МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ
КАФЕДРА ЭЛЕКТРОННЫХ ВЫЧИСЛИТЕЛЬНЫХ МАШИН

В.Ю. Мельцов
Л.Ф. Фоминых

ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

Учебное пособие

Утверждено
Ученым советом ВятГУ
в качестве учебного пособия

Киров 2002

УДК 681.324. – 50.07.05
М 48

Мельцов В.Ю., Фоминых Л.Ф. Высокопроизводительные вычислительные системы: Учебное пособие - Киров: Изд-во ВятГУ, 2002.- 160 с.

Увеличение производительности ЭВМ достигается совершенствованием элементной базы и применением новых "нетрадиционных" архитектур параллельной обработки. которые лежат в основе всех высокопроизводительных вычислительных систем (ВПВС). Многочисленные исследования и разработки параллельных архитектур привели к большому числу публикаций и множеству новых терминов, часто трактуемых неоднозначно. К сожалению, в нашей стране публикации по высокопроизводительным системам в основном ориентированы на узких специалистов. Вследствие этого авторы ставили перед собой задачу в доступной для студентов и аспирантов форме изложить принципы организации параллельной обработки информации, а также пояснить значение основных терминов из этой области.

Пособие рассчитано на студентов старших курсов и аспирантов, обучающихся по специальности 220100 «Вычислительные машины, комплексы, системы и сети» при изучении дисциплин "Организация ЭВМ и систем", "Вычислительные системы и комплексы". Оно может быть полезным студентам других специальностей, знакомых с архитектурой традиционных последовательных машин, элементной базой и основами программирования.

Рецензенты: профессор, доктор технических наук, заведующий кафедрой автоматики и телемеханики ВятГУ В.И. Пономарев,
доцент, кандидат технических наук, директор государственного предприятия "Лаборатория автоматики и вычислительной техники" Н.А. Чарушин

Содержание

1. Введение.

Электронные вычислительные машины создавались как инструмент для облегчения и ускорения вычислений. Поэтому требование повышения их быстродействия определяется целью создания ЭВМ и, наряду с требованием повышения надежности, является постоянно действующим фактором, оказывающим самое существенное влияние на развитие архитектуры, элементной базы, конструкции и программного обеспечения.

Высокопроизводительные машины, обладающие наивысшими достижимыми показателями быстродействия и производительности на данный период времени, занимают особое место на каждом этапе развития вычислительной техники. Такие машины часто называю суперЭВМ, хотя в действительности в настоящее время они представляют собой сложные аппаратно-программные комплексы и системы, содержащие разнообразные устройства и подчас работающие под управлением специально выделенной ЭВМ. В высокопроизводительных ВС всех поколений сосредотачивались наиболее прогрессивные решения как в области архитектуры, так и в области элементной базы, которые затем переносились на ЭВМ других классов в последующих поколениях машин. Так принцип конвейерной обработки команд, впервые реализованный в суперЭВМ второго поколения БЭСМ-6, теперь проник даже в персональные ЭВМ и микропроцессоры, широко используется кэш-память, в состав включаются матричные сопроцессоры и т.д.

Сокращение времени выполнения программ является одной из основных задач при проектировании ЭВМ. Это объясняется желанием пользователя получать ответы за возможно более короткие интервалы времени, а также желанием выполнять все большие объемы вычислительной работы при решении различных задач. Во многих областях сокращение времени ответа, или реакции системы, имеет решающее значение, так как запаздывание в получении результатов может полностью их обесценить. К настоящему времени создались отрасли науки и техники, развитие которых не может происходить без применения высокопроизводительных ВС. К таким отраслям относятся:

- Квантовая физика: физика элементарных частиц, ядерная физика, физика молекул (исследование и предсказание молекулярных свойств материалов), физика плазмы (моделирование поведения плазмы на ЭВМ).
- Квантовая химия (структура молекул и кристаллов, химические реакции).
- Науки о Земле: физика атмосферы, метеорология, климатология (предсказание погоды и изменения климата), геофизика (движение Земной коры и землетрясения), физика океана.
- Биология, экология (прогнозирование развития экосистем).
- Экономика и эконометрия: вычислительная экономика (новая, развивающаяся наука о применении компьютерного моделирования к исследованию сложных, реалистичных моделей экономических процессов), макроэкономика , теория массового обслуживания (например моделирование развития транспортной системы страны, грузопотоков и пассажиропотоков) и теория оптимального управления, финансовая деятельность (моделирование рынка ценных бумаг, банковской деятельности).
- Социальные науки (моделирование демографической ситуации в стране, миграции и занятости населения, социального поведения).
- Математическая лингвистика: распознавание речи, анализ текста и автоматический перевод.
- Информатика: ведение баз данных, распознавание образов, распределенные вычислительные системы.
- Механика сплошных сред: гидродинамика и газодинамика (кораблестроение и самолетостроение, течения, турбостроение), теория сопротивления материалов (устойчивость конструкций и их нагрузочная способность).

- Баллистика (наведение баллистических снарядов и управление реактивным движением).

- Медицина, фармацевтика (моделирование лекарственных препаратов).

- Промышленность, в том числе автомобиле- и авиастроение, нефте- и газодобыча и т.д.

Во всех вышеперечисленных отраслях часто возникают вычислительные задачи и задачи обработки информации, требующие больших затрат вычислительных ресурсов.

Традиционно архитектура ЭВМ была последовательной. Это означало, что в любой момент времени выполнялась только одна операция и только над одним операндом. Появление в середине шестидесятых первых компьютеров класса суперЭВМ, разработанных в фирме Control Data Corporation (CDC) знаменитым Сеймуром Крэем, ознаменовало рождение новой - векторной архитектуры. Основная идея, положенная в основу новой архитектуры, заключалась в распараллеливании процесса обработки данных, когда одна и та же операция применяется одновременно к массиву (вектору) значений. В этом случае можно надеяться на определенный выигрыш в скорости вычислений. Идея параллелизма оказалась плодотворной и нашла воплощение на различных уровнях создания и функционирования компьютера.

На самом нижнем уровне - это передовая технология конструирования и изготовления быстродействующих элементов и плат с высокой плотностью монтажа. В этой сфере лежит наиболее прямой путь к увеличению скорости, поскольку если бы, например, удалось все задержки в машине сократить в **k** раз, то это привело бы к увеличению быстродействия в такое же число раз.

Следующий уровень охватывает способы реализации основных операций, таких как сложение, умножение и деление. Для того, чтобы увеличить скорость выполнения этих операций, необходимо использовать алгоритмы, которые приводили бы к быстродействующим комбинационным схемам и требовали небольшого числа циклов.

Далее, быстродействие вычислительных систем может быть повышено за счет реализации аппаратными или программно-аппаратными средствами встроенных сложных команд, соответствующих тем или иным функциям, встречающимся во многих практических вычислениях. К таким функциям относятся, например, квадратный корень, сложение векторов, умножение матриц и быстрое преобразование Фурье. Это позволило бы сократить число команд в программах и более эффективного использовать машинные ресурсы.

Еще один резерв, используемый для повышения эффективности работы процессора, - это сокращение временных затрат при обращении к памяти. Обычные подходы здесь состоят, во-первых, в расширении путей доступа за счет разбиения памяти на модули, обращение к которым может осуществляться одновременно; во-вторых, в применении дополнительной сверхбыстродействующей памяти (кэш-памяти) и, наконец, в увеличении числа внутренних регистров в процессоре.

Следующий шаг на пути повышения быстродействия - это уменьшение длительности исполнения одной команды, за счет временного перекрытия различных ее фаз. Этот подход требует дополнительного оборудования.

На уровне структуры алгоритма, по которому работает система, основной подход к повышению быстродействия состоит в том, чтобы выполнять одновременно несколько команд. Распараллеливание позволяет значительно увеличить производительность систем при решении широкого класса задач.

Перечисленные подходы касаются аппаратуры, логической организации и архитектуры систем. Усилия, затрачиваемые в этих областях, имеют своей целью обеспечение необходимого ускорения вычислений на программно-алгоритмическом уровне. На этом уровне должны использоваться либо специальные языки программирования, предоставляющие средства для явного описания параллелизма, либо методы выявления параллелизма в последовательных программах. Кроме того, сам алгоритм должен обладать внут-

ренним параллелизмом, соответствующим особенностям архитектуры. Использование неадекватных алгоритмов и языков способно, практически, свести на нет возможности для реализации высокоскоростных вычислений, заложенные в архитектуре.

2. История создания высокопроизводительных систем

Вот лишь краткая история создания высокопроизводительных систем, использую принцип распараллеливания обработки.

Зарубежная вычислительная техника.

IBM 701 (1953), **IBM 704** (1955): *разрядно-параллельная память, разрядно-параллельная арифметика***.** Все самые первые компьютеры (EDSAC, EDVAC, UNIVAC) имели разрядно-последовательную память, из которой слова считывались последовательно бит за битом. Первым коммерчески доступным компьютером, использующим разрядно-параллельную память (на CRT) и разрядно-параллельную арифметику, стал IBM 701, а наибольшую популярность получила модель IBM 704 (продано 150 экз.), в которой, помимо сказанного, была впервые применена память на ферритовых сердечниках и аппаратное АУ с плавающей точкой.

IBM 709 (1958): *независимые процессоры ввода/вывода*. Процессоры первых компьютеров сами управляли вводом/выводом. Однако скорость работы самого быстрого внешнего устройства, а то тем временам это магнитная лента, была в 1000 раз меньше скорости процессора, поэтому во время операций ввода/вывода процессор фактически простаивал. В 1958г. к компьютеру IBM 704 присоединили 6 независимых процессоров ввода/вывода, которые после получения команд могли работать параллельно с основным процессором, а сам компьютер переименовали в IBM 709. Данная модель получилась удивительно удачной, так как вместе с модификациями было продано около 400 экземпляров, причем последний был выключен в 1975 году - 20 лет существования!

IBM STRETCH (1961): *опережающий просмотр вперед, расслоение памяти*. В 1956 году IBM подписывает контракт с Лос-Аламосской научной лабораторией на разработку компьютера STRETCH, имеющего две принципиально важные особенности: опережающий просмотр вперед для выборки команд и расслоение памяти на два банка для согласования низкой скорости выборки из памяти и скорости выполнения операций.

ATLAS (1963): *конвейер команд***.** Впервые конвейерный принцип выполнения команд был использован в машине ATLAS, разработанной в Манчестерском университете. Выполнение команд разбито на 4 стадии: выборка команды, вычисление адреса операнда, выборка операнда и выполнение операции. Конвейеризация позволила уменьшить время выполнения команд с 6 мкс до 1,6 мкс. Данный компьютер оказал огромное влияние, как на архитектуру ЭВМ, так и на программное обеспечение: в нем впервые использована мультипрограммная ОС, основанная на использовании виртуальной памяти и системы прерываний.

CDC 6600 (1964): *независимые функциональные устройства*. Фирма Control Data Corporation (CDC) при непосредственном участии одного из ее основателей, Сеймура Р.Крэя (Seymour R.Cray) выпускает компьютер CDC-6600 - первый компьютер, в котором использовалось несколько независимых функциональных устройств. Для сравнения с сегодняшним днем приведем некоторые параметры компьютера:

- время такта 100нс,
- производительность 2-3 млн. операций в секунду,
- оперативная память разбита на 32 банка по 4096 60-ти разрядных слов,
- цикл памяти 1мкс,
- 10 независимых функциональных устройств.

Машина имела громадный успех на научном рынке, активно вытесняя машины фирмы IBM.

CDC 7600 (1969): *конвейерные независимые функциональные устройства*. CDC выпускает компьютер CDC-7600 с восемью независимыми конвейерными функциональные устройствами - сочетание параллельной и конвейерной обработки. Основные параметры:

- такт 27,5 нс,
- 10-15 млн. опер/сек.,
- 8 конвейерных ФУ,
- 2-х уровневая память.

ILLIAC IV (1974): *матричные процессоры***.**

Проект: 256 процессорных элементов (ПЭ) = 4 квадранта по 64ПЭ, возможность реконфигурации: 2 квадранта по 128ПЭ или 1 квадрант из 256ПЭ, такт 40нс, производительность 1Гфлоп;

- работы начаты в 1967 году, к концу 1971 изготовлена система из 1 квадранта, в 1974г. она введена в эксплуатацию, доводка велась до 1975 года;
- центральная часть: устройство управления (УУ) + матрица из 64 ПЭ;
- УУ это простая ЭВМ с небольшой производительностью, управляющая матрицей ПЭ; все ПЭ матрицы работали в синхронном режиме, выполняя в каждый момент времени одну и ту же команду, поступившую от УУ, но над своими данными;
- ПЭ имел собственное АЛУ с полным набором команд, ОП - 2Кслова по 64 разряда, цикл памяти 350нс, каждый ПЭ имел непосредственный доступ только к своей ОП;
- сеть пересылки данных: двумерный тор со сдвигом на 1 по границе по горизонтали.

Рис.2.1. Система Cray-1.

Несмотря на результат в сравнении с проектом: стоимость в 4 раза выше, сделан лишь 1 квадрант, такт 80нс, реальная производительность до 50Мфлоп - данный проект оказал огромное влияние на архитектуру последующих машин, построенных по схожему принципу, в частности: PEPE, BSP, ICL DAP.

Cray 1 (1976): ***векторно-конвейерные процессоры.*** В 1972 году С. Крэй покидает CDC и основывает свою компанию Cray Research, которая в 1976 году выпускает первый векторно-конвейерный компьютер Cray-1 (рис.2.1.): время такта 12.5нс, 12 конвейерных функциональных устройств, пиковая производительность 160 миллионов операций в секунду, оперативная память до 1Мслова (слово-64 разряда), цикл памяти 50нс. Главным новшеством является введение векторных команд, работающих с целыми массивами независимых данных и позволяющих эффективно использовать конвейерные функциональные устройства.

Отечественные высокопроизводительные системы.

БЭСМ-6. Эта машина занимает особое место в истории развития отечественной вычислительной техники. Это была исключительно удачная оригинальная разработка. В 1967 г. запущенная в серию, она стала первой в СССР вычислительной машиной класса супер-ЭВМ с огромной по тем временам производительностью в 1 млн операций в секунду. Новые принципы, заложенные в ее архитектуру, структурную организацию и математическое обеспечение, в значительной степени повлияли на создаваемые позднее вычислительные комплексы следующих поколений.

БЭСМ-6 была построена на элементной базе транзисторных переключателей тока, диодно-резисторной комбинаторной логики и ферритовой памяти. Использовалась высокочастотная система элементов (впервые в СССР была достигнута тактовая частота 10

МГц) и компактная конструкция с короткими связями между блоками (внутренний монтаж в стойке с двусторонним расположением блоков).

Конвейерный принцип организации управления ("водопроводный", как называл его С. А. Лебедев) позволял совмещать до 14 машинных команд, находящихся на разных стадиях выполнения. При этом работа модулей оперативной памяти, устройства управления и арифметико-логического устройства осуществлялась параллельно и асинхронно благодаря наличию буферных устройств промежуточного хранения команд и данных.

Для ускорения конвейерного выполнения команд в устройстве управления были предусмотрены регистровая память хранения индексов и модуль адресной арифметики.

Ассоциативная память на быстрых регистрах (типа cache) позволяла автоматически сохранять в ней наиболее часто используемые операнды и тем самым сократить число обращений к оперативной памяти.

"Расслоение" оперативной памяти обеспечивало возможность одновременного обращения к нескольким ее модулям из различных устройств машины.

В арифметико-логическом устройстве были реализованы ускоренные алгоритмы умножения и деления (умножение на четыре цифры множителя, вычисление четырех цифр частного за один такт синхронизации), а также сумматор без цепей сквозного переноса, представляющий результат операции в виде двухрядного кода (поразрядных сумм и переносов) и оперирующий с входным трехрядным кодом (новый операнд и двухрядный результат предыдущей операции).

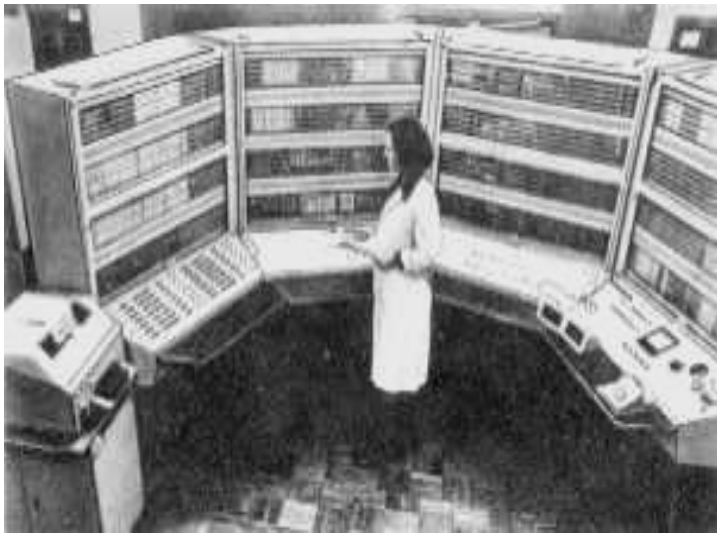


Рис.2.2. Отечественная ЭВМ БЭСМ-6.

В БЭСМ-6 были предусмотрены:

- развитая система прерываний;
- страничная организация памяти с аппаратным преобразованием математических (виртуальных) адресов в физические адреса (механизм "приписки");
- аппаратные механизмы защиты памяти, что обеспечивало возможность организации мультипрограммного режима работы;
- динамическое распределение памяти в процессе вычислений средствами операционной системы.

Основные технические характеристики машины БЭСМ-6 таковы:

- Быстродействие - около 1 млн. операций/сек.;
- объем ОЗУ - от 32 до 128 тысяч машинных слов;
- время выполнения сложения с плавающей запятой - 1,1 мксек;
- время умножения - 1,9 мксек;
- время деления - 4,9 мксек;
- время выполнения логических поразрядных операций - 0,5 мксек.
- Работа арифметического устройства совмещена с выборкой операндов из памяти.
- Разрядность машинного слова - 48 двоичных разрядов.
- Объем промежуточной памяти на магнитных барабанах - 512 тысяч слов.

Основные участники разработки БЭСМ-6 - С. А. Лебедев (главный конструктор), В. А. Мельников, Л. Н. Королев (заместители главного конструктора), В. П. Иванников и Э. З. Любимский - разработчики программного обеспечения БЭСМ-6. При создании БЭСМ-6 использовались основные принципы системы автоматизации проектирования ЭВМ

(САПР). Компактная запись схем машины формулами Булевой алгебры явилась основой ее эксплуатационной и наладочной документации. Документация для монтажа выдавалась на завод в виде таблиц, полученных на инструментальной ЭВМ.

БЭСМ-6 серийно выпускалась Московским заводом счетно-аналитических машин (САМ) с 1967 г. **в течение 17 лет**. Она отличалась надежностью в работе, ею оснащались основные вычислительные центры СССР.

За более чем 25-летнюю эксплуатацию БЭСМ-6 и ее преемников было накоплено огромное и чрезвычайно важное по значимости математическое обеспечение. Без сомнения, эта разработка стала вершиной отечественной вычислительной техники.

Эльбрус-1 (1977). Вычислительный комплекс со средствами аппаратной поддержки развитой структуризации программ и данных (В.С.Бурцев, Б.А.Бабаян).

ПС 2000 и ПС 3000 (1979). Начало серийного выпуска высокопроизводительных многопроцессорных универсальных вычислительных комплексов (УВК) с перестраиваемой структурой ПС 2000 и ПС 3000, 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.

Эльбрус-2 (1985). Начало выпуска многопроцессорного вычислительного комплекса Эльбрус-2 производительностью при 10 процессорах до 125 млн. оп/сек (MIPS). Разработчики: В.С. Бурцев и др..

Эльбрус 3.1 (1990). Выпуск и ввод в эксплуатацию векторно-конвейерной супер-ЭВМ Эльбрус 3.1 на базе модульных конвейерных процессоров разработки ИТМ и ВТ имени С.А. Лебедева. Разработчики: Г.Г. Рябов, А.А. Соколов, А.Ю. Бяков. Производительность в однопроцессорном варианте – 400 MFLOPS

3. Общие принципы организации высокопроизводительных систем

Эффективность использования различных вариантов организации систем связана со структурами вычислительных алгоритмов. Алгоритм в данном случае представляет собой описание вычислительного процесса в виде последовательности более простых вычислительных конструкций. Он включает спецификации (описания) этих вычислительных конструкций и отношений следования между ними. Конструкции, образующие алгоритм, могут быть различными по степени сложности: от простых арифметических операторов до целых программ. Принято выделять несколько **уровней** параллелизма вычислительных конструкций:

1. Параллелизм заданий - каждый процессор загружается своей собственной независимой от других вычислительной задачей. Параллелизм такого типа представляет интерес скорее для системных администраторов, чем рядовых пользователей.

2. Параллелизм на уровне программы - вычислительная программа разбивается на несколько достаточно больших частей, которые могут выполняться одновременно на различных процессорах (подпрограммы, независимые ветви программ).

3. Параллелизм циклов и итераций – в большинстве случаев это векторная обработка, иногда (при независимости по данным последующих шагов) – параллельная обработка.

4. Параллелизм команд - обычно реализован на низком уровне, например внутри процессора (конвейеры команд и т.д.).

5. Параллелизм на уровне машинных слов и арифметических операций - в некоторых ситуациях (например, сложение двух операндов) выполняется одновременным сложением всех их двоичных разрядов.

Вложенность этих уровней определяет глубину распараллеливания и является одним из важнейших свойств при анализе параллельных вычислений. Поскольку концепция ал-

горитма подразумевает иерархию (любая вычислительная конструкция в свою очередь может быть описана в виде алгоритма, состоящего из еще более «простых» вычислительных конструкций) – это приводит к понятию *детализации* параллелизма. Детализацию называют *мелкой*, если вычислительные конструкции алгоритма являются примитивными (т.е. реализуемыми одной командой), и *крупной*, если эти конструкции являются сложными (т.е. реализуются с помощью конструкций более низкого уровня). Спектр значений детализации простирается от очень мелкой до очень крупной. Соответственно вычислительные системы могут обладать мелко-, средне- или крупноблочной структурой.

Алгоритмические структуры отличаются друг от друга содержанием простых вычислительных конструкций и типом отношения следования. Рассмотрим четыре основных типа отношений следования вычислительных конструкций и соответствующие им типы логической организации систем.

3.1. Последовательная организация

В обычной последовательной алгоритмической структуре (рис.3.1.*а*) в каждый момент времени исполняется только одна команда. Соответствующая вычислительная машина проста: в ней имеется единственная память для хранения данных и программ, одно арифметическое устройство, исполняющее текущую команду, одно устройство управления, осуществляющее контроль за исполнением, счетчик команд, хранящий адрес текущей команды, и простой механизм его модификации. Взаимные связи между перечисленными компонентами также просты. Быстродействие системы ограничено ее последовательной сущностью и определяется временем исполнения каждой команды.

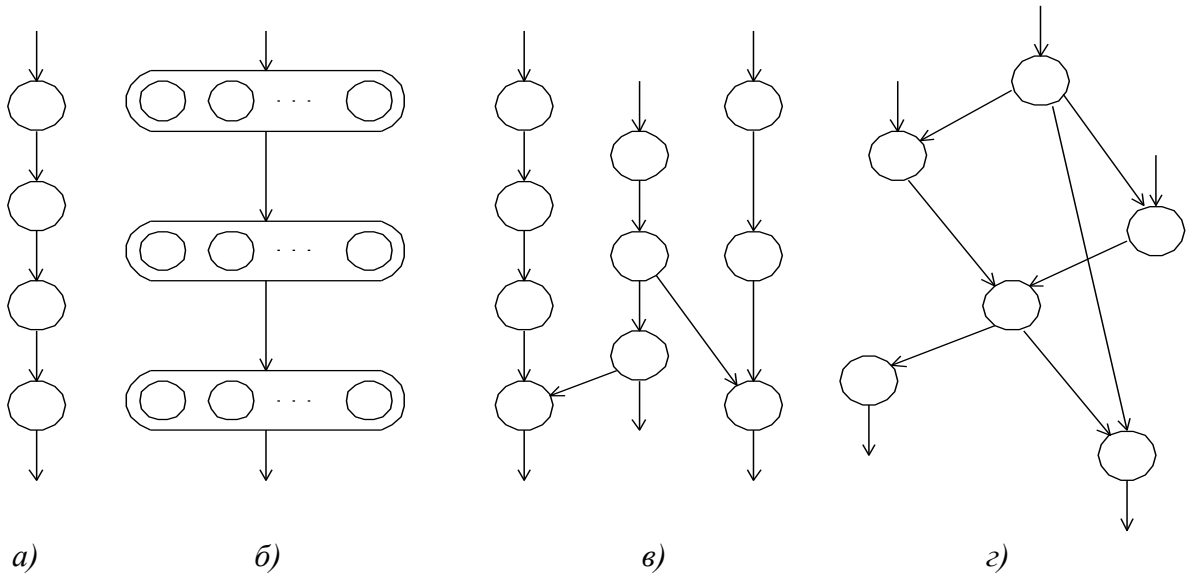


Рис.3.1. Алгоритмические структуры:
а - последовательная; *б* - последовательно групповая; *в* - совокупность слабосвязанных потоков; *г* - параллельная структура общего вида.

Важное преимущество последовательного подхода состоит в том, что для машин данного типа разработано огромное количество алгоритмов и программ, накоплен богатый опыт программирования, разработаны фундаментальные языки и технологии программирования. По этой причине дальнейшее совершенствование организации шло так, что последовательная структура сохранялась, однако к ней добавлялись механизмы, позволяющие исполнять несколько команд одновременно. Все эти механизмы так или иначе сводятся к общему принципу *опережающего просмотра команд*, поскольку для того, что-

бы иметь возможность одновременно исполнять несколько команд, процессор должен просматривать поток команд на несколько шагов вперед и находить те команды, которые допускают одновременное выполнение. Выявление параллелизма в этом случае сводится к тому, чтобы исключить те зависимости, которые привнесены в алгоритм в процессе его последовательной записи, и сохранить только те зависимости, которые обусловлены самой природой алгоритма. Такие зависимости бывают двух типов: *зависимости по данным* (когда команда не может исполниться до тех пор, пока не подготовлены все операнды, которые являются результатами исполнения других команд) и *зависимости по управлению* (когда условный переход не может быть осуществлен до тех пор, пока не вычислено условное выражение).

Для одновременного выполнения команд, не являющихся зависимыми, существуют две дополнительные возможности: использование нескольких процессоров (по числу команд) и применение одного процессора конвейерного типа. Конвейеризация обработки - это общий метод повышения пропускной способности систем, выполняющих повторяющиеся операции. Основа этого подхода состоит в том, что система может быть разделена на ступени обрабатывающих устройств, позволяющих начинать исполнение новой команды прежде, чем будет завершена предыдущая. В такой системе пропускная способность определяется временем прохождения самого медленного звена. Данный подход может быть применен, например, к обработке команд, поскольку процесс исполнения команды состоит из нескольких стадий, таких как вызов команды, расшифровка, вычисление адреса, вызов операнда и др.

Факторы, от которых зависит ускорение вычислений за счет опережающего просмотра, следующие:

- степень параллелизма, заключенного в программе. В типовых программах число зависимостей по данным невелико. Это создает широкие предпосылки для использования параллелизма. Но с другой стороны, число зависимостей по управлению, соответствующих условным переходам, весьма значительно, и оно накладывает ограничения но возможность использования параллелизма. В некоторых вычислительных машинах имеется механизм, который не останавливается на условных переходах, а продолжает опережающий просмотр по обеим ветвям или по предпочтительной ветви, определяемой с помощью механизма прогнозирования перехода. Этот метод требует сложного дополнительного оборудования в устройстве управления, и в то же время имеет ограниченный эффект из-за высокой вероятности того, что еще до завершения предыдущего ветвления появится новая ветвь.

- способность устройства управления обнаруживать зависимости. Если поставлена задача обнаружить все или почти все операции, допускающие одновременное исполнение, то устройство управления получается довольно сложным.

- наличие конвейерного процессора и/или набора функциональных устройств, предназначенных для одновременного исполнения нескольких команд.

- наличие памяти с широким трактом доступа, способным обеспечить поставку команд и данных процессору в таком темпе, чтобы он оказывался постоянно активным. Поскольку быстродействие памяти обычно ниже, чем процессора, то для организации широкого тракта доступа необходимо разбить память на блоки, обращение к которым может происходить одновременно. При этом максимальная пропускная способность тракта, связывающего процессор с памятью, зависит от быстродействия блоков и их общего количества. Однако, поскольку возможны конфликты по доступу, пропускная способность оказывается зависящей и от других факторов, таких как механизм адресации, адресный поток, а также возможность приоритетного обслуживания запросов.

Еще один подход к уменьшению отрицательного влияния на производительность зависимостей по данным и по управлению состоит в том, что процессор работает как бы в мультиплексном (по времени) режиме, обслуживая несколько процессов. Этот принцип

похож на используемый в мультипрограммировании, с той лишь разницей, что в данном случае мультиплексирование осуществляется с квантом, равным времени выполнения одной команды.

3.2. Последовательно групповая организация

В последовательно групповой структуре вычислительные конструкции, образующие алгоритм, объединены в группы, а отношение следования состоит в том, что конструкции внутри группы могут выполняться одновременно, а сами группы последовательно (рис. 2.1.*б*). Вычислительные конструкции внутри одной группы могут быть одинаковыми или разными. Реализация таких алгоритмов на высокопроизводительных скалярных процессорах с помощью обычных циклов сопровождается рядом факторов, ограничивающих максимальную скорость вычислений:

- перед каждой скалярной операцией необходимо вызвать и декодировать скалярную команду;
- для каждой команды необходимо вычислять адреса элементов данных;
- данные должны вызываться из памяти, а результаты запоминаться в памяти;
- необходимо осуществлять упорядочение выполнения операций в функциональных устройствах, которые в целях увеличения производительности строятся по конвейерному принципу;
- реализация команд построения циклов (счетчик и переход) сопровождается накладными расходами.

Влияние этих факторов уменьшается при введении *векторных команд,* с помощью которых задается одна и та же операция над элементами одного или нескольких векторов, и организации системы, которая обеспечивает эффективное исполнение таких команд. Этот подход реализуется в системах двух типов: **матричных и векторно-конвейерных.** В векторно-конвейерных системах вычислительные конструкции внутри групп исполняются конвейерным процессором, в то время как в матричных системах для этого используются несколько одновременно работающих процессоров.

Оба подхода позволяют достичь значительного ускорения по сравнению со скалярными машинами. Более того, ускорение в системах матричного типа может быть больше, чем в конвейерных, поскольку увеличить число процессорных элементов проще, чем число ступеней в конвейерном устройстве.

Для современного уровня технологии векторно-конвейерные системы являются более гибкими и эффективными с точки зрения стоимости.

3.2.1 Векторно-конвейерная обработка

В самом широком смысле векторизация - это преобразование операций, выполняемых по ходу процесса решения задачи, из скалярной традиционной формы в векторную.

Наиболее общим типом векторизации является *синтаксическая векторизация*, т.е. такой способ преобразования последовательности команд, при котором не учитывается смысловое значение команд и данных, а учитывается только из форма. Самая сильная сторона этого типа одновременно является и его самой слабой. С одной стороны, поскольку учитывается только форма представления программ и данных, имеется теоретическая возможность реализовать процесс векторизации с помощью специальных программных средств - так называемых автоматических векторизаторов. Но с другой стороны, что же делать, если эти программные векторизаторы не найдут формального преобразования, позволяющего превратить некоторую программно-информационную структуру в векторную?

Для того, чтобы избежать данной ситуации, необходимо обратиться к другому подходу - *семантической векторизации*. Семантическая векторизация подразумевает интерпретацию содержащихся в программах структур данных и подстановку векторных конструкций, выполняемых с учетом смыслового содержания задачи, а не ее формы.

В широком смысле существуют два препятствия на пути векторизации. Во-первых, структура данных, над которой должна быть произведена та или иная операция, может и не быть вектором. Это может происходить по разным причинам, начиная от постоянного присутствия «небольших» дыр в структурах данных до сильно разреженных структур. Второе препятствие - это рекурсивные вычисления. Ключевое условие реализации векторной или параллельной обработки состоит в том, чтобы элементы, участвующие в одних и тех же операциях, не взаимодействовали между собой при исполнении этих операций. Рекурсивные отношения делают это невозможным.

Таким образом, с точки зрения архитектуры вычислительных систем векторная обработка представляет собой совершенно особый эволюционный этап в развитии параллельных систем и вычислений, и соответственно, требует оборудования, обладающего большими логическими возможностями.

Векторно-конвейерная вычислительная система организована так, как показано на рис.3.2. В такой системе имеется один (или небольшое количество) процессор, в котором обработка производится по принципу конвейера.

Рис.3.2. Структура векторной вычислительной системы.

Процессор выполняет векторные команды путем засылки элементов векторов в конвейер с интервалом, равным длительности прохождения одной стадии обработки. Результаты операций в виде потока данных пересылаются в память. Управление доступом к

данным, находящимся в памяти, осуществляется адресным генератором, а параметры векторных операций устанавливаются векторным устройством управления.

В ряде систем такого типа используется широкий набор регистров для хранения векторов, что позволяет уменьшить число обращений к памяти. Память делится на главную и память второго уровня, что обеспечивает необходимую емкость и пропускную способность при приемлемой скорости. Векторные системы имеют в своем составе также скалярный процессор, исполняющий скалярные операции, причем для того чтобы обеспечить наиболее полную загрузку конвейерного процессора, скалярные операции выполняются параллельно с векторными.

На рис.3.3 изображены этапы, которые проходит векторная команда. Наиболее простой путь уменьшения времени выполнения команд - это повышение быстродействия элементной базы. Однако во многих случая этого недостаточно. Дальнейшее увеличение пропускной способности может быть достигнуто за счет применения многоуровневой конвейеризации и параллельной обработки на всех стадиях, которые проходит команда в процессе выполнения. Как уже было отмечено, одним из главных преимуществ векторной обработки является то, что она очень эффективно осуществляется в конвейерном режиме.

Рис.3.3. Обработка векторной команды.

На *первом уровне* исполнение команды делится на две стадии (ступени): *подготовка* и *исполнение* (рис.3.4). При наличии такого конвейера одна команда исполняется, а другая готовится к исполнению.

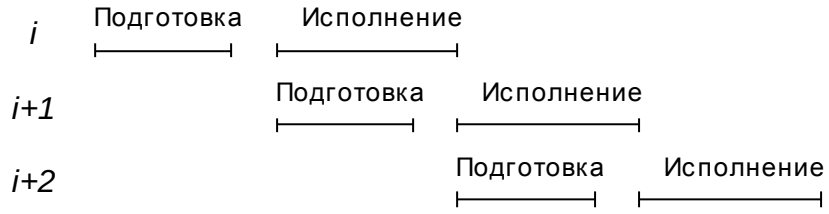


Рис.3.4. Конвейеризация I уровня.

Рис.3.5. Конвейеризация II уровня:
а - устройство подготовки команд; *б* -устройство исполнения команд.

Поскольку обе эти стадии (ступени), в свою очередь делятся на более мелкие операции, устройства подготовки и исполнения также могут быть конвейеризованы, образуя *второй уровень* конвейеризации (рис.3.5). Обычно устройство подготовки включает следующий конвейер: вызов, декодирование и засылку команды в исполнительное устройство, которое также может быть конвейеризовано: вызов операндов, исполнение операции и засылка результатов.

Некоторые из перечисленных операций являются достаточно сложными и требуют дальнейшего разбиения. Так образуется *третий уровень* конвейеризации. Особенно важна конвейеризация доступа к памяти и арифметических операций (рис.3.6).

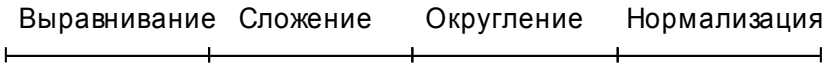


Рис.3.6. Конвейеризация III уровня: арифметическое устройство.

Пропускная способность конвейера определяется временем прохождения самой медленной стадии. Следовательно, при разработке конструкции необходимо обеспечить равенство скоростей обработки на всех стадиях. В сложных конвейерных устройствах, данное условие нереализуемо. В таких случаях для повышения пропускной способности применяются два способа:

- межстадийная буферизация и
- параллельное дублирование медленных ступеней конвейера.

Метод буферизации предназначен для тех машин, в которых времена задержек на каждой ступени различны. Благодаря данному методу пропускная способность конвейера определяется не максимальной, а средней задержкой (рис.3.7). Буферизация оказывается полезной во многих местах конвейера, но особенно - в интерфейсе памяти, поскольку, во-первых, ее быстродействие относительно невелико, а, во-вторых, запросы к памяти могут носить разный характер (вызов команды, формирование адреса, вызов операндов, запись результатов). Применение метода дублирования медленных стадий (к которым относятся, например, стадия обращения к памяти и стадия выполнения операций в устройстве обработки) позволяет увеличить пропускную способность. Так, память обычно выполняется в виде набора модулей, а для разных арифметических операций используются отдельные устройства (универсальные или специализированные).

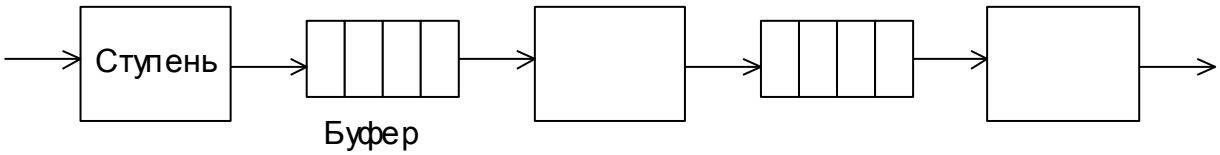


Рис.3.7. Конвейер с буферизацией между ступенями.

Основным показателем производительности такой идеализированной векторной машины является *максимальная пропускная способность* (R_{max}). Она равная пропускной способности самой медленной ступени, которая в свою очередь, определяется как результат деления числа однотипных устройств этой ступени на величину времени прохождения:

$$R_{\max} = \left(\frac{M}{t_s} \right)_{\min}$$

Главная задача при создании процессора состоит в получении максимально высокой пропускной способности. Для этого необходимо обеспечить минимальные времена прохождения всех стадий конвейера, а там, где это возможно, применить параллельно-последовательную аппаратуру.

Ввиду сложного характера взаимодействия между факторами, снижающими производительность, и зависимостью степени влияния этих факторов от конкретных программ оценка производительности в целом должна осуществляться на специально подобранных прикладных задачах. Приближенная оценка может быть получена с помощью контрольных задач, состоящих из набора типовых тестовых программ. Проблема, связанная с такой приближенной оценкой, заключается в трудности определения термина «типовой». Искусственные контрольные задачи, содержащие в нужном соотношении скалярные операции и векторные команды различных типов, могут дать лишь самые грубые оценки. Недостаток данного метода состоит в том, что он не учитывает взаимосвязи между командами, роль которых в формировании реального показателя производительность может быть значительной.

Различные типы организации векторно-конвейерных систем. Как уже было отмечено, все векторно-конвейерные ЭВМ строятся на одних и тех же организационных принципах, однако между ними существует целый ряд различий. Рассмотрим альтернативные варианты в организации таких машин в более систематизированной форме.

1. Арифметические конвейерные устройства.

Условием достижения высокой производительности арифметических устройств является из организация по конвейерному принципу. Арифметические конвейерные устройства могут быть *однофункциональными* или *многофункциональными* в зависимости от числа типов операций, которые они выполняют. Однофункциональные устройства имеют более простую конструкцию и могут быть оптимально приспособлены для выполнения своей единственной операции. В то же время многофункциональные устройства отличаются большей гибкостью и более выгодны в стоимостном отношении.

Многофункциональные конвейерные устройства должны обладать способностью к перенастройке. Перенастройка может быть *статической* или *динамической*. В первом случае конвейер заранее настраивается на выполнение определенной операции в рамках конкретной задачи и при необходимости может быть на короткое время, но заблаговременно перенастроен на выполнение другой операции. Во втором случае конвейер может обрабатывать поток данных, которым могут соответствовать различные операции. Конвейерные устройства с динамической перенастройкой имеют гораздо более сложную конструкцию и схему управления, вследствие чего их применение в векторных системах по соображениям стоимостной эффективности, нецелесообразно.

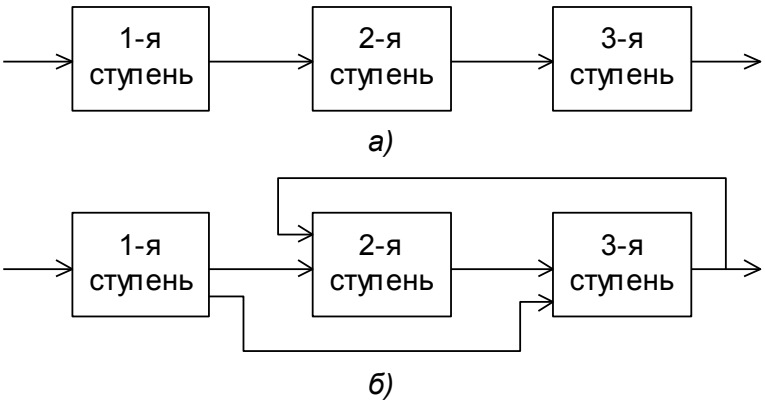


Рис.3.8. Типы конвейерных устройств:
а - линейный конвейер; *б* - нелинейный конвейер.

Конвейерные устройства могут быть *линейными* или *нелинейными.* В линейных конвейерах данные могут следовать строго по порядку ступеней: от ступени i к ступени i+1, как показано на рис.3.8. В нелинейных устройствах допускаются как *опережающие,* так и *обратные* связи. Линейные конвейерные устройства отличаются простотой конструкции и управления, однако нелинейные конвейеры позволяют исполнять, например, рекурсивные операции или вычислять скалярное произведение.

Число конвейерных устройств, типы операций, которые они должны выполнять, число ступеней конвейеров и время прохождения каждой ступени - это те параметры, выбор которых составляет основную задачу конструирования вычислительных систем.

2. Локальная память.

Локальная память играет роль буфера между основной памятью и процессором. Для связи локальной памяти с основной памятью и процессором требуется несколько информационных шин: одна или две для операндов, поступающих из основной памяти, одна для записи результатов в память, две для подачи операндов в конвейерное обрабатывающее устройство (или устройства если их несколько) и одна для приема результатов из конвейера. Для обеспечения максимальной пропускной способности некоторые из названных шин должны иметь работать параллельно друг другу (рис.3.9).

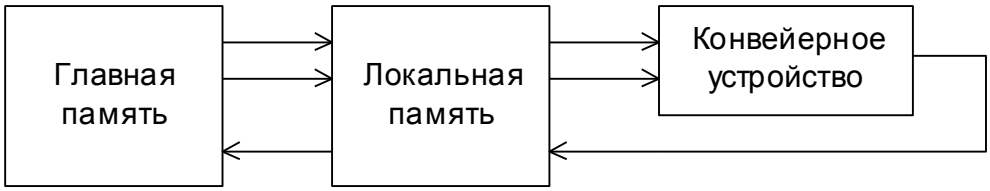


Рис.3.9. Информационные связи локальной памяти.

Следовательно, локальная память не может быть конструктивно выполнена в виде одного блока обычной памяти с произвольным доступом или в виде регистрового файла, поскольку расширение количества одновременно выполняемых указанными устройствами операций потребовало бы чрезмерного усложнения их конструкции.

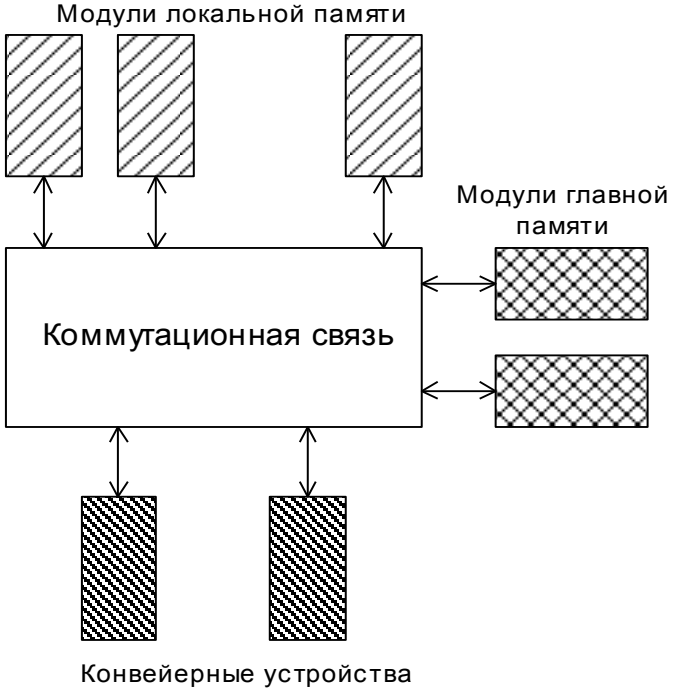


Рис.3.10. Схема подключения локальной памяти

Вот почему локальная память должна быть разбита на ряд блоков, которые соединены с главной памятью и конвейерными устройствами через коммутационную сеть (рис.3.10). Один из возможных путей организации взаимодействия, который приводит к упрощению коммутационной сети, состоит в том, чтобы иметь для каждого конвейерного устройства отдельный буфер. Но тогда возникают ограничения при использовании результатов одного конвейера в качестве операндов другого конвейера. Кроме того, локальная память используется не оптимально, поскольку не является общим ресурсом для всех конвейеров.

Как уже было сказано, локальная память может быть невидимой для программиста, как в векторных процессорах, в которых команды исполняются в режиме «память - память», или, наоборот, видимой, как в системах, в которых команды исполняются в режиме «регистр - регистр». Кроме того, гибкость в использовании регистров можно увеличить, если сделать число элементов этих регистров переменным. Таким образом, можно сделать вывод, что размеры и структура локальной памяти существенным и весьма сложным образом влияют на производительность процессора.

3. Адресное устройство.

Адресное устройство генерирует адреса главной памяти, по которым осуществляет доступ к элементам векторов. В простейшем и наиболее распространенном случае адреса элементов образуют естественную последовательность. В некоторых векторных системах организация памяти такова, что запись и считывание осуществляется целыми блоками, состоящими из последовательно расположенных слоев (такой блок иногда называют «суперсловом»). В этом случае адресное устройство должно генерировать единственный адрес. Применяются также схемы адресации, в которых элементы располагаются на некотором постоянном расстоянии друг от друга. В этих случаях генерация адресов производится с использованием операций сложения, выделения подматриц, состоящей в формировании комбинации последовательных и равноотстоящих адресов и произвольного доступа.

4. Главная память.

Количество модулей главной памяти, на которые одна должна быть разбита в целях обеспечения необходимой пропускной способности, зависит от быстродействия каждого их них и заданной пропускной способности. При определении этого количества необходимо учитывать возможные задержки, связанные с конфликтами доступа при работе с векторами, и взаимные помехи, создаваемые разными портами друг другу. Существуют три типовые схемы организации памяти: с простым, сложным и смешанным расслоением. Кроме того, поскольку обработка скалярных величин оказывает существенное влияние на общий показатель производительности, важно, чтобы время доступа к модулю памяти при работе со скалярами было возможно меньшим. Важное значение имеет и объем памяти, поскольку решение многих прикладных задач связано с обработкой больших массивов данных.

5. Система команд.

Главной отличительной чертой векторно-конвейерных систем является наличие в их «репертуаре» векторных команд. С помощью этих команд задаются операции над полными векторами. Эти операции заменяют обычные циклы, широко используемые в скалярных вычисления.

Векторная вычислительная система обычно имеет широкий набор разнообразных команд. Кроме обычных скалярных команд этот набор включает векторные команды и специальные команды, предназначенные для управления векторными операциями.

Типовой набор векторных команд включает: арифметические операции, операции сравнения, команды загрузки и записи в память, а также команды для обработки разреженных векторов. Скалярные команды используются для формирования адресов элементов и контроля длин векторов. В некоторых векторных ЭВМ наиболее часто встречающи-

еся в программах операции, такие как скалярное произведение, вычисление среднего значения и ряд других, реализованы в виде *векторных макрокоманд*. Это позволяет использовать одну команду вместо целой подпрограммы.

Так же как и скалярная, векторная команда должна включать: код операции; адреса и длины векторных операндов и результата; статус процессора при записи; адрес следующей исполняемой команды. С помощью *кода операции* производится выбор одной из заранее составленных таблиц, на основании которой осуществляется управление конвейерным функциональным устройством и производятся необходимые для начального запуска этого устройства действия. *Операнды (результаты)* находятся в памяти верхнего уровня или в регистрах. Если операнды хранятся в памяти верхнего уровня, то в векторной команде должны быть заданы: базовый адрес вектора, число элементов, тип данных в каждом элементе, схема хранения вектора в памяти.

Поскольку элементы векторов иногда располагаются по адресам, не образующим непрерывной последовательности, процедура формирования адресов обращений к этим элементам может оказаться весьма сложной. Для операндов, хранящихся в регистрах, векторная команда задает номер регистра, число элементов, тип данных и еще некоторую информацию, необходимую для организации доступа.

Для векторных команд в большей степени, чем для скалярных, характерны такие понятия, как *состояние* и *обработка особых ситуаций.* Поскольку в результате выполнения операции вырабатывается много величин, то с практической точки зрения невозможно и нежелательно держать состояния, т.е. признаки выполнения индивидуальных условий, в составе самой команды. Вместо этого вводится специальная векторная команда, с помощью которой состояния вырабатываются и записываются в явном виде в формате битовых векторов. Например, векторная команда сравнения на равенство VCOMPEQ (X, Y, Z) вырабатывает битовый вектор Z по следующему правилу:

$$Z(I) = \begin{cases} 1, & \text{если } X(I) = Y(I) \\ 0, & \text{если } X(I) \neq Y(I) \end{cases}$$

Битовый вектор, сформированный в результате операции сравнения, используется для управления векторными командами.

Обработка особых ситуаций (ошибок) в векторных командах затруднена из-за множественности операций. Обычно при появлении ошибки исполнение векторной команды либо прекращается, либо фиксируется сообщение об ошибке, а выполнение команды продолжается.

Поскольку векторные процессоры работают в соответствии с последовательно-групповой моделью вычислений, очередность команд устанавливается так же, как и в обычных процессорах: счетчик команд указывает на следующую команду после получения очередного приращения или, в случае команды перехода, - после загрузки в этот счетчик начального адреса новой ветви.

6. Хранение и адресация векторов.

Векторные операнды хранятся в памяти таким образом, при котором уменьшаются требования к объему памяти и обеспечивается эффективный доступ к различным векторным подструктурам. Схема размещения данных в памяти зависит от требований, предъявляемых к подструктурам, возможностей механизма адресации и организации памяти.

Существует два класса схем размещения данных в памяти, соответствующих двум типам векторных структур: плотным и разреженным. В первом случае хранятся все элементы векторов, причем последовательно идущие элементы занимают, как правило, последовательно расположенные ячейки памяти. Во втором случае хранятся только ненулевые элементы и вектор соответствующих индексов или заменяющий его битовый вектор.

При работе с плотными массивами используются следующие схемы распределения данных в памяти:

- последовательная (т.е. хранение элементов векторов в последовательно расположенных ячейках);
- с регулярной структурой (т.е. столбцовые элементы матриц хранятся по строкам);
- в виде подматриц (рисунок 2.11.*в*);
- в произвольном порядке (эта ситуация возникает, например, при исполнении программ, содержащих зависимости по данным, и при непрямой адресации).

Схема доступа к нерегулярным структурам реализуется с помощью *битовых векторов* и *индексных векторов.* Использование битовых векторов (т.е. таких, элементы которых принимают значения 0 или 1 в зависимости от выполнения или невыполнения какого-либо условия) обеспечивает быстрый и эффективный способ избирательного доступа. Эти векторы генерируются с помощью ряда команд, таких, как сравнение векторов и операции ИЛИ/И над векторами, или с помощью условной установки разрядов. Битовые векторы хранятся в компактной форме, занимая слово или последовательность слов, и используются в качестве управляющих векторов в векторных операциях. Вектор индексов представляет собой целочисленный вектор, элементы которого могут использоваться в качестве полных адресов памяти или смещений, которые добавляются к базовому адресу.

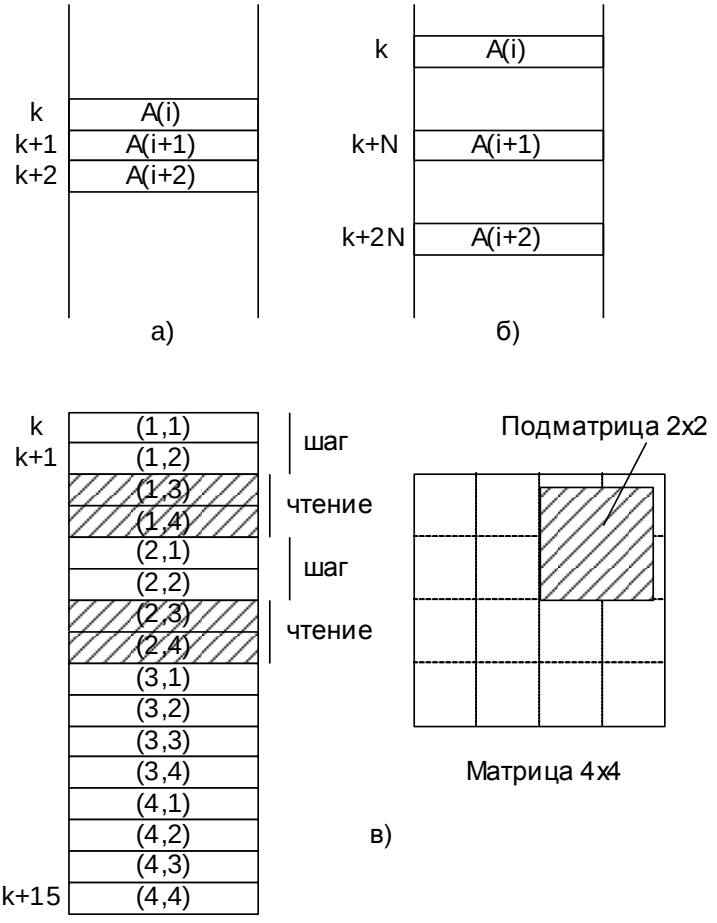


Рис.3.11. Схемы распределения данных в памяти:
а - последовательная; *б* - с регулярной структурой; *в* - в виде подматриц.

7. Обработка длинных векторов.

В векторных системах, работающих по принципу «регистр - регистр», длины векторов, непосредственно участвующих в операциях, фиксированы и определяются размерами векторных регистров. Для работы с векторами произвольной дины существует так называемый режим *векторного цикла*. Например, в системе Cray-1 при работе с векторами, со-

держащими более 64 элементов, используется векторный цикл, в котором управление осуществляется с помощью скалярных регистров параллельно с выполнением векторных операций, что позволяет исключить потери, связанные с организацией циклического перехода.

8. Операции редукции.

Операции типа скалярного произведения требуют специальных команд, которые превращают векторы в скаляры. Эти команды называются *векторной редукцией.* В некоторых системах операции данного типа включены в систему команд. Так в системе Cyber 205 имеется ряд векторных макрокоманд для различных видов редукции. В тех векторных системах, которые не имеют команд редукции, данная операция осуществляется иными способами. Например, операция редукции в системе Cray-1 осуществляется с помощью обычных векторных команд и аппаратно реализованного счетчика элементов.

Промышленные векторно-конвейерные системы. К наиболее известным векторно-конвейерным системам относятся такие суперЭВМ, как Cray-1 и Cray-2 фирмы Cray Research, STAR 100 и Cyber 205 фирмы CDC, S-810 фирмы Hitachi, SX-1 и SX-2 фирмы NEC, VP-200 фирмы Fujitsu.

Cray-1. Первая промышленная ВС конвейерного типа Cray-1 была выпущена фирмой Cray Research в 1976 г. Вся структура этой системы подчинена идее параллелизма. Она включает 12 конвейерных функциональных устройств, разбитых на 4 группы (адресную, скалярную, операций с плавающей точкой и векторную), и несколько групп быстрых регистров. Быстрые регистры являются одной из важнейших архитектурных особенностей системы Cray. Команды векторной обработки имеют структуру типа "регистр-регистр" (R-R), т.е. регистры адресуются непосредственно командами. Организация R-R позволила сделать слова команд более короткими, упростить выявление зависимостей между командами, но из-за ограниченного числа регистров привела к необходимости ограничения длины векторов на этапе трансляции программ.

В состав центрального процессора Cray-1 входят:

- главная память, объемом до 1048576 слов, разделенная на 16 независимых по обращению блоков, емкостью 64К слов каждый;
- регистровая память, состоящая из пяти групп быстрых регистров, предназначенных для хранения и преобразования адресов, для хранения и обработки векторных величин;
функциональные модули, в состав которых входят 12 параллельно работающих устройств, служащих для выполнения арифметических и логических операций над адресами, скалярными и векторными величинами;
- устройство, выполняющее функции управления параллельной работой модулей, блоков и узлов центрального процессора;
- 24 канала ввода-вывода, организованные в 6 групп с максимальной пропускной способностью 500000 слов в секунду (2 млн. байт в сек.).

Каждая операция разбивается на несколько этапов (сегментов) и выполняется по конвейерному принципу. Число сегментов для всех операций лежит в пределах 2-3 за исключением сложения с плавающей точкой (6 сегментов), умножения с плавающей точкой (7 сегментов) и вычисления обратной величины (14 сегментов). Сравнительно короткие конвейеры не требуют значительных затрат времени на перезагрузку. Группы быстрых регистров включают: восемь 24-разрядных адресных регистров (А-регистры), восемь 64-разрядных скалярных регистров (S-регистры), восемь векторных регистров (V-регистры), каждый из которых представляет собой массив из 64 элементарных 64-разрядных регистров; кроме того используется две группы буферных регистров: адресные (В-регистры) и скалярные (Т-регистры), а также специальный регистр маски (М-регистр).

Длительность такта конвейеров, т.е. длительность выполнения сегмента - 12,5 нс при скорости работы регистров - 6 нс. Цикл основной памяти составляет 50 нс; объем оперативной памяти (ОП) - от 1 до 4 млн. 64-разрядных слов. Согласование скоростей работы

конвейерных устройств обработки и ОП достигается расслоением - память выполнена из 16 независимых модулей. Ввод-вывод информации осуществляется посредством 24 автономных каналов. Для загрузки функциональных устройств используются конвейер команд, построенный на основе четырех буферов, каждый из которых состоит из 64 16-разрядных регистров.

В системе Cray-1 приняты команды двух форматов: 16- и 32- разрядного. Команды арифметических и логических операций типа R-R имеют формат Op,R1,R2,R3 (код операции Оп - 7 бит, адреса регистров операндов и результата - по 3 бита). При обращении к ОП используется 32-разрядный формат команды. В системе предусмотрена возможность сцепления векторных операций.

Кроме перечисленных архитектурных, структурных и технологических особенностей, обеспечивающих высокое быстродействие системы Cray-1, нужно отметить и дополнительные качества, способствующие повышению производительности. В первую очередь - компактность конструкции всей системы, которая позволяет сделать связи между элементами весьма короткими, что обеспечивает высокое физическое быстродействие. Второе - это наличие мощной подсистемы дисковой памяти и третье, программное обеспечение, включающее операционную систему COS или CTSS, оптимизирующий компилятор с Фортрана, удобный для реализации векторных команд, библиотеку стандартных подпрограмм и т.п.

В дальнейшем были разработаны модели Cray-1S, X-MP, -2 и -3, которые базировались на идеях, реализованных в Cray-1 и подтвердивших высокую эффективность при векторной обработке. Эти модели отличаются числом независимых конвейерных процессоров, числом регистров в каждом процессоре, числом и типом функциональных конвейеров, а также элементной базой (такт конвейера Cray X-MP составляет 9,5нс, в Cray-2 - 4,1нс, конвейер реализован на СИС ЭСЛ типа), объем ОП (до 2 ГБ в Cray-2).

S-810. Векторная вычислительная система S-810 – это первая суперЭВМ фирмы Hitachi, ставшая началом серии одноименных систем сверхвысокой производительности, предназначенных специально для научных и инженерных расчетов. В начале 90-х годов эта система считалась одной из самых быстродействующих в мире. Пиковая производительность системы S-810 превышает 630 млн. операций с плавающей запятой в секунду, а объем главной памяти составляет 256 Мбайт. Особенно высокие показатели S-810 достигаются при решении задач имитационного моделирования различных физических систем и явлений. Благодаря своей исключительно высокой производительности и большому объему памяти система использовалась как инструмент для решения крупных научных задач, которые прежде считались практически не решаемыми из-за огромной трудоемкости.

Высокие характеристики данной суперЭВМ достигнуты благодаря тому, что система создана на основе передовых (на начало 90-х годов) концепций, главные из которых заключаются в следующем.

1. Высокая производительность.

Максимальная производительность модели S-810/10 составляет 315, а модели S-810/20 - 630 млн. операций с плавающей запятой в секунду. Эти показатели обеспечиваются за счет следующих особенностей:

обработка данных ведется в параллельно-конвейерном режиме, т.е. когда одновременно работают несколько конвейерных арифметических устройств, число которых зависит от степени параллелизма программы (имеются также другие уровни параллелизма);

- программы составляются в векторной форме благодаря расширенному набору векторных команд;
- осуществляется высокопроизводительная скалярная обработка;
- используется расширенная память.

2. Новая система организации памяти.

Одно из обязательных требований при организации высокоскоростных вычислений состоит в том, что для исключения задержек, связанных с вводом-выводом, память данных должна иметь большую емкость. Поэтому в состав S-810 корме быстродействующей главной памяти включена так называемая расширенная память, имеющая большой объем. В модели S-810/10 объем главной памяти составляет 128 Мбайт, а расширенной – 1024 Мбайта. В модели S-810/20 объем главной памяти соответственно равен 256 Мбайт, а расширенной – 512 Мбайта, т.е. на половину меньше, чем в модели S-810/10. Скорость обмена данными между главной и расширенной памятью составляет для модели S-810/20 1000 Мбайт/с, что более чем в 300 раз превышает скорость обмена между главной и дисковой памятью.

3. Гибкая конфигурация системы.

В системе S-810 могут быть образованы две конфигурации: многопроцессорная со слабой связью и автономная. В первой конфигурации S-810 используется в качестве характерной многопроцессорной суперЭВМ, а во второй – как коммерческая система с экстраординарными возможностями.

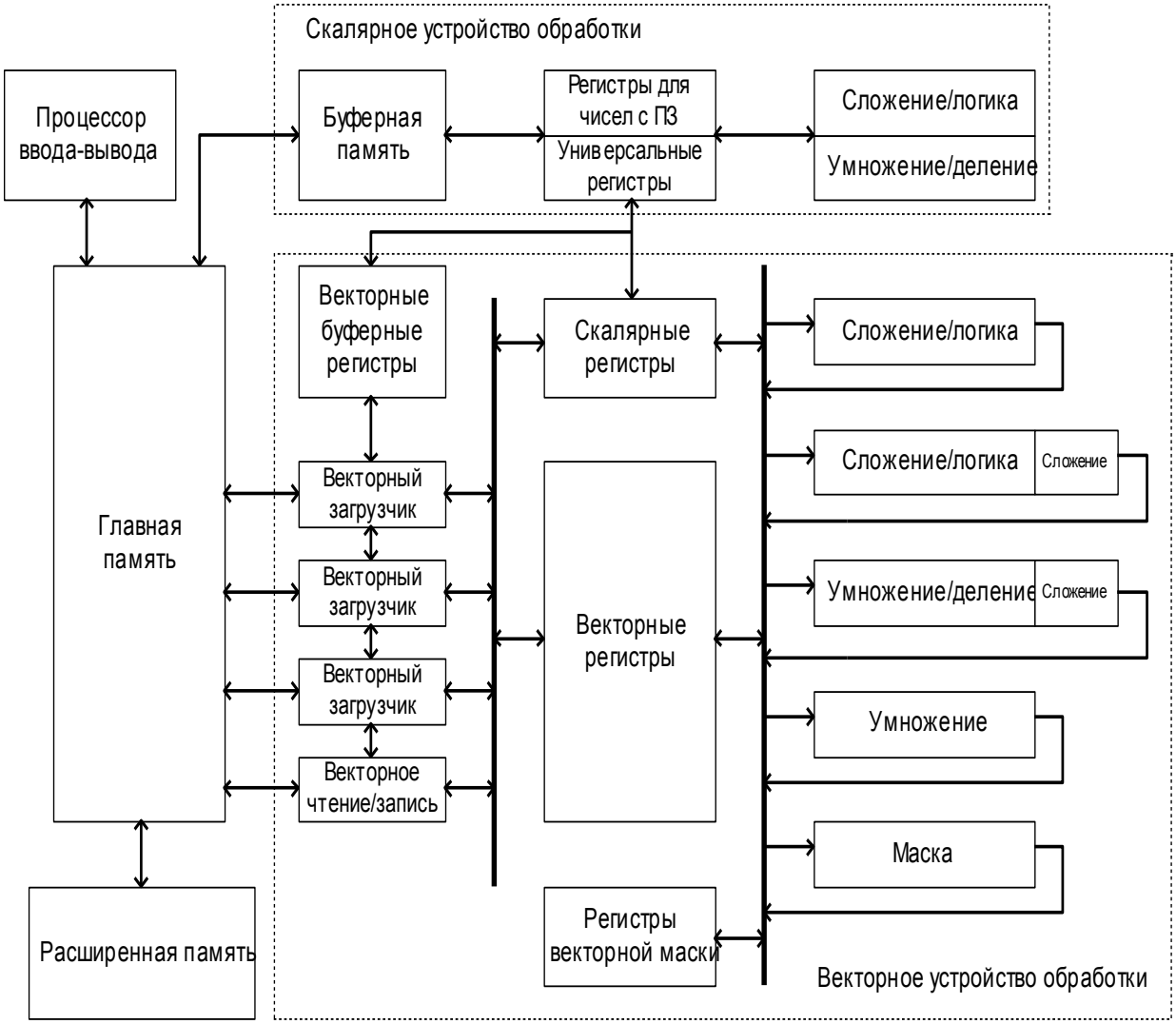


Рис.3.12. Структурная схема системы S-810 (модель 10).

4. Высокая надежность.

Надежность системы S-810 обеспечивается как за счет высокой надежности элементной базы, так и за счет специально встроенных аппаратных средств. К этим сред-

ствам относятся: трассировщик, фиксирующий работу конвейерных ступеней, устройство регистрации ошибок и др. В составе системы имеются также средства обеспечения эксплуатации, управляемые так называемым процессором технического обслуживания.

Кроме того, при создании системы S-810 была применена новейшая технология, позволившая получить высокие значения многих ключевых показателей. К ним в частности относятся малое время доступа к памяти и регистрам и малая длительность конвейерного такта арифметических устройств.

На рис.3.12 приведена структурная схема модели S-810/10. Векторное устройство обработки этой модели содержит следующие основные элементы: 6 арифметических устройств, 3 векторных загрузчика, одно векторное устройство загрузки/записи, 16 векторных регистров, 16 скалярных регистров, 8 регистров векторных масок. Векторное устройство модели S-810/20 содержит вдвое больше арифметических устройств, векторных загрузчиков и устройств записи, а также векторных и скалярных регистров.

5. Программное обеспечение.

С целью наиболее полного использования высоких потенциальных возможностей оборудования системы S-810 был специально разработан компилятор с языка Фортран FORT 77/HAP. FORT 77/HAP является векторизующим компилятором, способным превращать программы, написанные на стандартном Фортране, в объектные коды, содержащие векторные команды.

Некоторые характеристики моделей векторно-конвейерных ВС приведены в таблице 3.1. Пиковая производительность достигается только на "подходящих" задачах для данной ВС и резко снижается при решении произвольных задач других классов, причем это снижение для всех ВС различно.

Таблица 3.1

Модель ВС	Рмакс, (Мфлопс)	Число ПЭ	Адресация	Такт, нс		Тип ИС	объем ОП, (Мслов)
				вект.	скал.		
CRAY-1	80	1	R-R	12,5	12,5	ИС ЭСЛ	1
CRAY-X-MP	210	1-4	R-R	9,5	9,5	СИС ЭСЛ	2-16
CRAY-2	487	4	R-R	4,1	4,1	СИС ЭСЛ	128
VP-200	533	1	R-R	7,5	15,0	БИС ЭСЛ	32
S-810/20	840	1	R-R	14,0	28,0	БИС ЭСЛ	32
CYBER-205	40	1+1	S-S	20,0	40,0	БИС ЭСЛ	32-128

3.2.2 Матричная обработка

Другой тип вычислительных машин, ориентированных на алгоритмы, содержащие главным образом векторные операции, - это матричные системы. Общая идея построения таких систем чрезвычайно проста - имеется большое количество одинаковых обрабатывающих устройств, в каждый из которых поступают свои данные, но которое одновременно выполняет одну и ту же операцию.

Как показано на рис.3.13, матричная система состоит из множества процессорных элементов (ПЭ), организованных таким образом, что они исполняют векторные команды, задаваемые общим для всех устройством управления, причем каждый ПЭ работает с отдельным элементом вектора. ПЭ соединены через коммутационное устройство с многомодульной памятью. Исполнение векторной команды включает чтение из памяти элементов векторов, распределение их по процессорам, выполнение заданной операции и засылку результатов обратно в память. Доступ процессоров к данным, хранящимся в главной памяти, осуществляется через коммутационную сеть. Скалярные команды исполняются управляющим процессором (УП).

Идея матричной обработки привлекательна как с точки зрения технологии - большое число однотипных обрабатывающих устройств позволяет эффективно использовать возможности СБИС, так и с точки зрения повышения производительности ВС. Действительно, теоретически матричная система из **p** процессоров может работать в **p** раз быстрее, чем однопроцессорная, однако этому мешает ряд факторов, аналогичных тем, которые действуют и в отношении векторно-конвейерных систем:

- в случае, когда число элементов обрабатываемых векторов не кратно числу процессоров, появляются незанятые процессоры и общая пропускная способность уменьшается;
- алгоритмы, как правило, содержат скалярные операции, выполнение которых не может перекрываться с выполнением векторных команд. Таким образом, весь массив процессоров какое-то время бездействует;
- при считывании векторов из памяти возникают конфликты. Память, как и в случае векторных машин, выполняется в виде набора модулей и как только в одном модуле оказывается более одной компоненты вектора, возникает конфликт. В связи с этим, с целью уменьшить отрицательное влияние таких конфликтов на производительность, был разработан ряд запоминающих устройств для хранения матриц и векторов;
- коммутационная сеть имеет ограниченные возможности. Данные должны передаваться из памяти в соответствующие процессоры одновременно. Чтобы обеспечить возможность любому процессору работать с любым модулем памяти, необходимо использовать коммутатор перекрестных связей. Однако стоимость такого коммутатора растет пропорционально произведению числа процессоров на число модулей памяти и выходит за рамки разумного уже при относительно небольших количествах этих устройств. По этой причине большие усилия были направлены на создание других типов коммутационных сетей, которые при меньшей стоимости обеспечивали бы достаточно высокую скорость передачи данных.

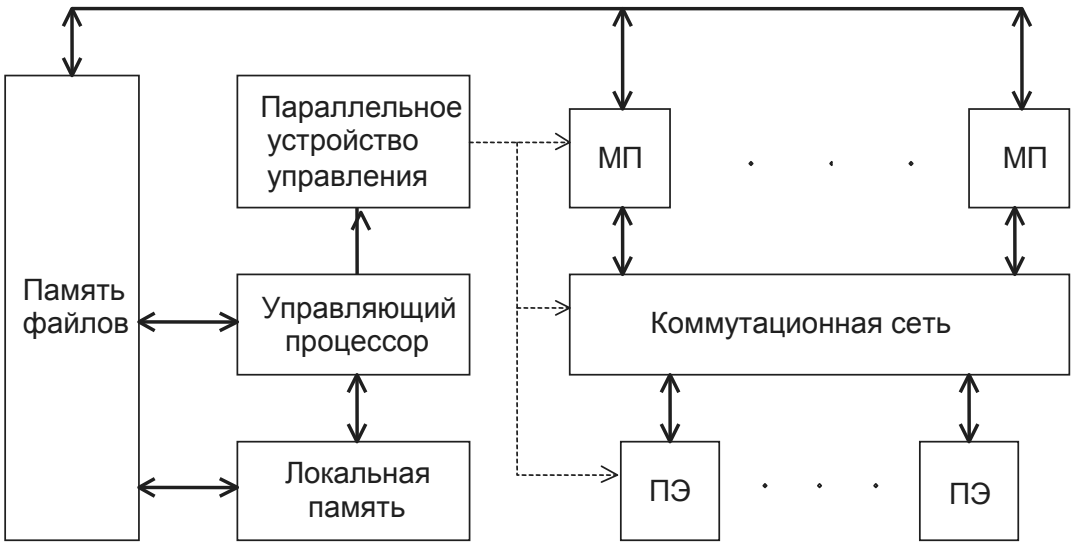


Рис.3.13. Структура матричной вычислительной системы
(МП – модуль памяти, ПЭ - процессорный элемент)

В первых ВС матричного типа коммуникационная сеть выполнялась в виде связей ПЭ с четырьмя ближайшими соседями, т.е. совокупность ПЭ можно было рассматривать как двумерную матрицу. В последующем стали использовать различные коммуникационные сети. Управляющий процессор включает в себя АЛУ, регистры, узлы управления, локальную память; он обеспечивает передачу команд на обрабатывающие ПЭ, организует

работу системы ввода-вывода, управляет работой коммуникационной сети. Во многих матричных ВС управляющий процессор выполнен в виде отдельной ЭВМ общего назначения или мини-ЭВМ. Основная его задача - выбрать команду и определить является ли она командой матричной операции. В случае матричной операции команда должны быть передана в обрабатывающие ПЭ, модули памяти которых хранят соответствующие элементы векторов или матриц. В простейшем случае локальные памяти ПЭ хранят по одному элементу матричных операндов; при одновременном получении команды все ПЭ выполняют параллельно операцию над операндами матрицы, т.е. операция над вектором или матрицей выполняется за один цикл. В случае, если выбранная команда не является матричной, она выполняется в самом УП. Таким образом название "матричная ВС" отражает не только структурные особенности, заключающиеся в использовании "матрицы" ПЭ, но и организацию ВС на выполнение операций над матрицами или векторами. Обрабатывающие ПЭ являются ведомыми по отношению к УП; они не способны воспринимать и обрабатывать команды условных переходов. ПЭ могут быть предназначены для обработки слов или отдельных разрядов. В первом случае матричные системы обычно предназначаются для матричных операций при научно-технических расчетах. Операнды представляются в виде слов с плавающей точкой или комплексных чисел и имеют разрядность от 32 до 64 бит. В случае однобитовых ПЭ матрица процессоров обычно используется для обработки изображений. Поскольку основная задача при создании матричных ВС заключается в повышении быстродействия, то необходимо обеспечить высокую пропускную способность каналов связи между ПЭ и памятью и между УП и ПЭ. Для устранения возможных задержек из-за конфликтов одновременных обращений в память необходимо предусмотреть буфер команд достаточно большой глубины. Выборку команд и операндов можно выполнять параллельно, если отказаться от размещения операндов в общей памяти. Кроме того, конвейеризация команд может позволить параллельно выполнять матричные команды в ПЭ и скалярные - в УП. Все сказанное позволяет сделать вывод о том, что эффективная производительность матричной ВС зависит не только от алгоритма, но и от размещения данных в локальных памятях ПЭ. Если размеры обрабатываемых матриц превышают число ПЭ, то в локальной памяти каждого ПЭ необходимо разместить строку, столбец или подматрицу структуры данных; эти подструктуры теперь должны обрабатываться последовательно, что естественно приведет к снижению быстродействия. Поскольку размерность матриц в задачах произвольна, то при алгоритмизации задачи и оптимизации программы необходимо найти наиболее распространенное размещение элементов данных. В качестве критерия для оптимизации часто используют произведение времени реализации алгоритма **t** на число ПЭ, т.е. **t*n -> min.** Это позволяет найти алгоритм, наилучшим образом соответствующий данной матричной ВС. При проектировании специализированных ВС обычно приходится искать структуру, наилучшим образом соответствующую заданному алгоритму. Примером может служить обработка изображений.

Матричные операции и режимы работы ПЭ. Под матричными операциями обычно понимают различные операции над векторами, матрицами или другими структурами данных, производимые совокупностью ПЭ матричной ВС. Соответствующие команды выделяются из общего потока в управляющем процессоре, преобразуются в нем и передаются на исполнение всем ПЭ.

Одной из простейших матричных операций является операция поэлементного сложения матриц A и B: $\mathbf{C_{ij} = A_{ij} + B_{ij}}$, i = 1,m, j = 1,n.

При наличии в системе **m x n** ПЭ элементы исходных матриц A и B и матрицы результата C можно распределить по локальным блокам памяти процессорных элементов так, чтобы номер ПЭ соответствовал индексам элементов матрицы, т.е. в ПЭ (ij) содержатся элементы a_{ij}, b_{ij}, c_{ij}. Управляющему процессору для выполнения операции сложения матриц достаточно передать на ПЭ команду **ADD A,B,C.** При меньшем числе ПЭ в ВС, например если их всего **m**, можно распределить элементы матриц по строкам, чтобы в

локальной памяти $ПЭ_1$ находились все элементы одной строки a_{1j}, b_{1j}, c_{1j}, j = 1,n, в $ПЭ_2$ - a_{2j}, b_{2j}, c_{2j} и т.д. Для выполнения операции сложения матриц в этом случае УП должен передать на все процессорные элементы последовательность команд типа ADD A(к), B(к), C(к) (к = 1,n), причем при выполнении каждой такой команды в соответствующем ПЭ будет формироваться по одному элементу $\mathbf{c_{ij}}$, т.е. всеми вместе ПЭ одновременно будет сформирован один **k**-тый столбец матрицы C. При наличии индексных регистров в каждом ПЭ матричной ВС управляющий процессор может передать одну команду с указанием индекса. Процессорный элемент должен при этом самостоятельно формировать последовательность исполнительных команд для вычисления $\mathbf{c_{ij}}$. В любом из рассмотренных случаев особенностью действий при выполнении операции сложения матриц является отсутствие взаимодействия отдельных ПЭ между собой: исходные данные выбираются из собственного блока локальной памяти и в нее же помещается результат обработки.

Более сложной для реализации представляется операция скалярного произведения двух векторов A и B; результат S определяется формулой $\mathbf{S = \sum a_i * b_i,}$ i = 1,n.

Если элементы векторов a_i и b_i распределены по локальным памятям $ПЭ_i$ (i=1,n), то вычисление произведений $\mathbf{c_i = a_i\ b_i}$ может быть выполнено по схеме аналогичной описанной выше для сложения матриц. Однако для вычисления суммы $\mathbf{S=\sum c_i}$ требуется передавать значения $\mathbf{c_i}$ в другие процессорные элементы $ПЭ_j$. Для этого необходимо в архитектуре матричного ВС предусмотреть коммуникационную сеть связи между отдельными ПЭ. Наличие такой коммуникационной сети не только значительно усложняет матричную ВС, но и приводит к снижению быстродействия из-за потерь времени на пересылки операндов по этой сети. Возможные способы построения коммуникационных сетей рассмотрены ниже.

Еще одна причина снижения производительности состоит в необходимости выполнения матричных операций условной обработки. Рассмотрим типичный пример.

Пусть необходимо сформировать вектор C так, что

$c_i = a_i + b_i$ если $a_i < 0$ и

$c_i = a_i - b_i$ если $a_i \in 0$,

То есть различные ПЭ матричной ВС должны выполнять разные операции, однако управляющий процессор формирует для них общий поток команд. Решением может служить режим "маски", когда ПЭ принимает команду от управляющего процессора на исполнение только в том случае, если в нем установлен в единицу специальный триггер маски, в противном случае команда в данном ПЭ игнорируется. Тогда для формирования вектора C должна быть выполнена следующая последовательность команд управляющего процессора:

TEST A < 0	:Все ПЭ принимают эту команду и выполняют сравнение a_i с 0.
	:Если $a_i <$ 0, то в ПЭ устанавливается признак маски (m_i= 1).
ADD A,B,C	:В тех ПЭ, где m_i= 1, производится сложение ci = ai + bi.
COMP M	:В ПЭ передается команда на инвертирование значения признака маски.
SUB M A,B,C	:В тех ПЭ, где теперь установлены признаки маски, выполняется
	:вычитание ci = ai - bi. Этими ПЭ являются те, в которых не произ-
	:водилось сложение при выполнении второй команды.
SET M	:Управляющий процессор передает всем ПЭ команду установки маски,
	:позволяющую перевести все арифметические ПЭ в рабочее состояние.

Таким образом, для одной операции над векторами управляющему процессору пришлось сформировать последовательность из пяти команд и передать их в ПЭ. Эта последовательность должна выполняться полностью, даже если все элементы $\mathbf{a_i}$ (i=1,n) отвечают указанному условию и последние три команды становятся лишними. Предотвратить их выдачу исполнительным процессором можно в том случае, если он получает информацию

о состоянии маски из всех ПЭ. Команда установки маски предоставляют возможность управляющему процессору выбирать определение строки и столбцы при операциях над матрицами. Это очень важно в случаях, когда число элементов матрицы (или вектора) меньше числа ПЭ в системе, а также при работе с разреженными матрицами. В большинстве современных матричных ВС каждый ПЭ имеет несколько триггеров, служащих для индивидуального управления ПЭ. Помимо триггера маски может быть использован триггер конфигурации - его функции аналогичны функциям триггера маски, но установка или сброс производятся перед началом выполнения задачи. Таким образом создается возможность исключения из системы неисправных или ненужных ПЭ. Кроме того, предусматривается несколько триггеров, выполняющих функции стека признаков маски (при выполнении циклов), а также несколько триггеров для запоминания признака результата индивидуальной операции (например, знака результата, равенства нулю, переполнения, деление на ноль и т.п.). Скалярные операции, как указывалось выше, реализуются в управляющем процессоре, поэтому при выполнении скалярной операции все ПЭ могут простаивать. Для уменьшения простоев ПЭ, во-первых, стремятся увеличить быстродействие управляющего процессора и, во-вторых, реализовать сложение схемы буферизации команд, позволяющее организовать совмещение скалярной обработки в УП и матричных операций в ПЭ.

Коммутационные сети Коммутационные сети служат для организации взаимодействия между отдельными ПЭ матричной ВС. Поскольку все ПЭ выполняют одну и ту же команду от управляющего процессора, то взаимодействие ПЭ заключается в пересылке между ними результатов выполнения примитивной операции. При выполнении произвольного алгоритма может возникнуть необходимость пересылки частичного результата между любыми ПЭ. Это требование может быть удовлетворено коммутационной сетью, реализующей соединение всех ПЭ посредством общей разделяемой шины, полного коммутатора или системы прямых соединений.

В высокопроизводительных матричных ВС требования к пропускной способности коммутационной сети очень высоки, так как в значительной степени она определяет максимальную производительность всей ВС. С другой стороны, всегда существуют экономические и технологические ограничения. Из-за малой пропускной способности в матричных ВС не может использоваться общая разделяемая шина; в синхронной системе все частичные результаты в ПЭ формируются одновременно и желательна их одновременная передача другим ПЭ. В этом случае использование разделяемой шины привело бы к очень значительному снижению быстродействия. Крайним случаем, является синхронная коммутационная сеть, в которой любой ПЭ прямо связан с каждым из всех остальных ПЭ системы и все передачи могут выполняться синхронно. Такая система соединений получила название *полносвязанной.*

Для построения полносвязанной системы, состоящей из **n** ПЭ, потребуется **n(n-1)** линий связи, т.е. при большом числе ПЭ полносвязанная система становится нереализуемой по экономическим и конструктивным соображениям.

Одной из наиболее распространенных коммутационных сетей в матричных ВС является двухмерная сеть, соединяющая каждый ПЭ с ближайшими соседями. Подобная сеть эффективна при реализации достаточно широкого класса алгоритмов, например алгоритмов решения дифференциальных уравнений в частных производных, обработки изображений и т.п. Недостатки такой коммутационной сети сразу же проявляются при выполнении алгоритмов, требующих других видов перестановок элементов матрицы.

Второй широко распространенной схемой коммутационной сети является сеть с полной тасовкой. Эта схема эффективна для широкого круга алгоритмов упорядо-

чения (сортировки) символьной информации, преобразований Фурье и Уолша, транспонирования матриц и т.п.

Промышленные матричные ВС. Впервые идея создания матричной ВС нашла воплощение в проекте системы **SOLOMON**, содержащей матрицы 32 х 32 ПЭ, выполняющих арифметические и логические операции над словами длиной от 1 до 128 бит. Каждый ПЭ соединен с четырьмя соседними. В состав ПЭ входят локальная память объемом 16 Кбит. Однако из-за конструктивно-технологических факторов реализация проекта оказалась неэффективной.

Дальнейшее развитие идеи матричной обработки получили в системе **ILLIAC-IV**. Первоначальный проект предусматривал создание ВС из 256 ПЭ, разбитых на четыре автономных квадранта, но вследствие технологических ограничений была реализована ВС, включающая всего один квадрант из 64 ПЭ. В качестве управляющей машины использовалась ЭВМ общего назначения В-6700 фирмы BURROUGHS. Система ILLIAC-IV построена в единственном экземпляре и эксплуатировалась с 1974 г. в течение нескольких лет.

Каждый ПЭ системы ILLIAC-IV включает АЛУ, способное выполнять широкий набор команд с фиксированной и плавающей точкой над 64-, 32, 24- и 8-разрядными словами, локальную память объемом 2К 64-разрядных слов, а также схемы локального управления. Каждый ПЭ связан с четырьмя соседними по принципу двухмерной прямоугольной матрицы. Схемы локального управления служат для приема команд от системы глобального управления, для интерпретации кода операции, выработки адресов операндов, управления маской, управления пересылками между ПЭ, а также для ряда действий, связанных с загрузкой локальных памятей в процессе глобально управляемой операции ввода-вывода. Операция сложения двух 64-разрядных чисел в АЛУ ПЭ занимала 240 нс, умножения - 400 нс; общее быстродействие системы оценивалось в 200 млн.операций/с. Система ILLIAC-IV использовалась для многоканальной свертки и фильтрации радиолокационных сигналов, БПФ, задач линейного программирования.

По-существу первой промышленной системой этого класса явилась разработанная в СССР вначале 80-х годов параллельная система **ПС-2000**. Одним из факторов, способствующих ее успеху, было развитие технологии ИС. Основой ПС-2000 является параллельный процессор ППС-200, включающий в свой состав от 8 до 64 одинаковых ПЭ, которые объединялись по 8 штук в модули обработки и наращивание системы производится добавлением таких модулей. Все ПЭ объединены между собой двумя основными магистралями: магистралью команд и магистралью микрокоманд. Помимо магистральных связей каждый ПЭ соединен непосредственными связями (регулярным каналом) с двумя соседними ПЭ. В состав ППС входит также устройство глобального управления, вырабатывающее последовательность команд и микрокоманд и осуществляющее их выдачу в ПЭ. Общее управление в ПС-2000 осуществляется мониторной системой, построенной на базе мини-ЭВМ СМ-2М, для которой ППС является как бы специализированным процессором. Управление операциями ввода-вывода и внешней памятью также возлагается на мониторную систему. При сравнительно низкой стоимости ПС-2000 на ряде задач позволо достичь эффективного быстродействия порядка 200млн. операций/с.

Одной из наиболее мощных ВС матричного типа является система **MPP** фирмы *Goodyear Aerospace Co*. Она включает 16К одноразрядных ПЭ, способных выполнять до 6 миллиардов сложений 8-разрядных слов в секунду. Все операции в матрице ПЭ осуществляются под управлением узла управления матрицей (УУМ); УУМ принимает информацию о состоянии отдельных ПЭ, формирует управляющие сигналы (К), а также выполняет скалярные операции. Общее управление потоками данных и команд осуществляет процессор команд, который также выполняет управление операциями ввода-вывода. Передача данных в матрицу ПЭ и из нее осуществляется 128-разрядными словами через буферные ЗУ и

коммутаторы. Каждый ПЭ двумерной матрицы (реализованы связи с четырьмя соседними ПЭ) содержит одноразрядный последовательный полный сумматор, шесть регистров, компаратор, локальную память объемом 1024 бит, простейшие схемы управления и коммутации. В одном корпусе СБИС размещена подматрица из 2 х 4 ПЭ (все логические схемы и регистры, локальные памяти размещены на отдельных БИС). Кроме того, в матрице предусмотрено дополнительно четыре столбца ПЭ, которые обеспечивают отказоустойчивость системы, что очень важно при большом числе компонентов. Система MPP весьма эффективна при обработке изображений. Каждый пиксел обрабатывается отдельным ПЭ. Поразрядная последовательность обработки в каждом ПЭ не только хорошо согласуется со структурой плоскографического изображения (видеопамятью), но и в значительной степени смягчает требования к системе ввода-вывода: последовательная обработка увеличивает длительность интервала, с течение которого должен быть введен (выведен) очередной квант информации, а поразрядная обработка позволяет осуществлять передачу данных широким словом (128 бит). Высокое быстродействие при этом достигается за счет очень большого числа ПЭ, реализованных на СБИС.

Широкую известность также получили матричные системы DAP (фирмы Active Memory Technology), MP (MasPar Computer Co), CM (Thinking Machines Co).

Система **DAP-610** построена в виде матрицы 64 х 64 одноразрядных ПЭ, реализованных по кМОП технологии (в более ранней модели DAP-500 матрица состояла из 32х32 ПЭ) с циклом 100 нс.

Каждый ПЭ содержит локальную память не менее 32 Кбит; он соединен с ближайшими соседями непосредственными связями, а кроме того ортогональными магистралями соединены все ПЭ каждой строки и столбца. Каждый ПЭ построен в виде совокупности двух элементов - основного и дублирующего, выполняющих одну и ту же операцию; при несовпадении ее результатов сигнал об ошибке передается в узел центрального управления, который служит для интерпретации команд, выполнения скалярных операций и контроля за сигналами об ошибках. Особенность ПЭ системы DAP заключается в том, что к ним могут подключаться 8-разрядные сопроцессоры. Измеренное быстродействие оценивается в 28 MFLOPS (на операциях умножения) и свыше 3 GFLOPS для логических операций.

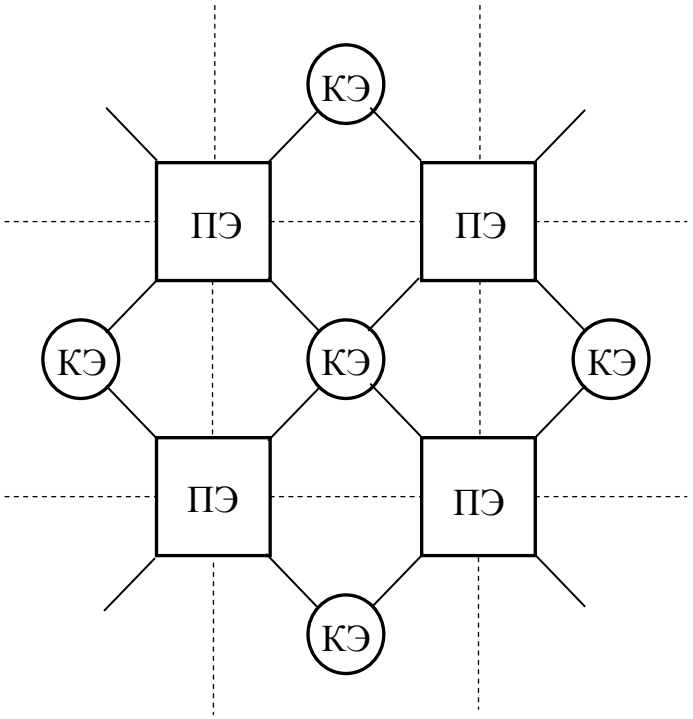


Рис 3.13. Х-сеть.

Система **МР-1** включает от 1024 до 16384 ПЭ, имеющих архитектуру с сокращенным набором команд. В состав ПЭ входит локальная память 16 Кбит, четырехразрядное АЛУ, узел мантисс (64 бита), узел порядков (16 бит), узел логических операций (1 бит), что позволяет производиться обработку 1-,8-, 16-, 32- и 64-разрядных слов. Процессорные элементы объединены в кластеры 4х4, каждый кластер связан с коммутационной сетью. Использована Х-сеть (рис 2.13), в которой обеспечивается соединение каждого процессорного элемента с 8 соседними (непосредственно или через коммутирующий элемент - КЭ).

Подсистема глобального управления реализована на RISC-

процессоре, содержит память команд и память данных. Эта подсистема осуществляет выборку и декодирование команд, вычисление адресов, выполняет скалярные операции и следит за состоянием процессорных элементов. Быстродействие МР-1 оценивается в 1,3 GFLOPS при максимальной конфигурации матрицы.

СМ-1 или **Connection Machine** включает от 4096 до 65536 одно·разрядных ПЭ, выполненных по кМОП технологии. Каждый ПЭ имеет локальную память объемом от 64 Кбит до 1 Мбит. В одном корпусе БИС размещено 16 ПЭ; к каждой паре корпусов может быть подключен ускоритель операций с плавающей точкой. ПЭ соединяются по схеме с четырьмя ближайшими соседями или по схеме гиперкуб, для чего в каждом ПЭ использован маршрутизатор. ПЭ объединены в октанты (по 8192), каждый из которых подключается посредством двух каналов ввода-вывода к графическому монитору, дисковой системе и т.п.

Краткие сравнительные характеристики некоторых матричных систем приведены в таблице 3.2.

Таблица 3.2

Модель	Число ПЭ	Тип ПЭ	ЛП (Кбит)	Тип связи	Производительность (MFLOPS)
ПС-2000	8 - 64	RISC	4К х 24	Кольцо	200
MPP	16384		1	4 соседа	6000
DAP-610	16384		32	4 соседа	28000
МР-1	16384		16	Х-сеть	1300
СМ	4096 - 65536		1000	Гиперкуб	28000

3.2.3. Систолические системы

Систолические архитектуры возникли из желания совместить преимущества конвейерной и матричной обработки. Первоначально систолические архитектуры разрабатывались для узкоспециализированных ВС, однако в дальнейшем оказалось возможным найти подходящие алгоритмы для достаточно широкого класса задач, позволяющие реализовать принципы систолической обработки.

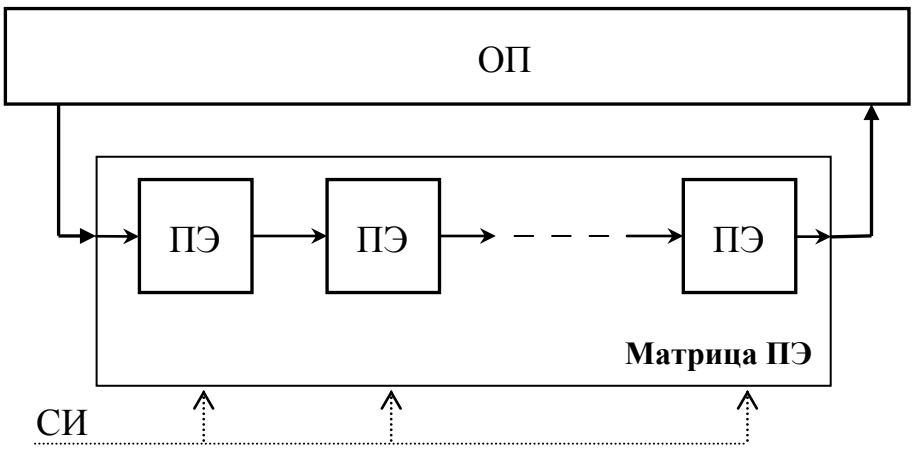
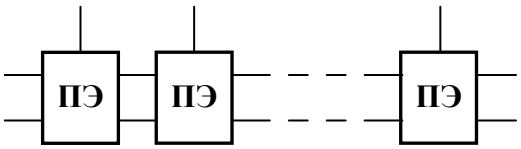


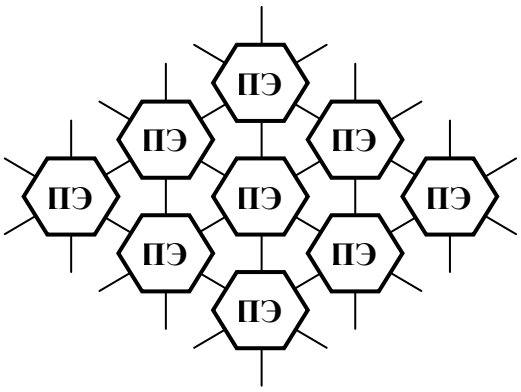
Рис.3.14. Принцип систолической обработки.

Основной принцип систолической обработки заключается в том, чтобы выполнить все стадии обработки каждого элемента данных, извлеченного из памяти, прежде чем

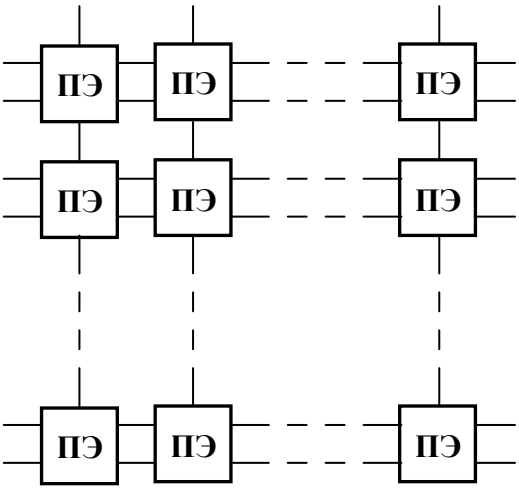
вновь поместить в память результат этой обработки. Этот принцип реализуется систолической матрицей ПЭ, в которой отдельные ПЭ объединены между собой прямыми и регулярными связями, образующими конвейеры (рис.3.14). По этим конвейерам как бы прокачиваются операнды, т.е. каждый элемент данных извлекается из памяти и ритмически продвигается по матрице ПЭ прежде, чем опять попадет в память. Таким образом, может формироваться несколько потоков данных, каждый из которых образован исходными операндами (элементами структуры данных, хранящейся в памяти), промежуточными результатами, получаемыми при выполнении элементарных операций в каждом ПЭ, и элементами результирующей структуры. Потоки данных синхронизированы единой для всех ПЭ системой тактовых сигналов. Во время тактового интервала все ПЭ выполняют короткую неизменную последовательность команд.



а) Линейная структура



в) Гексагональная структура



б) Прямоугольная структура

Рис.3.15. Конфигурации систолических матриц.

Преимущества систолической обработки очевидны:

минимизируются обращения в память, что позволяет сравнительно просто согласовать скорость работы ОП со скоростью обработки;

облегчается решение проблем ввода-вывода вследствие уменьшения конфликтов при обращениях в ОП;

эффективно используются возможности технологии СБИС за счет регулярности структуры систолической матрицы;

минимизируются связи между ПЭ за счет регулярности потоков данных и управляющих сигналов.

Однако для того, чтобы реализовать эти преимущества, необходимо найти для каждой задачи соответствующие систолические алгоритмы, которые могут быть "отображены" на систолическую структуру ВС. Такие алгоритмы найдены для широкого спектра задач, среди которых задачи числовой обработки, обработки сигналов и символов: умножение и обращение матриц, решение линейных систем, дискретное преобразование Фурье, кодирование и декодирование числовых последовательностей и т.п. Большинство этих алгоритмов сводится к рекуррентным соотношениям того или иного вида. В зависимости от вида систолического алгоритма, т.е. числа операндов, участвующих в примитив-

ных операциях, типа этих элементарных операций и последовательности их выполнения, выбирается тип ПЭ и структура локальных регулярных связей между ними.

Конфигурации систолических матриц, среди которых наиболее распространенными являются линейная, прямоугольная и гексагональная (рис.3.15), приспособлены для выполнения вполне определенных функций. Так линейная конфигурация служит для реализации алгоритмов фильтрации при обработке сигналов сравнения цепочек литер при обработке баз данных; квадратная - позволяет производить перемножение матриц, находить двумерное дискретное преобразование Фурье (ДПФ); гексагональная - операции обращения матриц, решение линейных систем уравнений и т.п.

Для пояснения принципов действия систолической системы рассмотрим выполнение операции умножения квадратных матриц.

Результат перемножения двух матриц **C = A * B** формируется согласно формуле:

$$\mathbf{c_{ij} = \sum a_{ik}\, b_{kj},}$$

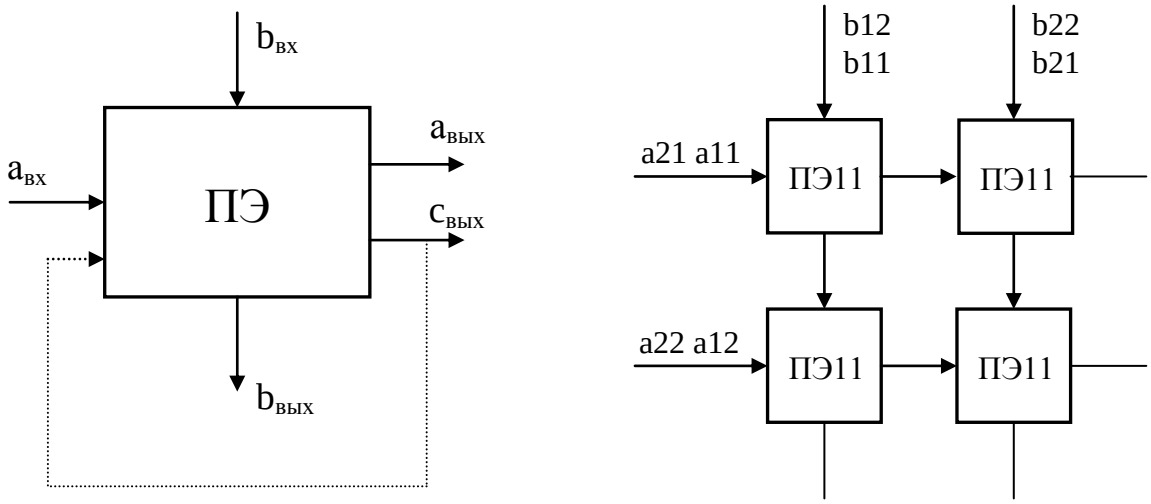
причем каждый элемент матрицы C = [c_{ij}] можно рассматривать как результат рекуррентных вычислений: $c_{ij}^{a(0)} = 0;$ $c_{ij}^{a(k)} = c_{ij}^{a(k-1)} + a_{ik}\, b_{kj},$ k=1,n.

Такие суммы образуются при накапливании частичных сумм в процессе продвижения $\mathbf{a_{ij}}$ и $\mathbf{b_{ij}}$ по систолической матрице.

Пусть обе матрицы **A** и **B** имеют размерность 2x2:

$$A = \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} \quad B = \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix}$$

и операция умножения производится с помощью ПЭ, изображенных на рис.3.16а, объединенных в прямоугольную матрицу (рис3.16б). Загрузка элементов матриц A и B из памяти производится по строкам (A) и по столбцам (B) с характерным для систолической обработки сдвигом. Этот сдвиг на рис.3.17 обозначен 0.



а) Процессорный элемент. б) Систолическая матрица.

Рис.3.16. Пример систолической системы.

В начальный момент все ПЭ установлены в исходное нулевое состояние. На первом такте в ПЭ1 загружаются элементы a11 и b11 и вычисляется произведение a11b11. На втором такте элементы a11 и b11 поступают на входе ПЭ12 и ПЭ21, соответственно. На ПЭ11 поступают a21 и b12 и в нем вычисляется сумма произведений a11b11 + a21b12 (первое слагаемое этой суммы сохранено в ПЭ11 с первого такта). Одновременно в ПЭ12 и ПЭ21

вычисляются произведения a12b11 и a11b21, соответственно. Этот процесс продолжается, как показано на рис.3.17.

Весь процесс перемножения матриц требует **3n-2** шагов, где **n** - размерность перемножаемых матриц. Если вспомнить, что в последовательной ЭВМ перемножение матриц выполняется за n^3 шагов, то становится ясно, что достигаемое ускорение для систолической ВС составляет $n^3/(3n-2)$, т.е. имеет порядок $O(n^2)$.

Такт	ПЭ11	ПЭ12	ПЭ21	ПЭ22	
1	**a11b11**	**0**	**0**	**0**	c11=a11b11+a21b12
2	**c11**	**a12b11**	**a11b21**	**0**	c12=a12b11+a22b12
3		**c12**	**c21**	**a12b21**	c21=a11b21+a21b22
4				**c22**	c11=a12b21+a22b22

Рис.3.17. Диаграмма работы систолической матрицы.

В рассмотренных систолических структурах использованы одинаковые ПЭ, хотя принципиально в матрицы можно соединить и разнотипные ПЭ, но обязательно выполняющие лишь одну примитивную операцию. Такие систолические системы позволяют производить достаточно узкий круг преобразований регулярных структур данных - векторов, матриц, цепочек символов, которые характеризуются однородностью как собственно обработки, так и передач. По этой причине далеко не все алгоритмы могут быть сведены к систолическим и во многих случаях приходится отказываться от алгоритмов с меньшей сложностью в пользу более сложных, но регулярных алгоритмов, отвечающих требованиям систолической обработки.

Если в состав каждого ПЭ ввести дополнительные средства обработки, а также средства дешифрации команд, то можно организовать систолическую матрицу с командным управлением. В такой матрице наряду с потоком операндов существует поток команд, продвигаемых по ПЭ матрицы аналогично элементам данных. Поступающая в ПЭ команда настраивает его на определенную операцию; эта настройка сохраняется в течение одного тактового интервала, т.е. до поступления следующей команды в данный ПЭ, а предыдущая команда из него передается на следующий ПЭ.

Дальнейшее расширение функциональных возможностей систолических ВС связано с увеличением вычислительной мощности каждого ПЭ, т.е. введение в его состав локальной памяти команд и данных, усложнением АЛУ, расширением системы команд ПЭ, введением средств приема и дешифрации команд, а также организацией произвольной настройки ПЭ. Такие ПЭ в систолических ВС получили название программируемых, а составленная из них систолическая матрица - программируемой. Программируемые систолические матрицы предполагалось использовать в качестве базы для создания систолических ВС, рассчитанных на достаточно широкий класс задач. Однако создание универсальных ВС с систолической архитектурой связано с рядом проблем. Одна из них заключается в том, что если в каждом ПЭ требуется выполнение операций из достаточно большого набора, то резко усложняется организация соединений, увеличивается время на настройку ПЭ и, как следствие, падает производительность. Вторая проблема - более низкое быстродействие по сравнению с операционными конвейерами; поэтому в некоторые ПЭ систолической матрицы могут внедряться конвейеры операций. Указанные трудности практиче-

ски не проявляются в специализированных ВС и именно поэтому систолические архитектуры характерны именно для специализированных систем и процессоров.

Промышленно выпускается несколько типов СБИС, специально предназначенных для использования в систолических матрицах. Эти СБИС позволяют проектировать системы различного назначения. Например, фирма **IBM** разработала кМОП СБИС, служащую для реализации суммирования попарных произведений. Схема предназначена для обработки изображений и формирования диаграммы направленности гидролокатора; она содержит два умножителя 16х16, два 16-разрядных сумматора, два 24-разрядных сумматора, регистры и схемы управления. Также для обработки изображений служит кМОПСБИС NCR45CG72 - систолическая матрица из 6 х 12 одноразрядных ПЭ. Каждый ПЭ содержит АЛУ, ОЗУ на 128 разрядов и ряд регистров; предусмотрено внешнее управление ПЭ по управляющим и адресным линиям. Быстродействие СБИС соответствует 28 млн. сложений 8-разрядных слов в секунду.

Для систолической ВС **WARP** был разработан комплект, включающий СБИС 32-разрядного накопителя попарных произведений, АЛУ и блока регистров, который содержит порты ввода-вывода (4), мультиплексор (2) и буферные регистры (4). ПЭ, построенный из перечисленных схем, работает на частоте 20 Мгц и обеспечивает быстродействие 10Мфлопс. В состав полной системы WARP входит 10 таких ПЭ, соединенных в линейную систолическую матрицы. Матрица подключается в качестве периферийного процессора к ведущей ЭВМ, работающей под управлением ОС UNIX, и выполняет роль спецпроцессора. Большая вычислительная мощность ПЭ и крупноблочная структура матрицы WARP позволяет использовать систему для решения достаточно сложных задач вычислительного характера.

Следующим шагом на пути создания систолических ВС стала разработка семейства микропроцессоров фирмы INMOS, получивших название *"транспьютер"*. Основными чертами транспьютеров являются RISC-архитектура процессора, наличие четырех каналов последовательной связи на одном кристалле с процессором, наличие внутреннего ОЗУ и интерфейса расширения ОЗУ. Семейство транспьютеров и система поддержки проектирования систем на базе этого микропроцессора оказались очень удачными не только для создания систолических структур, но и для других конфигураций. Более подробно системы на базе транспьютеров рассмотрены.

3.3. Слабосвязанные потоки

Следующий путь ускорения вычислений за счет параллельного выполнения операций - это использование алгоритмов, структура которых представляет собой совокупность слабосвязанных потоков команд (последовательных процессов). В этом случае программа распадается на несколько последовательных процессов, между которыми существует относительно небольшое число взаимосвязей (см. рис.1*в*). Каждый процесс может исполняться на отдельном последовательном процессоре, который при необходимости осуществляет взаимодействие с другими процессорами. Такие вычислительные системы называются *многопроцессорными*, и ключевым элементом в них является механизм синхронизации и взаимосвязи между процессами.

Одной из отличительных особенностей многопроцессорной вычислительной системы является сеть обмена, с помощью которой процессоры соединяются друг с другом или с памятью. Модель обмена настолько важна для многопроцессорной системы, что многие характеристики производительности и другие оценки выражаются отношением времени обработки к времени обмена, соответствующим решаемым задачам. Существуют две основные модели межпроцессорного обмена: одна основана на **передаче сообщений,** другая - на использовании **общей памяти.**

В многопроцессорной системе с общей памятью один процессор осуществляет запись в конкретную ячейку, а другой процессор производит считывание из этой ячейки памяти. Чтобы обеспечить согласованность данных и синхронизацию процессов, обмен часто реализуется по принципу взаимно исключающего доступа к общей памяти методом "почтового ящика". Механизм, построенный на основе общей памяти, является более простым, но для его реализации необходимо, чтобы все процессы имели доступ к одной и той же памяти, а это в, свою очередь, накладывает ограничение на число процессоров.

В архитектурах с локальной памятью непосредственное разделение памяти невозможно. Вместо этого процессоры получают доступ к совместно используемым данным посредством передачи сообщений по сети обмена. С механизмом, базирующимся на обмене сообщениями, связаны значительные накладные временные расходы, из-за которых его использование оказывается возможным только в тех случаях, когда необходимость синхронизации и обмена данными между процессами возникает крайне редко.

Основные отличия между ними состоят в организации памяти и пропускной способности коммутационного устройства. В первом случае для уменьшения задержек требуется, чтобы коммутационное устройство имело высокое быстродействие, а память была организована таким образом, чтобы число конфликтов доступа было минимальным. Один из способов достижения этого - иметь в каждом процессоре локальную быстродействующую память (**кэш-память**). Во втором случае по мере возрастания требований к обмену следует учитывать возможность перегрузки сети. Объем передаваемой информации может быть сокращен за счет тщательной функциональной декомпозиции задачи и тщательного диспетчирования выполняемых функций.

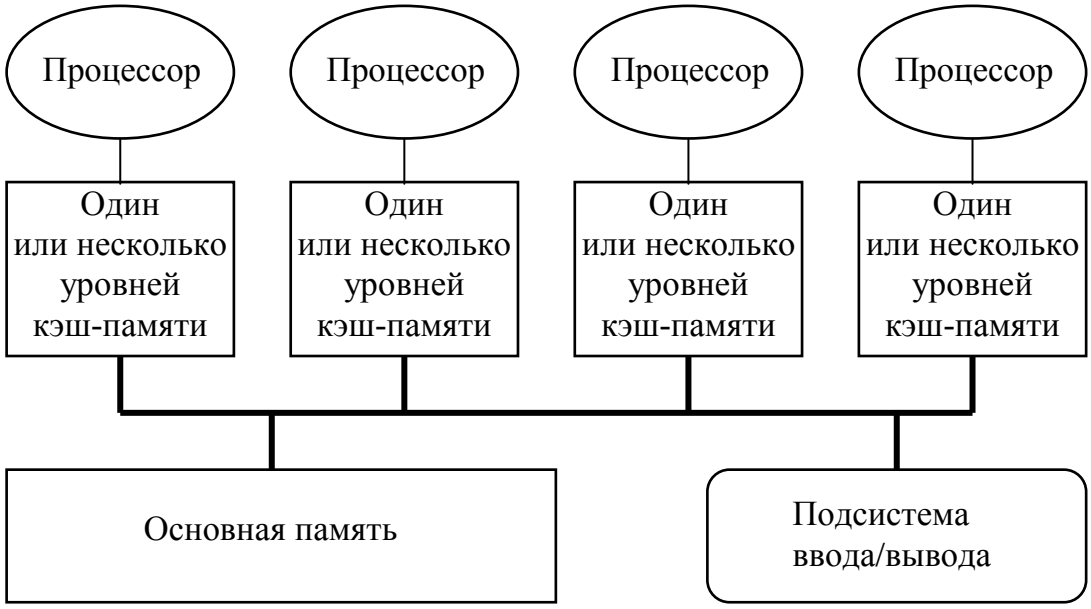


Рис.3.18. Типовая архитектура мультипроцессорной системы с общей памятью.

Таким образом, существующие многопроцессорные системы распадаются на два основных класса в зависимости от способа организации памяти и методики их межсоединения. К первой группе относятся **машины с общей (разделяемой) основной памятью** (*True shared memory)*, объединяющие до нескольких десятков (обычно менее 32) процессоров. Сравнительно небольшое количество процессоров в таких машинах позволяет иметь одну централизованную общую память и объединить процессоры и память с помощью одной шины. При наличии у процессоров кэш-памяти достаточного объема высокопроизводительная шина и общая память могут удовлетворить обращения к памяти, поступающие от нескольких процессоров. Поскольку имеется единственная память с одним и

тем же временем доступа, эти машины иногда называются **UMA** (*Uniform Memory Access*). Структура подобной системы представлена на рис.3.18.

Однако в связи с резким увеличением производительности процессоров и связанным с этим ужесточением требований к увеличению полосы пропускания памяти, масштаб систем (т.е. число процессоров в системе) уменьшается, так как уменьшается число процессоров, которые удается поддерживать на одной разделяемой шине и общей памяти. С ростом числа процессоров просто невозможно обойти вниманием необходимость реализации модели распределенной памяти с высокоскоростной сетью для связи процессоров.

Поэтому вторую группу машин составляют крупномасштабные **системы с распределенной памятью** (*Distributed memory)***.** Для того чтобы поддерживать большое количество процессоров приходится распределять основную память между ними, в противном случае полосы пропускания памяти просто может не хватить для удовлетворения запросов, поступающих от очень большого числа процессоров. Естественно при таком подходе также требуется реализовать связь процессоров между собой. На рис.3.19 показана структура такой системы.

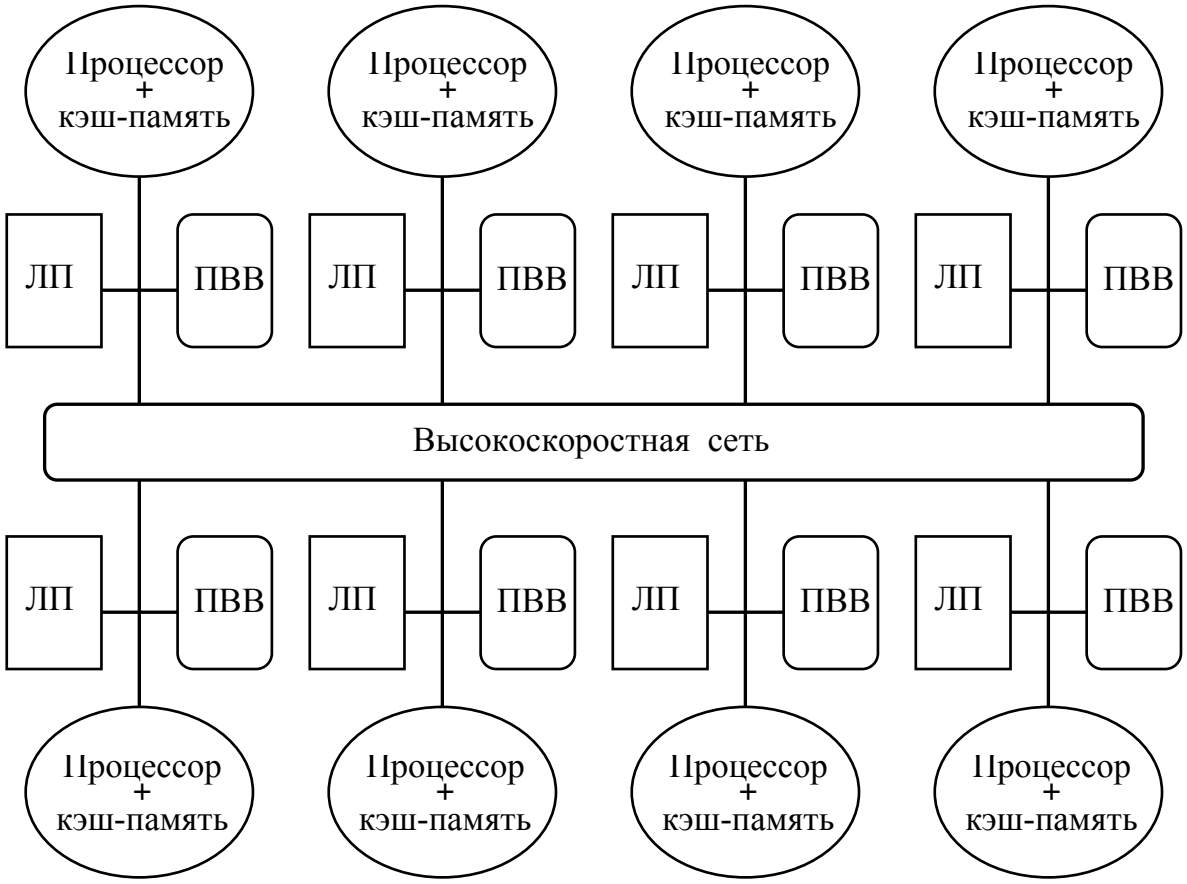


Рис.3.19. Типовая архитектура машин с распределенной памятью.
(ЛП – локальная память, ПВВ –подсистема ввода/вывода)

Распределение памяти между отдельными узлами системы имеет два главных преимущества. Во-первых, это эффективный (с точки зрения стоимости) способ увеличения полосы пропускания памяти, поскольку большинство обращений могут выполняться параллельно к локальной памяти в каждом узле. Во-вторых, это уменьшает задержку обращения (время доступа) к локальной памяти.

Обычно устройства ввода/вывода, также как и память, распределяются по узлам и в действительности узлы могут состоять из небольшого числа (2-8) процессоров, соединен-

ных между собой некоторым способом. Хотя такая кластеризация нескольких процессоров с памятью и сетевой интерфейс могут быть достаточно полезными с точки зрения эффективности в стоимостном выражении, это не очень существенно для понимания того, как такая машина работает, поэтому мы пока остановимся на системах с одним процессором на узел.

Как уже было отмечено, любая крупномасштабная многопроцессорная система должна использовать множество устройств памяти, которые физически распределяются вместе с процессорами. Имеется две альтернативных организации адресации этих устройств памяти и связанных с этим *два альтернативных метода* передачи данных между процессорами. Физически отдельные устройства памяти могут адресоваться как логически единое адресное пространство, что означает, что любой процессор может выполнять обращения к любым ячейкам памяти, предполагая, что он имеет соответствующие права доступа. Такие системы называются машинами с **распределенной разделяемой (общей) памятью** (*DSM - distributed shared memory*), или машины с виртуальной разделяемой (общей) памятью (*virtual shared memory)*, а иногда **NUMA** системы - *Non-Uniform Memory Access*, поскольку время доступа зависит от расположения ячейки в памяти. В альтернативном случае, адресное пространство состоит из отдельных адресных пространств, которые логически не связаны и доступ к которым не может быть осуществлен аппаратно другим процессором. В таком примере каждый модуль процессор-память представляет собой отдельный компьютер, поэтому такие системы называются **многомашинными** (*multicomputers*).

С каждой из этих организаций адресного пространства связан свой механизм обмена. Для машины с единым адресным пространством оно может быть использовано для обмена данными посредством операций загрузки и записи. Это определяет следующие преимущества:
совместимость с хорошо понятными используемыми как в однопроцессорных, так и маломасштабных многопроцессорных системах, механизмами обмена.

- простота программирования (особенно когда модели обмена между процессорами сложные или динамически меняются во время выполнения), так как проблемы обмена в значительной степени скрыты от программиста.
- более низкая задержка обмена и лучшее использование полосы пропускания при обмене малыми порциями данных.
- возможность использования аппаратно управляемого кэширования для снижения частоты удаленного обмена, что допускает кэширование всех данных как разделяемых, так и неразделяемых.

Основными преимуществами обмена с помощью передачи сообщений являются:
аппаратная поддержка обмена может быть более простой, особенно по сравнению с моделью разделяемой памяти, которая поддерживает масштабируемую когерентность кэш-памяти.

- модели обмена понятны, принуждают программистов (или компиляторы) уделять внимание обмену (минимизировать), который, обычно, имеет высокую временную стоимость.

Конечно, основные трудности возникают при работе с сообщениями, которые могут быть неправильно выровнены и сообщениями произвольной длины в системе памяти, которая обычно ориентирована на передачу выровненных блоков данных, организованных как блоки кэш-памяти. Эти трудности можно преодолеть либо с небольшими потерями производительности программным способом, либо существенно без потерь при использовании небольшой аппаратной поддержки.

При оценке любого механизма обмена критичными являются три характеристики производительности:

1. *Полоса пропускания:* в идеале полоса пропускания механизма обмена будет ограничена полосами пропускания процессора, памяти и системы межсоединений, а не какими-либо аспектами механизма обмена. Связанные с механизмом обмена накладные расходы (например, длина межпроцессорной связи) прямо воздействуют на полосу пропускания.

2. *Задержка:* в идеале задержка должна быть настолько мала, насколько это возможно. Для ее определения важны накладные расходы аппаратуры и программного обеспечения, связанные с инициированием и завершением обмена.

3. *Скрытие (упрятывание) задержки:* насколько хорошо механизм скрывает задержку путем перекрытия обмена с вычислениями или с другими обменами.

Каждый из этих параметров воздействует на характеристики обмена. В частности, задержка и полоса пропускания могут меняться в зависимости от размера элемента данных. В общем случае, механизм, который одинаково хорошо работает как с небольшими, так и с большими объемами данных будет более гибким и эффективным.

Таким образом, отличия машин с распределенной памятью определяются моделью памяти и механизмом обмена.

Исторически машины с распределенной памятью первоначально были построены с использованием механизма передачи сообщений, поскольку это было очевидно проще, и многие разработчики не верили, что единое адресное пространство можно построить и в машинах с распределенной памятью. С недавнего времени модели обмена с общей памятью действительно начали поддерживаться практически в каждой разработанной машине (характерным примером могут служить системы с симметричной мультипроцессорной обработкой).

Хотя машины с централизованной общей памятью, построенные на базе общей шины все еще доминируют на компьютерном рынке по количеству проданных систем, долговременные технические тенденции направлены на использование преимуществ распределенной памяти даже в машинах "умеренного размера". При этом, возможно, наиболее важным вопросом, который встает при создании машин с распределенной памятью, является вопрос о кэшировании и когерентности кэш-памяти.

Рассмотрим два типа (или, точнее, три, с учетом деления систем второго типа на два достаточно различных класса) многопроцессорных систем более подробно.

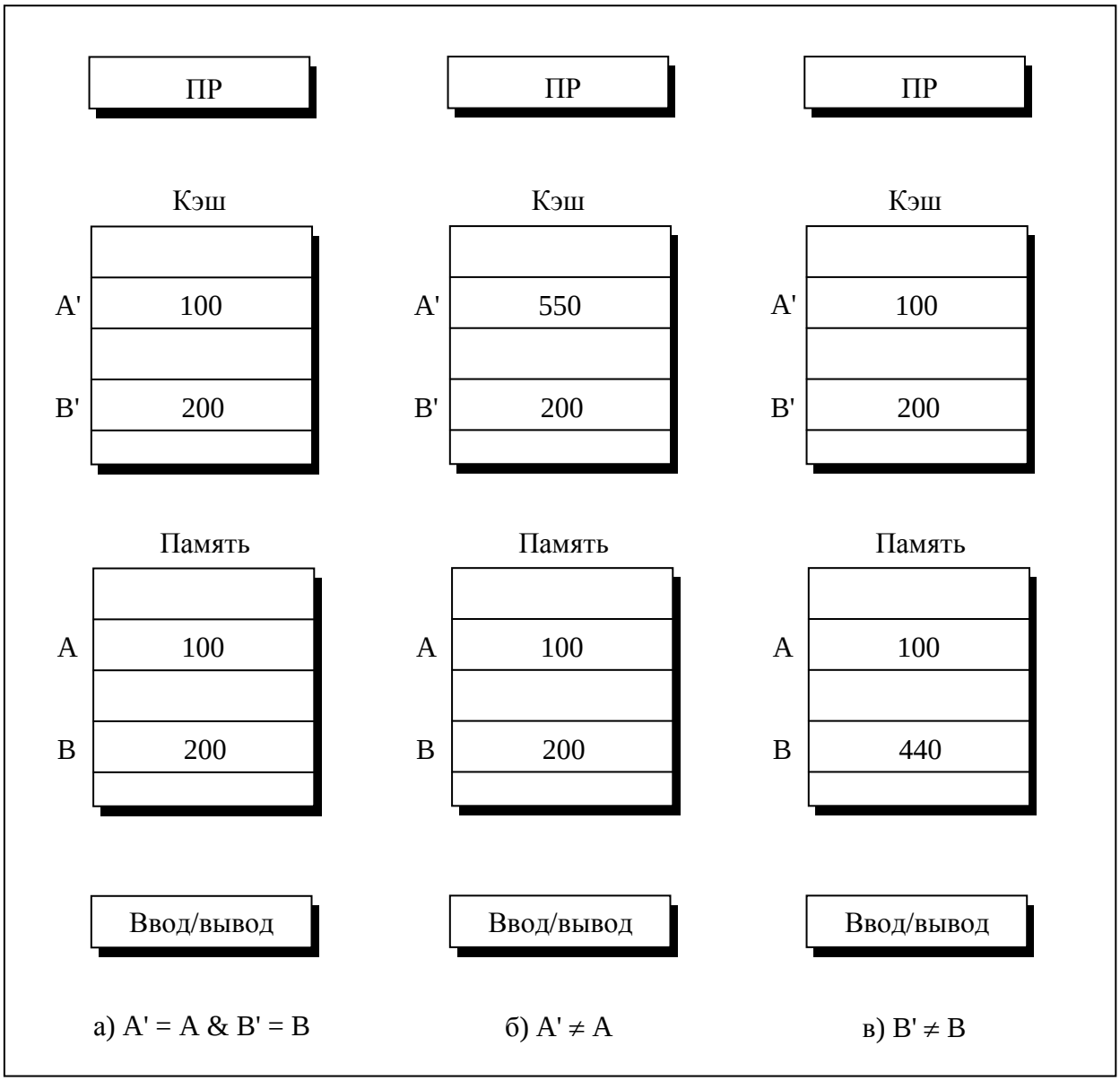
3.3.1. Многопроцессорные системы с общей памятью

Требования, предъявляемые современными процессорами к полосе пропускания памяти, можно существенно сократить путем применения больших многоуровневых кэшей. Тогда несколько процессоров смогут разделять доступ к одной и той же памяти.

Начиная с 1980 года эта идея, подкрепленная широким распространением микропроцессоров, стимулировала многих разработчиков на создание небольших мультипроцессоров, в которых несколько процессоров разделяют одну физическую память, соединенную с ними с помощью разделяемой шины. Из-за малого размера процессоров и заметного сокращения требуемой полосы пропускания шины, достигнутого за счет возможности реализации достаточно большой кэш-памяти, такие машины стали исключительно эффективными по стоимости.

В первых разработках подобного рода машин удавалось разместить весь процессор и кэш на одной плате. Плата затем вставлялась в заднюю панель, с помощью которой реализовывалась шинная архитектура. Современные конструкции позволяют разместить до четырех процессоров на одной плате (см. рис.3.18.)

В такой машине кэши могут содержать как разделяемые, так и частные данные. Частные данные - это данные, которые используются одним процессором, в то время как разделяемые данные используются многими процессорами, по существу обеспечивая обмен между ними. Когда кэшируется элемент частных данных, их значение переносится в кэш для сокращения среднего времени доступа, а также требуемой полосы пропускания. Поскольку никакой другой процессор не использует эти данные, этот процесс идентичен процессу для однопроцессорной машины с кэш-памятью.



а) Кэш и память когерентны. A' и B' – кэшированные копии элементов A и B в основной памяти. ПР – процессор.

б) Кэш и память некогерентны. Предполагается использование кэш-памяти с отложенным обратным копированием, когда ПР записывает значение 550 в ячейку A. В результате A' содержит новое значение, а в основной памяти осталось старое значение 100. При попытке вывода A из памяти будет получено новое решение.

в) Кэш и память некогерентны. Подсистема ввода/вывода вводит в ячейку памяти B новое значение 440, а в кэш-памяти осталось старое значение B.

Рис. 3.20. Иллюстрация проблемы когерентности кэш-памяти.

Если кэшируются разделяемые данные, то разделяемое значение реплицируется и может содержаться в нескольких кэшах. Кроме сокращения задержки доступа и требуемой полосы пропускания такая репликация данных способствует также общему сокращению количества обменов. Однако кэширование разделяемых данных вызывает новую проблему - *когерентности кэш-памяти.* Проблема возникает из-за того, что значение элемента данных в памяти, хранящееся в двух разных процессорах, доступно этим процессорам только через их индивидуальные кэши (рис.3.20).

Проблема когерентности памяти для мультипроцессоров и устройств ввода/вывода имеет много аспектов. Обычно в малых мультипроцессорах используется аппаратный механизм, называемый протоколом когерентности кэш-памяти. Существуют два класса таких протоколов:

1. Протоколы на основе справочника (*directory based*). Информация о состоянии блока физической памяти содержится только в одном месте, называемом справочником (физически справочник может быть распределен по узлам системы).

2. Протоколы наблюдения (*snooping*). Каждый кэш, который содержит копию данных некоторого блока физической памяти, имеет также соответствующую копию служебной информации о его состоянии. Централизованная система записей отсутствует. Обычно кэши расположены на общей (разделяемой) шине и контроллеры всех кэшей наблюдают за шиной (просматривают ее) для определения того, не содержат ли они копию соответствующего блока.

В мультипроцессорных системах, использующих микропроцессоры с кэш-памятью, подсоединенные к централизованной общей памяти, протоколы наблюдения приобрели популярность, поскольку для опроса состояния кэшей они могут использовать заранее существующее физическое соединение - шину памяти.

Неформально, проблема когерентности памяти состоит в необходимости гарантировать, что любое считывание элемента данных возвращает последнее по времени записанное в него значение. Это определение не совсем корректно, поскольку невозможно требовать, чтобы операция считывания мгновенно видела значение, записанное в этот элемент данных некоторым другим процессором. Если, например, операция записи на одном процессоре предшествует операции чтения той же ячейки на другом процессоре в пределах очень короткого интервала времени, то невозможно гарантировать, что чтение вернет записанное значение данных, поскольку в этот момент времени записываемые данные могут даже не покинуть процессор. Вопрос о том, когда точно записываемое значение должно быть доступно процессору, выполняющему чтение, определяется выбранной моделью согласованного (непротиворечивого) состояния памяти и связан с реализацией синхронизации параллельных вычислений. Поэтому, с целью упрощения, предположим, что мы требуем только, чтобы записанное операцией записи значение было доступно операции чтения, возникшей немного позже записи и что операции записи данного процессора всегда видны в порядке их выполнения.

С этим простым определением согласованного состояния памяти мы можем гарантировать когерентность путем обеспечения двух свойств:

1. Операция чтения ячейки памяти одним процессором, которая следует за операцией записи в ту же ячейку памяти другим процессором получит записанное значение, если операции чтения и записи достаточно отделены друг от друга по времени.

2. Операции записи в одну и ту же ячейку памяти выполняются строго последовательно (иногда говорят, что они сериализованы). Это означает, что две подряд идущие операции записи в одну и ту же ячейку памяти будут наблюдаться другими процессорами именно в том порядке, в котором они появляются в программе процессора, выполняющего эти операции записи.

Первое свойство очевидно связано с определением когерентного (согласованного) состояния памяти: если бы процессор всегда считывал только старое значение данных, мы сказали бы, что память некогерентна.

Необходимость строго последовательного выполнения операций записи является более тонким, но также очень важным свойством. Представим себе, что процессор P1 должен записать данные в ячейку памяти, а затем в эту ячейку выполнит запись процессор P2. Если последовательность операций записи не соблюдается, то может возникнуть ситуация, когда какой-нибудь процессор будет наблюдать сначала операцию записи процессора P2, а затем операцию записи процессора P1, и будет хранить это, записанное P1, значение неограниченно долго. Другая проблема возникает с поддержанием разумной модели порядка выполнения программ и когерентности памяти для пользователя. Представим, что третий процессор постоянно читает ту же самую ячейку памяти, в которую записывают информацию процессоры P1 и P2. Он никогда не сможет увидеть значения, записанного P1, если запись от P2 возникла раньше операции чтения. Поэтому вопрос строгого соблюдения порядка операций записи очень важен в многопроцессорных ВС. Вопрос о том, **когда** процессор P3 должен увидеть значение, записанное другим процессором (P2), достаточно сложен и имеет заметное воздействие на производительность, особенно в системах с очень большим количеством процессоров.

Альтернативные протоколы. Имеются две методики поддержания описанной выше когерентности. Один из методов заключается в том, чтобы гарантировать, что процессор должен получить исключительные права доступа к элементу данных перед выполнением записи в этот элемент данных. Такой тип протоколов называется **протоколом записи с аннулированием** *(write ivalidate protocol),* поскольку при выполнении записи он аннулирует другие копии. Это наиболее часто используемый протокол как в схемах на основе справочников, так и в схемах наблюдения. Исключительное право доступа гарантирует, что во время выполнения записи не существует никаких других копий элемента данных, в которые можно писать или из которых можно читать: все другие кэшированные копии элемента данных аннулированы. Чтобы увидеть, как такой протокол обеспечивает когерентность, рассмотрим операцию записи, вслед за которой следует операция чтения другим процессором. Поскольку запись требует исключительного права доступа, любая копия, поддерживаемая читающим процессором должна быть аннулирована (в соответствии с названием протокола). Когда возникает операция чтения, происходит промах кэш-памяти, который вынуждает выполнить выборку новой копии данных. Для выполнения операции записи мы можем потребовать, чтобы процессор имел *достоверную (valid)* копию данных в своей кэш-памяти прежде, чем выполнять в нее запись. Таким образом, если два процессора попытаются записать в один и тот же элемент данных одновременно, один из них выиграет состязание у второго и вызовет аннулирование его копии. Другой процессор для завершения своей операции записи должен сначала получить новую копию данных, которая теперь уже должна содержать обновленное значение.

Альтернативой записи с аннулированием является обновление всех копий элемента данных в случае записи в этот элемент данных. Такой тип протокола называется **протоколом записи с обновлением** *(write update protocol)* или **протоколом записи с трансляцией** *(write broadcast protocol).* Обычно в этом протоколе для снижения требований к полосе пропускания полезно отслеживать, является ли слово в кэш-памяти разделяемым объектом, или нет, а именно, содержится ли оно в других кэшах. Если нет, то нет никакой необходимости обновлять другой кэш или транслировать в него обновленные данные.

Разница в производительности между протоколами записи с обновлением и с аннулированием определяется тремя характеристиками:

1. Несколько последовательных операций записи в одно и то же слово, не перемежающихся операциями чтения, требуют нескольких операций трансляции при использо-

вании протокола записи с обновлением, но только одной начальной операции аннулирования при использовании протокола записи с аннулированием.

2. При наличии многословных блоков в кэш-памяти каждое слово, записываемое в блок кэша, требует трансляции при использовании протокола записи с обновлением, в то время как только первая запись в любое слово блока нуждается в генерации операции аннулирования при использовании протокола записи с аннулированием. Протокол записи с аннулированием работает на уровне блоков кэш-памяти, в то время как протокол записи с обновлением должен работать на уровне отдельных слов (или байтов, если выполняется запись байта).

3. Задержка между записью слова в одном процессоре и чтением записанного значения другим процессором обычно меньше при использовании схемы записи с обновлением, поскольку записанные данные немедленно транслируются в процессор, выполняющий чтение (предполагается, что этот процессор имеет копию данных). Для сравнения, при использовании протокола записи с аннулированием в процессоре, выполняющим чтение, сначала произойдет аннулирование его копии, затем будет производиться чтение данных и его приостановка до тех пор, пока обновленная копия блока не станет доступной и не вернется в процессор.

Эти две схемы во многом похожи на схемы работы кэш-памяти со сквозной записью и с записью с обратным копированием. Также как и схема задержанной записи с обратным копированием требует меньшей полосы пропускания памяти (она использует преимущества операций над целым блоком), протокол записи с аннулированием обычно требует менее тяжелого трафика, чем протокол записи с обновлением, поскольку несколько записей в один и тот же блок кэш-памяти не требуют трансляции каждой записи. При сквозной записи память обновляется почти мгновенно после записи (возможно с некоторой задержкой в буфере записи). Подобным образом при использовании протокола записи с обновлением другие копии обновляются так быстро, насколько это возможно. Наиболее важное отличие в производительности протоколов записи с аннулированием и с обновлением связано с характеристиками прикладных программ и с выбором размера блока.

Основы реализации. Ключевым моментом в реализации многопроцессорных систем с небольшим числом процессоров является использование механизма шины для выполнения операции обновления или аннулирования. Процессор просто захватывает шину и транслирует по ней адрес, по которому должно производиться обновление или аннулирование данных. Все процессоры непрерывно наблюдают за шиной, контролируя появляющиеся на ней адреса. Процессоры проверяют не находится ли в их кэш-памяти адрес, появившийся на шине. Если это так, то соответствующие данные в кэше либо аннулируются, либо обновляются в зависимости от используемого протокола. Последовательный порядок обращений, присущий шине, обеспечивает также строго последовательное выполнение операций записи, поскольку когда два процессора конкурируют за выполнение записи в одну и ту же ячейку, один из них должен получить доступ к шине раньше другого.

В дополнение к аннулированию или обновлению соответствующих копий блока кэш-памяти, в который производилась запись, необходимо также разместить элемент данных, если при записи происходит промах кэш-памяти. В кэш-памяти со сквозной записью последнее значение элемента данных найти легко, поскольку все записываемые данные всегда посылаются также и в память, из которой последнее записанное значение элемента данных может быть выбрано (наличие буферов записи может привести к некоторому усложнению). Однако для кэш-памяти с обратным копированием задача нахождения последнего значения элемента данных сложнее, поскольку это значение скорее всего находится в кэше, а не в памяти. В этом случае используется та же самая схема наблюдения, что и при записи: каждый процессор наблюдает и контролирует адреса, помещаемые на шину. Если процессор обнаруживает, что он имеет модифицированную ("грязную") ко-

пию блока кэш-памяти, то именно он должен обеспечить пересылку этого блока в ответ на запрос чтения и вызвать отмену обращения к основной памяти. Поскольку кэши с обратным копированием предъявляют меньшие требования к полосе пропускания памяти, они намного предпочтительнее в мультипроцессорах, несмотря на некоторое увеличение сложности. Поэтому далее мы рассмотрим вопросы реализации кэш-памяти с обратным копированием.

Для реализации процесса наблюдения могут быть использованы обычные теги кэша. Более того, упоминавшийся ранее *бит достоверности (valid bit),* позволяет легко реализовать аннулирование. Промахи операций чтения, вызванные либо аннулированием, либо каким-нибудь другим событием, также не сложны для понимания, поскольку они просто основаны на возможности наблюдения. Для операций записи мы хотели бы также знать, имеются ли другие кэшированные копии блока, поскольку в случае отсутствия таких копий, запись можно не посылать на шину, что сокращает время на выполнение записи, а также требуемую полосу пропускания.

Чтобы отследить, является ли блок разделяемым, мы можем ввести дополнительный *бит состояния* (shared), связанный с каждым блоком, точно также как это делалось для битов достоверности (valid) и модификации (modified или dirty) блока. Добавив бит состояния, определяющий является ли блок разделяемым, мы можем решить вопрос о том, должна ли запись генерировать операцию аннулирования в протоколе с аннулированием, или операцию трансляции при использовании протокола с обновлением. Если происходит запись в блок, находящийся в состоянии "разделяемый" при использовании протокола записи с аннулированием, кэш формирует на шине операцию аннулирования и помечает блок как *частный* (private). Никаких последующих операций аннулирования этого блока данный процессор посылать больше не будет. Процессор с *исключительной* (exclusive) копией блока кэш-памяти обычно называется "владельцем" (owner) блока кэш-памяти. Когда посылается операция аннулирования, состояние блока меняется с "разделяемый" на "неразделяемый" (или "частный"). Позже, если другой процессор запросит этот блок, состояние снова должно измениться на "разделяемый". Поскольку наш наблюдающий кэш видит также все промахи, он знает, когда этот блок кэша запрашивается другим процессором, и его состояние должно стать "разделяемый".

При использовании протокола записи с обновлением, если блок находится в состоянии "разделяемый", то каждая запись в этот блок должна транслироваться.

Поскольку любая транзакция на шине контролирует адресные теги кэша, потенциально это может приводить к конфликтам обращения к кэшу со стороны процессора. Число таких потенциальных конфликтов можно снизить применением одного из двух методов: *дублированием тегов*, или *использованием многоуровневых кэшей* с "охватом" (inclusion), когда уровни, находящиеся ближе к процессору являются поднабором уровней, находящихся дальше от него.

Если теги дублируются, то обращения процессора к кэшу и наблюдение за шиной могут выполняться параллельно. Конечно, если при обращении процессора происходит промах, он должен будет выполнять арбитраж с механизмом наблюдения для обновления обоих наборов тегов.

Точно также, если механизм наблюдения за шиной находит совпадающий тег, ему будет нужно проводить арбитраж и обращаться к обоим наборам тегов кэша (для выполнения аннулирования или обновления бита "разделяемый"). Таким образом, при использовании схемы дублирования тегов процессор должен приостановиться только в том случае, если он выполняет обращение к кэшу в тот же самый момент времени, когда механизм наблюдения обнаружил копию в кэше. Более того, активность механизма наблюдения задерживается только тогда, когда кэш имеет дело с промахом.

Таблица 3.3. Примеры протоколов наблюдения

Наименование	Тип протокола	Стратегия записи в память	Уникальные свойства. Применение
Одиночная запись	Запись с аннулированием	Обратное копирование при первой записи	Первый описанный в литературе протокол наблюдения
Synapse N+1	Запись с аннулированием	Обратное копирование	Точное состояние, где "владельцем является память" Машины Synapse Первые машины с когерентной кэш-памятью
Berkely	Запись с аннулированием	Обратное копирование	Состояние "разделяемый" Машина SPUR университета Berkely
Illinois	Запись с аннулированием	Обратное копирование	Состояние "приватный" может передавать данные из любого кэша Серии Power и Challenge компании Silicon Graphics
"Firefly"	Запись с трансляцией	Обратное копирование для "приватных" блоков и сквозная запись для "разделяемых"	Обновление памяти во время трансляции SPARCcenter 2000

Если процессор использует многоуровневый кэш со свойствами охвата, тогда каждая строка в основном кэше имеется и во вторичном кэше. Таким образом, активность по наблюдению может быть связана с кэшем второго уровня, в то время как большинство активностей процессора могут быть связаны с первичным кэшем. Если механизм наблюдения получает попадание во вторичный кэш, тогда он должен выполнять арбитраж за первичный кэш, чтобы обновить состояние и возможно найти данные, что обычно будет приводить к приостановке процессора. Такое решение было принято во многих современных системах, поскольку многоуровневый кэш позволяет существенно снизить требования к полосе пропускания. Иногда может быть даже полезно дублировать теги во вторичном кэше, чтобы еще больше сократить количество конфликтов между "активностями" процессора и механизма наблюдения.

В реальных системах существует много вариаций схем когерентности кэша, в зависимости от того используется ли схема на основе аннулирования или обновления, построена ли кэш-память на принципах сквозной или обратной записи, когда происходит обновление, а также имеет ли место состояние "владения" и как оно реализуется. В таблице 3.2 представлены несколько протоколов с наблюдением и некоторые машины, которые используют эти протоколы.

3.3.2. Многопроцессорные системы с локальной памятью и многомашинные системы

Существуют два различных способа построения крупномасштабных систем с распределенной памятью. Простейший способ заключается в том, чтобы исключить аппаратные механизмы, обеспечивающие когерентность кэш-памяти, и сосредоточить внимание на создании масштабируемой системы памяти. Наиболее известным примером такой системы является компьютер T3D компании Cray Research. В этих машинах память распределяется между узлами (процессорными элементами) и все узлы соединяются между со-

бой посредством того или иного типа сети. Доступ к памяти может быть локальным или удаленным. Специальные контроллеры, размещаемые в узлах сети, могут на основе анализа адреса обращения принять решение о том, находятся ли требуемые данные в локальной памяти данного узла, или размещаются в памяти удаленного узла. В последнем случае контроллеру удаленной памяти посылается сообщение для обращения к требуемым данным.

Чтобы обойти проблемы когерентности, разделяемые (общие) данные не кэшируются. Конечно, с помощью программного обеспечения можно реализовать некоторую схему кэширования разделяемых данных путем их копирования из общего адресного пространства в локальную память конкретного узла. В этом случае когерентностью памяти также будет управлять программное обеспечение. Преимуществом такого подхода является практически минимальная необходимая поддержка со стороны аппаратуры, хотя наличие, например, таких возможностей как блочное (групповое) копирование данных было бы весьма полезным. Недостатком такой организации является то, что механизмы программной поддержки когерентности подобного рода кэш-памяти компилятором весьма ограничены. Существующая в настоящее время методика в основном подходит для программ с хорошо структурированным параллелизмом на уровне программного цикла.

Таблица 3.4 Характеристики межсоединений некоторых коммерческих MPP

Фирма	Название	Количество узлов	Базовая топология	Разрядность связи (бит) Частота синхронизации (Мгц) Пиковая полоса пропускания (Мб/с) Общая полоса пропускания (Мб/с) Год выпуска				
Thinking Machines	CM-2	1024-4096	12-мер-ный куб	1	7	1	1024	1987
nCube	nCube/ten	1-1024	10-мер-ный куб	1	10	1,2	640	1987
Intel	iPSC/2	16-128	7-мерный куб	1	16	2	345	1988
Maspar	MP-1216	32-512	2-мерная сеть +ступенчатая Omega	1	25	3	1300	1989
Intel	Delta	540	2-мерная сеть	16	40	40	640	1991
Thinking Machines	CM-5	32-2048	многоступенчатое толстое дерево	4	40	20	10240	1991
Meiko	CS-2	2-1024	многоступенчатое толстое дерево	8	70	50	50000	1992
Intel	Paragon	4-1024	2-мерная сеть	16	100	200	6400	1992
Cray Research	T3D	16-1024	3-мерный тор	16	150	300	19200	1993

Системы с архитектурой, подобной Cray T3D, называют машинами с массовым параллелизмом (*MPP - Massively Parallel Processor*). К таким машинам предъявляются взаимно исключающие требования. Чем больше объем системы, тем большее число процессоров она содержит, тем длиннее каналы передачи управления и данных, а значит меньше тактовая частота. Происшедшее возрастание "нормы массивности" для больших машин до 512 и даже 64К процессоров обусловлено не ростом размеров машины, а повышением степени интеграции схем, позволившей за последние годы резко повысить плотность размещения элементов в устройствах. Топология сети обмена между процессорами в такого

рода системах может быть различной. В таблице 3.4 приведены характеристики сети обмена для некоторых коммерческих МРР.

Для построения крупномасштабных систем альтернативой рассмотренному в предыдущем разделе протоколу наблюдения может служить протокол на основе справочника, который отслеживает состояние кэшей. Такой подход предполагает, что логически единый справочник хранит состояние каждого блока памяти, который может кэшироваться. В справочнике обычно содержится информация о том, в каких кэшах имеются копии данного блока, модифицировался ли данный блок и т.д. В существующих реализациях этого направления справочник размещается рядом с памятью. Имеются также протоколы, в которых часть информации размещается в кэш-памяти.

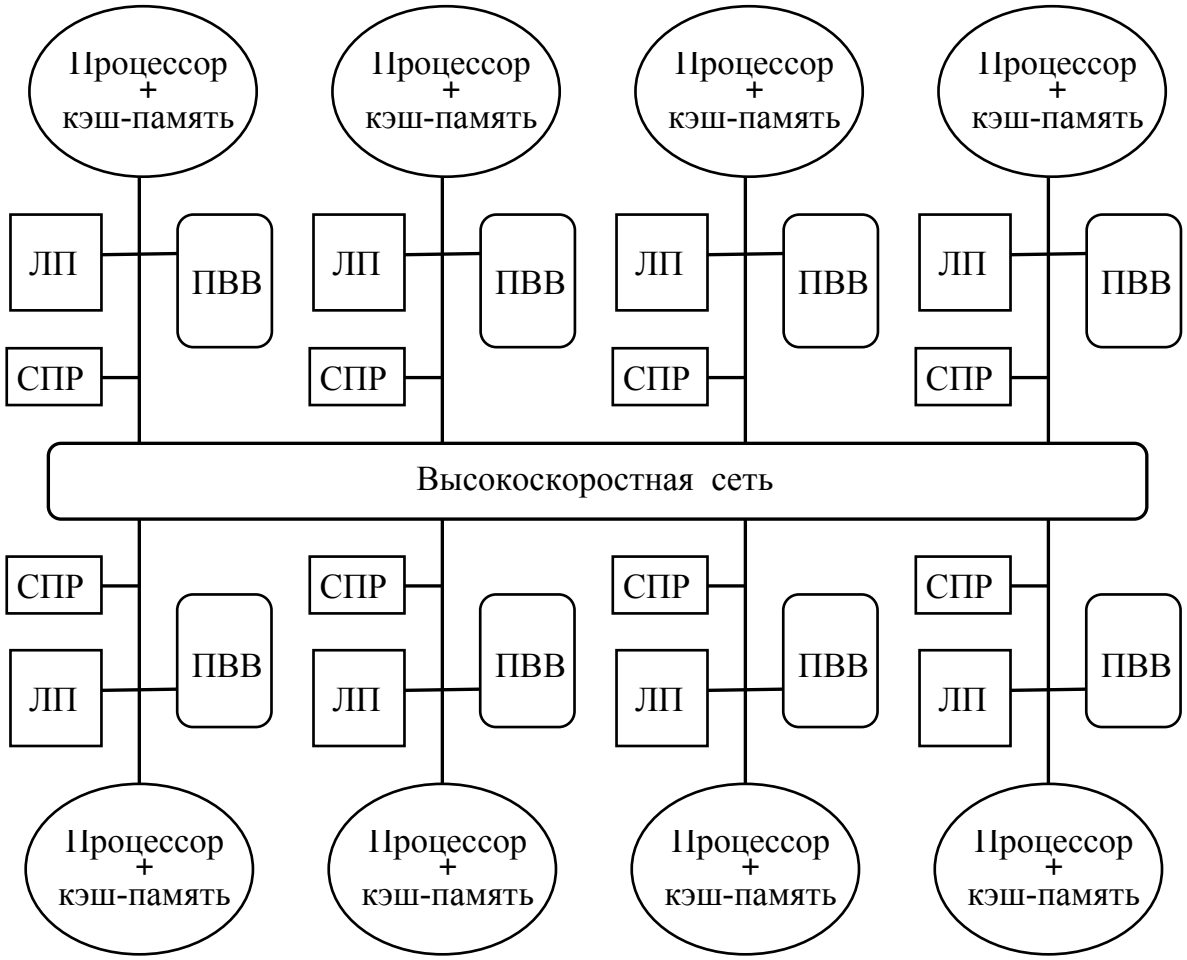


Рис.3.21. Архитектура системы с распределенной внешней памятью и распределенным по узлам справочником.
ЛП – локальная память, ПВВ – подсистема ввода/вывода, СПР – справочник.

Положительной стороной хранения всей информации в едином справочнике является простота протокола, связанная с тем, что вся необходимая информация сосредоточена в одном месте. Недостатком такого рода справочников является его размер, который пропорционален общему объему памяти, а не размеру кэш-памяти. Это не составляет проблемы для машин, состоящих, например, из нескольких сотен процессоров, поскольку связанные с реализацией такого справочника накладные расходы можно преодолеть. Но для машин большего размера необходима методика, позволяющая эффективно масштаби-

ровать структуру справочника.

В частности, чтобы предотвратить появление узкого места в системе с единым справочником, можно поместить части этого справочника на устройства распределенной локальной памяти. Таким образом можно добиться того, что обращения к разным справочникам (частям единого справочника) будут выполняться параллельно, точно также как обращения к локальным блокам памяти. Это существенно увеличит общую полосу пропускания памяти. В распределенном справочнике сохраняется главное свойство подобных схем, заключающееся в том, что состояние любого разделяемого блока данных всегда находится во вполне определенном известном месте. На рис.3.21 показан общий вид подобного рода систем с распределенной памятью. Вопросы детальной реализации протоколов когерентности памяти для таких машин подробно рассмотрены в.

Максимальный коэффициент ускорения, который может быть получен в многопроцессорной системе, состоящей из n процессоров, равен n. К сожалению, реальный коэффициент ускорения оказывается ниже, в силу некоторых отрицательных факторов:

- недостаточный параллелизм алгоритма, т.е. малое число процессов, допускающих параллельное выполнение. Если это число меньше n, то не все процессоры системы оказываются загруженными. По известному закону Амдала ускорение **k**, которое может быть получено на компьютере из **n** процессоров, при доле последовательных операций **s**, присутствующих в алгоритме, определяется как: $k \le \frac{1}{S + (1-S)/n}$.

Следовательно, для того, чтобы эффективно использовать ресурсы многопроцессорной системы, может оказаться целесообразным исполнять одновременно несколько разных программ;

- конфликты и задержки при обращениях к памяти (в случае общей памяти). Эти потери можно уменьшить путем приписывания каждому процессору некоторого числа локальных компонент. Однако при этом использование объема памяти становится менее эффективным;

- значительные накладные расходы, связанные с механизмом синхронизации между потоками, из-за чего эффективность достигается только для слабосвязанных потоков.

3.3.3. Конфигурация ВПВС

Важнейшим элементом архитектуры любого компьютера, а высокопроизводительных вычислительных систем в особенности, являются средства обмена данными между процессором и оперативной памятью, процессором и другим процессором, процессором и устройствами ввода/вывода и т.д.

При использовании систем с общей памятью число процессорных элементов ограничено, поэтому коммутационная сеть представляет собой, в большинстве случаев, обычный матричный коммутатор, а то и вовсе общую шину.

Второй способ организации взаимодействия между устройствами, а именно обмен сообщениями, предназначен для систем с большим числом процессоров. Именно поэтому матричный коммутатор оказывается малопригодным из-за высокой стоимости и недостаточного числа входов, а мультиплексная шина - из-за высокой вероятности возникновения конфликтов.

Многопроцессорные ВС с распределенной памятью состоят из множества вычислительных узлов, или элементарных вычислительных модулей (ВМ), объединенных в единую систему посредством коммуникационной сети. Каждый ВМ, включающий процессор и локальную память, может независимо от других машин выполнять программы, хранящиеся в собственной локальной памяти. Многочисленные ВС с распределенной памятью

различаются реализацией ВМ, реализацией коммуникационной среды, способами распределения задач по ВМ, способами синхронизации и управления процессами.

Было предложено несколько типов коммуникационных сетей с различными топологиями, но наиболее широкое распространение нашли: кольцевая (рис.3.22), в виде двумерной решетки (рис.3.23) или тора (рис.3.24), древовидная (рис.3.25), гиперкубическая (рис.2.36). При пересылке сообщения от ВМ-источника к ВМ-получателю из-за отсутствия прямых связей они могут проходить через промежуточные ВМ-ретрансляторы. В каждой сети существуют наиболее удаленные друг от друга ЭМ, сообщения между которыми проходят через наибольшее число ретрансляторов. Расстояние между наиболее удаленными ВМ, т.е. число связей модуль-модуль, принято называть коммуникационным диаметром; естественно, что он зависит от **N** - общего числа ВМ в системе.

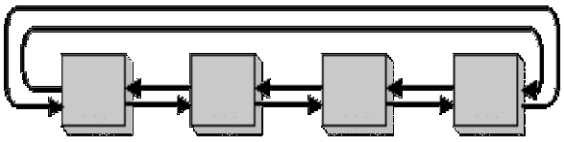


Рис.3.22. Топология кольцо.

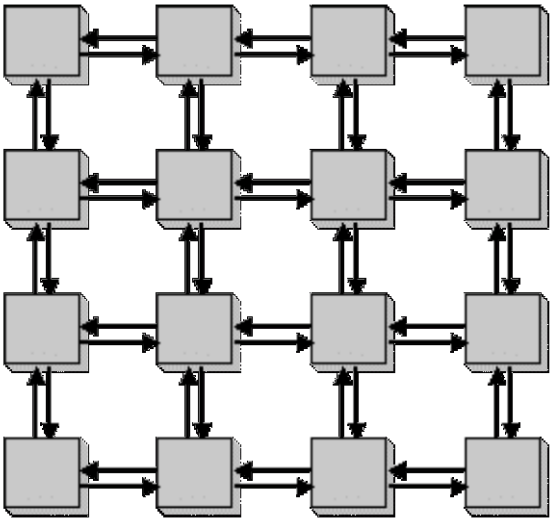


Рис.3.23. Топология 2D-решетка.

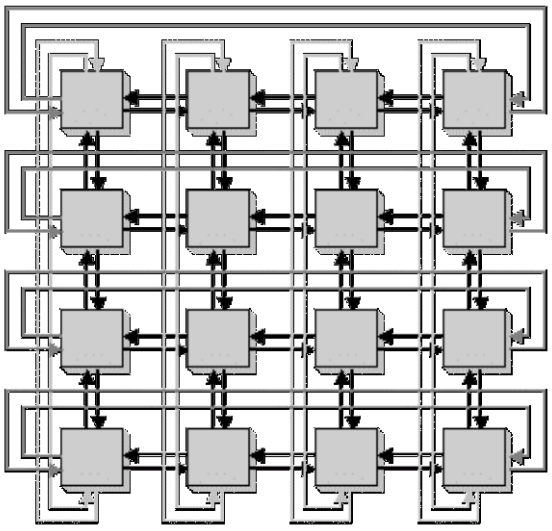


Рис.3.24. Топология 2D-тор.

Кольцевые сети имеют коммуникационный диаметр N/2. В них обычно используются сообщения фиксированного формата, в котором предусмотрен адрес ВМ-назначения. Такие сети наиболее просты и могут быть рекомендованы для ВС с небольшим числом ВМ и малой интенсивностью обменов.

Сеть в виде *двумерной прямоугольной решетки* связывает каждый ВМ с четырьмя ближайшими соседними. Наиболее удаленными являются диагонально расположенные ВМ. Коммуникационный диаметр составляет **2((√N)-1)**. Его можно уменьшить до величины **2(ent(√N)/2)**, если соединить крайние в каждом столбце и строке узлы между собой, т.е. получить топологию - *тор*. Топология таких сетей "совпадает" с топологией задач обработки двумерных полей, плоских изображений и т.п..

Древовидные сети первоначально были разработаны для ВС, предназначенных для реализации алгоритмов поиска и сортировки, поэтому наиболее распространены бинарные деревья. Коммуникационный диаметр для полного двоичного дереве весьма значителен и составляет **2(n-1),** где n - число уровней дерева, т.е. $N = 2^n-1$. Существует множество вариантов его уменьшения, один из наиболее распространенных - введение дополнительных связей между ВМ одного уровня. Однако многие алгоритмы строятся по иерархическому принципу управления, т.е. не требуют непосредственной передачи сообщений между ВМ одного уровня. Именно для такой иерархической организации вычислительного процесса сети типа дерева представляют наибольший интерес.

Сеть в виде *гиперкуба* (двоичного n-мерного куба) объединяет $N=2^n$ ВМ, каждый из которых связан с $\mathbf{n=\log_2 N}$ соседними узлами. Сеть, в основном, ориентирована на решение сложных вычислительных задач (математическая физика, физическая химия, генная инженерия и т.д.).

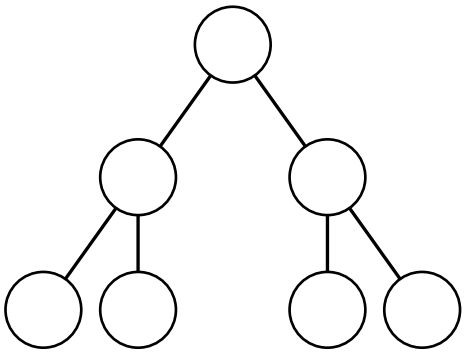


Рис.3.25. Топология дерево.

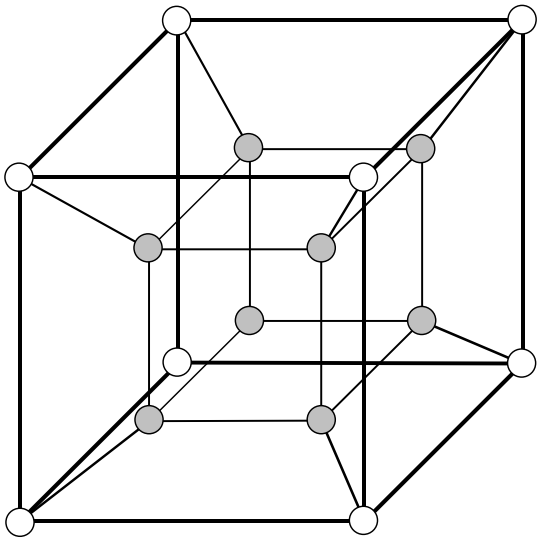


Рис.3.26. Топология гиперкуб.

Более сложный тип коммуникации дается многоступенчатыми соединениями. В этом случае на одном конце соединения находятся процессоры (ВМ), а на другом блоки памяти, устройства ввода/вывода или другие процессоры (ВМ). Между ними располагаются переключатели. При передаче данных от узла к узлу переключатели устанавливаются таким образом, чтобы обеспечить требуемое соединение. Очевидно, для этого требуется некоторое время - "время установки". Примерами многоступенчатых соединений являются соединения "бабочка" и "омега" (рис.3.27). Эти соединения формируются из переключателей, имеющих 2 входа и 2 выхода. Входы и выходы коммутируются. Количество переключателей, необходимое для нормальной работы сети с **n** входами и **n** выходами составляет O(n log2n).

Необходимо выделить еще один класс коммуникационных сетей - *с перестраиваемой топологией*. В такую структуру входит программируемый коммутатор, обеспечивающий произвольные соединения между ВМ. Такие структуры могут "настраиваться" на выполнение конкретного алгоритма и после настройки представляют собой специализированную ВС. Настройка может выполняться статически до начала решения задачи или динамически в процессе решения.

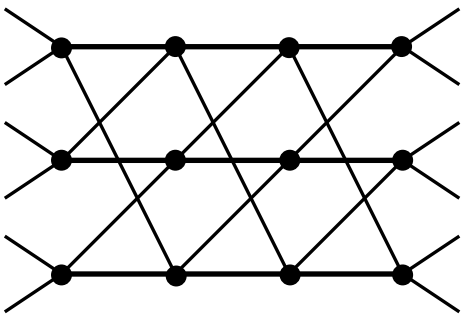


Рис.3.27. Соединение "бабоч-

Масштабируемость характеризует возрастание сложности соединений при добавлении в конфигурацию новых узлов. Если система обладает высокой степенью масштабируемости, ее сложность будет незначительно изменяться при наращивании системы, неизменным будет и диаметр сети.

Также важнейшими атрибутами системы коммуникаций являются стратегии управления, синхронизации и переключения. Что касается управления, то здесь можно выделить две альтернативы: централизованное управление единым узлом (модулем

управления) и распределенное управление. Примером распределенного управления является работа многоступенчатых соединений, где каждый узел принимает решение, как поступить с полученным сообщением - оставить его себе или передать соседу.

Синхронизация тоже может быть глобальной, когда синхронизующая последовательность импульсов передается всем узлам вычислительной системы, но может быть и локальной, когда каждый узел имеет свой собственный генератор. Последний вариант используется в асинхронных системах. Преимущество глобальной синхронизации заключается в более простой аппаратной и программной реализации, а асинхронные системы более гибкие и производительные.

Информационная связь между абонентами может устанавливаться тремя способами: коммутацией каналов, коммутацией сообщений и коммутацией пакетов. **Коммутация каналов** обеспечивает выделение физического канала для прямой передачи данных между узлами. Время передачи зависит от длины передаваемого сообщения, пропускной способности канала и времени распространения сигнала поканалу. **Коммутация сообщений** производится путем передачи сообщения, содержащего заголовок и данные, по маршруту, определяемому узлами сети. При **коммутации пакетов** сообщение разбивается на более мелкие пакеты, пересылаемые по сети. Преодолев очередное соединение, пакет попадает на узел, который и определяет, куда этот пакет должен быть отправлен и должен ли он быть отправлен вообще. В конце концов пакеты должны прибыть на узел-адресат, причем пути их могут быть разными. На последнем узле пакеты собираются и восстанавливается переданное сообщение.

Таким образом, важнейшей характеристикой коммутационной сети является время доставки данных. Оно зависит от структуры сети связи, пропускной способности линий связи, способа организации канала связи и способа передачи данных по каналу. Среди элементов информации, пересылаемых между множеством процессоров (или ВМ), можно выделить: общие переменные, сообщения переменного размера, значения данных, примитивы синхронизации, семафоры состояний и сигналы прерываний (таблица 3.5).

Таблица 3.5. Схемы межпроцессорного взаимодействия.

Модель	**Средства аппаратного обеспечения**	**Механизм связи**
Cray X-MP	Общая память, общие регистры	Семафор и общие переменные
ETA-10	Общая память, связной буфер	Общие переменные и функции синхронизации
HEP-1	Общая память	Теговые переменные
iPSC	Соединительная сеть "гиперкуб"	Передача сообщений (коммутация пакетов)
Cyberplus	Несколько колец связи	Передача сообщений (коммутация пакетов)
Alliant FX/8	Общие кэш / память	Синхронизация через специальную шину
CM 1	Гиперкуб / клеточная структура	Маркер / передача сообщений
Машина Больцмана	Виртуальные соединения	Передача значений
BBN Butterfly	Коммутационная сеть "бабочка"	Общие переменные в распределенной памяти
IBM RP3	Многокаскадная сеть	Общие переменные и передача сообщений
IBM GF-11	ОКМД - сеть	ОКМД - трансляция
FPS-T	Гиперкуб / транспьютер	Канальные команды языка Оккам

Как видно из таблицы 3.5, кроме общей памяти и передачи сообщений для межпроцессорного взаимодействия иногда используются передача маркера и передача значений.

В системах с *передачей маркера* связь между процессорами (или ВМ) осуществляется путем передачи однобитовых маркеров. Каждый ВМ с помощью простых логических операций может обрабатывать несколько маркерных битов. Маркеры в совокупности указывают на объекты, обладающие общим свойством, и идентифицируются в одном широковещательном сообщении. Благодаря этому отсутствует необходимость в синхронизации, как это имеет место в системах с топологией гиперкуб, в которых все элементы функционируют синхронно под внешним управлением (например, Connection Machine 1).

В системах с *пересылкой значений* передается непрерывный поток чисел и выполняются простые арифметические действия над ними. Несколько значений, поступающих в процессор одновременно, объединяются в одно значение и, следовательно, вероятность возникновения конфликтной ситуации равна нулю – синхронизации не требуется. Данная схема была предложена для структур типа нейронных сетей и машин Больцмана. Схемы с передачей маркера и значений нашли применение, главным образом, в системах искусственного интеллекта.

3.4. Параллельная структура общего вида

При значительном увеличении частоты взаимодействия между потоками характер алгоритмической структуры совершенно меняется - она превращается в параллельную структуру общего вида (см. рис.3.1г). Такая структура содержит в себе наибольшие возможности для организации параллельных вычислений и увеличения скорости обработки, но в тоже время, она таит серьезные трудности, связанные с представлением программ в системе, определением последовательности выполнения операций и накладными расходами на синхронизацию. Для параллельных структур общего вида практически не возможно явно задать последовательность выполнения команд в объектном коде, поэтому возникает необходимость с помощью указателей задавать отношение предшествования на множестве операторов, что значительно усложняет представление программ.

Далее из-за того, что в условие инициализации операции входит завершение предшествующих операций, управление порядком исполнения команд становится более сложным. Для осуществления такого управления предложено два механизма:

1. **С помощью потоков признаков.**
2. **С помощью потоков данных.**

Ряд проблем при реализации механизма управления с помощью потоков данных возникает в тех случаях, когда алгоритм содержит либо циклы (т.е. когда какую-либо группу команд необходимо исполнить несколько раз подряд), либо подпрограммы. Существуют две группы способов реализации потокового управления:

- способы, основанные на *статическом* подходе, когда циклы и подпрограммы раскрываются на этапе трансляции, так что в результате каждая команда должна исполняться только один раз,
- способы, основанные на *динамическом* подходе, когда операнды снабжаются метками и группируются в пары таким образом, чтобы для использования разных реализаций тела цикла (или подпрограммы) оказывалось возможным генерировать копию одной и той же команды.

Преимущество статического подхода состоит в простоте управления последовательностью операций, поскольку в этом случае принцип потокового управления реализуется в чистом виде - команды исполняются сразу, как только поступают операнды. В динамическом случае, прежде чем воспроизвести единственную копию команды, необходимо собрать вместе операнды, помеченные одной и той же меткой. Это значительно усложняет реализацию данного подхода. С другой стороны, динамический вариант позво-

ляет получать менее объемные программы и к тому же реализовывать параллелизм, появляющийся в процессе счета (например, при исполнении рекурсивных процедур или циклов, зависящих от данных).

Еще одна проблема, связанная с потоковыми машинами, заключается в описании структуры данных. Будучи представленным в чистом виде, а также в интересах наилучшего параллелизма, граф алгоритма содержит только операции и скалярные величины, не облаченные в какую-либо структурированную форму. Однако такое представление создает определенные трудности с точки зрения модульности программ. Использование структурированных данных требует расширения концепции потоковой обработки и определения операций над такими данными. С точки зрения потоковой обработки наиболее прямолинейный подход состоит в том, чтобы рассматривать конкретную структуру в качестве элементарной единицы, такой же, как, например, скаляр. Недостатки данного подхода - невозможность одновременного манипулирования с такими «элементами», а также необходимость посылать структуру в функциональные модули целиком. Последняя проблема может быть решена, если в функциональный модуль вместо значений передавать указатели структур.

Одна из характерных особенностей большинства разработок в области потоковых вычислений состоит в том, что все они основаны на использовании параллелизма на нижнем уровне дробления операций. Это обстоятельство, с одной стороны, является преимуществом, поскольку способствует использованию всего имеющегося параллелизма, но, с другой стороны, вычисления, базирующиеся на нижнем неструктурированном уровне параллелизма, сопровождаются слишком большими накладными расходами. В связи с этим был предложен иерархический подход, при котором на разных уровнях могут быть использованы различные виды параллелизма (последовательно-групповой и в виде параллельных процессов). Принцип потоковой обработки целесообразно применять только в тех случаях, когда любые иные более структурированные подходы оказываются неэффективными.

Примерами систем потоковой обработки являются: потоковые машины Массачусетского технологического института (статический принцип обработки), потоковая машина с тегированными метками, Манчестерская потоковая машина, потоковая машина TI и потоковая машина с одноразовыми присваиваниями LAU.

Комбинированные подходы. Необходимо отметить, что ни один из рассмотренных выше принципов организации вычислительных систем не является абсолютно адекватным для всех прикладных областей. Поэтому для расширения диапазона применения создаваемых машин используются комбинации разных типов архитектур. При таких комбинациях вступают в действие разные виды параллелизма и разные уровни дробления вычислений. Благодаря тому, что любой вид параллелизма может быть использован на любом уровне дробления, возможно получение большого разнообразия комбинаций. В качестве примера такого подхода можно привести реализованную во многих векторных и матричных системах комбинацию скалярной (с опережающим просмотром) и векторной обработок. Система Cray X-MP включает два центральных процессора, каждый из которых содержит скалярный и векторный процессоры, - это комбинация скалярной, векторной обработки и параллельного исполнения процессов.

4. Классификация вычислительных систем

4.1. Классификация Флинна

По-видимому, самой ранней и наиболее известной является классификация архитектур вычислительных систем, предложенная в 1966 году М.Флинном [1]. 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, SIMD, MISD, MIMD.

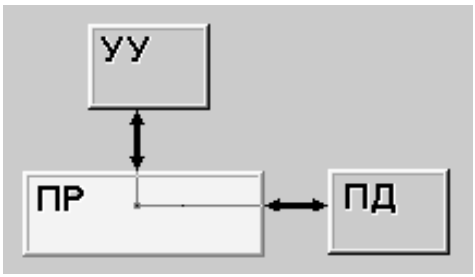


Рис.4.1. SISD – архитектура.

SISD (single instruction stream / single data stream) - одиночный поток команд и одиночный поток данных (рис.4.1.). К этому классу относятся, прежде всего, классические последовательные машины, или иначе, машины фон-неймановского типа. 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 внутри процессора.

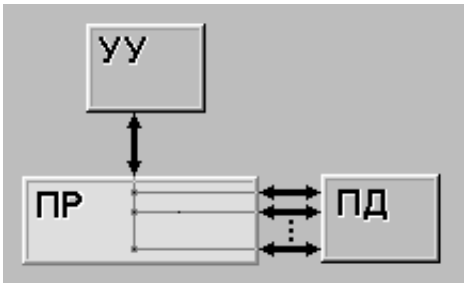


Рис.4.2. SIMD – архитектура.

SIMD (single instruction stream / multiple data stream) - одиночный поток команд и множественный поток данных (рис.4.2.). 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 производится либо процессорной матрицей, как в ILLIAC IV, либо с помощью конвейера, как, например, в машине Cray-1.

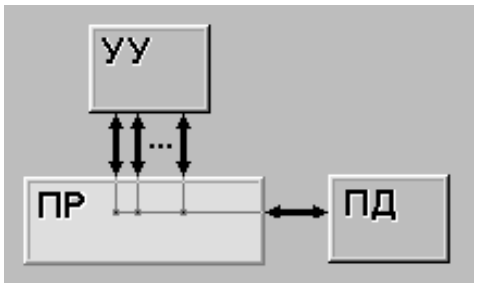


Рис.4.3. MISD – архитектура.

MISD (multiple instruction stream / single data stream) - множественный поток команд и одиночный поток данных (рис.4.3.). 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 относят конвейерные машины к данному классу, однако это не нашло окончательного признания в научном сообществе.

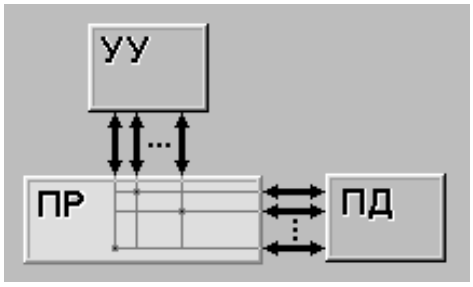


Рис.4.4. MIMD – архитектура.

MIMD (multiple instruction stream / multiple data stream) - множественный поток команд и множественный поток данных (рис.4.4.). 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. К этому классу относятся как многомашинные системы так и некоторые типы много-

процессорных систем.

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, MPP Goodyear Aerospace, 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 потоковые (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 "неопределенного" класса MISD не стоит считать недостатком схемы. Такие классы, по мнению некоторых исследователей в области классификации архитектур, могут стать чрезвычайно полезными для разработки принципиально новых концепций в теории и практике построения вычислительных систем.

4.2. Дополнения Ванга и Бриггса к классификации Флинна

В работе К.Ванга и Ф.Бриггса сделаны некоторые дополнения к классификации Флинна. Оставляя четыре ранее введенных базовых класса (SISD, SIMD, MISD, MIMD), авторы внесли следующие изменения.

Класс **SISD** разбивается на два подкласса:

- архитектуры с единственным функциональным устройством, например, PDP-11 и
- архитектуры, имеющие в своем составе несколько функциональных устройств - CDC 6600, Cray-1, FPS AP-120B, CDC Cyber 205, FACOM VP-200.

В класс **SIMD** также вводится два подкласса:

- архитектуры с пословно-последовательной обработкой информации - ILLIAC IV, PEPE, BSP и
- архитектуры с разрядно-последовательной обработкой - STARAN, ICL DAP.

В классе **MIMD** авторы различают:

- вычислительные системы со слабой связью между процессорами, к которым они относят все системы с распределенной памятью (например, Cosmic Cube) и
- вычислительные системы с сильной связью (системы с общей памятью), куда попадают такие компьютеры, как C.mmp, BBN Butterfly, Cray Y-MP, Denelcor HEP.

4.3. Классификация Базу

По мнению А.Базу (A.Basu), любую параллельную вычислительную систему можно однозначно описать последовательностью решений, принятых на этапе ее проектирования, а сам процесс проектирования представить в виде дерева. В самом деле, корень дерева - это вычислительная система, а последующие ярусы дерева, фиксируя уровень параллелизма, метод реализации алгоритма, параллелизм инструкций и способ управления, последовательно дополняют друг друга, формируя описание системы (рис.4.5).

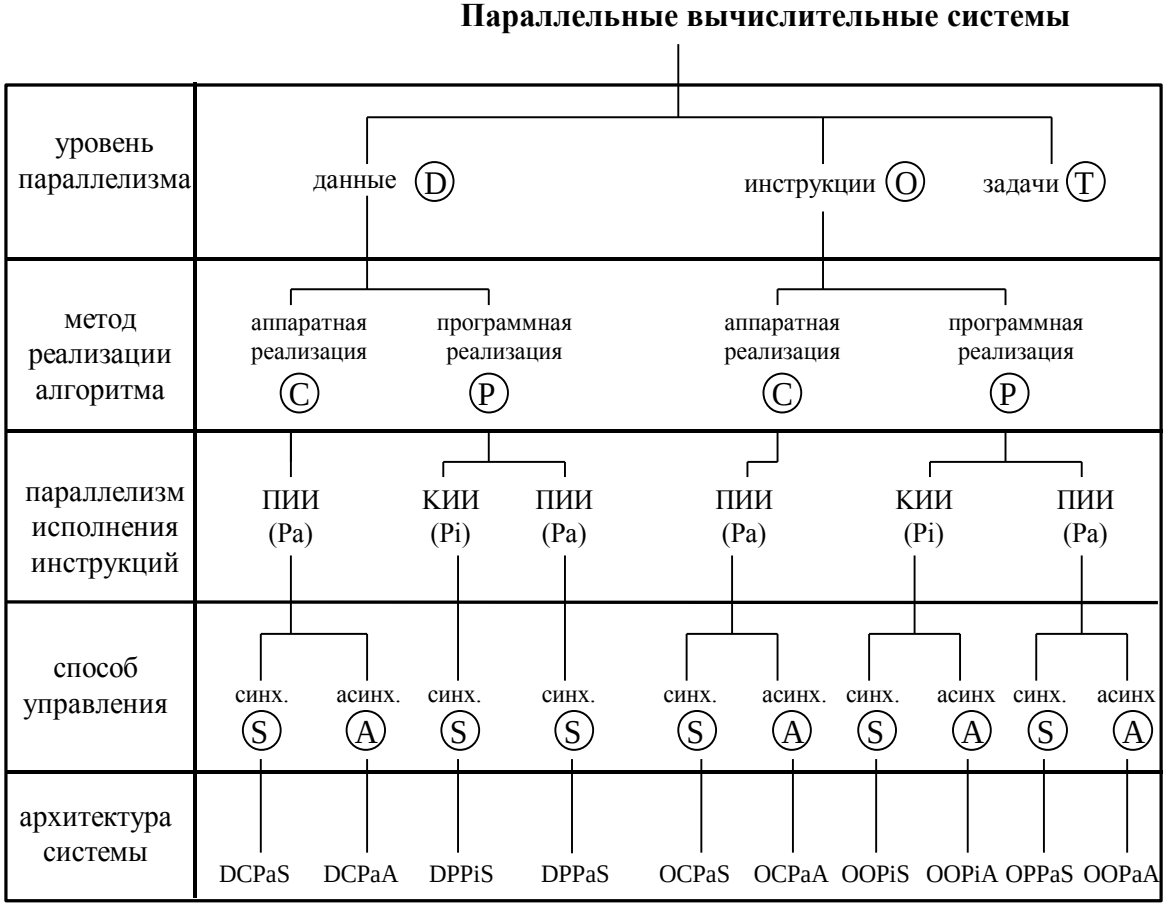


Рис.4.5. Классификация Базу.

На первом этапе мы определяем, какой уровень параллелизма используется в вычислительной системе. Одна и та же операция может одновременно выполняться над целым набором данных, определяя параллелизм на уровне данных (обозначается буквой *D* на рисунке). Способность выполнять более одной операции одновременно говорит о параллелизме на уровне команд (буква *O*). Если же компьютер спроектирован так, что целые последовательности команд могут быть выполнены одновременно, то будем говорить о параллелизме на уровне задач (буква *T*).

Второй уровень в классификационном дереве фиксирует метод реализации алгоритма. С появлением сверхбольших интегральных схем (СБИС) стало возможным реализовывать аппаратно не только простые арифметические операции, но и алгоритмы целиком.

Например, быстрое преобразование Фурье, произведение матриц и LU-разложение относятся к классу тех алгоритмов, которые могут быть эффективно реализованы в СБИС'ах. Данный уровень классификации разделяет системы с аппаратной реализацией алгоритмов (буква *C*) и системы, использующие традиционный способ программной реализации (буква *P*).

Третий уровень конкретизирует тип параллелизма, используемого для обработки инструкций машины: конвейеризация инструкций (*Pi*) или их независимое (параллельное) выполнение (*Pa*). В большей степени этот выбор относится к компьютерам с программной реализацией алгоритмов, так как аппаратная реализация всегда предполагает параллельное исполнение команд. Отметим, что в случае конвейерного исполнения имеется в виду лишь конвейеризация самих команд, разбивающая весь цикл обработки на выборку команды, дешифрацию, вычисление адресов и т.д., - возможная конвейеризация вычислений на данном уровне не принимается во внимание.

Последний уровень данной классификации определяет способ управления, принятый в вычислительной системе: синхронный (*S*) или асинхронный (*A*). Если выполнение команд происходит в строгом порядке, определяемом только сигналами таймера и счетчиком команд, то будем говорить о синхронном способе управления. Если же для инициации команды определяющими являются такие факторы, как, например, готовность данных, то попадаем в класс машин с асинхронным управлением. Наиболее характерными представителями систем с асинхронным управлением являются data-driven и demand-driven компьютеры

Описав основные принципы классификации, посмотрим, куда попадают различные типы параллельных вычислительных систем. Изучение систолических массивов, имеющих, как правило, одномерную или двумерную структуру, показывает, что обозначения *DCPaS* и *DCPaA* могут быть использованы для их описания в зависимости от того, как происходит обмен данными: синхронно или асинхронно. Систолические деревья, введенные Кунгом для вычисления арифметических выражений могут быть описаны как *OCPaS* либо *OCPaA* по аналогичным соображениям. Конвейерные компьютеры, такие, как IBM 360/91, Amdahl 470/6 и многие современные *RISC* процессоры, разбивающие исполнение всех инструкций на несколько этапов, в данной классификации имеют обозначение *OPPiS*. Более естественное применение конвейеризации происходит в векторных машинах, в которых одна команда применяется к вектору независимых данных, и за счет непрерывного использования арифметического конвейера достигается значительное ускорение. К таким компьютерам подходит обозначение *DPPiS*. Матричные процессоры, в которых целое множество арифметических устройств работает одновременно в строго синхронном режиме, принадлежат к группе *DPPaS*. Если вычислительная система подобно CDC 6600 имеет процессор с отдельными функциональными устройствами, управляемыми централизованно, то ее описание выглядит так: *OPPaS*. Data-flow компьютеры, в зависимости от особенностей реализации, могут быть описаны либо как *OPPiA*, либо *OPPaA*.

Системы с несколькими процессорами, использующими параллелизм на уровне задач, не всегда можно корректно описать в рамках предложенного формализма. Если процессоры дополнительно не используют параллелизм на уровне операций или данных, то для описания можно использовать лишь букву *T*. В противном случае, Базу предлагает использовать знак "*" между символами, обозначающими уровни параллелизма, одновременно присутствующие в системе. Например, комбинация *T*D* означает, что некоторая система может одновременно исполнять несколько задач, причем каждая из них может использовать векторные команды.

Очень часто в реальных системах присутствуют особенности, характерные для компьютеров из разных групп данной классификации. В этом случае для корректного описания автор использует знак "+". Например, практически все векторные компьютеры имеют скалярную и векторную части, что можно описать как *OPPiS+DPPiS* (пример - это TI ASC

и CDC STAR-100). Если в системе есть возможность одновременного выполнения более одной векторной команды (как в Cray-1) то для описания векторной части можно использовать запись *O*DPPiS*, а полное описание данного компьютера выглядит так: *O*DPPiS+OPPiS*. Действуя по такому же принципу, можно найти описание и для систем Cray X-MP и Cray Y-MP. В самом деле, данные системы объединяют несколько процессоров, имеющих схожую с Cray-1 структуру, и потому их описание имеет вид: *T*(O*DPPiS+OPPiS).*

4.4. Классификация Кришнамарфи

Е.Кришнамарфи для классификации параллельных вычислительных систем предлагает использовать четыре характеристики, очень похожие на характеристики классификации А.Базу:

1. степень гранулярности;
2. способ реализации параллелизма;
3. топология и природа связи процессоров;
4. способ управления процессорами.

Принцип построения классификации очень прост. Для каждой степени гранулярности будем рассматривать все возможные способы реализации параллелизма. Для каждого полученного таким образом варианта рассмотрим все комбинации топологии связи и способов управления процессорами. В результате получим дерево (рис.4.6), в котором каждый ярус соответствует своей характеристике, каждый лист представляет отдельную группу компьютеров в данной классификации, а путь от вершины дерева однозначно определяет значения указанных выше характеристик. Разберем характеристики подробнее.

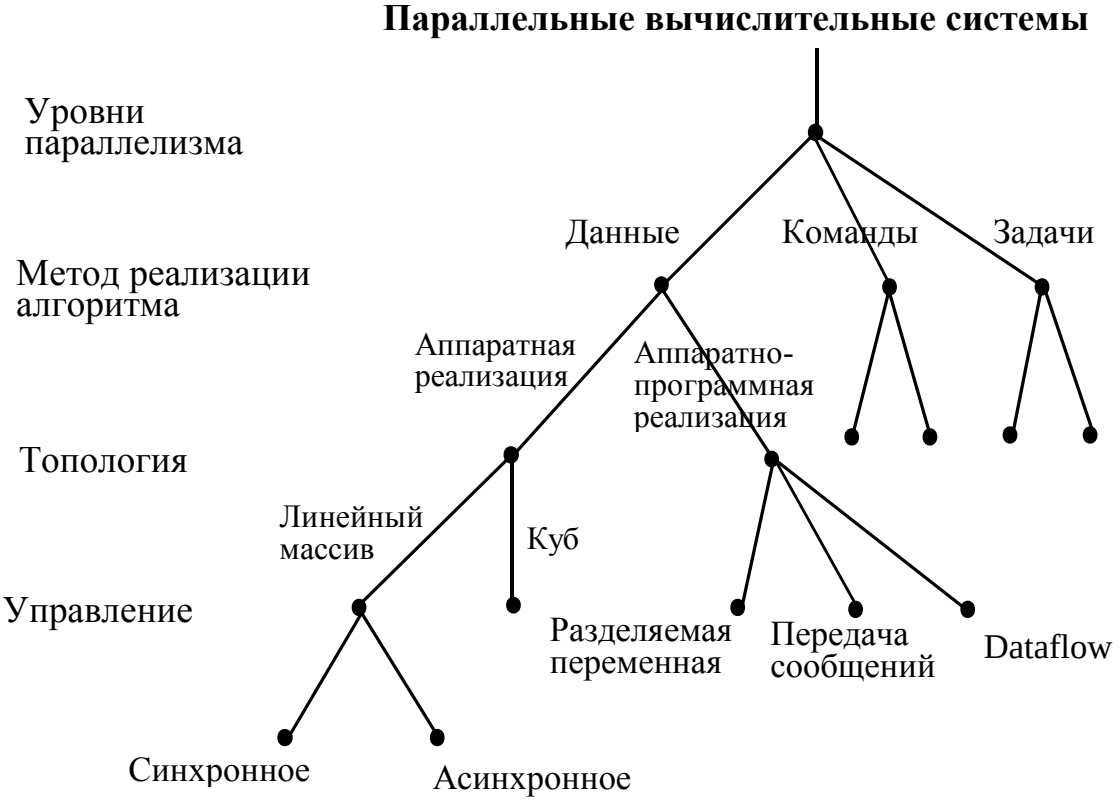


Рис.4.6. Классификация Кришнамарфи.

Первые два уровня практически один к одному повторяют А.Базу, поэтому останавливаться подробно на них мы не будем. Третий уровень классификации, топология и природа связи процессоров, тесно связан со вторым. Если был выбран аппаратный способ ре-

ализации параллелизма, то надо рассмотреть топологию связи процессоров (матрица, линейный массив, тор, дерево, звезда и т.п.) и степень связности процессоров между собой (сильная, слабая или средняя), которая определяется относительной долей накладных расходов при организации взаимодействия процессоров. В случае комбинированной реализации параллелизма, помимо топологии и степени связности, надо дополнительно учесть механизм взаимодействия процессоров: передача сообщений, разделяемые переменные или принцип dataflow (по готовности операндов).

Наконец, последний, четвертый уровень - способ управления процессорами, определяет общий принцип функционирования всей совокупности процессоров вычислительной системы: синхронный, dataflow или асинхронный.

На основе выделенных четырех характеристик нетрудно определить место наиболее известных классов архитектур в данной систематике.

Векторно-конвейерные компьютеры: гранулярность - на уровне данных; реализация параллелизма - аппаратная; связь процессоров - простая топология со средней связностью; способ управления - синхронный.

Классические мультипроцессоры: гранулярность - на уровне задач; реализация параллелизма - комбинированная; связь процессоров - простая топология со слабой связностью и использованием разделяемых переменных; способ управления - асинхронный.

Матрицы процессоров: гранулярность - на уровне данных; реализация параллелизма - аппаратная; связь процессоров - двумерные массивы с сильной связностью; способ управления - синхронный.

Систолические массивы: гранулярность - на уровне данных; реализация параллелизма - аппаратная; связь процессоров - сложная топология с сильной связностью; способ управления - синхронный.

Архитектура типа wavefront (волновая): гранулярность - на уровне данных; реализация параллелизма - аппаратная; связь процессоров - двумерная топология с сильной связностью; способ управления - dataflow.

Архитектура типа dataflow (потоковая): гранулярность - на уровне команд; реализация параллелизма - комбинированная; связь процессоров - простая топология с сильной либо средней связностью и использованием принципа dataflow; способ управления - асинхронно-dataflow.

Несмотря на то, что классификация Е. Кришнамарфи построена лишь на четырех признаках, она позволяет выделить и описать такие "нетрадиционные" параллельные системы, как систолические массивы, машины типа dataflow и wavefront. Однако эта же простота является и основной причиной ее недостатков: некоторые архитектуры нельзя однозначно отнести к тому или иному классу, например, компьютеры с архитектурой гиперкуба и ассоциативные процессоры. Для более точного описания таких машин потребуется ввести еще целый ряд характеристик, таких, как размещение задач по процессорам, способ маршрутизации сообщений, возможность реконфигурации, аппаратная поддержка языков программирования и другие. Вместе с тем ясно, что эти признаки формализовать гораздо труднее, поэтому есть опасность вместо ясности внести в описание лишь дополнительные трудности.

4.5. Классификация Фенга

В 1972 году Т.Фенг предложил классифицировать вычислительные системы на основе двух простых характеристик. Первая - число бит **n** в машинном слове, обрабатываемых параллельно при выполнении машинных инструкций. Практически во всех современных компьютерах это число совпадает с длиной машинного слова. Вторая характеристика равна числу слов **m**, обрабатываемых одновременно данной вычислительной системой. Немного изменив терминологию, функционирование любого компьютера можно представить как параллельную обработку **n** битовых слоев, на каждом из которых незави-

симо преобразуются **m** бит. Опираясь на такую интерпретацию, вторую характеристику обычно называют шириной битового слоя.

Если рассмотреть предельные верхние значения данных характеристик, то каждую вычислительную систему C можно описать парой чисел (n,m) и представить точкой на плоскости в системе координат длина слова - ширина битового слоя. Площадь прямоугольника со сторонами n и m определяет интегральную характеристику потенциала параллельности P архитектуры и носит название *максимальной степени параллелизма* вычислительной системы: P(C)=mn. По существу, данное значение есть ничто иное, как пиковая производительность, выраженная в других единицах. В период появления данной классификации, а это начало 70-х годов, еще казалось возможным перенести понятие пиковой производительности как универсального средства сравнения и описания потенциальных возможностей компьютеров с традиционных последовательных машин на параллельные. Понимание того факта, что пиковая производительность сама по себе не столь важна, пришло позднее, и данный подход отражает, естественно, степень осмысления специфики параллельных вычислений того времени.

Рассмотрим компьютер Advanced Scientific Computer фирмы Texas Instruments (**TI ASC**). В основном режиме он работает с 64-х разрядным словом, причем все разряды обрабатываются параллельно. Арифметико-логическое устройство имеет четыре одновременно работающих конвейера, содержащих по восемь ступеней. Такая организация дает 4x8=32 бита в каждом битовом слое, и значит компьютер TI ASC может быть представлен в виде (64,32).

На основе введенных понятий все вычислительные системы в зависимости от способа обработки информации, заложенного в их архитектуру, можно разделить на четыре класса.

Разрядно-последовательные пословно-последовательные (n=m=1). В каждый момент времени такие компьютеры обрабатывают только один двоичный разряд. Представителем данного класса служит давняя система MINIMA с естественным описанием (1,1).

Разрядно-параллельные пословно-последовательные (n > 1 , m = 1). Большинство классических последовательных компьютеров, так же как и многие вычислительные системы, эксплуатируемые до сих пор, принадлежит к данному классу: IBM 701 с описанием (36,1), PDP-11 (16,1), IBM 360/50 и VAX 11/780 - обе с описанием (32,1).

Разрядно-последовательные пословно-параллельные (n = 1 , m > 1). Как правило вычислительные системы данного класса состоят из большого числа одноразрядных процессорных элементов, каждый из которых может независимо от остальных обрабатывать свои данные. Типичными примерами служат STARAN (1, 256) и MPP (1,16384) фирмы Goodyear Aerospace, прототип известной системы ILLIAC IV компьютер SOLOMON (1, 1024) и ICL DAP (1, 4096).

Разрядно-параллельные пословно-параллельные (n > 1, m > 1). Большая часть существующих параллельных вычислительных систем, обрабатывая одновременно mn двоичных разрядов, принадлежит именно к этому классу: ILLIAC IV (64, 64), TI ASC (64, 32), C.mmp (16, 16), CDC 6600 (60, 10), BBN Butterfly GP1000 (32, 256).

Недостатки предложенной классификации достаточно очевидны и связаны со способом вычисления ширины битового слоя m. По существу Фенг не делает никакого различия между процессорными матрицами, векторно-конвейерными и многопроцессорными системами. Не делается акцент на том, за счет чего компьютер может одновременно обрабатывать более одного слова: множественности функциональных устройств, их конвейерности или же какого-то числа независимых процессоров. Если в системе N независимых процессоров имеют каждый по F конвейерных функциональных устройств с длиной конвейера L, то для вычисления ширины битового слоя надо просто найти произведение данных характеристик.

Конечно же, опираясь на данную классификацию, достаточно трудно (а иногда и невозможно) осознать специфику той или иной вычислительной системы. Однако достоинством является *введение единой числовой метрики* для всех типов компьютеров, которая вместе с описанием потенциала вычислительных возможностей конкретной архитектуры позволяет сравнить любые два компьютера между собой.

4.6. Классификация Хендлера

В основу классификации В.Хендлер закладывает явное описание возможностей параллельной и конвейерной обработки информации вычислительной системой. При этом он намеренно не рассматривает различные способы связи между процессорами и блоками памяти и считает, что коммуникационная сеть может быть нужным образом сконфигурирована и будет способна выдержать предполагаемую нагрузку.

Предложенная классификация базируется на различии между тремя уровнями обработки данных в процессе выполнения программ:

- уровень выполнения программы - опираясь на счетчик команд и некоторые другие регистры, устройство управления (УУ) производит выборку и дешифрацию команд программы;
- уровень выполнения команд - арифметико-логическое устройство компьютера (АЛУ) исполняет команду, выданную ему устройством управления;

уровень битовой обработки - все элементарные логические схемы процессора (ЭЛС) разбиваются на группы, необходимые для выполнения операций над одним двоичным разрядом.

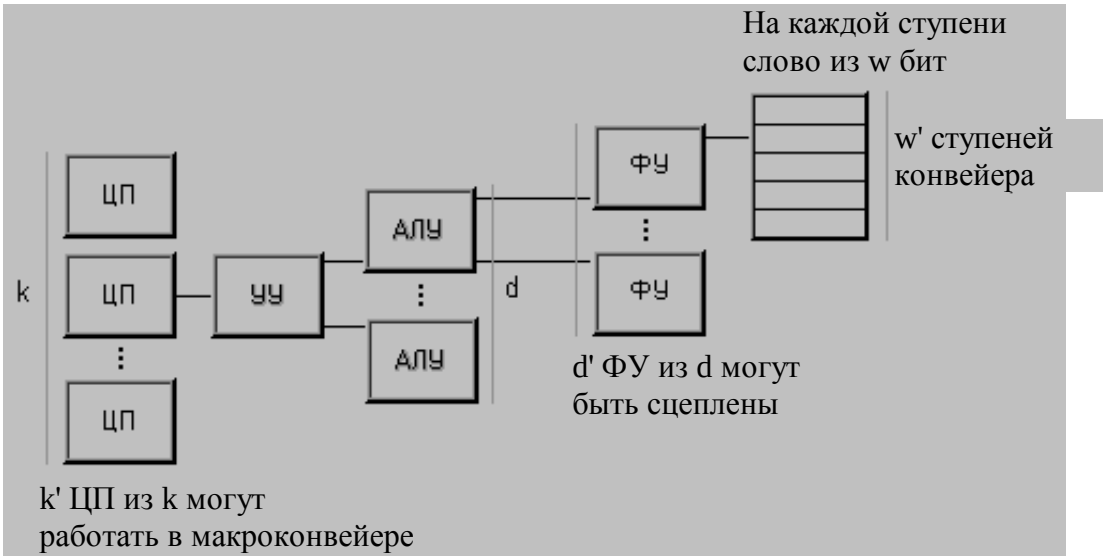


Рис.4.7. Классификация Хендлера

Таким образом, подобная схема выделения уровней предполагает (рис.4.7.), что вычислительная система включает какое-то число процессоров каждый со своим устройством управления. Каждое устройство управления связано с несколькими арифметико-логическими устройствами, исполняющими одну и ту же операцию в каждый конкретный момент времени. Наконец, каждое АЛУ объединяет несколько элементарных логических схем, ассоциированных с обработкой одного двоичного разряда (число ЭЛС есть ничто иное, как длина машинного слова). Если на какое-то время не рассматривать возможность конвейеризации, то число устройств управления k , число арифметико-логических устройств d в каждом устройстве управления и число элементарных логических схем w в

каждом АЛУ составят тройку для описания данной вычислительной системы С:

t(C) = (k, d, w)

В таких обозначениях описания некоторых хорошо известных вычислительных систем будут выглядеть следующим образом:

t (MINIMA) = (1,1,1);
t (IBM 701) = (1,1,36);
t (SOLOMON) = (1,1024,1);
t (ILLIAC IV) = (1,64,64);
t (STARAN) = (1,8192,1) - в полной конфигурации;
t (C.mmp) = (16,1,16) - основной режим работы;
t (PRIME) = (5,1,16);
t (BBN Butterfly GP1000) = (256,~1,~32).

Несмотря на то, что перечисленным системам присущ параллелизм разного рода, он без особого труда может быть отнесен к одному из трех выделенных уровней.

Теперь можно расширить возможности описания, допустив возможность конвейерной обработки на каждом из уровней. В самом деле, конвейерность на самом нижнем уровне (т.е. на уровне ЭЛС) это конвейерность функциональных устройств. Если функциональное устройство обрабатывает w-разрядные слова на каждой из w' ступеней конвейера, то для характеристики параллелизма данного уровня естественно рассмотреть произведение w×w'. Знак умножения × будем использовать на каждом уровне чтобы отделить число, представляющее степень параллелизма, от числа ступеней в конвейере. Компьютер TI ASC имеет четыре конвейерных устройства по восемь ступеней в каждом для обработки 64-х разрядных слов, следовательно, он может быть описан так:

t (TI ASC) = (1,4,64×8)

Следующий уровень конвейерной обработки - это конвейеризация на уровне команд. Предполагается, что в вычислительной системе есть несколько функциональных устройств, которые могут работать одновременно в рамках одного потока команд (в настоящее время используется специальный термин для обозначения данной возможности - *сцепление функциональных устройств*). Классическим примером этому могут служить компьютеры фирмы Cray Research. А исторически первой, по всей вероятности, является машина CDC 6600, содержащая десять независимых последовательных функциональных устройств, способных подавать результат своей работы на вход другим функциональным устройствам, образуя единый поток команд:

t (CDC 6600) = (1,1×10,~64) (описан только центральный процессор без учета управляющих и периферийных подсистем).

Наконец, нам осталось рассмотреть конвейеризацию на самом верхнем уровне, известную как макро-конвейер. Поток данных, проходя через один процессор, поступает на вход другому, возможно через некоторую буферную память. Если независимо работают n процессоров, то в идеальной ситуации при отсутствии конфликтов и полной сбалансированности получаем ускорение в n раз по сравнению с использованием только одного процессора. Так компьютер PEPE, имея фактически три независимых системы из 288-ми устройств, описывается следующим образом:

t(PEPE) = (1×3,288,32)

После расширения трехуровневой модели параллелизма средствами описания потенциальных возможностей конвейеризации каждая тройка

t (PEPE) = (k×k',d×d',w×w')

интерпретируется так:

k - число процессоров (каждый со своим УУ), работающих параллельно
k' - глубина макроконвейера из отдельных процессоров
d - число АЛУ в каждом процессоре, работающих параллельно
d' - число функциональных устройств АЛУ в цепочке

w - число разрядов в слове, обрабатываемых в АЛУ параллельно

w' - число ступеней в конвейере функциональных устройств АЛУ

Очевидна связь между классификацией Фенга и классификацией Хендлера: для получения максимальной степени параллелизма в терминах Фенга надо найти произведение всех шести величин в описании Хендлера. Здесь же заметим, что заложив в основу своей схемы явное указание на присутствующий параллелизм и возможную конвейеризацию, В.Хендлер сразу снимает массу вопросов, характерных для предшествующих схем Флинна, Шора и Фенга, по крайней мере, в плане описания векторно-конвейерных машин.

В дополнение к изложенному способу описания архитектур Хендлер предлагает использовать три операции, которые будучи примененными к тройкам, позволят описать:

- сложные структуры с подсистемами ввода-вывода, хост-компьютером или какими-то другими особенностями;

возможные режимы функционирования вычислительных систем, поддерживаемые для оптимального соответствия структуре программ.

Первая операция (×) в каком-то смысле отражает конвейерный принцип обработки и предполагает последовательное прохождение данных сначала через первый ее аргумент-подсистему, а затем через второй. Описание упомянутого выше компьютера CDC 6600 можно уточнить следующим образом:

t(CDC 6600) = (10,1,12) × (1,1×10,64),

где первый аргумент отражает существование десяти 12-ти разрядных периферийных процессоров и тот факт, что любая программа должна сначала быть обработана одним из них и лишь после этого передана центральному процессору для исполнения. Аналогично можно получить описание машины PEPE, принимая во внимание, что в качестве хост-компьютера она использует CDC 7600:

t(PEPE) = t(CDC 7600) × (1×3, 288, 32) = = (15, 1, 12) × (1, 1×9, 60) × (1×3, 288, 32)

Поток данных последовательно проходит через три подсистемы, что мы и отразили, соединив их знаком '×'.

Заметим, что все подсистемы последнего примера достаточно сложны и, вообще говоря, исходя только из данного описания могут представляться по-разному. Чтобы внести большую ясность, аналогично операции конвейерного исполнения, Хендлер вводит *операцию параллельного исполнения* (+), фиксирующую возможность независимого использования процессоров разными задачами:

t(n,d,w) = [{(1,d,w) +... + (1,d,w)] *{n раз}*

В случае CDC 7600 уточненная запись вида:

(15, 1, 12) × (1, 1×9, 60) = [(1, 1, 12) +... +(1, 1, 12)]} *{15 раз}* × (1, 1×9, 60)

говорит о том, что каждая задача может захватить свой периферийный процессор, а затем одна за одной они будут поступать в центральный процессор. И наконец третья операция - *операция альтернативы* (V), показывает возможные альтернативные режимы функционирования вычислительной системы. Чем больше для системы таких режимов, тем более гибкой архитектурой, по мнению Хендлера, она обладает. Например, компьютер C.mmp может быть запрограммирован для использования в трех принципиально разных режимах:

t(C.mmp) = (16, 1, 16) V (1×16, 1,1 6) V (1, 16, 16).

4.7. Классификация Шора

Классификация Дж.Шора, появившаяся в начале 70-х годов, интересна тем, что представляет собой попытку выделения типичных способов компоновки вычислительных систем на основе фиксированного числа базисных блоков: устройства управления, арифметико-логического устройства, памяти команд и памяти данных. Дополнительно предполагается, что выборка из памяти данных может осуществляться словами, то есть выбираются все разряды одного слова, и/или битовым слоем - по одному разряду из одной и той

же позиции каждого слова (иногда эти два способа называют горизонтальной и вертикальной выборками соответственно). Конечно же, при анализе данной классификации надо делать скидку на время ее появления, так как предусмотреть невероятное разнообразие параллельных систем настоящего времени было в принципе невозможно. Итак, согласно классификации Шора все компьютеры разбиваются на шесть классов, которые он так и называет: машина типа I, II и т.д.

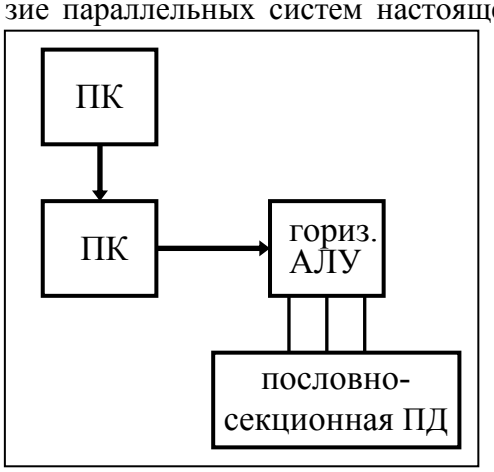


Рис.4.8. Машина I

Машина I - это вычислительная система, которая содержит устройство управления, арифметико-логическое устройство, память команд и память данных с пословной выборкой (рис.4.8). Считывание данных осуществляется выборкой всех разрядов некоторого слова для их параллельной обработки в арифметико-логическом устройстве. Состав АЛУ специально не оговаривается, что допускает наличие нескольких функциональных устройств, быть может конвейерного типа. По этим соображениям в данный класс попадают как классические последовательные машины (IBM 701, PDP-11, VAX 11/780), так и конвейерные скалярные (CDC 7600) и векторно-конвейерные (Cray-1).

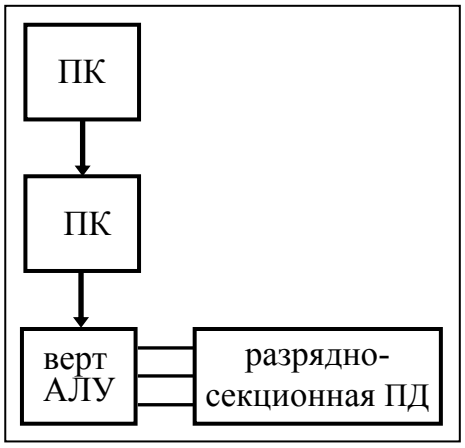


Рис.4.9. Машина II

Если в машине I осуществлять выборку не по словам, а выборкой содержимого одного разряда из всех слов, то получим **машину II** (рис.4.9)**.** Слова в памяти данных по прежнему располагаются горизонтально, но доступ к ним осуществляется иначе. Если в машине I происходит последовательная обработка слов при параллельной обработке разрядов, то в машине II - последовательная обработка битовых слоев при параллельной обработке множества слов.

Структура машины II лежит в основе ассоциативных компьютеров (например, центральный процессор машины STARAN), причем фактически такие компьютеры имеют не одно арифметико-логическое устройство, а множество сравнительно простых устройств поразрядной обработки. Другим примером служит матричная система ICL DAP, которая может одновременно обрабатывать по одному разряду из 4096 слов.

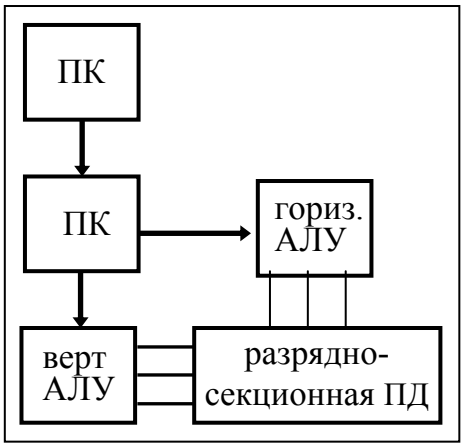


Рис.4.10. Машина III

Если объединить принципы построения машин I и II, то получим **машину III** (рис.4.10). Эта машина имеет два арифметико-логических устройства - горизонтальное и вертикальное, и модифицированную память данных, которая обеспечивает доступ как к словам, так и к битовым слоям. Впервые идею построения таких систем в 1960 году выдвинул У.Шуман , называвший их ортогональными (если память представлять как матрицу слов, то доступ к данным осуществляется в направлении, "ортогональ-ном" традиционному - не по словам (строкам), а по битовым слоям (столбцам)). В принципе, как машину STARAN, так и ICL DAP можно запрограммировать на выполнение функций машины III, но поскольку они не имеют отдельных АЛУ для обработки слов

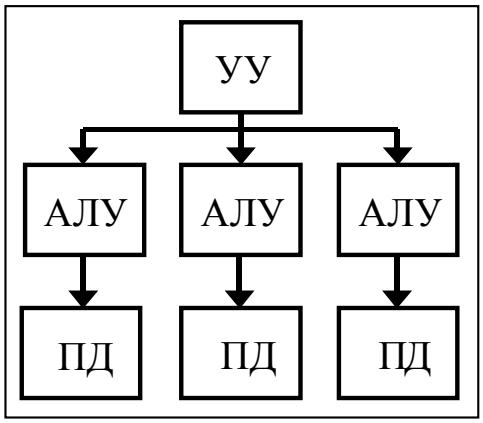


Рис.4.11. Машина **IV.**

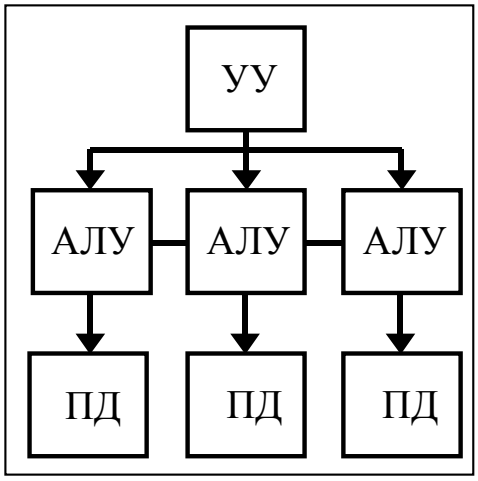


Рис.4.12. Машина V.

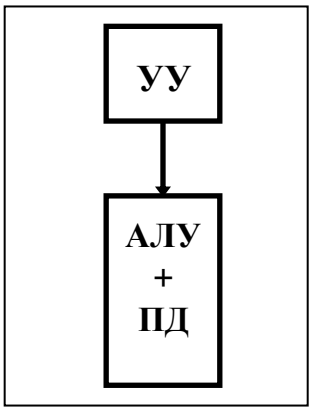


Рис.4.13. Машина VI.

и битовых слоев, отнести их к данному классу нельзя. Полноправными представителями машин класса III являются вычислительные системы семейства OMEN-60 фирмы Sanders Associates, построенные в прямом соответствии с концепцией ортогональной машины.

Если в машине I увеличить число пар арифметико-логическое устройство <==> память данных (иногда эту пару называют *процессорным элементом*) то получим **машину IV** (рис.4.11). Единственное устройство управления выдает команду за командой сразу всем процессорным элементам. С одной стороны, отсутствие соединений между процессорными элементами делает дальнейшее наращивание их числа относительно простым, но с другой, сильно ограничивает применимость машин этого класса. Такую структуру имеет вычислительная система PEPE, объединяющая 288 процессорных элементов.

Если ввести непосредственные линейные связи между соседними процессорными элементами машины IV, например в виде матричной конфигурации, то получим схему **машины V**. Любой процессорный элемент теперь может обращаться к данным как в своей памяти, так и в памяти непосредственных соседей. Подобная структура характерна, например, для классического матричного компьютера ILLIAC IV.

Заметим, что все машины с I-ой по V-ю придерживаются концепции разделения памяти данных и арифметико-логических устройств, предполагая наличие шины данных или какого-либо коммутирующего элемента между ними. **Машина VI** (рис.4.13.)**,** названная *матрицей с функциональной памятью* (или памятью с встроенной логикой), представляет собой другой подход, предусматривающий распределение логики процессора по всему запоминающему устройству**.** Примерами могут служить как простые ассоциативные запоминающие устройства, так и сложные ассоциативные процессоры.

4.8. Классификация Джонсона

Е.Джонсон предложил проводить классификацию MIMD архитектур на основе структуры памяти и реализации механизма взаимодействия и синхронизации между процессорами.

По структуре оперативной памяти существующие вычислительные системы делятся на две большие группы: либо это системы с общей памятью, прямо адресуемой всеми процессорами, либо это системы с распределенной памятью, каждая часть которой доступна только одному процессору. Одновременно с этим, и для межпроцессорного взаимодействия существуют две альтернативы: через разделяемые переменные или с помощью механизма передачи сообщений. Исходя из таких предположе-

ний, можно получить четыре класса MIMD архитектур, уточняющих систематику Флинна:

1. общая память - разделяемые переменные (GMSV);
2. распределенная память - разделяемые переменные (DMSV);
3. распределенная память - передача сообщений (DMMP);
4. общая память - передача сообщений (GMMP).

Опираясь на такое деление, Джонсон вводит названия для некоторых классов. Так вычислительные системы, использующие общую разделяемую память для межпроцессорного взаимодействия и синхронизации, он называет системами с разделяемой памятью, например, Cray Y-MP (по его классификации это класс 1). Системы, в которых память распределена по процессорам, а для взаимодействия и синхронизации используется механизм передачи сообщений он называет архитектурами с передачей сообщений, например NCube, (класс 3). Системы с распределенной памятью и синхронизацией через разделяемые переменные, как в BBN Butterfly, называются гибридными архитектурами (класс 2).

В качестве уточнения классификации автор отмечает возможность учитывать вид связи между процессорами: общая шина, переключатели, разнообразные сети и т.п.

4.9. Классификация Шнайдера

В 1988 году Л.Шнайдер (L.Snyder) предложил новый подход к описанию архитектур параллельных вычислительных систем, попадающих в класс SIMD систематики Флинна. Основная идея заключается в выделении этапов выборки и непосредственно исполнения в потоках команд и данных. Именно разделение потоков на адреса и их содержимое позволяет описать такие ранее "неудобные" для классификации архитектуры, как компьютеры с длинным командным словом, систолические массивы и целый ряд других.

Введем необходимые для дальнейшего изложения понятия и обозначения. Назовем *потоком ссылок* (reference stream) *S* некоторой вычислительной системы конечное множество бесконечных последовательностей пар:

$$S = \{ (a_1 < t_1 >) (a_2 < t_2 >)...,$$
$$(b_1 < u_1 >) (b_2 < u_2 >)...,$$
$$(c_1 < v_1 >)(c_2 < v_2 >)...\},$$

где первый компонент каждой пары - это неотрицательное целое число, называемое *адресом*, второй компонент - это набор из n неотрицательных целых чисел, называемых *значениями*, причем n одинаково для всех наборов всех последовательностей. Например, пара $(b_2 < u_2 >)$ определяет адрес b_2 и значение $< u_2 >$. Если значения рассматривать как команды, то из потока ссылок получим *поток команд I*; если же значения интерпретировать как данные, то соответствующий поток - это *поток данных D.*

Интерпретация введенных понятий очень проста. Элементы каждой последовательности это адрес и его содержимое, выбираемое из (или записываемое в) память. Последовательность пар адрес-значение можно рассматривать как историю выполнения команд либо перемещения данных между процессором и памятью компьютера во время выполнения программы. Число инструкций, которое данный компьютер может выполнять одновременно, определяет число последовательностей в потоке команд. Аналогично, число различных данных, которое компьютер может обработать одновременно, определяет число последовательностей в потоке данных.

Пусть S произвольный поток ссылок. *Последовательность адресов* потока S, обозначаемая Sa, - это последовательность, чей i-й элемент - набор, сформированный из адресов i-х элементов каждой последовательности из S:

$$Sa = < a_1 b_1...c_1 > , < a_2 b_2...c_2 > ,...$$

Последовательность значений потока S, обозначаемая S_v, - это последовательность, чей i-й элемент - набор, образованный слиянием наборов значений i-х элементов каждой

последовательности из *S*:

$$Sv = < t_1\ u_1...v_1 > , < t_2\ u_2...v_2 > ,...$$

Если Sx - последовательность элементов, где каждый элемент - набор из n чисел, то для обозначения "ширины" последовательности будем пользоваться обозначением:

$$w(Sx) = n.$$

Из определений Sa, Sv и w сразу следует утверждение: если S - это поток ссылок со значениями из n чисел, то

$$w(Sa) = |S| \quad \text{и} \quad w(Sv) = n\,|S|,$$ где |S| обозначает мощность множества *S*.

Каждую пару *(I, D)* с потоком команд *I* и потоком данных *D* будем называть *вычислительным шаблоном*, а все компьютеры будем разбивать на классы в зависимости от того, какой шаблон они могут исполнить. В самом деле, компьютер может исполнить шаблон *(I, D)*, если он в состоянии:

выдать w(Ia) адресов команд для одновременной выборки из памяти;

- декодировать и проинтерпретировать одновременно w(Iv) команд;

выдать одновременно w(Da) адресов операндов и

- выполнить одновременно w(Dv) операций над различными данными.

Если все эти условия выполнены, то компьютер может быть описан следующим образом:

$$I_{w(Ia)w(Iv)}D_{w(Da)w(Dv)}$$

Рассмотрим классическую последовательную машину. Согласно классификации Флинна, она попадает в класс SISD, следовательно | I | = |D| = 1. Используя утверждение 1, получаем, что w(Ia) = w(Da) = 1. Из-за того, что в подобного рода компьютерах команды декодируются последовательно, следует равенство w(Iv) = 1, а последовательное исполнение команд дает w(Dv) = 1. Поэтому описание однопроцессорной машины с фоннеймановской архитектурой будет выглядеть так:

$$\mathbf{I_{1,1}D_{1,1}}$$

Теперь возьмем две машины из класса SIMD: Goodyear Aerospace MPP и ILLIAC IV, причем не будем принимать во внимание разницу в способах обработки данных отдельными процессорными элементами. Единственный поток команд означает | I | = 1 для обеих машин. По тем же соображениям, использованным только что для последовательной машины, для потока команд получаем равенство w(Ia) = w(Iv) = 1. Далее, вспомним, что для доступа к операндам устройство управления MPP рассылает один и тот же адрес всем процессорным элементам, поэтому в этой терминологии MPP имеет единственную последовательность в потоке данных, т.е. |D| = 1. Однако затем выборка данных из памяти и последующая обработка осуществляется в каждом процессорном элементе, поэтому $w(D_v)$=16384, а вся система MPP может быть описана так:

$$\mathbf{I_{1,1}D_{1,16384}}$$

В ILLIAC IV устройство управления, так же, как и в MPP, рассылает один и тот же адрес всем процессорным элементам, однако каждый из них может получить свой уникальный адрес, добавляя содержимое локального индексного регистра. Это означает, что |D| = 64 и в системе присутствуют 64 потока адресов данных, определяющих одиночные потоки операндов, т.е. w(Da) = w(Dv) = 64. Суммируя сказанное, приходим к описанию ILLIAC IV:

$$\mathbf{I_{1,1}D_{64,64}}$$

Для более четкой классификации Шнайдер вводит три предиката для обозначения значений, которые могут принимать величины w(Ia), w(Iv), w(Da) и w(Dv):

s - предикат "равен 1";

c - предикат "от 1 до некоторой (небольшой) константы";

m - предикат "от 1 до произвольно большого конечного числа".

В этих обозначениях, например, фон-неймановская машина принадлежит к классу IssDss. Несмотря на то, что и "c" и "m" в принципе не имеют определенной верхней границы, они отражают разные свойства архитектуры компьютера. Описатель "c" предполагает жесткие ограничения сверху со стороны аппаратуры, и соответствующий параметр не может быть значительно увеличен относительно простыми средствами. Примером может служить число инструкций, упакованных в командном слове VLIW компьютера. С другой стороны, описатель "m" используется тогда, когда обозначаемая величина может быть легко изменена, т.е. другими словами, компьютер по данному параметру масштабируем. Например, относительная проста в увеличении числа процессорных элементов в системе MPP является основанием для того, чтобы отнести ее к классу IssDsm. Конечно же, различие между "c" и "m" в достаточной мере условное и, как правило, порождает массу вопросов. В частности, как описать машину, в которой процессоры связаны через общую шину? С одной стороны, нет никаких принципиальных ограничений на число подключаемых процессоров. Однако каждый дополнительный процессор увеличивает загруженность шины, и при достижении некоторого порога подключение новых процессоров бессмысленно. Как описать такую систему, "c" или "m"? Автор оставляет данный вопрос открытым.

На основе указанных предикатов можно выделить следующие классы компьютеров:

IssDss - фон-неймановские машины;

IssDsc - фон-неймановские машины, в которых заложена возможность выбирать данные, расположенные с разным смещением относительно одного и того же адреса, над которыми будет выполнена одна и та же операция. Примером могут служить компьютеры, имеющие команды, типа одновременного выполнения двух операций сложения над данными в формате полуслова, расположенными по указанному адресу.

IssDsm - SIMD компьютеры без возможности получения уникального адреса для данных в каждом процессорном элементе, включающие MPP, Connection Machine 1 так же, как и систолические массивы.

IssDcc - многомерные SIMD машины - фон-неймановские машины, способные расщеплять поток данных на независимые потоки операндов;

IssDmm - это SIMD компьютеры, имеющие возможность независимой модификации адресов операндов в каждом процессорном элементе, например, ILLIAC IV и Connection Machine 2.

IscDcc - вычислительные системы, выбирающие и исполняющие одновременно несколько команд, для доступа к которым используется один адрес. Типичным примером являются компьютеры с длинным командным словом (VLIW).

IccDcc - многомерные MIMD машины. Фон-неймановские машины, которые могут расщеплять свой цикл выборки/выполнения с целью обработки параллельно нескольких независимых команд.

ImmDmm - к этому классу относятся все компьютеры типа MIMD.

Достаточно ясно, что не нужно рассматривать все возможные комбинации описателей "s", "c" и "m", так как архитектура реальных компьютеров накладывает ряд вполне разумных ограничений. Очевидно, что число адресов w(Sa) не должно превышать числа возвращенных значений w(Sv), которое компьютер может обработать. Отсюда следуют неравенства: w(Ia) <= w(Iv) и w(Da) <= w(Dv). Другим естественным предположением является тот факт, что число выполняемых команд не должно превышать числа обрабатываемых данных: w(Iv) <= w(Dv).

Подводя итог, можно отметить два положительных момента в классификации Шнайдера: более избирательная систематизация SIMD компьютеров и возможность описания нетрадиционных архитектур типа систолических массивов или компьютеров с длинным командным словом. Однако почти все вычислительные системы типа MIMD опять попали в один и тот же класс ImmDmm. Это и не удивительно, так как критерий

классификации, основанный лишь на потоках команд и данных без учета распределенности памяти и топологии межпроцессорной связи, слишком слаб для подобных систем.

4.10. Классификация Дункана

Р.Дункан излагает свой взгляд на проблему классификации архитектур параллельных вычислительных систем, причем сразу определяет тот набор требований, на который, с его точки зрения, может опираться искомая классификация:

Из класса параллельных машин должны быть исключены те, в которых параллелизм заложен лишь на самом низком уровне, включая:

- конвейеризацию на этапе подготовки и выполнения команды (*instruction pipelining*), т.е. частичное перекрытие таких этапов, как дешифрация команды, вычисление адресов операндов, выборка операндов, выполнение команды и сохранение результата;
- наличие в архитектуре нескольких функциональных устройств, работающих независимо, в частности, возможность параллельного выполнения логических и арифметических операций;
- наличие отдельных процессоров ввода/вывода, работающих независимо и параллельно с основными процессорами.

Причины исключения перечисленных выше особенностей автор объясняет следующим образом. Если рассматривать компьютеры, использующие только параллелизм низкого уровня, наравне со всеми остальными, то, во-первых, практически все существующие системы будут классифицированны как "параллельные" (что заведомо не будет позитивным фактором для классификации), и, во-вторых, такие машины будут плохо вписываться в любую модель или концепцию, отражающую параллелизм высокого уровня.

Классификация должна быть согласованной с классификацией Флинна, показавшей правильность выбора идеи потоков команд и данных. Классификация должна описывать архитектуры, которые однозначно не укладываются в систематику Флинна, но, тем не менее, относятся к параллельным архитектурам (например, векторно-конвейерные).

Учитывая вышеизложенные требования, Дункан дает неформальное определение параллельной архитектуры, причем именно неформальность дала ему возможность включить в данный класс компьютеры, которые ранее не вписывались в систематику Флинна. Итак, *параллельная архитектура* - это такой способ организации вычислительной системы, при котором допускается, чтобы множество процессоров (простых или сложных) могло бы работать одновременно, взаимодействуя по мере надобности друг с другом. Следуя этому определению, все разнообразие параллельных архитектур Дункан систематизирует так, как показано на рис.4.14.

По существу систематика очень простая: процессоры системы работают либо синхронно, либо независимо друг от друга, либо в архитектуру системы заложена та или иная модификация идеи MIMD. На следующем уровне происходит детализация в рамках каждого из этих трех классов.

В *систолические архитектурах* обращение к памяти может осуществляться только через определенные процессоры на границе массива. Выборка операндов из памяти и передача данных по массиву осуществляется в одном и том же темпе. Направление передачи данных между процессорами фиксировано. Каждый процессор за интервал времени выполняет небольшую инвариантную последовательность действий.

Гибридные MIMD/SIMD архитектуры, dataflow, reduction и wavefront вычислительные системы осуществляют параллельную обработку информации на основе асинхронного управления, как и MIMD системы. Но они выделены в отдельную группу, поскольку все имеют ряд специфических особенностей, которыми не обладают системы, традиционно относящиеся к MIMD. *MIMD/SIMD* архитектура предполагает, что в MIMD системе

можно выделить группу процессоров, представляющую собой подсистему, работающую в режиме SIMD (PASM, Non-Von). Такие системы отличаются относительной гибкостью, поскольку допускают реконфигурацию в соответствии с особенностями решаемой прикладной задачи.

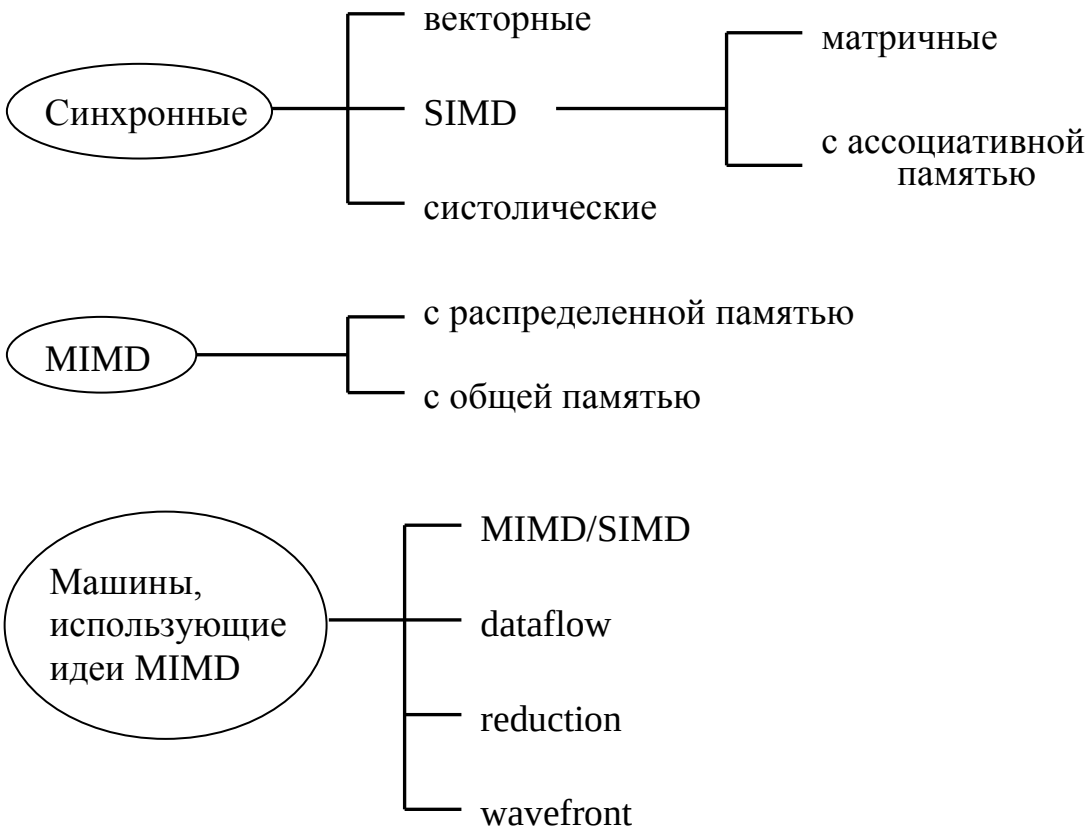


Рис.4.14. Классификация Дункана.

Остальные три вида архитектур используют нетрадиционные модели вычислений. *Dataflow* используют модель, в которой команда может выполнятся сразу же, как только вычислены необходимые операнды. Таким образом, последовательность выполнения команд определяется зависимостью по данным, которая может быть выражена, например, в форме графа.

Модель вычислений, применяемая в редукционных (*reduction)* машинах, иная и состоит в следующем: команда становится доступной для выполнения тогда и только тогда, когда результат ее работы требуется другой, доступной для выполнения, команде в качестве операнда.

Wavefront array архитектура объединяет в себе идею систолической обработки данных и модель вычислений, используемой в dataflow. В данной архитектуре процессоры объединяются в модули и фиксируются связи, по которым процессоры могут взаимодействовать друг с другом. Однако, в противоположность ритмичной работе систолических массивов, данная архитектура использует асинхронный механизм связи с подтверждением (handshaking), из-за чего "фронт волны" вычислений может менять свою форму по мере прохождения по всему множеству процессоров.

4.11. Классификация Скилликорна

В 1989 году была сделана очередная попытка расширить классификацию Флинна и, тем самым, преодолеть ее недостатки. Д.Скилликорн разработал подход [6], пригодный для описания свойств многопроцессорных систем и некоторых нетрадиционных архитек-

тур, в частности dataflow и reduction machine.

Предлагается рассматривать архитектуру любого компьютера, как абстрактную структуру, состоящую из четырех компонент:

1. *процессор команд* (IP - Instruction Processor) - функциональное устройство, работающее, как интерпретатор команд; в системе, вообще говоря, может отсутствовать;

2. *процессор данных* (DP - Data Processor) - функциональное устройство, работающее как преобразователь данных, в соответствии с арифметическими операциями;

3. *иерархия памяти* (IM - Instruction Memory, DM - Data Memory) - запоминающее устройство, в котором хранятся данные и команды, пересылаемые между процессорами;

4. *переключатель* - абстрактное устройство, обеспечивающее связь между процессорами и памятью.

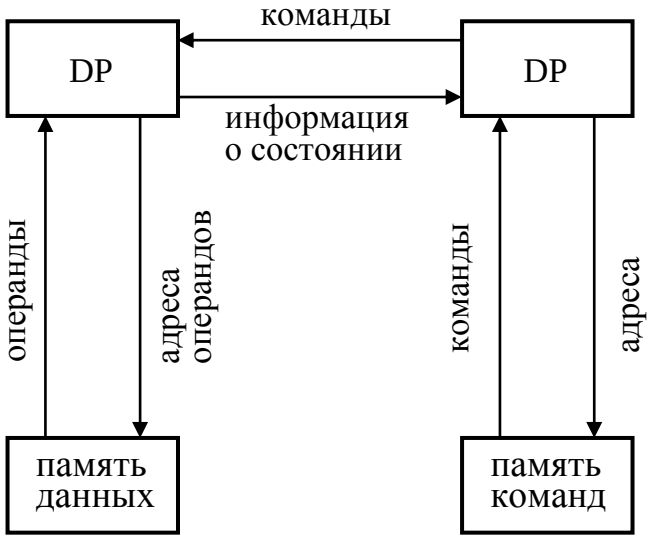


Рис.4.15. Классификация Скилликорна.

Функции процессора команд во многом схожи с функциями устройств управления последовательных машин и, согласно Д.Скилликорну, сводятся к следующим:

- на основе своего состояния и полученной от DP информации IP определяет адрес команды, которая будет выполняться следующей;
- осуществляет доступ к IM для выборки команды;
- получает и декодирует выбранную команду;
- сообщает DP команду, которую надо выполнить;
- определяет адреса операндов и посылает их в DP;
- получает от DP информацию о результате выполнения команды.

Функции процессора данных делают его , во многом, похожим на арифметическое устройство традиционных процессоров:

- DP получает от IP команду, которую надо выполнить;

получает от IP адреса операндов;

- выбирает операнды из DM;
- выполняет команду;
- запоминает результат в DM;
- возвращает в IP информацию о состоянии после выполнения команды.

В терминах таким образом определенных основных частей компьютера структуру традиционной фон-неймановской архитектуры можно представить в следующем виде (рис.4.15).

Это один из самых простых видов архитектуры, не содержащих переключателей. Для описания параллельных вычислительных систем автор зафиксировал четыре типа переключателей, без какой-либо явной связи с типом устройств, которые они соединяют:

1-1 - переключатель такого типа связывает пару функциональных устройств;

n-n - переключатель связывает i-е устройство из одного множества устройств с i-м устройством из другого множества, т.е. фиксирует попарную связь;

1-n - переключатель соединяет одно выделенное устройство со всеми функциональными устройствами из некоторого набора;

n x n - каждое функциональное устройство одного множества может быть связано с любым устройством другого множества, и наоборот.

Примеров подобных переключателей можно привести очень много. Так, все матричные процессоры имеют переключатель типа 1-n для связи единственного процессора команд со всеми процессорами данных. В компьютерах семейства Connection Machine каждый процессор данных имеет свою локальную память, следовательно, связь будет описываться как n-n. В тоже время, каждый процессор команд может связаться с любым другим процессором, поэтому данная связь будет описана как nxn.

Классификация Д.Скилликорна состоит из двух уровней. На первом уровне она проводится на основе восьми характеристик:

1. количество процессоров команд (IP);
2. число запоминающих устройств (модулей памяти) команд (IM);
3. тип переключателя между IP и IM;
4. количество процессоров данных (DP);
5. число запоминающих устройств (модулей памяти) данных (DM);
6. тип переключателя между DP и DM;
7. тип переключателя между IP и DP;
8. тип переключателя между DP и DP.

Рассмотрим упомянутый выше компьютер Connection Machine 2 (CM-2). В терминах данных характеристик его можно описать:

(1, 1, 1-1, n, n, n-n, 1-n, nxn).

Условное изображение архитектуры приведено на рис.4.16.

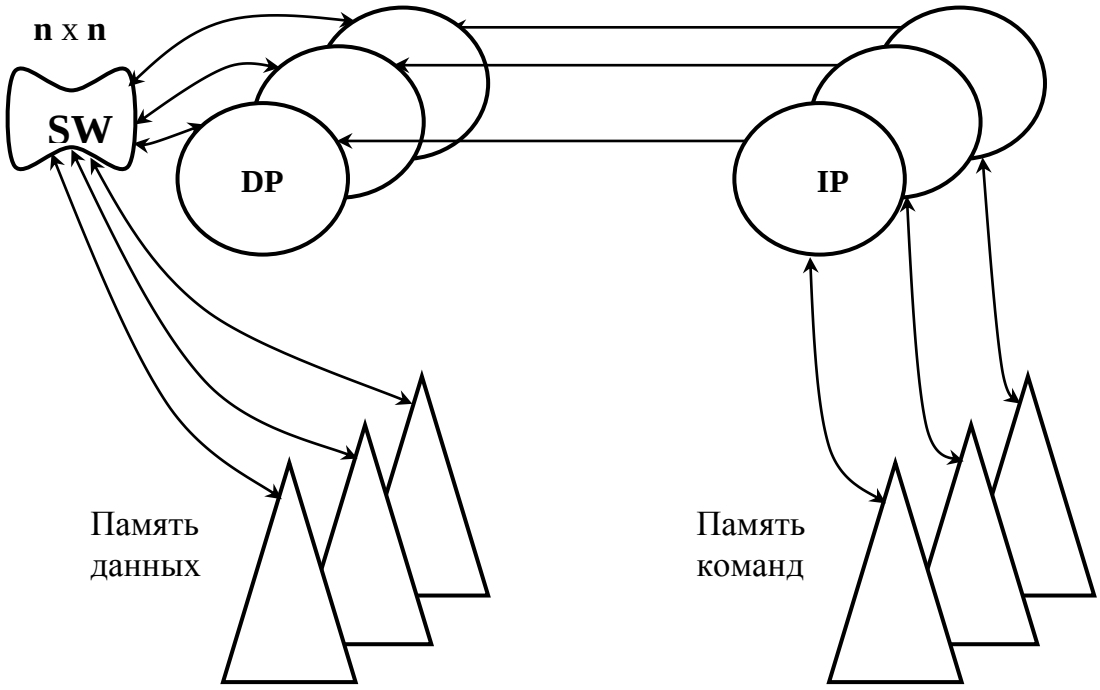


Рис.4.16. Структура Connection Machine 2.

Для сильно связанных мультипроцессоров (BBN Butterfly, C.mmp) ситуация иная. Такие системы состоят из множества процессоров, соединенных с модулями памяти с помощью динамического переключателя. Задержка при доступе любого процессора к любому модулю памяти примерно одинакова. Связь и синхронизация между процессорами осуществляется через общие (разделяемые) переменные. Описание таких машин в рамках данной классификации выглядит так:

(**n, n, n-n, n, n, n** x **n, n - n**, нет),

а саму архитектуру можно изобразить так, как на рис.4.17.

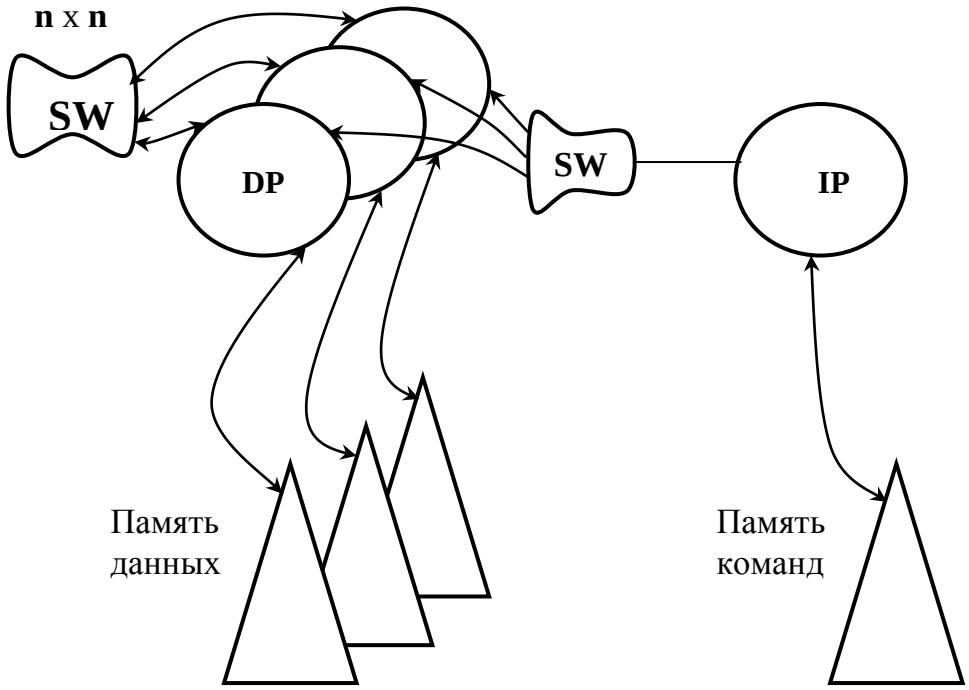


Рис.4.17. Структура BBN Butterfly.

Используя введенные характеристики и предполагая, что рассмотрение количественных характеристик можно ограничить только тремя возможными вариантами значений: 0, 1 и n (т.е. больше одного), можно получить 28 классов архитектур.

В классах 1-5 находятся компьютеры типа dataflow и reduction, не имеющие процессоров команд в обычном понимании этого слова. Класс 6 это классическая фоннеймановская последовательная машина. Все разновидности матричных процессоров содержатся в классах 7-10. Классы 11 и 12 отвечают компьютерам типа MISD классификации Флинна и на настоящий момент, по мнению автора, пусты. Классы с 13-го по 28-й занимают всесозможные варианты мультипроцессоров, причем в 13-20 классах находятся машины с достаточно привычной архитектурой, в то время, как архитектура классов 21-28 пока выглядит экзотично.

На втором уровне классификации Д.Скилликорн просто уточняет описание, сделанное на первом уровне, добавляя возможность конвейерной обработки в процессорах команд и данных.

В конце данного описания имеет смысл привести сформулированные автором три цели, которым должна служить хорошо построенная классификация:

- облегчать понимание того, что достигнуто на сегодняшний день в области архитектур вычислительных систем, и какие архитектуры имеют лучшие перспективы в будущем;
- подсказывать новые пути организации архитектур - речь идет о тех классах, которые в настоящее время по разным причинам пусты;
- показывать, за счет каких структурных особенностей достигается увеличение производительности различных вычислительных систем; с этой точки зрения, классификация может служить моделью для анализа производительности.

4.12. Классификация Хокни

Р. Хокни - известный английский специалист в области параллельных вычислительных систем, разработал свой подход к классификации, введенной им для систематизации

компьютеров, попадающих в класс MIMD по систематике Флинна.

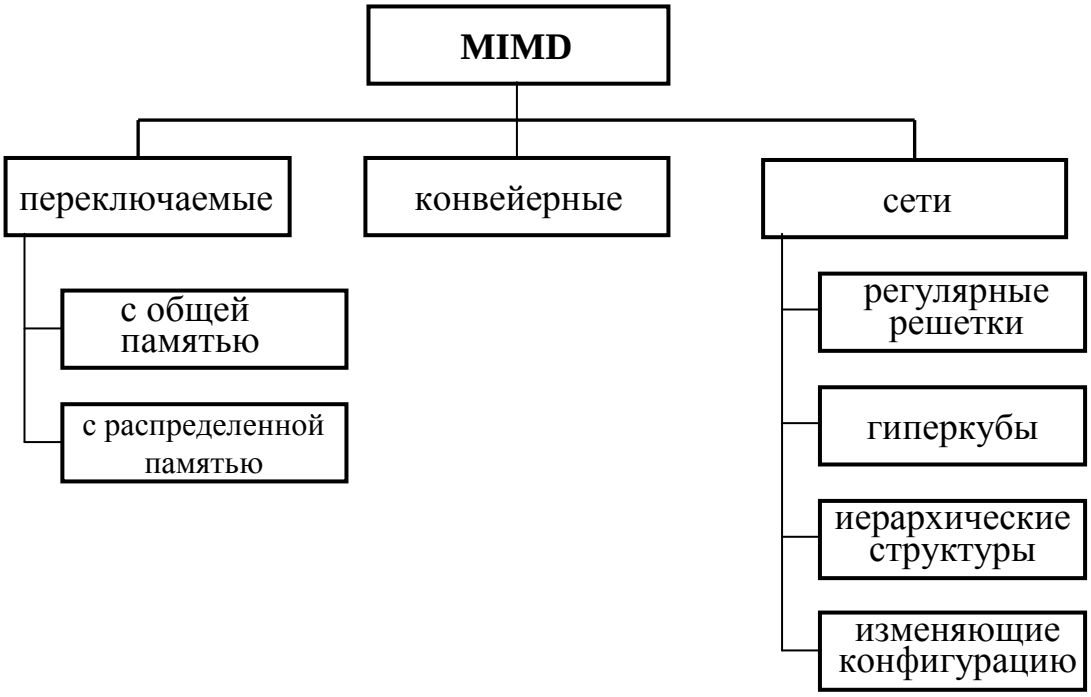


Рис.4.18. Классификация Хокни.

Как отмечалось выше (см. классификацию Флинна), класс MIMD чрезвычайно широк, причем наряду с большим числом компьютеров он объединяет и целое множество различных типов архитектур. Хокни, пытаясь систематизировать архитектуры внутри этого класса, получил иерархическую структуру, представленную на рис.4.18.

Основная идея классификации состоит в следующем. Множественный поток команд может быть обработан двумя способами: либо одним конвейерным устройством обработки, работающем в режиме разделения времени для отдельных потоков, либо каждый поток обрабатывается своим собственным устройством. Первая возможность используется в MIMD компьютерах, которые автор называет конвейерными (например, процессорные модули в Denelcor HEP). Архитектуры, использующие вторую возможность, в свою очередь опять делятся на два класса:

MIMD компьютеры, в которых возможна прямая связь каждого процессора с каждым, реализуемая с помощью переключателя;

MIMD компьютеры, в которых прямая связь каждого процессора возможна только с ближайшими соседями по сети, а взаимодействие удаленных процессоров поддерживается специальной системой маршрутизации через процессоры-посредники.

Далее, среди MIMD машин с переключателем Хокни выделяет те, в которых вся память распределена среди процессоров как их локальная память (например, PASM, PRINGLE). В этом случае общение самих процессоров реализуется с помощью очень сложного переключателя, составляющего значительную часть компьютера. Такие машины носят название MIMD машин с распределенной памятью. Если память это разделяемый ресурс, доступный всем процессорам через переключатель, то такие MIMD являются системами с общей памятью (Cray X-MP, BBN Butterfly). В соответствии с типом переключателей можно проводить классификацию и далее: простой переключатель, многокаскадный переключатель, общая шина.

Многие современные вычислительные системы имеют как общую разделяемую память, так и распределенную локальную. Такие системы автор рассматривает как гибридные MIMD с переключателем.

При рассмотрении MIMD машин с сетевой структурой считается, что все они имеют распределенную память, а дальнейшая классификация проводится в соответствии с топологией сети: звездообразная сеть (lCAP), регулярные решетки разной размерности (Intel Paragon, Cray T3D), гиперкубы (NCube, Intel iPCS), сети с иерархической структурой, такой, как деревья, пирамиды, кластеры (Cm^*, CEDAR) и, наконец, сети, изменяющие свою конфигурацию.

Заметим, что если архитектура компьютера спроектирована с использованием нескольких сетей с различной топологией, то, по всей видимости, по аналогии с гибридными MIMD с переключателями, их стоит назвать гибридными сетевыми MIMD, а использующие идеи разных классов - просто гибридными MIMD. Типичным представителем последней группы, в частности, является компьютер Connection Machine 2, имеющим на внешнем уровне топологию гиперкуба, каждый узел которого является кластером процессоров с полной связью.

5. Особенности программирования для высокопроизводительных систем

Если у Вас есть векторно-конвейерная система с длиной векторного регистра 128 элементов или параллельная система с таким же числом процессоров, то теоретически Ваша программа может быть выполнена в 128 раз быстрее, чем на аналогичной по быстродействию однопроцессорной скалярной машине. Это применение векторных и параллельных систем не только возможно, но и целесообразно. Выигрыш в быстродействии по отношению к обычной системе может окупить финансовые затраты на аренду или покупку суперЭВМ и трудовые затраты на написание и оптимизацию векторной или параллельной версии программы.

Целью программиста, наверное, не должно быть получение правильного результата вычислений любой ценой, но получение правильного результата наибыстрейшим, оптимальным способом обязательно. Если программа предназначена для однократного использования, то лучше написать ее как можно проще, не оптимизируя ее быстродействие и используемую память, чтобы потратить минимум усилий на тестирование и отладку. Если программа предназначена для частого использования или время ее работы будет гораздо больше времени ее написания и отладки, то не следует жалеть труда на оптимизацию ее быстродействия. Однако, в результате программа, оптимальная для одного типа систем, окажется совсем не оптимальной для машин других типов.

Методы программирования для скалярных, векторных и параллельных систем коренным образом отличаются друг от друга! Конечно, любая программа, написанная на языке высокого уровня, может быть оттранслирована в коды любой ЭВМ и исполнена как на скалярной, так и на векторной или параллельной машинах и результаты работы всех программ скорее всего будут близкими. Эту оговорку надо принять во внимание. Дело в том, что результат действий с вещественными числами может изменяться при изменении последовательности операций, т.к. в компьютерной арифметике закон сочетаний не всегда выполняется**:** **(A+B)+C не всегда равно A+(B+C).**

Самый простой вариант попробовать ускорить имеющуюся программу - это воспользоваться встроенными в транслятор (обычно с ФОРТРАНа или Си) средствами векторизации или распараллеливания. При этом никаких изменений в программу вносить не придется. Однако вероятность существенного ускорения (в разы или десятки раз) невелика. Трансляторы с ФОРТРАНа и Си векторизуют и распараллеливают программы очень аккуратно и при любых сомнениях в независимости обрабатываемых данных оптимизация не проводится. Поэтому, кстати, и не приходится ожидать ошибок от компиляторов, если

программист явно не указывает компилятору выполнить векторную или параллельную оптимизацию какой-либо части программы.

Второй этап работы с такой программой - анализ затрачиваемого времени разными частями программы и определение наиболее ресурсопотребляющих частей. Последующие усилия должны быть направлены именно на оптимизацию этих частей. В программах наиболее затратными являются циклы и усилия компилятора направлены прежде всего на векторизацию и распараллеливание циклов. Диагностика компилятора поможет установить причины, мешающие векторизовать и распараллелить циклы. Возможно, что простыми действиями удастся устранить эти причины. Это может быть простое исправление стиля программы, перестановка местами операторов (цикла и условных), разделение одного цикла на несколько, удаление из критических частей программы лишних операторов (типа операторов отладочной печати). Небольшие усилия могут дать здесь весьма существенный выигрыш в быстродействии.

Третий этап - замена алгоритма вычислений в наиболее критичных частях программы. Способы написания оптимальных (с точки зрения быстродействия) программ существенно отличаются в двух парадигмах программирования - в последовательной и в параллельной (векторной). Поэтому программа, оптимальная для скалярного процессора, с большой вероятностью не может быть векторизована или распараллелена. В то же время специальным образом написанная программа для векторных или параллельных систем будет исполняться на скалярных машинах довольно медленно. Замена алгоритма в наиболее критических частях программы может привести к серьезному ускорению программы при относительно небольших потраченных усилиях. Дополнительные возможности предоставляют специальные векторные и параллельные библиотеки подпрограмм. Используя библиотечные функции, которые оптимизированы для конкретной ЭВМ, можно упростить себе задачу по написанию и отладке программы. Единственный недостаток данного подхода состоит в том, что программа может стать не переносимой на другие машины (даже того же класса), если на них не окажется аналогичной библиотеки.

Написание программы "с нуля" одинаково сложно (или одинаково просто) для машин любых типов. Этот способ является идеальным для разработки эффективных, высокопроизводительных векторных или параллельных программ. Начинать надо с изучения специфики программирования для векторных и параллельных систем, изучения алгоритмов, которые наиболее эффективно реализуются на ЭВМ данных типов. После этого надо проанализировать поставленную задачу и определить возможность применения векторизуемых и распараллеливаемых алгоритмов для решения конкретной задачи. Возможно, что придется переформулировать какие-то части задачи, чтобы они решались с применением векторных или параллельных алгоритмов. Программа, специально написанная для векторных или параллельных систем, даст наибольшее ускорение при ее векторизации и распараллеливании.

Программист должен быть готов к поддержанию одновременно нескольких (двух или даже трех) версий программы, предназначенных для работы на ЭВМ разного типа. Это, наверно, самая существенная из всех "издержек", связанная с одновременным использованием скалярных, векторных и параллельных машин.

5.1. Применение разных языков программирования

Перенос готовой программы и оптимизация ее для исполнения на машине другого типа может потребовать весьма существенных усилий по исправлению стиля написаниния программы. Опыт показывает, что проще всего адаптировать к исполнению на векторных или параллельных машинах программы на ФОРТРАНе. Конструкции ФОРТРАНа для организации циклов и для работы с массивами менее разнообразны, чем в Си, а именно циклы по обработке массивов и являются главными объектами оптимизации при векториза-

ции или распараллеливанию программ. Более того, в ФОРТРАНе-90 встроенные функции по работе с массивами и арифметические операции с массивами имеют довольно эффективную реализацию в виде стандартных библиотечных векторных или параллельных функций или даже в виде генерируемого компилятором объектного кода (*in line*).
Больше всего трудностей появится при переносе программ на Си, оптимальным образом написанных для скалярных процессоров. Следующие программы на Си показывают два принципиально разных подхода к операциям с массивами. Традиционный Си'шный подход к копированию массивов является недопустимым при написании векторных или параллельных программ. Приведенный ниже цикл не может быть векторизован или распараллелен из-за того, что значение ссылок на элементы массивов вычисляются рекуррентно (зависят от значений на предыдущем шаге):

```
      float x[100], a[100];
register float *xx = x, *aa = a;
      register int i;
      for( i=0; i<100; i++){ *xx++ = *aa++ };
```

В то время как другой цикл, написанный совсем не в традициях Си, может быть распараллелен и векторизован:

```
      float x[100], a[100];
      register int i;
      for( i=0; i<100; i++){ x[i] = a[i]; }
```

Сравните: ФОРТРАН-77 позволяет единственным способом записать цикл и этот способ совпадает со вторым, нетрадиционным способом в Си:

```
      real x(100), a(100)
      do i=1, 100
          x(i) = a(i)
      enddo
```

ФОРТРАН-90 разрешает применить векторную операцию присваивания, что упрощает анализ программы транслятором:

```
      real x(100), a(100)
      x = a ! векторное присваивание
```

Приведенный пример показывает, что существенным фактором при адаптации готовых программ к работе на параллельных и векторных систем является не только стиль написания программы, но и сам использованный язык программирования.

5.2. Различие и сходство между распараллеливанием и векторизацией программ

Сходство алгоритмов - параллелизм данных. Из архитектуры векторных и параллельных компьютеров следует самое важное свойство алгоритмов, которые могут быть эффективно исполнены на таких системах: эти алгоритмы должны включать одинаковые действия над некоторыми наборами данных (массивами), при этом результаты действий над любой частью данных не должны зависеть от результатов действий над другой их частью. Иначе говоря, последовательность вычисления величин не должна играть роль в

данном алгоритме. Это есть параллелизм данных. Простейшим примером такого алгоритма может служить следующий фрагмент программы:

```
 real x(100), a(100), b(100)
 do   i=1, 100
x(i) = real(i)**2 + 3.0
a(i) = b(i) + x(i)
 enddo
```

или в другой интерпретации:

```
 real x(100), a(100), b(100)
 do   i=100, 1, -1
      x(i) = real(i)**2 + 3.0
      a(i) = b(i) + x(i)
 enddo
```

Предположим, что i=5, тогда тело цикла примет вид:

```
 x(5) = 5.0**2 + 3.0
 a(5) = b(5) + x(5)
```

Порядок исполнения этих двух операторов не может быть изменен, но 5-ые элементы массивов a и x могут быть вычислены независимо от 4-го, 6-го и других элементов. Приведенная программа удовлетворяет основному требованию к векторизуемым алгоритмам - для любого значения i вычисления можно проводить независимо.

Предположим, что у нас имеется векторно-конвейерная ЭВМ с векторными регистрами длиной 100 элементов. Тогда весь цикл по всем элементам массива записывается как линейная (не циклическая) последовательность команд, а каждая команда оперирует со всеми 100 элементами массива. Выигрыш в увеличении быстродействия программ (по отношению к быстродействию не векторизованных программ на той же ЭВМ) может ожидаться равным числу элементов в массиве.

Теперь представим, что у нас есть 100-процессорная параллельная система и каждый процессор имеет прямой доступ ко всем элементам массивов. Тогда для каждого скалярного процессора можно написать программу для вычисления значений a(i) и x(i) для одного конкретного i, совпадающего с номером процессора (процессор должен знать свой номер). В принципе такая программа также, как и векторная, будет линейной. Т.к. у нас в распоряжении 100 процессоров, то после исполнения каждым процессором своих команд все 100 элементов массивов a и x будут вычислены. Очевидно, что это будет в 100 раз быстрее, чем вычисление всех элементов одним процессором.

Различие алгоритмов - параллелизм действий. Можно проследить аналогию в векторной и параллельной реализациях предыдущей программы. Каждая машинная команда вызывает действия сразу над 100 числами: в векторной программе явно выполняются операции над всеми элементами регистра, в параллельной программе каждый из 100 процессоров выполняет более или менее синхронно одинаковые машинные команды, оперирует со своими собственными регистрами и в результате выполняются действия одновременно со 100 числами.

Справедливо следующее утверждение: алгоритм, который можно векторизовать, можно и распараллелить. Обратное утверждение не всегда верно.

В многопроцессорной системе каждый процессор исполняет свой поток команд. В общем случае для каждого из процессоров параллельной ЭВМ можно составить свою программу, не повторяющую программы для других процессоров. Предположим, что скалярная величина y есть сумма двух функций: **y = F(x) + G(x).**

Тогда один процессор может считать значение **F(x)**, а второй - **G(x)**. После счета достаточно сложить полученные два числа, чтобы получить требуемое значение y. Такой параллелизм действий может быть достигнут (и так поступают наиболее часто) в единой программе для обоих процессоров:

```
if( номер_процессора.eq. 1 ) then
    y1 = F(x)
    else if( номер_процессора.eq. 2 ) then
        y2 = G(x)
endif
ждать завершения работы обоих процессоров
if( номер_процессора.eq. 1 ) then
    y = y1 + y2
endif
```

Параллельность действий (т.е. применение параллельных систем) дает большое преимущество в программировании перед чистой параллельностью данных (применением векторных систем). Даже простой вызов подпрограммы приводит к невозможности векторизации цикла, в то время, как распараллеливание возможно:

```
real a(100)
do i=1, 100
   call proc( a(i) )
enddo
```

Теперь мы подробнее рассмотрим преимущества при векторизации и распараллеливании программ.

5.3. Векторные машины и векторные программы

Предельное быстродействие векторных программ. Мы будем здесь рассматривать машины с векторными регистрами. Векторный процессор выполняет математические операции сразу над всеми элементами векторного регистра. Если число элементов регистра равно 128, то операция над всеми 128 числами выполняется в векторном режиме так же быстро, как над одним числом в скалярном режиме. Это и есть теоретический предел повышения быстродействия программ при их векторизации. Однако необходимо учесть, что любая векторная операция требует больше машинных тактов для своего исполнения, чем такая же скалярная операция. С другой стороны циклическое N-кратное исполнение скалярной команды для обработки массива требует исполнения еще нескольких команд, организующих собственно цикл. В результате исполнение векторной команды может оказаться эффективнее более, чем в 128 раз. Для простоты мы будем считать, что предельное повышение эффективности векторных программ равно числу элементов в векторном регистре ЭВМ. Чаще всего это 128 или 256.

В любой программе существуют две части - векторизуемая и не векторизуемая. Например алгоритмы построения последовательностей, заданных рекуррентным отношением, нельзя векторизовать - каждый последующий элемент зависит от предыдущих и, соответственно, не может быть вычислен ни ранее, ни одновременно с предыдущими. Другие невектори-

зуемые части программ - ввод/вывод, вызов подпрограмм или функций, организация циклов, разветвленные алгоритмы, работа со скалярными величинами. Это довольно широкий класс подзадач, он гораздо шире класса векторизуемых алгоритмов. При векторизации программ на самом деле ускоряется выполнение только части программы (большей или меньшей). Поэтому каждую программу можно представить такой упрощенной схемой (рис.5.1.), если собрать все векторизуемые и все скалярные части в единые блоки.

Для начала исполним программу в полностью скалярном варианте (т.е. умышленно

Рис.5.1. Основные части программы.

не будем векторизовать исполняемый код). Обозначим время исполнения каждой из частей программы через T1...T5. Тогда полное время работы невекторизуемых частей программы будет равно

$$Tскал = T1 + T3 + T4 + T5,$$

а полное время работы программы будет

$$T' = Tскал + T2.$$

Далее перетранслируем программу в режиме векторизации. Единственная часть программы, которая ускорит свое выполнение, будет вторая. Предположим, что мы достигли ускорения работы этой части в N раз. Тогда полное время работы всей программы составит

$$T'' = Tскал + T2/N.$$

А выигрыш в эффективности работы всей программы будет

$$P = \frac{T'}{T''} = \frac{Tскал + T2}{Tскал + T2/N},$$ что совсем не равно N.

Предположим, что Tскал = 1с, T2 = 99с, а N=100 (очень хороший показатель для 128-элементных векторных процессоров). В нашем варианте эффективность P будет всего (1+99)/(1+99/100)=50 или 40% от предельно возможных 128 раз, а при Tскал = 2с значение P будет (2+98)/(2+98/100)=34 (27%). Хотя само по себе увеличение быстродействия программы в 50 или даже в 30 раз при ее векторизации является очень большим (вычисления будут занимать 1 сутки вместо 1 месяца), но предельно возможное ускорение в 128 (или 256) раз не может быть достигнуто на векторных ЭВМ. Практика показывает, что хорошим показателем увеличения быстродействия P можно считать уже значения 6-10, что соответствует времени исполнения скалярной части всего 10-15% от полного времени исполнения программы (T2 составляет 85-90%).

Дополнительные затраты на организацию векторных вычислений во время работы программы. Для работы на векторных ЭВМ наиболее удобными являются массивы с длиной, равной длине векторного регистра. Однако это благое пожелание очень редко выполняется. Более того, часто число элементов массива вообще не кратно 128. Рассмотрим простейший цикл, который можно векторизовать. Пусть дан массив m длиной 128, который надо заполнить по следующему алгоритму:

```
do i=1, 128
    m(i) = i
enddo
```

Мы будем пользоваться командами ассемблера несуществующей ЭВМ, но вполне отражающими смысл операций с векторными регистрами. Приведенный цикл можно записать на ассемблере так:

```
SETLEN #128  ; установить используемое число
                ; элементов в векторных регистрах (во всех)
SETINC #4       ; установить смещение к последующему
                ; элементу массива в памяти (4 байта)
SETNUM v0    ; записать в элементы векторного регистра v0
                ; их номера начиная с нуля и кончая 127
ADD #1, v0      ; добавить 1 к каждому элементу регистра v0
SAVE v0, m      ; записать элементы v0 в ОЗУ в последовательные
                ; слова (смещение = 4) начиная с адреса m.
```

Обратите внимание на 3 команду - каждый элемент векторного регистра "знает" свой номер. Это делает очень простым вычисление переменной цикла: после добавления 1 значение переменной получается записанным в соответствующий элемент вектора. Всего 5 последовательных команд векторного процессора выполняют цикл из 128 повторений. Здесь нет ни команд сравнения, ни условных переходов.

Теперь увеличим размер массива и число повторений цикла до N:

```
do i=1, N
    m(i) = i
enddo
```

Для правильной работы процессора мы обязаны установить число используемых элементов вектора не более, чем 128. Если N будет произвольным, то нам придется превратить данный одинарный цикл в двойной:

```
 1 do  inc=0, N-1, 128
 2       NN = min0( 128, N-inc )
 3   do  i=1, NN
           m(inc+i) = inc+i
       enddo
enddo
```

Цикл с меткой 1 выполняяет "разбиение" массива на подмассивы длиной 128 элементов. Переменная inc имеет смысл смещения от первого элемента массива к очередному подмассиву: 0, 128, 256... Переменная NN определяет длину подмассива. Обычно она равна 128, но последний подмассив может иметь меньшую длину, если N не кратно 128. Функция min0 выбора минимального значения в операторе с меткой 2 выдает значение не 128 только для последнего подмассива. Внутренний цикл с меткой 3 практически эквивалентен циклу из предыдущего примера. Имеется только 3 отличия:

- число элементов в векторном регистре равно NN,
- к параметру цикла дополнительно надо добавлять значение inc,
- значение регистра записывать в память начиная не с начала массива, а с элемента с номером i+inc

Этот цикл может быть записан в машинных командах примерно так:

```
 SETLEN NN    ; число элементов в векторных регистрах = NN
 SETINC #4    ; смещение к последующему элементу массива
 SETNUM v0    ; записать в элементы векторного регистра v0
; их номера начиная с нуля и кончая 127
 ADD #1, v0     ; добавить 1 к каждому элементу регистра v0
 ADD inc, v0    ; добавить значение переменной inc
 MOVE inc, r0   ; записать в скалярный регистр r0 значение
                ; переменной inc - смещение от первого
                ; элемента массива к m(inc+1)
 MUL #4, r0     ; умножить на 4 - смещение в байтах
 ADD #m, r0     ; добавить адрес массива m, получается адрес
               ; элемента m(inc+1)
 SAVE v0, @r0 ; записать элементы v0 в ОЗУ в последовательные
                ; слова (смещение = 4) начиная с адреса,
                ; хранящегося в регистре r0.
```

По отношению к простейшему циклу добавились одна векторная и три скалярных команды. Однако это только внутренний цикл. Охватывающий его цикл с меткой 1 и вычисление NN создадут дополнительный код, который будет выполняться столько же раз, сколько и код для внутреннего цикла. Транслятор всегда будет создавать охватывающий цикл, если N есть переменная, а не константа со значением от 1 до 128 (для 128-элементного векторного регистра).

Из сказанного выше следует, что эффективность векторной программы будет невысокой при работе с небольшими массивами, длина которых заранее неизвестна. Циклы с тремя повторениями могут в векторном режиме выполняться медленнее, чем в скалярном на той же машине.

Ограниченное число векторных регистров. Число векторных регистров в процессоре обычно гораздо меньше, чем число скалярных регистров (8 регистров). Это накладывает ограничения на сложность выражений (т.е. число массивов и скалярных переменных), стоящих в теле векторизуемого цикла. К числу непосредственно массивов (индексированных переменных) добавляется сам параметр цикла. Выше было показано, что простая скалярная переменная i "векторизуется" и преобразуется в массив из 128 (или NN) последовательных значений параметра цикла. Очевидно, что любые арифметические выражения, содержащие i, в дополнение к индексированным этой переменной элементам массивов, и все скалярные переменные, зависящие от i, тоже должны быть векторизованы. Число используемых векторных переменных легко может превысить число векторных регистров даже в выражениях, явным образом содержащих только один-два массива. В такой ситуации транслятор создает в ОЗУ дополнительные 128-элементные массивы для сохранения промежуточных значений векторных регистров и при вычислении сложных выражений предусматривает сохранение и загрузку векторных регистров из этих временных массивов. Перезагрузка регистров (*swaping*) может стать фактором, существенно замедляющим программу.

Не следует опасаться, что все используемые массивы программа будет постоянно хранить в векторных регистрах. "Оптимизатор" транслятора может переставлять местами строки исходного текста так, чтобы не изменить смысл программы, но в то же время уменьшить число используемых регистров. Например, написанный программистом код

```
do i=1, N
    x(i) = i
    y(i) = 0.0
    z(i) = x(i)+1.0
enddo
```

может быть изменен на такой:

```
   do i=1, N
1     x(i) = i
2     z(i) = x(i)+1.0
3     y(i) = 0.0
   enddo
```

Но во втором примере после векторизации i в операторе 1 и добавлении к этому вектору 1.0 в операторе 2 можно повторно использовать тот же векторный регистр для векторизации нуля в операторе 3. В этом варианте цикла один векторный регистр будет использован последовательно для разных целей.

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.

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 век-

торный 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 выриант 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 систем 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.

Использование векторных операций и функций ФОРТРАНа-90. Векторные операции и функции - это, пожалуй, единственные конструкции ФОРТРАНа-90, которые нашли быстрое и эффективное воплощение в ФОРТРАНе для векторных ЭВМ. Практически на всех машинах (векторных и скалярных) стандартным сейчас является ФОРТРАН-77. Поэтому векторные операции и функции являются для него расширением и их использование в программах может привести к несовместимости исходного кода между различными ЭВМ.

5.4. Параллельные системы и параллельные программы

Предельное быстродействие параллельных программ. При работе на параллельных ЭВМ пользователь имеет возможность запускать программу или на всех процессорах сразу, или на ограниченном их числе. Поскольку все процессоры в параллельных системах одинаковые (в составе параллельной ЭВМ могут работать еще и специализированные процессоры ввода/вывода, но на них вычисления не производятся), то можно ожидать, что программа будет выполняться в N раз быстрее (N – число процессоров). Однако и здесь существуют ограничения. Как и в векторных программах, в любом параллельном алгоритме присутствуют параллельная и последовательная части. В отличие от векторизации, внутренние циклы, ветвящиеся алгоритмы, вызовы подпрограмм и функций не являются препятствием для распараллеливания программы. При распараллеливании программы могут быть оптимизированы внешние, самые всеобъемлющие циклы. Однако любые рекуррентные вычисления, ввод/вывод, вычисления, понижающие размерность массивов (вплоть до скаляра), не могут быть (полностью) распараллелены.

Исполнение разных частей программы разными процессорами или, если быть точнее, разными процессами вносит дополнительный обязательный фрагмент в программу, а именно, обмен данными между процессами. Современные параллельные системы исполняют разные копии одинаковой программы в качестве отдельных задач, т.е. процессов.

Каждый процесс может иметь свои локальные данные и глобальные данные, к которым есть доступ у всех процессов. Результаты, сосчитанные в одних процессах, в определенные моменты должны передаваться в другой или другие процессы для дальнейшей работы. Это процесс обмена данными.

Рассмотрим такой фрагмент программы, который будет исполняться на 4-х процессорной ЭВМ:

```
   real x(4)
 1 do i=1,4
       x(i) = func(i)
   enddo
 2 s = 0.0
 3 do i=1,4
       s = s + x(i)
   enddo
```

Все 4 элемента массива **x** можно вычислить параллельно в цикле с меткой 1. При этом вообще цикл не понадобится, т.к. у нас число процессов будет равно числу искомых элементов массива. Переменную s должен инициализовать только один процесс - это последовательный фрагмент программы. Цикл с меткой 3 должен также выполняться одним процессом. Для этого надо сначала передать этому процессу все значения x(i), i=1..4, из других процессов. Этот цикл не сможет начаться раньше, чем будет вычислен и передан последний (не по номеру, а по времени) элемент массива x. Т.е. главный процесс (проводящий суммирование) будет ожидать завершение передачи элементов массива всеми остальными процессами.

Время обмена данными зависит от архитектуры параллельной системы. Оно может быть равно нулю для многопроцессорных рабочих станций с общей оперативной памятью и организацией распараллеливания в пределах одного процесса или составлять значительную величину при обмене в кластерах, связанных компьютерной сетью.

Выводы: объем данных, предназначенных для обмена, должен быть по возможности меньше, а последовательная часть программы должна быть как можно быстрее. Часто это удается совмещать путем проведения частичных вычислений в параллельном режиме (например, вычисление частичных сумм) с последующей передачей промежуточных результатов в главный процесс (например, для вычисления полной суммы).

Синхронизация процессов, равномерность загрузки процессов. Еще один важный фактор, влияющий на ускорение работы параллельных программ, есть равномерность загрузки процессов. При обсуждении предыдущей программы было сказано, что главный процесс перед началом исполнения цикла 3 должен получить все элементы массива x. Даже если собственно время обмена данными будет равно нулю, то все равно цикл не сможет начаться до окончания вычисления последнего (не по номеру, а по времени) из элементов x. За этим следит одна из важнейших частей параллельного алгоритма, которая часто называется "барьером" и осуществляет синхронизацию процессов.

Предположим, что время вычисления x(i) будет равно 1, 2, 3 и 4 секундам для соответствующих i. Тогда самое последнее значение x(4) будет получено через 4 с после начала вычислений, а цикл 3 не сможет начаться ранее этого времени.

Если к концу предыдущей программы дописать такой распараллеливаемый фрагмент:

```
 4 do i=1,4
       x(i) = x(i)/s
   enddo
```

то, несмотря на незагруженность трех процессов исполнением цикла 3, они не смогут продолжить работу до его (цикла) окончания и рассылки главным процессом значения s во все процессы. Перед циклом 4 неявно запрограммирована синхронизация всех процессов, которая может привести к их простою. Предположим, что главным процессом у нас является третий. Тогда первый процесс после завершения вычисления x(1) (на это у него уйдет 1 секунда) перейдет в режим ожидания значения **s**: 3 с для завершения вычисления x(4) плюс время обмена элементами массива плюс время вычисления суммы третьим процессом и, наконец, плюс время получения **s.**

Важный вывод из сказанного выше - программист должен распределить вычислительную работу как можно более равномерно между всеми процессами.

Средства распараллеливания в трансляторах и параллельные библиотеки. Так называемые "высокопроизводительные ФОРТРАН и Си" (*high-performance FORTRAN and C - HPF and HPC*) являются новыми стандартами на компиляторы для параллельных суперкомпьютеров. Эти языки полностью совместимы с "обычными" ФОРТРАН-77 и Си/Си++. Обычная программа может быть без каких-либо изменений оттранслирована для супер-ЭВМ и исполнена на любом числе процессоров. Однако такой простейший подход приведет к тому, что каждый из процессов на супер-ЭВМ будет полностью от начала и до конца исполнять всю программу без какого-либо реального распараллеливания.

Для распараллеливания программы с помощью HPF или HPC надо вставлять специальные комментарии (*прагмы*), не влияющие на смысл программы, но указывающие транслятору как разместить данные (наиболее важно для массивов) и как распараллелить циклы по обработке этих массивов. При трансляции на других машинах эти прагмы не будут восприниматься трансляторами и как-либо влиять на результирующий машинный код. Программы являются переносимыми на обычные скалярные ЭВМ.

HPF или HPC реализуют концепцию параллелизма данных. Приведем здесь простейший пример прагм на диалекте MPP Fortran для системы Cray-T3D:

```
с описания:
        real x(1024)
CDIR$ SHARED X(:BLOCK)
с действия:
CDIR$ DOSHARED (I) ON X(I)
      do i=1,1024
        x(i) = func(i)
      enddo
```

Первая прагма (комментарий, начинающиеся с символов "CDIR$" в первой колонке) в разделе описаний указывает компилятору, что элементы массива x должны быть распределены между процессами. Вторая прагма указывает, что действия по выполнению цикла должны быть распределены между процессами так, как были распределены элементы массива. Т.е. каждый процесс будет обрабатывать только свои локальные элементы массива. В системе Cray-T3D каждый процесс (=процессор) может обращаться к любым элементам распределенных (shared) массивов, но обращение к элементам, хранящимся в памяти самого процессора, очень эффективно (как к любым своим локальным переменным), а обращение к элементам, хранящимся в памяти других процессоров, требует заметного времени. Поэтому все циклы по обработке распределенных иассивов должны быть аналогичным образом распределены между процессорами.

При написании прагм программист может и не знать, что в машинный код попадают дополнительные команды, направленные на распараллеливание программы, пересылку

данных и синхронизацию процессов. Программист пишет параллельную программу почти так же, как обычную последовательную. Он может подразумевать, что везде выполняется один процесс, но только в некоторых циклах этот процесс будет выполнять меньше работы, чем последовательная версия.

Принципиально другой подход к распределению данных и работы между процессами - использование специальных распараллеливающих библиотек. При использовании библиотек программист может реализовать любую (или сразу обе!) концепцию параллельного программирования - распределение данных или распределение действий. Все переменные являются локальными и программа (процесс) не имеет доступ к переменным других процессов. Программист должен явно писать обращения к подпрограммам из библиотеки для передачи и приема данных, синхронизации, распределения вычислительной работы. В то же время явное использование вызовов подпрограмм позволяет оптимально и более гибко писать программу.

Библиотеки распараллеливающих подпрограмм (например **MPI** или **PVM**) являются переносимыми и позволяют использовать в качестве "супер-ЭВМ" даже кластеры вычислительных модулей, соединенных компьютерной сетью. Однако выбор между распараллеливанием с помощью транслятора (проще написать или адаптировать программу, но есть вероятность, что у других параллельных машин будет другой диалект языка) или библиотеки (более быстродействующие программы, переносимость между всеми супер-ЭВМ, на которых есть данные библиотеки, но программы труднее писать) надо делать исходя из конкретных задач и имеющихся (в наличии или в перспективе) супер-ЭВМ.

6. Заключение

Проблема построения вычислительной системы с высоким (превосходящим серийные образцы компьютеров на несколько порядков) уровнем производительности в настоящее время является актуальной и, как свидетельствует вся история развития вычислительной техники, будет оставаться таковой всегда. Это обстоятельство обусловлено, во-первых, наличием практически важных научных, управленческих и хозяйственных задач, асимптотическая временная и емкостная сложность которых растет существенно быстрее, чем линейно. Во-вторых, при любом уровне развития вычислительной техники существуют такие задачи, которые возможно решить лишь до определенного и всегда недостаточного размера. В-третьих, указанная актуальность обусловлена наличием информационных противоборств, которые приводят к постоянно растущим потребностям в области производительности вычислительных средств.

В настоящее время имеет место тенденция роста интереса к вычислительным средствам высокой производительности. Наряду с проблемными областями, традиционно считавшимися потребителями таких средств (ядерная физика, гидроаэродинамика, криптография, метеорология), заказчиками мощных вычислительных систем становятся и учреждения, работающие в совсем других областях (генетика, биология, прикладная математика, математическая физика, управление различными процессами и многих других, необязательно научных). Следует отметить, что в основном, суперкомпьютеры да и другие вычислительные средства, обладающие высокой вычислительной мощностью, до самого недавнего времени создавались по индивидуальным заказам научных или военных организаций при полном финансировании из государственного бюджета (в США - программа ASCI). Расширение сферы коммерчески оправданных применений суперкомпьютеров (в частности в области Интернет-технологий), увеличившее спрос на такую технику, по всей вероятности, приведет к технологическим прорывам в данной области, сделает более доступными как сами средства, так и их компоненты.

До сегодняшнего дня эффективность самых быстрых компьютеров возросла почти по экспоненте. Первые компьютеры выполнили несколько десятков операций с плавающей запятой в секунду, а производительность параллельных компьютеров начала XXI века достигает сотен и даже тысяч миллиардов операций в секунду, и, скорее всего, этот рост будет продолжаться. Однако архитектура вычислительных систем, определяющих этот рост, изменилась радикально - от последовательной до массивно-параллельной.

Эффективность компьютера зависит непосредственно от времени, требуемого для выполнения базовой операции и числа базовых операции, которые могут быть выполнены одновременно. Время выполнения базовой операции ограничено временем выполнения внутренней элементарной операции процессора (тактом процессора). Уменьшение такта ограничено физическими пределами, такими, например, как скорость света. Чтобы обойти эти ограничения, производители процессоров пытаются реализовать параллельную работу внутри чипа - при выполнении элементарных и базовых операций. Однако теоретически было показано, что стратегия Сверхвысокого Уровня Интеграции (*Very Large Scale Integration - VLSI*) является дорогостоящей, что время выполнения вычислений сильно зависит от размера микросхемы. Поэтому наряду с VLSI для повышения производительности компьютера повсеместно используются и другие (давно известные в области высокопроизводительных вычислений!) способы: конвейерная обработка (различные стадии отдельных команд выполняется одновременно), многофункциональная обработка (отдельные функциональные модули - умножители, сумматоры, и т.д.), параллельная обработка (целочисленное исполнительное устройство и устройство обработки с плавающей точкой).

Все чаще и чаще в ЭВМ включается несколько "компьютеров" и соответствующая логика их соединения. Успехи VLSI-технологии в уменьшении размеров компонент ком-

пьютера, облегчают создание таких ЭВМ. Кроме того, увеличение интеграции компонент позволяет увеличить число "процессоров" в ЭВМ при не очень значительном повышении стоимости.

Другая важная тенденция развития вычислений - это огромное увеличение производительности коммутационных сетей. Если еще недавно сети имели быстродействие в 1.5 Мбит/с, то в конце девяностых, сети с быстродействием в 1000 Мбит/с используются повсюду. Наряду с увеличением быстродействия коммутации увеличивается надежность передачи данных. Это позволяет разрабатывать приложения, которые используют физически распределенные ресурсы, как будто они являются частями одного многопроцессорного компьютера. Например, коллективное использование удаленных баз данных, обработка графических данных на одном или нескольких графических компьютерах, вывод и управление в реальном масштабе времени на рабочих станциях.

Рассмотренные тенденции развития архитектуры вычислительных систем и использования современных компьютеров и коммутационного оборудования позволяют предположить, что в скором будущем параллельность не будет уделом только суперкомпьютеров, она проникнет и на рынок рабочих станций, персональных компьютеров и сетей ЭВМ. Программы будут использовать не только множество процессоров компьютера, но и процессоры, доступные по сети. Поскольку большинство существующих алгоритмов предполагают использование одного процессора, это потребуются новые алгоритмы и программы способные выполнять много операций одновременно. Наличие и использование ***параллелизма*** будет становиться основным требованием при разработке не только новых вычислительных систем, но и алгоритмов и программ.

Библиографический список

1. СуперЭВМ. Аппаратная и программная реализация/ Под редакцией С.Фернбаха: Пер. с англ.- М.: Радио и связь, 1991. –320с.: ил.
2. Смирнов А.Д. Архитектура вычислительных систем: Учеб. пособие для вузов.- М.: Наука, 1990. –320с.: ил.
3. Вычислительные комплексы системы и сети/ А.М. Ларионов, С.А. Майоров, Г.И. Новиков: Учебник для вузов. -Л.: Энергоатомиздат, Ленинградское отд-ние, 1987. –288с.
4. Электронные вычислительные машины и системы: Учеб. пособие для вузов.- М.: Энергоатомиздат, 1991. –592с.: ил.
5. Систолические структуры: Пер. с англ./ Под ред. У. Мура, Э. Маккейба, Р. Уркхарта.- М.: Радио и связь, 1993. –416.: ил.
6. Компьютеры на СБИС: Кн.1: Пер. с япон./ Мотоока Т., Томита С., Танака Х. И др.- М.: Мир, 1988. -392с.: ил.
7. Компьютеры на СБИС: Кн.2: Пер. с япон./ Мотоока Т., Хорикоси Х., Сакаути М. и др.- М.: Мир, 1988. -336с.: ил.
8. Симонс Дж. ЭВМ пятого поколения: компьютеры 90-х годов. - М.: Финансы и статистика, 1985. -173с.: ил.
9. Барский А.Б. Параллельные процессы в вычислительных системах. Планирование и организация.- М.: Радио и связь, 1990. -256с.: ил.
10. Амамия М., Танака Ю. Архитектура ЭВМ и искусственный интеллект / Пер. с японск.- М.: Мир, 1993. -568с.: ил.
11. Транспьютеры. Архитектура и программное обеспечение: Пер. с англ./ Под ред. Г.Харпа.- М.: Радио и связь, 1993. -304с.: ил.