ТЕХНОЛОГИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

1. Численное моделирование и параллельные вычисления

В настоящее время практически нет такой сферы человеческой деятельности, где бы в той или иной форме не использовалось численное моделирование. Особенно это относится к областям науки и современных технологий. Поиски новых знаний, решений все более усложняющихся технологических задач привели к пониманию того, что чисто эмпирический путь решения сложных проблем давно себя исчерпал. Эффективное использование накопленного многими поколениями запаса знаний должно быть направлено на построение математических моделей, достаточно адекватно отражающих суть изучаемых явлений или объектов. Как правило, математическая модель представляет собой некоторую совокупность дифференциальных, интегральных, алгебраических или каких-то иных соотношений, определенных в области, так или иначе связанных с предметом исследований. Для реализации этих моделей нужно построить алгоритм решения задачи, описывающей эту модель, что в подавляющем большинстве случаев возможно лишь с помощью численных методов. Чаще всего здесь используется принцип дискретизации изучаемого объекта вместе с окружающей его средой, когда они разбиваются на отдельные элементы, на которые переносятся дискретные (численные) аналоги связей, описываемых математической моделью. В результате получаются системы уравнений с очень большим числом неизвестных. Понятно, что чем больше число таких дискретных элементов, тем более точным получается результат, а потому уровень дискретизации определяется только возможностью компьютера решить возникающую систему уравнений за разумное время. Так, при расчетах обтекания летательного аппарата сложной конфигурации общее число элементов, на которые разбивается расчетная область, составляет порядка 10^6. В каждом таком элементе надо знать 5 величин (3 компоненты вектора скорости, температуру и давление), т.е. всего $5 \cdot 10^6$ неизвестных. Если рассчитывается нестационарный режим обтекания (а это наиболее важный режим), то приходится отыскивать значения упомянутых неизвестных в $10^2 - 10^4$ слоях по времени. Поэтому одних только значимых результатов промежуточных вычислений необходимо найти примерно $10^9 - 10^{11}$. Для нахождения и обработки указанных 5 величин только в одном дискретном элементе нужно выполнить $10^2 - 10^3$ арифметических операций. Поэтому даже приближенные оценки показывают, что общее число операций, необходимых для решения задачи обтекания, составляет величину $10^{15} - 10^{16}$. Поэтому для достижения разумного времени решения такой задачи необходима вычислительная система с производительностью порядка $10^9 - 10^{10}$ арифметических операций с плавающей запятой в секунду при оперативной памяти не менее 10^9 слов.

Примеров подобных и более сложных задач, для решения которых необходимы вычислительные системы сверхвысокой производительности, можно привести десятки и сотни из самых разнообразных областей человеческой деятельности. Потребность в их решении заставляет создавать все более совершенные компьютеры. В свою очередь, более совершенные компьютеры позволяют улучшать математические модели и ставить все новые задачи, требующие от компьютеров еще большей производительности. В результате появились параллельные вычислительные системы. Задачи от этого параллельными не стали, но начали развиваться алгоритмы с параллельной структурой вычислений. Опять появилась возможность увеличить размерность решаемых задач. Теперь на очереди стоят оптические и квантовые компьютеры. Конца этому процессу не видно.

2. История развития многопроцессорных комплексов и параллельных вычислений

Прошло немногим более 50 лет с момента появления первых электронных вычислительных машин - компьютеров. За это время сфера их применения охватила практически все области человеческой деятельности - словом, все те области, где необходима обработка больших объемов информации. Как уже упоминалось, такие задачи возникли в середине прошлого века в связи с развитием атомной энергетики, авиастроения, ракетно-космических технологий и ряда других областей науки и техники.

Вычислительное направление применения компьютеров всегда оставалось основным двигателем прогресса в самих компьютерных технологиях. Не удивительно поэтому, что в качестве основной характеристики компьютеров используется такой показатель, как производительность - величина, показывающая, какое количество арифметических операций он может выполнить за единицу времени. Именно этот показатель с наибольшей очевидностью демонстрирует масштабы прогресса, достигнутого в компьютерных технологиях. Так, например, производительность одного из самых первых компьютеров EDSAC составляла всего около 100 операций в секунду, тогда как пиковая производительность Earth Simulator, одного из самых мощных на сегодняшний день суперкомпьютеров, оценивается в 40 триллионов операций в секунду. Т.е. произошло увеличение быстродействия в 400 миллиардов раз! Невозможно назвать другую сферу человеческой деятельности, где прогресс был бы столь очевиден и так велик. Естественно, что у любого человека сразу же возникает вопрос: за счет чего это оказалось возможным? Как ни странно, ответ довольно прост: примерно 1000-кратное увеличение скорости работы электронных схем и максимально широкое распараллеливание обработки данных.

Идея параллельной обработки данных как мощного резерва увеличения производительности вычислительных аппаратов была высказана Чарльзом Бэббиджем примерно за сто лет до появления первого электронного компьютера. Однако уровень развития технологий середины 19-го века не позволил ему реализовать эту идею. С появлением первых электронных компьютеров эти идеи неоднократно становились отправной точкой при разработке самых передовых и производительных вычислительных систем. Без преувеличения можно сказать, что вся история развития высокопроизводительных вычислительных систем - это история реализации идей параллельной обработки на том или ином этапе развития компьютерных технологий, естественно, в сочетании с увеличением скорости и надежности работы электронных схем.

Принципиально важными решениями в повышении производительности вычислительных систем были: введение конвейерной организации выполнения команд; включение в систему команд векторных операций, позволяющих одной командой обрабатывать целые массивы данных; распределение вычислений на множество процессоров. Сочетание этих 3-х механизмов в архитектуре суперкомпьютера Earth Simulator, состоящего из 5120 векторно-конвейерных процессоров, и позволило ему достичь рекордной производительности, которая в 20000 раз превышает производительность современных персональных компьютеров.

Очевидно, что такие системы чрезвычайно дороги и изготавливаются в единичных экземплярах . Ну, а что же производится сегодня в промышленных масштабах? Широкое разнообразие производимых в мире компьютеров с большой степенью условности можно разделить на четыре класса: персональные компьютеры (Personal Computer - PC); рабочие станции (WorkStation - WS); суперкомпьютеры (Supercomputer - SC); кластерные системы.

Условность разделения связана в первую очередь с быстрым прогрессом в развитии микроэлектронных технологий. Производительность компьютеров в каждом из классов удваивается в последние годы примерно за 18 месяцев (в соответствии с так называемым законом Мура). В связи с этим, суперкомпьютеры начала 90-х годов зачастую уступают в производительности современным рабочим станциям, а персональные компьютеры начинают успешно конкурировать по производительности с рабочими станциями. Тем не менее, попытаемся каким-то образом классифицировать их.

Персональные компьютеры. Как правило, в этом случае подразумеваются однопроцессорные системы на платформе Intel или AMD, работающие под управлением однопользовательских операционных систем (Microsoft Windows и др.). Используются в основном как персональные рабочие места.

Рабочие станции. Это чаще всего компьютеры с RISC процессорами с многопользовательскими операционными системами, относящимися к семейству

ОС UNIX. Содержат от одного до четырех процессоров. Поддерживают удаленный доступ. Могут обслуживать вычислительные потребности небольшой группы пользователей.

Суперкомпьютеры. Отличительной особенностью суперкомпьютеров является то, что это, как правило, большие и, соответственно, чрезвычайно дорогие многопроцессорные системы. В большинстве случаев в суперкомпьютерах используются те же серийно выпускаемые процессоры, что и в рабочих станциях. Поэтому зачастую различие между ними не столько качественное, сколько количественное. Например, можно говорить о 4-х процессорной рабочей станции фирмы SUN и о 64-х процессорном суперкомпьютере фирмы SUN. Скорее всего, в том и другом случае будут использоваться одни и те же микропроцессоры.

Кластерные системы. В последние годы широко используются во всем мире как дешевая альтернатива суперкомпьютерам. Система требуемой производительности собирается из готовых серийно выпускаемых компьютеров, объединенных опять же с помощью некоторого серийно выпускаемого коммуникационного оборудования. Таким образом, многопроцессорные системы, которые ранее ассоциировались в основном с суперкомпьютерами, в настоящее время прочно утвердились во всем диапазоне производимых вычислительных систем, начиная от персональных компьютеров и заканчивая суперкомпьютерами на базе векторно-конвейерных процессоров. Это обстоятельство, с одной стороны, увеличивает доступность суперкомпьютерных технологий, а с другой, повышает актуальность их освоения, поскольку для всех типов многопроцессорных систем требуется использование специальных технологий программирования для того, чтобы программа могла в полной мере использовать ресурсы высокопроизводительной вычислительной системы . Обычно это достигается разделением программы с помощью того или иного механизма на параллельные ветви, каждая из которых выполняется на отдельном процессоре.

3. Использование многопроцессорных систем

Суперкомпьютеры разрабатываются в первую очередь для того, чтобы с их помощью решать сложные задачи, требующие огромных объемов вычислений. При этом подразумевается, что может быть создана единая программа, для выполнения которой будут задействованы все ресурсы суперкомпьютера. Однако не всегда такая единая программа может быть создана или ее создание целесообразно. В самом деле, при разработке параллельной программы для многопроцессорной системы мало разбить программу на параллельные ветви. Для эффективного использования ресурсов необходимо обеспечить равномерную загрузку всех процессоров, что в свою очередь означает, что все ветви программы должны выполнить примерно одинаковый объем вычислительной работы. Однако не всегда этого можно достичь. Например, при

решении некоторой параметрической задачи для разных значений параметров, время поиска решения может значительно различаться. В таких случаях, видимо, разумнее независимо выполнять расчеты для каждого параметра с помощью обычной однопроцессорной программы. Но даже в таком простом случае могут потребоваться суперкомпьютерные ресурсы, поскольку выполнение полного расчета на однопроцессорной системе может потребовать слишком длительного времени. Параллельное выполнение множества программ для различных значений параметров позволяет существенно ускорить решение задачи. Наконец, следует отметить, что использование суперкомпьютеров всегда более эффективно для обслуживания вычислительных потребностей большой группы пользователей, чем использование эквивалентного количества однопроцессорных рабочих станций, так как в этом случае с помощью некоторой системы управления заданиями легче обеспечить равномерную и более эффективную загрузку вычислительных ресурсов.

В отличие от обычных многопользовательских систем, для достижения максимальной скорости выполнения программ операционные системы суперкомпьютеров, как правило, не позволяют разделять ресурсы одного процессора между разными, одновременно выполняющимися программами. Поэтому, как два противоположных варианта, возможны следующие режимы использования n-процессорной системы:

- все ресурсы отдаются для выполнения одной программы, и тогда мы вправе ожидать n-кратного ускорения работы программы по сравнению с однопроцессорной системой;
- одновременно выполняется n обычных однопроцессорных программ, при этом пользователь вправе рассчитывать, что на скорость выполнения его программы не будут оказывать влияния другие программы.

4. Параллелизм

Параллельная обработка данных, воплощая идею одновременного выполнения нескольких действий, имеет две разновидности: конвейерность и собственно параллельность. Оба вида параллельной обработки интуитивно понятны, поэтому сделаем лишь небольшие пояснения.

Параллельная обработка. Если некое устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что имеется пять таких же независимых устройств, способных работать одновременно и независимо, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично, система из N устройств ту же работу выполнит примерно за 1000/N единиц времени.

Конвейерная обработка. Что необходимо для сложения двух вещественных чисел, представленных в форме с плавающей запятой? Целое множество мелких операций, таких как сравнение порядков, выравнивание порядков, сложение

мантисс, нормализация и т.п. Процессоры первых компьютеров выполняли все эти "микрооперации" для каждой пары аргументов последовательно одна за другой до тех пор, пока не доходили до окончательного результата, и лишь после этого переходили к обработке следующей пары слагаемых.

Идея конвейерной обработки заключается в выделении отдельных этапов выполнения общей операции, причем каждый этап, выполнив свою работу, передает результат следующему, одновременно принимая новую порцию входных данных. Получаем очевидный выигрыш в скорости обработки за счет совмещения прежде разнесенных во времени операций. Предположим, что в операции можно выделить пять микроопераций, каждая из которых выполняется за одну единицу времени. Если есть одно неделимое последовательное устройство, то 100 пар аргументов оно обработает за 500 единиц. Если же каждую микрооперацию выделить в отдельный этап (или иначе говорят - ступень) конвейерного устройства, то на пятой единице времени на разной стадии обработки такого устройства будут находиться первые пять пар аргументов, первый результат будет получен через 5 единиц времени, каждый следующий - через одну единицу после предыдущего, а весь набор из ста пар будет обработан за 5+99=104 единицы времени, то есть будет получено ускорение по сравнению с последовательным устройством почти в пять раз (по числу ступеней конвейера).

Приблизительно так же будет и в общем случае. Если конвейерное устройство содержит L ступеней, а каждая ступень срабатывает за одну единицу времени, то время обработки *n* независимых операций этим устройством составит L+n-1 единиц. Если это же устройство использовать в монопольном режиме (как последовательное), то время обработки будет равно Lxn. В результате получим ускорение почти в L раз за счет использования конвейерной обработки данных.

Казалось бы, конвейерную обработку можно с успехом заменить обычным параллелизмом, для чего продублировать основное устройство столько раз, сколько ступеней конвейера предполагается выделить. Однако стоимость и сложность получившейся системы будет несопоставима со стоимостью и сложностью конвейерного варианта, а производительность будет почти такой же.

Рассмотрим в самых общих чертах анализ эффективности распараллеливания алгоритмов, например, на кластерных системах. Такая параллельная вычислительная система состоит из процессоров и коммуникационной сети, обеспечивающей обмен информацией между процессорами. Примем за коэффициент ускорения вычислений отношение времени T, затраченного на решение данной задачи на одном процессоре (последовательный алгоритм) ко времени, затраченному на N процессорах. Тогда приближенно можно ввести следующие оценки затрат времени на кластерной системе. Время работы одного процессора, затрачиваемого на решение задачи есть не мене, чем $t = T / N$. Предположим, что время, затрачиваемое на обмены одним процессором есть

величина постоянная, равная $t_{об}$. Тогда общие затраты времени параллельного алгоритма на N процессорах можно оценить как $T_N = T/N + N \cdot t_{об}$ и коэффициент ускорения вычислений запишется в виде

$$K_y = \frac{T}{\frac{T}{N} + N \cdot t_{об}} = \frac{N}{1 + N^2 \cdot \frac{t_{об}}{T}}$$

Из этой формулы следует, что только при отсутствии потерь времени на обмены коэффициент ускорения K_y может достигать своего теоретического значения $K_y = N$. В противном случае при $N \to \infty$ $K_y \to 0$, т.е. функция $K_y(N)$ выпуклая, и для каждой задачи существует некоторое свое оптимальное число процессоров N_{opt}, обеспечивающих максимальное ускорение вычислений, которое приближенно можно найти из условия $\frac{dK_y}{dN} = 0$, полагая, что N – непрерывная величина. Тогда

$$N_{opt} = \sqrt{\frac{T}{t_{об}}} \ .$$

5. Основные принципы распараллеливания численных алгоритмов

Независимо от того, как устроены параллельные вычислительные системы (ВС), все они базируются на одной и той же методологической основе. Именно, каждая такая система имеет какое-то число функциональных устройств, которые могут работать независимо друг от друга и способны выполнять операции алгоритма. Это означает, что для того, чтобы алгоритм мог быть реализован на параллельной системе, он должен быть представим в виде последовательности ансамблей операций. При этом все операции одного ансамбля должны быть независимы от операций других ансамблей и обладать возможностью быть выполненными одновременно на имеющихся в системе функциональных устройствах. Понятно, что чем больше в алгоритме задачи таких ансамблей, тем большую возможность для распараллеливания имеет алгоритм.

Поскольку технические возможности параллельных ВС предполагают наличие большого числа процессоров, поиски и разработки новых алгоритмов для таких систем с неограниченным числом процессоров интенсивно начались в 50 – 60 гг. прошлого столетия, однако ни к каким принципиально новым результатам они не привели. Причиной этого явилось сложность коммуникационных сетей для их реализации, большое число конфликтов в памяти, повышенная численная неустойчивость таких алгоритмов. Поэтому в настоящее время программное обеспечение существующих параллельных систем почти целиком состоит из программ, реализующих те же методы, которые хорошо зарекомендовали себя при их реализации на последовательных

компьютерах с выделением параллельных ветвей, ускоряющих их выполнение на параллельных ВС.

Общая методология здесь следующая. Как на последовательном, так и на параллельном компьютере решение задачи находится в результате выполнения множества простых операций, имеющих обычно аргументы двух типов – входные и выходные, причем входными являются, как правило, результаты выполнения других операций. Понятно, что любая операция – потребитель аргументов – не может начать выполняться раньше, чем закончится выполнение всех операций – поставщиков для нее аргументов. Тем самым на множестве всех операций алгоритма разработчик программы явно или неявно устанавливает **частичный порядок**. Для любых двух операций порядок определяет одну из возможностей: либо указывает, какая из операций должна выполняться раньше, либо констатирует, что обе операции могут выполняться независимо друг от друга. При этом оказывается, что при одном и том же частичном порядке общий временной порядок всего множества операций может быть различным, но любой из них дает один и тот же результат. Поэтому сохранение частичного порядка, заданного программой, и есть то условие, выполнение которого гарантирует однозначность результата. При этом в рамках одного и того же частичного порядка возможен выбор любой реализации.

Эта концепция распараллеливания уже не требует дополнительных исследований параллельного алгоритма на его вычислительную устойчивость и широко применяется на практике.

Пусть программа описывает какой-то алгоритм. Построим ориентированный граф, вершинами которого будут множества всех операций алгоритма. Выберем любую пару вершин (u, v). Пусть, согласно частичному порядку, операция, соответствующая вершине u, должна поставлять аргументы операции, соответствующей вершине v. Тогда проведем дугу из вершины u в вершину v. Если соответствующие операции могут выполняться независимо друг от друга, дугу проводить не будем. Если аргументом операции являются начальные данные или выходные данные, то эти вершины не будут иметь соответственно входящих и исходящих дуг. Построенный таким образом граф есть граф информационной зависимости реализации алгоритма при фиксированных входных данных или просто – **граф алгоритма**. Такой граф есть ориентированный ациклический мультиграф. Его ацикличность следует из того, что в любых программах реализуются только явные вычисления (даже в рекурсиях) и никакая величина не может определяться через саму себя. Граф действительно является мультиграфом, поскольку две вершины могут быть связаны несколькими дугами. Это будет иметь место тогда, когда в качестве разных аргументов одной и той же операции используется одна и та же величина.

Итак, обозначим граф стандартным образом G = (V, E), где V – множество вершин, E – множество дуг. Выберем в графе вершины, не имеющие предшествующих, и пометим их индексом 1. Удалим из графа помеченные вершины и инцидентные им дуги. Оставшийся граф также является ацикличным. Выберем в нем вершины, не имеющие предшествующих, и пометим их индексом

2 и т.д. до тех пор, пока не исчерпаем весь граф. Полученная форма графа называется **канонической параллельной формой** (в дальнейшем просто параллельной формой). Группа вершин, имеющих одинаковый индекс, называется *ярусом*, а число вершин в такой группе – его *шириной.* Число ярусов в параллельной форме называется высотой параллельной формы, а максимальная ширина ярусов – ее шириной.

Конечно, это идеальное представление алгоритма, предполагающее, что все операции одного яруса начинаются и заканчиваются одновременно. Кроме неодновременности завершения операций не учитывается время, затрачиваемое на передачу аргументов, т.е. на пересылки и многое другое. Однако, приведение алгоритма к канонической параллельной форме позволяет оценить запас параллелизма в нем, что позволяет понять, как такой алгоритм лучше всего реализовать на компьютере с конкретной параллельной архитектурой. Конечно, представить сколько-нибудь сложный алгоритм, содержащий сотни и тысячи простых операций в параллельной форме не представляется возможным, однако ничто не мешает в вершинах графа размещать не отдельные операторы, а целые блоки программы, с небольшим числом входных и выходных аргументов, имеющие свою внутреннюю структуру. Тем самым можно с помощью параллельной формы реализовывать и стратегию крупноблочного распараллеливания.

Рассмотрим простой пример. Пусть нужно перемножить 8 чисел $a_1 \cdot a_2 \cdot \dots \cdot a_8$. Обычная схема процесса последовательного умножения выглядит следующим образом

1) $a_1 \cdot a_2$
2) $(a_1 \cdot a_2) \cdot a_3$
3) $(a_1 \cdot a_2 \cdot a_3) \cdot a_4$

……..

7) $(a_1 \cdot a_2 \cdot \dots \cdot a_7) \cdot a_8$

Высота алгоритма равна 7, ширина равна 1.
Если вычислительная система имеет более одного процессора, то при данной схеме умножения на всех шагах все они будут простаивать, кроме одного.
Изменим теперь алгоритм умножения, называемый *процессом сдваивания*.

1) $a_1 \cdot a_2$ $a_3 \cdot a_4$ $a_5 \cdot a_6$ $a_7 \cdot a_8$
2) $(a_1 \cdot a_2) \cdot (a_3 \cdot a_4)$ $(a_5 \cdot a_6) \cdot (a_7 \cdot a_8)$
3) $(a_1 \cdot a_2 \cdot a_3 \cdot a_4)$ $(a_5 \cdot a_6 \cdot a_7 \cdot a_8)$

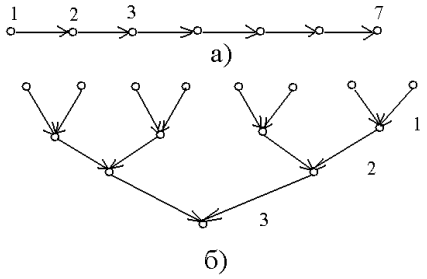


Рис. 1.1. Параллельные формы

Графы параллельной формы этих алгоритмов приведены на рис. 1.1.
Алгоритм сдваивания можно применять и для любых ассоциативных операций, например, сложения чисел, умножения матриц и т.д.

6. Использование параллелизма

Предположим теперь, что мы научились каким-то образом строить и исследовать информационную историю выполнения программы. Выделив в ней множества информационно независимых операций, приходим к вопросу: каким же образом распределять эти множества между процессорами или другими обрабатывающими устройствами?

Выше был описан достаточно общий подход, основанный на построении ярусно-параллельной формы программы. Однако ярусно-параллельная форма почти никогда не используется для практического распараллеливания программ. Причиной этого является то, что описанный способ распараллеливания плохо согласован как с конструкциями реальных языков программирования, так и с архитектурными особенностями современных компьютеров. Чтобы убедиться в этом, можете попробовать сконструировать таким способом параллельную реализацию практически любого алгоритма и оценить как сложность написания такой программы, так и количество необходимых коммуникаций между процессорами.

В реальности гораздо чаще используются другие способы распараллеливания, использующие характерные особенности наиболее распространенных языков программирования. Так, самым простым вариантом распределения работ между процессорами является примерно следующая конструкция, называемая ***крупноблочным распараллеливанием*:**

```
if (MyProc =0) {/* операции, выполняемые 0-ым процессором */}
……..
if (MyProc = K) {/* операции, выполняемые K-ым процессором */}
```

При этом предполагается, что каждый процессор каким-то образом может получить уникальный номер, присвоить его переменной MyProc и использовать в

дальнейшем для получения участка кода для независимого исполнения. Таким образом, в приведенном примере операции в первых фигурных скобках будут выполнены только процессором с номером 0, операции во вторых фигурных скобках - процессором с номером к и т.д. При этом, естественно, необходимо, чтобы одновременно разные процессоры могли выполнять только блоки информационно независимых операций, что может потребовать операций синхронизации процессоров, а также при необходимости нужно обеспечивать обмен данными между процессорами.

Однако далеко не всегда удается выделить в программе достаточно большое число блоков независимых операций, а значит, при значительном числе процессоров в используемом компьютере, часть из них будет простаивать. Поэтому наряду с предыдущим способом используют также более низкоуровневое распараллеливание. Как показывает практика, наибольший ресурс параллелизма в программах сосредоточен в циклах. Поэтому наиболее распространенным способом распараллеливания является то или иное *распределение итераций циклов*. Если между итерациями некоторого цикла нет информационных зависимостей, то их можно тем или иным способом раздать разным процессорам для одновременного исполнения. Условно это может быть выражено примерно следующей конструкцией:

```
for (i = 0; i < N; i++) {
……….
      if (i ~ MyProc) { /* операции i-й итерации для выполнения
процессором MyProc */
            ……
```

Здесь конструкция i ~ MyProc применена для того, чтобы указать, что номер итерации i каким-то образом соотносится с номером процессора MyProc. Конкретный способ задания этого соотношения определяет то, какие итерации цикла на какие процессоры будут распределяться. В принципе, ничто не мешает, например, раздать всем процессорам по одной итерации, а все остальные итерации выполнить каким-то одним процессором. Однако очевидно, что в подавляющем числе случаев такое распределение неэффективно, поскольку все процессоры, кроме одного, выполнив свою итерацию, будут, скорее всего, простаивать. Таким образом, одним из требований к распределению итераций (как, впрочем, и к процессу распараллеливания вообще) является по возможности *равномерная загрузка процессоров*. Наиболее распространенные способы распределения итераций циклов в той или иной степени удовлетворяют этому требованию.

Блочное распределение итераций предполагает, что распределение итераций цикла по процессорам ведется блоками по несколько последовательных итераций. В простейшем случае количество итераций цикла N делится на число про-

цессоров p, результат округляется до ближайшего целого сверху и число [N/p] определяет количество итераций в блоке. При этом почти все процессоры получают одинаковое количество итераций, однако один или несколько последних процессоров могут простаивать. Выбор меньшего размера блока может уменьшить этот дисбаланс, однако при этом часть итераций останется нераспределенной. Если нераспределенные итерации снова начать распределять такими же блоками, начиная с первого процессора, то получим *блочно-циклическое распределение* итераций. Если уменьшать количество итераций дальше, то дойдем до распределения по одной итерации, которое называется *циклическим распределением.* Циклическое распределение позволяет минимизировать дисбаланс в загрузке процессоров, возникавший при блочных распределениях.

Рассмотрим небольшой пример. Пусть требуется распределить по процессорам итерации следующего цикла:

```
for  (i =  0;  i < N;  i++)

a[i]=a[i]+b[i];
```

Пусть в целевом компьютере имеется `P` процессоров с номерами `0...P-1`. Тогда блочное распределение итераций этого цикла можно записать следующим образом:

```
k = (N-1)/P + 1; /* размер блока итераций */
ibeg = MyProc * k; /* начало блока итераций процессора MyProc */
iend = (MyProc + 1) * k - 1; /* конец блока итераций процессора
MyProc */
if (ibeg >= N) iend = ibeg - 1; /* если процессору не досталось
итераций */
else if (iend >= N) iend = N - 1; /* если процессору досталось
меньше итераций */
for (i = ibeg; i <= iend;
  a[i]=a[i]+b[i];
```

Циклическое распределение итераций того же цикла можно записать так:

```
for  (i  = MyProc;  i  < N;  i+=P)
a[i]=a[i]+b[i];
```

Нужно заметить, что все соображения о распределении итераций циклов с целью достижения равномерности загрузки процессоров имеют смысл только в предположении, что распределяемые итерации приблизительно равноценны по времени исполнения. Во многих реальных случаях (например, при решении матричных задач с треугольной матрицей) это может быть не так, а значит, могут потребоваться совершенно другие способы распределения.

Однако только равномерной загрузки процессоров обычно недостаточно для получения эффективной параллельной программы. Только в крайне редких случаях программа не сдержит информационных зависимостей вообще. Если же есть информационная зависимость между операциями, которые при выбранной схеме распределения попадают на разные процессоры, то потребуется пересылка данных. Обычно пересылки требуют достаточно большого времени для своего осуществления, поэтому другой важной целью при распараллеливании является *минимизация количества и объема необходимых пересылок данных.* Так, например, при наличии информационных зависимостей между i-й и (i+1)-й итерациями некоторого цикла блочное распределение итераций может оказаться эффективнее циклического, потому что при блочном распределении соседние итерации попадают на один процессор, а значит, потребуется меньшее количество пересылок, чем при циклическом распределении.

До сих пор говорилось о распределении итераций одномерных циклов, однако в программах часто встречаются многомерные циклические гнезда, причем каждый цикл такого гнезда может содержать некоторый ресурс параллелизма. Для его использования производят анализ и разбиение пространства итераций исследуемого фрагмента. *Пространством итераций* гнезда тесно вложенных циклов называют множество целочисленных векторов I, координаты которых задаются значениями параметров циклов данного гнезда. Задача распараллеливания при этом сводится к разбиению множества векторов I на подмножества, которые выполняются последовательно друг за другом, но в рамках каждого такого подмножества итерации могут быть выполнены одновременно и независимо.

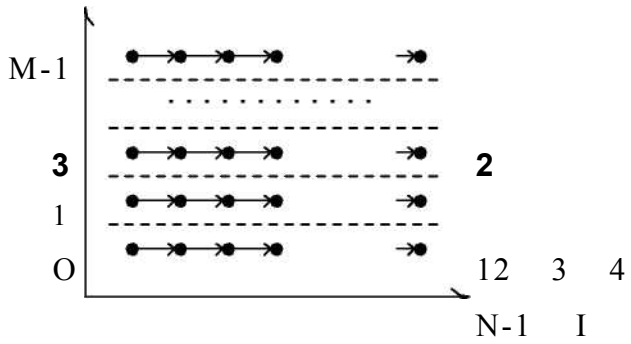
Среди методов анализа пространства итераций можно выделить несколько наиболее известных: методы гиперплоскостей, координат, параллелепипедов и пирамид.

Метод гиперплоскостей заключается в том, что пространство итераций размерности n разбивается на гиперплоскости размерности n-1 так, что все операции, соответствующие точкам одной гиперплоскости, могут выполняться одновременно и асинхронно. *Метод координат* заключается в том, что пространство итераций фрагмента разбивается на гиперплоскости, ортогональные одной из координатных осей. *Метод параллелепипедов* является логическим развитием двух предыдущих методов и заключается в разбиении пространства итераций на n-мерные параллелепипеды, объем которых определяет результирующее ускорение программы. В *методе пирамид* выбираются итерации, вырабатывающие значения, которые далее в теле гнезда циклов не используются. Каждая такая итерация служит основанием отдельной параллельной ветви, в которую также входят все итерации, влияющие информационно на выбранную. При этом зачастую информация в разных ветвях дублируется, что может привести к потере эффективности.

Рассмотрим небольшой пример:

```
for (i = 1; i < N; i++)
    for (j = 1; j < M; j++)
    a[i, j] = a[i − 1, j] + a[i, j];
```

Пространство итераций данного фрагмента можно изобразить следующим образом:

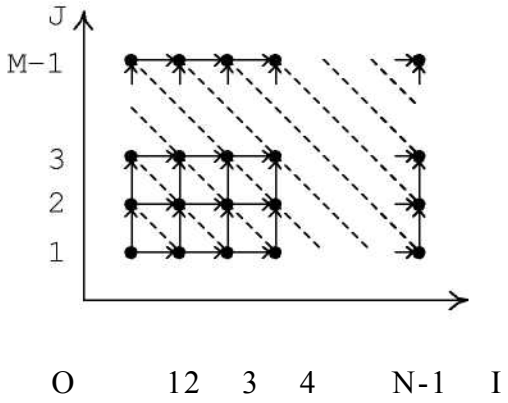


На этом рисунке кружки соответствуют отдельным срабатываниям оператора присваивания, а стрелки показывают информационные зависимости. Сразу видно, что разбиение пространства итераций по измерению I приведет к разрыву информационных зависимостей. Однако информационных зависимостей по измерению J нет, поэтому возможно применение метода координат с разбиением пространства итераций гиперплоскостями, ортогональными оси J, например, как показано на рисунке пунктирными линиями. Затем операции, попавшие в одну группу, распределяются для выполнения на один процессор целевого компьютера.

Немного усложним пример:

```
for (i = 1; i < N; i++)
    for (j = 1; j < M; j++)
    a[i, j] = a[i − 1, j] + a[i, j − 1];
```

Пространство итераций данного фрагмента можно изобразить следующим образом:



Очевидно, что метод координат в данном случае неприменим, потому что любое разбиение как по измерению I, так и по измерению J приведет к разрыву информационных зависимостей. Однако на рисунке пунктирными линиями показаны гиперплоскости I + J = const, которые содержат вершины, между которыми нет информационных зависимостей. Это означает возможность применения метода гиперплоскостей, при котором осуществляется перебор гиперплоскостей I + J = const, и для каждой из них соответствующие операции распределяются между процессорами целевого компьютера.

Если переходить от распараллеливания отдельных циклических конструкций к распараллеливанию целой программы, то может оказаться невыгодным использовать весь найденный ресурс параллелизма (в особенности, на компьютерах с распределенной памятью), поскольку потребуется большое количество перераспределений данных между выполнением циклов.

В некоторых случаях можно добиться более эффективного распараллеливания программы при помощи *эквивалентных преобразований* - таких преобразований кода программы, при которых полностью сохраняется результат ее выполнения. Существует достаточно большое число подобных преобразований, полезных в различных случаях, например: перестановки циклов, схлопывание циклов, расщепление циклов и т.д.

Приведем один небольшой пример. Допустим, что в программе содержится такой цикл:

```
for (i = 1; i < N; i++) {

    a[i] = a[i] + b[i];
    c[i] = c[i-1] + b[i];
}
```

В данном цикле есть информационные зависимости между срабатываниями второго оператора из тела цикла на i-й и (i -1)-й итерациях. Поэтому нельзя просто раздать итерации исходного цикла для выполнения различным процессорам. Однако, достаточно очевидно, что этот цикл можно разбить на два, первый из которых станет параллельным:

```
for (i = 1; i < N; i++)
     a[i] = a[i] + b[i];
for (i = 1; i < N; i++)
     c[i] = c[i-1] + b[i];
```

В некоторых случаях и эквивалентные преобразования бессильны помочь в распараллеливании программы, но допустимы другие методы при предположе-

нии, что можно пренебречь ошибками округления. Например, пусть нужно произвести суммирование элементов массива:

```
for  (i = 0,  s = 0;  i < N;  i++)
            s+=a[i];
```

Если подойти к этому циклу формально, то распараллелить его не получится, так как между итерациями цикла существуют информационные зависимости. Однако, если ошибками округления, вызванными разным порядком суммирования элементов массива a, допустимо пренебречь, то данный фрагмент можно распараллелить при помощи широко известной *схемы сдваивания:* на первом шаге первый процессор суммирует элементы a[0] и a[1], второй процессор суммирует элементы a[2] и a[3] и т.д., на следующем шаге попарно суммируются эти частичные суммы и так до получения окончательного результата. Если имеется N/2 процессоров, то весь алгоритм суммирования выполняется за $\log_2 N$ параллельных шагов.

7. Эффективность распараллеливания

Естественно, что, используя параллельную систему с p вычислительными устройствами, пользователь ожидает получить ускорение своей программы в p раз по сравнению с последовательным вариантом. Но действительность практически всегда оказывается далека от идеала.

Предположим, что структура информационных зависимостей программы определена (что в общем случае является весьма непростой задачей), и доля операций, которые нужно выполнять последовательно, равна f, где $0 \le f \le 1$ (при этом доля понимается не по статическому числу строк кода, а по времени выполнения последовательной программы). Крайние случаи в значениях f соответствуют полностью параллельным (f = 0) и полностью последовательным (f = 1) программам. Тогда для того, чтобы оценить, какое ускорение S может быть получено на компьютере из p процессоров при данном значении f, можно воспользоваться *законом Амдала:*

$$S \le \frac{1}{f + \frac{1-f}{p}}$$

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

что не любая программа может быть эффективно распараллелена. Для того чтобы это было возможно, необходимо, чтобы доля информационно независимых операций была очень большой. В принципе, это не должно отпугивать от параллельного программирования, потому что, как показывает практика, большинство вычислительных алгоритмов устроено в этом смысле достаточно хорошим образом.

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

Однако даже если все процессоры одинаковы, возникают другие проблемы. Процессоры выполнили свою работу, но результатами чаще всего надо обмениваться для продолжения вычислений, а на передачу данных уходит время, и в это время процессоры опять простаивают... Кроме указанных, есть и еще большое количество факторов, влияющих на эффективность выполнения параллельных программ, причем все они действуют одновременно, а значит, все в той или иной степени должны учитываться при распараллеливании.

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