15.1 Уровни параллелизма

В основе архитектуры большинства современных ВМ лежит представление алгоритма решения задачи в виде программы последовательных вычислений. Базовые архитектурные идеи ВМ, ориентированной на последовательное исполнение команд программы, были сформулированы Джоном фон Нейманом. В условиях постоянно возрастающих требований к производительности вычислительной техники все очевидней становятся ограничения классической фон-неймановской архитектуры, обусловленные исчерпанием всех основных идей ускорения последовательного счета. Дальнейшее развитие вычислительной техники связано с переходом к параллельным вычислениям как в рамках одной ВМ, так и путем создания многопроцессорных систем и сетей, объединяющих большое количество отдельных процессоров или отдельных вычислительных машин. Для такого подхода вместо термина «вычислительная машина» более подходит термин «вычислительная система» (ВС). Отличительной особенностью вычислительных систем является наличие в них средств, реализующих параллельную обработку, за счет построения параллельных ветвей в вычислениях, что не предусматривалось классической структурой ВМ. Идея параллелизма как средства увеличения быстродействия ВМ возникла очень давно — еще в XIX веке.

Методы и средства реализации параллелизма зависят от того, на каком уровне он Должен обеспечиваться. Обычно различают следующие *уровни параллелизма:*

■ **Уровень заданий**. Несколько независимых заданий одновременно выполняются на разных процессорах, практически не взаимодействуя друг с другом. Этот уровень реализуется на ВС с множеством процессоров в многозадачном режиме.

■ **Уровень программ**. Части одной задачи выполняются на множестве процессоров. Данный уровень достигается на параллельных ВС.

■ **Уровень команд**. Выполнение команды разделяется на фазы, а фазы нескольких последовательных команд могут быть перекрыты за счет конвейеризации. Уровень достижим на ВС с одним процессором.

■ **Уровень битов** (арифметический уровень). Биты слова обрабатываются один за другим, это называется *бит-последовательной операцией.* Если биты слова обрабатываются одновременно, говорят о *бит-параллельной операции.*

К понятию уровня параллелизма примыкает понятие *гранулярности.* Это мера отношения объема вычислении, выполненных в параллельной задаче, к объему коммуникаций (для обмена сообщениями). Степень гранулярности варьируется от (fine grained) мелкозернистой до крупнозернистой (coarse grained).

Крупнозернистый параллелизм*:* каждое параллельное вычисление достаточно независимо от остальных, причем требуется относительно редкий

обмен информацией между отдельными вычислениями. Единицами распараллеливания являются большие и независимые программы, включающие тысячи команд. Этот уровень параллелизма обеспечивается операционной системой.

Среднезернистый параллелизм: единицами распараллеливания являются вызываемые процедуры, включающие в себя сотни команд. Обычно организуется как программистом, так и компилятором.

Мелкозернистый параллелизм: каждое параллельное вычисление достаточно мало и элементарно, составляется из десятков команд. Обычно распараллеливаемыми единицами являются элементы выражения или отдельные итерации цикла, имеющие небольшие зависимости по данным. Сам термин «мелкозернистый параллелизм» говорит о простоте и быстроте любого вычислительного действия. Характерная особенность мелкозернистого параллелизма заключается в приблизительном равенстве интенсивности вычислений и обмена данными. Этот уровень параллелизма часто используется распараллеливающим (векторизирующим) компилятором.

Эффективное параллельное исполнение требует баланса между степенью гранулярности программ и величиной коммуникационной задержки, возникающей между разными гранулами. В частности, если коммуникационная задержка минимальна, то наилучшую производительность обещает мелкоструктурное разбиение программы. Это тот случай, когда действует параллелизм данных. Если коммуникационная задержка велика (как в слабосвязанных системах), предпочтительней крупнозернистое разбиение программ.

Параллелизм уровня задания возможен между независимыми заданиями или их фазами. Основным средством реализации параллелизма на уровне заданий служат многопроцессорные и многомашинные вычислительные системы, в которых задания распределяются по отдельным процессорам или машинам. Однако, если трактовать каждое задание как совокупность независимых задач, реализация данного уровня возможна и в рамках однопроцессорной ВС. В этом случае несколько заданий могут одновременно находиться в основной памяти ВС, при условии, что в каждый момент выполняется только одно из них. Когда выполняемое задание требует ввода/вывода, такая операция запускается, а до ее завершения остальные ресурсы ВС передаются другому заданию. По завершении ввода/вывода ресурсы ВС возвращаются заданию, инициировавшему эту операцию. Здесь параллелизм обеспечивается за счет того, что центральный процессор и система ввода/вывода работают одновременно, обслуживая разные задания.

О ***параллелизме на уровне программы*** говорят в двух случаях. Во-первых, когда в программе могут быть выделены независимые участки, которые допустимо выполнять параллельно.

Второй тип параллелизма программ возможен в пределах отдельного программного цикла, если в нем отдельные итерации не зависят друг от друга. Программный параллелизм может быть реализован за счет большого

количества процессоров или множества функциональных блоков.

Общая форма параллелизма на уровне программ проистекает из разбиения программируемых, данных на подмножества. Это разделение называют *декомпозицией области* (domain decomposition), а параллелизм, возникающий при этом, носит название *параллелизма данных.* Подмножества Данных назначаются разным вычислительным процессам, и называется этот процесс *распределением данных* (data distribution). Процессоры выделяются определенным процессам либо по инициативе программы, либо в процессе работы операционной системой. На каждом процессоре может выполняться более чем один процесс.

Параллелизм на уровне команд имеет место, когда обработка нескольких команд или выполнение различных этапов одной и той же команды может перекрываться во времени. Разработчики вычислительной техники издавна прибегали к методам, известным под общим названием «совмещения операций», при котором аппаратура ВМ в любой момент времени выполняет одновременно более одной операции. Этот общий принцип включаете себя два понятия: *параллелизм* и *конвейеризацию.* Хотя у них много общего и их зачастую трудно различать на практике, термины эти отражают два принципиально различных подхода.

В первом варианте совмещение операций достигается за счет того, что в составе вычислительной системы отдельные устройства присутствуют в нескольких копиях. Так, в состав процессора может входить несколько АЛУ, и высокая производительность обеспечивается за счет одновременной работы всех этих АЛУ. Второй подход был рассмотрен в Лекции 7 Конвейерная обработка команд.

15.2 Профиль параллелизма программы

Число процессоров, параллельно выполняющих программу в каждый момент времени *t*, задает степень параллелизма *P(t)* (Degree Of Parallelism). Графическое представление параметра *P(t)* называют профилем параллелизма программы. Изменения числа параллельно работающих процессоров (за время наблюдения) зависят от многих факторов (алгоритма, доступных ресурсов, степени оптимизации, обеспечиваемой компилятором и т. д.). Пример профиля параллелизма для некоторого алгоритма показан на рис. 15.2.1.

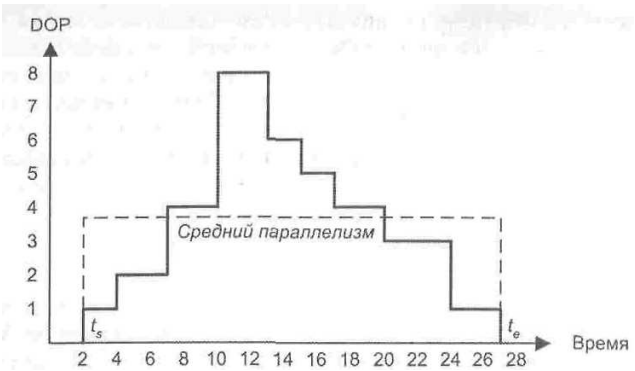

Рис. 15.2.1. Профиль параллелизма

Если, система состоит из n процессоров, а производительность Δ одиночного процессора системы выразим как количество операций в единицу (квант) времени, не учитывая издержек, связанных с обращением к памяти и пересылкой данных. Тогда если за наблюдаемый период (определенное количество квантов времени) загружены i процессоров, то $P = i$. Общий объем вычислений $O(n)$ за период от стартового момента $t_н$ до момента завершения $t_к$ пропорционален площади под кривой профиля параллелизма:

$$O(n) = \Delta \sum_{i=1}^{n} it_i,$$

где t_i — интервал времени (общее количество квантов времени), в течение которого $P = i$, а

$$P = i, \text{ а } \sum_{i=1}^{n} t_i = t_к - t_н$$ — общее время вычислений.

Средний параллелизм A определяется как:

$$A = \frac{\sum_{i=1}^{n} it_i}{\sum_{i=1}^{n} t_i}.$$

15.3 Метрики параллельных вычислений

В силу особенностей параллельных вычислений для оценки их эффективности используют специфическую систему метрик.

Метрики параллельных вычислений — это система показателей, позволяющая оценить преимущества, получаемые при параллельном решении задачи на n процессорах, по сравнению с последовательным решением той же задачи на единственном процессоре. Под параллельными вычислениями будем понимать последовательность шагов, где каждый шаг состоит из i операций, выполняемых одновременно набором из i параллельно работающих процессоров.

Для определения метрик будем считать, что: n — количество процессоров, используемых для организации параллельных вычислений;

$O(n)$ — объем вычислений, выраженный через количество операций, выполняемых n процессорами в ходе решения задачи;

$T(n)$ — общее время вычислений (решения задачи) с использованием n процессоров.

Пусть время изменяется дискретно, а за один квант времени процессор выполняет любую операцию. Тогда справедливо следующие: $T(1) = O(1)$, $T(n) \le O(n)$. Это соотношение формулирует утверждение: ***время вычислений можно сократить за счет распределения объема вычислений по нескольким процессорам***.

По существу, можно выделить четыре группы метрик.

Первая группа характеризует скорость вычислений. Эта группа

представлена парой метрик — индексом параллелизма и ускорением.

Индекс параллелизма (Parallel Index) характеризует среднюю скорость параллельных вычислений через количество выполненных операций:

$$PI(n) = \frac{O(n)}{T(n)}.$$

Ускорение (Speedup) за счет параллельного выполнения программы служит показателем эффективной скорости вычислений. Вычисляется ускорение как отношение времени, затрачиваемого на проведение вычислений на однопроцессорной ВС (в варианте наилучшего последовательного алгоритма), ко времени решения той же задачи на параллельной *n*-процессорной системе (при использовании наилучшего параллельного алгоритма):

$$S(n) = \frac{T(1)}{T(n)}.$$

Вторую группу образуют метрики эффективность и утилизация, дающие возможность судить об эффективности привлечения к решению задачи дополнительных процессоров.

Эффективность (Efficiency) характеризует целесообразность наращивания числа процессоров через ту долю ускорения, достигнутого за счет параллельных вычислений, которая приходится на один процессор:

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}.$$

Утилизация (Utilization) учитывает вклад каждого процессора при параллельном вычислении, но в виде количества операций, выполненных процессором в единицу времени

$$U(n) = \frac{O(n)}{nT(n)}.$$

Третья группа метрик, — избыточность и сжатие, — характеризует эффективность параллельных вычислений путем сравнения объема вычислений, выполненного при параллельном и последовательном решении задачи.

Избыточность (Redundancy) — это отношение объема параллельных вычислений к объему эквивалентных последовательных вычислений:

$$R(n) = \frac{O(n)}{O(1)}.$$

Важность данной метрики в том, что она исходит не из относительных показателей ускорения и эффективности, полученных из времени вычислений, а из абсолютных показателей, базирующихся на объеме выполненной вычислительной работы.

Избыточность отражает степень соответствия между программным и аппаратным параллелизмом.

Сжатие (Compression) вычисляется как величина, обратная избыточности:

$$C(n) = \frac{O(1)}{O(n)}.$$

Четвертую группу образует единственная метрика — качество, объединяющая три рассмотренных группы метрик.

Качество (Quality) определяется как:

$$Q(n) = \frac{T^3(1)}{nT^2(n)O(n)} = S(n)E(n)C(n).$$

Эта метрика увязывает метрики ускорения, эффективности и сжатия, она является более объективным показателем улучшения производительности за счет параллельных вычислений.

15.4 Законы параллельных вычислений

Приобретая для решения своей задачи параллельную вычислительную систему, пользователь рассчитывает на значительное повышение скорости вычислений за счет распределения вычислительной нагрузки по множеству параллельно работающих процессоров. В идеальном случае система из n процессоров могла бы ускорить вычисления в *n* раз. В реальности достичь такого показателя не удается из-за невозможности полного распараллеливания ни одной из задач. Распараллелена может быть лишь оставшаяся часть программы. Обозначим долю операций, которые должны выполняться последовательно одним из процессоров, через *f*, где $0 \le f \le 1$ (здесь доля понимается не по числу строк кода, а по числу реально выполняемых операций). Доля, приходящаяся на распараллеливаемую часть программы, составит $1 - f$. Крайние случаи в значениях *f* соответствуют полностью распараллеливаемым $(f = 0)$ и полностью последовательным $(f = 1)$ программам. Данную ситуацию иллюстрирует рис. 15.4.1, в котором использованы следующие обозначения:

t_s — время обработки последовательной части программы с использованием одного процессора;

$t_p(1)$ — время обработки распараллеливаемой части программы с использованием одного процессора;

$t_p(n)$ — время обработки распараллеливаемой части программы с использованием *n* процессоров.

Рис. 15.4.1. Иллюстрация распараллеливания задачи в *n*-процессорной вычислительной системе

Для распараллеливаемой части задачи идеально, когда параллельные ветви программы постоянно загружают все процессоры системы, причем так, чтобы нагрузка на каждый процессор была одинакова. К сожалению, оба этих условия на практике трудно реализуемы. Таким образом, ориентируясь на параллельную ВС, необходимо четко сознавать, что добиться увеличения производительности прямо пропорционального числу процессоров не удастся, и, естественно, встает вопрос о том, на какое реальное ускорение можно рассчитывать. Ответ зависит от того, каким образом пользователь собирается использовать вычислительные мощности ВС, возросшие в результате увеличения числа процессоров. Наиболее характерными являются три варианта:

1. Объем вычислений не изменяется, а главная цель — сократить время вычислений. Достижимое в этом случае ускорение определяется **законом Амдала**.
2. Время вычислений с расширением системы не меняется, но при этом увеличивается объем решаемой задачи, т.е. выполнить максимальный объем вычислений. Эту ситуацию характеризует **закон Густафсона-Бариса**.
3. Данный вариант похож на предыдущий, но с одним условием: увеличение объема решаемой задачи ограничено емкостью доступной памяти. Ускорение в такой формулировке определяет **закон Сана и Ная**.

15.5 Закон Амдала

Проблема ускорения производимых вычислений рассматривалась Джином Амдалом исходя из положения, что объем решаемой задачи (рабочая нагрузка — число выполняемых операций) с изменением числа процессоров, участвующих в ее решении, остается неизменным. С учетом этого объем закон Амдала может быть представлен следующим выражением:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{fT(1) + \dfrac{(1-f)T(1)}{n}} = \frac{1}{f + \dfrac{1-f}{n}}.$$

При безграничном увеличении числа процессоров имеем:

$$\lim_{n\to\infty} S = \frac{1}{f}.$$

Это означает, что если в программе 25% последовательных операций (то есть $f = 0{,}25$), то сколько бы процессоров не использовалось, ускорения работы программы более чем в четыре раза никак не получить, да и то и 4 — это теоретическая верхняя оценка самого лучшего случая, когда никаких других негативных факторов нет.

15.6 Закон Густафсона-Бариса

Джон Густафсон исходил из другого предположения. Он считал, что пользователь, получая в свое распоряжение более мощную систему, как правило не стремится сократить время вычислений, а, сохраняя его практически неизменным, старается пропорционально возросшей мощности ВС увеличить объем решаемой задачи. И тут оказывается, что наращивание общего объема программы касается главным образом распараллеливаемой части программы. Это ведет к сокращению значения *f*. Примером может служить решение дифференциального уравнения в частных производных. Если доля последовательного кода составляет 10% для 1000 узловых точек, то для 100 000 точек доля последовательного кода снизится до 0,1%.

Таким образом, повышение ускорения обусловлено тем, что, оставаясь практически неизменной, последовательная часть в общем объеме увеличенной программы имеет уже меньший удельный вес. Чтобы оценить степень ускорения вычислений, когда объем последних увеличивается с ростом количества процессоров в системе (при постоянстве общего времени вычислений), Густафсон рекомендует использовать выражение, предложенное Е. Барсисом (Ed Barsis):

$$S(n) = f + (1 - f)n.$$

Данное выражение известно как **закон масштабируемого ускорения** или закон Густафсона–Барсиса. Выражение констатирует, что ускорение является линейной функцией числа процессоров, если рабочая нагрузка масштабируется так, чтобы поддерживать неизменным время вычислений.

В заключение еще раз отметим, что закон Густафсона-Барсиса не противоречит закону Амдала. Различие состоит в форме использования дополнительной мощности ВС, возникающей вследствие увеличения числа процессоров.

15.7 Закон Сана–Ная

В многопроцессорной параллельной ВС каждый процессор обычно имеет независимую локальную память сравнительно небольшой емкости. Общая память ВС образуется объединением локальной памяти каждого процессора ВС. При решении задача разделяется на подзадачи и распределяется по множеству процессоров. Подзадача размещается в

локальной памяти процессора. Как и в постановке Густафсона, увеличение числа процессоров сопровождается возрастанием размера решаемой задачи, но до предела, обусловленного емкостью доступной памяти. Иными словами, объем задачи увеличивается так, чтобы каждая подзадача полностью занимала локальную память процессора. Такая постановка лежит в основе закона, сформулированного Ксиан-Хе Саном (Xian-He Sun) и Лайонелом Наем (Lionel M. Ni), и носит название **закона ускорения, ограниченного памятью**.

Пусть M — это емкость локальной памяти одного процессора. В этом случае суммарная память n-процессорной системы будет равна nM. В формулировке проблемы с ограничением, обусловленным памятью, предполагается, что память каждого процессора задействована полностью, а рабочая нагрузка на один процессор равна $O(1)$, где $O(1) = fO(1) (1 - f)O(1)$. Положим, что при использовании всех n процессоров распараллеливаемая часть задачи может масштабироваться в $G(n)$ раз. В этом случае масштабируемая рабочая нагрузка может быть описана выражением:

$$S(n) = \frac{f + (1-f)G(n)}{f + (1-f)\dfrac{G(n)}{n}}.$$

Закон Сана–Ная представляет собой обобщение законов Амдала и Густафсона- Барсиса.

При $G(n) = 1$ размер задачи фиксирован, что соответствует постановке Амдала. В результате получаем формулу Амдала:

$$S(n) = \frac{f + (1-f)1}{f + (1-f)\dfrac{1}{n}} = \frac{1}{f + \dfrac{1-f}{n}}.$$

При $G(n) = n$ соответствует случаю, когда с увеличением емкости памяти в n раз рабочая нагрузка также возрастает в n раз. Это идентично постановке Густафсона- Барсиса:

$$S(n) = \frac{f + (1-f)n}{f + (1-f)} = f + (1-f)n$$

В случае когда вычислительная нагрузка возрастает быстрее, чем требования к памяти ($G(n) > n$), модель с ограничением по памяти дает более оптимистичную оценку ускорения.

Рисунок 15.7.1 иллюстрирует оценки ускорения, получаемые по каждой из трех рассмотренных моделей ускорения.

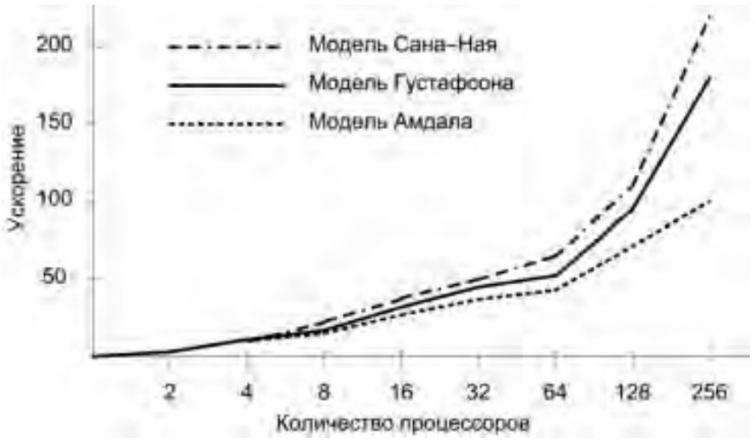

Рис. 15.7.1 Сравнение трех моделей ускорения

15.8 Классификация параллельных вычислительных систем

Даже краткое перечисление типов современных параллельных вычислительных систем (ВС) дает понять, что для ориентирования в этом многообразии необходима четкая система классификации. От ответа на главный вопрос — что заложить в основу классификации — зависит, насколько конкретная система классификации помогает разобраться с тем, что представляет собой архитектура ВС и насколько успешно данная архитектура позволяет решать определенный круг задач. Попытки систематизировать вес множество архитектур параллельных вычислительных систем предпринимались достаточно давно и длятся по сей день, но к однозначным выводам пока не привели.

Среди множества систем классификации ВС наибольшее распространение получила классификация, предложенная в 1966 году М. Флинном. В ее основу положено понятие потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. В зависимости от количества потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD, MISD, STMD, MIMD.

SISD (Single Instruction Stream/Single Data Stream) — одиночный ноток команд и одиночный поток данных (рис. 15.8.1, *а)*. Представителями этого класса являются, прежде всего, классические фон-неймановские ВМ, где имеется только один поток команд, команды обрабатываются последовательно и каждая команда инициирует одну операцию с одним потоком данных. То, что для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка, не имеет значения, поэтому в класс SISD одновременно попадают как ВМ CDC 6000 со скалярными функциональными устройствами, так и CDC 7600 с конвейерными. Некоторые специалисты считают, что к SISD-системам можно причислить и векторно-конвейерные ВС, если рассматривать вектор как неделимый элемент данных для соответствующей команды.

MISD (Multiple Instruction Stream/Single Data Stream) - множественный

поток команд и одиночный поток данных (рис. 15.8.1, *б).* Из определения следует, что в архитектуре ВС присутствует множество процессоров, обрабатывающих один и тот же поток данных. Примером могла бы служить ВС, на процессоры которой подается искаженный сигнал, а каждый из процессоров обрабатывает этот сигнал с помощью своего алгоритма фильтрации. Тем не менее ни Флинн, ни другие специалисты в области архитектуры компьютеров до сих пор не сумели представить убедительный пример реально существующей вычислительной системы, построенной на этом принципе. Поэтому принято считать, что пока данный класс пуст.

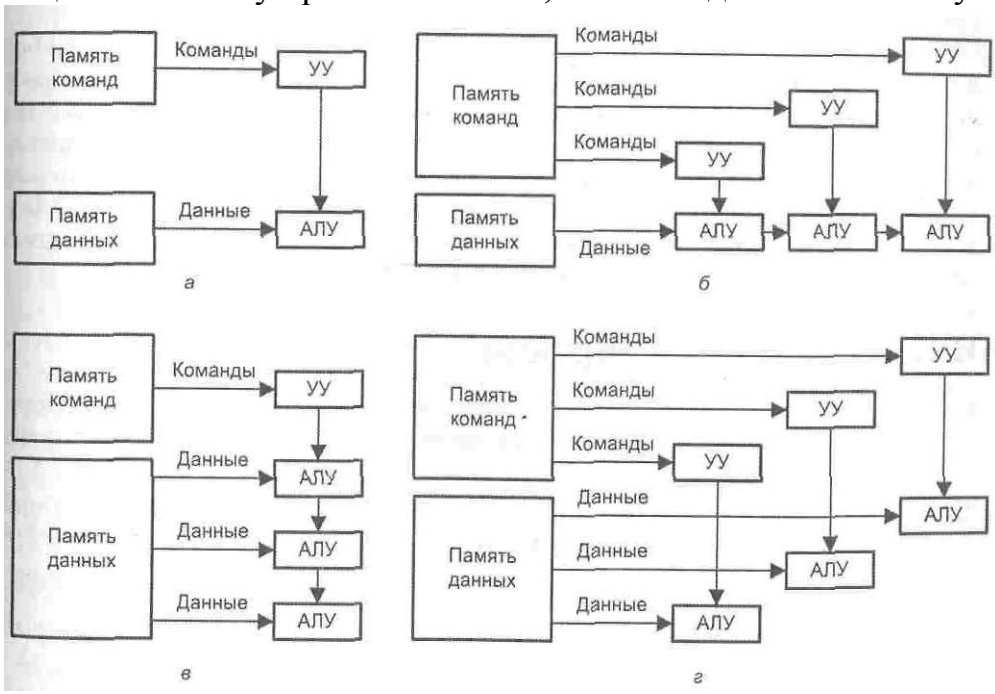

Рис. 15.8.1 Архитектура вычислительных систем по Флинну:
а — SISD; *б* — MISD; в — SIMD; г— MIMD

SIMD (Single Instruction Stream/Multiple Data Stream) - одиночный поток команд и множественный поток данных (рис. 10,5, *в).* ВМ данной архитектуры позволяют выполнять одну арифметическую операцию сразу над многими данными — элементами вектора. Бесспорными представителями класса SIMD считаются матрицы процессоров, где единое управляющее устройство контролирует множество процессорных элементов. Все процессорные элементы получают от устройства управления одинаковую команду и выполняют ее над своими локальными данными В принципе в этот класс можно включить и векторно-конвейерные ВС, если каждый элемент вектора рассматривать как отдельный элемент потока данных.

MIMD (Multiple Instruction Stream/Multiple Data Stream) — множественный поток команд в множественный поток данных (рис. 15.8.1, г*).* Класс предполагает наличие в вычислительной системе множества устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных. Класс MIMD чрезвычайно широк, поскольку включает в себя всевозможные мультипроцессорные системы. Кроме того, приобщение к классу MIMD зависит от трактовки. Так, ранее упоминавшиеся векторно-конвейерные ВС

можно вполне отнести й к классу MIMD, если конвейерную обработку рассматривать как выполнение множества команд (операций ступеней конвейера) над множественным скалярным потоком.

Схема классификации Флинна вплоть до настоящего времени является наиболее распространенной при первоначальной оценке той или иной ВС, поскольку позволяет сразу оценить базовый принцип работы системы, чего часто бывает достаточно. Однако у классификации Флинна имеются и очевидные недостатки, например неспособность однозначно отнести некоторые архитектуры к тому или иному классу. Другая слабость — это чрезмерная насыщенность класса MIMD. Все это породило множественные попытки либо модифицировать классификацию Флинна, либо предложить иную систему классификации.