Параллельная обработка данных

(лектор Воеводин Вл. В.)

Лекция 1. Введение в предмет.

Параллельные компьютеры и супер-ЭВМ

О том, что где-то существуют некие мистические "очень мощные" компьютеры слышал, наверное, каждый. В самом деле, не так давно было много разговоров о поставке в Гидрометеоцентр России могучих компьютеров фирмы Cray Research. В ноябре 1999 года состоялось официальное открытие Межведомственного суперкомпьютерного центра, который в настоящий момент имеет компьютер с 768 процессорами. Опять же, если компьютер с именем Deep Blue обыграл самого Гарри Каспарова, то он, согласитесь - и здесь интуиция Вас не подвела, ну никак не может быть простой персоналкой.

Для многих подобные компьютеры так и остаются тайной за семью печатями, некой TERRA INCOGNITA, с которой ассоциации всегда связаны с чем-то большим: огромные размеры, большие задачи, крупные фирмы и компании, невероятные скорости работы или что-то иное, но обязательно это будет "на грани", для чего "обычного" явно мало, а подойдет только "супер", суперкомпьютер или супер-ЭВМ. В этом интуитивном восприятии есть изрядная доля истины, поскольку к классу супер-ЭВМ принадлежат лишь те компьютеры, которые имеют максимальную производительность в настоящее время.

Быстрое развитие компьютерной индустрии определяет относительность данного понятия - то, что десять лет назад можно было назвать суперкомпьютером, сегодня под это определение уже не попадает. Например, производительность персональных компьютеров, использующих Pentium-III/500MHz, сравнима с производительностью суперкомпьютеров начала 70-х годов, однако по сегодняшним меркам суперкомпьютерами не являются ни те, ни другие.

В любом компьютере все основные параметры тесно связаны. Трудно себе представить универсальный компьютер, имеющий высокое быстродействие и мизерную оперативную память, либо огромную оперативную память и небольшой объем дисков. Следуя логике, делаем вывод: супер-ЭВМ это компьютеры, имеющие в настоящее время не только максимальную производительность, но и максимальный объем оперативной и дисковой памяти (вопрос о специализированном ПО, с помощью которого можно эффективно всем этим воспользоваться, пока оставим в стороне).

Так о чем же речь и какие суперкомпьютеры существуют в настоящее время в мире? Вот лишь несколько параметров, дающих достаточно красноречивую характеристику машин этого класса. Компьютер ASCI WHITE, занимающий первое место в списке пятисот самых мощных компьютеров мира, объединяет 8192 процессора Power 3 с общей оперативной памятью в 4 Терабайта и производительностью более 12 триллионов операций в секунду.

Супер-ЭВМ и сверхвысокая производительность: зачем?

Простые рассчеты показывают, что конфигурации подобных систем могут стоить не один миллион долларов США - ради интереса прикиньте, сколько стоят, скажем, лишь 4 Тбайта оперативной памяти? Возникает целый ряд естественных вопросов: какие задачи настолько важны, что требуются компьютеры стоимостью несколько миллионов долларов? Или, какие задачи настолько сложны, что хорошего Пентиума не достаточно? На эти и подобные им вопросы хотелось бы найти разумные ответы.

Для того, чтобы оценить сложность решаемых на практике задач, возьмем конкретную предметную область, например, оптимизацию процесса добычи нефти. Имеем подземный нефтяной резервуар с каким-то число пробуренных скважин: по одним на поверхность откачивается

нефть, по другим обратно закачивается вода. Нужно смоделировать ситуацию в данном резервуаре, чтобы оценить запасы нефти или понять необходимость в дополнительных скважинах.

Примем упрощенную схему, при которой моделируемая область отображается в куб, однако и ее будет достаточно для оценки числа необходимых арифметических операций. Разумные размеры куба, при которых можно получать правдоподобные результаты - это 100*100*100 точек. В каждой точке куба надо вычислить от 5 до 20 функций: три компоненты скорости, давление, температуру, концентрацию компонент (вода, газ и нефть - это минимальный набор компонент, в более реалистичных моделях рассматривают, например, различные фракции нефти). Далее, значения функций находятся как решение нелинейных уравнений, что требует от 200 до 1000 арифметических операций. И, наконец, если исследуется нестационарный процесс, т.е. нужно понять, как эта система ведет себя во времени, то делается 100-1000 шагов по времени. Что получилось:

$$10^6 \ \textit{точек сетки} * 10 \ \textit{функций} * 500 \ \textit{операций} * 500 \ \textit{шагов по времени} = 2{,}5 * 10^{12}$$

2500 миллиардов арифметических операций для выполнения одного лишь расчета! А изменение параметров модели? А отслеживание текущей ситуации при изменении входных данных? Подобные расчеты необходимо делать много раз, что накладывает очень жесткие требования на производительность используемых вычислительных систем.

Примеры использования суперкомпьютеров можно найти не только в нефтедобывающей промышленности. Вот лишь небольшой список областей человеческой деятельности, где использование суперкомпьютеров действительно необходимо:

- автомобилестроение
- нефте- и газодобыча
- фармакология
- прогноз погоды и моделирование изменения климата
- сейсморазведка
- проектирование электронных устройств
- синтез новых материалов
- и многие, многие другие

В 1995 году корпус автомобиля Nissan Maxima удалось сделать на 10% прочнее благодаря использованию суперкомпьютера фирмы Cray (The Atlanta Journal, 28 мая, 1995г). С помощью него были найдены не только слабые точки кузова, но и наиболее эффективный способ их удаления.

По данным Марка Миллера (Mark Miller, Ford Motor Company), для выполнения crash-тестов, при которых реальные автомобили разбиваются о бетонную стену с одновременным замером необходимых параметров, съемкой и последующей обработкой результатов, компании Форд понадобилось бы от 10 до 150 прототипов новых моделей при общих затратах от 4 до 60 миллионов долларов. Использование суперкомпьютеров позволило сократить число прототипов на одну треть.

Совсем недавний пример - это развитие одной из крупнейших мировых систем резервирования Amadeus, используемой тысячами агенств со 180000 терминалов в более чем ста странах. Установка двух серверов Hewlett-Packard T600 по 12 процессоров в каждом позволила довести степень оперативной доступности центральной системы до 99.85% при текущей загрузке около 60 миллионов запросов в сутки.

И подобные примеры можно найти повсюду. В свое время исследователи фирмы DuPont искали замену хлорофлюорокарбону. Нужно было найти материал, имеющий те же положительные качества: невоспламеняемость, стойкость к коррозии и низкую токсичность, но без вредного воздействия на озоновый слой Земли. За одну неделю были проведены необходимые расчеты на суперкомпьютере с общими затратами около 5 тысяч долларов. По оценкам специалистов DuPont, использование традиционных экспериментальных методов исследований потребовало бы около трех месяцев и 50 тысяч долларов и это без учета времени, необходимого на синтез и очистку необходимого количества вещества.

Увеличение производительности ЭВМ, за счет чего?

А почему суперкомпьютеры считают так быстро? Вариантов ответа может быть несколько, среди которых два имеют явное преимущество: развитие элементной базы и использование новых решений в архитектуре компьютеров.

Попробуем разобраться, какой из этих факторов оказывается решающим для достижения рекордной производительности. Обратимся к известным историческим фактам. На одном из первых компьютеров мира - EDSAC, появившемся в 1949 году в Кембридже и имевшем время такта 2 микросекунды ($2*10^{-6}$ секунды), можно было выполнить $2*n$ арифметических операций за $18*n$ миллисекунд, то есть в среднем 100 арифметических операций в секунду. Сравним с одним вычислительным узлом современного суперкомпьютера Hewlett-Packard V2600: время такта приблизительно 1.8 наносекунды ($1.8*10^{-9}$ секунд), а пиковая производительность около 77 миллиардов арифметических операций в секунду.

Что же получается? За полвека производительность компьютеров выросла более чем в ***семьсот миллионов*** раз. При этом выигрыш в быстродействии, связанный с уменьшением времени такта с 2 микросекунд до 1.8 наносекунд, составляет лишь около 1000 раз. Откуда же взялось остальное? Ответ очевиден - использование новых решений в архитектуре компьютеров. Основное место среди них занимает принцип параллельной обработки данных, воплощающий идею одновременного (параллельного) выполнения нескольких действий.

Параллельная обработка данных на ЭВМ

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 есть пять таких же независимых устройств, способных работать одновременно, 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 за $1000/N$ единиц времени. Подобные аналогии можно найти и в жизни: если один солдат вскопает огород за 10 часов, то рота солдат из пятидесяти человек с такими же способностями, работая одновременно, справятся с той же работой за 12 минут - принцип параллельности в действии!

Кстати, пионером в параллельной обработке потоков данных был академик А.А.Самарский, выполнявший в начале 50-х годов расчеты, необходимые для моделирования ядерных взрывов. Самарский решил эту задачу, посадив несколько десятков барышень с арифмометрами за столы. Барышни передавали данные друг другу просто на словах и откладывали необходимые цифры на арифмометрах. Таким образом, в частности, была рассчитана эволюция взрывной волны. Работы было много, барышни уставали, а Александр Андреевич ходил между ними и подбадривал. Это, можно сказать, и была первая параллельная система. Хотя расчеты водородной бомбы были мастерски проведены, точность их была очень низкая, потому что узлов в используемой сетке было мало, а время счета получалось слишком большим.

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 одной 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, передавал бы 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 каждую микрооперацию выделить в отдельный этап (или иначе говорят - ступень) конвейерного устройства, то на пятой единице времени на разной стадии обработки такого устройства будут находится первые пять пар аргументов, а весь набор из ста пар будет обработан за 5+99=104 единицы времени - ускорение по сравнению с последовательным устройством почти в пять раз (по числу ступеней конвейера).

Казалось бы конвейерную обработку можно с успехом заменить обычным параллелизмом, для чего продублировать основное устройство столько раз, сколько ступеней конвейера предполагается выделить. В самом деле, пять устройств предыдущего примера обработают 100 пар аргументов за 100 единиц времени, что быстрее времени работы конвейерного устройства! В чем же дело? Ответ прост, увеличив в пять раз число устройств, мы значительно увеличиваем как объем аппаратуры, так и ее стоимость. Представьте себе, что на автозаводе решили убрать конвейер, сохранив темпы выпуска автомобилей. Если раньше на конвейере одновременно находилась тысяча автомобилей, то действуя по аналогии с предыдущим примером надо набрать тысячу бригад, каждая из которых (1) в состоянии полностью собрать автомобиль от начала до конца, выполнив сотни разного рода операций, и (2) сделать это за то же время, что машина прежде находилась на конвейере. Представили себестоимость такого автомобиля? Нет? Согласен, трудно, разве что Ламборгини приходит на ум, но потому и возникла конвейерная обработка...

Краткая история появления параллелизма в архитектуре ЭВМ

Сегодня параллелизмом в архитектуре компьютеров уже мало кого удивишь. Все современные микропроцессоры, будь то Pentium III или PA-8700, MIPS R14000, E2K или Power3 используют тот или иной вид параллельной обработки. В ядре Pentium 4 на разных стадиях выполнения может одновременно находиться до 126 микроопераций. На презентациях новых чипов и в пресс-релизах корпораций это преподносится как последнее слово техники и передовой край науки, и это действительно так, если рассматривать реализацию этих принципов в миниатюрных рамках одного кристалла.

Вместе с тем, сами эти идеи появились очень давно. Изначально они внедрялись в самых передовых, а потому единичных, компьютерах своего времени. Затем после должной отработки технологии и удешевления производства они спускались в компьютеры среднего класса, и наконец сегодня все это в полном объеме воплощается в рабочих станциях и персональных компьютерах.

Для того чтобы убедиться, что все основные нововведения в архитектуре современных процессоров на самом деле используются еще со времен, когда ни микропроцессоров, ни понятия суперкомпьютеров еще не было, совершим маленький экскурс в историю, начав практически с момента рождения первых ЭВМ.

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

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

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

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

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

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

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

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

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

ILLIAC IV (1974): **матричные процессоры**. Проект: 256 процессорных элементов (ПЭ) = 4 квадранта по 64ПЭ, возможность реконфигурации: 2 квадранта по 128ПЭ или 1 квадрант из 256ПЭ, такт 40нс, производительность 1Гфлоп;

работы начаты в 1967 году, к концу 1971 изготовлена система из 1 квадранта, в 1974г. она введена в эксплуатацию, доводка велась до 1975 года;

- центральная часть: устройство управления (УУ) + матрица из 64 ПЭ;
- УУ это простая ЭВМ с небольшой производительностью, управляющая матрицей ПЭ; все ПЭ матрицы работали в синхронном режиме, выполняя в каждый момент времени одну и ту же команду, поступившую от УУ, но над своими данными;
- ПЭ имел собственное АЛУ с полным набором команд, ОП - 2Кслова по 64 разряда, цикл памяти 350нс, каждый ПЭ имел непосредственный доступ только к своей ОП;
- сеть пересылки данных: двумерный тор со сдвигом на 1 по границе по горизонтали;

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

CRAY 1 (1976): **векторно-конвейерные процессоры**. В 1972 году С.Крэй покидает CDC и основывает свою компанию Cray Research, которая в 1976г. выпускает первый векторно-конвейерный компьютер CRAY-1: время такта 12.5нс, 12 конвейерных функциональных устройств, пиковая производительность 160 миллионов операций в секунду, оперативная память до 1Мслова (слово - 64 разряда), цикл памяти 50нс. Главным новшеством является введение век-

торных команд, работающих с целыми массивами независимых данных и позволяющих эффективно использовать конвейерные функциональные устройства.

Иерархия памяти. Иерархия памяти прямого отношения к параллелизму не имеет, однако, безусловно, относится к тем особенностям архитектуры компьютеров, которые имеет огромное значение для повышения их производительности (сглаживание разницы между скоростью работы процессора и временем выборки из памяти). Основные уровни: регистры, кэш-память, оперативная память, дисковая память. Время выборки по уровням памяти от дисковой памяти к регистрам уменьшается, стоимость в пересчете на 1 слово (байт) растет. В настоящее время, подобная иерархия поддерживается даже на персональных компьютерах.

А что же сейчас используют в мире?

По каким же направлениям идет развитие высокопроизводительной вычислительной техники в настоящее время? Основных направлений четыре.

1. Векторно-конвейерные компьютеры. Конвейерные функциональные устройства и набор векторных команд - это две особенности таких машин. В отличие от традиционного подхода, векторные команды оперируют целыми массивами независимых данных, что позволяет эффективно загружать доступные конвейеры, т.е. команда вида $A = B + C$ может означать сложение двух массивов, а не двух чисел. Характерным представителем данного направления является семейство векторно-конвейерных компьютеров CRAY куда входят, например, CRAY EL, CRAY J90, CRAY T90 (в марте 2000 года американская компания TERA перекупила подразделение CRAY у компании Silicon Graphics, Inc.).

2. Массивно-параллельные компьютеры с распределенной памятью. Идея построения компьютеров этого класса тривиальна: возьмем серийные микропроцессоры, снабдим каждый своей локальной памятью, соединим посредством некоторой коммуникационной среды - вот и все. Достоинств у такой архитектуры масса: если нужна высокая производительность, то можно добавить еще процессоров, если ограничены финансы или заранее известна требуемая вычислительная мощность, то легко подобрать оптимальную конфигурацию и т.п.

Однако есть и решающий "минус", сводящий многие "плюсы" на нет. Дело в том, что межпроцессорное взаимодействие в компьютерах этого класса идет намного медленнее, чем происходит локальная обработка данных самими процессорами. Именно поэтому написать эффективную программу для таких компьютеров очень сложно, а для некоторых алгоритмов иногда просто невозможно. К данному классу можно отнести компьютеры Intel Paragon, IBM SP1, Parsytec, в какой-то степени IBM SP2 и CRAY T3D/T3E, хотя в этих компьютерах влияние указанного минуса значительно ослаблено. К этому же классу можно отнести и сети компьютеров, которые все чаще рассматривают как дешевую альтернативу крайне дорогим суперкомпьютерам.

3. Параллельные компьютеры с общей памятью. Вся оперативная память таких компьютеров разделяется несколькими одинаковыми процессорами. Это снимает проблемы предыдущего класса, но добавляет новые - число процессоров, имеющих доступ к общей памяти, по чисто техническим причинам нельзя сделать большим. В данное направление входят многие современные многопроцессорные SMP-компьютеры или, например, отдельные узлы компьютеров HP Exemplar и Sun StarFire.

4. Последнее направление, строго говоря, не является самостоятельным, а скорее представляет собой комбинации предыдущих трех. Из нескольких процессоров (традиционных или векторно-конвейерных) и общей для них памяти сформируем вычислительный узел. Если полученной вычислительной мощности не достаточно, то объединим несколько узлов высокоскоростными каналами. Подобную архитектуру называют кластерной, и по такому принципу построены CRAY SV1, HP Exemplar, Sun StarFire, NEC SX-5, последние модели IBM SP2 и другие. Именно это направление является в настоящее время наиболее перспективным для конструирования компьютеров с рекордными показателями производительности.

Использование параллельных вычислительных систем

К сожалению чудеса в жизни редко случаются. Гигантская производительность параллельных компьютеров и супер-ЭВМ с лихвой компенсируется сложностями их использования. Начнем с самых простых вещей. У вас есть программа и доступ, скажем, к 256-процессорному компьютеру. Что вы ожидаете? Да ясно что: вы вполне законно ожидаете, что программа будет выполняться в 256 раз быстрее, чем на одном процессоре. А вот как раз этого, скорее всего, и не будет.

Закон Амдала и его следствия

Предположим, что в вашей программе 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 числу операций в процессе выполнения). Крайние случаи в значениях f соответствуют полностью параллельным ($f = 0$) и полностью последовательным ($f = 1$) программам. Так вот, для того, чтобы оценить, какое ускорение S может быть получено на компьютере из p процессоров при данном значении f, можно воспользоваться законом Амдала:

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

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

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

Отсюда первый вывод - прежде, чем основательно переделывать код для перехода на параллельный компьютер (а любой суперкомпьютер, в частности, является таковым) надо основательно подумать. Если, оценив заложенный в программе алгоритм, вы поняли, что доля последовательных операций велика, то на значительное ускорение рассчитывать явно не приходится и нужно думать о замене отдельных компонент алгоритма. В ряде случаев последовательный характер алгоритма изменить не так сложно. Допустим, что в программе есть следующий фрагмент для вычисления суммы n чисел:

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

(можно тоже самое на любом другом языке).

По своей природе он строго последователен, так как на i-й итерации цикла требуется результат с $(i-1)$-й и все итерации выполняются одна за одной. Имеем 100% последовательных операций, а значит и никакого эффекта от использования параллельных компьютеров. Вместе с тем, выход очевиден. Поскольку в большинстве реальных программ (вопрос: а почему в большинстве, а не во всех?) нет существенной разницы, в каком порядке складывать числа, выберем иную схему сложения. Сначала найдем сумму пар соседних элементов: $a_1 + a_2$, $a_3 + a_4$, $a_5 + a_6$ и т.д. Заметим, что при такой схеме все пары можно складывать одновременно! На следующих шагах будем действовать абсолютно аналогично, получив вариант параллельного алгоритма.

Казалось бы, в данном случае все проблемы удалось разрешить. Но представьте, что доступные вам процессоры разнородны по своей производительности. Значит, будет такой момент, ко-

гда кто-то из них еще трудится, а кто-то уже все сделал и бесполезно простаивает в ожидании. Если разброс в производительности компьютеров большой, то и эффективность всей системы при равномерной загрузке процессоров будет крайне низкой.

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

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

Заключительный вопрос. Как вы думаете, верно ли утверждение: чем мощнее компьютер, тем быстрее на нем можно решить данную задачу?

Заключительный ответ. Нет, это не верно. Это можно пояснить простым бытовым примером. Если один землекоп выкопает яму 1м*1м*1м за 1 час, то два таких же землекопа это сделают за 30 мин - в это можно поверить. А за сколько времени эту работу сделают 60 землекопов? За 1 минуту? Конечно же нет! Начиная с некоторого момента они будут просто мешаться друг другу, не ускоряя, а замедляя процесс. Так же и в компьютерах: если задача слишком мала, то мы будем дольше заниматься распределением работы, синхронизацией процессов, сборкой результатов и т.п., чем непосредственно полезной работой.

Совершенно ясно, что не все так просто...

Лекция 2. Архитектура векторно-конвейерных супер-ЭВМ CRAY C90.

Общая структура компьютера CRAY Y-MP C90.

CRAY Y-MP C90 - это векторно-конвейерный компьютер, объединяющий в максимальной конфигурации 16 процессоров, работающих над общей памятью. Время такта компьютера CRAY Y-MP C90 равно 4.1 нс, что соответствует тактовой частоте почти 250MHz.

Разделяемые ресурсы процессора.

Структура оперативной памяти.

Оперативная память этого компьютера разделяется всеми процессорами и секцией ввода/вывода. Каждое слово состоит из 80-ти разрядов: 64 для хранения данных и 16 для коррекции ошибок. Для увеличения скорости выборки данных память разделена на множество банков, которые могут работать одновременно.

Каждый процессор имеет доступ к ОП через четыре порта с пропускной способностью два слова за один такт каждый, причем один из портов всегда связан с секций ввода/вывода и по крайней мере один из портов всегда выделен под операцию записи.

В максимальной конфигурации вся память разделена на 8 секций, каждая секция на 8 подсекций, каждая подсекция на 16 банков. Адреса идут с чередованием по каждому из данных параметров:

```
адрес 0 - в 0-й секции, 0-подсекции, 0-м банке,
адрес 1 - в 1-й секции, 0-подсекции, 0-м банке,
адрес 2 - в 2-й секции, 0-подсекции, 0-м банке,
   ...
адрес 8 - в 0-й секции, 1-подсекции, 0-м банке,
адрес 9 - в 1-й секции, 1-подсекции, 0-м банке,
   ...
адрес 63 - в 7-й секции, 7-подсекции, 0-м банке,
адрес 64 - в 0-й секции, 0-подсекции, 1-м банке,
адрес 65 - в 1-й секции, 0-подсекции, 1-м банке,
   ...
```

При одновременном обращении к одной и той же секции из разных портов возникает задержка в 1 такт, а при обращении к одной и той же подсекции одной секции задержка варьируется от 1 до 6 тактов. При выборке последовательно расположенных данных или при выборке с любым нечетным шагом конфликтов не возникает.

Секция ввода/вывода.

Компьютер поддерживает три типа каналов, которые различаются скоростью передачи:

- Low-speed (LOSP) channels - 6 Mbytes/s
- High-speed (HISP) channels - 200 Mbytes/s
- Very high-speed (VHISP) channels - 1800 Mbytes/s

Секция межпроцессорного взаимодействия.

Секция межпроцессорного взаимодействия содержит разделяемые регистры и семафоры, предназначенные для передачи данных и управляющей информации между процессорами. Регистры и семафоры разделены на одинаковые группы (кластеры), каждый кластер содержит 8 (32-разрядных) разделяемых адресных (SB) регистра, 8 (64-разрядных) разделяемых скалярных (ST) регистра и 32 однобитовых семафора.

Вычислительная секция процессора.

Все процессоры имеют одинаковую вычислительную секцию, состоящую из регистров, функциональных устройств (ФУ) и сети коммуникаций. Регистры и ФУ могут хранить и обрабатывать три типа данных: адреса (A -регистры, B -регистры), скаляры (S -регистры, T -регистры) и вектора (V -регистры).

Регистры.

Каждый процессор имеет три набора основных регистров (A, S, V), которые имеют связь как с памятью, так и с ФУ. Для регистров A и S существуют промежуточные наборы регистров B и T, играющие роль буферов для основных регистров.

Адресные регистры: A -регистры, 8 штук по 32 разряда, для хранения и вычисления адресов, индексации, указания величины сдвигов, числа итераций циклов и т.д. B -регистры, 64 штуки по 32 разряда.

Скалярные регистры: S -регистры, 8 штук по 64 разряда, для хранения аргументов и результатов скалярной арифметики, иногда содержат операнд для векторных команд. T -регистры, 64 штуки по 64 разряда. Скалярные регистры используются для выполнения как скалярных, так и векторных команд.

Векторные регистры: V -регистры, 8 штук на 128 64-разрядных слова каждый. Векторные регистры используются только для выполнения векторных команд.

Регистр длины вектора: 8 разрядов.

Регистр маски вектора: 128 разрядов.

Функциональные устройства.

ФУ исполняют свой набор команд и могут работать одновременно друг с другом. Все ФУ конвейерные и делятся на четыре группы: адресные, скалярные, векторные и для работы с плавающей точкой.

Адресные ФУ (2): целочисленное сложение/вычитание, целочисленное умножение.

Скалярные ФУ (4): целочисленное сложение/вычитание, логические поразрядные операции, сдвиг, число единиц/число нулей до первой единицы.

Векторные ФУ (5-7): целочисленное сложение/вычитание, сдвиг, логические поразрядные операции (1-2), число единиц/число нулей до первой единицы (1-2), умножение битовых матриц (0-1). Предназначены для выполнения только векторных команд.

ФУ с плавающей точкой (3): сложение/вычитание, умножение, нахождение обратной величины. Предназначены для выполнения как векторных, так и скалярных команд.

Векторные ФУ и ФУ с плавающей точкой продублированы: векторные команды разбивают 128 элементов векторных регистров на четные и нечетные, обрабатываемые одновременно двумя конвейерами (pipe 0, pipe 1). Когда завершается выполнение очередной пары операций результаты записываются на соответствующие четные и нечетные позиции выходного регистра. В

полностью скалярных операциях, использующих ФУ с плавающей точкой, работает только один конвейер.

ФУ имеют различное число ступеней конвейера, но каждая ступень срабатывает за один такт, поэтому при полной загрузке все ФУ могут выдавать результат каждый такт.

Секция управления процессора.

Команды выбираются из ОП блоками и заносятся в буфера команд, откуда они затем выбираются для исполнения. Если необходимой для исполнения команды нет в буферах команд, то происходит выборка очередного блока.

Команды имеют различный формат и могут занимать 1 пакет (16 разрядов), 2 пакета или 3 пакета (в одном слове 64 разряда, следовательно, в слове содержится 4 пакета). Максимальная длина программы на CRAY C90 равна 1 Гигаслову.

Параллельное выполнение программ.

Конвейеризация выполнения команд.

Все основные операции, выполняемые процессором: обращения в память, обработка команд и выполнение инструкций являются конвейерными.

Независимость функциональных устройств.

Большинство ФУ в CRAY C90 являются независимыми, поэтому несколько операций могут выполняться одновременно. Для операции $A = (B + C) * D * E$ порядок выполнения может быть следующим (все аргументы загружены в S регистры). Генерируются три инструкции: умножение D и E, сложение B и C и умножение результатов двух предыдущих операций. Первые две операции выполняются одновременно, затем третья.

Векторная обработка.

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

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

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

Перед тем, как векторная операция начнет выдавать результаты, проходит некоторое время (startup), связанное с заполнением конвейера и подкачкой аргументов. Чем больше длина векторов, тем менее заметным оказывается влияние данного начального промежутка времени на все время выполнения программы.

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

Зацепление функциональных устройств.

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

Многопроцессорная обработка: multiprogramming, multitasking

Multiprogramming - это выполнение нескольких независимых программ на различных процессорах.

Multitasking - это выполнение одной программы на нескольких процессорах.

Пиковая производительность CRAY Y-MP C90.

Пиковая производительность компьютера CRAY Y-MP C90 вычисляется так: функциональные устройства выдают два результата каждый такт (сдвоенные конвейеры), зацепление сложения и умножения дает четыре операции за такт, что составляет почти 1 Гфлопс (10^9 опер/с). Если работают все 16 процессоров, то 16 Гфлопс.

Лекция 3. Легко ли достичь пиковой производительности компьютера CRAY C90?

Понятие о векторизации программы.

Вектора данных, вектора в программах.

Вектор данных - это упорядоченный набор данных, размещенных в памяти на одинаковом "расстоянии" друг от друга.

Примерами векторов в программах могут служить строки, столбцы, диагонали, массивы целиком, в тоже время поддиагональная часть двумерной матрицы вектором не является.

Возможность векторной обработки программ.

Некоторый фрагмент программы может быть обработан в векторном режиме, если для его выполнения могут быть использованы векторные команды (соответственно полная или частичная векторизация). Поиск таких фрагментов в программе и их замена на векторные команды называется ***векторизацией программы***. Для векторизации необходимы вектора-аргументы + независимые операции над ними. Кандидаты для векторизации - это самые внутренние циклы программы.

Пример. Нужно выполнить независимую обработку всех элементов поддиагональной части массива; в этом случае можем векторизовать по строкам, можем по столбцам, но не можем обработать все данные сразу в векторном режиме из-за нерегулярности расположения элементов поддиагональной части массива в памяти.

Пример векторизуемого фрагмента, для которого выполнены все указанные условия:

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

Пример невекторизуемого фрагмента (очередная итерация не может начаться, пока не закончится предыдущая):

```
Do i=1,n
   A(i) = A(i-1)+s
End Do
```

Препятствия для векторизации.

Препятствий для векторизации конкретного цикла может быть много, вот лишь некоторые из них:

- Зависимость по данным (предыдущий фрагмент).
- Отсутствие регулярно расположенных векторов:

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

- Присутствие цикла, вложенного в данный - для реализации такого фрагмента нет соответствующих векторных команд.
- Вызов неизвестных подпрограмм и функций:

```
Do i=1,n
   CALL SUBR(A,B)
End Do
```

Анализ узких мест в архитектуре компьютера CRAY C90 (один процессор).

Анализ факторов, снижающих реальную производительность компьютеров, начнем с обсуждения известного ***закона Амдала***. Смысл его сводится к тому, что время работы программы определяется ее самой медленной частью. В самом деле, предположим, что одна половина некоторой программы - это сугубо последовательные вычисления. Тогда вне всякой зависимости от свойств другой половины, которая может идеально векторизоваться либо вообще выполняться мгновенно, ускорения работы всей программы более чем в два раза мы не получим.

Влияние данного фактора надо оценивать с двух сторон. Во-первых, по природе самого алгоритма все множество операций программы Γ разбивается на последовательные операции Γ_1 и операции Γ_2, исполняемые в векторном режиме. Если доля последовательных операций велика, то программист сразу должен быть готов к тому, что большого ускорения он никакими средствами не получит и, быть может, следует уже на этом этапе подумать об изменении алгоритма.

Во-вторых, не следует сбрасывать со счетов и качество компилятора, который может не распознать векторизуемость отдельных конструкций и, тем самым, часть "потенциально хороших" операций из Γ_2 перенести в Γ_1.

Прекрасной иллюстрацией к действию закона Амдала могут служить наши эксперименты, которые мы проводили с алгоритмом компактного размещения данных. Задача ставилась следующим образом: l разрядов из каждого элемента массива $A(i)$ надо плотно упаковать по элементам массива B. Если очередная секция из l разрядов целиком не помещается в очередном элементе $B(j)$, то оставшаяся часть должна быть перенесена в следующий элемент $B(j+1)$ (примерная схема данного преобразования показана на рис.1). Общее число элементов массива A было порядка одного миллиона.

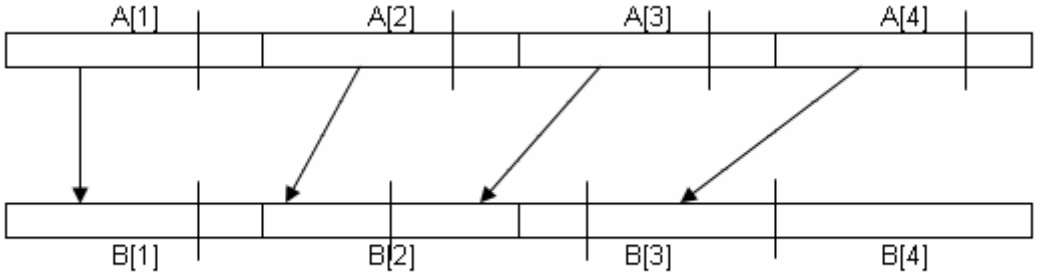


Рис. 1. Примерная схема сжатия данных

Первоначально был реализован самый простой алгоритм, который перебирает по очереди все элементы массива A, для каждого выделяет битовую секцию из l разрядов, записывает в соответствующий элемент $B(j)$ и, если необходимо, увеличивает j на единицу, записывая остаток из текущей секции в следующий элемент массива B. Алгоритм в таком виде не мог быть векторизован, выполнялся в скалярном режиме, а значит не мог в принципе получить никакого выигрыша от параллельной структуры компьютера CRAY Y-MP C90.

Немного подумав, нетрудно понять, каким образом можно модифицировать данный алгоритм и сделать его основную часть векторизуемой. Ясно, что, начиная с некоторого номера i_0, смещения пакуемых секций всех элементов массива A в массиве B будут повторяться, причем будут повторяться циклически с периодом не более, чем 63 (число разрядов в слове данного компьютера равно 64, но знаковый разряд в каждом элементе $B(j)$ надо было пропускать). По-

этому битовые секции всех элементов $A(i_{beg}+i*63)$, $i=0,1,2,...$ для каждого фиксированного значения $i_{beg}=1,2,...,63$ будут расположены, начиная с одного и того же смещения в элементах массива B, причем эти элементы массива B будут расположены так же с некоторым фиксированным шагом! Становиться понятной общая схема нового алгоритма: упаковку первых 63 элементов делаем как и прежде в скалярном режиме, запоминая смещения в элементах массива B и определяя шаг по этому массиву, а затем в векторном режиме располагаем секции элементов $A(i_{beg}+i*63)$, $i=0,1,2,...$ для каждого значения i_{beg} отдельно. После реализации второго алгоритма оказалось, что время его работы в 30 раз (!) меньше времени работы первого алгоритма. Основная причина очевидна - значительное увеличение доли операций Γ_2, выполняемых в векторном режиме.

Примерная схема первого варианта (каждая итерация ждет окончания предыдущей для определения смещения в слове массива B):

- выделение битовой секции из $A(i)$;
- запись в $B(j)$ с позиции p;
- если вся секция в $B(j)$ не поместилось, то переносим остаток в $B(j+1)$, $j=j+1$;
- модификация p, $i=i+1$;
- следующий шаг.

Примерная схема векторного варианта:

- определение смещений ($p[i_{beg}]$) для первых 63 элементов массива A - для каждого $i_{beg}=1,2,...,63$;
- выделение битовой секции из A(i*63+ibeg) ;
- запись очередной секции в $B(j)$ с позиции $p[i_{beg}]$;
- $i=i+1$, $j=j+b_{step}$.

Следующие два фактора, снижающие реальную производительность (они же определяют невозможность достижения пиковой производительности) - ***секционирование длинных векторных операций*** на порции по 128 элементов и ***время начального разгона конвейера***, относятся к накладным расходам на организацию векторных операций на конвейерных функциональных устройствах. Временем начального разгона конвейера, в частности, определяется тот факт, что очень короткие циклы выгоднее выполнять не в векторном режиме, а в скалярном, когда этих накладных расходов нет.

В отличие от секционирования операций дополнительное время на разгон конвейера требуется лишь один раз при старте векторной операции. Это стимулирует к работе с длинными векторами данных, так как с ростом длины вектора доля накладных расходов в общем времени выполнения операции быстро падает. Рассмотрим следующий фрагмент программы:

```
Do i=1,n
   a(i)=b(i)*s + c(i)
End Do
```

В табл.1 показана зависимость производительности компьютера CRAY Y-MP C90 на данном фрагменте от длины вектора, т.е. от значения n (указанные значения производительности здесь и далее могут на практике немного меняться в зависимости от текущей загрузки компьютера и некоторых других факторов - их нужно рассматривать скорее как некоторый ориентир).

длина вектора	производительность Mflops/s	длина вектора	производительность Mflops/s
1	7.0	150	413.2
2	14.0	256	548.0
4	27.6	257	491.0

16	100.5	512	659.2
32	181.9	1024	720.4
64	301.0	2048	768.0
128	433.7	8192	802.0

Табл. 1. Производительность CRAY Y-MP C90 на операции

$$a_i = b_i * s + c_i .$$

Конфликты при обращении в память у компьютеров CRAY Y-MP полностью определяются аппаратными особенностями организации доступа к оперативной памяти. Память компьютеров CRAY Y-MP C90 в максимальной конфигурации разделена на 8 секций, каждая секция - на 8 подсекций, а каждая подсекция на 16 банков памяти. Ясно, что наибольшего времени на разрешение конфликтов потребуется при выборке данных с шагом 8*8=64, когда постоянно совпадают номера и секций и подсекций. С другой стороны, выборка с любым нечетным шагом проходит без конфликтов вообще, и в этом смысле она эквивалентна выборке с шагом единица. Возьмем следующий пример:

```
Do i=1,n*k,k
a(i)=b(i)*s + c(i)
End Do
```

В зависимости от значения k, т.е. шага выборки данных из памяти, происходит выполнение векторной операции $a_i = b_i * s + c_i$ длины n в режиме с зацеплением. Производительность компьютера (с двумя секциями памяти) на данной операции показана в таблице 2.

шаг по памяти	производительность на векторах из		
	100 элементов	1000 элементов	12800 элементов
1	240.3	705.2	805.1
2	220.4	444.6	498.5
4	172.9	274.6	280.1
8	108.1	142.8	147.7
16	71.7	84.5	86.0
32	41.0	44.3	38.0
64	22.1	25.7	22.3
128	21.2	20.6	20.3

Табл. 2. Влияние конфликтов при обращении к памяти: производительность компьютера CRAY Y-MP C90 в зависимости от длины векторов и шага перемещения по памяти.

Как видим, производительность падает катастрофически. Однако еще одной неприятной стороной конфликтов является то, что "внешних" причин их появления может быть много (в то время как истинная причина, конечно же, одна - неудачное расположение данных). В самом деле, в предыдущем примере конфликты возникали при использовании цикла с нееденичным четным шагом. Значит, казалось бы, не должно быть никаких причин для возникновения конфликтов при работе фрагмента следующего вида:

```
Do i=1,n
Do j=1,n
Do k=1,n
   X(i,j,k) = X(i,j,k)+P(k,i)*Y(k,j)
End Do
End Do
End Do
```

Однако это не совсем так и все зависит от того, каким образом описан массив X. Предположим, что описание имеет вид:

```
DIMENSION X(40,40,100)
```

По определению Фортрана массивы хранятся в памяти "по столбцам", следовательно при изменении последнего индексного выражения на единицу реальное смещение по памяти будет равно произведению размеров массива по предыдущим размерностям. Для нашего примера, расстояние между соседними элементами $X(i,j,k)$ и $X(i,j,k+1)$ равно 40*40=1600=25*64, т.е. всегда кратно наихудшему шагу для выборки из памяти. В тоже время, если изменить лишь описание массива, добавив единицу к первым двум размерностям:

```
DIMENSION X(41,41,100),
```

то никаких конфликтов не будет вовсе.

Последний пример возможного появления конфликтов - это использование косвенной адресации. В следующем фрагменте

```
Do j=1,n
Do i=1,n
   XYZ(IX(i),j) = XYZ(IX(i),j)+P(i,j)*Y(i,j)
End Do
End Do
```

в зависимости от того, к каким элементам массива XYZ реально происходит обращение, конфликтов может не быть вовсе (например, $IX(i)$ равно i) либо их число может быть максимальным (например, $IX(i)$ равно одному и тому же значению для всех i).

Следующие два фактора, снижающие производительность, определяются тем, что перед началом выполнения любой операции данные должны быть занесены в регистры. Для этого в архитектуре компьютера CRAY Y-MP предусмотрены три независимых канала передачи данных, два из которых могут работать на чтение из памяти, а третий на запись. Такая структура хорошо подходит для операций, требующих не более двух входных векторов для выполнения в максимально производительном режиме с зацеплением, например, $A = B * s + C$.

Однако операции с тремя векторными аргументами, как, например, $A = B * C + D$, не могут быть реализованы столь же оптимально. Часть времени будет неизбежно потрачено впустую на ожидание подкачки третьего аргумента для запуска операции с зацеплением, что является прямым следствием ***ограниченной пропускной способности каналов передачи данных*** (memory bottleneck). С одной стороны, максимальная производительность достигается на операции с зацеплением, требующей три аргумента, а с другой на чтение одновременно могут работать лишь два канала. В таблице 3 приведена производительность компьютера на указанной выше векторной операции, требующей три входных вектора B, C, D, в зависимости от их длины.

длина вектора	производительность Mflop/s
10	57.0
100	278.3
1000	435.3
12801	445.0

Табл. 3. Производительность CRAY Y-MP C90 на операции $a_i = b_i * c_i + d_i$.

Теперь предположим, что пропускная способность каналов не является узким местом. В этом случае на предварительное занесение данных в регистры все равно требуется некоторое дополнительное время. Поскольку нам ***необходимо использовать векторные регистры перед выполнением операций***, то требуемые для этого операции чтения/записи будут неизбежно снижать общую производительность. Довольно часто влияние данного фактора можно заметно ослабить, если повторно используемые вектора один раз загрузить в регистры, выполнить постро-

енные на их основе выражения, а уже затем перейти к оставшейся части программы. Рассмотрим следующий фрагмент:

```
Do j=1,120
Do i=1,n
   DP(i) = DP(i) + s*P(i,j-1) + t*P(i,j)
End Do
End Do
```

На каждой итерации по j для выполнения векторной операции требуются три входных вектора $DP(i)$, $P(i, j-1)$, $P(i, j)$ и один выходной - $DP(i)$, следовательно, за время работы всего фрагмента будет выполнено 120*3=360 операций чтения векторов и 120 операций записи. Если явно выписать каждые две последовательные итерации цикла по j и преобразовать фрагмент к виду:

```
Do j=1,120,2
Do i=1,n
   DP(i) = DP(i)+s*P(i,j-1)+t*P(i,j)+s*P(i,j)+t*P(i,j+1)
End Do
End Do
```

то на каждой из 60-ти итераций внешнего цикла потребуется четыре входных вектора $DP(i)$, $P(i, j-1)$, $P(i, j)$, $P(i, j+1)$ и опять же один выходной. Суммарно, для нового варианта будет выполнено 60*4=240 операций чтения и 60 операций записи. Преобразование подобного рода носит название "раскрутки" и имеет максимальный эффект в том случае, когда на соседних итерациях цикла используются одни и те же данные.

Теоретически, одновременно с увеличением глубины раскрутки растет и производительность, приближаясь в пределе к некоторому значению. Однако на практике максимальный эффект достигается где-то на первых шагах, а затем производительность либо остается примерно одинаковой, либо падает. Основная причина данного несоответствия теории и практики заключается в том, что компьютеры CRAY Y-MP C90 имеют сильно ***ограниченный набор векторных регистров***: 8 регистров по 128 слова в каждом. Как правило, любая раскрутка требует подкачки дополнительных векторов, а следовательно, и дополнительных регистров. Таблица 4 содержит данные по производительности CRAY Y-MP C90 на обсуждаемом выше фрагменте в зависимости от длины векторов и глубины раскрутки.

глубина раскрутки	производительность на векторах из		
	64 элементов	128 элементов	12800 элементов
1	464.4	612.9	749.0
2	591.4	731.6	730.1
3	639.3	780.7	752.5
4	675.3	807.7	786.8

Табл. 4. Зависимость производительности CRAY Y-MP C90 от глубины раскрутки и длины векторов.

Теперь вспомним, что значение пиковой производительности вычислялось при условии одновременной работы всех функциональных устройств. Значит, если некоторый алгоритм выполняет одинаковое число операций сложения и умножения, но все сложения выполняются сначала и лишь затем операции умножения, то в каждый момент времени в компьютере будут задействованы только устройства одного типа. Присутствующая ***несбалансированность в использовании функциональных устройств*** является серьезным фактором, сильно снижающим реальную производительность компьютера - соответствующие данные можно найти в таблице 5.

В наборе функциональных устройств ***нет устройства деления***. Для выполнения данной операции используется устройство обратной аппроксимации и устройство умножения. Отсюда

сразу следует, что, во-первых, производительность фрагмента в терминах операций деления будет очень низкой и, во-вторых, использование деления вместе с операцией сложения немного выгоднее, чем с умножением. Конкретные значения производительности показаны в таблице 5.

длина вектора	производительность на операции				
	$a_i = b_i + c_i$	$a_i = b_i * c_i$	$a_i = b_i / c_i$	$a_i = s / b_i + t$	$a_i = s / b_i * t$
10	35.5	41.9	24.8	45.7	46.1
100	202.9	198.0	88.4	197.4	166.5
1000	343.8	341.2	117.2	283.8	215.9
12800	373.1	376.8	120.0	297.0	222.5

Табл. 5. Производительность CRAY Y-MP C90 на операциях одного типа и операциях с делением.

Если структура программы такова, что в ней либо происходит частое обращение к различным небольшим подпрограммам и функциям, либо структура управления очень запутана и построена на основе большого числа переходов, то потребуется частая ***перезагрузка буферов команд***, а, значит, возникнут дополнительные накладные расходы. Наилучший результат достигается в том случае, если весь фрагмент кода уместился в одном буфере команд. Незначительные потери производительности будут у фрагментов, расположенных в нескольких буферах. Если же перезагрузка частая, т.е. фрагмент или программа обладают малой локальностью вычислений, то производительность может изменяться в очень широких пределах в зависимости от способа организации каждой конкретной программы.

Суммарное влияние отрицательных факторов на производительность компьютера.

Что следует из проведенного анализа архитектуры суперкомпьютера CRAY Y-MP C90? Выводов можно сделать много, но главный помогает сразу понять ***причину низкой производительности неоптимизированных программ***, в частности программ, перенесенных с традиционных последовательных компьютеров. Дело в том, что на производительность реальных программ одновременно оказывают влияние в той или иной степени одновременно ***ВСЕ перечисленные выше факторы*.** В самом деле, программы не бывают векторизуемыми на все 100% --- всегда есть некоторая инициализация, ввод/вывод или что-то подобное. Вместе с этим обязательно будет присутствовать какое-то число конфликтов в памяти плюс, быть может легкая, несбалансированность в использовании функциональных устройств. Для части операций может не хватать каналов чтения/записи, векторных регистров и т.д. по всем изложенным выше факторам (плюс влияние компилятора - что-то не векторизовал, что-то сделал не эффективно и т.д.).

Если предположить, что влияние каждого отдельного фактора в некоторой программе таково, что позволяет достичь 85% пиковой производительности, то их суммарное воздействие снизит реальную производительность менее, чем до 20% от пика!

Вывод: если хотим добиться хорошей производительности компьютера, то принимать во внимание необходимо все указанные выше факторы одновременно, минимизируя их суммарное проявление в программе.

Лекция 4. Архитектура массивно-параллельных компьютеров (на примере CRAY T3D). Особенности программирования.

Массивно-параллельные компьютеры. Общие черты.

Основные причины появления массивно-параллельных компьютеров - это, во-первых, необходимость построения компьютеров с гигантской производительностью, и, во-вторых, необходимость производства компьютеров в большом диапазоне как производительности, так и стоимости. Не все в состоянии купить однопроцессорный CRAY Y-MP C90, да и не всегда такие мощности и нужны. Для массивно-параллельного компьютера, в котором число процессоров

может сильно меняться, всегда можно подобрать конфигурацию с заранее заданной производительностью и/или стоимостью.

С некоторой степенью условности, массивно-параллельные компьютеры можно характеризовать следующими параметрами:

- используемые микропроцессоры: Intel Paragon - i860, IBM SP2 - PowerPC 604e или Power2 SC, CRAY T3D - DEC ALPHA;
- коммуникационная сеть: Intel Paragon - двумерная прямоугольная решетка, IBM SP2 - коммутатор, CRAY T3D - трехмерный тор;
- организация памяти: Intel Paragon, IBM SP2, CRAY T3D - распределенная память;
- наличие или отсутствие host-компьютера: Intel Paragon, IBM SP2 - нет, CRAY T3D - есть.

Общая структура компьютера CRAY T3D.

Компьютер CRAY T3D - это массивно-параллельный компьютер с распределенной памятью, объединяющий от 32 до 2048 процессоров. Распределенность памяти означает то, что каждый процессор имеет непосредственный доступ только к своей локальной памяти, а доступ к данным, расположенным в памяти других процессоров, выполняется другими, более сложными способами.

CRAY T3D подключается к хост-компьютеру (главному или ведущему), роль которого, в частности, может исполнять CRAY Y-MP C90. Вся предварительная обработка и подготовка программ, выполняемых на CRAY T3D, проходит на хосте (например, компиляция). Связь хост-машины и T3D идет через высокоскоростной канал передачи данных с производительностью 200 Мбайт/с.

Массивно-параллельный компьютер CRAY T3D работает на тактовой частоте 150MHz и имеет в своем составе три основные компоненты: сеть межпроцессорного взаимодействия (или по-другому коммуникационную сеть), вычислительные узлы и узлы ввода/вывода.

Вычислительные узлы и процессорные элементы.

Вычислительный узел состоит из двух процессорных элементов (ПЭ), сетевого интерфейса контроллера блочных передач. Оба процессорных элемента, входящие в состав вычислительного узла, идентичны и могут работать независимо друг от друга.

Процессорный элемент. Каждый ПЭ содержит микропроцессор, локальную память и некоторые вспомогательные схемы.

Микропроцессор - это 64-х разрядный RISC (Reduced Instruction Set Computer) процессор ALPHA фирмы DEC, работающий на тактовой частоте 150 MHz. Микропроцессор имеет внутреннюю кэш-память команд и кэш-память данных.

Объем ***локальной памяти*** ПЭ - 8 Мслов. Локальная память каждого процессорного элемента является частью физически распределенной, но логически разделяемой (или общей), памяти всего компьютера. В самом деле, память физически распределена, так как каждый ПЭ содержит свою локальную память. В тоже время, память разделяется всеми ПЭ, так как каждый ПЭ может обращаться к памяти любого другого ПЭ, не прерывая его работы.

Обращение к памяти другого ПЭ лишь в 6 раз медленнее, чем обращение к своей собственной локальной памяти.

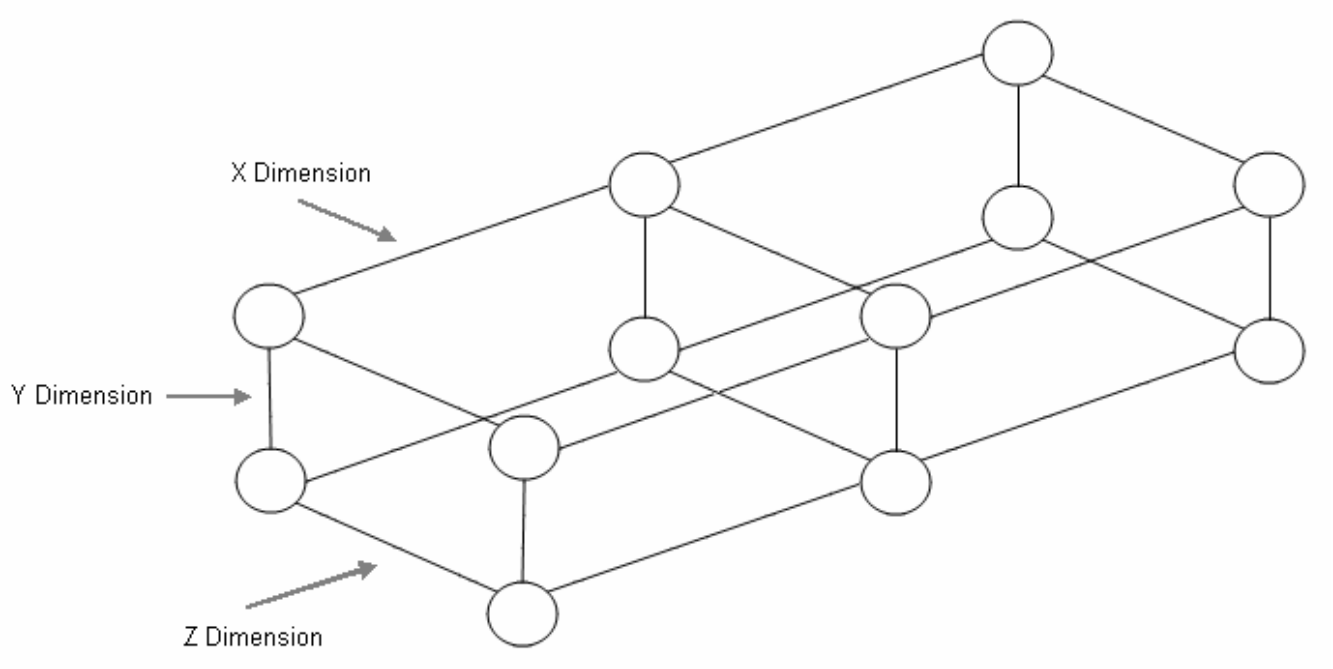
Сетевой интерфейс формирует передачи перед посылкой через коммуникационную сеть другим вычислительным узлам или узлам ввода/вывода, а также принимает приходящие сообщения и распределяет их между двумя процессорными элементами узла.

Контроллер блочных передач - это контроллер асинхронного прямого доступа в память, который помогает перераспределять данные, расположенные в локальной памяти разных ПЭ компьютера CRAY T3D, без прерывания работы самих ПЭ.

Коммуникационная сеть.

Коммуникационная сеть обеспечивает передачу информации между вычислительными узлами и узлами ввода/вывода с максимальной скоростью в 140М байт/с. Сеть образует трехмер-

ную решетку, соединяя сетевые маршрутизаторы узлов в направлениях X, Y, Z. Каждая элементарная связь между двумя узлами - это два однонаправленных канала передачи данных, что допускает одновременный обмен данными в противоположных направлениях.

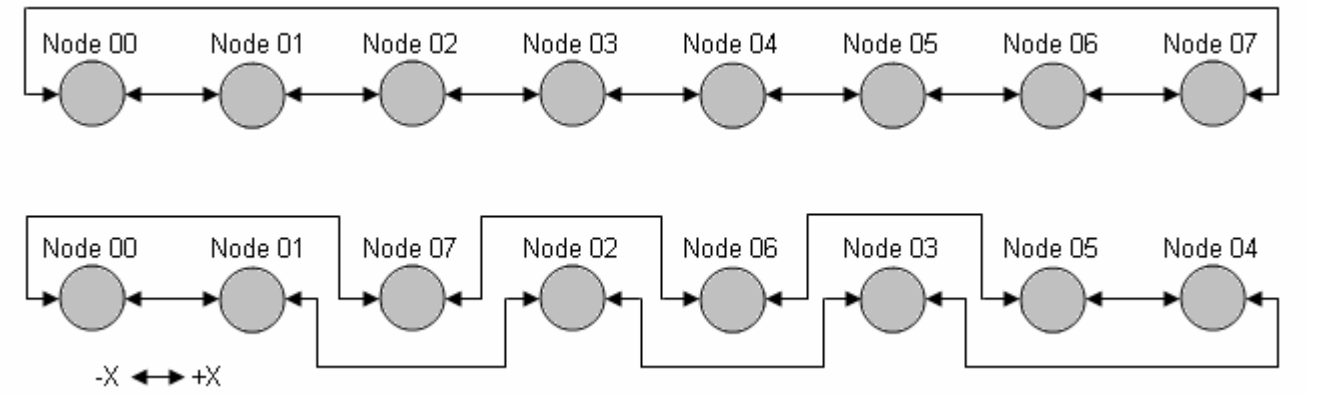


Топология сети, чередование вычислительных узлов

Коммуникационная сеть компьютера CRAY T3D организована в виде двунаправленного трехмерного тора, что имеет свои преимущества перед другими способами организации связи:

- быстрая связь граничных узлов и небольшое среднее число перемещений по тору при взаимодействии разных ПЭ: максимальное расстояние в сети для конфигурации из 128 ПЭ равно 6, а для 2048 ПЭ равно 12;
- возможность выбора другого маршрута для обхода поврежденных связей.

Все узлы в коммуникационной сети в размерностях X и Z расположены с чередованием, что позволяет минимизировать длину максимального физического соединения между ПЭ.



Маршрутизация в сети и сетевые маршрутизаторы.

При выборе маршрута для обмена данными между двумя узлами сетевые маршрутизаторы всегда сначала выполняют смещение по размерности X, затем по Y, а в конце по Z. Так как смещение может быть как положительным, так и отрицательным, то этот механизм помогает минимизировать число перемещений по сети и обойти поврежденные связи.

Сетевые маршрутизаторы каждого вычислительного узла определяют путь перемещения каждого пакета и могут осуществлять параллельный транзит данных по каждому из трех измерений X, Y, Z.

Нумерация вычислительных узлов.

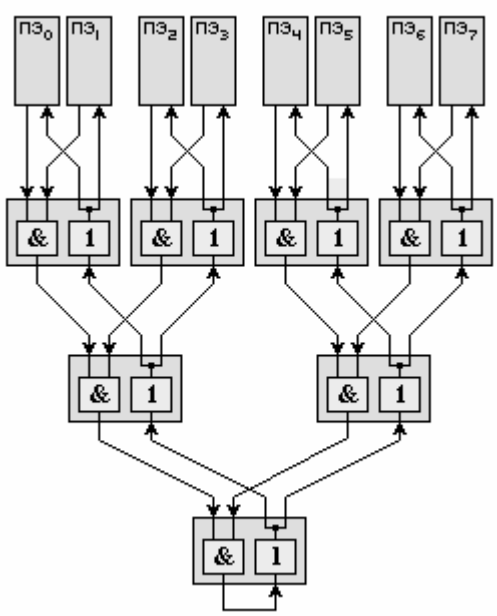
Каждому ПЭ в системе присвоен уникальный ***физический номер***, определяющий его физическое расположение, который и используется непосредственно аппаратурой.

Не обязательно все физические ПЭ принимают участие в формировании логической конфигурации компьютера. Например, 512-процессорная конфигурация компьютера CRAY T3D реально содержит 520 физических ПЭ, 8 из которых находятся в резерве. Каждому физическому ПЭ присваивается ***логический номер***, определяющий его расположение в логической конфигурации компьютера, которая уже и образует трехмерный тор.

Каждой программе пользователя из трехмерной решетки вычислительных узлов выделяется отдельный раздел, имеющий форму прямоугольного параллелепипеда, на котором работает только данная программа (не считая компонент ОС). Для последовательной нумерации ПЭ, выделенных пользователю, вводится ***виртуальная нумерация***.

Особенности синхронизации процессорных элементов.

Для поддержки синхронизации процессорных элементов предусмотрена аппаратная реализация одного из наиболее "тяжелых" видов синхронизации - барьеров синхронизации. ***Барьер*** - это точка в программе, при достижении которой каждый процессор должен ждать до тех пор, пока остальные также не дойдут до барьера, и лишь после этого момента все процессы могут продолжать работу дальше.



В схемах поддержки каждого ПЭ предусмотрены два 8-ми разрядных регистра, причем каждый разряд регистров соединен со своей независимой цепью реализации барьера (всего 16 независимых цепей). Каждая цепь строится на основе схем AND и ДУБЛИРОВАНИЕ (1-2). До барьера соответствующие разряды на всех ПЭ обнуляются, а как только процесс на ПЭ доходит до барьера, то записывает в свой разряд единицу. На выходе схемы AND появляется единица только в том случае, когда на обоих входах выставлены 1. Устройство 1-2 просто дублирует свой вход на выходы:

Если схемы AND заменить на OR, то получится цепь для реализации механизма "эврика": как только один ПЭ выставил значение 1, эта единица распространяется всем ПЭ, сигнализируя о некотором событии на ПЭ. Это исключительно полезно, например, в задачах поиска.

Факторы, снижающие производительность параллельных компьютеров.

Начнем с уже упоминавшегося закона Амдала. Для массивно-параллельных компьютеров он играет еще большую роль, чем для векторно-конвейерных. В самом деле, в таблице 6 показано, на какое максимальное ускорение работы программы можно рассчитывать в зависимости от доли последовательных вычислений и числа доступных процессоров. Предполагается, что параллельная секция может быть выполнена без каких-либо дополнительных накладных расходов. Так как в программе всегда присутствует инициализация, ввод/вывод и некоторые сугубо последовательные действия, то недооценивать данный фактор никак нельзя - практически вся программа должна исполняться в параллельном режиме, что можно обеспечить только после анализа всей (!) программы.

число ПЭ	для последовательных вычислений				
	50%	25%	10%	5%	2%
2	1.33	1.60	1.82	1.90	1.96
8	1.78	2.91	4.71	5.93	7.02
32	1.94	3.66	7.80	12.55	19.75

512	1.99	3.97	9.83	19.28	45.63
2048	2.00	3.99	9.96	19.82	48.83

Табл. 6. Максимальное ускорение работы программы в зависимости от доли последовательных вычислений и числа используемых процессоров.

Поскольку CRAY T3D - это компьютер с распределенной памятью, то взаимодействие процессоров, в основном, осуществляется посредством передачи сообщений друг другу. Отсюда два других замедляющих фактора - ***время инициализации посылки сообщения*** (латентность) и собственно ***время передачи сообщения по сети***. Максимальная скорость передачи достигается на больших сообщениях, когда латентность, возникающая лишь в начале, не столь заметна на фоне непосредственно передачи данных.

Возможность асинхронной посылки сообщений и вычислений. Если или аппаратура, или программное обеспечение не поддерживают возможности проводить вычислений на фоне пересылок, то возникнут неизбежные накладные расходы, связанные с ожиданием полного завершения взаимодействия параллельных процессов.

Для достижения эффективной параллельной обработки необходимо добиться ***равномерной загрузки всех процессоров***. Если равномерности нет, то часть процессоров неизбежно будет простаивать, ожидая остальных, хотя в этот момент они могли бы выполнять полезную работу. Иногда равномерность получается автоматически, например, при обработке прямоугольных матриц достаточно большого размера, однако уже при переходе к треугольным матрицам добиться хорошей равномерности не так просто.

Если один процессор должен вычислить некоторые данные, которые нужны другому процессору, и если второй процесс первым дойдет до точки приема соответствующего сообщения, то он с неизбежностью будет простаивать, ожидая передачи. Для того чтобы минимизировать ***время ожидание прихода сообщения*** первый процесс должен отправить требуемые данные как можно раньше, отложив независящую от них работу на потом, а второй процесс должен выполнить максимум работы, не требующей ожидаемой передачи, прежде, чем выходить на точку приема сообщения.

Чтобы не сложилось совсем плохого впечатления о массивно-параллельных компьютерах, надо заканчивать с негативными факторами, потому последний фактор - это ***реальная производительность одного процессора***. Разные модели микропроцессоров могут поддерживать несколько уровней кэш-памяти, иметь специализированные функциональные устройства, регистровую структуру и т.п. Каждый микропроцессор, в конце концов, может иметь векторно-конвейерную архитектуру и в этом случае ему присущи практически все те факторы, которые мы обсуждали в лекции, посвященной особенностям программирования векторно-конвейерных компьютеров.

К сожалению, на работу каждой конкретной программы сказываются в той или иной мере все эти факторы одновременно, дополнительно усугубляя ситуацию с эффективностью параллельных программ. Однако в отличие от векторно-конвейерных компьютеров все изложенные здесь факторы, за исключением быть может последнего, могут снизить производительность не в десятки, а в сотни и даже тысячи раз по сравнению с пиковыми показателями производительности компьютера. Добиться на этих компьютерах, в принципе, можно многого, но усилий это может потребовать во многих случаях очень больших.

Лекция 5. Технологии параллельного программирования. Message Passing Interface (MPI).

MPI. Терминология и обозначения.

MPI - message passing interface - библиотека функций, предназначенная для поддержки работы параллельных процессов в терминах передачи сообщений.

Номер процесса - целое неотрицательное число, являющееся уникальным атрибутом каждого процесса.

Атрибуты сообщения - номер процесса-отправителя, номер процесса-получателя и идентификатор сообщения. Для них заведена структура *MPI_Status*, содержащая три поля: *MPI_Source* (номер процесса отправителя), *MPI_Tag* (идентификатор сообщения), *MPI_Error* (код ошибки); могут быть и добавочные поля.

Идентификатор сообщения (msgtag) - атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от 0 до 32767.

Процессы объединяются в *группы*, могут быть вложенные группы. Внутри группы все процессы перенумерованы. С каждой группой ассоциирован свой *коммуникатор*. Поэтому при осуществлении пересылки необходимо указать идентификатор группы, внутри которой производится эта пересылка. Все процессы содержатся в группе с предопределенным идентификатором *MPI_COMM_WORLD*.

При описании процедур MPI будем пользоваться словом OUT для обозначения "выходных" параметров, т.е. таких параметров, через которые процедура возвращает результаты.

Общие процедуры MPI.

```
int MPI_Init (int* argc,
              char*** argv)
```

MPI_Init - инициализация параллельной части приложения. Реальная инициализация для каждого приложения выполняется не более одного раза, а если MPI уже был инициализирован, то никакие действия не выполняются и происходит немедленный возврат из подпрограммы. Все оставшиеся MPI-процедуры могут быть вызваны только после вызова *MPI_Init*.

Возвращает: в случае успешного выполнения - *MPI_SUCCESS*, иначе - код ошибки. (То же самое возвращают и все остальные функции, рассматриваемые в данном руководстве.)

```
int MPI_Finalize (void)
```

MPI_Finalize - завершение параллельной части приложения. Все последующие обращения к любым MPI-процедурам, в том числе к *MPI_Init*, запрещены. К моменту вызова *MPI_Finalize* некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Сложный тип аргументов *MPI_Init* предусмотрен для того, чтобы передавать всем процессам аргументы *main*:

```
int main(int argc, char** argv)
{
      MPI_Init(&argc, &argv);
          ...
      MPI_Finalize();
}
```

```
int MPI_Comm_size (MPI_Comm comm,
                   int* size)
```

Определение общего числа параллельных процессов в группе *comm*.

- *comm* - идентификатор группы;
- OUT *size* - размер группы.

```
int MPI_Comm_rank (MPI_Comm comm,
                   int* rank)
```

Определение номера процесса в группе *comm*. Значение, возвращаемое по адресу *&rank*, лежит в диапазоне от 0 до *size_of_group-1*.

- *comm* - идентификатор группы;
- OUT *rank* - номер вызывающего процесса в группе *comm*.

```
double MPI_Wtime (void)
```

Функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Гарантируется, что этот момент не будет изменен за время существования процесса.

Прием/передача сообщений между отдельными процессами.

Прием/передача сообщений с блокировкой.

```
int MPI_Send (void* buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int msgtag,
              MPI_Comm comm)
```

- *buf* - адрес начала буфера посылки сообщения
- *count* - число передаваемых элементов в сообщении
- *datatype* - тип передаваемых элементов
- *dest* - номер процесса-получателя
- *msgtag* - идентификатор сообщения
- *comm* - идентификатор группы

Блокирующая посылка сообщения с идентификатором *msgtag*, состоящего из *count* элементов типа *datatype*, процессу с номером *dest*. Все элементы сообщения расположены подряд в буфере *buf*. Значение *count* может быть нулем. Тип передаваемых элементов *datatype* должен указываться с помощью предопределенных констант типа. Разрешается передавать сообщение самому себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу *dest*, остается за MPI. Следует специально отметить, что возврат из подпрограммы *MPI_Send* не означает ни того, что сообщение уже передано процессу *dest*, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший *MPI_Send*.

```
int MPI_Recv (void* buf,
              int count,
              MPI_Datatype datatype,
              int source,
              int msgtag,
              MPI_Comm comm,
              MPI_Status *status)
```

- OUT *buf* - адрес начала буфера приема сообщения
- *count* - максимальное число элементов в принимаемом сообщении
- *datatype* - тип элементов принимаемого сообщения
- *source* - номер процесса-отправителя
- *msgtag* - идентификатор принимаемого сообщения
- *comm* - идентификатор группы
- OUT *status* - параметры принятого сообщения

Прием сообщения с идентификатором *msgtag* от процесса *source* с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения *count*. Если число принятых элементов меньше значения *count*, то гарантируется, что в буфере *buf* изменятся только

элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в сообщении, то можно воспользоваться подпрограммой *MPI_Probe*.

Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере *buf*.

В качестве номера процесса-отправителя можно указать предопределенную константу *MPI_ANY_SOURCE* - признак того, что подходит сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать константу *MPI_ANY_TAG* - признак того, что подходит сообщение с любым идентификатором.

Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову *MPI_Recv*, то первым будет принято то сообщение, которое было отправлено раньше.

```
int MPI_Get_count (MPI_Status *status,
                   MPI_Datatype datatype,
                   int *count)
```

- *status* - параметры принятого сообщения
- *datatype* - тип элементов принятого сообщения
- OUT *count* - число элементов сообщения

По значению параметра *status* данная подпрограмма определяет число уже принятых (после обращения к *MPI_Recv*) или принимаемых (после обращения к *MPI_Probe* или *MPI_Iprobe*) элементов сообщения типа *datatype*.

```
int MPI_Probe (int source,
               int msgtag,
               MPI_Comm comm,
               MPI_Status *status)
```

- *source* - номер процесса-отправителя или *MPI_ANY_SOURCE*
- *msgtag* - идентификатор ожидаемого сообщения или *MPI_ANY_TAG*
- *comm* - идентификатор группы
- OUT *status* - параметры обнаруженного сообщения

Получение информации о структуре ожидаемого сообщения с блокировкой. Возврата из подпрограммы не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Атрибуты доступного сообщения можно определить обычным образом с помощью параметра *status*. Следует обратить внимание, что подпрограмма определяет только факт прихода сообщения, но реально его не принимает.

Прием/передача сообщений без блокировки.

```
int MPI_Isend (void *buf,
               int count,
               MPI_Datatype datatype,
               int dest,
               int msgtag,
               MPI_Comm comm,
               MPI_Request *request)
```

- *buf* - адрес начала буфера посылки сообщения
- *count* - число передаваемых элементов в сообщении
- *datatype* - тип передаваемых элементов
- *dest* - номер процесса-получателя
- *msgtag* - идентификатор сообщения
- *comm* - идентификатор группы
- OUT *request* - идентификатор асинхронной передачи

Передача сообщения, аналогичная *MPI_Send*, однако возврат из подпрограммы происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере *buf*. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации о завершении данной посылки. Окончание процесса передачи (т.е. того момента, когда можно переиспользовать буфер *buf* без опасения испортить передаваемое сообщение) можно определить с помощью параметра *request* и процедур *MPI_Wait* и *MPI_Test*.

Сообщение, отправленное любой из процедур *MPI_Send* и *MPI_Isend*, может быть принято любой из процедур *MPI_Recv* и *MPI_Irecv*.

```
int MPI_Irecv (void *buf,
               int count,
               MPI_Datatype datatype,
               int source,
               int msgtag,
               MPI_Comm comm,
               MPI_Request *request)
```

- OUT *buf* - адрес начала буфера приема сообщения
- *count* - максимальное число элементов в принимаемом сообщении
- *datatype* - тип элементов принимаемого сообщения
- *source* - номер процесса-отправителя
- *msgtag* - идентификатор принимаемого сообщения
- *comm* - идентификатор группы
- OUT *request* - идентификатор асинхронного приема сообщения

Прием сообщения, аналогичный *MPI_Recv*, однако возврат из подпрограммы происходит сразу после инициализации процесса приема без ожидания получения сообщения в буфере *buf*. Окончание процесса приема можно определить с помощью параметра *request* и процедур *MPI_Wait* и *MPI_Test*.

```
int MPI_Wait (MPI_Request *request,
              MPI_Status *status)
```

- *request* - идентификатор асинхронного приема или передачи
- OUT *status* - параметры сообщения

Ожидание завершения асинхронных процедур *MPI_Isend* или *MPI_Irecv*, ассоциированных с идентификатором *request*. В случае приема, атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра *status*.

```
int MPI_Waitall (int count,
                 MPI_Request *requests,
                 MPI_Status *statuses)
```

- *count* - число идентификаторов
- *requests* - массив идентификаторов асинхронного приема или передачи
- OUT *statuses* - параметры сообщений

Выполнение процесса блокируется до тех пор, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива *statuses* будет установлено в соответствующее значение.

```
int MPI_Waitany (int count,
                 MPI_Request *requests,
                 int *index,
                 MPI_Status *status)
```

- *count* - число идентификаторов
- *requests* - массив идентификаторов асинхронного приема или передачи
- OUT *index* - номер завершенной операции обмена
- OUT *status* - параметры сообщений

Выполнение процесса блокируется до тех пор, пока какая-либо операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если несколько операций могут быть завершены, то случайным образом выбирается одна из них. Параметр *index* содержит номер элемента в массиве *requests*, содержащего идентификатор завершенной операции.

```
int MPI_Waitsome (int incount,
                  MPI_Request *requests,
                  int *outcount,
                  int *indexes,
                  MPI_Status *statuses)
```

- *incount* - число идентификаторов
- *requests* - массив идентификаторов асинхронного приема или передачи
- OUT *outcount* - число идентификаторов завершившихся операций обмена
- OUT *indexes* - массив номеров завершившихся операции обмена
- OUT *statuses* - параметры завершившихся сообщений

Выполнение процесса блокируется до тех пор, пока по крайней мере одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. Параметр *outcount* содержит число завершенных операций, а первые *outcount* элементов массива *indexes* содержат номера элементов массива *requests* с их идентификаторами. Первые *outcount* элементов массива *statuses* содержат параметры завершенных операций.

```
int MPI_Test (MPI_Request *request,
              int *flag,
              MPI_Status *status)
```

- *request* - идентификатор асинхронного приема или передачи
- OUT *flag* - признак завершенности операции обмена
- OUT *status* - параметры сообщения

Проверка завершенности асинхронных процедур *MPI_Isend* или *MPI_Irecv*, ассоциированных с идентификатором *request*. В параметре *flag* возвращает значение 1, если соответствующая операция завершена, и значение 0 в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра *status*.

```
int MPI_Testall (int count,
                 MPI_Request *requests,
                 int *flag,
                 MPI_Status *statuses)
```

- *count* - число идентификаторов
- *requests* - массив идентификаторов асинхронного приема или передачи
- OUT *flag* - признак завершенности операций обмена
- OUT *statuses* - параметры сообщений

В параметре *flag* возвращает значение 1, если все операции, ассоциированные с указанными идентификаторами, завершены (с указанием параметров сообщений в массиве *statuses*). В противном случае возвращается 0, а элементы массива *statuses* неопределены.

```
int MPI_Testany (int count,
                 MPI_Request *requests,
                 int *index,
```

```
                int *flag,
                MPI_Status *status)
```

- *count* - число идентификаторов
- *requests* - массив идентификаторов асинхронного приема или передачи
- OUT *index* - номер завершенной операции обмена
- OUT *flag* - признак завершенности операции обмена
- OUT *status* - параметры сообщения

Если к моменту вызова подпрограммы хотя бы одна из операций обмена завершилась, то в параметре *flag* возвращается значение 1, *index* содержит номер соответствующего элемента в массиве *requests*, а *status* - параметры сообщения.

```
int MPI_Testsome (int incount,
                  MPI_Request *requests,
                  int *outcount,
                  int *indexes,
                  MPI_Status *statuses)
```

- *incount* - число идентификаторов
- *requests* - массив идентификаторов асинхронного приема или передачи
- OUT *outcount* - число идентификаторов завершившихся операций обмена
- OUT *indexes* - массив номеров завершившихся операции обмена
- OUT *statuses* - параметры завершившихся операций

Данная подпрограмма работает так же, как и *MPI_Waitsome*, за исключением того, что возврат происходит немедленно. Если ни одна из указанных операций не завершилась, то значение *outcount* будет равно нулю.

```
int MPI_Iprobe (int source,
                int msgtag,
                MPI_Comm comm,
                int *flag,
                MPI_Status *status)
```

- *source* - номер процесса-отправителя или *MPI_ANY_SOURCE*
- *msgtag* - идентификатор ожидаемого сообщения или *MPI_ANY_TAG*
- *comm* - идентификатор группы
- OUT *flag* - признак завершенности операции обмена
- OUT *status* - параметры обнаруженного сообщения

Получение информации о поступлении и структуре ожидаемого сообщения без блокировки. В параметре *flag* возвращает значение 1, если сообщение с подходящими атрибутами уже может быть принято (в этом случае ее действие полностью аналогично *MPI_Probe*), и значение 0, если сообщения с указанными атрибутами еще нет.

Объединение запросов на взаимодействие.

Процедуры данной группы позволяют снизить накладные расходы, возникающие в рамках одного процессора при обработке приема/передачи и перемещении необходимой информации между процессом и сетевым контроллером. Несколько запросов на прием и/или передачу могут объединяться вместе для того, чтобы далее их можно было бы запустить одной командой. Способ приема сообщения никак не зависит от способа его посылки: сообщение, отправленное с помощью объединения запросов либо обычным способом, может быть принято как обычным способом, так и с помощью объединения запросов.

```
int MPI_Send_init (void *buf,
                   int count,
                   MPI_Datatype datatype,
```

```
                    int dest,
                    int msgtag,
                    MPI_Comm comm,
                    MPI_Request *request)
```

- *buf* - адрес начала буфера посылки сообщения
- *count* - число передаваемых элементов в сообщении
- *datatype* - тип передаваемых элементов
- *dest* - номер процесса-получателя
- *msgtag* - идентификатор сообщения
- *comm* - идентификатор группы
- OUT *request* - идентификатор асинхронной передачи

Формирование запроса на выполнение пересылки данных. Все параметры точно такие же, как и у подпрограммы *MPI_Isend*, однако в отличие от нее пересылка не начинается до вызова подпрограммы *MPI_Startall*.

```
int MPI_Recv_init (void *buf,
                   int count,
                   MPI_Datatype datatype,
                   int source,
                   int msgtag,
                   MPI_Comm comm,
                   MPI_Request *request)
```

- OUT *buf* - адрес начала буфера приема сообщения
- *count* - число принимаемых элементов в сообщении
- *datatype* - тип принимаемых элементов
- *source* - номер процесса-отправителя
- *msgtag* - идентификатор сообщения
- *comm* - идентификатор группы
- OUT *request* - идентификатор асинхронного приема

Формирование запроса на выполнение приема данных. Все параметры точно такие же, как и у подпрограммы *MPI_Irecv*, однако в отличие от нее реальный прием не начинается до вызова подпрограммы *MPI_Startall*.

```
MPI_Startall (int count,
              MPI_Request *requests)
```

- *count* - число запросов на взаимодействие
- OUT *requests* - массив идентификаторов приема/передачи

Запуск всех отложенных взаимодействий, ассоциированных вызовами подпрограмм *MPI_Send_init* и *MPI_Recv_init* с элементами массива запросов *requests*. Все взаимодействия запускаются в режиме без блокировки, а их завершение можно определить обычным образом с помощью процедур *MPI_Wait* и *MPI_Test*.

Совмещенные прием/передача сообщений.

```
int MPI_Sendrecv (void *sbuf,
                  int scount,
                  MPI_Datatype stype,
                  int dest,
                  int stag,
                  void *rbuf,
                  int rcount,
                  MPI_Datatype rtype,
                  int source,
```

```
                    MPI_Datatype rtag,
                    MPI_Comm comm,
                    MPI_Status *status)
```

- *sbuf* - адрес начала буфера посылки сообщения
- *scount* - число передаваемых элементов в сообщении
- *stype* - тип передаваемых элементов
- *dest* - номер процесса-получателя
- *stag* - идентификатор посылаемого сообщения
- OUT *rbuf* - адрес начала буфера приема сообщения
- *rcount* - число принимаемых элементов сообщения
- *rtype* - тип принимаемых элементов
- *source* - номер процесса-отправителя
- *rtag* - идентификатор принимаемого сообщения
- *comm* - идентификатор группы
- OUT *status* - параметры принятого сообщения

Данная операция объединяет в едином запросе посылку и прием сообщений. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Сообщение, отправленное операцией *MPI_Sendrecv*, может быть принято обычным образом, и точно также операция *MPI_Sendrecv* может принять сообщение, отправленное обычной операцией *MPI_Send*. Буфера приема и посылки обязательно должны быть различными.

Коллективные взаимодействия процессов.

В операциях коллективного взаимодействия процессов участвуют все процессы коммуникатора. Соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров. Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено. Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или посылки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата (если это возможно по смыслу операции).

```
int MPI_Bcast (void *buf,
               int count,
               MPI_Datatype datatype,
               int source,
               MPI_Comm comm)
```

- OUT *buf* - адрес начала буфера посылки сообщения
- *count* - число передаваемых элементов в сообщении
- *datatype* - тип передаваемых элементов
- *source* - номер рассылающего процесса
- *comm* - идентификатор группы

Рассылка сообщения от процесса *source* всем процессам, включая рассылающий процесс. При возврате из процедуры содержимое буфера *buf* процесса *source* будет скопировано в локальный буфер процесса. Значения параметров *count*, *datatype* и *source* должны быть одинаковыми у всех процессов.

```
int MPI_Gather (void *sbuf,
                int scount,
                MPI_Datatype stype,
                void *rbuf,
                int rcount,
                MPI_Datatype rtype,
                int dest,
                MPI_Comm comm)
```

- *sbuf* - адрес начала буфера посылки
- *scount* - число элементов в посылаемом сообщении
- *stype* - тип элементов отсылаемого сообщения
- OUT *rbuf* - адрес начала буфера сборки данных
- *rcount* - число элементов в принимаемом сообщении
- *rtype* - тип элементов принимаемого сообщения
- *dest* - номер процесса, на котором происходит сборка данных
- *comm* - идентификатор группы
- OUT *ierror* - код ошибки

Сборка данных со всех процессов в буфере *rbuf* процесса *dest*. Каждый процесс, включая *dest*, посылает содержимое своего буфера *sbuf* процессу *dest*. Собирающий процесс сохраняет данные в буфере *rbuf*, располагая их в порядке возрастания номеров процессов. Параметр *rbuf* имеет значение только на собирающем процессе и на остальных игнорируется, значения параметров *count*, *datatype* и *dest* должны быть одинаковыми у всех процессов.

```
int MPI_Allreduce (void *sbuf,
                   void *rbuf,
                   int count,
                   MPI_Datatype datatype,
                   MPI_Op op,
                   MPI_Comm comm)
```

- *sbuf* - адрес начала буфера для аргументов
- OUT *rbuf* - адрес начала буфера для результата
- *count* - число аргументов у каждого процесса
- *datatype* - тип аргументов
- *op* - идентификатор глобальной операции
- *comm* - идентификатор группы

Выполнение *count* глобальных операций *op* с возвратом *count* результатов во всех процессах в буфере *rbuf*. Операция выполняется независимо над соответствующими аргументами всех процессов. Значения параметров *count* и *datatype* у всех процессов должны быть одинаковыми. Из соображений эффективности реализации предполагается, что операция *op* обладает свойствами ассоциативности и коммутативности.

```
int MPI_Reduce (void *sbuf,
                void *rbuf,
                int count,
                MPI_Datatype datatype,
                MPI_Op op,
                int root,
                MPI_Comm comm)
```

- *sbuf* - адрес начала буфера для аргументов
- OUT *rbuf* - адрес начала буфера для результата
- *count* - число аргументов у каждого процесса
- *datatype* - тип аргументов
- *op* - идентификатор глобальной операции
- *root* - процесс-получатель результата
- *comm* - идентификатор группы

Функция аналогична предыдущей, но результат будет записан в буфер *rbuf* только у процесса *root*.

Синхронизация процессов.

```
int MPI_Barrier( MPI_Comm comm)
```

- *comm* - идентификатор группы

Блокирует работу процессов, вызвавших данную процедуру, до тех пор, пока все оставшиеся процессы группы *comm* также не выполнят эту процедуру.

Работа с группами процессов.

```
int MPI_Comm_split (MPI_Comm comm,
                    int color,
                    int key,
                    MPI_Comm *newcomm)
```

- *comm* - идентификатор группы
- *color* - признак разделения на группы
- *key* - параметр, определяющий нумерацию в новых группах
- OUT *newcomm* - идентификатор новой группы

Данная процедура разбивает все множество процессов, входящих в группу *comm*, на непересекающиеся подгруппы - одну подгруппу на каждое значение параметра *color* (неотрицательное число). Каждая новая подгруппа содержит все процессы одного цвета. Если в качестве *color* указано значение *MPI_UNDEFINED*, то в *newcomm* будет возвращено значение *MPI_COMM_NULL*.

```
int MPI_Comm_free (MPI_Comm comm)
```

- OUT *comm* - идентификатор группы

Уничтожает группу, ассоциированную с идентификатором *comm*, который после возвращения устанавливается в *MPI_COMM_NULL*.

Предопределенные константы.

Предопределенные константы типа элементов сообщений.

константы MPI	тип в C
MPI_CHAR	signed char
MPI_SHORT	signed int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Другие предопределенные типы

- *MPI_Status* - структура; атрибуты сообщений; содержит три обязательных поля:
- *MPI_Source* (номер процесса отправителя)
- *MPI_Tag* (идентификатор сообщения)
- *MPI_Error* (код ошибки)
- *MPI_Request* - системный тип; идентификатор операции посылки-приема сообщения
- *MPI_Comm* - системный тип; идентификатор группы (коммуникатора)

- *MPI_COMM_WORLD* - зарезервированный идентификатор группы, состоящей их всех процессов приложения

Константы-пустышки

- *MPI_COMM_NULL*
- *MPI_DATATYPE_NULL*
- *MPI_REQUEST_NULL*

Константа неопределенного значения

- *MPI_UNDEFINED*

Глобальные операции

- *MPI_MAX*
- *MPI_MIN*
- *MPI_SUM*
- *MPI_PROD*

Любой процесс/идентификатор

- *MPI_ANY_SOURCE*
- *MPI_ANY_TAG*

Код успешного завершения процедуры

- *MPI_SUCCESS*

Лекция 6. Технология программирования OpenMP.

Одним из наиболее популярных средств программирования компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология ***OpenMP***. За основу берется последовательная программа, а для создания ее параллельной версии ***пользователю предоставляется набор директив, процедур и переменных окружения.*** Стандарт OpenMP разработан для языков Фортран, C и C++. Поскольку все основные конструкции для этих языков похожи, то рассказ о данной технологии мы будем вести на примере только одного из них, а именно на примере языка Фортран.

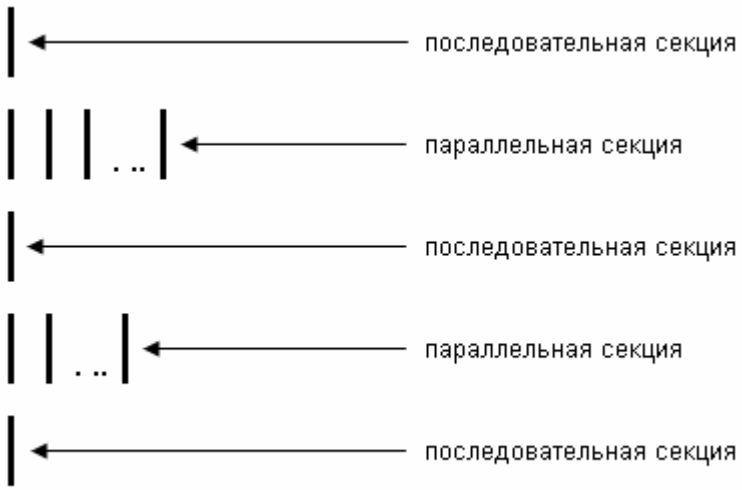


Рис. 2. Процесс исполнения OpenMp-программы.

Как, с точки зрения OpenMP, пользователь должен представлять свою параллельную программу? Весь текст программы разбит на последовательные и параллельные области (см. рис.2). В начальный момент времени порождается нить-мастер или "основная" нить, которая начинает выполнение программы со стартовой точки. Здесь следует сразу сказать, почему вместо традиционных для параллельного программирования процессов появился новый термин - нити (threads, легковесные процессы). Технология OpenMP опирается на понятие общей памяти, поэтому она, в значительной степени, ориентирована на SMP-компьютеры. На подобных архитектурах возможна эффективная поддержка нитей, исполняющихся на различных процессорах, что

позволяет избежать значительных накладных расходов на поддержку классических UNIX-процессов.

Основная нить и только она исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные нити. После порождения каждая нить получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все нити исполняют один и тот же код, соответствующий параллельной области. При выходе из параллельной области основная нить дожидается завершения остальных нитей, и дальнейшее выполнение программы продолжает только она.

В параллельной области все переменные программы разделяются на два класса: общие (*SHARED*) и локальные (*PRIVATE*). Общая переменная всегда существует лишь в одном экземпляре для всей программы и доступна всем нитям под одним и тем же именем. Объявление же локальной переменной вызывает порождение своего экземпляра данной переменной для каждой нити. Изменение нитью значения своей локальной переменной, естественно, никак не влияет на изменение значения этой же локальной переменной в других нитях.

По сути, только что рассмотренные два понятия: области и классы переменных, и определяют идею написания параллельной программы в рамках OpenMP: некоторые фрагменты текста программы объявляется параллельными областями; именно эти области и только они исполняются набором нитей, которые могут работать как с общими, так и с локальными переменными. Все остальное - это конкретизация деталей и описание особенностей реализации данной идеи на практике.

Рассмотрим базовые положения и основные конструкции OpenMP. ***Все директивы OpenMP располагаются в комментариях*** и начинаются с одной из следующих комбинаций: *!$OMP*, *C$OMP* или **$OMP* (напомним, что строка, начинающаяся с одного из символов '!', 'C' или '*' по правилам языка Фортран считается комментарием). В дальнейшем изложении при описании конкретных директив для сокращения записи мы иногда будем опускать эти префиксы, хотя в реальных программах они, конечно же, всегда должны присутствовать. Все переменные окружения и функции, относящиеся к OpenMP, начинаются с префикса *OMP_* .

Описание параллельных областей. Для определения параллельных областей программы используется пара директив

```
!$OMP PARALLEL
   < параллельный код программы >
!$OMP END PARALLEL
```

Для выполнения кода, расположенного между данными директивами, дополнительно порождается *OMP_NUM_THREADS*-1 нитей, где *OMP_NUM_THREADS* - это переменная окружения, значение которой пользователь, вообще говоря, может изменять. Процесс, выполнивший данную директиву (нить-мастер), всегда получает номер 0. Все нити исполняют код, заключенный между данными директивами. После *END PARALLEL* автоматически происходит неявная синхронизация всех нитей, и как только все нити доходят до этой точки, нить-мастер продолжает выполнение последующей части программы, а остальные нити уничтожаются.

Параллельные секции могут быть вложенными одна в другую. По умолчанию вложенная параллельная секция исполняется одной нитью. Необходимую стратегию обработки вложенных секций определяет переменная *OMP_NESTED*, значение которой можно изменить с помощью функции *OMP_SET_NESTED*.

Если значение переменной *OMP_DYNAMIC* установлено в 1, то с помощью функции *OMP_SET_NUM_THREADS* пользователь может изменить значение переменной *OMP_NUM_THREADS*, а значит и число порождаемых при входе в параллельную секцию нитей. Значение переменной *OMP_DYNAMIC* контролируется функцией *OMP_SET_DYNAMIC*.

Необходимость порождения нитей и параллельного исполнения кода параллельной секции пользователь может определять динамически с помощью дополнительной опции *IF* в директиве:

```
!$OMP PARALLEL IF( <условие> )
```

Если <*условие*> не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме.

Мы уже говорили о том, что все порожденные нити исполняют один и тот же код. Теперь нужно обсудить вопрос, как разумным образом распределить между ними работу. OpenMP предлагает несколько вариантов. Можно программировать на самом низком уровне, распределяя работу с помощью функций *OMP_GET_THREAD_NUM* и *OMP_GET_NUM_THREADS*, возвращающих номер нити и общее количество порожденных нитей соответственно. Например, если написать фрагмент вида:

```
IF( OMP_GET_THREAD_NUM() .EQ. 3 ) THEN
      < код для нити с номером 3 >
ELSE
      < код для всех остальных нитей >
ENDIF ,
```

то часть программы между директивами *IF:ELSE* будет выполнена только нитью с номером 3, а часть между *ELSE:ENDIF* - всеми остальными. Как и прежде, этот код будет выполнен всеми нитями, однако функция *OMP_GET_THREAD_NUM()* возвратит значение 3 только для нити с номером 3, поэтому и выполнение данного участка кода для третьей нити и всех остальных будет идти по-разному.

Если в параллельной секции встретился оператор цикла, то, согласно общему правилу, он будет выполнен всеми нитями, т.е. каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву

```
!$OMP DO [опция [[,] опция]:]
   ...
!$OMP END DO ,
```

которая относится к идущему следом за данной директивой оператору *DO*.

Опция *SCHEDULE* определяет конкретный способ распределения итераций данного цикла по нитям:

- *STATIC [,m]* - блочно-циклическое распределение итераций: первый блок из m итераций выполняет первая нить, второй блок - вторая и т.д. до последней нити, затем распределение снова начинается с первой нити; по умолчанию значение m равно 1;
- *DYNAMIC [,m]* - динамическое распределение итераций с фиксированным размером блока: сначала все нити получают порции из m итераций, а затем каждая нить, заканчивающая свою работу, получает следующую порцию опять-таки из m итераций;
- *GUIDED [,m]* - динамическое распределение итераций блоками уменьшающегося размера; аналогично распределению *DYNAMIC*, но размер выделяемых блоков все время уменьшается, что в ряде случаев позволяет аккуратнее сбалансировать загрузку нитей;
- *RUNTIME* - способ распределения итераций цикла выбирается во время работы программы в зависимости от значения переменной *OMP_SCHEDULE*.

Выбранный способ распределения итераций указывается в скобках после опции *SCHEDULE*, например:

```
!$OMP DO SCHEDULE (DYNAMIC, 10)
```

В данном примере будет использоваться динамическое распределение итераций блоками по 10 итераций.

В конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки. Если в подобной задержке нет необходимости, то директива *END DO NOWAIT* позволяет нитям уже дошедшим до конца цикла продолжить выполнение без синхронизации с остальными. Если директива *END DO* в явном виде и не указана, то в конце парал-

лельного цикла синхронизация все равно будет выполнена. Рассмотрим следующий пример, расположенный в параллельной секции программы:

```
!$OMP DO SCHEDULE (STATIC, 2)
      DO i = 1, n
            DO j = 1, m
                  A( i, j) = ( B( i, j-1) + B( i-1, j) ) / 2.0
            END DO
      END DO
!$OMP END DO
```

В данном примере внешний цикл объявлен параллельным, причем будет использовано блочно-циклическое распределение итераций по две итерации в блоке. Относительно внутреннего цикла никаких указаний нет, поэтому он будет выполняться последовательно каждой нитью.

Параллелизм на уровне независимых фрагментов оформляется в OpenMP с помощью директивы *SECTIONS : END SECTIONS*:

```
!$OMP SECTIONS
      < фрагмент 1>
!$OMP SECTIONS
      < фрагмент 2>
!$OMP SECTIONS
      < фрагмент 3>
!$OMP END SECTIONS
```

В данном примере программист описал, что все три фрагмента информационно независимы, и их можно исполнять в любом порядке, в частности, параллельно друг другу. Каждый из таких фрагментов будет выполнен какой-либо одной нитью.

Если в параллельной секции какой-то участок кода должен быть выполнен лишь один раз (такая ситуация иногда возникает, например, при работе с общими переменными), то его нужно поставить между директивами *SINGLE : END SINGLE*. Такой участок кода будет выполнен нитью, первой дошедшей до данной точки программы.

Одно из базовых понятий OpenMP - ***классы переменных***. Все переменные, используемые в параллельной секции, могут быть либо общими, либо локальными. Общие переменные описываются директивой *SHARED*, а локальные директивой *PRIVATE*. Каждая общая переменная существует лишь в одном экземпляре и доступна для каждой нити под одним и тем же именем. Для каждой локальной переменной в каждой нити существует отдельный экземпляр данной переменной, доступный только этой нити. Предположим, что следующий фрагмент расположен в параллельной секции:

```
I = OMP_GET_THREAD_NUM()
PRINT *, I
```

Если переменная I в данной параллельной секции была описана как локальная, то на выходе будет получен весь набор чисел от 0 до *OMP_NUM_THREADS*-1, идущих, вообще говоря, в произвольном порядке, но каждое число встретиться только один раз. Если же переменная I была объявлена общей, то единственное, что можно сказать с уверенностью - мы получим последовательность из *OMP_NUM_THREADS* чисел, лежащих в диапазоне от 0 до *OMP_NUM_THREADS*-1 каждое. Сколько и каких именно чисел будет в последовательности заранее сказать нельзя. В предельном случае, это может быть даже набор из *OMP_NUM_THREADS* одинаковых чисел I_0. Предположим, что все процессы, кроме процесса I_0, выполнили первый оператор, но затем их выполнение по какой-то причине было прервано. В это время процесс с номером I_0 присвоил это значение переменной I, а поскольку данная переменная является общей, то одно и тоже значение затем и будет выведено каждой нитью.

Целый набор директив в OpenMP предназначен для синхронизации работы нитей. Самый распространенный способ синхронизации - барьер. Он оформляется с помощью директивы

```
!$OMP BARRIER
```

Все нити, дойдя до этой директивы, останавливаются и ждут пока все нити не дойдут до этой точки программы, после чего все нити продолжают работать дальше.

Пара директив *MASTER : END MASTER* выделяет участок кода, который будет выполнен только нитью-мастером. Остальные нити пропускают данный участок и продолжают работу с выполнения оператора, расположенного следом за директивой *END MASTER*.

С помощью директив

```
!$OMP CRITICAL [ (<имя_критической_секции>) ]
   ...
!$OMP END CRITICAL [ (< имя_ критической_секции >) ],
```

оформляется критическая секция программы. В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью P_0, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока нить P_0 не закончит выполнение данной критической секции. Как только P_0 выполнит директиву *END CRITICAL*, одна из заблокированных на входе нитей войдет в секцию. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание. Все неименованные критические секции условно ассоциируются с одним и тем же именем.

Частым случаем использования критических секций на практике является обновление общих переменных. Например, если переменная *SUM* является общей и оператор вида *SUM=SUM+Expr* находится в параллельной секции программы, то при одновременном выполнении данного оператора несколькими нитями можно получить некорректный результат. Чтобы избежать такой ситуации можно воспользоваться механизмом критических секций или специально предусмотренной для таких случаев директивой *ATOMIC*:

```
!$OMP ATOMIC
SUM = SUM + Expr .
```

Данная директива относится к идущему непосредственно за ней оператору, гарантируя корректную работу с общей переменной, стоящей в левой части оператора присваивания.

Поскольку в современных параллельных вычислительных системах может использоваться сложная структура и иерархия памяти, пользователь должен иметь гарантии того, что в необходимые ему моменты времени каждая нить будет видеть единый согласованный образ памяти. Именно для этих целей и предназначена директива

```
!$OMP  FLUSH [ список_переменных ].
```

Выполнение данной директивы предполагает, что значения всех переменных, временно хранящиеся в регистрах, будут занесены в основную память, все изменения переменных, сделанные нитями во время их работы, станут видимы остальным нитям, если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п. Поскольку выполнение данной директивы в полном объеме может повлечь значительных накладных расходов, а в данный момент нужна гарантия согласованного представления не всех, а лишь отдельных переменных, то эти переменные можно явно перечислить в директиве списком.

Мы не будем далее разбирать конструкции данной технологии, желающие найти полные тексты спецификаций OpenMP для языков Фортран, С и С++ могут обратиться к сайту **http://www.openmp.org**.

Чем привлекательна технология OpenMP? Можно отметить несколько моментов, среди которых стоит особо подчеркнуть два. Во-первых, технология изначально спроектирована таким образом, чтобы пользователь мог работать с единым текстом для параллельной и последовательной программ. В самом деле, обычный компилятор на последовательной машине директивы OpenMP просто "не замечает", поскольку они расположены в комментариях. Единственным источником проблем могут стать переменные окружения и специальные функции, однако для них в спецификациях стандарта предусмотрены специальные "заглушки", гарантирующие корректную работу OpenMP-программы в последовательном случае - нужно только перекомпилировать программу и подключить другую библиотеку. Другим достоинством OpenMP является возможность постепенного, "инкрементного" распараллеливания программы. Взяв за основу последовательный код, пользователь шаг за шагом добавляет новые директивы, описывающие новые параллельные секции. Нет необходимости сразу писать параллельную программу, ее создание ведется последовательно, что упрощает и процесс программирования, и отладку.

Лекция 7. Система параллельного программирования Linda.

Идея построения системы ***Linda*** исключительно проста, а потому красива и очень привлекательна. Параллельная программа есть множество параллельных процессов, и каждый процесс работает согласно обычной последовательной программе. Все процессы имеют доступ к общей памяти, единицей хранения в которой является кортеж. Отсюда происходит и специальное название для общей памяти - пространство кортежей. Каждый кортеж это упорядоченная последовательность значений. Например,

```
( "Hello", 42, 3.14 ),   ( "P", 5, FALSE, 97, 1024, 2),   ( "worker", 5 )
```

Первый элемент кортежа всегда является символьной строкой и выступает в роли имени кортежа. Так первый кортеж предыдущего примера состоит из имени ("*Hello*"), элемента целого типа (42) и вещественного числа (3.14). Во втором кортеже кроме имени "P" есть элемент целого типа (5), элемент логического типа (*FALSE*) и три целых числа. Последний кортеж состоит из двух элементов: имени ("*worker*") и целого числа (5). Количество элементов в кортеже может быть любым.

Все процессы работают с пространством кортежей по принципу: поместить кортеж, забрать, скопировать. В отличие от традиционной памяти, процесс может забрать кортеж из пространства кортежей, после чего данный кортеж станет недоступным остальным процессам. В отличие от традиционной памяти, если в пространство кортежей положить два кортежа с одним и тем же именем, то не произойдет привычного для нас "обновления" значения переменной - в пространстве кортежей окажется два кортежа с одним и тем же именем. В отличие от традиционной памяти, изменить кортеж непосредственно в пространстве нельзя. Для изменения значений элементов кортежа, его нужно сначала оттуда изъять, затем процесс, изъявший кортеж, может изменить значения его элементов и вновь добавить измененный кортеж в память. В отличие от других систем программирования, процессы в системе Linda никогда не взаимодействуют друг с другом явно, и все общение всегда идет через пространство кортежей.

Интересно, что с точки зрения системы Linda в любой последовательный язык достаточно добавить лишь четыре новые функции, как он становится средством параллельного программирования! Эти функции и составляют систему Linda: три для операций над кортежами и пространством кортежей и одна функция для порождения параллельных процессов. Для определенности, дальнейшее обсуждение системы и ее функций будем вести с использованием языка C.

Функция OUT помещает кортеж в пространство кортежей. Например,

```
out ( "GoProcess", 5);
```

помещает в пространство кортежей кортеж *("GoProcess", 5)*. Если такой кортеж уже есть в пространстве кортежей, то появится второй, что, в принципе, позволяет иметь сколь угодно много экземпляров одинаковых кортежей. По этой же причине с помощью функции *out* нельзя изме-

нить кортеж, уже находящийся в пространстве. Для этого кортеж должен быть сначала оттуда изъят, затем изменен и после этого помещен назад. Функция *out* никогда не блокирует выполнивший ее процесс.

Функция IN ищет подходящий кортеж в пространстве кортежей, присваивает значения его элементов элементам своего параметра-кортежа и удаляет найденный кортеж из пространства кортежей. Например,

```
in( "P", int i, FALSE );
```

Этой функции соответствует любой кортеж, который состоит из трех элементов: значением первого элемента является "*P*", второй элемент может быть любым целым числом, а третий должен иметь значение FALSE. Подходящими кортежами могут быть *("P", 5, FALSE)* или *("P", 135, FALSE)* и т.п., но не *("P", 7.2, FALSE)* или *("Proc", 5, FALSE)*. Если параметру функции in соответствуют несколько кортежей, то ***случайным образом*** выбирается один из них. После нахождения кортеж удаляется из пространства кортежей, а неопределенным формальным элементам параметра-кортежа, содержащимся в вызове данной функции, присваиваются соответствующие значения. В предыдущем примере переменной *i* присвоится 5 или 135. Если в пространстве кортежей ни один кортеж не соответствуют функции, то вызвавший ее процесс ***блокируется*** до тех пор, пока соответствующий кортеж в пространстве не появится.

Элемент кортежа в функции *in* считается формальным, если перед ним стоит определитель типа. Если используется переменная без определителя типа, то берется ее значение и элемент рассматривается как фактический параметр. Например, во фрагменте программы

```
int i = 5;
in( "P", i, FALSE );
```

функции *in*, в отличие от предыдущего примера, соответствует только кортеж *("P", 5, FALSE)*.

Если переменная описана до вызова функции и ее надо использовать как формальный элемент кортежа, можно использовать ключевое слово *formal* или знак '?'. Например, во фрагменте программы

```
j = 15;
in( "P", formal i, j );
```

последнюю строку можно заменить и на оператор *in("P", ?i, j)*. В этом примере функции *in* будет соответствовать, например, кортеж *("P", 6, 15)*, но не *("P", 6, 12)*. Конечно же, формальными могут быть и несколько элементов кортежа одновременно:

```
in ( "Add_If", int i, bool b);
```

Если после такого вызова функции в пространстве кортежей будет найден кортеж ("*Add_If", 100, TRUE)*, то переменной *i* присвоится значение 100, а переменной *b* - значение *TRUE*.

Функция READ отличается от функции *in* лишь тем, что выбранный кортеж не удаляется из пространства кортежей. Все остальное точно так же, как и у функции *in*. Этой функцией удобно пользоваться в том случае, когда значения переменных менять не нужно, но к ним необходим параллельный доступ из нескольких процессов.

Функция EVAL похожа на функцию *out*. Разница заключается лишь в том, что дополнительным элементом кортежа у *eval* является функция пользователя. Для вычисления значения этой функции система Linda ***порождает параллельный процесс***, на основе работы которого она формирует кортеж и помещает его в пространство кортежей. Например,

```
eval ("hello", funct( z ), TRUE, 3.1415);
```

При обработке данного вызова система создаст новый процесс для вычисления функции *funct(z)*. Когда процесс закончится и будет получено значение *w = funct(z)*, в пространство

кортежей будет добавлен кортеж *("hello", w, TRUE, 3.1415)*. Функция, вызвавшая *eval*, не ожидает завершения порожденного параллельного процесса и продолжает свою работу дальше. Следует отметить и то, что пользователь не может явно управлять размещением порожденных параллельных процессов на доступных ему процессорных устройствах - это Linda делает самостоятельно.

Параллельная программа в системе Linda считается завершенной, если все порожденные процессы завершились или все они заблокированы функциями *in* и *read*.

По сути дела, описание системы закончено, и теперь можно привести несколько небольших примеров. Мы уже говорили о том, что параллельные процессы в системе Linda напрямую друг с другом не общаются, своего уникального номера-идентификатора не имеют и общего числа параллельно работающих процессов-соседей, вообще говоря, не знают. Однако если у пользователя есть в этом необходимость, то такую ситуацию очень просто смоделировать. Программа в самом начале вызывает функцию *out*:

```
out( "Next", 1);
```

Этот кортеж будет играть роль "эстафетной палочки", передаваемой от процесса процессу: каждый порождаемый параллельный процесс первым делом выполнит следующую последовательность:

```
in( "Next", formal My_Id);
out( "Next", My_Id+1);
```

Первый оператор изымает данный кортеж из пространства, на его основе процесс получает свой номер *My_Id*, и затем кортеж с номером для следующего процесса помещается в пространство. Заметим, что использование функции *in* в данном случае позволяет гарантировать монопольную работу с данным кортежем только одного процесса в каждый момент времени. После такой процедуры каждый процесс получит свой уникальный номер, а число уже порожденных процессов всегда можно определить, например, с помощью такого оператора:

```
read( "Next", formal Num_Processes);
```

Теперь рассмотрим возможную схему организации программы для перемножения *C=A*B* двух квадратных матриц размера *N*N*. Инициализирующий процесс использует функцию *out* и помещает в пространство кортежей исходные строки матрицы *A* и столбцы матрицы *B*:

```
out( "A",1, <1-я строка A>);
out( "A",2, <2-я строка A>);
   ...
out( "B",1, <1-й столбец B>);
out( "B",2, <2-й столбец B>);
   ...
```

Для порождения *Nproc* идентичных параллельных процессов можно воспользоваться следующим фрагментом:

```
for( i = 0; i < Nproc; ++i )
   eval( "ParProc", get_elem_result() );
```

Входные данные готовы, и нахождение всех N^2 элементов C_{ij} результирующей матрицы можно выполнять в любом порядке. Главное - это распределить работу между процессами, для чего процесс, инициирующий вычисления, в пространство помещает следующий кортеж:

```
out( "NextElementCij", 1);
```

Второй элемент данного кортежа всегда будет показывать, какой из N^2 элементов C_{ij} предстоит вычислить следующим. Базовый вычислительный блок функции *get_elem_result()* будет содержать следующий фрагмент:

```
in( "NextElementCij", formal NextElement);
if( NextElement < N*N )
   out("NextElementCij ", NextElement + 1);
Nrow = (NextElement - 1) / N + 1;
Ncol = (NextElement - 1) % N + 1;
```

В результате выполнения данного фрагмента для элемента с номером *NextElement* процесс определит его местоположение в результирующей матрице: номер строки *Nrow* и столбца *Ncol*. Заметим, что если вычисляется последний элемент, то кортеж с именем "*NextElementCij*" в пространство не возвращается. Когда в конце работы программы процессы обратятся к этому кортежу, они будут заблокированы, что не помешает нормальному завершению программы. И, наконец, для вычисления элемента C_{ij} каждый процесс *get_elem_result* выполнит следующий фрагмент:

```
read( "A", Nrow, formal row);
read( "B", Ncol, formal col);
out( "result", Nrow, Ncol, DotProduct(row,col) );
```

где *DotProduct* это функция, реализующая скалярное произведение. Таким образом, каждый элемент произведения окажется в отдельном кортеже в пространстве кортежей. Завершающий процесс соберет результат, поместив их в соответствующие элементы матрицы C:

```
for ( irow = 0; irow < N; irow++)
   for ( icol = 0; icol < N; icol++)
         in( "result", irow + 1, icol + 1, formal C[irow][icol]);
```

Не имея в системе Linda никаких явных средств для синхронизации процессов, совсем не сложно их смоделировать самому. Предположим, что в некоторой точке нужно выполнить барьерную синхронизацию N процессов. Какой-то один процесс, например, стартовый, заранее помещает в пространство кортеж *("ForBarrier", N)*. Подходя к точке синхронизации, каждый процесс выполняет следующий фрагмент, который и будет выполнять функции барьера:

```
in( "ForBarrier", formal Bar);
Bar = Bar - 1;
if( Bar != 0 ) {
   out( "ForBarrier", Bar);
   read( "Barrier" );
} else
   out( "Barrier" );
```

Если кортеж с именем "*ForBarrier*" есть в пространстве, то процесс его изымает, в противном случае блокируется до его появления. Анализируя второй элемент данного кортежа, процесс выполняет одно из двух действий. Если есть процессы, которые еще не дошли до данной точки, то он возвращает кортеж в пространство с уменьшенным на единицу вторым элементом и встает на ожидание кортежа "*Barrier*". В противном случае он сам помещает кортеж "*Barrier*" в пространство, который для всех является сигналом к продолжению работы.

В целом, сильные и слабые стороны системы Linda понятны. Простота и стройность концепции системы является основным козырем, однако эти же факторы оборачивается большой проблемой на практике. Как эффективно поддерживать пространство кортежей на практике? Если вычислительная система обладает распределенной памятью, то общение процессов через пространство кортежей заведомо будет сопровождаться большими накладными расходами. Если процесс выполняет функции *read* или *in* - как среди потенциально огромного числа кортежей в

пространстве быстро найти подходящий? Подобные проблемы пытаются решить разными способами, например, введением нескольких именованных пространств, но все предлагаемые решения либо усложняют систему, либо являются эффективными только для узкого класса программ.

Содержание