технике решаются задачи. И только хорошее знание структуры задачи и алгоритмов поможет решать задачи эффективно. Для больших задач это особенно важно.

ЛЕКЦИЯ 3

Компьютеры и параллельные формы алгоритмов

Содержание: *абстрактная модель последовательного компьютера, влияние последовательных вычислений, развитие параллелизма в компьютерах, концепция неограниченного параллелизма, граф алгоритма, необходимость новых сведений о структуре алгоритмов, параллельная форма алгоритма, абстрактная модель параллельной системы.*

Несмотря на огромное фактическое разнообразие, все существующие компьютеры и вычислительные системы с точки зрения пользователя условно можно разделить на две большие группы: последовательные и параллельные.

В простейшей интерпретации последовательные компьютеры выглядят следующим образом. Имеются два основных устройства. Одно из них, называемое *процессором* (центральным процессором, решающим устройством, арифметико-логическим устройством и т.п.), предназначено для выполнения некоторого ограниченного набора простых операций. В набор операций обычно входят сложение, вычитание и умножение чисел, логические операции над отдельными разрядами и их последовательностями, операции над символами и многое другое. Наборы операций, выполняемые процессорами разных компьютеров, могут отличаться как частично, так и полностью. Другое устройство, называемое *памятью* (запоминающим устройством и т.п.), предназначено для хранения всей информации, необходимой для организации работы процессора. Процессор является активным устройством, т.е. он имеет возможность преобразовывать информацию. Память является пассивным устройством, т.е. она не имеет такой возможности. Процессор и память связаны между собой *каналами* обмена информацией.

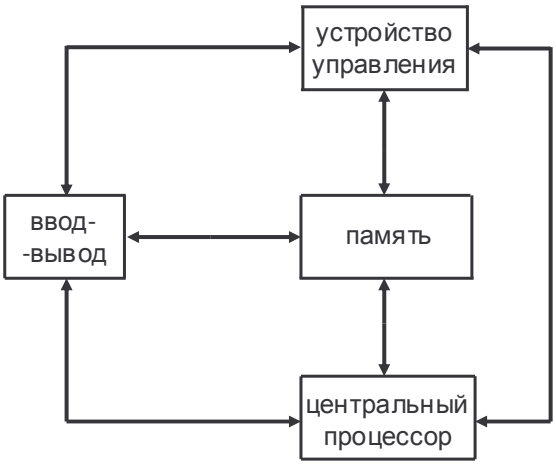

Рис.3.1. Абстрактная модель последовательного компьютера.

Работа однопроцессорного компьютера заключается в последовательном выполнении отдельных команд. Каждая команда содержит информацию о том, какая операция из заданного набора должна быть выполнена, а также из каких ячеек памяти должны быть взяты аргументы операции и куда должен быть помещен результат. Описание упорядоченной последовательности команд в виде *программы* находится в памяти. Там же размещаются необходимые для реализации алгоритма начальные данные и результаты промежуточных вычислений. Координирует работу всех узлов компьютера *устройство управления.* Оно организует последовательную выборку команд из памяти и их расшифровку, передачу из памяти в процессор операндов, а из процессора в память результатов выполнения команд, управляет работой процессора. Ввод начальных данных и выдачу результатов осуществляет *устройство ввода-вывода.*

По такой схеме устроены все однопроцессорные компьютеры. И это справедливо как для первых в истории электронных вычислительных машин – медленно работающих монстров, занимающих огромные помещения и потребляющих чудовищное количество энергии, так и для современных компактных высокоскоростных персональных компьютеров, размещающихся на письменном столе и потребляющих энергию меньше, чем обычная электрическая лампочка. Конечно, в действительности схемы однопроцессорных компьютеров обрастают большим числом дополнительных деталей. Но всегда процессор является *единственным* устройством, выполняющим полезную с точки зрения пользователя работу. В этом смысле у любого компьютера все другие устройства по отношению к процессору

оказываются обслуживающими, и их работа направлена только на то, чтобы обеспечить наиболее эффективный режим функционирования процессора.

Отсюда следует *очень важный вывод*: каким бы сложным ни был однопроцессорный компьютер, построенный по классическим канонам, в основе его архитектуры и организации процесса функционирования всегда лежит *принцип последовательного выполнения отдельных действий*. С точки зрения пользователя отклонения от этого принципа, как правило, не существенны. Именно поэтому такие компьютеры и называются *последовательными*.

На каждом конкретном последовательном компьютере время реализации любого алгоритма пропорционально, главным образом, числу выполняемых операций, и почти не зависит от того, как внутренне устроен сам алгоритм. Конечно, какие-то различия во временах реализаций могут появляться. Но они невелики и в обычной практике их можно не принимать во внимание. Это свойство последовательных компьютеров исключительно важно и влечет за собой разнообразные следствия. Пожалуй, самым главным из них является то, что для таких компьютеров оказалось возможным создавать компьютерно-независимые или, как их называют иначе, *машинно-независимые* языки программирования. По замыслу их создателей любая программа, написанная на любом из таких языков, должна без какой-либо переделки реализовываться на любой последовательной машине. Единственное, что формально требовалось для обеспечения работы программы, - это наличие на машине компилятора с соответствующего языка. Подобные языки стали возникать в большом количестве: Алгол, Кобол, Фортран, Си и др. Многие из них успешно используются до сих пор. Поскольку эти языки ориентированы на последовательные компьютеры, их также стали называть последовательными. Во всех программах, написанных на последовательных языках программирования, порядок выполнения команд всегда является строго *последовательным* и при заданных входных данных фиксируется *однозначно*.

Для математиков и разработчиков прикладного программного обеспечения такая ситуация открывала заманчивую перспективу. Не нужно было вникать в устройство вычислительных машин, так как языки программирования по существу мало чем отличались от языка математических описаний. В разработке вычислительных алгоритмов становились очевидными главные целевые функции их качества – минимизация числа выполняемых операций и устойчивость к влиянию ошибок округления. И больше ничего об алгоритмах не надо было знать, поскольку никаких причин для получения каких-либо других знаний и, тем более, знаний о структуре алгоритмов не возникало.

Привлекательность подобной перспективы на долгие годы сделала последовательную организацию вычислений неявным и во многом даже неосознанным фундаментом развития не только численных методов, но и всей вычислительной математики. Заметим, что по своему влиянию на

общую направленность исследований в вычислительных делах эта перспектива и в настоящее время во многом остается доминирующей.

На самом деле все, что связано с последовательными компьютерами, развивалось и достаточно сложно, и в какой-то степени драматично. Погоня за производительностью и конкуренция привели к тому, что появилось много разных последовательных машин. Для каждой из них приходилось делать свой компилятор, так или иначе учитывающий особенности конкретной машины. Не в каждом компиляторе удавалось оптимально учитывать эти особенности на всем множестве программ. Поэтому в реальности при переносе программ с одной машины на другую многие программы приходилось модифицировать. Объем изменений мог быть большим или малым и зависел от сложности машин и языков программирования. Проблема переноса программ с одного последовательного компьютера на другой последовательный компьютер становилась со временем все более актуальной и были предприняты значительные усилия на ее решение. В первую очередь, за счет введения различных стандартов на конструирование компьютеров и правила написания программ.

Машины, которые принято называть последовательными, можно называть таковыми лишь с некоторой оговоркой, поскольку в каждый момент времени в них независимо или, другими словами, *параллельно* выполняется много различных действий. Именно, реализуются какие-то операции, передаются данные от одного устройства к другому, происходит обращение к памяти и т.п. Весь этот параллелизм учитывается при создании компилятора. Степень учета параллелизма компилятором прямым образом сказывается на эффективности работы программ. Однако если параллелизм не виден через язык программирования, то для пользователя он как бы и не существует. Поэтому с точки зрения пользователя можно считать последовательными любые машины, эффективное общение с которыми осуществляется на уровне последовательных языков программирования. Подобная трактовка удобна для пользователей. Но есть в ней и серьезная опасность. Уповая на долговременную перспективу общения с вычислительной техникой на уровне последовательных языков, можно пропустить момент, когда количественные изменения в технике перейдут в качественные и общение с ней при помощи таких языков окажется невозможным. И тогда вроде бы совсем неожиданно, вдруг возникает вопрос о том, что же делать дальше. Именно это и произошло в истории освоения вычислительной техники.

В развитии вычислительной техники многое определяется стремлением повысить производительность и увеличить объем быстрой памяти. Мощности первых компьютеров были очень малы. Поэтому сразу после их появления стали предприниматься попытки объединения нескольких компьютеров в единую систему. Идея была чрезвычайно проста: если мощности одного компьютера нехватает для решения конкретной задачи, то нужно разделить задачу на две части и решать каждую часть на своем компьютере. А чтобы

было удобно передавать данные с одного компьютера на другой, необходимо соединить сами компьютеры подходящими по пропускной способности *линиями связи*. Так появились двухмашинные комплексы. Естественно, на них можно было решать задачи примерно вдвое быстрее. Аналогичным образом строились многомашинные комплексы, объединяющие три, четыре, пять и более отдельных однопроцессорных компьютеров в единую систему. Соответственно повышалась и мощность комплексов. Больших проблем с разделением исходной задачи на несколько независимых подзадач не возникало, поскольку их общее число было невелико.

Несмотря на плодотворность идеи объединения отдельных машин в единый комплекс, долгое время она не получала необходимое развитие. Основные трудности на пути ее практической реализации были связаны с большими размерами первых компьютеров и большими временны́ми потерями в процессах передачи информации между ними. Как следствие, значительного увеличения мощности добиться не удавалось. Но совершенствовались технологии, уменьшались размеры компьютеров, снижалось их энергопотребление. И через некоторое время стало возможно создавать многомашинный комплекс как единую многопроцессорную вычислительную систему с приемлемыми производственными параметрами.

Количество процессоров в системах увеличивалось постепенно. Вообще говоря, в целях достижения максимальной производительности составлять программы для каждого процессора надо было бы индивидуально. Но для пользователя это и не удобно и не привычно. К тому же, необходимо было обеспечить преемственность использования обычных последовательных программ, которых накопилось в мире уже очень много, на системах с несколькими процессорами. Поэтому решение проблемы адаптации последовательных программ к таким системам взяли на себя компиляторы. Однако постепенно внутреннего параллелизма в системах становилось все больше и больше. И, наконец, его стало столь много, что наработанные технологии компилирования программ оказались не в состоянии образовывать оптимальный машинный код. Для его получения в случае многих процессоров компилятору приходится иметь дело с задачей составления оптимального расписания. Решается она перебором и требует, в общем случае, выполнения экспоненциального объема операций по отношению к числу процессоров. Пока число процессоров было невелико, компиляторы как-то справлялись с такой задачей. Но как только их стало очень много, развитие традиционных технологий компилирования зашло в тупик.

Быстрое развитие элементной базы привело к тому, что уже в начале 60-х годов прошлого столетия стали серийно выпускаться вычислительные системы, в которых насчитывалось порядка десятка процессоров, работающих параллельно. Создателям компиляторов все еще удавалось прикрывать пользователей от такого параллелизма, и язык общения оставался практически последовательным. В конце 70-х годов появились серийные вычислительные

машины векторного типа, в которых ускорение достигалось за счет быстрого выполнения операций над векторами. Уровень внутреннего параллелизма в них был достаточно высок, хотя весьма специального вида. Создатели компиляторов снова попытались сделать язык программирования последовательным. И на этот раз потерпели неудачу. Формально все еще оставалась возможность пользоваться некоторым последовательным языком. Но заложенная в компилятор технология автоматического выявления векторных конструкций из текста программ оказалась не эффективной. Поэтому, если пользователя не устраивала скорость работы откомпилированной программы, ему нужно было просматривать служебную информацию о работе компилятора и на основе ее анализа самому находить узкие места компиляции и вручную перестраивать программу под векторные конструкции. О том, как именно это делать, конструктивных советов не предлагалось. На практике процедуру перестройки программ приходилось делать многократно.

По существу на этом закончился период, когда задачу, полностью описанную на последовательном языке, можно было более или менее эффективно решать на любой вычислительной технике. На этом же стало заканчиваться время создания классических последовательных компьютеров и началась эра параллельных компьютеров и больших вычислительных систем параллельной архитектуры. По сравнению с последовательными компьютерами в них все обстоит иначе. Рядовые их представители имеют десятки процессоров, а самые большие – десятки и даже сотни тысяч. Мощности параллельных компьютеров огромны и теоретически не ограничены. Практические скорости уже сегодня достигают сотен триллионов операций в секунду. Однако все эти преимущества имеют серьезную негативную сторону: в отличие от последовательных компьютеров использовать параллельные чрезвычайно трудно, интерфейс с ними оказывается совсем не дружественным, языки программирования перестали быть универсальными, от пользователя требуется много новой и трудно доступной информации о структуре алгоритмов и т.д.

На вычислительных системах параллельной архитектуры время решения задач *решающим образом* зависит от того, какова внутренняя структура алгоритма и в каком порядке выполняются его операции. Возможность ускоренной реализации алгоритма на параллельных системах достигается за счет того, что в них имеется достаточно большое число процессоров, которые могут *параллельно* выполнять операции алгоритма. Предположим для простоты, что все процессоры имеют одинаковую производительность и работают в синхронном режиме. Тогда общий коэффициент ускорения по сравнению с тем случаем, когда алгоритм реализуется на одном универсальном процессоре такой же производительности, оказывается примерно равным числу операций алгоритма, выполняемых в среднем на всех процессорах в каждый момент времени. Если параллельные вычислительные

системы имеют десятки и сотни тысяч процессоров, то отсюда никак не следует, что при решении конкретных задач всегда можно и, тем более, достаточно легко получить ускорение счета такого же порядка.

На любой вычислительной технике одновременно могут выполняться только *независимые* операции. Это означает следующее. Пусть снова все процессоры имеют одинаковую производительность и работают в синхронном режиме. Допустим, что в какой-то момент времени на каких-то процессорах выполняются какие-то операции алгоритма. Результат ни одной из них не только не может быть аргументом любой из выполняемых операций, но даже не может никаким косвенным образом оказывать влияние на их аргументы. Если рассмотреть процесс реализации алгоритма во времени, то в силу сказанного сам процесс на любой вычислительной системе, последовательной или параллельной, разделяет операции алгоритма на группы. Все операции каждой группы независимы и выполняются одновременно, а сами группы реализуются во времени последовательно одна за другой. Это неявно порождает некоторую специальную форму представления алгоритма, в которой фиксируются как группы операций, так и их последовательность. Называется она *параллельной формой алгоритма* [1]. Ясно, что при наличии в алгоритме ветвлений или условных передач управления его параллельная форма может зависеть от значений входных данных.

Параллельную форму алгоритма можно ввести и как эквивалентный математический объект, не зависящий от вычислительных систем. Зафиксируем входные данные и разделим все операции алгоритма на группы. Назовем их *ярусами* и пусть они обладают следующими свойствами. Во-первых, в каждом ярусе находятся только независимые операции. И, во-вторых, существует такая последовательная нумерация ярусов, что каждая операция из любого яруса использует в качестве аргументов либо результаты выполнения операций из ярусов с меньшими номерами, либо входные данные алгоритма. Ясно, что все операции, находящиеся в ярусе с наименьшим номером, всегда используют в качестве аргументов только входные данные. Будем считать в дальнейшем, что нумерация ярусов всегда осуществляется с помощью натуральных чисел подряд, начиная с 1.

Число операций в ярусе принято называть *шириной* яруса, число ярусов в параллельной форме – *высотой* параллельной формы. Очевидно, что ярусы математической параллельной формы являются не чем иным как теми самыми группами, о которых говорилось выше. При одних и тех же значениях входных данных между математическими параллельными формами алгоритма и реализациями того же алгоритма на конкретных или гипотетических вычислительных системах с одним или несколькими процессорами существует взаимно однозначное соответствие. Если какая-то параллельная форма отражает реализацию алгоритма на некоторой вычислительной системе, то ширина ярусов говорит о числе используемых в каждый момент времени независимых устройств, а высота – о времени реализации алгоритма.

Каждый алгоритм при фиксированных входных данных в общем случае имеет много параллельных форм. Формы, в которых все ярусы имеют ширину, равную 1, существуют всегда. Все они отражают последовательные вычисления и имеют максимально большие высоты, равные числу выполняемых алгоритмом операций. Даже таких параллельных форм алгоритм может иметь несколько, если он допускает различные эквивалентные реализации. Наибольший интерес представляют параллельные формы *минимальной* высоты, так как именно они показывают, насколько быстро может быть реализован алгоритм, по крайней мере, теоретически. По этой причине минимальная высота всех параллельных форм алгоритма называется *высотой алгоритма*. Существует параллельная форма, в которой каждая операция из яруса с номером k, $k>1$, получает в качестве одного из аргументов результат выполнения некоторой операции из $(k-1)$-го яруса. Такая параллельная форма называется *канонической*. Для любого алгоритма при заданных входных данных каноническая форма всегда существует, единственна и имеет минимальную высоту. Кроме этого, в канонической параллельной форме, как и в любой другой форме минимальной высоты, ярусы в среднем имеют максимально возможную ширину [1]. Таким образом, решая любую задачу на любой вычислительной системе с развитым параллелизмом на уровне функциональных устройств, пользователь *неизбежно*, явно или неявно, соприкасается с параллельной формой реализуемого алгоритма. И это происходит даже тогда, когда он ничего не знает обо всех этих понятиях. Если не сам пользователь или разработчик программы, то кто-то другой или что-то другое, например, компилятор, операционная система, какая-нибудь сервисная программа, а скорее всего все они вместе, закладывают в вычислительную систему некоторую программу действий, что и порождает соответствующую им параллельную форму.

А теперь вспомним, что достигаемое ускорение в среднем пропорционально числу операций, выполняемых в каждый момент времени. Если оно равно общему числу имеющихся процессоров, то данный алгоритм при заданных входных данных реализуется на используемой вычислительной системе эффективно. В этом случае возможный ресурс ускорения использован *полностью* и более быстрого счета достичь на выбранной системе *невозможно* ни практически, ни теоретически. Но если ускорение значительно меньше числа устройств?

Предположим, что оно существенно меньше средней ширины ярусов канонической параллельной формы реализуемого алгоритма. Это означает, что *не очень удачно* выбрана схема реализации алгоритма. Изменив ее, можно попытаться более полно использовать имеющийся потенциал параллелизма в алгоритме. Заметим, что в практике общения с параллельными компьютерами именно эта ситуация возникает наиболее часто. Если же ускорение равно средней ширине ярусов канонической параллельной формы, то весь потенциал параллелизма в алгоритме выбран полностью. В данном случае никаким

изменением схемы счета нельзя использовать большее число устройств системы и, следовательно, *нельзя* добиться большего ускорения. И, наконец, вполне возможно, что даже при использовании всех процессоров реальное ускорение не соответствует ожидаемому. Это означает, что при реализации алгоритма приходится осуществлять какие-то передачи данных, требующие длительного времени. Для того чтобы теперь понять причины замедления, необходимо особенно тщательно изучить *структуру* алгоритма и/или вычислительной системы. В практике общения с параллельными компьютерами эта ситуация также возникает достаточно часто.

Таким образом, как только возникает необходимость решать какие-то вопросы, связанные с анализом ускорения при решении задачи на вычислительной системе параллельной архитектуры, так обязательно требуется получить какие-то сведения относительно структуры алгоритма на уровне связей между отдельными операциями. Более того, чаще всего эти сведения приходится сопоставлять со сведениями об архитектуре вычислительной системы. Проведение совместного анализа представляет сложный процесс, но о нем почти ничего не говорится в образовательных курсах. Понятно, почему это происходит. Если об архитектурах вычислительных систем и параллельном программировании рассказывается хотя бы в специальных курсах, то обсуждение структур алгоритмов на уровне отдельных операций в настоящее время не входит ни в какие образовательные дисциплины. И это несмотря на то, что структуры алгоритмов обсуждаются в научной литературе в течение нескольких десятилетий, да и практика использования вычислительной техники параллельной архитектуры насчитывает не намного меньший период.

Отсутствие нужных сведений о структуре алгоритмов не могло остановить развитие собственно вычислительной техники. Стали создаваться новые языки и системы программирования, в огромном количестве и самого различного типа: от расширения последовательных языков параллельными конструкциями до создания на макроуровне параллельных языков типа автокода [1].

Очевидно, что разработчики компиляторов и средств программирования, так же как и конструктора вычислительной техники, не могут сказать что-либо существенное о структуре выполняемых алгоритмов. Но они хорошо понимают, какие множества операций на той или иной технике будут эффективно реализовываться в режиме параллельного счета. Поэтому во всех новых языках и системах программирования стали вводиться конструкции, позволяющие *описывать* такие множества. А вот *ответственность* за поиск в алгоритмах этих множеств и их описание соответствующими конструкциями языка была *возложена на разработчиков программ.* Как следствие, на них же перекладывалась и ответственность за эффективность функционирования создаваемого ими программного продукта. Конечно, в таких условиях даже не шла речь о какой-либо преемственности программ. Главными проблемами пользователей стали нахождение многочисленных и трудно добываемых

характеристик решаемых задач и организация параллельных вычислительных процессов. Вычислительным сообществом это рассматривается как неизбежная плата за возможность быстро решать задачи. Но во всем сообществе расплачиваются, главным образом, пользователи, постоянно переписывая свои программы.

Невнимание со стороны математиков к развитию вычислительной техники привело к серьезному разрыву между имеющимися знаниями в области алгоритмов и теми знаниями, которые были необходимы для быстрого решения задач на новейшей вычислительной технике. Образовавшийся разрыв сказывается до сих пор, и именно он лежит в основе многих трудностей практического освоения современных вычислительных систем параллельной архитектуры.

До сих пор специалистов в области вычислительной математики учили, как решать задачи математически правильно. Теперь надо, к тому же, учить, как решать задачи эффективно на современной вычислительной технике. А это совсем другая наука, математическая по своей сути, но которую пока почти не изучают в вузах.

Для успешного решения задач на вычислительных системах параллельной архитектуры приходится привлекать принципиально новые сведения о структуре алгоритмов на уровне связей отдельных операций между собой. В первую очередь, необходимо знать множества операций, которые можно выполнять независимо. Другими словами, нужны сведения как раз о тех параллельных формах алгоритмов, о которых говорилось выше.

Для большей наглядности проведенных рассуждений всюду выше явно или неявно предполагалось, что все процессоры вычислительной системы работают в синхронном режиме и выполняют любую операцию за одно и то же время. Такое предположение не принципиально по сути дела. Но оно позволяет более отчетливо показать смысл введения параллельных форм как очень важных объектов, описывающих тонкие структурные свойства и характеристики алгоритмов. Оказалось, что эти объекты удобно задавать *ориентированными графами*.

В самом деле, во всех рассуждениях не имело никакого значения, какие именно операции выполняют процессоры. Они могли быть и простейшими машинными операциями, и сколь угодно большими совокупностями операций, оформленными в виде функций, подпрограмм или программных комплексов. Будем считать операции алгоритма вершинами графа. На множестве вершин-операций естественным образом определен частичный порядок, показывающий, какие операции вслед за какими могут выполняться. Этот же порядок позволяет задать дуги графа. Если операция, соответствующая вершине B, использует в качестве хотя бы одного своего аргумента результат выполнения операции, соответствующей вершине A, то проведем дугу из вершины A в вершину B. В противном случае дуга между вершинами A и B не проводится. Проделаем то же самое для каждой пары

вершин. Тем самым будет построен ориентированный ациклический граф. Он называется *графом алгоритма*.

Граф алгоритма играет исключительно важную роль в изучении параллельных свойств самого алгоритма. Его значение определяется двумя факторами. Во-первых, он устроен значительно проще самого алгоритма, так как не связан ни с какой атрибутикой, сопровождающей описание алгоритма. Граф алгоритма можно рассматривать как классический математический объект и ничто не мешает исследовать его чисто математическими методами. И, во-вторых, по графу алгоритма очень просто строить параллельные формы. Действительно, выберем какое-то число не связанных дугами вершин, которые, в свою очередь, не имеют входящих дуг (независимых операций алгоритма, аргументами которых являются только входные данные алгоритма). Будем считать их первым ярусом параллельной формы. Предположим, что уже построено k, $k \geq 1$, ярусов параллельной формы. Выберем любое число не связанных дугами вершин, в которые могут входить дуги только из вершин первых k ярусов (независимых операций алгоритма, аргументами которых являются либо входные данные алгоритма, либо результаты выполнения операций, соответствующих первым k ярусам). Будем считать их (k+1)-ым ярусом параллельной формы. Продолжим процесс до тех пор, пока не будут исчерпаны все вершины графа. В результате всех таких действий будет построена *параллельная форма графа алгоритма*.

Очевидно, что понятия параллельной формы для алгоритма или его графа изоморфны. Следовательно, с точки зрения изучения параллелизма в алгоритмах безразлично, с какими из этих форм иметь дело. В дальнейшем терминологию и свойства, относящиеся к параллельным формам алгоритма, без дополнительных оговорок будем переносить на граф алгоритма. И наоборот.

Как уже отмечалось, граф алгоритма устроен существенно проще, чем сам алгоритм. Поэтому намечается следующая линия действий: сначала находится граф алгоритма, затем изучаются подходящие его параллельные формы и, наконец, на основе проведенных исследований выбирается нужная схема реализации алгоритма на вычислительной системе параллельной архитектуры. Но возникает очень много вопросов, касающихся, в первую очередь того, как строить и изучать графы алгоритмов. Ответы на них будут даны позднее.

Перспектива внедрения в практику параллельных вычислительных систем потребовала разработки математической концепции построения *параллельных алгоритмов*, т.е. алгоритмов, специально приспособленных для реализации на подобных системах. Такая концепция необходима для того, чтобы научиться понимать, как следует конструировать параллельные алгоритмы, что можно ожидать от них в перспективе и какие подводные камни могут встретиться на этом пути. Концепция начала активно развиваться в конце 50-х – начале 60-х годов прошлого столетия и получила название *концепции неограниченного параллелизма.*

Истоки этого названия связаны с выбором для нее абстрактной модели параллельной вычислительной системы. Поскольку концепция разрабатывалась для проведения математических исследований, то в требованиях к модели могло присутствовать самое минимальное число технических параметров. Тем более что в то время о структуре параллельных вычислительных систем и путях их совершенствования было вообще мало что известно. Лишь быстрое развитие элементной базы подсказывало, что число процессоров в системе вскоре может стать очень большим. Но что-нибудь другое спрогнозировать было трудно. Поэтому явно или неявно в модели остались только следующие предположения. Число процессоров может быть сколь угодно большим, все они работают в синхронном режиме и за единицу времени выполняют абсолютно точно любую операцию из заданного множества. Процессоры имеют общую память. Все вспомогательные операции, взаимодействие с памятью, управление компьютером и любые передачи информации осуществляются мгновенно и без конфликтов. Входные данные перед началом вычислений записаны в память. Каждый процессор считывает свои операнды из памяти и после выполнения операции записывает результат в память. После окончания вычислительного процесса все результаты остаются в памяти.

Для математических исследований нет особого смысла усложнять модель параллельной вычислительной системы. Математикам приходится изучать различные алгоритмы, в том числе весьма экзотические, если их рассматривать с позиций возможной реализации на существующей вычислительной технике. Но какой будет вычислительная техника завтра и, тем более, послезавтра – неизвестно. Одна из схем абстрактной модели параллельной вычислительной системы представлена на рис.3.2.

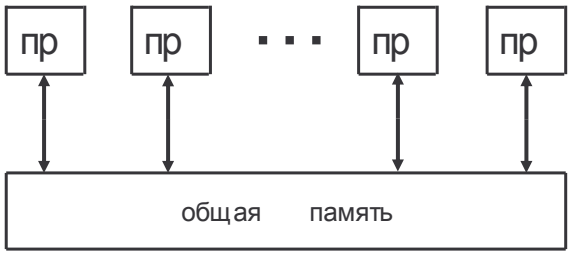

Рис. 3.2. Абстрактная модель параллельной системы.

Несмотря на то, что концепция неограниченного параллелизма предельно проста, она вполне приемлема для первого знакомства с параллельными вычислениями. Более того, именно в силу своей простоты она оказалась очень живучей и до сих пор используется для иллюстрации различных свойств

алгоритмов. Например, на ней легко демонстрируются параллельные формы алгоритмов, показывается зависимость ширины ярусов, числа задействованных процессоров и получаемого ускорения. Если бы концепция неограниченного параллелизма не нужна была ни для чего другого, ее все равно следовало бы создать. Просто из-за того, что параллельные формы являются одним из важнейших инструментов изучения структуры алгоритмов.

ЛЕКЦИЯ 4

Характеристики вычислительных процессов

Содержание: *простое и конвейерное функциональное устройство, загруженность, производительность, ускорение, система устройств, влияние связей между устройствами, законы Амдала и следствия.*

Для того чтобы вычислительная система имела высокую производительность, она должна состоять из большого числа одновременно работающих функциональных устройств (ФУ). Для оценки качества их работы предложено много различных характеристик. Нередко одни и те же по смыслу характеристики вводятся по-разному. И не всегда легко понять, отражают ли эти различия существо дело или же они связаны с какими-то другими причинами. Показательной в этом отношении является такая характеристика как производительность системы. Различных ее определений введено столь много и они столь сильно отличаются друг от друга, что объявляемая производительность вычислительной системы воспринимается лишь как элемент рекламы.

Тем не менее, характеристики процессов работы системы в целом и отдельных ее элементов очень важны, поскольку позволяют находить и устранять узкие места. Общий недостаток всех вводимых ранее характеристик – это отсутствие четкого обоснования. Именно их обоснованию и будет посвящена данная лекция. Будем рассматривать характеристики в рамках некоторой модели процесса работы устройств, вполне достаточной для практического применения. Мы не будем интересоваться содержанием операций, выполняемых функциональными устройствами. Они могут означать арифметические или логические функции, ввод-вывод данных, пересылку данных в память и извлечение данных из нее или что-либо иное. Более того, допускается, что при разных обращениях операции могут означать разные функции. Для нас сейчас важны лишь времена выполнения операций и то, на устройствах какого типа они реализуются.

Пусть введена система отсчета времени и установлена его единица, например, секунда. Будем считать, что длительность выполнения операций измеряется в долях единицы. Все устройства, реальные или гипотетические,