два способа оценки пиковой производительности компьютера. Один из них опирается на число команд, выполняемых компьютером за единицу времени. Единицей измерения, как правило, является MIPS (Million Instructions Per Second). Производительность, выраженная в MIPS, говорит о скорости выполнения компьютером своих же инструкций. Но, во-первых, заранее не ясно, в какое количество инструкций отобразится конкретная программа, а во-вторых, каждая программа обладает своей спецификой, и число команд от программы к программе может меняться очень сильно. В связи с этим данная характеристика дает лишь самое общее представление о производительности компьютера.

Другой способ измерения производительности заключается в определении числа вещественных операций, выполняемых компьютером за единицу времени. Единицей измерения является FLOPS (Floating point operations per second) – число операций с плавающей точкой, производимых компьютером за одну секунду. Такой способ является более приемлемым для пользователя, поскольку ему известна вычислительная сложность программы, и, пользуясь этой характеристикой, пользователь может получить нижнюю оценку времени ее выполнения.

Однако пиковая производительность получается только в идеальных условиях, т.е. при отсутствии конфликтов при обращении к памяти при равномерной загрузке всех устройств. В реальных условиях на выполнение конкретной программы влияют такие аппаратно-программные особенности данного компьютера как: особенности структуры процессора, системы команд, состав функциональных устройств, реализация ввода/вывода, эффективность работы компиляторов.

Методы оценки производительности.

Основу для сравнения различных типов компьютеров между собой дают стандартные методики измерения производительности. В процессе развития вычислительной техники появилось несколько таких стандартных методик. Они позволяют разработчикам и пользователям осуществлять выбор между альтернативами на основе количественных показателей, что дает возможность постоянного прогресса в данной области.

Единицей измерения производительности компьютера является время: компьютер, выполняющий тот же объем работы за меньшее время является более быстрым. Время выполнения любой программы измеряется в секундах. Часто производительность измеряется как скорость появления некоторого числа событий в секунду, так что меньшее время подразумевает большую производительность.

Однако в зависимости от того, что мы считаем, время может быть определено различными способами. Наиболее простой способ определения времени называется астрономическим временем, временем ответа (response time), временем выполнения (execution time) или прошедшим временем (elapsed time). Это задержка выполнения задания, включающая буквально все: работу процессора, обращения к диску, обращения к памяти, ввод/вывод и накладные расходы операционной системы. Однако при работе в мультипрограммном режиме во время ожидания ввода/вывода для одной программы, процессор может выполнять другую программу, и система не обязательно будет минимизировать время выполнения данной конкретной программы.

Для измерения времени работы процессора на данной программе используется специальный параметр - время ЦП (CPU time), которое не включает время ожидания ввода/вывода или время выполнения другой программы. Очевидно, что время ответа, видимое пользователем, является полным временем выполнения программы, а не временем ЦП. Время ЦП может далее делиться на время, потраченное ЦП непосредственно на выполнение программы пользователя и называемое пользовательским временем ЦП, и время ЦП, затраченное операционной системой на выполнение заданий, затребованных программой, и называемое системным временем ЦП.

В ряде случаев системное время ЦП игнорируется из-за возможной неточности измерений, выполняемых самой операционной системой, а также из-за проблем, связанных со сравнением производительности машин с разными операционными системами. С другой стороны, системный код на некоторых машинах является пользовательским кодом на других и, кроме того, практически никакая программа не может работать без некоторой операционной системы. Поэтому при измерениях производительности процессора часто используется сумма пользовательского и системного времени ЦП.

В большинстве современных процессоров скорость протекания процессов взаимодействия внутренних функциональных устройств определяется не естественными задержками в этих устройствах, а задается единой системой синхросигналов, вырабатываемых некоторым генератором тактовых импульсов, как правило, работающим с постоянной скоростью. Дискретные временные события называются тактами синхронизации (clock ticks), просто тактами (ticks), периодами синхронизации (clock periods), циклами (cycles) или циклами синхронизации (clock cycles). Разработчики компьютеров обычно говорят о периоде синхронизации, который определяется либо своей длительностью (например, 10 наносекунд), либо частотой (например, 100 МГц). Длительность периода синхронизации есть величина, обратная к частоте синхронизации.

Таким образом, время ЦП для некоторой программы может быть выражено двумя способами: количеством тактов синхронизации для данной программы, умноженным на длительность такта синхронизации, либо количеством тактов синхронизации для данной программы, деленным на частоту синхронизации.

Важной характеристикой, часто публикуемой в отчетах по процессорам, является среднее количество тактов синхронизации на одну команду - CPI (clock cycles per instruction). При известном количестве выполняемых команд в программе этот параметр позволяет быстро оценить время ЦП для данной программы.

Таким образом, производительность ЦП зависит от трех параметров: такта (или частоты) синхронизации, среднего количества тактов на команду и количества выполняемых команд. Невозможно изменить ни один из указанных параметров изолированно от другого, поскольку базовые технологии, используемые для изменения каждого из этих параметров, взаимосвязаны: частота синхронизации определяется технологией аппаратных средств и функциональной организацией процессора; среднее количество тактов на команду зависит от функциональной организации и архитектуры системы команд; а количество выполняемых в программе команд определяется архитектурой системы команд и технологией компиляторов. Когда сравниваются две машины, необходимо рассматривать все три компонента, чтобы понять относительную производительность.

В процессе поиска стандартной единицы измерения производительности компьютеров было принято несколько популярных единиц измерения, вследствие чего несколько безвредных терминов были искусственно вырваны из их хорошо определенного контекста и

использованы там, для чего они никогда не предназначались. В действительности единственной подходящей и надежной единицей измерения производительности является время выполнения реальных программ, и все предлагаемые замены этого времени в качестве единицы измерения или замены реальных программ в качестве объектов измерения на синтетические программы только вводят в заблуждение.

Опасности некоторых популярных альтернативных единиц измерения (MIPS и MFLOPS) будут рассмотрены в соответствующих подразделах.

MIPS

Одной из альтернативных единиц измерения производительности процессора (по отношению к времени выполнения) является MIPS - (миллион команд в секунду). Имеется несколько различных вариантов интерпретации определения MIPS.

В общем случае MIPS есть скорость операций в единицу времени, т.е. для любой данной программы MIPS есть просто отношение количества команд в программе к времени ее выполнения. Таким образом, производительность может быть определена как обратная к времени выполнения величина, причем более быстрые машины при этом будут иметь более высокий рейтинг MIPS.

Положительными сторонами MIPS является то, что эту характеристику легко понять, особенно покупателю, и что более быстрая машина характеризуется большим числом MIPS, что соответствует нашим интуитивным представлениям. Однако использование MIPS в качестве метрики для сравнения наталкивается на три проблемы. Во-первых, MIPS зависит от набора команд процессора, что затрудняет сравнение по MIPS компьютеров, имеющих разные системы команд. Во-вторых, MIPS даже на одном и том же компьютере меняется от программы к программе. В-третьих, MIPS может меняться по отношению к производительности в противоположенную сторону.

Классическим примером для последнего случая является рейтинг MIPS для машины, в состав которой входит сопроцессор плавающей точки. Поскольку в общем случае на каждую команду с плавающей точкой требуется большее количество тактов синхронизации, чем на целочисленную команду, то программы, используя сопроцессор плавающей точки вместо соответствующих подпрограмм из состава программного обеспечения, выполняются за меньшее время, но имеют меньший рейтинг MIPS. При отсутствии сопроцессора операции над числами с плавающей точкой реализуются с помощью подпрограмм, использующих более простые команды целочисленной арифметики и, как следствие, такие машины имеют более высокий рейтинг MIPS, но выполняют настолько большее количество команд, что общее время выполнения значительно увеличивается. Подобные аномалии наблюдаются и при использовании оптимизирующих компиляторов, когда в результате оптимизации сокращается количество выполняемых в программе команд, рейтинг MIPS уменьшается, а производительность увеличивается.

Другое определение MIPS связано с очень популярным когда-то компьютером VAX 11/780 компании DEC. Именно этот компьютер был принят в качестве эталона для сравнения производительности различных машин. Считалось, что производительность VAX 11/780 равна 1 MIPS (одному миллиону команд в секунду).

В то время широкое распространение получил синтетический тест Dhrystone, который позволял оценивать эффективность процессоров и компиляторов с языка C для программ

нечисловой обработки. Он представлял собой тестовую смесь, 53% которой составляли операторы присваивания, 32% - операторы управления и 15% - вызовы функций. Это был очень короткий тест: общее число команд равнялось 100. Скорость выполнения программы из этих 100 команд измерялась в Dhrystone в секунду. Быстродействие VAX 11/780 на этом синтетическом тесте составляло 1757Dhrystone в секунду. Таким образом 1MIPS равен 1757 Dhrystone в секунду.

Следует отметить, что в настоящее время тест Dhrystone практически не применяется. Малый объем позволяет разместить все команды теста в кэш-памяти первого уровня современного микропроцессора и он не позволяет даже оценить эффект наличия кэш-памяти второго уровня, хотя может хорошо отражать эффект увеличения тактовой частоты.

Третье определение MIPS связано с IBM RS/6000 MIPS. Дело в том, что ряд производителей и пользователей (последователей фирмы IBM) предпочитают сравнивать производительность своих компьютеров с производительностью современных компьютеров IBM, а не со старой машиной компании DEC. Соотношение между VAX MIPS и RS/6000 MIPS никогда широко не публиковались, но 1 RS/6000 MIPS примерно равен 1.6 VAX 11/780 MIPS.

MFLOPS

Измерение производительности компьютеров при решении научно-технических задач, в которых существенно используется арифметика с плавающей точкой, всегда вызывало особый интерес. Именно для таких вычислений впервые встал вопрос об измерении производительности, а по достигнутым показателям часто делались выводы об общем уровне разработок компьютеров. Обычно для научно-технических задач производительность процессора оценивается в MFLOPS (миллионах чисел-результатов вычислений с плавающей точкой в секунду, или миллионах элементарных арифметических операций над числами с плавающей точкой, выполненных в секунду).

Как единица измерения, MFLOPS, предназначена для оценки производительности только операций с плавающей точкой, и поэтому не применима вне этой ограниченной области. Например, программы компиляторов имеют рейтинг MFLOPS близкий к нулю вне зависимости от того, насколько быстра машина, поскольку компиляторы редко используют арифметику с плавающей точкой.

Ясно, что рейтинг MFLOPS зависит от машины и от программы. Этот термин менее безобидный, чем MIPS. Он базируется на количестве выполняемых операций, а не на количестве выполняемых команд. По мнению многих программистов, одна и та же программа, работающая на различных компьютерах, будет выполнять различное количество команд, но одно и то же количество операций с плавающей точкой. Именно поэтому рейтинг MFLOPS предназначался для справедливого сравнения различных машин между собой.

Однако и с MFLOPS не все обстоит так безоблачно. Прежде всего, это связано с тем, что наборы операций с плавающей точкой не совместимы на различных компьютерах. Например, в суперкомпьютерах фирмы Cray Research отсутствует команда деления (имеется, правда, операция вычисления обратной величины числа с плавающей точкой, а операция деления может быть реализована с помощью умножения делимого на обратную величину делителя). В то же время многие современные микропроцессоры имеют команды деления, вычисления квадратного корня, синуса и косинуса.

Другая, осознаваемая всеми, проблема заключается в том, что рейтинг MFLOPS меняется не только на смеси целочисленных операций и операций с плавающей точкой, но и на смеси быстрых и медленных операций с плавающей точкой. Например, программа со 100% операций сложения будет иметь более высокий рейтинг, чем программа со 100% операций деления.

Решение обеих проблем заключается в том, чтобы взять "каноническое" или "нормализованное" число операций с плавающей точкой из исходного текста программы и затем поделить его на время выполнения. На рисунке 2.1 показано, каким образом авторы тестового пакета "Ливерморские циклы", о котором речь пойдет ниже, вычисляют для программы количество нормализованных операций с плавающей точкой в соответствии с операциями, действительно находящимися в ее исходном тексте. Таким образом, рейтинг реальных MFLOPS отличается от рейтинга нормализованных MFLOPS, который часто приводится в литературе по суперкомпьютерам. Реальные операции с ПТ

Нормализованные операции с ПТ

Сложение, вычитание, сравнение, умножение 1

Деление, квадратный корень 4

Экспонента, синус, ... 8

Рис. 3.1. Соотношение между реальными и нормализованными операциями с плавающей точкой, которым пользуются авторы "ливерморских циклов" для вычисления рейтинга MFLOPS

Наиболее часто MFLOPS, как единица измерения производительности, используется при проведении контрольных испытаний на тестовых пакетах "Ливерморские циклы" и LINPACK.

Ливерморские циклы - это набор фрагментов фортран-программ, каждый из которых взят из реальных программных систем, эксплуатируемых в Ливерморской национальной лаборатории им.Лоуренса (США). Обычно при проведении испытаний используется либо малый набор из 14 циклов, либо большой набор из 24 циклов.

Пакет Ливерморских циклов используется для оценки производительности вычислительных машин с середины 60-х годов. Ливерморские циклы считаются типичными фрагментами программ численных задач. Появление новых типов машин, в том числе векторных и параллельных, не уменьшило важности Ливерморских циклов, однако изменились значения производительности и величины разброса между разными циклами.

На векторной машине производительность зависит не только от элементной базы, но и от характера самого алгоритма, т.е. коэффициента векторизуемости. Среди Ливерморских циклов коэффициент векторизуемости колеблется от 0 до 100%, что еще раз подтверждает их ценность для оценки производительности векторных архитектур. Кроме характера алгоритма, на коэффициент векторизуемости влияет и качество векторизатора, встроенного в компилятор.

На параллельной машине производительность существенно зависит от соответствия между структурой аппаратных связей вычислительных элементов и структурой вычислений в алгоритме. Важно, чтобы тестовый пакет представлял алгоритмы различных структур. В Ливерморских циклах встречаются последовательные, сеточные, конвейерные, волновые вычислительные алгоритмы, что подтверждает их пригодность и для параллельных машин. Однако обобщение результатов измерения производительности, полученных для одной параллельной машины, на другие параллельные машины или хотя бы некоторый подкласс

параллельных машин, может дать неверный результат, ибо структуры аппаратных связей в таких машинах гораздо более разнообразны, чем, скажем, в векторных машинах.

LINPACK - это пакет фортран-программ для решения систем линейных алгебраических уравнений. Целью создания LINPACK отнюдь не было измерение производительности. Алгоритмы линейной алгебры весьма широко используются в самых разных задачах, и поэтому измерение производительности на LINPACK представляют интерес для многих пользователей. Сведения о производительности различных машин на пакете LINPACK публикуются сотрудником Аргоннской национальной лаборатории (США) Дж. Донгаррой и периодически обновляются.

В основе алгоритмов действующего варианта LINPACK лежит метод декомпозиции. Исходная матрица размером 100×100 элементов (в последнем варианте размером 1000×1000) сначала представляется в виде произведения двух матриц стандартной структуры, над которыми затем выполняется собственно алгоритм нахождения решения. Подпрограммы, входящие в LINPACK, структурированы. В стандартном варианте LINPACK выделен внутренний уровень базовых подпрограмм, каждая из которых выполняет элементарную операцию над векторами. Набор базовых подпрограмм называется BLAS (Basic Linear Algebra Subprograms). Например, в BLAS входят две простые подпрограммы SAXPY (умножение вектора на скаляр и сложение векторов) и SDOT (скалярное произведение векторов). Все операции выполняются над числами с плавающей точкой, представленными с двойной точностью. Результат измеряется в MFLOPS.

Использование результатов работы тестового пакета LINPACK с двойной точностью как основы для демонстрации рейтинга MFLOPS стало общепринятой практикой в компьютерной промышленности. При этом следует помнить, что при использовании исходной матрицы размером 100×100 , она полностью может размещаться в кэш-памяти емкостью, например, 1 Мбайт. Если при проведении испытаний используется матрица размером 1000×1000 , то емкости такого кэша уже недостаточно и некоторые обращения к памяти будут ускоряться благодаря наличию такого кэша, другие же будут приводить к промахам и потребуют большего времени на обработку обращений к памяти. Для многопроцессорных систем также имеются параллельные версии LINPACK и такие системы часто показывают линейное увеличение производительности с ростом числа процессоров.

Однако, как и любая другая единица измерения, рейтинг MFLOPS для отдельной программы не может быть обобщен на все случаи жизни, чтобы представлять единственную единицу измерения производительности компьютера, хотя очень соблазнительно характеризовать машину единственным рейтингом MIPS или MFLOPS без указания программы.

Тесты SPEC

Важность создания пакетов тестов, базирующихся на реальных прикладных программах широкого круга пользователей и обеспечивающих эффективную оценку производительности процессоров, была осознана большинством крупнейших производителей компьютерного оборудования, которые в 1988 году учредили бесприбыльную корпорацию SPEC (Standard Performance Evaluation Corporation). Основной целью этой организации является разработка и поддержка стандартизованного набора специально подобранных тестовых программ для оценки производительности новейших

поколений высокопроизводительных компьютеров. Членом SPEC может стать любая организация, уплатившая вступительный взнос.

Главными видами деятельности SPEC являются:

Разработка и публикация наборов тестов, предназначенных для измерения производительности компьютеров. Перед публикацией объектные коды этих наборов вместе с исходными текстами и инструментальными средствами интенсивно проверяются на предмет возможности импортирования на разные платформы. Они доступны для широкого круга пользователей за плату, покрывающую расходы на разработку и административные издержки. Специальное лицензионное соглашение регулирует вопросы выполнения тестирования и публикации результатов в соответствии с документацией на каждый тестовый набор.

SPEC публикует ежеквартальный отчет о новостях SPEC и результатах тестирования: "The SPEC Newsletter", что обеспечивает централизованный источник информации для результатов тестирования на тестах SPEC.

Основным результатом работы SPEC являются наборы тестов. Эти наборы разрабатываются SPEC с использованием кодов, поступающих из разных источников. SPEC работает над импортированием этих кодов на разные платформы, а также создает инструментальные средства для формирования из кодов, выбранных в качестве тестов, осмысленных рабочих нагрузок. Поэтому тесты SPEC отличаются от свободно распространяемых программ. Хотя они могут существовать под похожими или теми же самыми именами, время их выполнения в общем случае будет отличаться.

В настоящее время имеется два базовых набора тестов SPEC, ориентированных на интенсивные расчеты и измеряющих производительность процессора, системы памяти, а также эффективность генерации кода компилятором. Как правило, эти тесты ориентированы на операционную систему UNIX, но они также импортированы и на другие платформы. Процент времени, расходуемого на работу операционной системы и функции ввода/вывода, в общем случае ничтожно мал.

Набор тестов CINT92, измеряющий производительность процессора при обработке целых чисел, состоит из шести программ, написанных на языке Си и выбранных из различных прикладных областей: теория цепей, интерпретатор языка Лисп, разработка логических схем, упаковка текстовых файлов, электронные таблицы и компиляция программ.

Набор тестов CFP92, измеряющий производительность процессора при обработке чисел с плавающей точкой, состоит из 14 программ, также выбранных из различных прикладных областей: разработка аналоговых схем, моделирование методом Монте-Карло, квантовая химия, оптика, робототехника, квантовая физика, астрофизика, прогноз погоды и другие научные и инженерные задачи. Две программы из этого набора написаны на языке Си, а остальные 12 - на Фортране. В пяти программах используется одинарная, а в остальных - двойная точность.

Результаты прогона каждого индивидуального теста из этих двух наборов выражаются отношением времени выполнения одной копии теста на тестируемой машине к времени ее выполнения на эталонной машине. В качестве эталонной машины используется VAX 11/780. SPEC публикует результаты прогона каждого отдельного теста, а также две составные оценки: SPECint92 - среднее геометрическое 6 результатов индивидуальных тестов из набора CINT92 и SPECfp92 - среднее геометрическое 14 результатов индивидуальных тестов из набора CFP92.

Следует отметить, что результаты тестирования на наборах CINT92 и CFT92 сильно зависят от качества применяемых оптимизирующих компиляторов. Для более точного выяснения возможностей аппаратных средств с середины 1994 года SPEC ввел две дополнительные составные оценки: SPECbase_int92 и SPECbase_fp92, которые накладывает определенные ограничения на используемые компиляторы поставщиками компьютеров при проведении испытаний.

Составные оценки SPECint92 и SPECfp92 достаточно хорошо характеризуют производительность процессора и системы памяти при работе в однозадачном режиме, но они совершенно не подходят для оценки производительности многопроцессорных и однопроцессорных систем, работающих в многозадачном режиме. Для этого нужна оценка пропускной способности системы или ее емкости, показывающая количество заданий, которое система может выполнить в течение заданного интервала времени. Пропускная способность системы определяется прежде всего количеством ресурсов (числом процессоров, емкостью оперативной и кэш-памяти, пропускной способностью шины), которые система может предоставить в распоряжение пользователя в каждый момент времени. Именно такую оценку, названную SPECrate и заменившую ранее применявшуюся оценку SPECthruput89, SPEC предложила в качестве единицы измерения производительности многопроцессорных систем.

При этом для измерения выбран метод "однородной нагрузки" (homogenous capacity method), заключающийся в том, что одновременно выполняются несколько копий одной и той же тестовой программы. Результаты этих тестов показывают, как много задач конкретного типа могут быть выполнены в указанное время, а их средние геометрические значения (SPECrate_int92 - на наборе тестов, измеряющих производительность целочисленных операций и SPECrate_fp92 - на наборе тестов, измеряющих производительность на операциях с плавающей точкой) наглядно отражают пропускную способность однопроцессорных и многопроцессорных конфигураций при работе в многозадачном режиме в системах коллективного пользования. В качестве тестовых программ для проведения испытаний на пропускную способность выбраны те же наборы CINT92 и CFT92.

При прогоне тестового пакета делаются независимые измерения по каждому отдельному тесту. Обычно такой параметр, как количество запускаемых копий каждого отдельного теста, выбирается исходя из соображений оптимального использования ресурсов, что зависит от архитектурных особенностей конкретной системы. Одной из очевидных возможностей является установка этого параметра равным количеству процессоров в системе. При этом все копии отдельной тестовой программы запускаются одновременно, и фиксируется время завершения последней из всех запущенных программ.

С середины 1994 года SPEC ввела две дополнительные составные оценки: SPECrate_base_int92 и SPECrate_base_fp92, которые накладывает ограничения на используемые компиляторы.

Следует отметить, что SPEC объявила о полном переходе с середины 1996 года на новый (третий) комплект тестов - CINT95, CFP95. Эти тесты удовлетворяют следующим ограничениям и требованиям:

размер кода и данных должен быть достаточно большим, чтобы он гарантированно не размещался целиком в кэш-памяти

время выполнения тестов должно быть увеличено с секунд до минут

используемые фрагменты программ должны быть реалистичными

применение усовершенствованного способа измерения времени

реализация более удобных инструментальных средств
стандартизация требований к компиляторам и методов вызова

Новый комплект тестов состоит из 8 целочисленных программ, написанных на языке Си и 10 программ вещественной арифметики, написанных на Фортране. Новые метрики получили соответствующие названия: SPECint95, SPECfp95, SPECint_base95, SPECfp_base95, SPECrate_int95, SPECrate_fp95, SPECrate_base_int95 и SPECrate_base_fp95.

Закон Амдала.

Предположим, что в вашей программе доля операций, которые нужно выполнять последовательно, равна f , где $0 \leq f \leq 1$ (при этом доля понимается не по статическому числу строк кода, а по числу операций в процессе выполнения). Крайние случаи в значениях f соответствуют полностью параллельным ($f=0$) и полностью последовательным ($f=1$) программам. Так вот, для того, чтобы оценить, какое ускорение S может быть получено на компьютере из ' p ' процессоров при данном значении f , можно воспользоваться законом Амдала:

$$S \leq \frac{1}{f - (1 - f)p}$$

Если 9/10 программы исполняется параллельно, а 1/10 по-прежнему последовательно, то ускорения более, чем в 10 раз получить в принципе невозможно вне зависимости от качества реализации параллельной части кода и числа используемых процессоров (ясно, что 10 получается только в том случае, когда время исполнения параллельной части равно 0).

Посмотрим на проблему с другой стороны: а какую же часть кода надо ускорить (а значит и предварительно исследовать), чтобы получить заданное ускорение? Ответ можно найти в следствии из закона Амдала: для того чтобы ускорить выполнение программы в q раз необходимо ускорить не менее, чем в q раз не менее, чем $(1-1/q)$ -ю часть программы. Следовательно, если есть желание ускорить программу в 100 раз по сравнению с ее последовательным вариантом, то необходимо получить не меньшее ускорение не менее, чем на 99.99% кода, что почти всегда составляет значительную часть программы!

Отсюда первый вывод - прежде, чем основательно переделывать код для перехода на параллельный компьютер (а любой суперкомпьютер, в частности, является таковым) надо основательно подумать. Если оценив заложенный в программе алгоритм вы поняли, что доля последовательных операций велика, то на значительное ускорение рассчитывать явно не приходится и нужно думать о замене отдельных компонент алгоритма.

В ряде случаев последовательный характер алгоритма изменить не так сложно. Допустим, что в программе есть следующий фрагмент для вычисления суммы n чисел:

```
s = 0
Do i = 1, n
  s = s + a(i)
EndDo
```

(можно тоже самое на любом другом языке)

По своей природе он строго последователен, так как на i -й итерации цикла требуется результат с $(i-1)$ -й и все итерации выполняются одна за одной. Имеем 100% последовательных операций, а значит и никакого эффекта от использования параллельных компьютеров. Вместе с тем, выход очевиден. Поскольку в большинстве реальных программ

(вопрос: а почему в большинстве, а не во всех?) нет существенной разницы, в каком порядке складывать числа, выберем иную схему сложения. Сначала найдем сумму пар соседних элементов: $a(1)+a(2)$, $a(3)+a(4)$, $a(5)+a(6)$ и т.д. Заметим, что при такой схеме все пары можно складывать одновременно! На следующих шагах будем действовать абсолютно аналогично, получив вариант параллельного алгоритма.

Казалось бы в данном случае все проблемы удалось разрешить. Но представьте, что доступные вам процессоры разнородны по своей производительности. Значит будет такой момент, когда кто-то из них еще трудится, а кто-то уже все сделал и бесполезно простаивает в ожидании. Если разброс в производительности компьютеров большой, то и эффективность всей системы при равномерной загрузке процессоров будет крайне низкой.

Но пойдем дальше и предположим, что все процессоры одинаковы. Проблемы кончились? Опять нет! Процессоры выполнили свою работу, но результат-то надо передать другому для продолжения процесса суммирования... а на передачу уходит время... и в это время процессоры опять простаивают...

Словом, заставить параллельную вычислительную систему или супер-ЭВМ работать с максимальной эффективностью на конкретной программе это, прямо скажем, задача не из простых, поскольку необходимо тщательное согласование структуры программ и алгоритмов с особенностями архитектуры параллельных вычислительных систем.

Нетрадиционные архитектуры: потоковые и нейронные вычислители.

Принципы потоковой обработки информации.

Такая ситуация сохранялась в программировании примерно до начала 80-х годов прошлого века. Далее, однако, обрабатываемые данные непрерывно усложнялись, пока, наконец, для некоторых задач не превзошли по сложности программы, которые обрабатывали эти данные. Рассмотрим в качестве примера большую базу данных, которая хранит текущее состояние дел некоторого крупного предприятия. Можно сказать, что база данных содержит сложно структурированную, изменяющуюся во времени *модель* этого предприятия. Программы, обрабатывающие информацию из базы данных (БД) называются обычно системой управления базой данных (СУБД) [20]. СУБД позволяет вводить, удалять и модифицировать данные в БД, а также обрабатывать запросы на поиск и выдачу из БД нужных сведений. Так вот, сколько бы программист не исследовал программы, входящие в СУБД, он практически ничего не узнает о самом предприятии, ни что оно выпускает, ни сколько человек на нём работает и т.д. Очевидно, что в нашей модели предприятия (БД+СУБД) данные играют основную роль, а обрабатывающие их программы – уже второстепенную.

Как Вы можете догадаться, примерно в это же время появилась идея коренным образом изменить архитектуру компьютера так, чтобы отказаться от принципа *программного управления*. Таким образом, если компьютеры традиционной архитектуры управляются потоком (или потоками) команд, то компьютеры новой, нетрадиционной архитектуры должны управляться потоком данных. Можно сказать, что не команды должны определять, когда какие данные надо обрабатывать, а, наоборот, данные выбирают для себя действия (операторы), которые в определённый момент надо выполнить над этими данными. Компьютеры такой архитектуры принято называть потоковыми ЭВМ (по-английски DFC – Data Flow Computers) [1,16].

Заметим, что сам по себе принцип потоковой обработки данных не представляет собой ничего загадочного или экзотического. Например, отметим, что уже изученные Вами ранее такие алгоритмические системы, как машина Тьюринга и Нормальные алгоритмы Маркова были обработчиками данных именно этого класса. Действительно, например, в машине Тьюринга именно *данные* (текущий символ, на который указывает головка), определял, какие именно *операции* необходимо было выполнить на этом шаге работы!

В то же время оказалось, что архитектура потоковых ЭВМ весьма сложна и сильно отличается от архитектуры традиционных ЭВМ. Например, в устройстве управления таких ЭВМ нет никакого счётчика адреса, а в их памяти нет программы (по крайней мере, в нашем традиционном понимании).

Схемы потоковых вычислителей.

Мы рассмотрим функционирование такой ЭВМ на одном очень простом примере.

Пусть нам необходимо вычислить такой оператор присваивания:

$x := (x+y)*p + (x+p)/z - p*y/z$

Представим этот оператор в виде так называемого графа потока данных, показанного на рис. 16.1. В этом ориентированном графе есть два вида узлов: это сами обрабатываемые данные, изображённые в виде квадратиков (в нашем примере – значения переменных x, y, z и p) и *обрабатывающие элементы* потоковой ЭВМ, которые мы обозначили просто знаками соответствующих арифметических операций в кружочках.¹ Основная идея потоковых вычислений состоит в следующем. Все действия над данными производят обрабатывающие элементы (операторы), в нашем примере эти элементы обозначены арифметическими операциями сложения, вычитания умножения и деления. Можно сказать, что всё арифметико-логическое устройство потоковой ЭВМ – это набор таких обрабатывающих элементов. Каждый обрабатывающий элемент начинает автоматически выполняться, когда есть в наличии (готовы к обработке) требуемые для него данные. Так, в нашем примере автоматически (и параллельно!) начинают выполняться операторы $+$, $+$ и $*$ во второй строке нашего графа, затем, на втором шаге работы, параллельно выполняются операторы $*$, $/$ и $/$ в третьей строке и т.д. Здесь просматривается большое сходство со способом работы электронных схем, составленных из вентилях.

Действительно, если граф потока данных "положить на левый бок", то он будет весьма похож на электронную схему двоичного сумматора, как мы показывали её на рис. 2.2а). При этом роль вентилях будут играть обрабатывающие элементы, а входных и выходных сигналов – обрабатываемые. Заметим, что обрабатываемые данные в потоковом компьютере вовсе не являются пассивными, как в ЭВМ традиционной архитектуры, наоборот, они "громко заявляют" о своей готовности к обработке (можно сказать, требуют к себе "внимания" со стороны обрабатывающих элементов). Вообще говоря, здесь есть все основания отказаться от принципа Фон Неймана, согласно которому память только хранит данные, но не обрабатывает их. Другими словами, представляется естественным совместить функции хранения и обработки данных, и разместить обрабатывающие элементы не в арифметико-логическом устройстве, а прямо в оперативной памяти. Вообще говоря, граф потока данных и является "программой" для потоковой ЭВМ, а программ в нашем привычном понимании (запись алгоритма на языке машины) здесь не существует. Можно сказать, что здесь сами данные управляют процессом своей обработки. Отсюда можно сделать вывод, что обычные языки программирования плохо подходят для записи алгоритмов обработки данных в потоковых ЭВМ, так как приходится делать сложный компилятор, преобразующий обычную последовательную программу в граф потока данных. Неудивительно поэтому, что вместе с идеей потоковых ЭВМ появились и специальные языки потоков данных (Data Flow Languages),¹ ориентированные на прямое описание графа потока данных [1].

Из рассмотренной схемы обработки данных в потоковых ЭВМ можно сделать вывод, что в них может быть реализован принцип максимального параллелизма в обработке данных, так

как любые готовые данные тут же могут поступать на выполнение соответствующему обрабатывающему элементу потоковой ЭВМ. Ясно, что быстрее решить задачу просто невозможно. Немного подумав, можно выделить две фундаментальные трудности, которые встают при реализации потоковой ЭВМ. Во-первых, для достаточно сложного алгоритма невозможно полностью построить граф потока данных до начала счёта. Действительно, алгоритм вводит свои входные данные и содержит условные операторы, выполнение которых может, в частности, зависеть и от этих входных данных. Следовательно, компилятор с некоторого традиционного языка программирования может построить только некоторый первоначальный граф потока данных, а устройство управления потоковой ЭВМ должно будет в процессе счёта динамически изменять этот граф. Ясно, что это весьма сложная задача для аппаратуры центрального процессора потоковой ЭВМ.

Вторая проблема заключается в том, что арифметико-логическое устройство потоковой ЭВМ должно содержать много одинаковых обрабатывающих элементов. Даже для приведённого нами в качестве примера простейшего графа потока данных нужно по два обрабатывающих элемента для выполнения операций сложения и деления. Для реальных алгоритмов число таких одинаковых обрабатывающих элементов должно исчисляться десятками и сотнями, иначе готовые к счёту данные будут долго простаивать в ожидании освобождения нужного им обрабатывающего элемента, и вся выгода от потоковых вычислений будет потеряна (и тогда ни стоило, как говорится, и огород городить).

Другая трудность заключается здесь в том, что выход каждого обрабатывающего элемента (результат его работы) может быть подан на вход любого другого обрабатывающего элемента. Такие вычислительные системы называются полносвязными. При достаточно большом числе обрабатывающих элементов (а выше мы обосновали, что иначе и быть не может), реализовать все связи между ними становится технически неразрешимой задачей.

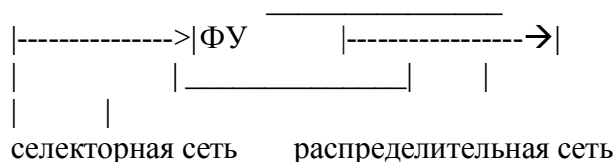
Ясно, что перед конструкторами потоковой ЭВМ встают очень большие трудности. В настоящее время универсальные потоковые ЭВМ не нашли широкого применения из-за сложности своей архитектуры, пока существуют только экспериментальные образцы таких ЭВМ. Несколько лучше обстоит дело, если обрабатывающие элементы в потоковых ЭВМ, сделать достаточно сложными, т.е. предназначенными для выполнения крупных шагов обработки данных. В этом случае каждый такой обрабатывающий элемент можно реализовать в виде отдельного микропроцессора, снабженного собственной (локальной) памятью и соответствующей программой (программой в традиционном понимании). Все такие обрабатывающие элементы объединены высокоскоростными линиями связи (шинами), причём каждый элемент начинает работать, как только он имеет все необходимые входные данные. Эти данные могут располагаться в локальной памяти или поступать на линии связи данного обрабатывающего элемента от других обрабатывающих элементов. Идея управления обработки информации потоками данных нашла применение и в таком быстроразвивающемся направлении вычислительной техники, как нейрокompьютеры.

Дополнение

Классическая архитектура потоковой ВС.

Основными компонентами потоковой ВС являются:

- память команд (ПК),
- селекторная (арбитражная) сеть,
- множество исполнительных (функциональных) устройств (ФУ),
- распределительная сеть.





Память команд состоит из "ячеек" активной памяти, каждая из которых может содержать одну команду вида <метка>: <операция>, <операнд1>, ..., <операндK>, <адр_рез1>, ..., <адр_рез.M>, где адреса результатов являются адресами ячеек памяти. С каждой командой связан подсчитывающий элемент, непрерывно ожидающий прибытие аргументов, который пересылает команду на выполнение при наличии полного комплекта аргументов. Активных характер памяти заключается в том, что ячейка, обладающая полным набором операндов, переходит в возбужденное состояние и передает в селекторную сеть информационный пакет, содержащий необходимую числовую и связующую информацию о команде.

Селекторная сеть обеспечивает маршрут от каждой командной ячейки к выбранному, в соответствии с кодом операции, исполнительному (функциональному) устройству из множества устройств. Пакет поступает на одно из исполнительных устройств, где соответствующая операция выполняется и результат подается в распределительную сеть.

Распределительная сеть обрабатывает результирующий пакет, состоящий из результатов вычислений и адресов назначения. В зависимости от содержимого пакета, результат вычислений поступает в соответствующие ячейки памяти команд, создавая, тем самым, условия возможности их активизации.

Потоковая архитектура (data-flow), как одна из альтернатив фон-Нейманновской, обладает следующими характерными чертами:

- отсутствие памяти как пассивного устройства, хранящего потребляемую информацию,
- отсутствие счетчика команд (и, следовательно, последовательной обработки команд программы, разветвлений по условию и т.д.).

Потоковые вычислительные системы позволяют использовать параллелизм вычислительных алгоритмов различных уровней, потенциально достигать производительность, недоступную традиционным вычислительным системам. Основные проблемы, препятствующие развитию потоковых машин:

1. Не решена проблема создания активной памяти большого объема, допускающей одновременную активизацию большого количества операций.
2. Создание широкополосных распределительных и селекторных сетей потоковых машин и систем управления коммуникационной сетью является сложной задачей.
3. Обработка векторных регулярных структур через механизмы потока данных менее эффективна, чем традиционные решения.
4. Языки программирования для потоковых машин существуют, в основном, в виде графических языков машинного уровня. Языки типа SISAL, ориентируемые на описания потоковых алгоритмов, достаточно сложны для программистов.

Нейронные сети.

Искусственная нейронная сеть — это математическая модель, а также устройства параллельных вычислений, представляющие собой систему соединенных и взаимодействующих между собой простых процессоров (искусственных нейронов).

Такие процессоры обычно исключительно просты, особенно в сравнении с процессорами, используемыми в персональных компьютерах.

Каждый процессор подобной сети имеет дело только с сигналами, которые он периодически получает, и сигналами, которые он периодически посылает другим процессорам. И тем не менее, будучи соединенными в достаточно большую сеть с

управляемым взаимодействием, такие локально простые процессоры вместе способны выполнять довольно сложные задачи.

Понятие возникло при изучении процессов, протекающих в мозге при мышлении, и при попытке смоделировать эти процессы. Полученные модели называются искусственными нейронными сетями (ИНС).

Нейронные сети не программируются в привычном смысле этого слова, они обучаются. Возможность обучения — одно из главных преимуществ нейронных сетей перед традиционными алгоритмами. Технически обучение заключается в нахождении коэффициентов связей между нейронами.

Разработчикам решения на основе нейронной сети требуется:
выбрать соответствующую модель сети;
определить топологию сети (число элементов и их связи);
указать параметры обучения.

Наиболее известным типом ИНС является персептрон. Доказано, что при нелинейной функции в нейроне могут быть подобраны коэффициенты связей так, что сеть может моделировать любую функцию. Для персептрона общепринятым является обучение методом обратного распространения ошибки. ИНС применяют при следующих классах задач: классификация данных, распознавание образов, предсказание временных рядов. Абсолютно точный ответ с помощью ИНС получить невозможно — например, бессмысленно решать таблицу умножения. Но есть задачи, нерешаемые другим путем. При решении таких задач самым важным является подбор данных и их подготовка.

Отличия от машин с архитектурой фон Неймана

Длительный период эволюции придал мозгу человека много качеств, которые отсутствуют в машинах с архитектурой фон Неймана:

массовый параллелизм, распределенное представление информации и вычисления, способность к обучению и обобщению, адаптивность, свойство контекстуальной обработки информации, толерантность к ошибкам, низкое энергопотребление.

Пример

Входные данные — курс акций за год. Задача — определить завтрашний курс. Проводится следующее преобразование — выстраивается ряд курс за сегодня, вчера, за позавчера, за позавчера. Следующий ряд — смещается по дате на один день и так далее. На полученном наборе обучается сеть с 3 входами и одним выходом — то есть выход курс на дату, входы курс на дату минус 1 день, минус 2 дня, минус 3 дня. Обученной сети подаем на вход курс за сегодня, вчера, позавчера и получаем ответ на завтра. Нетрудно заметить, что в этом случае сеть просто выведет зависимость одного параметра от трех предыдущих. Если желательно учитывать ещё какой-то параметр (например, общий индекс по отрасли), то его надо добавить как вход (и включить в примеры), переобучить сеть и получить новые результаты. Для наиболее точного обучения стоит использовать метод ОРО, как наиболее предсказуемый и несложный в реализации.

В последние десятилетия активно развиваются приложения на базе искусственных нейронных сетей. Среди таких приложений можно отметить следующие задачи: классификация образов, кластеризация/категоризация, аппроксимация функций, предсказание/прогноз, оптимизация, память, адресуемая по содержимому, управление. Эти

и подобные задачи успешно решаются средствами пакета Neural Networks, который входит в состав расширенных версий системы MATLAB.

Для решения задачи следует выбрать следующее: тип используемых нейронов (число входов, передаточные функции); архитектуру сети; входные и выходные параметры.

На втором этапе производится обучение созданной нейронной сети. После этого производится количественный анализ полученной модели искусственной нейронной сети. Нами использовалась модель нейронной сети, основанная на радиальном базисном нейроне. При ее «обучении» использовалась встроенная функция пакета Neural Networks системы MATLAB — `newrb`.

Результаты моделирования на наборах экспериментальных данных показали пригодность использования радиальных базисных нейронных сетей для аппроксимации функциональных зависимостей, в том числе и нелинейных. Можно отметить высокую скорость обучения и возможность применения результатов в реальных приложениях.

Добавление

Рассмотрим функцию $Y=f(X)$, которая ставит в соответствие m -мерному вектору X некоторый p -мерный вектор Y (Рис. 1). Например, в задаче классификации вектор X — это классифицируемый объект, характеризуемый m признаками; вектор Y , состоящий из одной единицы и остальных нулей, является индикатором класса, к которому принадлежит вектор X (позиция единицы означает номер класса). Функция f ставит в соответствие каждому объекту тот класс, к которому он принадлежит.

Предположим, нам нужно найти функцию f . Воспользуемся методом обучения на примерах. Предположим, что имеется репрезентативная выборка векторов X_i , для которых известно значение функции $Y_i=f(X_i)$. Набор пар (X_i, Y_i) будем называть обучающей выборкой. Рассмотрим теперь нейронную сеть, называемую многослойным перцептроном, определив, как устроены элементы сети («нейроны»), какова архитектура связей между элементами и по каким правилам будет происходить обучение сети.

Элемент сети функционирует в дискретном времени и на основании поступивших сигналов формирует результирующий сигнал. Элемент имеет несколько входов, каждому из которых предписан определенный «вес». Сигналы, поступающие по входам, суммируются с учетом соответствующих весов, и суммарный сигнал сравнивается с порогом срабатывания. Если суммарный сигнал меньше порога, то сигнал на выходе элемента близок или равен нулю, в противном случае сигнал близок к единице.

Каждый элемент входного слоя имеет один вход (с весом 1), по которому поступает соответствующая компонента вектора X . Каждый элемент скрытого слоя получает сигналы ото всех элементов входного слоя. Тем самым, элемент скрытого слоя имеет m входов, связывающих его с элементами входного слоя. Связи от элементов входного слоя к элементам скрытого слоя характеризуются матрицей «весов» связей w_1 , компоненты которой определяют величину эффективности связи. Каждый элемент выходного слоя получает сигналы ото всех элементов скрытого слоя. Таким образом, подавая на входной слой сети вектор X , мы получаем вектор активности элементов скрытого слоя и затем вектор Y на элементах выходного слоя. Результат работы сети зависит от числовых значений весов связей между элементами.

Обучение сети состоит в правильном выборе весов связей между элементами. Выбираются такие веса связей, чтобы суммарная среднеквадратичная ошибка для элементов обучающей

выборки была минимальной. Достичь этого можно разными методами. После обучения перцептрона проводится процедура тестирования, позволяющая оценить результаты работы. Для этого обучающую выборку обычно делят на две части. Одна часть используется для обучения, а другая, для которой известен результат, задействована в процессе тестирования. Процент правильных результатов работы сети на этапе тестирования является показателем качества работы перцептрона.

Надо сказать, что для очень многих практических задач удается достичь на удивление высокого качества работы сети (порядка 95% и выше). Существует ряд математических теорем, обосновывающих возможность применения многослойных перцептронов для аппроксимации достаточно широкого класса функций f .

Области применения нейронных сетей.

Области применения нейронных сетей весьма разнообразны — это распознавание текста и речи, семантический поиск, экспертные системы и системы поддержки принятия решений, предсказание курсов акций, системы безопасности, анализ текстов. Рассмотрим несколько особенно ярких и интересных примеров использования нейронных сетей в разных областях. Необходимо отметить, что мы старались по возможности выбирать наиболее ранние случаи применения нейронных сетей при решении соответствующей задачи.

Техника и телекоммуникации

В 1996 году фирмой Accurate Automation Corp(<http://www.accurate-automation.com>), Chattanooga, TN по заказу NASA и Air Force был разработан экспериментальный автопилотируемый гиперзвуковой самолет-разведчик LoFLYTE (Low-Observable Flight Test Experiment — рис. 4). Самолет имел длину всего 2,5 м и вес 32 кг и был предназначен для исследования новых принципов пилотирования. LoFLYTE использовал нейронные сети, позволяющие автопилоту обучаться, копируя приемы пилотирования летчика. Поскольку самолет был предназначен для полетов со скоростью 4-5 махов, то быстрота реакции пилота-человека могла быть недостаточной для адекватного отклика на изменение режима полета. В этом случае на помощь приходили нейронные сети, которые перенимали опыт управления у летчика и за счет высокой скорости обработки информации позволяли быстро находить выход в аварийных и экстремальных ситуациях (см. также <http://www.accurate-automation.com/Technology/Loflyte/loflyte.html> и <http://www.designation-systems.net/dusrm/app4/loflyte.html>).

Одна из важнейших задач в области телекоммуникаций, которая заключается в нахождении оптимального пути пересылки трафика между узлами, может быть успешно решена с помощью нейронных сетей. В данном случае необходимо принимать во внимание то, что, во-первых, предложенное решение должно учитывать текущее состояние сети, качество связи и наличие сбойных участков, а во-вторых, поиск оптимального решения должен осуществляться в реальном времени. Нейронные сети хорошо подходят для решения задач такого рода. Кроме управления маршрутизацией потоков, нейронные сети могут использоваться и при проектировании новых телекоммуникационных сетей, позволяя получать весьма эффективные решения.

Информационные технологии

Определение тематики текстовых сообщений — еще один пример успешного использования искусственных нейронных сетей. Так, сервер новостей Convectis (продукт компании Aptex Software, Inc.) был выбран в 1997 году компанией PointCast, Inc., являвшейся лидером персонализированной доставки новостей в Интернете, для

автоматической рубрикации сообщений по категориям. Определяя значения ключевых слов по контексту, сервер Convectis был способен в реальном времени распознавать тематику и автоматически рубрицировать огромные потоки текстовых сообщений, передаваемых по таким информационным сетям, как Reuters, NBC и CBS.

Нейросетевой продукт SelectCast от Aptex Software, Inc. позволял определять область интересов пользователей Интернета и предлагал им рекламу соответствующей тематики. Летом 1997 года компания Excite, Inc. лицензировала эту разработку для использования на своих поисковых серверах. После установки на серверах Excite и Infoseek нейросетевой рекламой было охвачено около трети всех пользователей сети на тот момент. Проведенные исследования установили, что отклик на такую тематическую рекламу была в среднем в два раза выше, чем на обычную, а для отдельных ее видов эффективность увеличивалась до пяти раз.

Распознавание речи является весьма популярным применением нейронных сетей, реализованным в ряде программных продуктов. В компании «НейроПроект» несколько лет назад была создана демонстрационная система для речевого управления встроенным в Windows калькулятором. Система позволяла без предварительного обучения уверенно распознавать каждое из 36 слов, сказанных в микрофон любым человеком. Для классификации использовалась иерархическая нейронная сеть, состоящая из двух каскадов: первый осуществлял примерное распознавание слова, относя его к одному из шести классов, а второй точно классифицировал слово внутри каждого из классов. В обучении этой нейронной сети принимали участие 19 дикторов.

Экономика и финансы

Нейронные сети активно применяются на финансовых рынках. Например, американский Citibank использует нейросетевые предсказания с 1990 года, и уже через два года после их внедрения, по свидетельству журнала The Economist, автоматический дилинг показывал доходность 25% годовых. Chemical Bank применяет нейросетевую систему фирмы Neural Data для предварительной обработки транзакций на валютных биржах ряда стран, отслеживая подозрительные сделки. Автоматизированные системы ведения портфелей с использованием нейросетей имеются на вооружении и у Deere & Co LBS Capital, причем экспертная система объединяется примерно с 900 нейронными сетями.

В сентябре 1992 года компания HNC, которая до этого занималась производством нейрокомпьютеров, выпустила программный продукт Falcon, позволяющий выявлять и предотвращать в реальном времени подозрительные сделки по краденым кредитным и дебетным картам. Искусственные нейронные сети обучались типичному поведению клиентов и могли обнаруживать резкое изменение характера покупок, сигнализирующее о возможной краже. Ежегодный ущерб крупных банков от подобных преступлений измерялся десятками миллионов долларов, но благодаря внедрению Falcon в 1994 году впервые за всю историю пластиковых карт эти потери пошли на убыль. Аналогичная система была разработана фирмой ИТС для мониторинга операций с кредитными картами Visa.

Несколько лет назад крупный канадский банк CIBC для управления рисками и идентификации злоумышленников установил программу KnowledgeSeeker фирмы Angoss. С ее помощью специалисты банка решили выяснить, кто из их клиентов в будущем будет с высокой долей вероятности задерживать выплаты по зкладным. Сначала предполагалось, что в первую очередь ими окажутся те, кто и прежде задерживал свои выплаты на несколько дней. Однако исследования показали, что в будущем проблемы с платежами возникнут у тех клиентов банка, которые на фоне регулярных выплат иногда якобы

забывали заплатить. Как выяснилось, подобная «забывчивость» была связана с серьезными финансовыми трудностями.

Реклама и маркетинг

Компания Neural Innovation Ltd использовала при работе с маркетинговыми компаниями стратегию прямой рассылки. Вначале она осуществляла рассылку всего 25% от общего числа предложений и собирала информацию об откликах и реакциях потребителей. Затем эти данные поступали на вход нейронной сети, с помощью которой осуществлялся поиск оптимального сегмента потребительского рынка для каждого товара. После этого остальные 75% предложений рассылались уже с учетом найденных закономерностей в указанный сегмент, и эффективность второй рассылки значительно возросла по сравнению с первоначальной.

При ведении бизнеса в условиях конкуренции компаниям необходимо поддерживать постоянный контакт с потребителями, обеспечивая обратную связь. Для этого некоторые компании проводят опросы потребителей, позволяющие выяснить, какие факторы являются решающими при покупке данного товара или услуги. Анализ результатов подобного опроса — непростая задача, поскольку необходимо исследовать большое количество связанных между собой параметров и выявить факторы, оказывающие наибольшее влияние на спрос. Существующие нейросетевые методы позволяют выяснить это и прогнозировать поведение потребителей при изменении маркетинговой политики, а значит, находить оптимальные стратегии работы компании.

Одно крупное английское издательство, выпускающее газеты, приобрело у фирмы Neural Innovation Ltd систему планирования цен и затрат, построенную на использовании нейронной сети и генетических алгоритмов. На основе накопленных данных эта система позволяла обнаруживать сложные зависимости между затратами на рекламу, объемом продаж, ценой газеты, ценами конкурентов, днем недели, временем года и рядом других факторов. В результате издательство могло подбирать оптимальную стратегию с точки зрения максимизации объема продаж или прибыли.

Несколько лет назад компания GoalAssist Corporation выполнила заказ крупной маркетинговой фирмы, которой требовалось исследовать стратегию поощрительных товаров (когда, например, присылая несколько этикеток с покупок, покупатель получает бесплатный сувенир). Обычные методы прогнозирования отклика потребителей в данном случае оказались неточными, в результате чего спрос на некоторые поощрительные товары оказался слишком высоким и многим покупателям пришлось подолгу ждать получения приза, в то время как другие подарки остались невостребованными. Чтобы повысить точность прогнозирования поведения потребителей, были использованы нейронные сети, обучающиеся на основе накопленной статистики. Для решения задачи применялись пакеты NeuroShell Classifier (рис. 5) и NeuroShell Predictor (рис. 6) компании Ward Systems Group (<http://www.wardsystems.com/>), а средняя ошибка предсказаний составила всего около 4%.

Здравоохранение

В свое время в США была введена в действие система обнаружения мошенничеств в области здравоохранения. Было подсчитано, что потери бюджета от такого рода фальсификаций составляют около 730 млн. долл. в год. Создание специализированной нейросетевой системы заняло у фирмы ИТС более года и обошлось всего в 2,5 млн. долл. Тестирование новой системы показало, что нейронная сеть позволяет обнаруживать 38% случаев мошенничества, тогда как использовавшаяся до нее экспертная система давала только 14%. Для настройки нейронной системы были применены также методы нечеткой логики и генетической оптимизации.

В медицинской диагностике нейронные сети нередко используются вместе с экспертными системами. Компанией «НейроПроект» была создана система объективной диагностики слуха у грудных детей. Общепринятая методика диагностики состоит в том, что в процессе обследования регистрируются отклики мозга в ответ на звуковой раздражитель, проявляющиеся в виде всплесков на электроэнцефалограмме. Для диагностики слуха ребенка опытному эксперту-аудиологу необходимо провести около 2 тыс. тестов, нейронная сеть способна с той же достоверностью определить уровень слуха уже по 200 наблюдениям в течение всего нескольких минут, причем без участия специалиста.

Приведенные примеры показывают, что технологии нейронных сетей применимы практически в любой области, а в таких задачах, как распознавание образов и прогнозирование котировок акций, они стали уже привычным и широко используемым инструментом. Повсеместное проникновение нейронных технологий в другие области — только вопрос времени. Конечно, внедрение новых наукоемких технологий — процесс сложный, однако практика показывает, что инвестиции не только окупаются и приносят пользу, но и дают тем, кто их использует, ощутимые преимущества.

Параллельное программирование

Модели программирования для систем с разделяемой, распределенной памятью.

Часто в литературе по параллельной обработке смешиваются или отождествляются понятия моделей (технологий, систем) программирования и моделей выполнения программ, т.е. моделей вычислений. В общем случае это приводит к тому, что детальный анализ структуры алгоритма и особенностей компьютерной архитектуры, на которой он реализуется, подменяется поиском системы программирования без учета того, насколько она адекватна архитектуре и решаемой задаче.

Модель вычислений служит связующим звеном между архитектурой и моделью программирования и в распределенных системах должна отражать взаимодействие процессов. Это взаимодействие представляется явно в некоторых моделях программирования и может быть организовано с помощью стандартных коммуникационных библиотек, например, MPI (Message Passing Interface) или PVM (Parallel Virtual Machine). Другие, более высокоуровневые модели освобождают программиста от выраженного управления параллельными процессами, примитивами передачи сообщений или синхронизации. Так, стандарт OpenMP может рассматриваться как модель программирования систем с разделяемой памятью и надстройка над библиотеками нитей, организуемыми с помощью интерфейса POSIX. Модель вычислений должна помогать в поисках ответа на вопрос: "Насколько эффективно реализуется программа на данной архитектуре?" Эффективность реализации может интерпретироваться как сбалансированность загрузки процессоров, согласованность степени параллелизма и трафика обмена данными, длительность простоя процессоров из-за отсутствия данных и т.д.

Несмотря на большое разнообразие моделей выполнения программ, в масштабируемых системах чаще всего используются две из них — модель обмена сообщениями и модель общей памяти (промежуточная форма — прямой доступ к удаленной памяти). Особенности моделей однако не предполагают их реализаций на архитектурах соответствующего типа. Так, первая может быть настроена над любой архитектурой. Вторая чаще всего применяется в SMP- и DSM-архитектурах, где программа рассматривается как система

нитей, взаимодействующих через общие переменные и примитивы синхронизации. При этом программист имеет дело с привычным для него единым адресным пространством. Основная проблема – обеспечение согласованного состояния памяти [2]. Модель обмена сообщениями позволяет избавиться от проблем разделения памяти и может быть надстроена над архитектурами с общей памятью. Реализация модели разделяемой памяти над архитектурами с передачей сообщений значительно сложнее [5].

Недетерминированные модели распределенной обработки, когда процессы взаимодействуют через асинхронный обмен сообщениями, адекватно представляют многие черты функционирования масштабируемых систем, такие как массовый параллелизм, взаимодействия типа "точка-точка", коммутация пакетов. Недетерминизм означает, что могут иметь место разные истории выполнения программ на одних и тех же входных данных. Помимо поддержки недетерминизма, эти модели обогащаются таким свойством, как произвольная структурированность (неоднородность) сообщений. В модели общей памяти обмен осуществляется строго структурированными данными: слово, кэш-строка, страница и т.д. В неоднородных сообщениях возможно различное число компонентов (например, пакетов), а также различие компонентов по числу операндов. В частности, в технологии SCI транзакции предусматривают передачу пакетов с блоками данных фиксированной и переменной длины. В среде Memory Channel пакеты данных могут иметь длину от 4 до 256 байтов.

Однако сочетание недетерминизма и неоднородности сообщений требует исследования реализуемости моделей (проблем однозначности результата и блокировки вычислений) и алгоритмической разрешимости задач их анализа.

Дополнение

В настоящее время существуют два основных подхода к распараллеливанию вычислений. Это параллелизм данных и параллелизм задач. В англоязычной литературе соответствующие термины - data parallel и message passing. В основе обоих подходов лежит распределение вычислительной работы по доступным пользователю процессорам параллельного компьютера. При этом приходится решать разнообразные проблемы. Прежде всего это достаточно равномерная загрузка процессоров, так как если основная вычислительная работа будет ложиться на один из процессоров, мы приходим к случаю обычных последовательных вычислений и в этом случае никакого выигрыша за счет распараллеливания задачи не будет. Сбалансированная работа процессоров - это первая проблема, которую следует решить при организации параллельных вычислений. Другая и не менее важная проблема - скорость обмена информацией между процессорами. Если вычисления выполняются на высокопроизводительных процессорах, загрузка которых достаточно равномерная, но скорость обмена данными низкая, основная часть времени будет тратиться впустую на ожидание информации, необходимой для дальнейшей работы данного процессора. Рассматриваемые парадигмы программирования различаются методами решения этих двух основных проблем. Разберем более подробно параллелизм данных и параллелизм задач.

Параллелизм данных

Основная идея подхода, основанного на параллелизме данных, заключается в том, что одна операция выполняется сразу над всеми элементами массива данных. Различные фрагменты такого массива обрабатываются на векторном процессоре или на разных процессорах параллельной машины. Распределением данных между процессорами занимается программа. Векторизация или распараллеливание в этом случае чаще всего выполняется уже на этапе компиляции - перевода исходного текста программы в машинные команды. Роль программиста в этом случае обычно сводится к заданию опций векторной

или параллельной оптимизации компилятору, директив параллельной компиляции, использованию специализированных языков для параллельных вычислений. Наиболее распространенными языками для параллельных вычислений являются Высокопроизводительный ФОРТРАН (High Performance FORTRAN) и параллельные версии языка С (это, например, С*).

Более детальное описание рассматриваемого подхода к распараллеливанию содержит указание на следующие его основные особенности:

Обработкой данных управляет одна программа;

Пространство имен является глобальным, то есть для программиста существует одна единственная память, а детали структуры данных, доступа к памяти и межпроцессорного обмена данными от него скрыты;

Слабая синхронизация вычислений на параллельных процессорах, то есть выполнение команд на разных процессорах происходит, как правило, независимо и только лишь иногда производится согласование выполнения циклов или других программных конструкций - их синхронизация. Каждый процессор выполняет один и тот же фрагмент программы, но нет гарантии, что в заданный момент времени на всех процессорах выполняется одна и та же машинная команда;

Параллельные операции над элементами массива выполняются одновременно на всех доступных данной программе процессорах.

Видим, таким образом, что в рамках данного подхода от программиста не требуется больших усилий по векторизации или распараллеливанию вычислений. Даже при программировании сложных вычислительных алгоритмов можно использовать библиотеки подпрограмм, специально разработанных с учетом конкретной архитектуры компьютера и оптимизированных для этой архитектуры.

Подход, основанный на параллелизме данных, базируется на использовании при разработке программ базового набора операций:

операции управления данными;

операции над массивами в целом и их фрагментами;

условные операции;

операции приведения;

операции сдвига;

операции сканирования;

операции, связанные с пересылкой данных.

Рассмотрим эти базовые наборы операций.

Управление данными.

В определенных ситуациях возникает необходимость в управлении распределением данных между процессорами. Это может потребоваться, например, для обеспечения равномерной загрузки процессоров. Чем более равномерно загружены работой процессоры, тем более эффективной будет работа компьютера.

Операции над массивами

Аргументами таких операций являются массивы в целом или их фрагменты (сечения), при этом данная операция применяется одновременно (параллельно) ко всем элементам массива. Примерами операций такого типа являются вычисление поэлементной суммы массивов, умножение элементов массива на скалярный или векторный множитель и т.д. Операции могут быть и более сложными - вычисление функций от массива, например.

Условные операции

Эти операции могут выполняться лишь над теми элементами массива, которые удовлетворяют какому-то определенному условию. В сеточных методах это может быть четный или нечетный номер строки (столбца) сетки или неравенство нулю элементов матрицы.

Операции приведения

Операции приведения применяются ко всем элементам массива (или его сечения), а результатом является одно единственное значение, например, сумма элементов массива или максимальное значение его элементов.

Операции сдвига

Для эффективной реализации некоторых параллельных алгоритмов требуются операции сдвига массивов. Примерами служат алгоритмы обработки изображений, конечно-разностные алгоритмы и некоторые другие.

Операции сканирования

Операции сканирования еще называются префиксными/суффиксными операциями. Префиксная операция, например, суммирование выполняется следующим образом. Элементы массива суммируются последовательно, а результат очередного суммирования заносится в очередную ячейку нового, результирующего массива, причем номер этой ячейки совпадает с числом просуммированных элементов исходного массива.

Операции пересылки данных

Это, например, операции пересылки данных между массивами разной формы (то есть имеющими разную размерность и разную протяженность по каждому измерению) и некоторые другие.

При программировании на основе параллелизма данных часто используются специализированные языки - CM FORTRAN, C*, FORTRAN+, MPP FORTRAN, Vienna FORTRAN, а также HIGH PERFORMANCE FORTRAN (HPF). HPF основан на языке программирования ФОРТРАН 90, что связано с наличием в последнем удобных операций над массивами (см., например, М.Меткалф и Дж.Рид Описание языка программирования ФОРТРАН 90, М. "Мир", 1995).

Параллелизм задач

Стиль программирования, основанный на параллелизме задач подразумевает, что вычислительная задача разбивается на несколько относительно самостоятельных подзадач и каждый процессор загружается своей собственной подзадачей. Компьютер при этом представляет собой MIMD - машину. Аббревиатура MIMD обозначает в известной классификации архитектур ЭВМ (см. в тексте классификацию Флинна – прим. А.Е.) компьютер, выполняющий одновременно множество различных операций над множеством, вообще говоря, различных и разнотипных данных. Для каждой подзадачи пишется своя собственная программа на обычном языке программирования, обычно это ФОРТРАН или С. Чем больше подзадач, тем большее число процессоров можно использовать, тем большей эффективности можно добиться. Важно то, что все эти программы должны обмениваться результатами своей работы, практически такой обмен осуществляется вызовом процедур специализированной библиотеки. Программист при этом может контролировать распределение данных между процессорами и подзадачами и обмен данными. Очевидно, что в этом случае требуется определенная работа для того, чтобы обеспечить эффективное совместное выполнение различных программ. По сравнению с подходом, основанным на параллелизме данных, данный подход более трудоемкий, с ним связаны следующие проблемы:

- повышенная трудоемкость разработки программы и ее отладки;
- на программиста ложится вся ответственность за равномерную загрузку процессоров параллельного компьютера;
- программисту приходится минимизировать обмен данными между задачами, так как пересылка данных - наиболее "времяемкий" процесс;
- повышенная опасность возникновения тупиковых ситуаций, когда отправленная одной программой посылка с данными не приходит к месту назначения.

Привлекательными особенностями данного подхода являются большая гибкость и большая свобода, предоставляемая программисту в разработке программы, эффективно использующей ресурсы параллельного компьютера и, как следствие, возможность

достижения максимального быстродействия. Примерами специализированных библиотек являются библиотеки MPI (Message Passing Interface) и PVM (Parallel Virtual Machines). Эти библиотеки являются свободно распространяемыми и существуют в исходных кодах. Библиотека MPI разработана в Аргоннской Национальной Лаборатории (США), а PVM - разработка Окриджской Национальной Лаборатории, университетов штата Теннесси и Эмори (Атланта).

Разделение последовательных программ на параллельные нити.

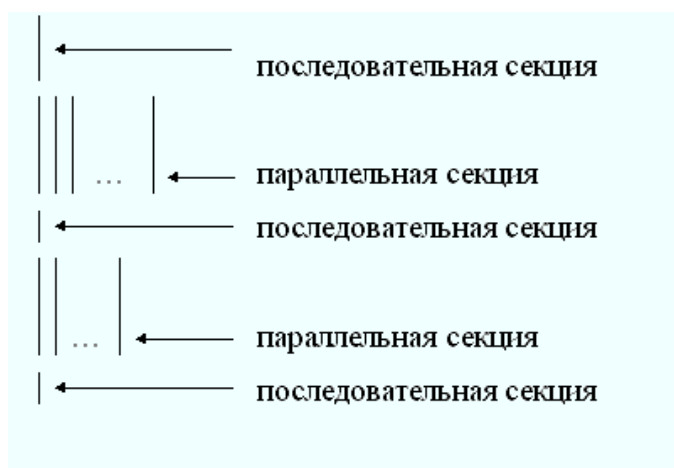


Рис. 1. Процесс исполнения OpenMP-программы

Как, с точки зрения OpenMP, пользователь должен представлять свою параллельную программу? Весь текст программы разбит на последовательные и параллельные области (см. рис.1). В начальный момент времени порождается нить-мастер или "основная" нить, которая начинает выполнение программы со стартовой точки. Здесь следует сразу сказать, почему вместо традиционных для параллельного программирования процессов появился новый термин - нити (threads, легковесные процессы). Технология OpenMP опирается на понятие общей памяти, поэтому она, в значительной степени, ориентирована на SMP-компьютеры. На подобных архитектурах возможна эффективная поддержка нитей, исполняющихся на различных процессорах, что позволяет избежать значительных накладных расходов на поддержку классических UNIX-процессов.

Основная нить и только она исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные нити. После порождения каждая нить получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все нити исполняют один и тот же код, соответствующий параллельной области. При выходе из параллельной области основная нить дожидается завершения остальных нитей, и дальнейшее выполнение программы продолжает только она.

В параллельной области все переменные программы разделяются на два класса: общие (SHARED) и локальные (PRIVATE). Общая переменная всегда существует лишь в одном экземпляре для всей программы и доступна всем нитям под одним и тем же именем. Объявление же локальной переменной вызывает порождение своего экземпляра данной переменной для каждой нити. Изменение нитью значения своей локальной переменной, естественно, никак не влияет на изменение значения этой же локальной переменной в других нитях.

По сути, только что рассмотренные два понятия: области и классы переменных, и определяют идею написания параллельной программы в рамках OpenMP: некоторые фрагменты текста программы объявляются параллельными областями; именно эти области и только они исполняются набором нитей, которые могут работать как с общими, так и с локальными переменными. Все остальное - это конкретизация деталей и описание особенностей реализации данной идеи на практике.

Рассмотрим базовые положения и основные конструкции OpenMP. Все директивы OpenMP располагаются в комментариях и начинаются с одной из следующих комбинаций: `!$OMP`, `C$OMP` или `*$OMP` (напомним, что строка, начинающаяся с одного из символов '!', 'C' или '*' по правилам языка Фортран считается комментарием). В дальнейшем изложении при описании конкретных директив для сокращения записи мы иногда будем опускать эти префиксы, хотя в реальных программах они, конечно же, всегда должны присутствовать. Все переменные окружения и функции, относящиеся к OpenMP, начинаются с префикса `OMP_`.

Описание параллельных областей. Для определения параллельных областей программы используется пара директив

```
!$OMP PARALLEL
    < параллельный код программы >
!$OMP END PARALLEL
```

Для выполнения кода, расположенного между данными директивами, дополнительно порождается `OMP_NUM_THREADS-1` нитей, где `OMP_NUM_THREADS` - это переменная окружения, значение которой пользователь, вообще говоря, может изменять. Процесс, выполнивший данную директиву (нить-мастер), всегда получает номер 0. Все нити исполняют код, заключенный между данными директивами. После `END PARALLEL` автоматически происходит неявная синхронизация всех нитей, и как только все нити доходят до этой точки, нить-мастер продолжает выполнение последующей части программы, а остальные нити уничтожаются.

Параллельные секции могут быть вложенными одна в другую. По умолчанию вложенная параллельная секция исполняется одной нитью. Необходимую стратегию обработки вложенных секций определяет переменная `OMP_NESTED`, значение которой можно изменить с помощью функции `OMP_SET_NESTED`.

Если значение переменной `OMP_DYNAMIC` установлено в 1, то с помощью функции `OMP_SET_NUM_THREADS` пользователь может изменить значение переменной `OMP_NUM_THREADS`, а значит и число порождаемых при входе в параллельную секцию нитей. Значение переменной `OMP_DYNAMIC` контролируется функцией `OMP_SET_DYNAMIC`.

Необходимость порождения нитей и параллельного исполнения кода параллельной секции пользователь может определять динамически с помощью дополнительной опции `IF` в директиве:

```
!$OMP PARALLEL IF( <условие> )
```

Если `<условие>` не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме.

Мы уже говорили о том, что все порожденные нити исполняют один и тот же код. Теперь нужно обсудить вопрос, как разумным образом распределить между ними работу. OpenMP предлагает несколько вариантов. Можно программировать на самом низком уровне,

распределяя работу с помощью функций OMP_GET_THREAD_NUM и OMP_GET_NUM_THREADS, возвращающих номер нити и общее количество порожденных нитей соответственно. Например, если написать фрагмент вида:

```
IF( OMP_GET_THREAD_NUM() .EQ. 3 ) THEN
```

```
    < код для нити с номером 3 >
```

```
ELSE
```

```
    < код для всех остальных нитей >
```

```
ENDIF ,
```

то часть программы между директивами IF:ELSE будет выполнена только нитью с номером 3, а часть между ELSE:ENDIF - всеми остальными. Как и прежде, этот код будет выполнен всеми нитями, однако функция OMP_GET_THREAD_NUM() возвратит значение 3 только для нити с номером 3, поэтому и выполнение данного участка кода для третьей нити и всех остальных будет идти по-разному.

Если в параллельной секции встретился оператор цикла, то, согласно общему правилу, он будет выполнен всеми нитями, т.е. каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву

```
!$OMP DO [опция [,] опция]:]
```

```
!$OMP END DO ,
```

которая относится к идущему следом за данной директивой оператору DO.

Опция SCHEDULE определяет конкретный способ распределения итераций данного цикла по нитям:

STATIC [,m] - блочно-циклическое распределение итераций: первый блок из m итераций выполняет первая нить, второй блок - вторая и т.д. до последней нити, затем распределение снова начинается с первой нити; по умолчанию значение m равно 1;

DYNAMIC [,m] - динамическое распределение итераций с фиксированным размером блока: сначала все нити получают порции из m итераций, а затем каждая нить, заканчивающая свою работу, получает следующую порцию опять-таки из m итераций;

GUIDED [,m] - динамическое распределение итераций блоками уменьшающегося размера; аналогично распределению DYNAMIC, но размер выделяемых блоков все время уменьшается, что в ряде случаев позволяет аккуратнее сбалансировать загрузку нитей;

RUNTIME - способ распределения итераций цикла выбирается во время работы программы в зависимости от значения переменной OMP_SCHEDULE.

Выбранный способ распределения итераций указывается в скобках после опции SCHEDULE, например:

```
!$OMP DO SCHEDULE (DYNAMIC, 10)
```

В данном примере будет использоваться динамическое распределение итераций блоками по 10 итераций.

В конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки. Если в подобной задержке нет необходимости, то директива END DO NOWAIT позволяет нитям уже дошедшим до конца цикла продолжить выполнение без синхронизации с остальными. Если директива END DO в явном виде и не указана, то в конце параллельного цикла синхронизация все равно будет выполнена. Рассмотрим следующий пример, расположенный в параллельной секции программы:

```
!$OMP DO SCHEDULE (STATIC, 2)
```

```
    DO i = 1, n
```

```

        DO j = 1, m
            A( i, j) = ( B( i, j-1) + B( i-1, j) ) / 2.0
        END DO
    END DO
!$OMP END DO

```

В данном примере внешний цикл объявлен параллельным, причем будет использовано блочно-циклическое распределение итераций по две итерации в блоке. Относительно внутреннего цикла никаких указаний нет, поэтому он будет выполняться последовательно каждой нитью.

Параллелизм на уровне независимых фрагментов оформляется в OpenMP с помощью директивы `SECTIONS : END SECTIONS`:

```

!$OMP SECTIONS
    < фрагмент 1>
!$OMP SECTIONS
    < фрагмент 2>
!$OMP SECTIONS
    < фрагмент 3>
!$OMP END SECTIONS

```

В данном примере программист описал, что все три фрагмента информационно независимы, и их можно исполнять в любом порядке, в частности, параллельно друг другу. Каждый из таких фрагментов будет выполнен какой-либо одной нитью.

Если в параллельной секции какой-то участок кода должен быть выполнен лишь один раз (такая ситуация иногда возникает, например, при работе с общими переменными), то его нужно поставить между директивами `SINGLE : END SINGLE`. Такой участок кода будет выполнен нитью, первой дошедшей до данной точки программы.

Одно из базовых понятий OpenMP - классы переменных. Все переменные, используемые в параллельной секции, могут быть либо общими, либо локальными. Общие переменные описываются директивой `SHARED`, а локальные директивой `PRIVATE`. Каждая общая переменная существует лишь в одном экземпляре и доступна для каждой нити под одним и тем же именем. Для каждой локальной переменной в каждой нити существует отдельный экземпляр данной переменной, доступный только этой нити. Предположим, что следующий фрагмент расположен в параллельной секции:

```

I = OMP_GET_THREAD_NUM()
PRINT *, I

```

Если переменная `I` в данной параллельной секции была описана как локальная, то на выходе будет получен весь набор чисел от 0 до `OMP_NUM_THREADS-1`, идущих, вообще говоря, в произвольном порядке, но каждое число встретиться только один раз. Если же переменная `I` была объявлена общей, то единственное, что можно сказать с уверенностью - мы получим последовательность из `OMP_NUM_THREADS` чисел, лежащих в диапазоне от 0 до `OMP_NUM_THREADS-1` каждое. Сколько и каких именно чисел будет в последовательности заранее сказать нельзя. В предельном случае, это может быть даже набор из `OMP_NUM_THREADS` одинаковых чисел `I0`. Предположим, что все процессы, кроме процесса `I0`, выполнили первый оператор, но затем их выполнение по какой-то причине было прервано. В это время процесс с номером `I0` присвоил это значение переменной `I`, а поскольку данная переменная является общей, то одно и тоже значение затем и будет выведено каждой нитью.

Целый набор директив в OpenMP предназначен для синхронизации работы нитей. Самый распространенный способ синхронизации - барьер. Он оформляется с помощью директивы

!\$OMP BARRIER .

Все нити, дойдя до этой директивы, останавливаются и ждут пока все нити не дойдут до этой точки программы, после чего все нити продолжают работать дальше.

Пара директив **MASTER : END MASTER** выделяет участок кода, который будет выполнен только нитью-мастером. Остальные нити пропускают данный участок и продолжают работу с выполнения оператора, расположенного следом за директивой **END MASTER**.

С помощью директив

!\$OMP CRITICAL [(<имя_критической_секции>)]

...

!\$OMP END CRITICAL [(< имя_ критической_ секции >)],

оформляется критическая секция программы. В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью P0, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока нить P0 не закончит выполнение данной критической секции. Как только P0 выполнит директиву **END CRITICAL**, одна из заблокированных на входе нитей войдет в секцию. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание. Все неименованные критические секции условно ассоциируются с одним и тем же именем.

Частым случаем использования критических секций на практике является обновление общих переменных. Например, если переменная **SUM** является общей и оператор вида **SUM=SUM+Expr** находится в параллельной секции программы, то при одновременном выполнении данного оператора несколькими нитями можно получить некорректный результат. Чтобы избежать такой ситуации можно воспользоваться механизмом критических секций или специально предусмотренной для таких случаев директивой **ATOMIC**:

!\$OMP ATOMIC

SUM=SUM+Expr .

Данная директива относится к идущему непосредственно за ней оператору, гарантируя корректную работу с общей переменной, стоящей в левой части оператора присваивания.

Поскольку в современных параллельных вычислительных системах может использоваться сложная структура и иерархия памяти, пользователь должен иметь гарантии того, что в необходимые ему моменты времени каждая нить будет видеть единый согласованный образ памяти. Именно для этих целей и предназначена директива

!\$OMP FLUSH [список_переменных].

Выполнение данной директивы предполагает, что значения всех переменных, временно хранящиеся в регистрах, будут занесены в основную память, все изменения переменных, сделанные нитями во время их работы, станут видимы остальным нитям, если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п. Поскольку выполнение данной директивы в полном объеме может повлечь значительных накладных расходов, а в данный момент нужна гарантия согласованного представления не всех, а лишь отдельных переменных, то эти переменные можно явно перечислить в директиве списком.

Ограничения на распараллеливание циклов.

Ограничения на используемые операторы в векторизуемых циклах

Существенным ограничением на конструкции, которые могут применяться в векторизуемых циклах, является использование только операторов присваивания и

арифметических выражений. Никакие команды перехода (условные ветвления, вызовы подпрограмм и функций, циклические операторы или безусловные переходы) не могут быть использованы в теле векторизуемого цикла.

Индексные выражения не должны иметь индекс в индексе A(B(C))

Из перечисленных запретов существует только два исключения. Первое - использование встроенных (INTRINSIC) в транслятор арифметических функций. Большинство таких функций реализуются в библиотеках языка, а некоторые транслируются в последовательности машинных команд. Обычно каждая функция имеет две реализации - скалярную и векторную. Про встроенные функции транслятор "знает" все, чтобы сгенерировать векторный код. Примером может служить функция cos(x). Скалярная реализация может брать свой аргумент в определенном регистре (например, r0) и возвращать значение в том же регистре сохраняя прежними значения остальных регистров. Векторный вариант может брать аргумент в векторном регистре (например, v0) и там же оставлять результат. Поэтому транслятору достаточно вычислить массив аргументов в заданном регистре, вызвать библиотечную функцию и далее работать с полученным массивом значений.

Важное замечание для любителей Си. В этом языке все математические функции внешние - они описываются в файле math.h и содержаться в дополнительной (!) библиотеке libm.a - и они не векторизуются (чаще всего). В ФОРТРАНе большое число функций (в т.ч. и комплексного аргумента) встроены. Разработчики программного обеспечения для векторных ЭВМ обычно расширяют стандартный набор функций, что позволяет использовать их в векторизуемых циклах.

Второе исключение - использование условного оператора присваивания

if(x(i) .lt. 0.0) z(i) = 0.0

Все арифметические операции (в т.ч. и само присваивание) будут выполняться не над всем векторным регистром, а только над теми его элементами, для которых было справедливо вычисленное логическое выражение (маскируемые операции). Команда сравнения устанавливает маску для каждого элемента вектора: "истина", если элемент вектора меньше нуля, и "ложь", если элемент больше или равен нулю. Команда присваивания не затронет те элементы массива z, для которых маска равна "ложь". Векторный процессор будет исполнять команды для вычисления арифметического выражения и команду присваивания, даже если не будет ни одного значения маски "истина". Это важное примечание. Команды исполнения по маске всегда будут занимать процессорное время.

Синхронизация параллельных процессов. Барьеры.

Основным механизмом синхронизации параллельных процессов, взаимодействующих с помощью передачи сообщений, является барьер. Барьер - это точка параллельной программы, в которой процесс ждёт все остальные процессы, с которыми он синхронизирует свою работу. Лишь только после того, как все процессы, синхронизирующие свою работу, достигли барьера, они продолжают дальнейшие вычисления. Если по какой-то причине хотя бы один из этих процессов не достигает барьера, то остальные процессы "зависают" в этой точке программы, а программа в целом уже никогда не сможет завершиться нормально.

Барьеры – весьма своеобразное средство синхронизации. Идея его в том, чтобы в определенной точке ожидания собралось заданное число потоков управления. Только после этого они смогут продолжить выполнение. (Поговорка "семеро одного не ждут" к барьерам не применима.)

Барьеры полезны для организации коллективных распределенных вычислений в многопроцессорной конфигурации, когда каждый участник (поток управления) выполняет часть работы, а в точке сбора частичные результаты объединяются в общий итог.

Отметим, что для барьеров отсутствует вариант синхронизации с контролем времени ожидания. Это вполне понятно, поскольку в случае срабатывания контроля барьер окажется в неработоспособном состоянии (требуемое число потоков, скорее всего, уже не соберется).

Критические секции. Двоичные и общие семафоры.

Важным понятием синхронизации процессов является понятие "критическая секция" программы. Критическая секция - это часть программы, в которой осуществляется доступ к разделяемым данным. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо обеспечить, чтобы в каждый момент в критической секции, связанной с этим ресурсом, находился максимум один процесс. Этот прием называют взаимным исключением.

Простейший способ обеспечить взаимное исключение - позволить процессу, находящемуся в критической секции, запрещать все прерывания. Однако этот способ непригоден, так как опасно доверять управление системой пользовательскому процессу; он может надолго занять процессор, а при крахе процесса в критической области крах потерпит вся система, потому что прерывания никогда не будут разрешены.

Другим способом является использование блокирующих переменных. С каждым разделяемым ресурсом связывается двоичная переменная, которая принимает значение 1, если ресурс свободен (то есть ни один процесс не находится в данный момент в критической секции, связанной с данным процессом), и значение 0, если ресурс занят. На рисунке 2.4 показан фрагмент алгоритма процесса, использующего для реализации взаимного исключения доступа к разделяемому ресурсу D блокирующую переменную F(D). Перед входом в критическую секцию процесс проверяет, свободен ли ресурс D. Если он занят, то проверка циклически повторяется, если свободен, то значение переменной F(D) устанавливается в 0, и процесс входит в критическую секцию. После того, как процесс выполнит все действия с разделяемым ресурсом D, значение переменной F(D) снова устанавливается равным 1.

Если все процессы написаны с использованием вышеописанных соглашений, то взаимное исключение гарантируется. Следует заметить, что операция проверки и установки блокирующей переменной должна быть неделимой. Поясним это. Пусть в результате проверки переменной процесс определил, что ресурс свободен, но сразу после этого, не успев установить переменную в 0, был прерван. За время его приостановки другой процесс занял ресурс, вошел в свою критическую секцию, но также был прерван, не завершив работы с разделяемым ресурсом. Когда управление было возвращено первому процессу, он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом был нарушен принцип взаимного исключения, что потенциально может привести к нежелательным последствиям. Во избежание таких ситуаций в системе команд машины желательно иметь единую команду "проверка-установка", или же реализовывать системными средствами соответствующие программные примитивы, которые бы запрещали прерывания на протяжении всей операции проверки и установки.

Реализация критических секций с использованием блокирующих переменных имеет существенный недостаток: в течение времени, когда один процесс находится в критической

секции, другой процесс, которому требуется тот же ресурс, будет выполнять рутинные действия по опросу блокирующей переменной, бесполезно тратя процессорное время. Для устранения таких ситуаций может быть использован так называемый аппарат событий. С помощью этого средства могут решаться не только проблемы взаимного исключения, но и более общие задачи синхронизации процессов. В разных операционных системах аппарат событий реализуется по своему, но в любом случае используются системные функции аналогичного назначения, которые условно назовем WAIT(x) и POST(x), где x - идентификатор некоторого события. На рисунке 2.5 показан фрагмент алгоритма процесса, использующего эти функции. Если ресурс занят, то процесс не выполняет циклический опрос, а вызывает системную функцию WAIT(D), здесь D обозначает событие, заключающееся в освобождении ресурса D. Функция WAIT(D) переводит активный процесс в состояние ОЖИДАНИЕ и делает отметку в его дескрипторе о том, что процесс ожидает события D. Процесс, который в это время использует ресурс D, после выхода из критической секции выполняет системную функцию POST(D), в результате чего операционная система просматривает очередь ожидающих процессов и переводит процесс, ожидающий события D, в состояние ГОТОВНОСТЬ.

Общий семафор - это целочисленная переменная, над которой разрешены только две неделимые операции P и V. Аргументом у этих операций является семафор. Определять семантику этих операций можно только для семафоров - положительных чисел, или включать и отрицательный диапазон. Операции могут быть определены так:

P(S) - декремент и анализ семафора

S := S-1

IF (S < 0) THEN <блокировка текущего процесса>

ENDIF

V(S) - инкремент семафора

S := S+1

IF S <= 0 THEN <запустить блокированный процесс>

ENDIF

P и V операции неделимы: в каждый момент только один процесс может выполнять P или V операцию над данным семафором. Если семафор используется как счетчик ресурсов, то:

S >= 1 - есть некоторое количество (S) доступных ресурсов,

S = 0 - нет доступного ресурса,

S < 0 - один или несколько (S) процессов находятся в очереди к ресурсу.

Вводится, также, операция инициализации семафора (присвоение начального значения).

Общий семафор - избыточный, т.к. его можно реализовать через двоичные семафоры, которые принимают только два значения 0,1.

Семафоры позволяют:

- синхронизировать работу процессов,
- управлять распределением ресурсом (использованием положительного диапазона значений семафора как счетчика ресурсов),
- организовывать очередь блокированных процессов (отрицательные значения семафора показывают число блокированных процессов).

В системах программирования вводится специальный тип данных, определяющий структуру семафора, например, SEMAPHOR, SIGNAL, GETA.

Использование семафоров

begin integer S; S:=1;

parbegin

task1: begin

do while (true)

P(S);

<критический участок 1>;

```

V(S);
<обычный участок>;
enddo
end;
task2: begin
do while (true)
P(S);
<критический участок 2>;
V(S);
<обычный участок>;
enddo
end
parend
end

```

Упорядоченные секции. Распараллелить цикл, используя упорядоченные секции и семафоры:

```

DO I=2,N
  A(I) = 2*A(I) + C(I)
  B(I) = C(I)*sin(X(I))
  Y(I) = Y(I-1) + X(I)
  D(I) = P(I)*B(I)/A(I)
ENDDO

```

Что-то на тему

Директива синхронизации OpenMP:

ORDERED ... END ORDERED //видимо, это и есть упорядоченные секции... прим. А.Е.

Определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле. Может использоваться для упорядочения вывода от параллельных нитей.

Ограничения на использования Ordered

Может появляться только в динамических расширениях следующих директив:

DO or PARALLEL DO (Fortran)

for or parallel for (C/C++)

В упорядоченную секцию допускается только одна нить в единицу времени.

Переходы на другие процессоры внутри Ordered-блока запрещены.

Итерация цикла не может исполнять одну и ту же Ordered-директиву более 1 раза и должна не исполнять более 1 Ordered-директивы.

Цикл, имеющий внутри Ordered-директиву должен быть циклом, определенным как Ordered

```

DO I=2,N
  A(I) = 2*A(I) + C(I)
  B(I) = C(I)*sin(X(I))
  Y(I) = Y(I-1) + X(I)
  D(I) = P(I)*B(I)/A(I)
ENDDO

```

Иногда, почти полностью распараллеливающийся по данным, алгоритм имеет единственную зависимость данных, которая на векторных машинах может потребовать расщепления цикла, чтобы изолировать зависимость.

Например:

```

DO I = 1,N
  A(I) = EXP(SIN(B(I))) * EXP(COS(B(I)))
  C(I) = (D(I) + E(I)) / F(I) * A(I)
  X(I) = S * X(I-1) + C(I)
  Z(I) = X(I) * Y(I) + EXP(Z(I))
ENDDO

```

Оператор присваивания $X(I)$ включает рекурсию, что запрещает векторизацию. Обычно либо компилятор, либо программист выделит этот оператор в отдельный скалярный цикл, чтобы позволить другим трем операторам присваивания выполняться в параллельном режиме.

Это не всегда необходимо делать при распараллеливании. Если имеется достаточно работы выше и/или ниже оператора присваивания $X(I)$ и N достаточно велико, можно достичь эффективного параллельного выполнения цикла путем помещения оператора присваивания $X(I)$ в упорядоченную критическую секцию. Она будет обеспечивать доступ только одной нити в один и тот же момент явным или неявным замковым (lock) механизмом. В данном случае требуется выполнять итерации в таком порядке, чтобы удовлетворить требования рекурсии для вычисления X .

Реальная система синхронизации для SEQ

```

CALL SETSEQ (SEQ,0)
* инициализация, установка SEQ=0
C$DIR DO_PARALLEL (ORDRED)
DO I = 1,N
  A(I) = EXP(SIN(B(I))) * EXP(COS(B(I)))
  C(I) = (D(I) + E(I)) / F(I) * A(I)
  CALL WAITSEQ (SEQ,I-1)
  * ждать, пока SEQ будет равно I-1
  X(I) = S * X(I-1) + C(I)
  CALL POSTSEQ (SEQ,I)
  * установить SEQ в I
  Z(I) = X(I) * Y(I) + EXP(Z(I))
ENDDO

```

Семантика (для случая четырех процессоров):

Инициализируются вычисления первых четырех итераций в заданном порядке на четырех параллельных процессорах. Все четыре процессора вычисляют свои $A(I), C(I)$ одновременно. Первый процессор выполнит для $I=1$ присваивание $X(1)$, в то время как другие ждут. Первый процессор для $I=1$ вычисляет $Z(1)$, в то время как $X(2)$ вычисляется вторым процессором для $I=2$. Первый процессор начинает вычислять новую итерацию для $I=5$, в то время как другие три процессора проходят сквозь критическую секцию также упорядочено.

Теперь, если работа по выполнению цикла хорошо сбалансирована, первый процессор для $I=5$ достигает точки WAITSEQ как раз тогда, когда четвертый процессор для $I=4$ закончит POSTSEQ. С этой точки все процессоры работают без простоя, пока они не достигнут последние четыре итерации цикла.

Системы передачи сообщений. Фортран-GNS.

Статический и динамический способы образования параллельных процессов.

"Процесс - группа ячеек памяти, содержимое которых меняется по определенным правилам. Эти правила описываются программой, которую интерпретирует процессор." /Цикритзис Д./.

Вычислительный процесс можно рассматривать как последовательность команд ЭВМ, которая работает с данными ей ресурсами. (Задача, задание, процесс, нить, TASK). Так, на микропроцессоре могут работать одновременно несколько независимых процессов: счет вычислительной задачи, редактирование текстов и т.д. Далее рассматриваются вычислительные процессы, одновременно выполняющиеся на нескольких процессорах для решения общей задачи. Работа такой задачи может начинаться с порождения главного процесса, который при работе может порождать другие процессы динамически. Эти процессы могут работать параллельно с главной и также порождать другие процессы.

Другим способом порождения процессов является статический способ, когда производится одновременная инициализация на всех процессорах одинаковых процессов (SPMD – Single Program Multiple Data). Эти процессы опрашивают состояние решающего поля и настраиваются на выполнение своей части вычислений. (Они могут узнать: сколько их порождено, свои уникальные, внутренние имена, и т.д.)

Процессы, работающие на равных правах (не вызовы процедур и не процессы, связанные понятиями "главная-подчиненная"), иногда называемые сопроцессами, могут выполняться параллельно и при этом общаться друг с другом - не часто (слабо связанные процессы).

Требования к системам программирования методом передачи сообщений.

Кое-что...

Основной моделью параллельного выполнения программы на кластере является модель передачи сообщений.

В этой модели параллельная программа представляет собой систему процессов, взаимодействующих посредством передачи сообщений.

Можно выбрать модель передачи сообщений и в качестве модели программирования. При этом возможны три способа построения языка программирования:

- Расширение стандартного языка последовательного программирования библиотечными функциями (например, Фортран+MPI);
- Расширение стандартного языка последовательного программирования специальными конструкциями (например, Fortran-GNS);
- Разработка нового языка (например, Occam).

Однако модель передачи сообщений является слишком низкоуровневой, непривычной и неудобной для программистов, разрабатывающих вычислительные программы. Она заставляет программиста иметь дело с параллельными процессами и низкоуровневыми примитивами передачи сообщений.

Поэтому вполне естественно, что прикладной программист хотел бы получить инструмент, автоматически преобразующий его последовательную программу в параллельную программу для кластера. К сожалению, такое автоматическое распараллеливание невозможно в силу следующих причин.

Во-первых, поскольку взаимодействие процессоров через коммуникационную систему требует значительного времени (латентность – время самого простого взаимодействия – велика по сравнению со временем выполнения одной машинной команды), то вычислительная работа должна распределяться между процессорами крупными порциями.

Совсем другая ситуация была на векторных машинах и на мультипроцессорах, где автоматическое распараллеливание программ на языке Фортран реально использовалось и давало хорошие результаты. Для автоматического распараллеливания на векторных машинах (векторизации) достаточно было проанализировать на предмет возможности параллельного выполнения (замены на векторные операции) только самые внутренние циклы программы. В случае мультипроцессоров приходилось уже анализировать объемлющие циклы для нахождения более крупных порций работы, распределяемых между процессорами.

Укрупнение распределяемых порций работы требует анализа более крупных фрагментов программы, обычно включающих в себя вызовы различных процедур. Это, в свою очередь, требует сложного межпроцедурного анализа. Поскольку в реальных программах на языке Фортран могут использоваться конструкции, статический анализ которых принципиально невозможен (например, косвенная индексация элементов массивов), то с увеличением порций распределяемой работы увеличивается вероятность того, что распараллеливатель откажется распараллеливать те конструкции, которые на самом деле допускают параллельное выполнение.

Во-вторых, в отличие от многопроцессорных ЭВМ с общей памятью, на системах с распределенной памятью необходимо произвести не только распределение вычислений, но и распределение данных, а также обеспечить на каждом процессоре доступ к удаленным данным – данным, расположенным на других процессорах. Для обеспечения эффективного доступа к удаленным данным требуется производить анализ индексных выражений не только внутри одного цикла, но и между разными циклами. К тому же, недостаточно просто обнаруживать факт наличия зависимости по данным, а требуется определить точно тот сегмент данных, который должен быть переслан с одного процессора на другой.

В третьих, распределение вычислений и данных должно быть произведено согласованно.

Несогласованность распределения вычислений и данных приведет, вероятнее всего, к тому, что параллельная программа будет выполняться гораздо медленнее последовательной. Если на системе с общей памятью распараллелить один цикл, занимающий 90 процентов времени решения задачи, то можно рассчитывать на почти десятикратное ускорение программы (даже если оставшиеся 10 процентов будут выполняться последовательно). На системе с распределенной памятью распараллеливание этого цикла без учета последовательной части может вызвать не ускорение, а замедление программы. Последовательная часть будет выполняться на одном процессоре или на всех процессорах. Если в этой части используются распределенные массивы, то для такого выполнения потребуется интенсивный обмен данными между процессорами.

Согласованное распределение вычислений и данных требует тщательного анализа всей программы, и любая неточность анализа может привести к катастрофическому замедлению выполнения программы.

Невозможность полностью автоматического распараллеливания имеющихся последовательных программ для их выполнения на кластерах, не означает, конечно, неактуальности работ в этом направлении. Если ввести некоторую дисциплину при

написании программ, и, возможно, позволить вставлять в программу некоторые подсказки распараллеливателю, то такие программы могут автоматически преобразовываться в программы, способные выполняться параллельно на кластере. Однако в этом случае следует говорить скорее не о распараллеливании имеющихся последовательных программ, а о написании новых параллельных программ на традиционных языках последовательного программирования или их расширениях.

Система программирования MPI.

MPI является на данный момент самой развитой системой параллельного программирования с передачей сообщений. MPI позволяет создавать эффективные и надежные, переносимые параллельные программы высокого уровня. Эффективность и надежность обеспечиваются такими механизмами как: определение MPI операций не процедурно, а логически, т.е. внутренние механизмы выполнения операций скрыты от пользователя; использование непрозрачных объектов в MPI (группы, коммутаторы, типы и т.д.); хорошей реализацией функций передачи данных, адаптирующихся к структуре системы и обеспечивающих необходимую буферизацию данных. Переносимость обеспечивается такими механизмами, как: определение одного вычислительного компьютера в виде виртуального компьютера и возможность задания произвольного количества таких виртуальных компьютеров в системе не зависимо от количества физических компьютеров (зависимость только от объема оперативной памяти в системе); компиляторы для соответствующих языков на компьютерах вычислительной системы. Уровень языка параллельного программирования определяется языковыми конструкциями, с помощью которых создаются параллельные программы. Чтобы более четко представить себе этот уровень нужно ответить на вопрос: - В чем одно из важных отличий в написании последовательной и параллельной программ? Здесь имеется ввиду параллельная программа для рассматриваемых MIMD систем. Прежде чем создавать параллельную программу, необходимо знать общую архитектуру (виртуальной) параллельной машины и топологию (виртуальных) межпроцессорных связей, которая существенно используется при программировании. Это связано с тем, что невозможно создание автоматического распараллеливателя, который позволял бы превращать последовательную программу в параллельную, и обеспечивал бы ее высокую производительность. Поэтому в программе приходится в явном виде задавать операторы инициации виртуальных топологий и операторы обменов данными между (виртуальными) процессорами. При написании же последовательной программы знать архитектуру процессора, на котором будет исполняться программа, зачастую нет необходимости, поскольку учет особенностей архитектуры скалярного процессора может быть сделан компилятором с приемлемыми потерями в производительности программы. Поэтому языковый уровень параллельной программы является заведомо ниже уровня последовательной программы, т.к. пользователю нужно в явном виде увязывать структуру алгоритма своей задачи со структурой вычислительной системы. Таким образом языковый уровень рассматривается в рамках уровней параллельных конструкций, для создания параллельных программ. Для решения своей задачи, пользователь с помощью команды (в командной строке): `mpirun [...] -np N`, задает вычислительную систему из N виртуальных компьютеров, соединенных между собой виртуальными каналами с топологией связи - "полный граф". В дополнение к этому, наличие операторов парных взаимодействий между процессорами в такой среде, достаточно для реализации всего спектра используемых в программах схем системных взаимодействий. Но отображение структуры задачи на архитектуру системы с помощью таких операторов требует некоторых усилий при программировании. Параллельная программа в этом случае будет громоздкой, плохо отлаживаемой, с неэффективным отображением структуры алгоритма на структуру системы. Наличие в системе таких средств как: виртуальные топологии, коллективные взаимодействия, создаваемые

пользователем типы данных и др., значительно повышают уровень параллельного программирования. Виртуальные топологии обеспечивают оптимальное приближение архитектуры системы к структурам задач при хорошей переносимости задач.

Главные цели создания системы параллельного программирования:

Создать интерфейс прикладного программирования для MPP систем;

Обеспечить возможность эффективных коммуникаций для коммуникации точка-точка, коллективных операций, группы процессов.

Допускать удобное сопряжение с языками C, Fortran 77, Fortran 90 и C++;

Простой способ создания процессов для модели SPMD (одна программа используется для обработки разных данных на разных процессорах);

Основные понятия языка

Группа - упорядоченное (от 0 до ранга группы) множество идентификаторов процессов. Группы служат для указания адресата при посылке сообщений (процесс-адресат специфицируется своим номером в группе), определяют исполнителей коллективных операций. Являются мощным средством функционального распараллеливания - позволяют разделить группу процессов на несколько подгрупп, каждая из которых должна выполнять свою параллельную процедуру. При этом существенно упрощается проблема адресации при использовании параллельных процедур.

Контекст - область "видимости" для сообщений, аналогичное области видимости переменных в случае вложенных вызовов процедур. Сообщения, посланные в некотором контексте, могут быть приняты только в этом же контексте. Контексты - также важные средства поддержки параллельных процедур.

Коммуникаторы - позволяют ограничить область видимости (жизни, определения) сообщений рамками некоторой группы процессов, т.е. могут рассматриваться как пара - группа и контекст. Кроме того, они служат и для целей оптимизации, храня необходимые для этого дополнительные объекты.

Имеются предопределенные коммуникаторы (точнее, создаваемые при инициализации MPI-системы): * MPI_COMM_ALL - все процессы и операции над группами (локальные, без обмена сообщениями), например, Дай размер группы. MPI_GROUP_SIZE(IN group, OUT size) Дай номер в группе обратившегося процесса. MPI_GROUP_RANK(IN group, OUT rank)

Основные операции - send, receive

Операции могут быть блокирующими и неблокирующими.

В операции send задается:

- адрес буфера в памяти;
- количество посылаемых элементов;
- тип данных каждого элемента;
- номер процесса-адресата в его группе;
- тег сообщения;
- коммуникатор.

MPI_SEND(IN start, IN count, IN datatype, IN dest, IN tag, IN comm) Типы данных - свои в MPI, но имеется соответствие между ними и типами Fortran и C.

В операции receive задается:

- адрес буфера в памяти;
- количество принимаемых элементов;
- тип данных каждого элемента;
- номер процесса-отправителя в его группе;
- тег сообщения;
- коммуникатор;
- ссылка на объект-статус, содержащий необходимую информацию о состоянии и результате операции.

Имеется возможность указать "любой отправитель" и "любой тег".

Имеется три режима коммуникаций - стандартный, режим готовности и синхронный.

В стандартном режиме последовательность выдачи операций send и receive произвольна, операция send завершается тогда, когда сообщение изъято из буфера и он уже может использоваться процессом.

В режиме готовности операция send может быть выдана только после выдачи соответствующей операции receive, иначе программа считается ошибочной и результат ее работы неопределен. В синхронном режиме последовательность выдачи операций произвольна, но операция send завершается только после выдачи и начала выполнения операции receive. Во всех трех режимах операция receive завершается после получения сообщения в заданный пользователем буфер приема.

Неблокирующие операции не приостанавливают процесс до своего завершения, а возвращают ссылку на коммуникационный объект, позволяющий опрашивать состояние операции или дожидаться ее окончания. Имеются операции проверки поступающих процессу сообщений, без чтения их в буфер (например, для определения длины сообщения и запроса затем памяти под него).

Средства описания и создания процессов в языке Фортран-GNS.

Тип данных – TASKID

По семантике языка Fortran GNS все процессы программы получают уникальные имена - системные идентификаторы, по которым производится отождествление корреспондентов при обмене сообщениями между процессами и в терминах которых задается среда выполнения функций редукции. Эти имена вырабатываются функцией порождения процессов - встроенной функцией языка NEWTASK. Структура системного идентификатора определяется реализацией, это имя не имеет внешнего представления.

Для манипулирования именами процессов внутри программы в язык Fortran GNS вводится новый для Фортрана тип данных - тип TASKID. Значения этого типа представляют собой имена процессов, которые вырабатываются функцией порождения процессов.

При статической модели параллелизма системные идентификаторы процессов вырабатываются системой запуска программы и могут быть получены пользователем встроенными функциями.

Добавляется константа .NOTASKID., которая представляет собой "нулевое" значение для объектов этого типа. Данные типа TASKID должны быть описаны в операторе TASKID, синтаксис которого аналогичен синтаксису операторов объявления типа в Фортране 77. Данные этого типа могут использоваться в операторах EQUIVALENCE, но при этом могут быть эквивалентными только данным типа TASKID. Допускается использование TASKID в качестве описателя типа подпрограммы-функции.

Использование переменных данного типа в операторах COMMON оговаривается в инструкциях по использованию языка на конкретных установках, в них же приводятся размеры имени процесса для расчетов длин общих блоков. В реализации на базе Фортран MBK [3] допускается использование данных типа TASKID в общих блоках, их длина - одно слово (как у данных типа INTEGER).

Для данных типа TASKID определены следующие операции: операции .EQ. , .NE. и присваивание (когда операнды операций, левая и правая части операторов присваивания имеют тип TASKID).

Данные типа TASKID используются в операторах передачи/приема сообщений - SEND и RECEIVE для задания абонентов - получателя или отправителя. Такие данные могут также передаваться в качестве параметров процедурам и функциям, а также могут быть переданы другим процессам с помощью операторов передачи сообщений.

Значения типа TASKID не могут использоваться в списках ввода-вывода, так как не имеют внешнего представления.

Начальная инициализация переменных этого типа также, как и других переменных Фортрана, не предусматривается, поэтому использование данных до присваивания им значений некорректно. Присвоение начальных значений этим данным в операторах DATA не предусмотрено.

Спецификация данных типа TASKID в виде массива, структура которого совпадает с конфигурацией вычислительной системы, облегчает программирование задачи, например, может унифицировать операторы передачи данных между соседними процессами. В языке есть возможность заданием идентификатора массива данных типа TASKID указать массовую операцию для всех его элементов: в функциях порождения процессов, редукции и операциях передачи сообщений. Такая запись может ускорить выполнение программы, так как такие массовые операции могут выполняться параллельно.

Для задания массовых операций над частью массива - подмассива можно присвоить подмассиву имя оператором EQUIVALENCE, динамическое формирование структур подмассивов можно производить операторами языка Фортран MBK. Например:

```
TASKID T1(100),T2(100),TM  
DIMENSION TM(100,2)  
EQUIVALENCE (TM(1,1),T1(1)),(TM(1,2),T2(1))
```

При статической модели параллелизма инициализация данных этого типа производится функциями преобразования номеров виртуальных процессоров в имена задач.

3. Задачи и процессы

В языке вводится дополнительный вид программных единиц - программные единицы-задачи. Программа может содержать (помимо традиционных для Фортрана программных единиц) одну или более программных единиц-задач.

Главная программная единица (main program) определяет главный и единственный - начальной процесс программы. Эта единица может иметь заголовок:

PROGRAM *n*, где *n* - имя программы.

По Фортрану, заголовок программы может отсутствовать.

Имя программы (MAIN при его отсутствии) идентифицирует главную программную единицу программы, ссылки на него невозможны. (Конкретные реализации могут разрешать использование этого имени в операторах передачи сообщений в качестве адресата.)

При запуске начального процесса ему присваивается другое имя - системный идентификатор, по которому на него могут ссылаться другие процессы программы при передаче сообщений или при выполнении функций редукции. Это имя может быть получено любым процессом при помощи встроенной функцией MASTER без параметров. Результат функции имеет тип TASKID.

Программная единица-задача является описанием задачи. Вычислительные процессы создаются по описанию задачи динамически и во время создания им присваиваются уникальные имена - системные идентификаторы. По одному описанию задачи может быть образовано произвольное число процессов данного класса.

Первым оператором программной единицы-задачи является оператор заголовка, имеющий вид: TASK PROGRAM *n*, где *n* - имя программной единицы-задачи.

В остальном, структура программной единицы-задачи не отличается от структуры главной программной единицы Фортрана 77. В частности, последним оператором, как обычно, является оператор END.

Задачи не могут иметь общей памяти с другими задачами, в частности, с главной, поэтому общие блоки (COMMON-блоки) задач локальны в пределах каждой задачи и программных единицах, которые в ней используются. Семантика COMMON-блоков в пределах одной задачи - традиционная.

3.1. Создание процессов.

Для создания процессов в динамической модели параллелизма используется стандартная функция NEWTASK. Обращение к этой функции производится операторами присваивания вида:

$it = \text{NEWTASK}(n, ia)$, где

n - имя программной единицы-задачи;

ia - константа, имя переменной, элемента массива или массива целого типа; значения этого параметра определяют виртуальные номера вычислителей, на которых создаются порождаемые процессы.

it - имя переменной, элемента массива или массива типа TASKID, куда заносятся имена (системные идентификаторы) порожденных процессов.

Функция NEWTASK создает по описанию задачи с именем n вычислительные процессы, количество которых определяется числом виртуальных номеров вычислителей. Это число равно количеству элементов массива параметра ia или равно одному для скалярного параметра.

Результатом выполнения функции NEWTASK является имя процесса или массив имен (системных идентификаторов) созданных процессов. Эти имена присваиваются переменным it .

Если n не совпадает с именем программной единицы-задачи, по которой образован текущий процесс, то оно должно быть описано. По аналогии с оператором EXTERNAL, для описания имен внешних задач вводится оператор спецификации TASK EXTERNAL, имеющий вид:

$\text{TASK EXTERNAL } n [, ni] \dots$ где

n, ni - имена программных единиц-задач.

Имена, указанные в списке этого оператора, разрешается использовать в качестве фактических параметров процедур (в том числе в качестве параметров функций NNAME и NEWTASK) и в качестве адресатов в операторе SEND. Если в качестве таковых параметров используется имя текущей единицы - задачи, то оно также должно быть описано как внешнее. Таким образом, передать сообщение процессу можно, идентифицируя его именем программной единицы-задачи, по которой он был образован (по имени класса), или по системному имени. Оператор TASK EXTERNAL должен быть размещен в программе-задаче до первого исполняемого оператора. Имена (системные идентификаторы) процессов не имеют внешнего представления, они скрыты от пользователя, их можно присваивать переменным типа TASKID. Процесс может определить собственное имя, имя процесса, породившего его, и имя главного процесса программы с помощью стандартных функций MYTASKID, PARENT и MASTER. Определение имен процессов по номерам виртуальных процессоров приведено в п. 7.

Созданные процессы загружаются и сразу начинают выполняться на процессорах, виртуальные (логические) номера которых заданы значениями параметра ia . Допускается использование одинаковых логических номеров для размещения процессов на одном виртуальном процессоре (для статической модели параллелизма это не так!).

Конкретное отображение виртуальных номеров процессоров на процессоры рабочей машины, способ загрузки процессов определяется реализацией, например, конструкциями языка описания конфигурации (аналогично тому, как номера каналов ввода-вывода в стандартах Фортрана не закреплены, а определяются реализацией, в частности, языком управления заданиями). Процедура создания главного процесса программы и порядок его размещения на процессоре также определяется реализацией.

При реализации статической модели параллелизма распределение процессов по процессорам производится статически, на этапе подготовки задания. В этом случае, функции NEWTASK в программе отсутствуют и процессы запускаются при инициализации задачи. В статическом варианте имена процессов (системные идентификаторы)

определяются с помощью функции GETTASKID, имеются еще ряд стандартных функций для работы в этом режиме (см. 7.).

Примеры использования функций порождения процессов

Операторы порождения будут иметь окружение программы-задачи C.

```
TASK PROGRAM C
TASK EXTERNAL A,B,READ,PRINT
TASKID TI, TM(100),TM2(100,2)
INTEGER K,KM(100),KM2(100,2)
.....
END
```

Инициализация процессов: 1 TI = NEWTASK(A,K)

2 TM=NEWTASK(A,KM)

3 TM2=NEWTASK(A,KM2)

DO 4 I=1,100

4 TM(I) = NEWTASK(A,KM(I))

DO 5 I=1,100

TM2(I,1) = NEWTASK(A,KM2(I,1))

5 TM2(I,2) = NEWTASK(B,KM2(I,2))

Операторы с метками 2 и 4 (3 и 5) производят одинаковые действия, но инициализация в форме 2 (3) предпочтительнее, так как эта запись позволяет производить инициализацию процессов параллельно.

Отсутствие в Фортране 77 аппарата вырезок из массивов не позволяет иметь такую запись для алгоритмов:

```
DO 6 I=1,99,2
```

```
TM(I) = NEWTASK(A,KM(I))
```

```
6 TM(I+1) = NEWTASK(B,KM(I+1))
```

или

```
TM(1) = NEWTASK(READ,KM(1))
```

```
TM(100) = NEWTASK(PRINT,KM(100))
```

```
DO 7 I=2,99
```

```
7 TM(I) = NEWTASK(B,KM(I))
```

В результате выполнения операторов: TI = NEWTASK(PRINT,1) 8 TI = NEWTASK(READ,2))

на виртуальном процессоре с номером 1 будет образован и запущен процесс PRINT, внутреннее имя которого потеряно и недоступно программе из-за выполнения оператора 8. Доступ к нему для передачи сообщений возможен только по программному имени, заказать от него прием сообщений невозможно. С процессом на втором процессоре возможен обмен сообщениями в обе стороны.

В языке определена только одна синтаксическая конструкция использования функции NEWTASK - в операторе присваивания. Тем не менее входные языки могут разрешать использование этой функции и в других контекстах, там, где употребляются переменные типа TASKID. Например, оператор:

SEND (NEWTASK(PRINT,MI)) RES породив процесс печати, передает ему данные через RES.

Виртуальные номера процессоров.

Кодировка виртуальных номеров процессоров произвольная, также как задание номера меток программы. Эти номера могут быть переданы процессу, который порождает другие, через каналы ввода, операторами передачи сообщений, а также сгенерированы программно.

Одинаковые номера процессоров могут быть использованы для оптимального распределения ресурсов путем выравнивания времени работы процессов. Определив время работы (число операций) самого время-емкого процесса, можно объединять на одном

процессоре выполнение нескольких других процессов, если их суммарное время меньше. При этом можно ускорить выполнение программы из-за упрощения коммуникаций между процессами, выполняемых на одном процессоре.

Виртуальные номера процессоров могут быть использованы для настройки решающего поля. Параллельные алгоритмы, формулируемые в теоретической постановке обычно как n -размерные, при программировании приводятся к параметрам реальной структуры решающего поля. Алгоритмы, реализуемые как стандартные процедуры, также должны адаптироваться к структуре выделяемых ресурсов. Настройка структуры (под) программ под параметры вычислительной среды может управляться заданием номеров виртуальных вычислителей.

При запуске программы на конкретной вычислительной платформе возможно потребуется сопроводить программу спецификацией об используемых значениях номеров виртуальных процессоров и схемой их оптимальной дислокации на вычислителях платформы, и об именах задач, связанных с этими номерами для последующей загрузки соответствующих процессов. Конкретные реализации, оговаривая правила загрузки задач на процессоры, могут содержать средства задания порядка выполнения (приоритета) нескольких процессов на одном процессоре.

Для статической модели параллелизма балансировка загрузки вычислительной системы может производиться на уровне языка конфигурации, размещением на одном физическом процессоре нескольких виртуальных.

3.2. Завершение программы и завершение процессов. Выполнение программы на языке Fortran GNS завершается в следующих случаях:

- а) при завершении выполнения главного процесса (процедурой STOP или выходом через END главной программной единицы);
- б) при "авостве" в главном процессе;
- в) при выполнении в каком-либо процессе программы стандартных процедур ABORT и FINISH.
- г) при выполнении в каком-либо процессе программы оператора FINISH.

Из определения следует, что для завершения программы не требуется ожидания естественного (или авостного) завершения всех инициированных процессов программы, в частности, завершения всех обменов сообщениями между ними и выборки поступивших сообщений из почтовых ящиков. Вопрос о буферах ввода-вывода при "авостве" в главном процессе решается соглашениями ОС.

Выполнение одного процесса завершается в следующих случаях:

- а) если выполняется одно из указанных выше условий, приводящих к завершению всей программы;
- б) при выполнении оператора STOP в данном процессе;
- в) при выполнении оператора END в задаче, по которой образован процесс; RETURN, END в функциях и процедурах имеет традиционный смысл.
- г) в случае "авоста" в данном процессе.

Таким образом любой процесс может завершить выполнение всей программы выполнением стандартных процедур ABORT, FINISH; завершение процесса другими способами не прекращает выполнение программы. В частности, при выполнении оператора END в задаче, процесс завершается, ссылка на него сохраняется. Проблема почтовых ящиков завершенных процессов решаются при реализации языка.

При статической модели параллелизма работа программы завершается выполнением в одном из процессов ABORT или FINISH.

В языке нет средств для определения статуса процесса, поэтому, проблемы использования процессора, освободившегося по завершению выполняемого на нем процесса, связаны с наличием динамической загрузки процессов и должны решаться программированием синхронизирующих механизмов.

Средства передачи и приема сообщений в языке Фортран-GNS.

Передача сообщений

Обмен информацией между процессами осуществляется с помощью передачи сообщений. Для этого используются операторы:

SEND - послать сообщение и RECEIVE - принять сообщение.

4.1. Способы передачи сообщений

Предусмотрено три способа передачи сообщений: синхронный, асинхронный и передача без ожидания.

а. Синхронный способ

При синхронном способе передачи сообщений посылающий и принимающий процессы приостанавливают выполнение своих программ и переходят в состояние ожидания до тех пор, пока не выполнятся оба синхронных оператора SEND и RECEIVE.

Сообщения данного типа могут передаваться без использования буферов почтовых ящиков, так как передача синхронного сообщения будет производиться только при готовности абонентов к обмену и может происходить непосредственно из памяти передающего процесса в память принимающего процесса.

Процесс может посылать синхронные сообщения как отдельному процессу, так и группе процессов. В последнем случае процесс отправитель продолжит выполнение только после получения подтверждения о выполнении соответствующих операторов RECEIVE во всех процессах - получателях. Получив сообщение, получатель группового синхронного сообщения может продолжить работу, не ожидая завершения всех остальных обменов по данному оператору SEND. Получатель синхронного сообщения может ждать его только от процесса, имя которого указано в параметре оператора RECEIVE, т.е. нельзя заказать в одном операторе ожидание синхронных сообщений от нескольких процессов.

б. Асинхронный способ

При асинхронном способе передачи сообщения посылающий процесс продолжает выполнение независимо от того, выполнил ли получающий процесс оператор RECEIVE или нет.

Сразу же после выполнения оператора SEND процесс может изменить значения данных, перечисленных в списке передаваемых значений этого оператора, однако абоненту передаются те значения, которые были на момент выполнения оператора SEND.

Выполнение асинхронного оператора RECEIVE и продолжение работы процесса - получателя производиться только после поступления сообщения. Если сообщение пришло к получателю до выполнения у него оператора RECEIVE, то сообщение помещается в почтовый ящик процесса без прерывания его выполнения. Сообщения с одинаковым значением тега от одного и того же отправителя помещаются в почтовом ящике в том же порядке, как они посылались и в таком же порядке они будут выбраны операторами RECEIVE.

Сообщения, передаваемые асинхронным способом, идентифицируются именами отправителя и тегами (положительными значениями целого типа). Имя отправителя можно опускать в процессе - получателе, т.е. можно заказывать ожидание сообщений этого типа (с заданным тегом) от произвольного процесса. В этом случае отправителя можно идентифицировать при получения сообщения при помощи параметра SENDER в операторе RECEIVE.

Стандартные функции TESTMSG, TESTTAG, PROBE позволяют анализировать наличие сообщений, значения тегов и имен отправителей асинхронных сообщений в почтовом ящике процесса и, соответственно, управлять порядком выбора поступивших сообщений.

Для организации селекции и установки режимов ожидания заданных асинхронных сообщений можно использовать также конструкции выбора (см. раздел 5.).

в. Передача без ожидания

При передаче сообщения способом "без ожидания" момент передачи сообщений (копирование данных для передачи в процессе отправителе и присваивание соответствующим переменным значений данных из принятого сообщения в процессе получателе) не определен в языке и зависит от системы интерпретации. При такой передаче работа обоих процессов (отправителя и получателя) продолжается после выполнения операторов этого вида, независимо от того, выполнялась ли фактически передача сообщения.

Так как значения данных, указанных в списке переменных оператора SEND, могут быть изменены любым следующим оператором процесса, то передаются такие значения, которые имеют эти переменные в момент фактической передачи. Оператор приема сообщений данного типа формирует за-

явку на прием сообщения, а переменные, перечисленные в списке не изменяются, если сообщение еще не поступило в почтовый ящик процесса на момент выполнения оператора. При поступлении сообщения, данные пересылаются из почтового ящика в память процесса (присваиваются соответствующим переменным) без прерывания его работы.

В каждом операторе обмена этого типа один из его параметров - логическая переменная FLAG служит для фиксации факта передачи сообщения. Значение этой логической переменной-флага связывается с фактом выполнения передачи данных, заданной оператором обмена для того процесса, в котором определена эта переменная. Система интерпретации присваивает флагу в процессе отправителе значение .TRUE. в момент снятия копии с данных отправителя для передачи, и флагу в процессе-получателе после присвоения этих значений переменным получателя. До выполнения оператора передачи / приема сообщения значение логической переменной - флага, связанной с этим оператором не определено.

Определить факт передачи данных и приостановить выполнение процессов до выполнения фактической передачи сообщения этого типа (для отправителя - до снятия копии с передаваемых данных, для получателя - до поступления сообщения в почтовый ящик или записи переданных данных в поле задачи) можно при помощи стандартных процедур - MSGDONE, ALLDONE, TESTFLAG.

Только установка значения флага в .TRUE. отправителю дает возможность изменять переменные, перечисленные в операторе SEND, без риска изменить передаваемые данные, для получателя - начать использовать полученные данные. Использование этих логических переменных вне операторов передачи сообщений и перечисленных выше процедур не допускается.

По аналогии с асинхронным протоколом допускается ожидание асинхронных сообщений от произвольного процесса (в операторах RECEIVE можно опускать имя отправителя).

Замечание:

Понятие "почтовый ящик процесса", используемое в данном описании языка, носит методический характер и служит моделью для пояснения семантических понятий аппарата передачи сообщений. Алгоритмы реализации этого аппарата зависят от особенностей вычислительной платформы и ее системного обеспечения.

Протоколы передачи и приема сообщений в языке Фортран-GNS.

Операторы передачи сообщений

Операторы SEND и RECEIVE имеют вид:

1.0

SEND (*sm* [,*ss*]...) [*список*]
 RECEIVE (*rm* [,*rs*]...) [*список*]

1.5 где

sm или *rm* - спецификация сообщения - определяет адресата (процесс или процессы, которым посылается сообщение) или отправителя и способ синхронизации;

список - список передаваемых данных имеет такой же вид, как списки в операторах ввода/вывода Фортрана 77, т.е. элементом списка может быть константа, имя переменной, переменная с индексом, имя массива или неявный цикл;

ss или *rs* - дополнительная спецификация.

Вид спецификации сообщения *sm* и *rm* зависит от способа синхронизации.

а) Синхронный способ: *sm* **есть** [TASKID =] *адресат*,

где *адресат* **есть** *adr* или ALL, а

adr - ссылка на функцию, имя переменной, элемента массива или имя массива типа TASKID или имя программной единицы-задачи.

Если *adr* - имя программной единицы-задачи, то сообщение посылается всем процессам программы, образованным по образцу указанной программной единицы (исключая посылающую).процедуры.

ALL - означает, что сообщение посылается всем процессам программы, образованным на момент выполнения оператора передачи сообщения,исключая процесс-отправитель. *rm* **есть** [TASKID =] *t*, где *t* - переменная или элемент массива типа TASKID; параметр специфицирует процесс - отправитель.

Когда *adr* (*rm*) - значение типа TASKID, оно должно ссылаться на незавершенный процесс.

б) Асинхронный способ

sm **есть** [TAG=] *ie*, [TASKID =] *адресат*

или TASKID = *адресат*, [TAG =] *ie*

rm **есть** [TAG=] *ie* [, [TASKID =] *t*]

или TASKID = *t*, [TAG =] *ie*

где *адресат* и *t* определяются как *rm* для синхронного способа,

ie - выражение целого типа, значение которого определяет тег сообщения.

в) Способ без ожидания

sm **есть** [TASKID =] *адресат*, FLAG = *l*

rm **есть** [TASKID =] *adr*, FLAG = *l*

или [FLAG =] *l*

где

адресат и *adr* определяется как и для предыдущего способа;

l - имя логической переменной.

Переменная *l* может использоваться также в стандартных процедурах MSGDONE(*l*) и TESTFLAG(*l*) (см. п.7.). В других ситуациях ее использование не допускается.

Дополнительная спецификация

Дополнительная спецификация *ss* и *rs* , как видно из синтаксиса, является необязательной.

ss **есть** ERR = *l*

и / или SRSTAT = *ios*

где

l - специфицирует метку, которой передается управление в случае ошибки при передаче сообщения;

ios - имя переменной или элемента массива целого типа; специфицирует статус состояния (аналог спецификации IOSTAT в операторах ввода/вывода).

В результате выполнения оператора, переменной *ios* присваивается значение 0, если не было обнаружено ошибки, и положительное значение, если обнаружена ошибка. Классификация видов ошибок определяется реализацией языка. Если спецификация SRSTAT отсутствует, то значение статуса состояния присваивается системной переменной.

Если спецификация ERR отсутствует, то в случае ошибки, задача (и вся программа) завершается выполнением стандартной процедуры ABORT.

В качестве дополнительной спецификации *rs* для оператора RECEIVE, помимо аналогичных спецификаций ERR и SRSTAT, можно также использовать необязательную спецификацию SENDER, т.е.

rs есть ERR = *l*

и / или SRSTAT = *i*

и / или SENDER = *t*

где *t* - переменная типа TASKID.

Спецификацию SENDER полезно использовать, если отсутствует имя процесса-отправителя. Переменной *t* после выполнения оператора RECEIVE присваивается значение имени процесса-отправителя.

Замечание.

1.Способ передачи сообщения и число элементов в списках оператора SEND и соответствующего оператора RECEIVE должны совпадать; для каждого элемента - тип, а для массивов также размерность и конфигурация должны соответствовать друг другу. Конфигурация массива - это одномерный массив целого типа, размер которого равен размерности исходного массива, а элементы равны размерам соответствующих измерений.

Несоответствие структуры данных при синхронной передаче считается ошибкой в обоих процессах, при других способах передачи - ошибкой в задаче-отправителе.

2. Если значение переменной адресата есть .NOTASKID. или ссылка на завершённый процесс, оператор обмена для этого процесса не выполняется, а спецификация SENDER получает значение .NOTASKID. .

3. Не считается ошибкой наличие в почтовом ящике невостребованных сообщений при завершении процесса.

Использование операторов передачи сообщений

Синхронный режим передачи сообщений

По определению, синхронный режим передачи сообщений требует одновременного выполнения в процессах соответствующих операторов, что при рассогласовании работы процессов может быть источником "зависания" программы. Достоинством данного метода передачи сообщений является принципиальная возможность проводить обмен данными без использования системных буферов, что может ускорить время обмена.

Операторы передачи сообщений с пустым списком данных могут использоваться для синхронизации вычислений программы.

Для этого в процессе, управляющем синхронизацией, следует выполнить оператор: SEND(ALL), во всех процессах программы - RECEIVE (T1), где T1 должен ссылаться на управляющую задачу. Управляющий процесс продолжится только после выполнения во всех задачах оператора RECEIVE. Оператор SEND(A) требует синхронизации только от процессов, образованных по программе-задаче A, а SEND(TM) - от процессов, на которые ссылаются элементы массива TM. Если процессам известно, что randevu будет требовать начальный процесс, то их синхронизирующий оператор может быть записан: RECEIVE (MASTER()). Точная синтаксическая запись: T1=MASTER()

RECEIVE (T1)

так как в параметрах операторов передачи сообщений не разрешается использования выражений. Реализации могут иметь расширенное толкование семантики, в частности, допускать использование функций типа TASKID в позициях переменных этого типа. Некоторые реализации могут запрещать в операторах приема сообщений неявные циклы, управляемые параметром, находящимся в этом же списке ввода. Так, в Фортране MBK запрещены такие конструкции: RECEIVE (<.>) N, (KM(I),I=1,N)

Асинхронный режим передачи сообщений

Использование данного способа обмена сообщениями делает программу еще менее критичной к согласованию операторов обмена сообщениями, так как процесс-отправитель продолжает работу после передачи сообщения, не дожидаясь конца фактической передачи сообщения (и даже начала, так как система интерпретации "должна" сразу же, скопировав передаваемые данные в буфер, "отпустить" процесс) . Получатель сообщений этого типа может выдавать директиву приема сообщения, только удостоверившись в наличии нужного сообщения в почтовом ящике процесса при помощи функций: TESTMSG,TESTTAG,PROBE. Выполнение оператора RECEIVE без проверки наличия сообщения в почтовом ящике процесса, по аналогии с синхронным способом обмена, приводит к задержки выполнения процесса до приема сообщения с заказанным тегом и ,возможно, с заданным именем отправителя. Но в отличие от синхронного способа, асинхронный способ позволяют проводить селекцию поступающих сообщений при помощи конструкций выбора. Пусть процесс может получать сообщения с тегом 1, но от процесса TI - скаляр целого типа, а от процесса TM(1) - массив из 100 целых чисел. Тогда,прием таких сообщений может быть запрограммированно так:

```
IF (TESTTAG(1)) THEN
  SELECT MESSAGE
  CASE(1,TI)
  RECEIVE (1) K
  CASE(1,TM(1))
  RECEIVE (1) KM
  END SELECT
END IF или:
```

```
SELECT MESSAGE
CASE(1,TI)
RECEIVE (1) K
CASE(1,TM(1))
RECEIVE (1) KM
CASE DEFAULT
GO TO 2
END SELECT 2  CONTINUE
```

Ожидание этих сообщений, то есть, по аналогии с синхронной передачей сообщений, прерывание работы процесса до получения сообщения (в данном случае любого из ожидаемого) записывается так:

```
SELECT MESSAGE
CASE(1,TI)
RECEIVE (1) K
CASE(1,TM(1))
RECEIVE (1) KM
END SELECT
```

Если сообщения различаются тегами, то конструкция их ожидания может иметь вид:

```
SELECT MESSAGE
CASE(1)
RECEIVE (1) K
```

```
CASE(2,TM(1))  
RECEIVE (2) KM  
END SELECT
```

Режим передачи сообщений без ожидания

Передачи сообщений этого типа рекомендуется использовать для передачи данных "впрок", заблаговременно. Если после операторов SEND / RECEIVE без ожидания, поместить операторы: CALL MSDONE(L), семантика операторов будет совпадать с семантикой асинхронных операторов. Следует иметь в виду, что при использовании операторов данного вида в цикле без использования процедур ожидания фактической передачи сообщений возможно искажение передаваемых данных; семантика функции TESTFLAG при этом будет двусмысленной.

Структура элементов списка передаваемых сообщений

Структура элементов списка передаваемых (принимаемых) сообщений, как объявлено, совпадает с списком ввода вывода операторов обмена Фортрана. Широкие возможности предоставляет аппарат неявных циклов.

Пусть имеются описания: REAL A(10),B(5,5). Тогда оператор:

SEND(ALL)A,(B(I),I=1,5),(B(I,1),I=1,5),(B(I,I),I=1,5) перешлет всем процессам программы следующие объекты: - массив A целиком, - первый столбец массива B, - диагональные элементы массива B.

Этим способом можно организовать не только передачу (прием) любых вырезок их массивов, но и множества одинаковых данных, например, переслать пять копий массива B: SEND(ALL)(B,I=1,5)

Идентификация абонентов при передачи сообщений в языке Фортран-GNS.

Идентификация абонентов

Идентификация абонентов при организации передачи сообщений может производиться по имени процесса (массиву имен) или идентификатору задачи, по описанию которой образован процесс. Есть способы безадресных (широковещательных) посылок сообщений - "всем", и приема от произвольного отправителя сообщений с заказанной "маркой" - тегом. Идентификация абонентов может быть произведена:

а) По имени процесса

Основным способом идентификации абонентов при передаче сообщений является задание имени процесса (системного идентификатора) переменными типа TASKID и стандартными функциями этого типа MYTASKID, PARENT и MASTER. Использование этих стандартных функций является единственным первоначальным способом сослаться из порождаемых процессов на имена порождающих процессов, так как начальный диалог между процессами возможен только на уровне порожденный-порождаемый. Путем обмена сообщениями можно организовать передачу имен процессов для организации произвольной абонентской сети.

Переменные типа TASKID - параметры операторов передачи сообщений могут:

- ссылаться на функционирующий процесс,
- ссылаться на заверченный процесс,
- иметь значение .NOTASKID.,
- быть не инициализированными.

Реакция программы на три последних случая не оговаривается в языке и определяется реализацией. Одно из решений - игнорирование оператора обмена сообщениями с такими параметрами (кроме синхронных операторов, для которых такой случай следует считать "авостом" в программе). Процесс-отправитель имеет возможность организовать дублирование передаваемых сообщений для передачи их одновременно нескольким процессам-получателем. Для этого допускается задание абонентов в параметрах оператора передачи сообщений в виде массива типа TASKID. Сообщение будет передано при этом всем процессам, на которые есть ссылки через элементы массива. Данная массовая операция предпочтительнее передачи сообщений, организованной по-элементной циклической конструкцией. Цикл предписывает порядок перебора элементов массива - абонентов передачи, а для синхронного обмена и завершение очередной передачи до посылки следующего сообщения. Массовая операция обмена может проводиться параллельно и независимо для всех заказанных абонентов. Массовые операции асимметричны, нельзя заказать ожидание сообщения от нескольких процессов сразу. Теговая идентификация сообщений и конструкции выбора допускает некоторую вольность в указаниях об именах отправителей.

б) По имени программной единицы-задачи

Задание в операторе передачи сообщений в качестве адресата - получателя имени программной единицы-задачи означает рассылку сообщений всем процессам, образованным по описанию указанной задачи на момент выполнения оператора. Случай, когда к этому моменту не было образовано ни одного процесса по указанному образцу, можно рассматривать по аналогии с передачей сообщения процессу с "именем" - .NOTASKID. . Возможность задать в качестве адресата имя главного процесса уточняется в описаниях входных версий языка.

в) Безымянные абоненты

Специальный идентификатор ALL в параметре адресата-получателя сообщения предписывает передать данное сообщение всем процессам программы, инициализированным на данный момент.

Передача сообщений двумя последними способами предполагает возможность параллельной рассылки сообщений и исключение процесса, выполняющего оператор передачи сообщения, из списка абонентов - получателей.

г) Помеченные сообщения

При передаче сообщений асинхронным способом сообщения при отправлении помечаются целым числом - тегом. Эта разметка дает возможность абонентам идентифицировать сообщения от разных операторов передачи сообщений одного процесса. Получив сообщение с заданным тегом, процесс может узнать имя отправителя, через аппарат SENDERов. Это замечание верно и для передачи сообщений без ожидания.

Адаптация последовательных программ к параллельным архитектурам. Векторизация и распараллеливание циклов. Методы координат, гиперплоскостей.

Автоматическое распараллеливание последовательных программ.

11.6 Автоматическое распараллеливание

Мультипроцессорные комплексы позволяют воспользоваться преимуществами параллелизма. Большинство программ пишутся для последовательного выполнения. Вычислительные системы получают больше пользы от параллельной обработки благодаря мультипрограммному выполнению нескольких процессов, чем используя параллелизм в

рамках одного процесса. Обнаружение параллелизма (распараллеливание), выполняемое программистами, языковыми трансляторами, аппаратными средствами или операционными системами — это сложная проблема, вызывающая сегодня особый интерес. Независимо от того, каким образом в конце концов будет обнаруживаться параллелизм, мультипроцессорные системы позволяют с успехом использовать его, одновременно выполняя параллельные ветви вычислений.

Параллелизм в программах может быть либо явным, либо неявным (скрытым). Явный параллелизм программист указывает в своей программе при помощи специальной конструкции, обозначающей параллельное вычисление, например COBEGIN/COEND, следующим образом:

```
COBEGIN;  
оператор-1;  
оператор-2;  
...  
оператор-n;  
COEND;
```

В мультипроцессорной системе, рассчитанной на реализацию параллелизма, каждый из программных операторов может выполнять отдельный процессор, с тем, чтобы все вычисления завершались быстрее, чем при чисто последовательной работе.

Явное указание параллелизма налагает определенную ответственность на программиста. Это достаточно трудоемкая процедура, причем в ряде случаев программист может ошибочно указать, что определенные операции можно выполнять параллельно, в то время как в действительности этого делать нельзя. Программист может упустить многие ситуации, допускающие распараллеливание. Весьма вероятно, что программист обнаружит параллелизм и закодирует его явным образом в наиболее очевидных ситуациях. Однако многие случаи параллелизма вручную в алгоритмах обнаружить трудно, поэтому они будут просто пропускаться.

Одно из довольно неприятных последствий явного указания параллелизма заключается в том, что программы могут оказаться более сложными для модификации. При внесении изменений в программу, имеющую большое число явных параллельных конструкций, легко можно допустить ошибки, причем, вообще говоря, довольно тонкие.

Реально, на что можно надеяться при решении подобной проблемы,— это на автоматическое обнаружение неявного параллелизма, т. е. параллелизма, присущего алгоритму, но не указанного явно программистом. В компиляторы, операционные системы и аппаратные средства компьютеров необходимо включать специальные механизмы распараллеливания. Это с гораздо большей вероятностью, чем явное указание параллелизма, обеспечит создание быстро выполняющихся и корректных программ.

Два распространенных способа, реализуемых в компиляторах для использования неявного параллелизма программ,— это расщепление цикла и редукция высоты дерева.

11.6.1 Расщепление цикла

Программный цикл предполагает многократное выполнение некоторой последовательности операторов, называемой телом цикла, до тех пор, пока не станет истинным соответствующее условие завершения. В определенных случаях в тело цикла входят

операторы, которые могут допускать параллельное выполнение. Рассмотрим следующий цикл:

```
FORI = 1 TO 4 DO  
A(I)=B(I)+C(I);
```

Этот цикл суммирует соответствующие элементы массивов В и С и помещает суммы этих элементов в массив А. По данному программному циклу последовательный процессор выполняет поочередно следующие операции:

```
A(1)=B(1)+C(1);  
A(2)=B(2)+C(2);  
A(3)=B(3)+C(3);  
A(4)=B(4)+C(4);
```

В мультипроцессорной системе эти операторы можно выполнять параллельно при помощи четырех процессоров, благодаря чему существенно уменьшится время выполнения всего цикла. Компилятор, осуществляющий автоматическое распараллеливание, может преобразовать приведенный выше цикл в следующую форму:

```
COBEGIN;  
A(1)=B(1)+C(1);  
A(2)=B(2)+C(2);  
A(3)=B(3)+C(3);  
A(4)=B(4)+C(4);  
COEND;
```

Тем самым он указывает вычислительной системе на возможность параллельного выполнения операторов. Этот способ называется расщеплением, или разбиением программного цикла на параллельные цепочки, и его относительно легко реализовать. Естественно, что во многих программных циклах число повторений гораздо больше четырех, как в данном примере, поэтому метод разбиения цикла обычно дает более высокую степень параллелизма, чем количество имеющихся процессоров. Операционная система и аппаратные средства компьютера должны затем решить, какое подмножество параллельных ветвей будет выполняться одновременно.

11.6.2 Редукция высоты дерева

Применяя ассоциативные, коммутативные и дистрибутивные свойства арифметики, компиляторы могут обнаруживать неявный параллелизм в алгебраических выражениях и генерировать объектный код для мультипроцессорных систем с указанием операций, которые можно выполнять одновременно. Компиляторы должны действовать в соответствии со старшинством операций. Не все компиляторы используют одинаковые правила старшинства, однако во многих случаях применяются примерно следующие правила:

1. Прежде всего выполняются операции, указанные во вложенных скобках, и, в частности, те операции, которые имеют более глубокий уровень вложенности.
2. Затем выполняются операции в скобках.
3. Затем выполняются операции возведения в степень.
4. Затем выполняются операции умножения и деления.
5. Затем выполняются операции сложения и вычитания.

6. Если операции, которые можно выполнить следующими, имеют одинаковое старшинство, то они выполняются слева направо.

Применение таких правил старшинства операций дает компилятору возможность устанавливать однозначный последовательный порядок действий для алгебраического выражения, а затем переходить к генерации машинного кода, реализующего вычисление этого выражения.

Зачастую однозначность и последовательность порядка действий при вычислениях не обязательны. Например, поскольку операции сложения и умножения являются коммутативными, компилятор, генерирующий код для перемножения p и q , может выдать либо $p*q$, либо $q*p$; результаты вычислений будут, естественно, одинаковыми. Аналогично, сложение p и q можно выполнить либо как $p+q$, либо как $q+p$. Используя свойство коммутативности, а также ассоциативности и дистрибутивности, компилятор может довольно гибко перестраивать выражения, делая их более удобными для параллельных вычислений.

На рис. 11.2 показано, как обычный компилятор оттранслировал бы алгебраическое выражение и как оно могло бы быть обработано распараллеливающим компилятором. Левая диаграмма на этом рисунке — структура дерева, показывающего, каким образом обычный компилятор может генерировать код программы для выполнения приведенных операций. Цифры 1, 2 и 3 показывают порядок,

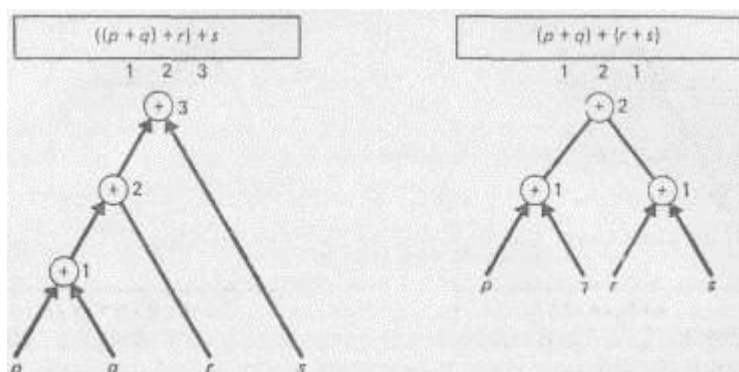


Рис. 11.2 Уменьшение высоты дерева за счет ассоциативности.

в котором должны выполняться эти операции. Используя ассоциативность сложения, можно преобразовать выражение $((p+q)+r)+s$ в выражение вида $(p+q)+(r+s)$, которое более приспособлено для параллельных вычислений. Структуре первого выражения соответствует трехуровневое дерево, в то время как структуре второго — лишь двухуровневое, и поэтому его можно выполнить гораздо быстрее на мультипроцессорной системе.

Цель многих систем новых поколений будет заключаться в том, чтобы производить вычисления в минимально возможное время, независимо от того, сколько процессоров придется использовать для выполнения этой работы. Поэтому новые вычислительные системы будут содержать специальные механизмы, которые будут преобразовывать последовательные алгоритмы в параллельные, разлагая их на шаги, которые можно было бы выполнять одновременно на многих процессорах системы.

На рис. 11.3 показано, каким образом благодаря коммутативности сложения можно преобразовать последовательную цепь операций, изображаемую трехуровневым деревом, в параллельный вариант, требующий всего лишь двух уровней. На рис. 11.4 показано, каким образом дистрибутивность умножения и сложения может быть использована для уменьшения числа уровней дерева с четырех до трех.

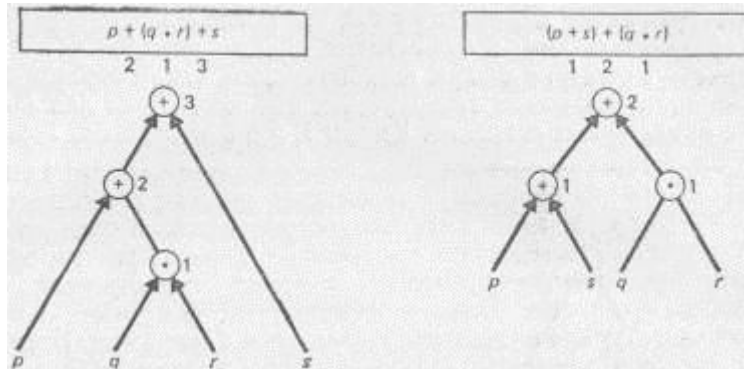


Рис. 11.3 Уменьшение высоты дерева за счет коммутативности.

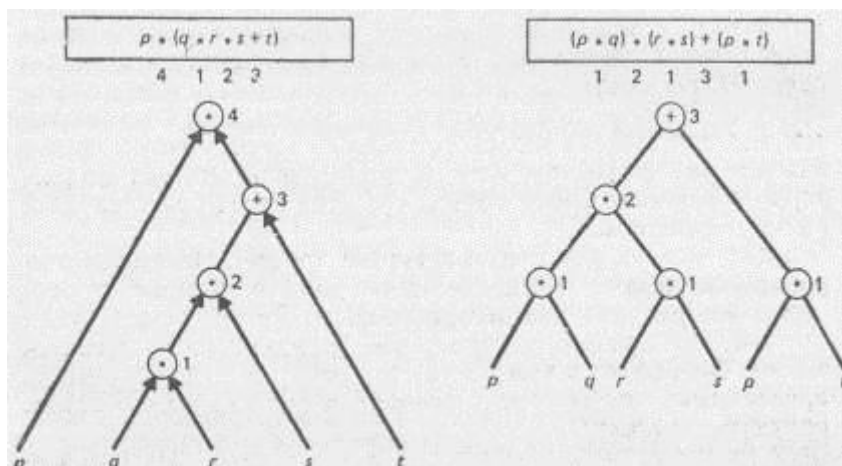


Рис. 11.4 Уменьшение высоты дерева за счет дистрибутивности.

Применение указанных методов компиляции с целью оптимизации программ для выполнения на мультимикропроцессорных системах не обходится бесплатно. Дело в том, что за уменьшение количества времени выполнения, затрачиваемого на данное вычисление, приходится платить увеличенными затратами времени и ресурсов в период компиляции. Такую взаимосвязь необходимо учитывать, тщательно оценивая необходимые затраты и возможные выгоды в каждом индивидуальном случае. Например, для производственного счета целесообразно добиваться минимального времени выполнения программ. Однако в условиях разработки, когда программа, возможно, будет выполняться всего один или два раза до внесения очередных изменений и повторной компиляции, затраты на оптимизацию могут значительно превысить получаемые выгоды.

Семантика циклов, выполняемых параллельно на ОКМД системах.

Не уверен, что это самое то, но близко...

Классы задач, которые можно эффективно векторизовать или распараллелить

Обработка массивов

Одномерные массивы

Данные задачи встречаются довольно часто. Если значения элементов массива определяются довольно сложным выражением, а вычислять их надо многократно, то векторизация или распараллеливание цикла для вычисления элементов массива может оказаться очень эффективным. В отдельный параграф мы вынесли решение систем дифференциальных уравнений, что по своей сути также является обработкой массивов функций, производных и т.д. Но на самом деле эффективными могут также быть вычисления сверток, сумм, функций от каждого элемента массива и т.п.

Конечно, не имеет смысла векторизовать или распараллеливать действия над короткими массивами кроме тех случаев, когда собственно вычисления каждого элемента занимают большое время.

Двумерные массивы

При исполнении вложенных циклов эффективно распараллеливаются самые внешние циклы, а векторизуются только внутренние. Однако практически все действия с матрицами (сложение, умножение, умножение на вектор, прямое произведение) могут быть выполнены быстро на супер-ЭВМ. Многие алгоритмы линейной алгебры (но не все) могут быть эффективно векторизованы и распараллелены. Некоторые библиотеки подпрограмм (например, LAPACK(?)) существуют для параллельных и векторных машин.

Совершенно неэффективно использовать векторные ЭВМ для работы с матрицами размерности 3x3. Но можно переписать алгоритм для одновременной обработки нескольких (к примеру 1000) матриц - обращение, поиск собственных чисел и т.д.

При увеличении размера матриц растет эффективность работы программы, но растет и размер требуемой памяти для хранения матриц. При работе на векторной машине с небольшим (128-512 Мбайт) объемом ОЗУ это может стать причиной общего снижения ее быстродействия из-за частого обращения к дискам при записи/чтении данных.

Вычисления в узлах сеток и решеток

Инженерные и научные задачи

Во многих областях знания встречаются задачи, которые сводятся к вычислению эволюции объектов, расположенных в дискретных точках и взаимодействующих с ближайшими соседями. Простейшей и, наверно, наиболее широко известной такой задачей является игра "жизнь".

Все игровое поле представляет двумерную сетку с квадратными ячейками. Каждая ячейка может быть в одном из двух состояний - пустая или живая. Если на каком-то шаге игры вокруг пустого поля существуют ровно три живых поля, то на следующем шаге игры в этом поле рождается жизнь. Если число живых полей вокруг пустого поля не равно трем, то в это поле остается пустым. Если вокруг живого поля существуют два или три других живых поля, то жизнь в в этом поле продолжается. Если число живых соседей отлично от двух или трех, то поле погибает (становится пустым). Начальная конфигурация выбирается произвольным образом. Игровое поле может иметь края или может быть свернуто в тор. Вот пример конфигурации:

о о о	Обозначения:
о х о	Оо - пустые
О Х О	Хх - живые
о х о	О - родится, о - останется
о о о	Х - останется, х - погибнет

Пример игры реализован в качестве заставки на дисплее в системе OpenWindows на ЭВМ фирмы SUN.

В игре "жизнь" реализуется простейшая модель "взаимодействия" пустых и живых ячеек с соседями, определяющая эволюцию всей системы. Но вот пример из физики, который по способу построения модели идентичен этой игре. Модель магнетиков Изинга представляет собой набор спинов (элементарных магнитов), расположенных в узлах решетки и взаимодействующих только с ближайшими соседями. Вычислительный интерес представляют трех- и четырехмерные модели. Двумерные Изинговские магнетики с анизотропными (зависящими от направления) дальними взаимодействиями (т.е. не только с ближайшими, но и через один узел) также нельзя исследовать аналитически, а только с помощью компьютерных экспериментов. Очевидно, что алгоритм построения эволюции Изинговских магнетиков будет во многом идентичен алгоритму игры.

Многие молекулярные модели сплошных сред в статистической физике также строятся по принципу узлов (или ячеек) в кубической решетке. Молекулы могут располагаться только в ячейках. Взаимодействия молекул в соседних ячейках определяют их ориентацию (для несферических молекул) и предпочтительность (вероятность) переходов в соседние пустые ячейки.

В задачах по динамике сплошных сред (аэро- и гидродинамика) также применяется метод разбиения всего пространства на маленькие объемы и моделирование перемещения этих объемов при движении твердых тел или при обтекании средой неподвижных предметов.

Инженерные расчеты по распределению нагрузок в сложных конструкциях также могут проводиться в рамках сеточных моделей. Вся конструкция можно рассматривать как сетку, в которой каждое соединение узлов имеет определенные силовые коэффициенты и предельные нагрузки на сжатие и растяжение. Таким образом можно моделировать деформацию конструкции и предельную нагрузочную способность при приложении силы в любой точке конструкции.

Видно, что класс задач, использующих сеточные методы довольно широк и крайне важен. В следующем параграфе мы рассмотрим алгоритмы решения таких задач.

Алгоритмы преобразования программ методом координат.

Одним из методов распараллеливания программ является метод координат, обеспечивающий векторизацию циклов для синхронных ЭВМ (ОКМД - SIMD архитектуры). Семантика синхронных циклов следующая: для каждой итерации цикла назначается процессор; вычисления должны производиться всеми процессорами (процессорными элементами) параллельно и синхронно. Операторы тела цикла выполняются по очереди, но каждый оператор выполняется всеми процессорами одновременно. При выполнении оператора присваивания сначала вычисляется правая часть, затем одновременно выполняется операция присваивания для всех элементов массива левой части.

Теоретическое обоснование классического метода координат предложено Лэмпортом. Метод применим для канонических ("чистых") циклов, тела которых состоят только из операторов присваивания, причем управляющая переменная заголовка цикла (параметр цикла) должна входить в индексные выражения операторов тела цикла линейно. Метод предназначен для синхронной векторизации одномерных циклов (многомерные циклы можно рассматривать по каждому параметру отдельно). Идея данного метода состоит в том, что для индексов, содержащих параметры цикла, рассматривается порядок следования, который и определяет очередность выборки/записи соответствующего элемента массива. Так как рассматриваются только линейные индексные выражения, порядок следования определяется значениями соответствующих алгебраических выражений. Строится граф характеристических множеств всех индексов, и при отсутствии в графе петель делается заключение о возможности векторизации всего цикла.

В некоторых случаях (устраняемая векторная зависимость) векторизация возможна лишь в случае переупорядочивания - перестановки операторов цикла или путем введения временной переменной (массива), куда будут копироваться элементы вектора для защиты перед их изменением для последующего использования в исходном виде. Итак, метод координат:

- позволяет определить возможность выполнения цикла в синхронном режиме;
- содержит алгоритмы преобразования тела цикла к синхронному виду.

Например, по Лэмпорту, цикл:

```
DO 24 I=2,M
DO 24 J=2,N
21  A(I,J) = B(I,J)+C(I)
22  C(I) = B(I-1,J)
23  B(I,J) = A(I+1,J) ** 2
24  CONTINUE
```

преобразуется в цикл:

```
DO 24 J=2,N
DO 24 SIM FOR ALL I {i:2<=i<=M}
C SIM - SI Multeneusly (одновременно)
TEMP(I) = A(I+1,J)
21  A(I,J) = B(I,J)+C(I)
23  B(I,J) = TEMP(I) ** 2
22  C(I) = B(I-1,J)
24  CONTINUE
```

Конструктивный алгоритм реализации метода координат.

Пусть, цикл одномерный, все операторы тела - операторы присваивания, в левых частях которых - переменные с индексом - параметром цикла. Индексы линейные, т.е. имеют вид $i+/-b$, где b - константа. Определяются отношения следования для всех пар одноименных переменных с индексами: генерируемых переменных (переменные в левой части операторов присваивания - f) и используемых переменных (входящих в формулы правых частей - q). Отношение следования для одной пары зависит от порядка обращения к одному и тому же элементу массива при выполнении цикла. Так, для $A(I) = A(I+1)$ отношение можно кодировать: $f < q$. Для многомерных циклов для всех пар (f, q) определяются направляющие вектора, элементы которых состоят из символов ($=$, $>$, $<$) в позиции соответствующего индекса, которая определяется порядком индексов в гнезде цикла. В примере Лемпорта (исходном) для A вектор имеет вид: $(<, =)$.

Для тела цикла, состоящего из одного оператора, отношения, и соответственно, семантика выполнения, могут быть: $< > =$ и их сочетание. Например, тело цикла: $A(I) = (A(I)+A(I+1))/2$ допускает векторное выполнение; оно невозможно при наличии в операторе хотя бы одного соотношения: $f > q$. Для тела цикла: $A(I) = A(I+C))/2$ при $C > 0$ возможно синхронное выполнение, при $C = 0$ кроме синхронного, возможно параллельное асинхронное; а при $C < 0$ параллельное выполнение невозможно.

Для многомерных (тесногнездовых) циклов такое исследование может проводиться для каждого параметра, а для распараллеливания выбирается наиболее оптимальный вариант. Для самого внутреннего цикла определение следования можно проводить, считая его одномерным. Для любого другого цикла, исследование можно проводить по такой же схеме, если семантика цикла допускает перестановку его на позицию самого внутреннего. Перестановка допустима, если для всех новых направляющих векторов крайне левый, отличный от $=$, элемент сохранил направление. Например, для двумерного цикла, наличие вектора $(<, >)$ делает перестановку некорректной. Так для цикла:

DO 99 J=2,M
DO 99 K=2,M
99 U(J,K) = (U(J+1,K)+U(J,K+1)+U(J-1,K)+U(J,K-1)) .25

направляющие вектора:

1 <U(J,K),U(J+1,K)> = (<,<=)
2 <U(J,K),U(J,K+1)> = (<,<=)
3 <U(J,K),U(J-1,K)> = (>,<=)
2 <U(J,K),U(J,K-1)> = (>,<=)

Отсюда:

- операторы циклов по I и J можно менять местами,
- часть оператора, включающая вхождения из векторов 1 и 2, можно вынести в отдельный оператор, который векторизуется как по I так и по J.

Для двух операторов тела цикла отношения следования переменных с индексами можно систематизировать и кодировать так (qi в общем случае, список - информационные поля q и f): A - независимые отношения B - (qi>qj), C - (qi<qj), K - (qi=qj) G - B+K,B+C,B+K+C, H - неоднозначные отношения. Тогда по координатам приводимой ниже таблице можно определить типы связей и метод выполнения цикла для этих операторов. Если объединить информационные поля двух операторов, то таблицу можно использовать для анализа третьего оператора и т.д.

* Таблица решений для метода координат

* Оператор 1 O1 = I1 Oi,Ii - список переменных с индексами

* Оператор 2 O2 = I2 -> - порядок анализа отношений

```

*-----
*           O2 -> I1
*           .
*           . A . B . K . C . G . H .
*           . НЕЗАВ. . O > I . O = I . O < I . O <=> I * .
* I2 -> O1 . . . . .
*-----

```

```

*
* A НЕЗАВ. . P . R . S . M . R(K,C). T .
*

```

```

*
* B I > O . R . R . M(B) . M(B) . R(K,C). T .
*

```

```

*
* K I = O . S . T . S . M . T . T .
*

```

```

*
* C I < O . M . T . M . M . T . T .
*

```

```

*
* G I <=> O. M(B) . T . M(B) . M(B) . T . T .
*

```

```

*
* H * . T . T . T . T . T . T .
*

```

* Кодировка отношений следования

* A - независимые отношения

* B - (qi>qj), C - (qi<qj), K - (qi=qj)

* G - B+K,B+C,B+K+C

* H - неоднозначные отношения

- * ТИПЫ СВЯЗЕЙ
- * P - независимые операторы
- * M - SIMD, M(EA) - SIMD с защитой EA
- * R - SIMD с реверсом, R(EA) - и с защитой EA
- * S - PAR (асинхронная параллельность)
- * T - запрет векторизации
- * Защита EA - копирование массивов EA перед

В общем случае, алгоритм векторизации методом координат с использованием данной таблицы следующий.

Обработка тела цикла начинается с анализа на возможность векторного выполнения каждого оператора тела в отдельности. Затем к первому оператору добавляется второй и проводится анализ на возможную их векторизацию. Пара получает тип, определяющий возможность векторизации ее компонент, информационные поля операторов сливаются, и на следующем шаге применения процедуры векторизации пара рассматривается как единый супероператор. Таким образом из всех операторов тела цикла образуется один супероператор и, если его тип есть T, то векторизация тела цикла невозможна. Для всех других типов производится обратный анализ полученного графа (супероператора). Если при этом связки имели тип R или R(EA), M(EA), то хотя они и допускают асинхронное параллельное выполнение, но необходимы преобразование тела цикла. Связки типа T дают операторы, векторизация которых невозможна. Интерпретация остальных типов связей очевидна. В процессе формирования супероператоров к связкам типа T могут применяться процедуры поиска минимальных границ области типа T и чистки области T.

Примеры векторизации

Исходные тела циклов Преобразованные тела циклов

$A(I) = B(I)$ $C(I) = A(I+1)$

$C(I) = A(I+1)$ $A(I) = B(I)$

$A(I) = B(I)$ $D(I) = A(I)$

$C(I) = A(I) + A(I+1)$ $A(I) = B(I)$
 $C(I) = A(I) + D(I+1)$

$A(I) = B(I) + B(I+1)$ $D(I) = B(I)$

$B(I) = A(I+1)$ $B(I) = A(I+1)$
 $A(I) = D(I) + B(I+1)$

$A(I) = B(I) + B(I-)$ Векторизация невозможна

$B(I) = A(I)$

Дополнение

Наибольшим внутренним параллелизмом, который можно использовать для векторизации программ, являются циклические участки, так как на них приходится основное время вычислений, и в случае распараллеливания вычисления в каждой ветви производятся по одному и тому же алгоритму. В методах распараллеливания циклов используется довольно сложный математический аппарат. Наиболее распространенными методами распараллеливания являются: метод параллелепипедов, метод координат, метод гиперплоскостей. Объектами векторизации в этих методах являются циклы типа DO в Фортране. При постановке в общем виде задачи распараллеливания циклов вводится понятие “пространство итераций” — n-мерное целочисленное пространство с

координатными осями I_1, K, I_n , соответствующими индексным переменным исходного цикла. Каждая итерация (повторение тела цикла при различных значениях индексов) представляет собой точку в этом пространстве и характеризуется значением вектора (i_1, K, i_n) , где ij — номер выполнения итерации на j -м уровне вложенности. Исходный тесногнездовой цикл имеет вид

```
DO 1 I1 = 1, M1
```

```
...
```

```
DO 1 In = 1, Mn
```

```
S(i1, ..., in)
```

```
1 CONTINUE
```

Шаг изменения цикла предполагается равным единице. Тело цикла $S(i_1, \dots, i_n)$ состоит из последовательности пронумерованных по порядку операторов. На тело цикла, как правило, накладываются некоторые ограничения, зависящие от типа ЭВМ и метода распараллеливания: все индексные выражения должны быть линейными функциями от параметров цикла; не допускаются условные и безусловные переходы из тела цикла за его пределы, а внутри цикла передача управления может осуществляться только вперед, т. е. операторы IF и GO TO нельзя применять для организации цикла. Не допускается использование в теле цикла операторов ввода-вывода и обращений к подпрограммам.

Для цикла (5.3) пространство итераций представляет собой

набор целочисленных векторов $I = \{ i_1, K, i_n \}$, $1 \leq i_j \leq M_j$. Распараллеливание цикла (5.3) заключается в разбиении этого пространства на подобласти, внутри которых все итерации могут выполняться одновременно и при этом будет сохраняться порядок информационных связей исходного цикла. Порции независимых итераций можно искать в виде n -мерных параллелепипедов, гиперплоскостей и т. д. в соответствии с методом распараллеливания.

Необходимое условие параллельного выполнения i -й и j -й итераций цикла записывается в виде

$$(OUT(i) \wedge IN(j)) \vee (IN(i) \wedge OUT(j)) \vee (OUT(i) \wedge OUT(j)) = \emptyset \quad (5.4)$$

Здесь $IN(i)$ и $OUT(i)$ — множества входных и выходных переменных i -й итерации. Два первых дизъюнктивных члена из (5.2) задают учет информационной зависимости между итерациями, т. е. параллельное выполнение может быть невозможным, если одна и та же переменная используется в теле цикла как входная и как выходная. Третий член учитывает конкуренционную зависимость, которая появляется в теле цикла в качестве выходной больше одного раза.

Отношение зависимости между итерациями можно представить в виде графа D . Вершины графа соответствуют итерациям цикла. Вершины i и j соединяет дуга, если они зависимы, и тогда $D(i, j) = 1$. Задачу распараллеливания цикла можно сформулировать как задачу разбиения графа D на несвязные подграфы D_i . Вершины, входящие в один подграф, должны быть независимыми и иметь смежные номера. Проиллюстрируем применение метода параллелепипедов на примере следующего цикла [10]:

```
DO 1 I = 2, 5
```

```
DO 1 J = 2, 4 (5.5)
```

```
1 X(I, J) = X(I - 1, J - 1) ** 2
```

Пространство итераций и информационные связи приведены на рис. 5.4, а. В данном случае вершины, лежащие вдоль диагонали прямоугольника, информационно связаны. Так, в итерации с номером (2, 2) генерируется переменная $X(2, 2)$, которая затем используется в качестве входной переменной в итерации (3, 3). Разбиение итераций на параллелепипеды можно осуществить вдоль осей I и J (см. рис. 5.4, а).

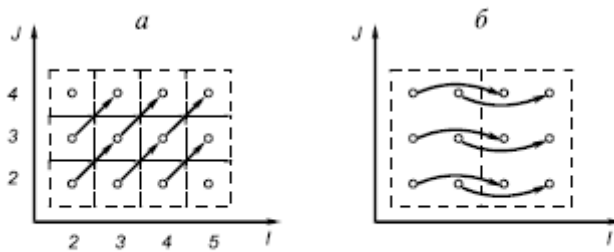


Рис.5.4. Разбиение итераций:
а — для цикла (5.5); б — для цикла (5.6)

Проиллюстрируем применение метода параллелепипедов для другого цикла:

```
DO 1 I = 2, 5
```

```
DO 1 J = 2, 4 (5.6)
```

```
1 X (I, J) = X (I - 2, J) ** 2
```

Пространство итераций и информационные связи приводятся на рис. 5.4, б. Разбиение на параллелепипеды можно провести вдоль оси I с шагом, равным двум, а вдоль оси J — с максимально возможным шагом (в данном случае равным трем). Таким образом, внутри каждого параллелепипеда оказалось по шесть независимых итераций, которые можно выполнить параллельно. Если в цикле есть информационные связи между операторами, запрещающие параллельное выполнение, то можно попытаться устранить эти связи, применив *метод координат*. Данный метод предполагает осуществление в теле цикла некоторых преобразований, а именно: перестановку операторов, введение дополнительных переменных и массивов. Подобные преобразования производятся таким образом, чтобы изменить направление или устранить

запрещающие связи между операторами в различных итерациях. При этом результат вычислений не должен изменяться. В качестве примера использования метода координат рассмотрим следующий цикл:

```
DO 4 I = 2, N
```

```
1 X (I) = Y (I) + Z (I)
```

```
2 Z (I) = Y (I - 1)
```

```
3 Y (I) = X (I + 1) ** 2
```

```
4 CONTINUE
```

Запрещающими связями в данном примере является связь между использованием переменной $Y(I-1)$ во втором и генерацией $Y(I)$ в третьем операторах, а также связь генерации $X(I)$ в первом

операторе с использованием генерации $X(I+1)$ в третьем операторе. Направление первой связи по Y можно изменить на противоположное перестановкой второго и третьего операторов. Вторую

связь по X можно устранить введением дополнительного массива T . Таким образом, эквивалентный исходному преобразованный цикл, который можно выполнить одновременно для всех значений индекса I , будет иметь такой вид:

```
DO 4 I = 2, N
```

```
R (I) = X (I + 1)
```

```
1 X (I) = Y (I) + Z (I)
```

```
3 Y (I) = R (I) ** 2
```

```
2 Z (I) = Y (I - 1)
```

```
4 CONTINUE
```

Необходимо отметить, что методом параллелепипедов и методом координат можно распараллеливать как одномерные, так и многомерные тесногнездовые циклы.

В рассматриваемых методах распараллеливания индексные выражения элементов массивов должны быть линейными функциями относительно параметров цикла, т. е. иметь вид $A \cdot I \pm B$, где A

и B — целые константы; I — параметр цикла DO . Необходимо также, чтобы в индексных выражениях использовались одноименные индексные переменные. Не допускается применение нелинейных индексов типа $X(I*J*K)$ или косвенной индексации, например $X(N(I))$.

Основным препятствием к распараллеливанию циклов является так называемое условие Рассела — использование в теле цикла простой неиндексированной переменной раньше, чем этой переменной присваивается в цикле некоторое значение, например:

```
DO 1 I = 1, N DO 1 I = 1, N
1 S = S + X (I) или X (I) = S * Y (I)
1 S = Z (I) ** 2
```

Распараллеливанию препятствуют и обратные информационные и конкуренционные связи между операторами тела цикла.

Рассмотрим пример:

```
DO 1 I = 1, N
1 X (I) = X (I - 1)
```

Итерации этого цикла связаны обратной информационной зависимостью с шагом, равным единице. Вообще говоря, конструкции вида

```
DO 1 I = 1, N
...
X (I) = X (I + K)
...
1 CONTINUE
```

где K — целочисленная переменная, векторизуются только в том случае, если знак переменной K совпадает со знаком инкремента цикла. В большинстве же случаев знак числа K не может быть определен на этапе компиляции, поэтому такие конструкции считаются не векторизуемыми. Из соотношения (5.4) следует, что если в теле цикла нет одноименных входных и выходных переменных, то подобный цикл векторизуется. Если в цикле генерация простой переменной встречается раньше, чем ее использование, то этот цикл также векторизуется:

```
DO 1 I = 1, N
S = X (I) + Y (I)
1 Z (I) = Z (I) - S
```

Информационная зависимость между итерациями появляется только в том случае, если в теле цикла имеются одноименные входные и выходные переменные с несовпадающими индексными выражениями, например $X(I)$ и $X(I - 1)$. Направление связи (прямое или обратное) зависит от номеров операторов, в которых используются эти переменные. Если совпадающие индексные выражения

и связи между итерациями отсутствуют, то такие циклы распараллеливаются:

```
DO 1 I = 1, N
X (I) = Y (I + 1) + Z (I)
1 Y (I + 1) = X (I) ** 2
```

В этом случае одноименные пары $X(I)$ и $Y(I + 1)$ имеют совпадающие индексы и не препятствуют векторизации. Рассмотрим часто встречающуюся на практике циклическую конструкцию, в которой переменная целого типа используется в качестве неявного параметра цикла:

```
J = 0
DO 1 I = 1, N
J = J + 2
1 X (I) = Y (I)
```

В этом случае в массив X заносятся четные элементы из массива Y . Роль второго параметра цикла играет переменная J , и хотя для J формально выполняется условие Рассела, этот цикл можно распараллелить. Иногда цикл, в котором выполняется условие Рассела, оказы-

вается вложенным в другой цикл, например:

```
DO 1 I = 1, N
S = 0
DO 2 J = 1, N
2 S = S + X (I, J)
Y (I) = S
1 CONTINUE
```

В вектор Y заносятся значения суммы элементов строк матрицы X . Внутренний цикл по J не распараллеливается (выполняется условие Рассела для переменной S), а внешний цикл по I можно распараллелить, так как переменная S в нем получает значение перед входом во внутренний цикл. Рассмотрим общую структуру ВК (рис. 5.5). Как правило, ВК состоит из синтаксического анализатора, распараллеливателя циклов, модуля оценки качества распараллеливания, генераторов программ на параллельном ЯВУ или в машинных кодах.

Схема преобразования программ методом гиперплоскостей.

Метод гиперплоскостей применим только к многомерным циклам. В пространстве итераций ищется прямая (плоскость), на которой возможно параллельное асинхронное выполнение тела цикла, причем в отличие от метода координат, эта прямая (плоскость) может иметь наклон по отношению к осям координат. Цикл вида:

```
DO 5 I = 2,N
DO 5 J = 2,M
5 A(I,J) = ( A(I-1,J) + A(I,J-1) ) * 0.5
```

методом координат не векторизуется. Действительно, при фиксированном значении переменной I ($I = i$) значение, вычисленное в точке (i,j) пространства итераций, зависит от результата вычислений в предыдущей точке $(i,j-1)$, так что параллельное выполнение тела цикла по переменной J невозможно. Аналогично нельзя проводить параллельные вычисления по переменной I .

Однако можно заметить, что результат будет также правильным, если вычисления проводить в следующем порядке:

- на 1-м шаге - в точке (2,2),
- на 2-м шаге - в точках (3,2) и (2,3),
- на 3-м шаге - в точках (4,2), (3,3) и (2,4),
- на 4-м шаге - в точках (5,2), (4,3), (3,4) и (2,5)

Вычисления в указанных точках для каждого шага, начиная со второго, можно проводить параллельно и асинхронно. Перечисленные кортежи точек лежат на параллельных прямых вида $I+J=K$, а именно: на первом шаге - на прямой $I+J=4$, на втором - $I+J=5$, на третьем шаге - $I+J=6$ и т.д., на последнем $((N-2)+(M-2)+1)$ - ом шаге - на прямой $I+J=M+N$.

В общем случае для n -мерного тесногнездового цикла ищется семейство параллельных гиперплоскостей в n -мерном пространстве итераций, таких что во всех точках каждой из этих гиперплоскостей возможно параллельное выполнение тела цикла.

Для приведенного примера множество точек, в которых возможно параллельное выполнение, является однопараметрическим (прямая) и определяется из решения уравнения $1I+J=K$. Цикл (5) может быть преобразован к виду:

```
DO 5 K = 4,M+N
Tн = MMAX(2,K-N)
Тк = MMIN(M,K-2)
DO 5 T = Тн,Тк 1: PAR
I = T
```

$$J = K - T$$

$$5 \ A(I,J) = (A(I-1,J) + A(I,J-1)) * 0.5$$

Функция $\text{MMAX}(X,Y)$ выбирает ближайшее целое, большее или равное максимуму из чисел X и Y , а функция $\text{MMIN}(X,Y)$ - ближайшее целое, меньшее или равное минимуму из X и Y .

Внутренний цикл по переменной T может выполняться параллельно для всех значений T . Границы изменения переменной цикла T меняются при переходе с одной прямой (гиперплоскости) на другую, поэтому их необходимо перевычислять во внешнем цикле. Число итераций внутреннего цикла, то есть потенциальная длина векторной операции, меняется с изменением K . Для приведенного примера диапазон изменения T сначала возрастает, а потом убывает, причем для начального и конечного значения K он равен единице. В некоторых случаях для отдельных значений K накладные расходы на организацию векторного вычисления могут превысить эффект ускорения от векторного выполнения.

Вопрос оценки целесообразности проведения векторизации данным методом должен рассматриваться для каждого конкретного случая отдельно.

Метод параллелепипедов.

Идея метода заключается в выявлении зависимых итераций цикла и объединении их в последовательности - ветви, которые могут быть выполнены независимо друг от друга. При этом в пространстве итераций определяются области (параллелепипеды), все точки которых принадлежат разным ветвям. Задача максимального распараллеливания заключается в поиске параллелепипеда наибольшего объема; тогда исходный цикл выполняется наибольшим числом параллельных ветвей, каждая из которых представляет собой исходный цикл, но с другим шагом изменения индекса.

Для исходного цикла:

```
DO 7 I = 1,7
DO 7 J = 1,3
7 X(I,J) = X(I-2,J-1)
```

параллельное представление в виде:

```
DO 7 (K,L) = (1,1) (P1,P2) 1: PAR
DO 7 I = K,7,P1
DO 7 J = L,3,P2
7 X(I,J) = X(I-2,J-1)
```

допускается для различных разбиений пространства итераций: пара $(P1,P2)$ может иметь, например, значения $(2,1)$, $(2,3)$ или $(7,1)$. Таким образом, исходный цикл (7) преобразуется в последовательность параллельных ветвей, имеющих циклический вид.

В общем виде задача, рассматриваемая методом параллелепипедов, для одномерных циклов состоит в определении возможности представления цикла:

```
DO L I = 1,r
L T(I)
```

(где $T(i)$ - тело цикла) в виде следующей языковой конструкции:

```
DO L K = 1,p 1: PAR
DO L I = K,r,p
```

L T(I)

Оценить возможность параллельного выполнения цикла:

```
DO i = 2,N  
  A(i) = (B(i) + C(i))/A(i+CONST)  
ENDDO
```

Параллельное выполнение цикла вида

DO i=2,N A(I)=(B(i)+C(i))/A(i+const) ENDDO.

Если const = 0, то все итерации цикла независимы и такой цикл может быть выполнен на любой многопроцессорной ЭВМ, включая Иллиак-4. (каждый виток цикла выполняется на отдельном процессоре)

Если const > 0 (пусть =1) то при параллельное выполнение цикла на ЭВМ класса МКМД без дополнительных мер по синхронизации работы процессоров невозможно. Например, пусть N=3 и два процессора вычисляют параллельно эти итерации, тогда, первый процессор вычисляет A2 по B2, C2, A3, а второй, A3 по B2, C2, A4. При отсутствии синхронизации может случиться ситуация, при которой второй процессор завершит свою работу до начала работы первого. Тогда первый процессор для вычислений будет использовать A3, которое обновил второй процессор, что неверно, ибо здесь нужно “старое” значение A3. Однако, этот цикл выполняется параллельно на ЭВМ ОКМД (SIMD) так как там этот цикл может быть выполнен такими командами:

1. Считать B_i в сумматоры каждого из n АЛУ.
2. Сложить C_i со своим содержимом сумматора.
3. Разделить содержимое каждого i-сумматора на A_{i+1} .
4. Записать содержимое i- сумматоров в A_i .

Из за того, что выборка из памяти и запись в память производится синхронно (одновременно), то работа цикла – корректна.

Если const < 0 то параллельное выполнение цикла невозможно, ибо для выполнения очередной итерации цикла необходимы результаты работы предыдущей (рекурсия). Однако известны приемы преобразования такого рода циклов к виду, допускающие параллельное выполнение.

Языки параллельного программирования

Стандарты OpenMP.

1. Введение. Что такое OpenMP?

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP, ccNUMA, etc.) в модели общей памяти (shared memory model). В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды.

Кто разрабатывает стандарт?

Разработкой стандарта занимается организация OpenMP ARB (ARchitecture Board), в которую вошли представители крупнейших компаний - разработчиков SMP-архитектур и программного обеспечения. Спецификации для языков Fortran и C/C++ появились соответственно в октябре 1997 года и октябре 1998 года. Открыт список рассылки для публичного обсуждения OpenMP (omp@openmp.org).

Зачем нужен новый стандарт?

До появления OpenMP не было подходящего стандарта для эффективного программирования на SMP-системах.

Наиболее гибким, переносимым и общепринятым интерфейсом параллельного программирования является MPI (интерфейс передачи сообщений). Однако модель передачи сообщений 1) недостаточно эффективна на SMP-системах; 2) относительно сложна в освоении, так как требует мышления в "невыхислительных" терминах.

POSIX-интерфейс для организации нитей (Pthreads) поддерживается широко (практически на всех UNIX-системах), однако по многим причинам не подходит для практического параллельного программирования:

нет поддержки Fortran-a,

слишком низкий уровень,

нет поддержки параллелизма по данным,

механизм нитей изначально разрабатывался не для целей организации параллелизма.

OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads (или аналогичными библиотеками нитей).

Многие поставщики SMP-архитектур (Sun, HP, SGI) в своих компиляторах поддерживают спецдирективы для распараллеливания циклов. Однако эти наборы директив, как правило, 1) весьма ограничены; 2) несовместимы между собой; в результате чего разработчикам приходится распараллеливать приложение отдельно для каждой платформы. OpenMP является во многом обобщением и расширением упомянутых наборов директив.

Какие преимущества OpenMP дает разработчику?

1. За счет идеи "инкрементального распараллеливания" OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы.

2. При этом, OpenMP - достаточно гибкий механизм, предоставляющий разработчику большие возможности контроля над поведением параллельного приложения.

3. Предполагается, что OpenMP-программа на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова процедур OpenMP могут быть подставлены заглушки (stubs), текст которых приведен в спецификациях.

4. Одним из достоинств OpenMP его разработчики считают поддержку так называемых "orphan" (оторванных) директив, то есть директивы синхронизации и распределения работы могут не входить непосредственно в лексический контекст параллельной области.

Как это работает?

Согласно терминологии POSIX threads, любой UNIX-процесс состоит из нескольких нитей управления, которые имеют общее адресное пространство, но разные потоки команд и отдельные стеки. В простейшем случае, процесс состоит из одной нити. Нити иногда называют также потоками, легковесными процессами, LWP (light-weight processes).

В OpenMP используется терминология и модель программирования, близкая к Pthreads (динамически порождаемые нити, общие и разделяемые данные, механизм "замков" для

синхронизации). Предполагается наиболее вероятным, что OpenMP будет реализован на базе Pthreads.

Как это выглядит?

Простой пример: вычисление числа "Пи". В последовательную программу вставлены две строчки, и она распараллелена!

```
program compute_pi
  parameter (n = 1000)
  integer i
  double precision w,x,sum,pi,f,a
  f(a) = 4.d0/(1.d0+a*a)
  w = 1.0d0/n
  sum = 0.0d0;
!$OMP PARALLEL DO PRIVATE(x), SHARED(w)
!$OMP& REDUCTION(+:sum)
```

```
  do i=1,n
    x = w*(i-0.5d0)
    sum = sum + f(x)
  enddo
```

```
  pi = w*sum
  print *, 'pi = ',pi
  stop
end
```

2. Директивы

Директивы OpenMP с точки зрения Фортрана являются комментариями и начинаются с комбинации символов "!\$OMP". Директивы можно разделить на 3 категории: определение параллельной секции, разделение работы, синхронизация. Каждая директива может иметь несколько дополнительных атрибутов - клауз. Отдельно специфицируются клаузы для назначения классов переменных, которые могут быть атрибутами различных директив.

Порождение нитей

PARALLEL ... END PARALLEL

Определяет параллельную область программы. При входе в эту область порождаются новые (N-1), образуется "команда" из N нитей, а порождающая нить получает номер 0 и становится основной нитью команды (т.н. "master thread"). При выходе из параллельной области основная нить дожидается завершения остальных нитей, и продолжает выполнение в одном экземпляре. Предполагается, что в SMP-системе нити будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы).

Каким образом между порожденными нитями распределяется работа - определяется директивами DO, SECTIONS и SINGLE. Возможно также явное управление распределением работы (а-ля MPI) с помощью функций, возвращающих номер текущей нити и общее число нитей. По умолчанию (вне этих директив), код внутри PARALLEL выполняется всеми нитями одинаково.

Вместе с PARALLEL может использоваться клауза IF(условие) - й параллельная работа инициируется только при выполнении указанного в ней условия.

Параллельные области могут динамически вложенными. По умолчанию (если вложенный параллелизм не разрешен явно), внутренняя параллельная область выполняется одной нитью.

Разделение работы (work-sharing constructs)

Параллельные циклы

DO ... [ENDDO]

Определяет параллельный цикл.

Клауза SCHEDULE определяет способ распределения итераций по нитям:

STATIC,m - статически, блоками по m итераций

DYNAMIC,m - динамически, блоками по m (каждая нить берет на выполнение первый еще невзятый блок итераций)

GUIDED,m - размер блока итераций уменьшается экспоненциально до величины m

RUNTIME - выбирается во время выполнения .

По умолчанию, в конце цикла происходит неявная синхронизация; эту синхронизацию можно запретить с помощью ENDDO NOWAIT.

Параллельные секции

SECTIONS ... END SECTIONS

Не-итеративная параллельная конструкция. Определяет набор независимых секций кода (т.н., "конечный" параллелизм). Секции отделяются друг от друга директивой SECTION.

Примечание. Если внутри PARALLEL содержится только одна конструкция DO или только одна конструкция SECTIONS, то можно использовать укороченную запись: PARALLEL DO или PARALLEL SECTIONS.

Исполнение одной нитью

SINGLE ... END SINGLE

Определяет блок кода, который будет исполнен только одной нитью (первой, которая дойдет до этого блока).

Явное управление распределением работы

С помощью функций OMP_GET_THREAD_NUM() и OMP_GET_NUM_THREADS нить может узнать свой номер и общее число нитей, а затем выполнять свою часть работы в зависимости от своего номера (этот подход широко используется в программах на базе интерфейса MPI).

Директивы синхронизации

MASTER ... END MASTER

Определяет блок кода, который будет выполнен только master-ом (нулевой нитью).

CRITICAL ... END CRITICAL

Определяет критическую секцию, то есть блок кода, который не должен выполняться одновременно двумя или более нитями

BARRIER

Определяет точку барьерной синхронизации, в которой каждая нить дожидается всех остальных.

ATOMIC

Определяет переменную в левой части оператора "атомарного" присваивания, которая должна корректно обновляться несколькими нитями.

ORDERED ... END ORDERED

Определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле. Может использоваться для упорядочения вывода от параллельных нитей.

FLUSH

Явно определяет точку, в которой реализация должна обеспечить одинаковый вид памяти для всех нитей. Неявно FLUSH присутствует в следующих директивах: BARRIER, CRITICAL, END CRITICAL, END DO, END PARALLEL, END SECTIONS, END SINGLE, ORDERED, END ORDERED.

В целях синхронизации можно также пользоваться механизмом замков (locks).

3. Классы переменных

В OpenMP переменные в параллельных областях программы разделяются на два основных класса:

SHARED (общие; под именем A все нити видят одну переменную) и
PRIVATE (приватные; под именем A каждая нить видит свою переменную).

Отдельные правила определяют поведение переменных при входе и выходе из параллельной области или параллельного цикла: REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYIN.

По умолчанию, все COMMON-блоки, а также переменные, порожденные вне параллельной области, при входе в эту область остаются общими (SHARED). Исключение составляют переменные - счетчики итераций в цикле, по очевидным причинам. Переменные, порожденные внутри параллельной области, являются приватными (PRIVATE). Явно назначить класс переменных по умолчанию можно с помощью клаузы DEFAULT.

SHARED

Применяется к переменным, которые необходимо сделать общими.

PRIVATE

Применяется к переменным, которые необходимо сделать приватными. При входе в параллельную область для каждой нити создается отдельный экземпляр переменной, который не имеет никакой связи с оригинальной переменной вне параллельной области.

THREADPRIVATE

Применяется к COMMON-блокам, которые необходимо сделать приватными. Директива должна применяться после каждой декларации COMMON-блока.

FIRSTPRIVATE

Приватные копии переменной при входе в параллельную область инициализируются значением оригинальной переменной.

LASTPRIVATE

По окончании параллельно цикла или блока параллельных секций, нить, которая выполнила последнюю итерацию цикла или последнюю секцию блока, обновляет значение оригинальной переменной.

REDUCTION(+:A)

Обозначает переменную, с которой в цикле производится reduction-операция (например, суммирование). При выходе из цикла, данная операция производится над копиями переменной во всех нитях, и результат присваивается оригинальной переменной.

COPYIN

Применяется к COMMON-блокам, которые помечены как THREADPRIVATE. При входе в параллельную область приватные копии этих данных инициализируются оригинальными значениями.

4. Runtime-процедуры и переменные среды

В целях создания переносимой среды запуска параллельных программ, в OpenMP определен ряд переменных среды, контролирующих поведение приложения.

В OpenMP предусмотрен также набор библиотечных процедур, которые позволяют: во время исполнения контролировать и запрашивать различные параметры, определяющие поведение приложения (такие как число нитей и процессоров, возможность вложенного параллелизма); процедуры назначения параметров имеют приоритет над соответствующими переменными среды.

использовать синхронизацию на базе замков (locks).

Язык Фортран-DVM.

1.2. DVM—подход к разработке параллельных программ

DVM-система предоставляет единый комплекс средств для разработки параллельных программ научно-технических расчетов на языках Си и Фортран 77.

Модель параллелизма DVM. Модель параллелизма DVM базируется на модели параллелизма по данным. Аббревиатура DVM отражает два названия модели: распределенная виртуальная память (Distributed Virtual Memory) и распределенная виртуальная машина (Distributed Virtual Mashine). Эти два названия указывают на адаптацию модели DVM как для систем с общей памятью, так и для систем с распределенной памятью. Высокоуровневая модель DVM позволяет не только снизить трудоемкость разработки параллельных программ, но и определяет единую формализованную базу для систем поддержки выполнения, отладки, оценки и прогноза производительности.

Языки и компиляторы. В отличие от стандарта HPF в системе DVM не ставилась задача полной автоматизации распараллеливания вычислений и синхронизации работы с общими данными. С помощью высокоуровневых спецификаций программист полностью управляет эффективностью выполнения параллельной программы. С другой стороны, при проектировании и развитии языка Fortran DVM отслеживалась совместимость с подмножеством стандартов HPF1 и HPF2.

Единая модель параллелизма встроена в языки Си и Фортран 77 на базе конструкций, которые “невидимы” для стандартных компиляторов, что позволяет иметь один экземпляр программы для последовательного и параллельного выполнения. Компиляторы с языков C-DVM и Fortran DVM переводят DVM-программу в программу на соответствующем языке (Си или Фортран 77) с вызовами функций системы поддержки параллельного выполнения. Поэтому единственным требованием к параллельной системе является наличие компиляторов с языков Си и Фортран 77.

Технология выполнения и отладки. Единая модель параллелизма позволяет иметь для двух языков единую систему поддержки выполнения и, как следствие, единую систему отладки, анализа и прогноза производительности. Выполнение и отладка DVM-программ может осуществляться в следующих режимах:

Последовательное выполнение и отладка средствами стандартных компиляторов с языков Си и Фортран 77.

Псевдо-параллельное выполнение на рабочей станции (среда WINDOWS и UNIX).

Параллельное выполнение на параллельной ЭВМ.

При псевдо-параллельном и параллельном выполнении возможны следующие режимы отладки:

автоматическая проверка правильности директив параллелизма;

трассировка и сравнение результатов параллельного и последовательного выполнения;

накопление и визуализация трассировки данных;

накопление данных о производительности и прогноз производительности параллельного выполнения.

2. Обзор языка

2.1. Модель программирования и модель параллелизма

Язык Fortran DVM (FDVM) представляет собой язык Фортран 77 [5], расширенный спецификациями параллелизма. Эти спецификации оформлены в виде специальных комментариев, которые называются директивами. Директивы FDVM можно условно разделить на три подмножества:

Распределение данных (разделы 2, 3, 4, 8, 9)

Распределение вычислений (разделы 5, 7)

Спецификация удаленных данных (раздел 6)

Модель параллелизма FDVM базируется на специальной форме параллелизма по данным: одна программа – множество потоков данных (ОПМД). В этой модели одна и та же программа выполняется на каждом процессоре, но каждый процессор выполняет свое подмножество операторов в соответствии с распределением данных.

В модели FDVM пользователь вначале определяет многомерный массив виртуальных процессоров, на секции которого будут распределяться данные и вычисления. При этом секция может варьироваться от полного массива процессоров до отдельного процессора.

На следующем этапе определяются массивы, которые должны быть распределены между процессорами (распределенные данные). Эти массивы специфицируются директивами отображения данных (раздел 4). Остальные переменные (распределяемые по умолчанию) отображаются по одному экземпляру на каждый процессор (размноженные данные). Размноженная переменная должна иметь одно и то же значение на каждом процессоре за исключением переменных в параллельных конструкциях (см. раздел 5.1.3, 5.1.4 и 7.5).

Модель FDVM определяет два уровня параллелизма:

параллелизм по данным на секции массива процессоров;

параллелизм задач – независимые вычисления на секциях массива процессоров.

Параллелизм по данным реализуется распределением витков тесно-гнездового цикла между процессорами (раздел 5). При этом каждый виток такого параллельного цикла полностью выполняется на одном процессоре. Операторы вне параллельного цикла выполняются по правилу собственных вычислений (раздел 5.2).

Параллелизм задач реализуется распределением данных и независимых вычислений на секции массива процессоров (раздел 7).

При вычислении значения собственной переменной процессору могут потребоваться как значения собственных переменных, так и значения несобственных (удаленных) переменных. Все удаленные переменные должны быть указаны в директивах доступа к удаленным данным (раздел 6).

Язык Sisal.

Цели создания языка SISAL:

- создание универсального функционального языка;
- разработка техники оптимизации для высокоэффективных параллельных программ;
- достижение эффективности исполнения, сравнимой с императивными языками типа Fortran и C;
- внедрение функционального стиля программирования для больших научных программ.

Название языка расшифровывается как "Streams and Iterations in a Single Assignment Language", сам он представляет собой дальнейшее развития языка VAL, известного в середине 70-х годов. Среди целей разработки языка SISAL следует отметить наиболее характерные, связанные с функциональным стилем программирования:

Эти цели создателей языка SISAL подтверждают, что функциональные языки способствуют разработке корректных параллельных программ. Одна из причин заключается в том, что функциональные программы свободны от побочных эффектов и ошибок, зависящих от реального времени. Это существенно снижает сложность отладки. Результаты переносимы на разные архитектуры, операционные системы или инструментальное окружение. В отличие от императивных языков, функциональные языки уменьшают нагрузку на кодирование, в них проще анализировать информационные потоки и схемы управления. Легко создать функциональную программу, которая является безусловно параллельной, если ее можно писать, освободившись от большинства сложностей параллельного программирования, связанных с выражением частичных отношений порядка между отдельными операциями уровня аппаратуры. Пользователь Sisal-а получает возможность сконцентрироваться на конструировании алгоритмов и разработке программ в терминах крупноблочных и регулярно организованных построений, опираясь на естественный параллелизм уровня постановки задачи.

Начнем с примера программы:

1. Вычисление числа π (пи).

For % инициирование цикла

 Approx := 1.0;

 Sign := 1.0;

 Denom := 1.0;

 i := 1

while i <= Cycles do % предусловие завершения цикла

 Sign := -Sign; % однократные

 Denom := Denom + 2.0; % присваивания

 Approx := Approx + Sign / Denom; % образуют

 i := i + 1 % тело цикла

```

returns Approx * 4.0
% выбор и вычисление результата цикла
end for

```

2. Это выражение также вычисляет число π (пи).

```

for i in [1..Cycles/2] do
    % пространство параллельно
    % исполнимых итераций

    val := 1.0/real(4*i-3) — 1.0/real(4*i-1);
    % тело цикла, для каждого i
    % исполняемое независимо

returns sum( val ) % выбор и свертка результатов
    % всех итераций цикла

end for * 4.0 % вычисление результата
    % выражения

```

Это выражение вычисляет сумму всех вычисленных значений val и умножает результат на 4.0.

3, 4. В for-выражениях операции dot и cross могут порождать пары индексов при формировании пространства итерирования:

```

for i in [1..2] dot j in [3..4] do
    % для пар индексов [1,3] и
    % [2,4]
    returns product (i+j)
    % произведение сумм
end for % = 24
for i in [1..2] cross j in [3..4] do
    % для пар [1,3], [1,4], [2,3]
    % и [2,4]
    returns product (i+j)
    % произведение сумм
end for % = 600

```

5. Итеративное for-выражение с обменом данными между итерациями:

```

for
    I := 1
while I < S do
    K := I;
    I := old I + 2;
    % значение из предыдущей итерации
    J := K + I;
returns product(I+J)
end for

```

Как это свойственно языкам функционального программирования, Sisal язык математически правильный — функции отображают аргументы в результаты без побочных эффектов, и программа строится как выражение, вырабатывающее значение. Наиболее

интересна форма параллельного цикла. Она включает в себя три части: генератор пространства итераций, тело цикла и формирователь возвращаемых значений.

SISAL-программа представляет собой набор функций, допускающих частичное применение, т.е. вычисление при неполном наборе аргументов. В таком случае по исходному определению функции строятся его проекции, зависящие от остальных аргументов, что позволяет оперативно использовать эффекты смешанных вычислений и определять специальные оптимизации программ, связанные с разнообразием используемых конструкций и реализационных вариантов параллельных вычислений.

```
function Sum (N); % Сумма квадратов result (+ ( sqw (1 .. N)));
```

Обычно рассматривают оптимизации, обеспечивающие устранение неиспользуемого кода, чистку циклов, слияние общих подвыражений, перенос участков повторяемости для обеспечения однородности распараллеливаемых ветвей, раскрутку или разбиение цикла, втягивание константных вычислений, уменьшение силы операций, удаление копий агрегатных конструкций и др.

Система программирования Норма.

Язык Норма является специализированным языком и предназначен для спецификации численных методов решения задач математической физики. Изначально он был ориентирован на решение задач математической физики разностными методами, однако, как показала практика, может быть использован для решения более широкого класса вычислительных задач.

Первоначально термин Норма расшифровывался следующим образом: Непроцедурное Описание Разностных Моделей Алгоритмов. Сейчас мы считаем уместной и другую трактовку - это Нормальный уровень общения прикладного математика с ЭВМ: расчетные формулы, полученные им в процессе решения прикладной задачи, почти непосредственно используются для ввода в вычислительную систему и проведения счета.

В записи на Норме не требуется никакой информации о порядке счета, способах организации вычислительных (циклических) процессов. Порядок предложений языка может быть произвольным -- информационные взаимосвязи выявляются и учитываются транслятором при организации вычислительного процесса. Программа на языке Норма может рассматриваться как описание запроса на вычисление, а реализация этого запроса с учетом архитектуры ЭВМ и возможностей выходного языка - то есть синтез выходной программы - возлагается на транслятор.

Выбор уровня языка Норма определяет характерную его черту - это язык с однократным присваиванием, то есть каждая переменная может принимать значение только один раз. Такие понятия, как память, побочный эффект, оператор присваивания, управляющие операторы в языке Норма отсутствуют по определению.

Эти свойства, и некоторые другие ограничения (в первую очередь, на вид индексных выражений и способы описания индексных пространств), позволяют строго обосновать разрешимость задачи синтеза выходной программы [2,3], так как в достаточно общей постановке решение этой задачи приводит к значительным математическим трудностям - она может оказаться NP-полной либо вообще неразрешимой. С другой стороны, исследования, связанные с разработкой и применением языка Норма показывают, что имеющиеся ограничения приемлемы с практической точки зрения [5].

Программа на Норме состоит из одного или нескольких разделов. Разделы могут быть трех видов - главный раздел, простой раздел и раздел-функция, вид раздела определяется ключевыми словами MAIN PART, PART, FUNCTION соответственно.

Разделы могут вызывать друг друга по имени и передавать данные при помощи механизма формальных и фактических параметров, или через внешние файлы при помощи описаний INPUT и OUTPUT. Для каждого раздела справедливо правило локализации: имена, описанные в разделе, локализованы в этом разделе; понятие глобальных переменных в языке отсутствует.

Главный раздел обязательно должен присутствовать в программе на Норме и быть единственным; формальных параметров он не имеет. Вызовы главного раздела, а также рекурсивные вызовы разделов запрещены. В заголовке раздела указывается имя раздела и список формальных параметров. Формальные параметры должны быть описаны в теле раздела при помощи declaration-of-scalar-variables, declaration-of-variables-on-domains или declaration-of-external

Параметры-величины, указанные до ключевого слова RESULT в списке формальных параметров, являются исходными данными для вычислений, описываемых в разделе; параметры, перечисленные после - являются результатами вычислений. Один и тот же параметр не может быть одновременно исходным и результатом: это приводит к переприсваиванию значений переменным (повторному присваиванию), что запрещено в Норме. В разделе-функции ключевое слово RESULT не используется: результат вычисления функции связывается с именем и типом функции.

В теле раздела могут быть заданы описания, операторы и итерации (порядок их расположения, вообще говоря, произвольный - возможные ограничения определяются при описании входного языка транслятора).

Система Норма состоит из следующих компонентов:

- Декларативный язык Норма
- Компилятор с языка Норма
- Конфигуратор программ на языке Норма
- Отладчик программ на языке Норма
- Пользовательский Windows-интерфейс

Особенности выполнения арифметических выражения на ЭВМ. Распараллеливание вычислительных алгоритмов. Метод распараллеливания алгоритма общей рекурсии 1-го порядка.

Распараллеливание алгоритмов сложения методом редукции

Параллельно суммирование последовательности n чисел можно произвести так: на первом этапе складываются соседние числа. Полученные суммы также складываются попарно, и т.д. Для $n = 2^q$ алгоритм состоит из $q = \log n$ этапов, на первом этапе выполняются $n/2$ сложений (степень параллельности этапа $n/2$), на втором - $n/4$ и т. д. Такой алгоритм называется сложение методом сдваивания, он имеет различную степень параллелизма на разных этапах вычислений. Граф, описывающий последовательность операций сложения, граф сдваивания (по Д. Ортега "fan-in graph.") представляет собой двоичное дерево, соответственно, выполняемые операции можно называть операциями на дереве.

Способ реализации процедуры суммирования данным методом зависит от архитектуры вычислительной системы. При наличии $n/2$ процессоров эту работу можно выполнить так: на первом этапе одновременно получить суммы четных/нечетных соседних элементов последовательности A_i , т.е. (A_1+A_2) , (A_3+A_4) , ..., $(A_{n-1}+A_n)$; затем такая процедура повторяется для суммирования полученных частных сумм и так далее. Если $n = 2^{**}q$, то через $q = \log_2 n$ шагов получается искомая сумма. Однако, потери на синхронизацию вычислений, на пересылки частных сумм могут оказаться сравнимы с временем вычисления суммы двух чисел в каждом процессоре.

Поэтому, с учетом особенностей характеристик вычислительной системы, дерево вычислений может быть преобразовано, например, с целью увеличения числа операций, выполняемых в узлах, повышения "зернистости" алгоритма.

Алгоритм сдваивания реализуются также в блоках оптимизации компиляторов последовательных ЭВМ для полной загрузки конвейерных вычислителей. Так алгоритм оптимизация "балансировка дерева вычислений" (tree-height reduction or balancing) будет трактовать вычисление суммы вещественных чисел: $A+B+C+D+E+F+G+H$, как последовательность операций: $((((A+B)+(C+D))+((E+F)+(G+H))))$.

Рекурсия - последовательность вычислений, при котором значение самого последнего терма в последовательности зависит от одного или несколько ранее вычисленных термов. Пусть группа вычислений может производиться параллельно, используя результаты вычислений, выполненных на предыдущих этапах (полученных в виде начальных данных). Тогда, каждая группа вычислений называется "ярусом" параллельной формы, число групп - "высотой", максимальное число операций в группе "шириной" параллельной формы. Один и тот же алгоритм может иметь несколько представлений в виде параллельных форм, различающиеся как шириной, так и высотой. Редукционный алгоритм сдваивания для суммирования чисел с получением частных сумм может иметь вид:

Данные	A1	A2	A3	A4	A5	A6	A7	A8
Ярус 1	A1+A2		A3+A4		A5+A6		A7+A8	
Ярус 2	A12+A3		A12+A34		A56+A7		A56+A78	
Ярус 3	A1234+A5		A1234+A56		A1234+A567		A1234+A5678	

Высота параллельной формы равна трем, ширина - четырем, причем загрузка вычислителей (четыре) полная. В данном алгоритме производится вычисления пяти 'лишних' чисел по сравнению с последовательным алгоритмом.

Каскадное суммирование

Примером параллельных алгоритмов, ориентированных на векторные вычислители, может служить метод вычисления каскадных сумм (алгоритм рекурсивного удвоения) для распараллеливания операций суммирования. Пусть необходимо просуммировать n чисел с сохранением промежуточных сумм: $S_i = S_{i-1} + A_i$ $i = 2, \dots, n$, $S_1 = A_1$. Исходный вектор A поэлементно складывается с вектором A_s , полученный из исходного со сдвигом на один элемент и заполнением позиции элемента A_0 нулем. Для вектора результата процедура повторяется, но сдвиг - на 2 позиции. Если $n = 2^{**}k$, то через k операций получается вектор результата.

Для $i=8$:

A1	0	A1	0	A1	0	A1
A2	A1	A12	0	A12	0	A12
A3	A2	A23	A1	A123	0	A123
A4 + A3 = A34 + A12 = A1234 + 0 = A1234						
A5	A4	A45	A23	A2345	A1	A12345

A6	A5	A56	A34	A3456	A12	A123456
A7	A6	A67	A45	A4567	A123	A1234567
A8	A7	A78	A56	A5678	A1234	A12345678

Алгебра данного метода может быть записана в виде вычисления (возможно, параллельного) частных сумм вида: $S_i = A_i$, где $A_i = A(l-1)i + A(l-1)(i-2^{**}(l-1))$, $A_0i = A_i$ для $i = 1, 2, \dots, n$.

Вычисления проводятся $l = 0, 1, \dots, \log_2 n$ раз, причем, если у A_i индекс i выходит из интервала $1 \leq i \leq n$ то он принимается равным нулю.

Хокни предлагает элегантную векторную форму записи алгоритма каскадного суммирования массива $D(n)$:

```

X = D
DO L = 1, LOG2(N)
  X = X + SHIFTR(X, 2**(L-1))
ENDDO

```

Результат векторной функции $SHIFTR(A, l)$ есть массив (вектор), полученный из A , элементы которого сдвинуты на L позиций вправо, а L левых элементов заполнены нулями. Практическая реализация алгоритма может исключить излишние операции сложения с нулем, однако, и после этого, по сравнению с последовательным алгоритмом, данный - требует лишние операции.

Метод распараллеливания алгоритма общей рекурсии 1-го порядка.

Редукция - упрощение, в биологии уменьшение размера органа вплоть до его полного исчезновения. Циклическая редукция - алгоритмы численного анализа для распараллеливания последовательных алгоритмов, основанный на последовательном, циклическом применении параллельных вычислений, число которых на каждом этапе уменьшается (делится пополам).

Общей линейной рекурсией первого порядка называется система уравнений вида: $X_1 = D_1$

$$X_2 = X_1 * A_2 + D_2$$

$$X_i = X_{i-1} * A_i + D_i$$

$$X_n = X_{n-1} * A_n + D_n$$

в общем виде: $X_i = X_{i-1} * A_i + D_i$, $i = 2, 3, \dots, n$, $X_1 = D_1$

Последовательный алгоритм вычислений может быть записан так:

$$X(1) = A(1) + D(1)$$

DO $i = 2, n$

$$X(i) = X(i-1) * A(i) + D(i)$$

ENDDO

Рекурсивная зависимость итераций цикла не позволяет ускорить вычисления за счет параллельной работы оборудования. Преобразуем данный алгоритм в параллельный методом циклической редукции. Рассмотрим два соседних уравнения:

$$X_{i-1} = X_{i-2} * A_{i-1} + D_{i-1}$$

$$X_i = X_{i-1} * A_i + D_i$$

и подставив первое во второе, получаем:

$$X_i = (X_{i-2} * A_{i-1} + D_{i-1}) * A_i + D_i = X_{i-2} * A_{1i} + D_{1i}, \text{ где}$$

$$A_{1i} = A_i * A_{i-1},$$

$$D_{1i} = A_i * D_{i-1} + D_i$$

Тогда, проведя эту операцию для всей системы уравнений, получим систему уравнений порядка $n/2$. Если повторить процедуру l раз (если $n = 2^{**}l$), то в результате получается значение: $X_n = D_{nl}$. Для получения полного вектора X необходимо модифицировать алгоритм, например, по аналогии с алгоритмами суммирования.

Очевидно, что вычисления A_j и D_j можно проводить параллельно методом каскадных сумм с сохранением частных сумм. Приведенные уравнения для уровня i имеют вид:

$$X_i = A_{li} * X_{i-2^{**}l} + D_{li}, \text{ где } l = 0, 1, \dots, \log_2 n, i = 1, 2, \dots, n$$

$$A_{li} = A_{l-1i} * A_{l-1(i-2^{**}l-1)}$$

$$D_{li} = A_{l-1i} * D_{l-1(i-2^{**}l-1)} + D_{l-1i}$$

Начальные данные: $A_{0i} = A_i, D_{0i} = D_i$

Если индекс i у любого A_{li}, D_{li} и X_i попадает вне диапазона $1 \leq i \leq n$, то он должен быть приравнен к нулю. Тогда, при $l = \log_2 n$ в уравнениях: $X_i = A_{li} * X_{i-2^{**}l} + D_{li}$ индекс $X_{i-2^{**}l} = X_{i-n}$ находится вне диапазона, и, следовательно, решением системы уравнений будет:

вектор: $X_i = D_{li}$, Векторная нотация Хокни для данного алгоритма: $X = D$

DO L = 1, LOG2(N)

$$X = A * \text{SHIFTR}(X, 2^{**}(L-1)) + X$$

$$A = A * \text{SHIFTR}(A, 2^{**}(L-1))$$

ENDDO

Представление машинных чисел.

Для хранения и обработки данных в ЭВМ используется двоичная система, так как она требует наименьшего количества аппаратуры по сравнению с другими системами. Все остальные системы счисления применяются только для удобства пользователей.

Перевод числа из одной системы в другую выполняется по универсальному алгоритму, заключающемуся в последовательном делении целой части числа и образующихся целых частных на основание новой системы счисления, записанное в исходной системе счисления, и в последующем умножении дробной части и дробных частей получающихся произведений на то же основание, записанное в исходной системе счисления.

Разряд двоичного числа представляется в ЭВМ некоторым техническим устройством, например, триггером, двум различным состояниям которого приписываются значения 0 и 1. Группа таких устройств, предназначенная для представления в машине многоразрядного числа, называется регистром.

Структура двоичного регистра, представляющего в машине n -разрядное слово:

$n-1 \quad n-2 \quad \dots \quad 1 \quad 0$

Отдельные запоминающие элементы пронумерованы от 0 до $n-1$. Количество разрядов регистра определяет точность представления чисел. Путем соответствующего увеличения числа разрядов регистра может быть получена любая точность вычислений, однако это сопряжено с увеличением количества аппаратуры (в лучшем случае зависимость линейная, в худшем - квадратичная).

В ЭВМ применяются две основные формы представления чисел: полулогарифмическая с плавающей запятой и естественная с фиксированным положением запятой.

При представлении чисел с фиксированной запятой положение запятой закрепляется в определенном месте относительно разрядов числа и сохраняется неизменным для всех чисел, изображаемых в данной разрядной сетке. Обычно запятая фиксируется перед старшим разрядом или после младшего. В первом случае в разрядной сетке могут быть представлены только числа, которые по модулю меньше 1, во втором - только целые числа.

Для кодирования знака числа используется старший ("знаковый") разряд.

При выполнении арифметических действий над правильными дробями могут получаться двоичные числа, по абсолютной величине больше или равные единице, что называется переполнением разрядной сетки. Для исключения возможности переполнения приходится масштабировать величины, участвующие в вычислениях.

Диапазон представления правильных двоичных дробей:

$$2^{-(x-1)} < A < 1 - 2^{-(x-1)}.$$

Числа, которые по абсолютной величине меньше единицы младшего разряда разрядной сетки, называются машинным нулем.

Диапазон представления целых двоичных чисел со знаком в n-разрядной сетке:

$$0 < A < 2^{-(x-1)} - 1.$$

Использование представления чисел с фиксированной запятой позволяет упростить схемы машины, повысить ее быстродействие, но представляет определенные трудности при программировании. В настоящее время представление чисел с фиксированной запятой используется как основное только в микроконтроллерах.

В универсальных ЭВМ основным является представление чисел с плавающей запятой. Представление числа с плавающей запятой в общем случае имеет вид:

$$A = m * N^p,$$

где N - основание системы счисления,

p - целое число, называемое порядком числа A,

m - мантисса числа A ($|m| < 1$).

Так как в ЭВМ применяется двоичная система счисления, то

$$A = m * 2^p,$$

причем порядок и мантисса представлены в двоичной форме.

Двоичное число называется нормализованным, если его мантисса удовлетворяет неравенству

$$1/2 < |m| < 1.$$

Неравенство показывает, что двоичное число является нормализованным, если в старшем разряде мантиссы стоит единица. Например, число $0,110100 * 10100$ - нормализованное, а $0,001101 * 10110$ - ненормализованное.

Ситуация, когда в процессе вычислений получено число с $|m| > 1$ называется переполнением разрядной сетки.

Нормализованное представление чисел позволяет сохранить в разрядной сетке большее количество значащих цифр и, следовательно, повышает точность вычислений. Однако современные ЭВМ позволяют, при необходимости, выполнять операции также и над ненормализованными числами.

Диапазон представления нормализованных двоичных чисел, взятых по абсолютному значению, удовлетворяет неравенству:

$$2^{-1} * 2^{-(2k-1)} < A < (1 - 2^{-l}) * 2^{2k-1},$$

где l - число разрядов мантиссы;

k - число разрядов порядка;

2^{-1} - наименьшее значение нормализованной мантиссы;

$1 - 2^{-l}$ - наибольшее значение нормализованной мантиссы.

Широкий диапазон представления чисел с плавающей запятой удобен для научных и инженерных расчетов. Для повышения точности вычислений во многих ЭВМ

предусмотрена возможность использования формата двойной длины, однако при этом происходит увеличение затрат памяти на хранение данных и замедляются вычисления.

Представление отрицательных чисел в ЭВМ.

Для кодирования знака двоичного числа используется старший ("знаковый") разряд (ноль соответствует плюсу, единица - минусу). Такая форма представления числа называется прямым кодом. Формула для образования прямого кода правильной дроби имеет вид:

$$[A]_{np} = \begin{cases} A, A > 0 \\ 1 - A, A < 0 \end{cases}$$

Примеры:

$$A = 0,110111 \rightarrow [A]_{np} = 0,110111$$

$$A = -0,110111 \rightarrow [A]_{np} = 1 - (-0,110111) = 1,110111$$

Прямой код целого числа получается по формуле:

$$[A]_{np} = \begin{cases} A, A > 0 \\ 10^{n-1} - A, A < 0 \end{cases}$$

где 10 - число 2 в двоичной системе счисления,

n - количество позиций в разрядной сетке.

Например, при n=8

$$A = 110111 \rightarrow [A]_{np} = 00110111$$

$$A = -110111 \rightarrow [A]_{np} = 10000000 - (-110111) = 10110111$$

В ЭВМ прямой код применяется только для представления положительных двоичных чисел. Для представления отрицательных чисел применяется либо дополнительный, либо обратный код, так как над отрицательными числами в прямом коде неудобно выполнять арифметические операции.

Формула для образования дополнительного кода дроби:

$$[A]_{доп} = 10 + A.$$

Формула для образования обратного кода дроби:

$$[A]_{обр} = 10 - 10^{-(n-1)} + A.$$

Например, при n = 8, для A = -0,1100001

$$[A]_{доп} = 10 + (-0,1100001) = 1,0011111$$

$$[A]_{обр} = 10 - 10^{-7} + (-0,1100001) = 1,1111111 - 0,1100001 = 1,0011110.$$

Формула для образования дополнительного кода целого числа:

$$[A]_{доп} = 10n + A.$$

Формула для образования обратного кода целого числа:

$$[A]_{обр} = 10n - 1 + A.$$

Например, при n = 8, для A = -1100001

$$[A]_{доп} = 100000000 + (-1100001) = 10011111$$

$$[A]_{обр} = 100000000 - 1 + (-1100001) = 11111111 - 1100001 = 10011110.$$

Таким образом, правила для образования дополнительного и обратного кода состоят в следующем:

- для образования дополнительного кода отрицательного числа необходимо в знаковом разряде поставить единицу, а все цифровые разряды инвертировать (заменить 1 на 0, а 0 - на 1), после чего прибавить 1 к младшему разряду;
- для образования обратного кода отрицательного числа необходимо в знаковом разряде поставить единицу, а все цифровые разряды инвертировать.

Примечание: при данных преобразованиях нужно учитывать размер разрядной сетки.

Прямой код можно получить из дополнительного и обратного по тем же правилам, которые служат для нахождения дополнительного и обратного кодов.

Замена вычитания двоичных чисел $A1 - A2$ сложением с дополнениями $[A1]_{\text{пр}} + [-A2]_{\text{доп}}$ или $[A1]_{\text{пр}} + [-A2]_{\text{обр}}$ позволяет оперировать со знаковыми разрядами так же, как и с цифровыми. При этом перенос из старшего знакового разряда, если он возникает, учитывается по разному для обратного и дополнительного кодов:

- при использовании дополнительного кода единица переноса из знакового разряда отбрасывается;
- при использовании обратного кода единица переноса из знакового разряда прибавляется к младшему разряду суммы (осуществляется так называемый циклический перенос).

Пример: складываем числа $A1=0,10010001$ и $A2=-0,01100110$

При использовании обратного кода получим:

$[A1]_{\text{пр}} = 0,10010001$

+

$[A2]_{\text{обр}} = 1,10011001$

10,00101010

|_____1

Результат: 0,00101011

При использовании дополнительного кода получим:

$[A1]_{\text{пр}} = 0,10010001$

+

$[A2]_{\text{доп}} = 1,10011010$

Результат: 0,00101011

Если знаковый разряд результата равен нулю, то в получено положительное число, которое представлено в прямом коде. Если в знаковом разряде единица, то результат отрицательный и представлен в обратном или дополнительном коде.

Для того, чтобы избежать ошибок при выполнении бинарных операций, перед переводом чисел в обратные и дополнительные коды необходимо выравнивать количество разрядов прямого кода операндов.

При сложении чисел, меньших единицы, в машине быть получены числа, по абсолютной величине большие единицы. Для обнаружения переполнения разрядной сетки в ЭВМ применяются модифицированные прямой, обратный и дополнительный коды. В этих кодах знак кодируется двумя разрядами, причем знаку "плюс" соответствует комбинация 00, а знаку "минус" - комбинация 11.

Правила сложения для модифицированных кодов те же, что и для обычных. Единица переноса из старшего знакового разряда в модифицированном дополнительном коде отбрасывается, а в модифицированном обратном коде передается в младший цифровой разряд.

Признаком переполнения служит появление в знаковом разряде суммы комбинации 01 при сложении положительных чисел (положительное переполнение) или 10 при сложении отрицательных чисел (отрицательное переполнение). Старший знаковый разряд в этих случаях содержит истинное значение знака суммы, а младший является старшей значащей цифрой числа. Для коррекции переполнения число нужно сдвинуть в разрядной сетке на один разряд вправо, а в освободившийся старший знаковый разряд поместить цифру, равную новому значению младшего знакового разряда. После корректировки переполнения мантиссы результата необходимо увеличить на единицу порядок результата.

Система вещественных чисел, применяемая при ручных вычислениях, предполагается бесконечной и непрерывной, т.е. не существует никаких ограничений на диапазон используемых чисел и точность их представления.

Однако в компьютерах реализация такой системы на аппаратном уровне была бы нецелесообразной, хотя программно может быть реализована любая точность вычислений. Нецелесообразность аппаратной реализации вычислений с произвольной точностью вызвана тем, что такие вычисления требуют неоправданно большого расхода основных машинных ресурсов: памяти и процессорного времени.

Во всех компьютерах размеры регистров и ячеек памяти фиксированы, что ограничивает систему представления чисел. Ограничения касаются как диапазона, так и точности представления чисел, т.е. система машинных чисел оказывается конечной и дискретной.

В любой универсальной ЭВМ существует несколько различных форматов представления как для чисел с фиксированной, так и для чисел с плавающей запятой. На некоторые из форматов имеются международные стандарты, и поэтому такие форматы являются общими для ЭВМ, построенных различными фирмами на различной элементной базе.

Следует отметить, что нестандартные форматы обычно являются неявно специализированными для определенных областей применения, причем разработчики аппаратуры могут не указать в документации, для чего был предназначен тот или иной формат.

С точки зрения программиста важно, какие из форматов данных обрабатываются аппаратными средствами данной ЭВМ, а какие - только программными средствами. Операции над данными любого формата, который не поддерживается аппаратурой, выполняются очень медленно. Любой формат данных, который превышает размер регистров процессора, не пригоден для быстрых вычислений.

Для представления целых чисел в ЭВМ обычно применяются 8-, 16-, 32- и 64-битовый стандартные форматы, причем интерпретация чисел как знаковых или без знаковых обычно возлагается на программиста или на компилятор с языка высокого уровня.

Для представления чисел с плавающей запятой также существует несколько стандартных форматов, различающихся по точности, но имеющих одинаковую структуру следующего вида:

$$\begin{array}{ccccccc} & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ n-1 & n-2 & & & 0 & & \dots \end{array}$$

Порядок p задается в так называемой смещенной форме: если для задания порядка выделено k разрядов, то к истинному значению порядка прибавляют смещение, равное $(2^{k-1} - 1)$. Использование смещенной формы позволяет производить операции над порядками, как над без знаковыми числами, что упрощает операции сравнения, сложения и вычитания порядков. Кроме того, использование смещенного порядка упрощает операцию сравнения нормализованных чисел с плавающей запятой, сводя ее к операции сравнения целых чисел.

Следует отметить, что вещественный формат с m -разрядной мантиссой позволяет абсолютно точно представлять m -разрядные целые числа, т.е. любое двоичное целое число, содержащее не более m разрядов, может быть без искажений преобразовано в вещественный формат.

Арифметика машинных чисел.

Множество чисел ограниченной разрядности является моделью расширенной числовой прямой, т.е. числовой прямой с тремя абстракциями (потенциальной осуществимости): нуль, положительная бесконечность, отрицательная бесконечность.

Целые числа (в математике) и их аналоги в n -разрядных арифметиках тождественны (по отражаемым им количествам) в рамках их представления в этой разрядности. При этом можно отметить основные отличия представления чисел в поле памяти человека и в поле памяти n -разрядной арифметики (компьютера):

бесконечное и счётное (нумеруемое) множество целых чисел Z представляется отрезком $[-N; +N]$, где N - максимальное число, представимое в этой арифметике (многоточие - общее число единиц равно n): $N = (111 \dots 1)_2$;

бесконечное и несчётное множество действительных чисел $(-\infty; +\infty)$, располагающееся на числовой оси равномерно и плотно, представляется в n -разрядной арифметике множеством с неравномерной плотностью (сгущение у нуля и сжатость со стороны меньших чисел);

нуль во множестве действительных чисел R в любой своей окрестности имеет множество чисел, а нуль в n -разрядной арифметике представлен изолированно: в окрестности с радиусом равным наименьшему представимо в этой арифметике числу нет других чисел.

С точки зрения обычной арифметики, например, в интервале $(-1; 1)$ имеется бесконечное множество “плотно” расположенных точек, причем в любой окрестности каждой такой точки имеется хотя бы одна точка из этого множества. Такую арифметику называют часто регулярной арифметикой.

Машинная же арифметика нерегулярна - точки интервала сгущаются около нуля. Кроме того, в этом интервале точка x “изолирована” - если взять её любую окрестность $(x - a; x + a)$, где a - число, которое не превосходит машинного нуля (наименьшего представимого в машине числа), то в этом интервале нет других точек (отличных от x). Говоря языком теории вероятностей, плотности распределения чисел в регулярной и нерегулярной арифметике - различны, как, впрочем, плотности распределения целых и вещественных чисел в одной и той же арифметике. Множество вещественных чисел в машинной арифметике представляется как подмножество множества рациональных чисел, определяемое разрядностью арифметики.

Есть и другие особенности этих множеств (связанные, например, с выполнением операций), но указанные выше особенности - основные.

Различия в представлении чисел в обычной и в машинной (n-разрядной) арифметике ограничивают как “арифметические возможности” компьютера, так и “компьютерные возможности” арифметики, математики, использование математических методов, алгоритмов в компьютерах.

Нужно всегда иметь в виду, что точность в теоретической математике - понятие абстрактное и в практической математике может возникать иллюзия точности там, где её на самом деле нет, - если не произведена достаточно корректная интерпретация научно-практической точности т.е. нет корректной договорённости о пределах возможных значений неизбежных погрешностей в рамках рассматриваемых вычислительных ресурсов, например, трудоёмкости и времени, а также не оговорена стратегия и тактика управления этой погрешностью.

Так как диапазон n-разрядных чисел системы счисления с основанием p находится в пределах $|(x)_p| \leq p^n - 1$, то для представления дробных чисел этот диапазон ещё уменьшается, так как часть разрядов необходимо отвести под изображение мантиссы. Таким образом, имеются так называемые “зоны нечувствительности” форм представления чисел в n-разрядных арифметиках.

В 1937 году немецким учёным Конрадом Цузе (разработавшим, кстати говоря, не только ряд положений арифметических основ цифровых машин, но и прототипы ЦВМ – машины “Ц-1”, “Ц-2”) для увеличения диапазона чисел, представимых в арифметике двоичных чисел, а также для повышения точности этого представления чисел было предложено представление чисел в плавающей, нормализованной форме.

Число x представляется в нормализованном виде: $x = m' \cdot p^k$, где m - мантисса числа, k - целый порядок числа, $p^{-1} \leq |m| < 1$.

Пусть даны два числа $x = m' \cdot p^k$ и $y = n' \cdot p^l$ ($k > l$). Тогда можно проверить, что результаты выполнения операций будут равны:

$$x + y = (m + n' \cdot p^{l-k})' \cdot p^k,$$

$$x - y = (m - n' \cdot p^{l-k})' \cdot p^k,$$

$$x' \cdot y = (m' \cdot n) \cdot p^{k+l},$$

$$x/y = (m/n)' \cdot p^{k-l},$$

Если из n разрядов, отводимых под изображение чисел, m двоичных разрядов отвести под мантиссу, k - под порядок, один разряд - под знак числа и один разряд - под знак порядка (например, 0 - плюс, 1 - минус), то диапазон представимых в форме с плавающей запятой чисел резко увеличивается ($m + k + 2 = n$):

$$-(0.111 \dots 1)_2 \cdot (10)_2^{+(111 \dots 1)_2} \leq x \leq (0.111 \dots 1)_2 \cdot (10)_2^{+(111 \dots 1)_2}$$

(многоточие соответствует k единицам).

Числа, меньшие нижней границы положительных чисел и большие верхней границы отрицательных чисел, считаются равными нулю, не различаются между собой. Числа, большие верхней границы положительных чисел полагаются равными положительной бесконечности, а меньшие нижней границы отрицательных – отрицательной бесконечности. Сравнение двух разных по величине чисел в арифметике с ограниченной разрядностью может приводить, поэтому, к неверному результату, как и сравнение двух равных в таких системах чисел с точки зрения математической.

Такое представление очень удобно для хранения в ЭВМ, так как на самом деле необходимо хранить не само число, а его знак, мантиссу, порядок и знак порядка и все операции с числами сводятся к операциям с этими, более “компактными”, объектами. Операции с этими объектами достаточно просты: сравнение знаков, увеличение,

уменьшение порядка, сложение мантисс, нормализация, т.е. в конечном итоге сводятся к достаточно просто реализуемым операциям сдвига, выравнивания, сравнения разрядов. Это упрощает аппаратную их реализацию и является основой для различных архитектур – микропрограммных, RISC и др.

Пример. В 16-разрядной арифметике двоичных чисел можно представить диапазон целых чисел x : $1 - 2^{15} < x < 2^{15} - 1$ (старший разряд отведён под знак числа). Если в этой арифметике (не меняя её разрядность) отвести 7 разрядов под мантиссу, а 7 разрядов - под порядок, то уже представим диапазон чисел: $-127 \cdot 2^{127} < x < 127 \cdot 2^{127}$ (два разряда - под знак числа и знак порядка; несколько упрощена и общая картина представления - для наглядности).

К “неудобствам” этой формы представления чисел можно отнести возможность возникновения следующих “особо опасных” ситуаций:

а) если число достаточно мало, например, $a = 0.12E+00$, то оно может быть представлено любым числом из наименьшего интервала включающего a , в частности, числом 0.120000001 или 0.199999999 и в этом случае сравнивать на равенство “в лоб” нельзя (вещественные числа в форме с плавающей запятой на совпадение опасно сравнивать);

б) порядок выполнения операций может влиять на результат, например, в 4-разрядной арифметике с фиксированной запятой $20.0000 + 0.0001 = 20.0001$, но при этом $0.2000E+02 + 0.1000E-05 = 0.2000E+02$;

в) может возникнуть так называемая ситуация “переполнения порядка” при сложении (умножении) “очень больших чисел” или “исчезновения порядка” при сложении (умножении) “очень малых чисел”, например, результат $0.6000E+39 \cdot 0.1200E+64$ равен $0.9999E+99$ (или не определен) и результат $0.6000E-35 \cdot 0.0200E-65$ равен $0.9999E-99$ (или не определен) при соответствующим образом определенной разрядности десятичной арифметики;

г) при сложении чисел с плавающей запятой (а в конечном счёте, все операции выполняются, как известно, через сложение, точнее, - через поразрядное сравнение и сдвиги) происходит выравнивание порядков для последующего сложения мантисс, а при выравнивании степеней может происходить потеря (усечение) младших разрядов, например, такая ситуация может возникнуть при сложении одного “очень большого числа” с одним “очень малым числом” (почему?).

Реализация операций в арифметике с плавающей запятой требует необходимости выравнивания порядков при сложении и вычитании и нормализации результатов. Если диапазоны чисел, представимых в арифметике с фиксированной запятой и с плавающей запятой, соизмеримы, то числа с фиксированной запятой могут более точно представлять (кодировать) величины, так как свободны от часто необходимой для чисел с плавающей запятой операции округления. При машинной реализации такая операция обычно выполняется в устройстве-предшественнике (например, сумматор) с высокой точностью (большой разрядностью), а затем отсылается в устройство-преемник (например, регистр) с учётом заданной (например, декларированной в описаниях типов и структур данных) точности или с сохранением всех значащих разрядов. Таким образом, копирование непосредственного результата операции происходит либо с помощью операции округления, либо с помощью операции усечения.

Эти две основные операции (кроме арифметических операций) вводятся следующим образом:

усечение, отбрасывание цифр числа до определённого разряда, например, до ближайшего, меньшего целого числа и т.п.;

округление, усечение с коррекцией числа по определённым правилам, например, до числа кратного заданному числу, до ближайшего целого и т.п.

Пример. Операция взятия целой части числа x (функция Антье - $[x]$ или $\text{int}(x)$) - операция усечения, $[2.65]=2$, $[-1.999]=-2$. Операция взятия целой части числа $x+0.5$ вместо значения числа x есть округление $\{x\}$ до ближайшего целого, $\{1.05\}=\{0.91\}=1$, $\{2.61\}=[2.6+0.5]=3$.

Дополнение

Целочисленные переменные

Тип целое число является основным для любого алгоритмического языка. Связано это с тем, что содержимое ячейки памяти или регистра процессора можно рассматривать как целое число. Адреса элементов памяти также представляют собой целые числа, с их помощью записываются машинные команды и т.д. Символы представляются в компьютере целыми числами - их кодами в некоторой кодировке. Изображения также задаются массивами целых чисел: для каждой точки цветного изображения хранятся интенсивности ее красной, зеленой и синей составляющей (в большинстве случаев - в диапазоне от 0 до 255). Как говорят математики, целые числа даны свыше, все остальное сконструировал из них человек.

Общепринятый в программировании термин целое число или целочисленная переменная, строго говоря, не вполне корректен. Целых чисел бесконечно много, десятичная или двоичная запись целого числа может быть сколь угодно длинной и не помещаться в области памяти, отведенной под одну переменную. Целая переменная в компьютере может хранить лишь ограниченное множество целых чисел в некотором интервале. В современных компьютерах под целую переменную отводится 4 байта, т.е. 32 двоичных разряда. Она может хранить числа от нуля до 2 в 32-й степени минус 1. Таким образом, максимальное целое число, которое может храниться в целочисленной переменной, равно $2^{32} - 1 = 4294967295$

Сложение и умножение значений целых переменных выполняется следующим образом: сначала производится арифметическая операция, затем старшие разряды результата, вышедшие за границу тридцати двух двоичных разрядов (т.е. четырех байтов), отбрасываются. Определенные таким образом операции удовлетворяют традиционным законам коммутативности, ассоциативности и дистрибутивности:

$$a+b = b+a, \quad ab = ba$$

$$(a+b) + c = a+(b+c), \quad (ab)c = a(bc)$$

$$a(b+c) = ab+ac$$

Кольцо вычетов по модулю m

Целочисленный тип компьютера в точности соответствует важнейшему понятию математики - понятию кольца вычетов по модулю m . В качестве m выступает число $2^{32} = 4294967296$. В математике кольцо Z_m определяется следующим образом. Все множество целых чисел Z разбивается на m классов, которые называются классами эквивалентности. Каждый класс содержит числа, попарная разность которых делится на m . Первый класс содержит числа

$$\{..., -2m, -m, 0, m, 2m, ...\}$$

второй

$$\{..., -2m+1, -m+1, 1, m+1, 2m+1, ...\}$$

последний

$$\{..., -m-1, -1, m-1, 2m-1, 3m-1, ...\}$$

Элементами кольца Z_m являются классы эквивалентности. Их ровно m , так что, в отличие от множества целых чисел Z , кольцо Z_m содержит конечное число элементов. Операции с классами выполняются следующим образом: надо взять по одному представителю из каждого класса, произвести операцию и определить, в какой класс попадает результат. Этот класс и будет результатом операции. Легко показать, что он не зависит от выбора представителей.

Все числа, принадлежащие одному классу эквивалентности, имеют один и тот же остаток при делении на m . Таким образом, класс эквивалентности однозначно определяется остатком от деления на m . Традиционно остаток выбирается неотрицательным, в диапазоне от 0 до $m-1$. Остатки используют для обозначения классов, при этом используются квадратные скобки. Так, выражение $[5]$ обозначает класс эквивалентности, состоящий из всех чисел, остатки которых при делении на m равны пяти. Все кольцо Z_m состоит из элементов $[0], [1], [2], \dots, [m-1]$, например, кольцо Z_5 состоит из элементов $[0], [1], [2], [3], [4]$.

В элементарной школьной математике результат операции остатка от деления традиционно считается неотрицательным. Операция нахождения остатка будет обозначаться знаком процента %, как в языке Си. Тогда, к примеру,

$$\begin{aligned} 3\%5 &= 3, \\ 17\%5 &= 2, \\ (-3)\%5 &= 2, \\ (-17)\%5 &= 3. \end{aligned}$$

Отсюда видно, что в школьной математике не выполняется равенство $(-a)\%b = -(a\%b)$, т.е. операции изменения знака и нахождения остатка не перестановочны (на математическом языке, не коммутируют друг с другом). В компьютере операция нахождения остатка от деления для отрицательных чисел определяется иначе, ее результат может быть отрицательным. В приведенных примерах результаты будут следующими:

$$\begin{aligned} 3\%5 &= 3, \\ 17\%5 &= 2, \\ (-3)\%5 &= -3, \\ (-17)\%5 &= -2. \end{aligned}$$

При делении на положительное число знак остатка совпадает со знаком делимого. При таком определении тождество $(-a)\%b = a\%(-b) = -(a\%b)$ справедливо. Это позволяет во многих алгоритмах не следить за знаками (так же, как в тригонометрии формулы, выведенные для углов, меньших 90 градусов, автоматически оказываются справедливыми для любых углов).

Вернемся к рассмотрению кольца Z_m . Выберем по одному представителю из каждого класса эквивалентности, которые составляют множество Z_m . Систему таких представителей называют системой остатков. Традиционно рассматривают две системы остатков: неотрицательную систему и симметричную систему. Неотрицательная система остатков состоит из элементов $0, 1, 2, 3, \dots, m-1$.

Очень удобна также симметричная система остатков, состоящая из отрицательных и неотрицательных чисел, не превосходящих $m/2$ по абсолютной величине. Пусть $k = \text{целая часть}(m/2)$

тогда симметричная система остатков при нечетном m состоит из элементов $-k, -k+1, \dots, -1, 0, 1, \dots, k-1, k$,

а при четном m - из элементов
 $-k, -k+1, \dots, -1, 0, 1, \dots, k-1$.

Например, при $m = 5$ симметричная система остатков состоит из элементов
 $-2, -1, 0, 1, 2$.

Кольцо Z_m можно представлять состоящим из элементов, принадлежащих выбранной системе остатков. Арифметические операции определяются следующим образом: надо взять два остатка, произвести над ними операцию как над обычными целыми числами и выбрать тот остаток, который лежит в том же классе эквивалентности, что и результат операции. Например, для симметричной системы остатков множества Z_5 имеем:

$$\begin{aligned} 1+1 &= 2, & 1+2 &= -2, \\ 1+(-2) &= -1, & 1+(-1) &= 0, \\ (-2)+2 &= 0, & (-2)+(-2) &= 1. \end{aligned}$$

Интерпретация положительных и отрицательных чисел

В кольце вычетов невозможно определить порядок, согласованный с операциями (т.е. так, чтобы, к примеру, сумма двух положительных чисел была положительной). Таким образом, в компьютере нет, строго говоря, положительных и отрицательных целых чисел, поскольку компьютерные целые числа - это на самом деле элементы кольца вычетов. Выбирая либо неотрицательную, либо симметричную систему остатков, можно интерпретировать эти числа либо как неотрицательные в диапазоне от нуля до $m-1$, либо как отрицательные и положительные числа в диапазоне от $-k$ до k , где k - целая часть от деления m на 2.

В программировании симметричная система остатков более популярна, поскольку трудно обойтись без отрицательных чисел. При этом следует понимать, что сумма двух положительных чисел может оказаться отрицательной, или, наоборот, сумма двух отрицательных чисел - положительной. Иногда в программировании такую ситуацию называют переполнением. Привычные свойства целочисленных операций в компьютере выполняются лишь для небольших чисел, когда результат операции не превосходит числа $m = 232$. В случае целочисленных переменных переполнение не является экстраординарной ситуацией и не приводит к аппаратным ошибкам или прерываниям. (Это, кстати, отличает компьютерные целые числа от вещественных.) Переполнение - совершенно нормальная ситуация, если вспомнить, что компьютер работает с элементами кольца вычетов по модулю m , а не с настоящими целыми числами.

Следует также отметить, что симметричная система остатков кольца Z_m в случае четного m (а m для компьютера равно 232, т.е. четно) не вполне симметрична. Поскольку ноль не имеет знака, то число положительных остатков не может равняться числу отрицательных.

Какой остаток выбрать в классе эквивалентности числа $k = m/2$? Для этого элемента выполняется непривычное с точки зрения школьной математики равенство
 $k+k \equiv 0 \pmod{m}$, т.е. $k \equiv -k \pmod{m}$

Как отрицательный остаток $-k$, так и положительный k в равной мере подходят для представления этого класса эквивалентности. По традиции выбирается отрицательный остаток. Таким образом, в компьютере количество отрицательных целых чисел на единицу больше, чем количество положительных. Так как $m = 232 = 4294967296$, то $k = 231 = 2147483648$, и симметричная система остатков состоит из элементов
 $-2147483648, -2147483647, \dots, -2, -1, 0, 1, 2, \dots, 2147483647$.

Вещественные переменные

Вещественные числа представляются в компьютере в так называемой экспоненциальной, или плавающей, форме. Вещественное число r имеет вид $r = \pm 2^e \cdot m$

Представление числа состоит из трех элементов:

знак числа - плюс или минус. Под знак числа отводится один бит в двоичном представлении, он располагается в старшем, т.е. знаковом разряде. Единица соответствует знаку минус, т.е. отрицательному числу, ноль - знаку плюс. У нуля знаковый разряд также нулевой;

показатель степени e , его называют порядком или экспонентой. Экспонента указывает степень двойки, на которую домножается число. Экспонента может быть как положительной, так и отрицательной (для чисел, меньших единицы). Под экспоненту отводится фиксированное число двоичных разрядов, обычно восемь или одиннадцать, расположенных в старшей части двоичного представления числа, сразу вслед за знаковым разрядом;

мантисса m представляет собой фиксированное количество разрядов двоичной записи вещественного числа в диапазоне от 1 до 2:

$$1 \leq m < 2$$

Следует подчеркнуть, что левое неравенство нестрогое - мантисса может равняться единице, а правое - строгое, мантисса всегда меньше двух. Разряды мантиссы включают один разряд целой части, который ввиду приведенного неравенства всегда равен единице, и фиксированное количество разрядов дробной части. Поскольку старший двоичный разряд мантиссы всегда равен единице, хранить его необязательно, и в двоичном коде он отсутствует. Фактически двоичный код хранит только разряды дробной части мантиссы.

Несколько примеров представления вещественных чисел в плавающей форме:

$$1.0 = +2^0 \cdot 1.0$$

Здесь порядок равен 0, мантисса - 1. В двоичном коде мантисса состоит из одних нулей, так как старший разряд мантиссы (всегда единичный) в коде отсутствует. Порядок хранится в двоичном коде в смещенном виде, он равен 127 в случае float и 1023 в случае double;

$$3.5 = +2^1 \cdot 1.75$$

Порядок равен единице, мантисса состоит из трех единиц, из которых в двоичном коде хранятся две: 1100...0; смещенный порядок равен 128 для float и 1024 для double;

$$0.625 = +2^{-1} \cdot 1.25$$

Порядок отрицательный и равен -1, дробная часть мантиссы равна 0100...0; смещенный порядок равен 126 для float и 1022 для double;

$$100.0 = +2^6 \cdot 1.5625$$

Порядок равен шести, дробная часть мантиссы равна 100100...0; смещенный порядок равен 133 для float и 1029 для double.

При выполнении сложения двух положительных плавающих чисел происходят следующие действия:

выравнивание порядков. Определяется число с меньшим порядком. Затем последовательно его порядок увеличивается на единицу, а мантисса делится на 2, пока порядки двух чисел не сравняются. Аппаратно деление на 2 соответствует сдвигу двоичного кода мантиссы вправо, так что эта операция выполняется быстро. При сдвигах правые разряды теряются, из-за этого может произойти потеря точности (в случае, когда правые разряды ненулевые); сложение мантисс;

нормализация: если мантисса результата стала равна или превысила двойку, то порядок увеличивается на единицу, а мантисса делится на 2. В результате этого мантисса попадает в

интервал $1 - m < 2$. При этом возможна потеря точности, а также переполнение, когда порядок превышает максимально возможную величину.

Вычитание производится аналогичным образом. При умножении порядки складываются, а мантиссы перемножаются как целые числа, после чего у результата правые разряды отбрасываются.

Погрешности при вычислениях чисел на параллельных системах. Оценить полную ошибку суммирования положительных чисел.

Источники погрешности при вычислениях на параллельных системах.

В общем случае, арифметические операции над элементами дискретного подмножества вещественных чисел F не корректны.

Результат арифметических операций чисел с плавающей запятой может:

- иметь абсолютное значение, больше M (максимального числа) - машинное переполнение;
- иметь ненулевое значение, меньше m (минимального числа) по абсолютной величине - машинный ноль;
- иметь значение в диапазоне $[m:M]$ и тем не менее не принадлежать множеству F (произведение двух чисел из F , как правило, записывается посредством $2t$ либо $2t-1$ значащих цифр);

Поэтому, на множестве чисел с плавающей запятой определяются и "плавающие" арифметические операции, за результаты которых, если они не выходят за границы множества F , принимаются ближайшие по значению элементы F . Примеры из четырехразрядной десятичной арифметики по Н. Вирту.

А) Пусть $x=9.900$ $y=1.000$ $z=-0.999$ и тогда:

$$1 \quad (x+y)+z = 9.910$$

$$2 \quad x+(y+z) = 9.901$$

В) Пусть $x=1100.$ $y=-5.000$ $z=5.001$ и тогда:

$$1 \quad (x*y)+(x*z) = 1.000$$

$$2 \quad x*(y+z) = 1.100$$

Здесь операции $+$ и $*$ - плавающие машинные операции. Такие 'численные' процессы называют иногда 'неточными', здесь нарушаются ассоциативный и дистрибутивный законы арифметики..

39. Оценить полную ошибку для суммирования положительных чисел.

Пример расчета полной ошибки для суммирования положительных чисел

Формула полной ошибки для суммирования положительных чисел $A_i (i=1,...,n)$ имеет вид:

$D_s = A_1*da_1 + A_2*da_2 + ... + A_n*da_n + d_1*(A_1+A_2) + ... + d_{(n-1)}*(A_1+...+A_n) + d_n$, где

da_i - относительные ошибки представления чисел в ЭВМ, а d_i - относительные ошибки округления чисел при каждой следующей операции сложения. Пусть: все $da_i = da$ и $d_i = d$, а $K_s = A_1+A_2+...+A_n$, тогда: $D_s = da*K_s + d*[(n-1)*A_1+(n-1)*A_2 + ... + 2*A_{(n-1)} + A_n]$

Очевидно, что наибольший "вклад" в сумму ошибок вносят числа, суммируемые вначале.

Следовательно, если суммируемые положительные числа упорядочить по возрастанию, максимально возможная ошибка суммы будет минимальной. Изменяя порядок суммирования чисел можно получать различные результаты. Но если даже слагаемые отличаются друг от друга незначительно, на точность результата может оказать влияние способ суммирования. Пусть суммируются 15 положительных чисел, тогда ошибка результата: $D_s = da*K_s + d*(14*A_1+14*A_2+13*A_3+...+2*A_{14}+A_{15})$.

Слагаемое $da*K_s$ не зависит от способа суммирования, и далее не учитывается. Пусть слагаемые имеют вид: $A_i = A_0+e_i$, где $i=1,...,15$, тогда: $D_{ss} = 199*(A_0+em)*d$, где $em = \max(e_i)$, d - ошибка округления при выполнении арифметической операции сложения.

Если провести суммирование этих чисел по группам (три группы по четыре числа и одна группа из трех чисел), то ошибки частных сумм имеют вид:

$$Ds1 = d*(3*A1+3*A2+2*A3+A4) \leq 9*d*(A0+em)$$

$$Ds2 = d*(3*A5+3*A6+2*A7+A8) \leq 9*d*(A0+em)$$

$$Ds3 = d*(3*A9+3*A10+2*A11+A12) \leq 9*d*(A0+em)$$

$$Ds4 = d*(3*A13+2*A14+A15) \leq 5*d*(A0+em)$$

а полная оценка ошибок округления будет $Ds \leq 32*d*(A0+em)$, что меньше

Dss . Итак суммирование по группам дает меньшую ошибку результата.

Например, разделив процесс суммирования массива положительных чисел на параллельные процессы, и затем получив сумму частных сумм, можно получить результат, отличный от последовательного суммирования в одном процесс.

Точность плавающей арифметики. Машинный эпсилон.

Действия с плавающими числами из-за ошибок округления лишь приближенно отражают арифметику настоящих вещественных чисел. Так, если к большому плавающему числу прибавить очень маленькое, то оно не изменится. Действительно, при выравнивании порядков все значащие биты мантиссы меньшего числа могут выйти за пределы разрядной сетки, в результате чего оно станет равным нулю. Таким образом, с плавающими числами возможна ситуация, когда

$$a+b = a \quad \text{при } b \neq 0$$

Более того, для сложения не выполняется закон ассоциативности: $a+(b+c) = (a+b)+c$

Действительно, пусть ϵ - максимальное плавающее число среди чисел, удовлетворяющих условию $1.0+\epsilon = 1.0$ (приведенные выше рассуждения показывают, что такие числа существуют). Тогда $(1.0+\epsilon)+\epsilon = 1.0+(\epsilon+\epsilon)$ поскольку левая часть неравенства равна единице, а правая строго больше единицы (это следует из максимальности числа ϵ).

Число ϵ часто называют машинным эпсилоном или, чуть менее корректно, машинным нулем, поскольку при прибавлении к единице оно ведет себя как ноль. Величина машинного эпсилона характеризует точность операций компьютера. Она примерно одинакова для всех современных компьютеров: большинство процессоров работают с восьмибайтовыми плавающими числами (тип `double` в Си), а арифметика плавающих чисел подчиняется строгим международным стандартам.

Оценим величину машинного эпсилона для типа `double`. Число 1.0 записывается в плавающей форме как $1.0 = +20*1.0$.

Порядок плавающего числа 1.0 равен нулю. При сложении 1.0 с числом ϵ производится выравнивание порядка путем многократного сдвига мантиссы числа ϵ вправо и увеличения его порядка на 1. Поскольку все разряды числа ϵ должны в результате выйти за пределы разрядной сетки, должно быть выполнено 53 сдвига. Порядок числа ϵ после этого должен стать равным порядку числа 1.0, т.е. нулю. Следовательно, изначально порядок числа ϵ должен быть равным -53: $\epsilon = 2^{-53}*m$ где m - число в диапазоне от единицы до двух. Таким образом, величина машинного эпсилона составляет примерно $2^{-53} \approx 10^{-16}$

Приблизительно точность вычислений составляет 16 десятичных цифр. (Это также можно оценить следующим образом: 53 двоичных разряда составляют примерно $53/\log_2 10 \approx 15.95$ десятичных, поскольку $53/\log_2 10 \approx 15.95$.)

В случае четырехбайтовых плавающих чисел (тип `float` языка Си) точность вычислений составляет примерно 7 десятичных цифр. Это очень мало, поэтому тип `float` чрезвычайно

редко применяется на практике. К тому же процессор сконструирован для работы с восьмибайтовыми вещественными числами, а при работе с четырехбайтовыми он все равно сначала приводит их к восьмибайтовому типу. В программировании следует избегать типа `float` и всегда пользоваться типом `double`.

Некоторые процессоры применяют внутреннее представление плавающих чисел с большим количеством разрядов мантииссы. Например, процессор Intel использует 80-битовое (десятибайтовое) представление. Поэтому точность вычислений, которые не записывают промежуточные результаты в память, может быть несколько выше указанных оценок.

Кроме потери точности, при операциях с вещественными числами могут происходить и другие неприятности:

переполнение - когда порядок результата больше максимально возможного значения. Эта ошибка часто возникает при умножении больших чисел;

исчезновение порядка - когда порядок результата отрицательный и слишком большой по абсолютной величине, т.е. порядок меньше минимально допустимого значения. Эта ошибка может возникнуть при делении маленького числа на очень большое или при умножении двух очень маленьких по абсолютной величине чисел.

Кроме того, некорректной операцией является деление на ноль. В отличие от операций с целыми числами, переполнение и исчезновение порядка считаются ошибочными ситуациями и приводят к аппаратному прерыванию работы процессора. Программист может задать реакцию на прерывание - либо аварийное завершение программы, либо, например, при переполнении присваивать результату специальное значение плюс или минус бесконечность, а при исчезновении порядка - ноль. Заметим, что среди двоичных кодов, представляющих плавающие числа, имеется несколько специальных значений. Перечислим некоторые из них:

бесконечно большое число - это плавающее число с очень большим положительным порядком и, таким образом, очень большое по абсолютной величине. Оно может иметь знак плюс или минус;

бесконечно малое, или денормализованное, число - это ненулевое плавающее число с очень большим отрицательным порядком (т.е. очень маленькое по абсолютной величине);

Not a Number, или NaN - двоичный код, который не является корректным представлением какого-либо вещественного числа.

Любые операции с константой NaN приводят к прерыванию, поэтому она удобна при отладке программы - ею перед началом работы программы инициализируются значения всех вещественных переменных. Если в результате ошибки программиста при вычислении выражения используется переменная, которой не было присвоено никакого значения, то происходит прерывание из-за операции со значением NaN и ошибка быстро отслеживается. К сожалению, в случае целых чисел такой константы нет: любой двоичный код представляет некоторое целое число.

Перечислить алгоритмы оптимизации объектных программ, которые могут повлиять на точность вычислений.

Оптимизационные преобразования программ для их оптимального выполнения на конвейерных вычислителях могут проводиться системами программирования. Эти преобразования, алгебраически эквивалентные, могут нарушить порядок вычислений, предписанный исходным текстом программы.

Последствия таких преобразований обсуждались выше. Наиболее характерные преобразования следующие.

1. Балансировка дерева вычислений

Балансировка дерева вычислений (tree-height reduction or balancing) выражений позволяют использовать конвейерное АУ без пропуска рабочих тактов. Так, вычисление суммы вещественных чисел: $A+B+C+D+E+F+G+H$, будет запрограммировано как последовательность операций: $((A+B)+(C+D))+((E+F)+(G+H))$; это нарушает заданную по умолчанию последовательность вычислений с накоплением одной частной суммы и может повлиять на результат.

2. Исключение общих подвыражений

Алгоритмы исключения общих подвыражений (Common subexpression elimination) также могут изменить порядок вычислений.

Если компилятор распознает в выражениях повторяющееся вычисление, то это вычисление производится один раз, его результат сохраняется на регистре, и в дальнейшем используется этот регистр. Тем самым исключается избыточность вычислений.

```
X = A + B + C + D    ---->    REG = B + C
Y = B + E + C          X  = A + D + REG
                        Y  = E + REG
```

3. Разворачивание циклов

Разворачивание циклов (loop unrolling) - расписывание цикла последовательностью операторов присваивания: либо полностью, либо разное количество раз с некоторым коэффициентом (фактором) размножения.

Производится частичное или полное разворачивание цикла в последовательный участок кода. При частичном разворачивании используется так называемый фактор разворачивания (который можно задавать в директиве компилятору).

```
DO I=1,100          DO I=1,100,4
  A(I) = B(I) + C(I)  A(I) = B(I) + C(I)
ENDDO               A(I+1) = B(I+1) + C(I+1)
                   A(I+2) = B(I+2) + C(I+2)
                   A(I+3) = B(I+3) + C(I+3)
ENDDO
```

При этом преобразовании снижается количество анализов итерационной переменной. Данный алгоритм также может привести к нарушению предписанного первоначально порядка вычислений. Например:

```
DO I=1,10          DO I=1,10,2
  S = S + A(I)      S = S + A(I)
ENDDO              S1 = S1 + A(I+1)
                   ENDDO
                   S = S + S1
```

Здесь, суммирование проводится отдельно для четных и нечетных элементов с последующим сложением частных сумм.