ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ

СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

ВОЕВОДИН ВЛАДИМИР ВАЛЕНТИНОВИЧ И ДР.

ВМК МГУ

БЛАГОДАРИМ ЗА ПОДГОТОВКУ КОНСПЕКТА
СТУДЕНТА ФИЗИЧЕСКОГО ФАКУЛЬТЕТА МГУ
БЕЛОВА МИХАИЛА АНАТОЛЬЕВИЧА

Оглавление

Лекция 1. Современные суперкомпьютеры

Введение. Понятие суперкомпьютера

Данный курс разделен на две части. Первая относится к понятию **суперкомпьютера**. Интуитивно ясно, что это определение не связано с обычными устройствами типа смартфонов, ноутбуков и мощных серверов. А что же это такое будет описано далее. И вторая часть нашего курса является не менее важной – **параллельная обработка данных, параллелизм**, то есть те идеи, которые активно привносятся в последние годы в компьютерный мир. Самый большой в мире параллельный компьютер – **интернет.** И если научиться объединять распределенные ресурсы вместе для решения одной заранее заданной задачи, то можно получить тот же параллельный компьютер. Люди применяют данные технологии, например, для взлома шифров, создания лекарств и поиска внеземных цивилизаций.

В курсе также будут представлены три основные области:

1. Устройство современных параллельных компьютеров.
2. Архитектура параллельных вычислительных систем.
3. Технологии параллельного программирования. Структуры программ и алгоритмов.

Что такое суперкомпьютер? Несмотря на то, что сам по себе этот объект существует достаточно давно, четкого определения ему нет. Это связано с тем, что по мере развития технологий суперкомпьютер 30-ти летней давности ни в коей мере сейчас суперкомпьютером не является. Поэтому основное определение следующее. **Суперкомпьютеры** – все те машины, которые в данный момент времени работают быстрее всех.

Вместе с этим существует огромное количество альтернативных определений. Одно из них звучит так: **суперкомпьютеры** – машины, которые всегда занимают большой зал. Что называется большим залом? Для расшифровки этого понятия приведем пример суперкомпьютера «Ломоносов». Он занимает площадь примерно в $1000\ м^2$.

То есть любой современный суперкомпьютер является огромным информационным заводом, у которого есть инженерная инфраструктура, специализированные источники питания, система охлаждения и т.д. Более того, **каждый суперкомпьютер весит больше одной тонны и стоит больше одного миллиона долларов** (вес СК «Ломоносов» 160 тонн).

Более предметное определение следующее: **суперкомпьютеры** – машины, которые сводят проблему вычислений к проблеме ввода-вывода. То есть суперкомпьютер справляется с вычислениями достаточно быстро, но проблема ввода-вывода становится узким местом. И не всегда получается ее распараллелить, так как операции часто по смыслу являются последовательными.

И, наконец, заключительное определение на сегодня. **Суперкомпьютеры** – машины, мощности которых чуть-чуть не хватает для решения сегодняшних задач.

Вычислительно сложные задачи существуют всегда. Поэтому с одной стороны, это достаточно серьезная проблема, а с другой стороны, это тот самый двигатель, который постоянно существует в вечном цикле. То есть появляются новые машины с большими возможностями, что позволяет ставить задачи в новой постановке, с новой размерностью и точностью.

Что касается сферы применения суперкомпьютеров, то можно смело сказать, что суперкомпьютеры используются везде. Наука, медицина, военная сфера, кинематограф – это лишь малая часть тех областей, где сегодня происходят вычисления с помощью суперкомпьютеров (Рис.1.1).

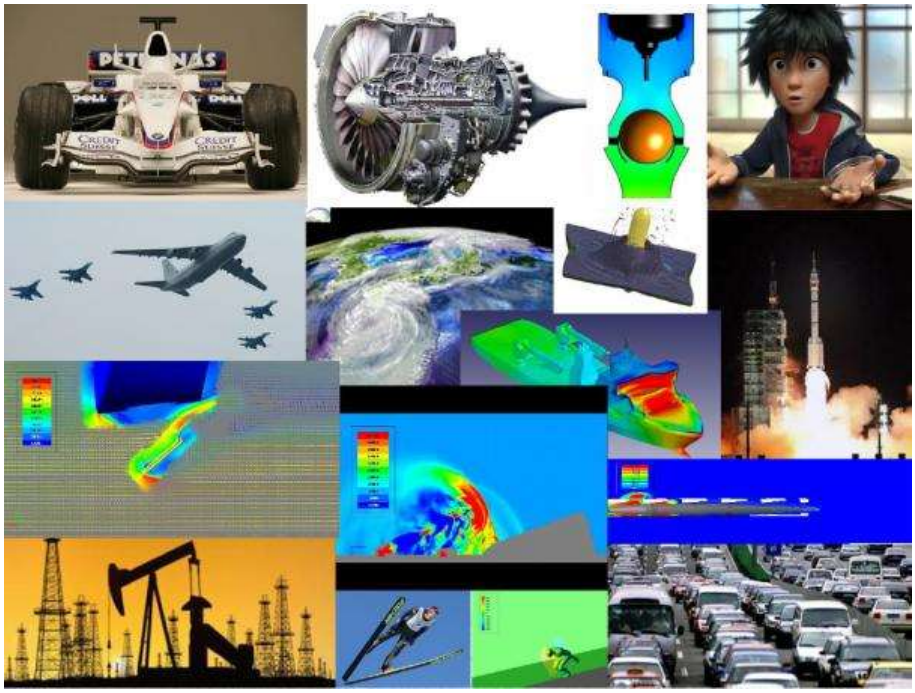


Рис.1.1 Области применения суперкомпьютеров.

Теперь если мы посмотрим на компьютерный мир в целом и представим его в виде пирамиды, то поймем, что в основании лежат мобильные устройства типа планшетов и смартфонов, далее появляются ноутбуки, персональные компьютеры, серверы и на вершине пирамиды находятся суперкомпьютеры.

Очень многие технологии зарождаются именно в суперкомпьютерной отрасли.

Производительность суперкомпьютеров

Далее сравним современный суперкомпьютер с привычным персональным компьютером 2018 года, у которого средняя производительность около 100 млрд. оп/с, оперативной памяти от $8 \div 64$ Гбайта и объем дисков порядка 500 Гбайт.

И для понимания часто встречаемых сокращений приведем их расшифровку:

- Мега (Mega) – 10^6 (миллион)
- Гига (Giga) – 10^9 (биллион / миллиард)
- Тера (Tera) – 10^{12} (триллион)

- Пета (Peta) – 10^{15} (квадриллион)
- Экза (Exa) – 10^{18} (квинтиллион)
- Флоп/с, Flop/s – ***Floating point operations per second***

$15\,Tflop/s = 15 * 10^{12}$ арифметических операций в секунду над вещественными числами, представленными в форме с плавающей точкой.

Самыми затратными операциями являются арифметические. Также выполняются операции над вещественными числами, которые представляются в форме с плавающей точкой.

Все эти значения операций в секунду являются пиковыми и теоретически достижимыми, то есть на практике можно встретиться, например, с проблемами компилятора и получить уже реальную производительность в разы меньше пиковых величин.

В данный момент современная техника работает на уровне «Пета» величин, но ожидается в районе 2020-2022 годов появления первых «Экза» флопных машин.

Первые отечественные суперкомпьютеры

Первая большая машина, которая появилась в Московском университете в 1956 году – **ЭВМ «Стрела»** (Рис.1.2).

Рис.1.2. ЭВМ «Стрела». Установлена в Московском государственном университете в 1956 году.

Машина стояла в вычислительном центре МГУ и на ней считались первый полет человека в космос и первый запуск спутника в том числе.

Вторая машина – **ЭВМ «Сетунь»** (Рис.1.3). Исключительное свойство данной машины в том, что она работала на троичной логике: $-1, 0$ и 1.

Рис.1.3. ЭВМ «Сетунь». 1959 год.

Из-за троичного представления программа становилась более компактной и самое главное исчезали проблемы, связанные с ошибками округления. Было выпущено порядка 100 машин этой серии.

Уникальной российской системой является **ЭВМ «БЭСМ-6»** (Рис.1.4), которая была спроектирована гениальным конструктором **Сергеем Алексеевичем Лебедевым**.

Рис.1.4. Слева портрет С.А.Лебедева. В центре ЭВМ «БЭСМ-6». 1968 год.

Эта машина была широко распространена по всей России и после С.А.Лебедева осталась целая школа и его ученики дальше проектировали машины нового поколения.

Рейтинги современных суперкомпьютеров

Теперь перейдем к современным суперкомпьютерам. Все дальнейшие примеры машин были взяты из списка *Top500* самых мощных систем мира.

Первая машина, которая перешла через «Пета» флопный рубеж была **IBM «RoadRunner»**, США (Рис.1.5).

Рис.1.5. IMB «RoadRunner», #1 Top500 в 2008-2009 г.

Интересный факт: в этом суперкомпьютере используется процессор *IBM Cell*, который был создан изначально для игровых приставок (PlayStation).

В 2011 году Япония завершила национальный проект по созданию суперкомпьютерной системы **«K Computer»** (Рис.1.6).

Рис.1.6. «K Computer», #1 Top500 в 2011 г.

Целью данного компьютера было достижение производительности в 10 $Pflop/s$. И располагается он в отдельном здании.

Следующий пример суперкомпьютер **«Sequoia», IBM BlueGene/Q** (Рис.1.7), США. Производительность в 16 $Pflop/s$.

Рис.1.7. IBM BlueGene/Q, «Sequoia», #1 Top500 в 2012 г.

Хорошо видно, что современный суперкомпьютер соблюдает очень четкую иерархию параллелизма. И на Рис.1.8 показана иерархичность архитектуры и параллелизма суперкомпьютеров.

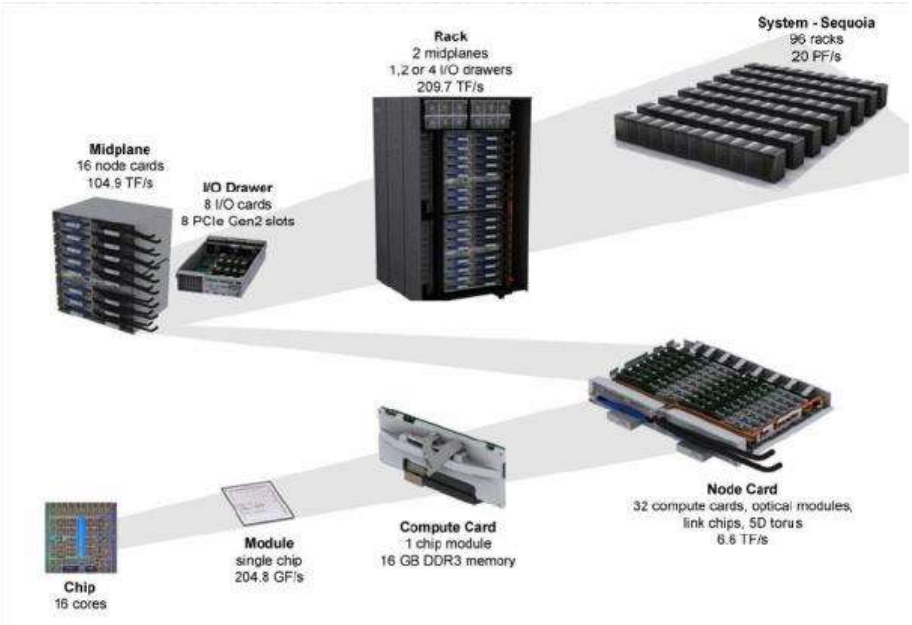


Рис.1.8. Архитектура IBM BlueGene/Q, «Sequoia».

В 2013 году на мировую арену суперкомпьютеров выходит Китай. И их машина **«Tianhe-2»** (Рис.1.9) занимает Топ-1 в рейтинге суперкомпьютеров. Более того, этот суперкомпьютер продержался рекордное количество редакций на первом месте до 2015 года.

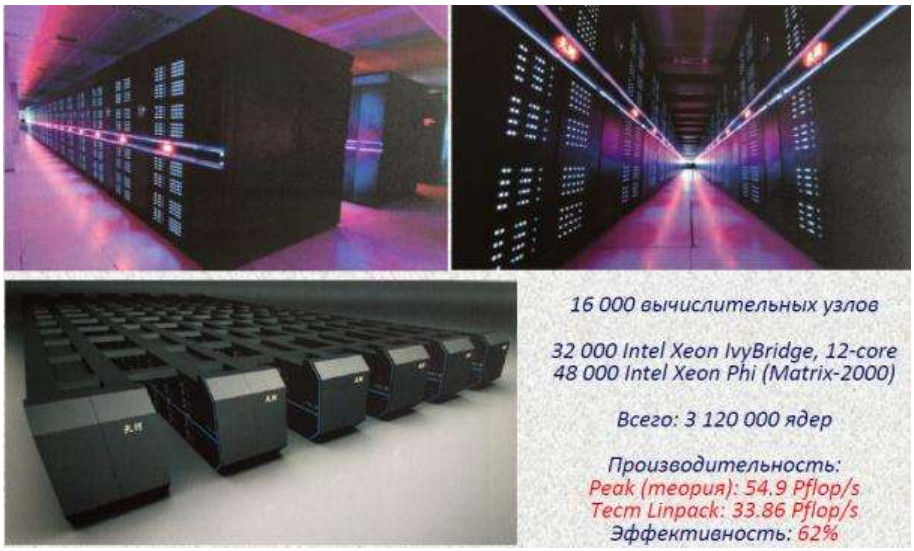


Рис.1.9. «Tianhe-2», #1 Top500 в 2013-2015 г.

Еще один китайский компьютер **«Sunway TaihuLight»** (Рис.1.10), который занял первое место в 2016 году.

Рис.1.10. «Sunway TaihuLight», #1 Top500 в 2016-2017 г.

Он интересен по многим параметрам. Компьютер полностью сделан на китайском процессоре, более того, он является полностью китайским. То есть Китай является самодостаточным в плане производства высокотехнологичных компонентов.

И перейдем к лидеру современных суперкомпьютеров (на 2018-2019 г.) – **IBM Summit**, США (Рис.1.11).

Рис.1.11. IBM Summit, #1 Top500 в 2018-2019 г.

Список *Top500* можно найти по адресу http://top500.org. Он представлен в виде таблицы, которая содержит много полезной и интересной информации: место в рейтинге, место расположения, название суперкомпьютера, количество ядер, пиковая и тестовая производительности и энергопотребление данного компьютера.

Давайте посмотрим на графики производительности суперкомпьютеров с 1993 года (Рис.1.12).

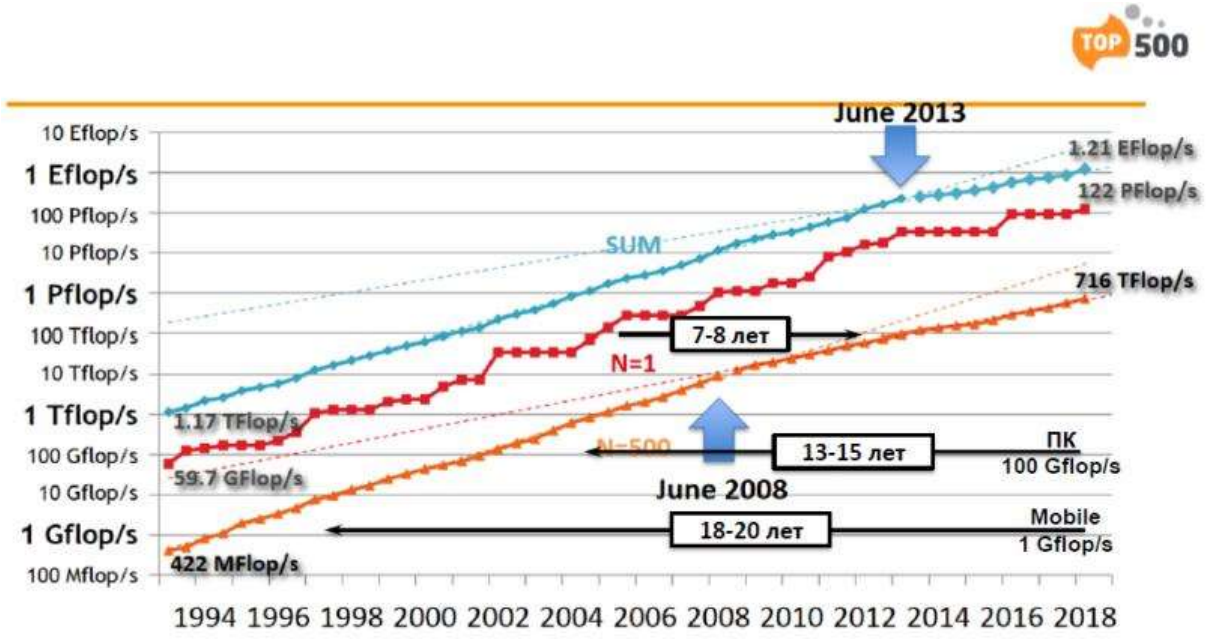


Рис.1.12. Производительность суперкомпьютеров.
N=1 первый в списке Top500, N=500 –последний. SUM – сумма производительности всех 500 машин.

Из Рис.1.12 видно, что наблюдается регулярное изменение производительности последней машины из списка *Top500*. Отчасти это можно объяснить законом Мура, но конкретных предложений не выявлено. Что касается первого места, то здесь стоит отметить большую роль политики и имиджевой составляющей государств и отдельных компаний. Поэтому наблюдаются регулярные всплески производительности.

Вообще закон Мура перестает действовать, так как нормы с каждым годом уменьшаются и проектирование суперкомпьютеров по старым технологиям становится все сложнее и сложнее. И на Рис.1.12 наблюдается отход от прямой с 2008 года для $N = 500$ и точно также в 2013 году общая сумма начинает замедляться.

Еще одним важным свойством развития отрасли является ее динамичность. То есть за 7-8 лет без обновления железа легко можно переместиться с первого на последнее место списка.

Огромные операционные расходы суперкомпьютеров связаны с потреблением энергии. Поэтому люди все чаще взвешивают возможную выгоду и затраты перед строительством сверхтехнологичных систем.

Для сравнения посмотрим на электропотребление вокруг нас (Рис.1.13).

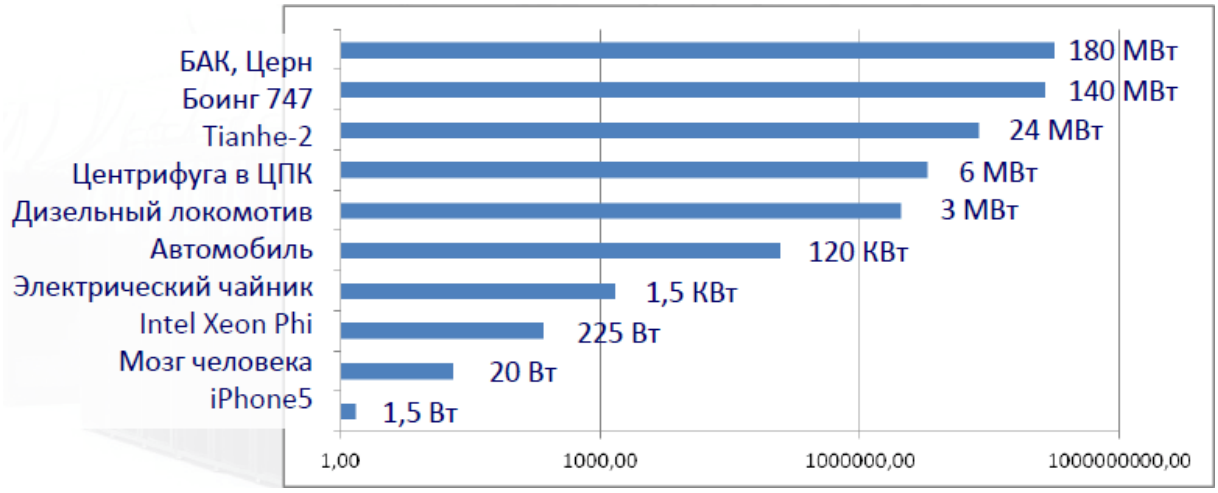


Рис.1.13. Гистограмма электропотребления.

Суперкомпьютеры «Ломоносов» и «Ломоносов-2»

«Ломоносов» имеет пиковую производительность в $1.7\,Pflop/s$ и число процессоров 12346 штук. Суперкомпьютер находится в здании ВМК МГУ.

В данный момент основной машиной является суперкомпьютер **«Ломоносов-2»** (Рис.1.14), который расположен в Ломоносовском корпусе МГУ.

Производительность порядка $5\,Pflop/s$. Компьютер используют тысячи пользователей и сотни организаций. Доступ можно получить через интернет.

Рис.1.14. «Ломоносов-2».

Назначение суперкомпьютеров

Теперь нам нужно ответить на следующие вопросы:

- Неужели есть настолько **сложные задачи**, что для их решения не хватает хорошего сервера?
- Неужели есть настолько **важные задачи**, которые оправдывают крайне высокую стоимость суперкомпьютеров?

На оба вопроса ответы **«Да»**.

В качестве примера приведем задачу о подсчете числа счастливых билетиков (когда сумма первых трех цифр билета равна сумме последних трех). Выясняется, что решение этой задачи путем введения шестерного цикла невозможно, когда в билете будет 22 цифры даже с помощью современных суперкомпьютеров. А обычные компьютеры не справляются уже с 16 цифрами.

Еще одним примером может служить задача о размещении N ферзей на шахматную доску размерами $K * K$.

Конечно, суперкомпьютеры конструируются для более серьезных задач, про которые речь пойдет в следующих лекциях.

Лекция 2. Сферы применения суперкомпьютеров

Суперкомпьютеры в сфере добычи нефти и газа

Любая нефтяная и газовая компании не обходятся без использования суперкомпьютеров. Традиционная задача, которая возникает в этом случае, звучит так: необходимо создать план действий для эффективной добычи полезных ископаемых на большой глубине. Для этого производят моделирование подземного резервуара.

Будем считать, что этот резервуар представляется в виде сеточной области (Рис.2.1), которая имеет 100 точек по каждому измерению.

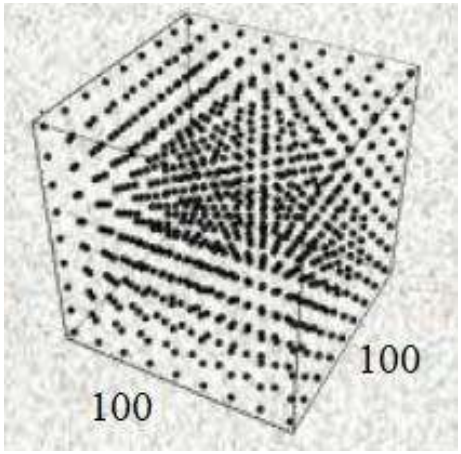


*Рис.2.1. Сеточная область размерами 100*100*100 точек.*

Затем в каждой точке сетки вычисляются от 5 до 20 функций. Это температура, три компоненты скорости, давление, концентрация, нефть, газ и т.д. Нахождение значений функций происходит путем решения нелинейных уравнений. Для этого требуется порядка 500 арифметических операций. Тогда мы получаем хорошую статическую картину. Однако, нам нужно узнать эволюцию системы во времени. Для этого выполняем порядка 1000 шагов по времени.

Тогда **трудоемкость процесса** будет равна:

$$10^6\ (\text{точек сетки}) * 10\ (\text{функций}) * 500\ (\text{операций}) * 500\ (\text{шагов}) = \\ = \mathbf{2500}\ \text{млрд. операций}$$

Очевидно, что обычные компьютеры с такой задачей не справятся, тем более, что мы максимально упростили нашу модельную систему.

Для нефтегазовых компаний эти расчеты позволяют выбирать лучшее место для бурения скважин, что позволяет существенно экономить деньги.

Примером может служить случай в практике компании TOTAL, когда имелись подозрения, что под соленым колпаком могут быть запасы углеводородов (Рис.2.2 верхняя половина). Но после серьезного моделирования стало понятно, что это был артефакт вычислительной природы. И по оценкам самой компании удалось сэкономить порядка 80 $M\$$.

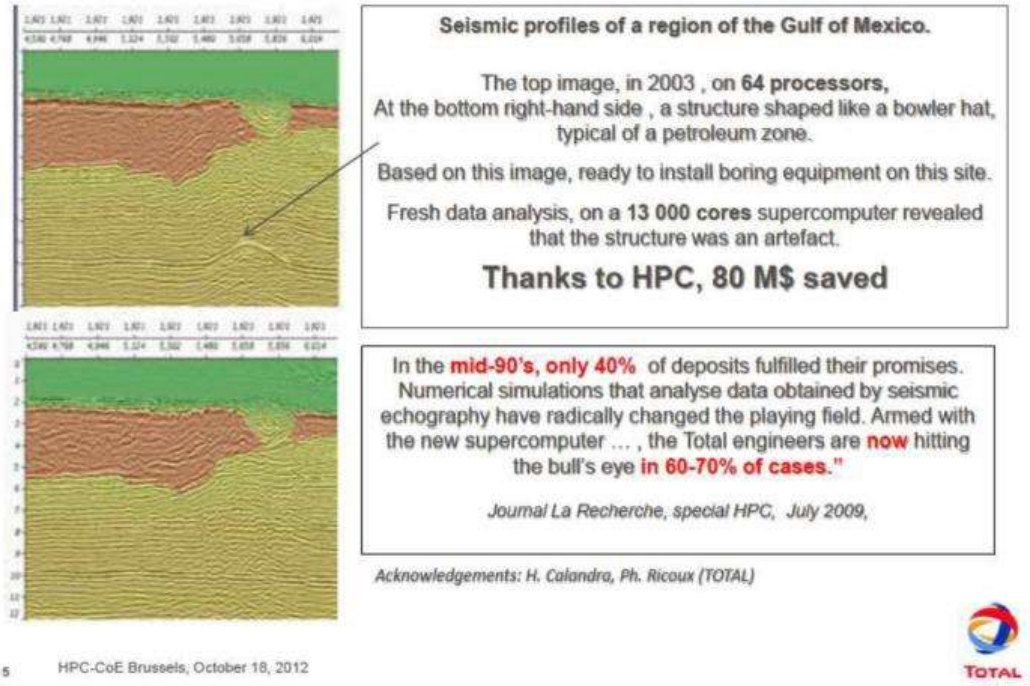


Рис.2.2. Пример использования суперкомпьютеров в деятельности нефтедобывающих компаний.

Суперкомпьютеры в автомобилестроении, авиации и космической отрасли

В области **автомобилестроения** расчеты на суперкомпьютерах производятся при краш-тестах, когда перед проведением экспериментов сначала строится модель и происходит ее расчет. На Рис.2.3 показан пример лобового столкновения автомобилей *Mercedes E-class* и *Smart*.

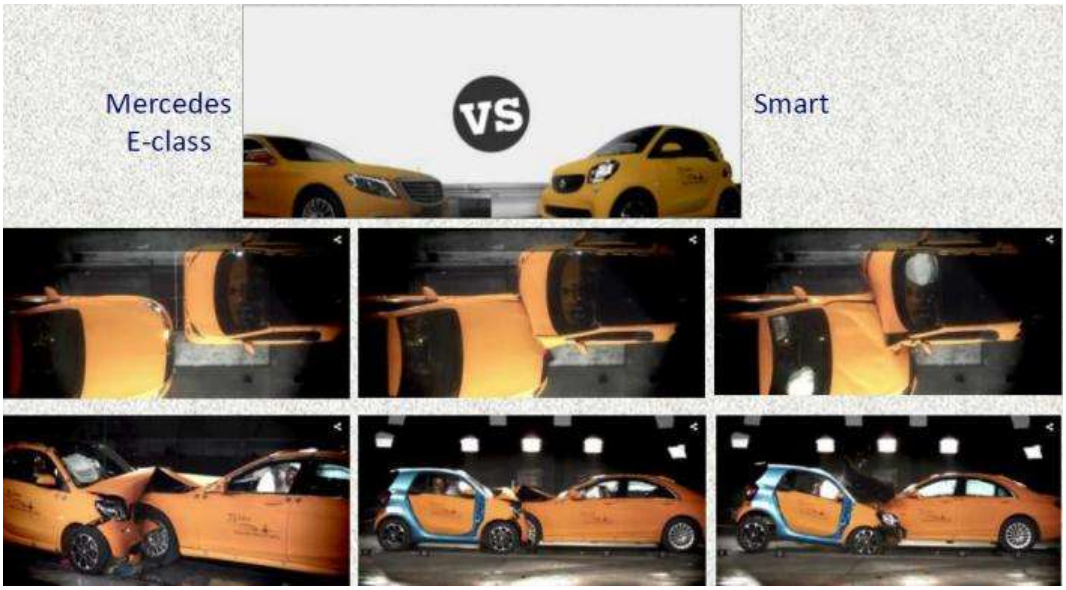


Рис.2.3. Реальный эксперимент краш-теста автомобилей.

Таким образом, компаниям очень удобно использовать моделирование, так как можно легко изменять параметры системы и получать соответствующие результаты. В настоящее время сертификация автомобилей происходит с использованием вычислительных экспериментов.

Еще одной областью использования суперкомпьютеров в автомобилестроении является «Формула-1». Для построения идеального болида необходимы расчеты на суперкомпьютерах. Современные технологии позволяют проектировать автомобили под конкретные трассы и погодные условия. Поэтому для обеспечения равенства между различными командами было введено ограничение на максимальную производительность суперкомпьютеров.

В **авиации** невозможно обойтись без мощностей суперкомпьютеров. В России есть несколько серьезных компаний, связанных с авиатехнологиями. Одна из них НПО «Сатурн», которая занимается двигателестроением. Авиационный двигатель является одним из самых сложных инженерных устройств когда-либо изобретенным человеком. На Рис.2.4 приведены основные детали двигателя и сравнение стоимости одного килограмма высокотехнологичных изделий.

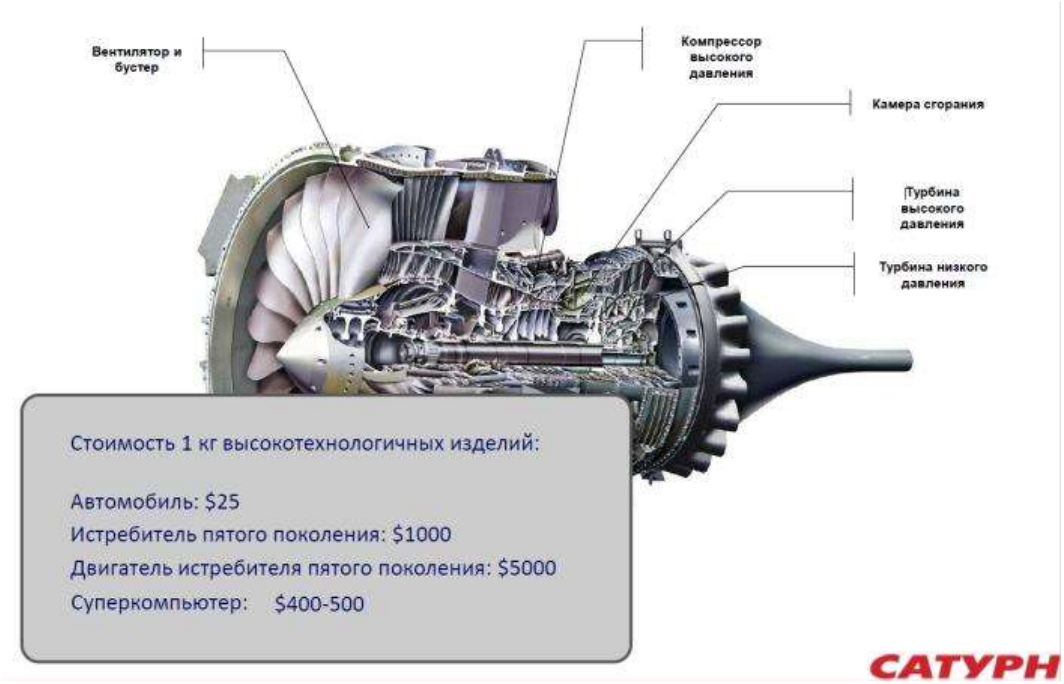


Рис.2.4. Строение авиационного двигателя и таблица стоимости 1 кг высокотехнологичных изделий.

Для того чтобы моделировать отдельные узлы нужно провести некоторое количество операций с плавающей точкой. Для стационарного режима работы двигателя данные приведены на Рис.2.5.

Экспериментов с двигателями много. Проверяют работу двигателя в различных погодных условиях. И опять получается, что моделирование позволяет сэкономить

большие деньги и при этом достоверность расчетов сохраняется на высоком, допустимом уровне.

Узел	Размер задачи (млн. ячеек)	Кол-во операций с плавающей точкой для расчета узла
Вентилятор	10	$2,5 * 10^{16}$
Компрессор низкого давления	8	$1,5 * 10^{16}$
Компрессор высокого давления	10	$4,5 * 10^{16}$
Камера сгорания	20	$2,4 * 10^{17}$
Турбина высокого давления	8	$1,1 * 10^{17}$
Турбина низкого давления	8	$1,1 * 10^{16}$

Рис.2.5. Объем расчетов стационарного режима двигателя.

В задачах **аэродинамики и аэроакустики** также применяются вычисления на суперкомпьютерах. Проведем расчет трудоемкости для моделирования простых элементов летательного аппарата: крыла без механизации, несущего винта вертолета, отсека вооружения и т.д. Вводится сетка порядка 50 млн. ячеек. Трудоемкость на ячейку составляет порядка 10-30 тыс. операций. Шагов по времени около 250 тыс. Тогда в итоге получаем $2.5 * 10^{16}$ операций.

Если проводить расчет сложных конфигураций: крыло с механизацией, крыло с двигателем, несущий винт с учетом фюзеляжа и т.д., то итоговое число операций возрастает до 10^{18}.

При моделировании турбулентных течений (Рис.2.6) также стоимость расчета будет равна порядка 10^{18} операций.

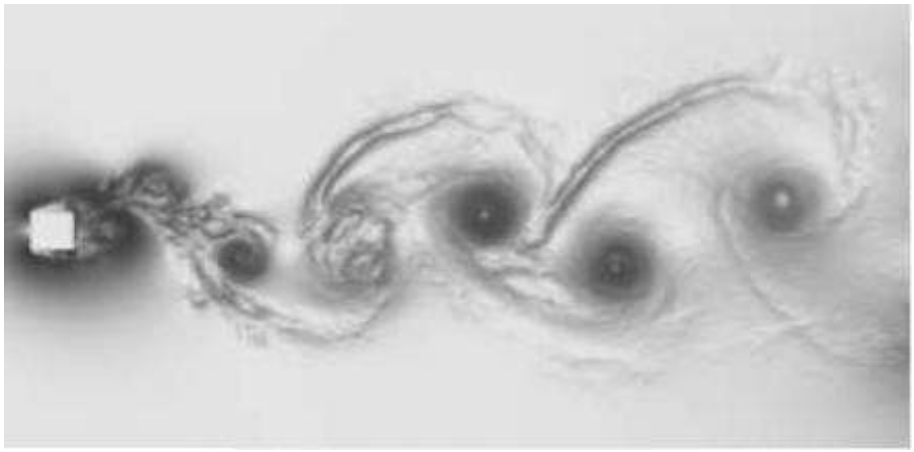


Рис.2.6. Турбулентный след за цилиндром (дорожка Кармана).

В **космической отрасли** суперкомпьютеры используются, например, для расчета минимального импульса отстрела крышки парашютного отсека пиропатронами, при котором крышка не ударит по корпусу корабля при отстреле (совместный проект МГУ и

компании ТЕСИС для РКК «Энергия»). Причем заметим, что эта задача не может быть отработана экспериментально. Здесь пригодно только численное моделирование.

Суперкомпьютеры в медицине

Для построения ультразвуковых томографов (Рис.2.7) необходимы современные суперкомпьютеры. При расчете используется сетка $500 * 500 * 500$ точек, количество источников ультразвука около 100, количество положений приемников около 1000. Решается обратная задача восстановления внутренний структуры $3D$ объекта с помощью итерационного процесса.

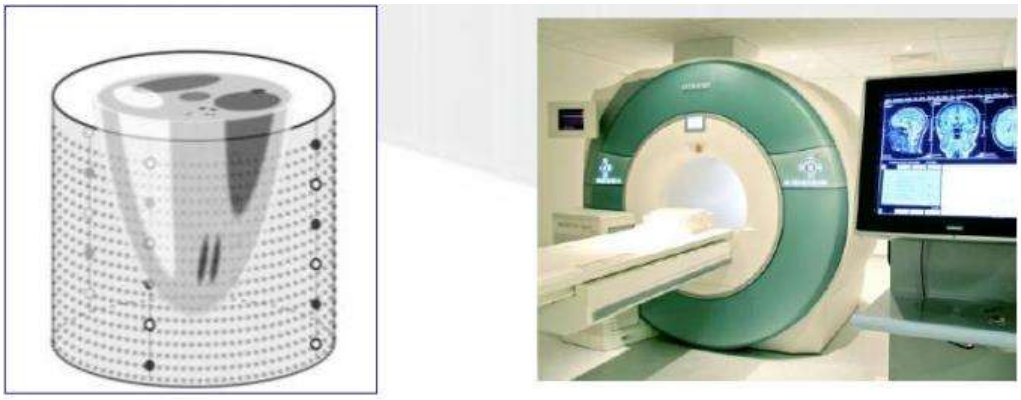


Рис.2.7. Слева – возможный вид сетки, справа – современный томограф.

Общая трудоемкость процесса составляет порядка $2.5 * 10^{17}$ операций.

Моделирование климата

Земля покрывается сеткой с минимальным размером $2000 * 1000 * 50$ точек (Рис.2.8). Далее выбирается несколько десятков прогностических переменных (три компоненты скорости, температура, соленость в океане и т.д.). На одну такую переменную необходимо около 100 операций. Как правило рассматривают достаточно длинные интервалы по времени (20,50 и 100 лет). А для расчета одного года используют 10^5 шагов по времени.

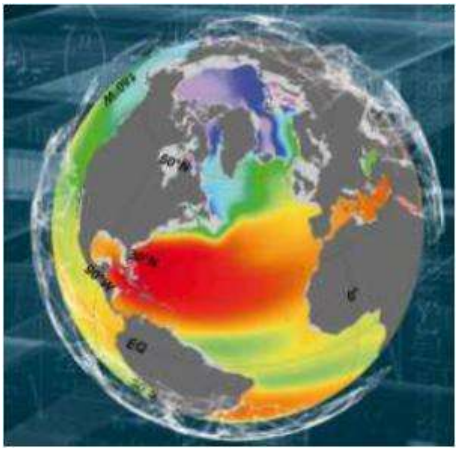


Рис.2.8. Сетка для поверхности Земли.

Тогда общая трудоемкость будет равна порядка 10^{18} операций.

Моделирование параметров течения в ветропарке

Цель данного моделирования заключается в определении физических параметров течения, эффективного расположения ветроэнергетических установок (ВЭУ) и оценке генерируемой мощности ветропарка.

Габариты расчетной области ветропарка: 4 км $* 4$ км $* 1$ км. Общее число ВЭУ составляет от 20 до 50 штук. Сетка размером $572 * 572 * 144$ точек. В зависимости от выбранной модели турбулентности в каждой точке сетки вычисляется от 25 до 40 функций. Порядка 100 операций приходится на каждую функцию и около 56000 шагов по времени для моделирования (Рис.2.9).

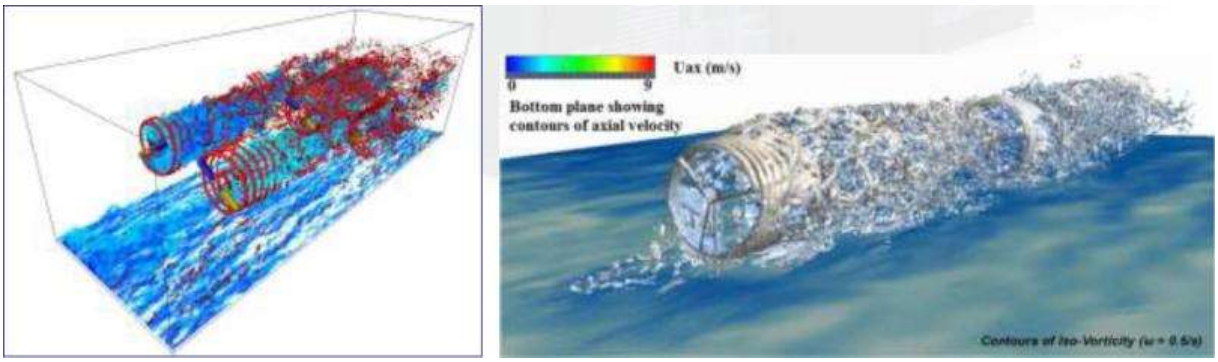


Рис.2.9. Слева – поперечное расположение 2 ВЭУ, справа – продольное.

Суммарная трудоемкость такого моделирования – $6.5 * 10^{15}$ опреаций.

Суперкомпьютеры в спорте

Изначально человека окутывают огромным количеством датчиков. Данные его движений записываются, далее строится каркасная модели в компьютеры и происходит расчет на суперкомпьютерах в специальных пакетах, например, $FlowVision$. То есть можно видеть динамику исследуемого процесса, что позволяет корректировать существующие проблемы.

Данные технологии применяется во многих видах спорта: конькобежный спорт, лыжный спорт, прыжки с трамплина, плавание и т.д.

Кинематограф и суперкомпьютеры

Традиционно при производстве современных фильмов используют для рендеринга мощности суперкомпьютеров. Так, например, фильм 2016 года «Книга Джунглей» является почти полностью цифровым. Всего было использовано 30 млн. процессорочасов.

А на суперкомпьютере «Ломоносов-2» происходил расчет спецэффектов для фильма «Время первых».

Суперкомпьютерные технологии и наука

Для России крайне важной задачей является **моделирование распределения вечной мерзлоты**, так как большое количество городов расположены на ней. Также инфраструктура, нефть, газ, электричество и дороги – всё лежит на вечной мерзлоте.

Очень актуальной и перспективной задачей является **проектирование лекарств**. Суперкомпьютеры позволяют ускорить разработку и удешевить ее в сотни раз. Так, например, разработка нового лекарства требует 10-15 лет и до $500 млн.

Суть процесса следующая: необходимо найти такую молекулу, которая будет ингибировать (то есть взаимодействовать с активным центром мишени без физиологических последствий) нужные белки (Рис.2.10).

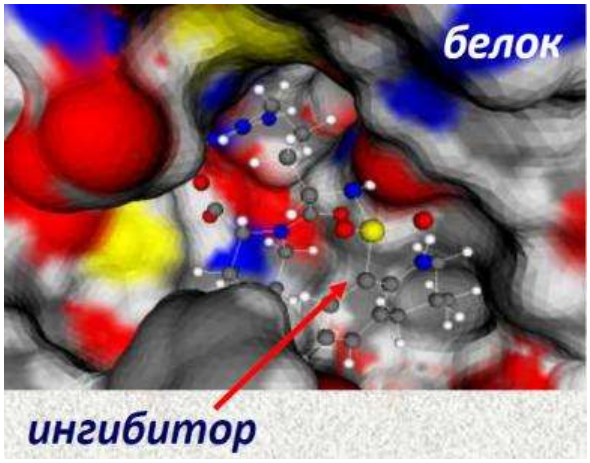


Рис.2.10. Показан активный центр белка, в котором находится молекула ингибитор.

Вычислительные сложности нужны для того, чтобы провести моделирование взаимодействия этого ингибитора с белком в заданных условиях. Кроме того, существует большая база возможных молекул-ингибиторов, с которыми необходимо провести расчеты.

Безопасность также сопряжена с использованием суперкомпьютеров. Классическая задача звучит так: имеется некоторое число (N – ключ), которое является произведением двух простых (p и q). Если мы знаем p и q, то мы также знаем ключ к шифру. Поэтому важно быстро раскладывать заданное число на такие сомножители.

Вся основная идея современных систем шифрования заключается в том, что если взять число большим, то разложить на простые сомножители не может никто и ничто за разумное время.

Посмотрим на общую статистику распределения по областям применения $Top500$ самых мощных суперкомпьютеров мира (Рис.2.11).

В промышленности задействовано более половины всех компьютеров (58.2%). Но замечаем, что рекордные производительности нужны именно в исследовательской области.

Рис.2.11. Распределение суперкомпьютеров по областям применения (июнь 2019 г.).

Параллелизм в архитектуре суперкомпьютеров

На следующем примере рассмотрим за счет чего происходит увеличение производительности компьютеров.

Один из первых компьютеров **EDSAC** 1949 года имел **время такта $2 * 10^{-6}$ с.** Его **производительность** равна **10^2 оп/с.**

Суперкомпьютер **Cray Titan** #1 2012 года. **Время такта $4.5 * 10^{-10}$ с.** А его **производительность $1.7 * 10^{16}$ оп/с.**

Оценим изменения. Тактовая частота увеличилась, то есть компьютер считает быстрее. С 1949 до 2012 года увеличение примерно в 1000 раз, здорово, но мало. А увеличение производительности примерно в 10^{14}. Получается, что прямой вклад микроэлектроники относительно небольшой.

Поэтому можно сделать **два вывода**:

1. Безусловно, без развития элементарной базы не было бы такого прогресса в развитии компьютеров.
2. Но основной вклад в увеличении производительности компьютеров – это развитие архитектуры, и прежде всего, за счет глубокого внедрения идей параллелизма.

Люди идут по пути увеличения степени параллельности из-за того, что законы физики не позволяют наращивать тактовую частоту настолько быстро, насколько это требуется.

На Рис.2.12 показаны изменения параметров микропроцессоров. Здесь видно, что число транзисторов неуклонно растет, тактовая частота не наращивается, но происходит рост числа ядер, которые можно внедрять на один процессор.

То есть в будущем все планшеты, мобильные телефоны, персональные компьютеры и ноутбуки будут суперпараллельными. И будущие разработчики должны иметь знания в области параллельного программирования.

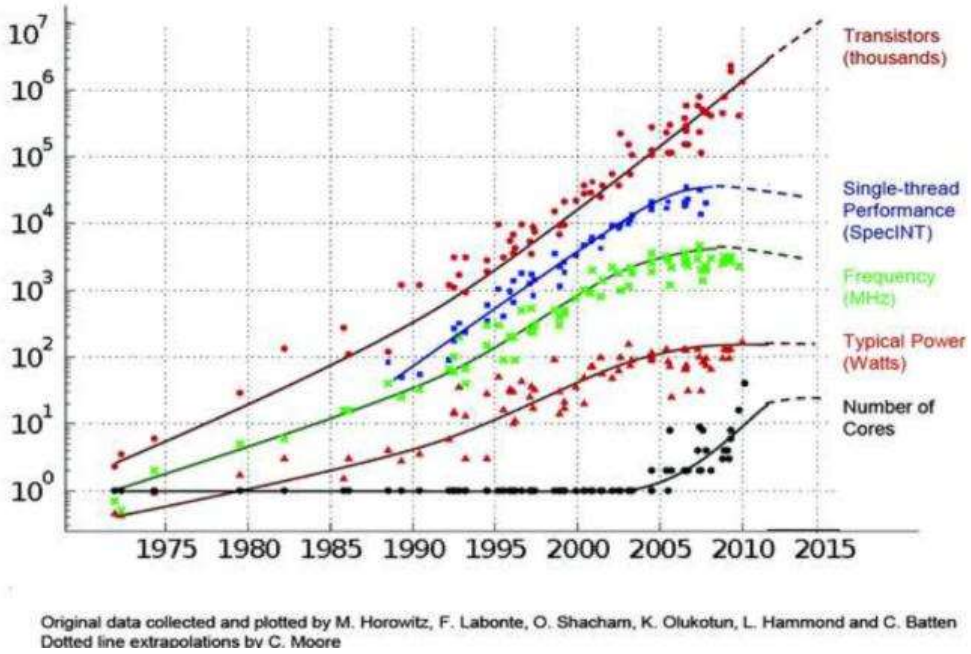


Рис.2.12. Изменение параметров микропроцессоров.

Мировые гиганты типа NVIDIA производят графический процессор **NVIDIA GPU (Tesla V100, Volta)**. Он имеет 5120 ядер, производительность на двойной точности 7.5 $Tflop/s$.

Существует **две** основные **идеи параллельной обработки**:

- Параллелизм
- Конвейерность

В качестве **примера параллелизма** рассмотрим задачу на последовательную обработку данных. Необходимо сложить два вектора, каждый из которых содержит по 100 элементов. Будем использовать одно последовательное устройство, которое выполняет одну операцию за 5 тактов.

На первом такте аргументы поступают на устройство. Далее 5 тактов устройство занято, потом появляется результат и устройство освобождается. Таким образом, за 500 тактов все 100 элементов будут обработаны.

Нужно ускорить решение задачи, то есть добавим еще одно такое же устройство. Тогда задачу выполним за 250 тактов. И т.д.

А в случае **конвейерной обработки данных** имеем следующее. На первом такте на первую ступень загружается первая пара аргументов, после обработки эти аргументы переходят на вторую ступень. И т.д.

Тогда время обработки всего набора из N входных данных:

$$T = L + (N - 1),$$

где L – время заполнения всех ступеней.

Интересная особенность, которая очень часто сопровождает конвейерное устройство – это **режим зацепления**. То есть ситуация, когда выход одного конвейерного устройства соединен со входом другого.

И в случае, когда нам нужно выполнить следующую операцию:

$$A_i = B_i + C_i d, i = 1 \dots N$$

Для обычного режима имеем:

$$T = L_2 + (N-1) + L_1 + (N-1)$$

И для обработки в режиме с зацеплением:

$$T = L_2 + L_1 + (N-1)$$

Наблюдается ускорение в два раза.

Этот прием очень части используется во многих процессорах.

Еще одно важное свойство – **иерархия памяти**. В данной момент она выглядит следующим образом:

- Регистры
- Кэш-память 1-го уровня
- Кэш-память 2-го уровня
- Оперативная память
- Дисковые устройства
- Ленточные устройства
- …

И важными характеристиками программ являются

- Локальность вычислений
- Локальность использования данных

А время чтения для различных уровней иерархии памяти будет определяться понятием **локальности**: насколько близко расположены данные/команды друг относительно друга.

Лекция 3. Характеристики параллельных программ

Производительность компьютера и время решения задачи

Как мы уже поняли, суперкомпьютеры – это колоссальная мощь, но для того, чтобы добиться быстрого решения поставленной задачи необходимо преодолеть проблемы, связанные с эффективной организацией параллельных вычислений.

Рассмотрим следующий пример. Необходимо вскопать огород размерами $10 * 10$ метров. Один человек справится с задачей, например, за 10 часов. Необходимо ускорить процесс. Тогда привлекаем к работе $5, 10, 50$ помощников и ожидаем, что наше итоговое время уменьшится в $5, 10, 50$ раз.

Теперь представим, что нужно выкопать яму размерами $1 * 1 * 1$ метров. Один рабочий справится за 2 часа, но хотелось бы быстрее. Но теперь, если мы позовем 10 человек, то наша производительность не увеличится в 10 раз, так как эту задачу нельзя распараллелить на 10 частей. То есть для получения ускорения теперь нужно использовать совершенно другие методы и подходы (например, аренда экскаватора).

На практике ситуация где-то посередине и ее можно хорошо проиллюстрировать следующим примером. Нам необходимо сделать фундамент. Причем сначала бригадир должен выполнить разметку, а после этого любое количество рабочих может приступать к заливке фундамента.

Тогда в обычном режиме, когда всего 1 бригадир и 1 рабочий, имеем:

$$1 \text{ час разметка, остальное} - 10 \text{ часов} = 11 \text{ часов}$$

При 10 рабочих выполним заливку за

$$1 \text{ час разметка, остальное} - 1 \text{ час} = 2 \text{ часа}$$

А при 100 рабочих время заливки сократится до

$$1 \text{ час разметка, остально} - 6 \text{ минут} = 1 \text{ час } 6 \text{ минут}$$

То есть ясно, что бригадир становится «узким местом», так как его работу распараллелить не удается.

Характеристики параллельных программ. Закон Амдала

Будем считать, что p – число процессоров, на которых хотим исполнять последовательную программу. Считаем, что T_1 – время исполнения программы одним процессором. Тогда T_p – время работы программы на всей системе из p процессоров. Также вводится понятие доли последовательных операций f $(0 \leq f \leq 1)$. Мы хотим оценить насколько быстро будет выполняться программа, если переходим от одного процессора к системе из p процессоров. Нам нужно научиться оценивать ускорение

$$S = \frac{T_1}{T_p}$$

Закон Амдала – ускорение работы программы при переходе с одного процессора на систему из p процессоров оценивается величиной

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

Этот закон интуитивно понятен, то есть это оценка влияния того бригадира (f). Причем на практике $S < p$, а идеальный случай без последовательных элементов $S = p$ – линейное ускорение работы программы.

При большом числе процессоров ускорение примерно равно

$$S \approx \frac{1}{f}$$

То есть мы можем прикинуть что получится в предположении, что у нас очень много процессоров. Если доля последовательных операций в некоторой программе равна 0.1, значит **вне зависимости от числа используемых процессоров** ускорение не превысит 10.

Еще одно следствие звучит так: **Для того чтобы ускорить программу в q раз, необходимо ускорить не менее, чем в q раз не менее, чем $(1 - \frac{1}{q})$-ю часть программы**.

Для ускорения работы программы в 100 раз необходимо ускорить не менее, чем в 100 раз не менее 99% этой программы.

Рост производительности параллельных вычислительных систем

Введем следующие понятия, которые мы будем использовать в курсе.

Пиковая производительность – производительность компьютера, которая оценивается по бумагам, схемам без каких-либо запусков. То есть все процессоры и исполняющие устройства работают с максимальной производительностью. Причем нет проверки на успеваемость памяти «подкидывать» результаты для того, чтобы устройство всегда было занято.

Реальная производительность – отношение числа операций, которые требуется для выполнения программы, к затраченному времени их выполнения.

Далее рассмотрим пример идеальной ситуации, когда мы говорим о производительности вычислительной системы (Рис.3.1).

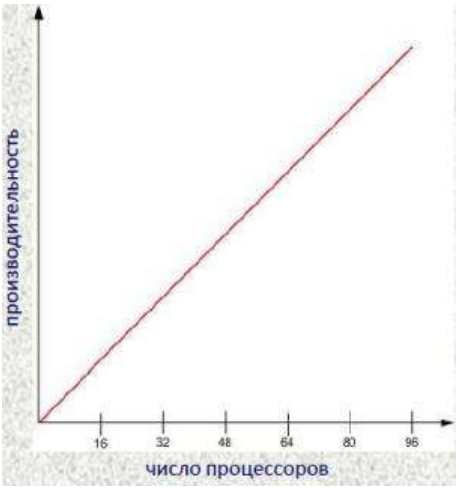


Рис.3.1. Рост производительности (идеальный случай).

Тут важно отметить, что это график роста пиковых производительностей, а реальные производительности ведут себя по-другому (Рис.3.2).

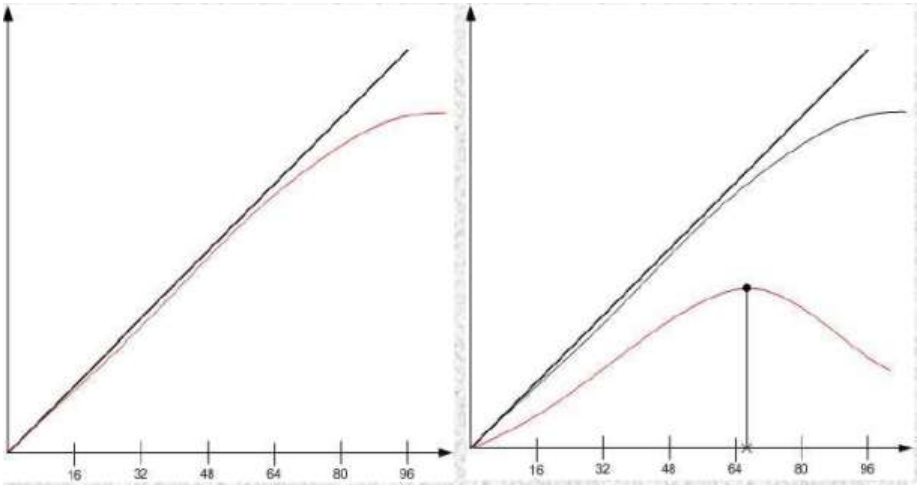


Рис.3.2. Графики зависимости реальной производительности от числа процессоров.

Здесь наблюдаются отхождения от теоретических кривых в виду особенностей программы, алгоритмов и системы.

Однако, еще возможны ситуации, когда реальная производительность превосходит пиковую из-за особенностей современной архитектуры (Рис.3.3).

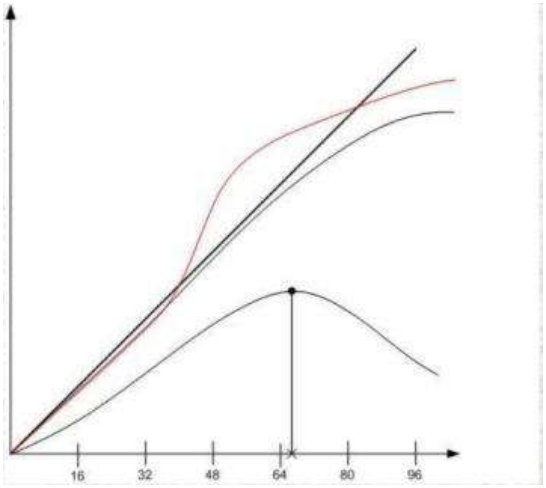


Рис.3.3. Случай превосходства реальной производительности над пиковой (красная линия).

То есть наблюдается полное противоречие теории за счет эффекта работы с кэш-памятью – **суперлинейного ускорения**.

Эффективность параллельных программ. Накладные расходы

Эффективность работы компьютера – отношение реальной производительности (R_{max}) к пиковой (R_{peak}).

$$\text{Эффективность} = \frac{R_{max}}{R_{peak}}$$

Отчасти это можно считать как КПД работы компьютера. В реальности мы имеем:

$$R_{max} \ll R_{peak}$$

Посмотрим на эффективность компьютеров, которые входят в список $Top500$ (Рис.3.4).

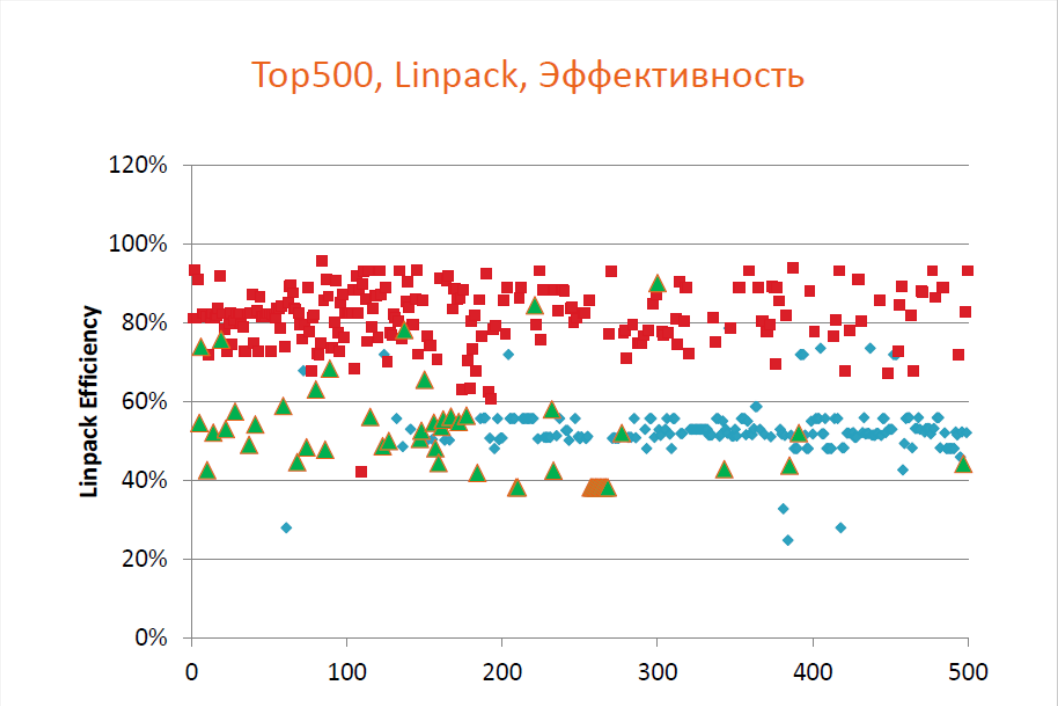


Рис.3.4. Показатели эффективности в списке Top500.

Интересно, что наибольшие показатели эффективности принадлежат **векторным суперкомпьютерам (эффективность $> 90\%$)**.

Еще одной характеристикой для понимания полноты использования процессоров является **эффективность распараллеливания** – отношение ускорения к числу процессоров p.

На практике эта величина очень важна, так как можно сразу оценить целесообразность использования той или иной конфигурации.

Также необходимо оценить **накладные расходы**, которые возникают при запуске параллельной программы.

Для начала введем понятие **стоимости вычислений** – число процессоров p, умноженное на время исполнения программы в параллельном режиме.

$$C = pT_p$$

То есть это по сути стоимость аренды оборудования. Тогда с этой точки зрения **накладные расходы** – это разница между стоимостью и временем исполнения на одном процессоре.

$$T_0 = pT_p - T_1$$

Накладные расходы очень важны, так как именно с ними необходимо бороться. Их источники могут быть разными. И оценить их объем можно до запуска с какой-то точность. Как правило, чем больше мы увеличиваем число доступных процессоров, тем больше возрастают накладные расходы.

Теперь выразим значение T_p и получим время работы параллельной программы:

$$T_p = \frac{T_0 + T_1}{p}$$

Используя определение для ускорения работы программы, получим

$$S = \frac{T_1}{T_p} = \frac{pT_1}{(T_0 + T_1)}$$

И для эффективности распараллеливания имеем

$$E = \frac{S}{p} = \frac{T_1}{(T_0 + T_1)} = \frac{1}{\left(1 + \frac{T_0}{T_1}\right)}$$

Данное выражение позволяет сделать много хороших **выводов**:

- Если T_1 зафиксировано, то при увеличении числа процессоров p эффективность распараллеливания E, как правило, уменьшается из-за роста накладных расходов T_0.
- Если фиксировано число процессоров p, то эффективность распараллеливания E можно увеличить, увеличив время (сложность) решаемой задачи T_1.

Масштабируемость параллельных программ

Масштабируемость – это способность системы увеличивать свою производительность при добавлении новых ресурсов (обычно аппаратных).

Система называется масштабируемой, если она способна увеличивать производительность пропорционально дополнительным ресурсам.

Масштабируемость:

- Компьютера или его подсистем (например, коммуникационной сети при увеличении числа вычислительных узлов),
- Алгоритмов (теоретический анализ параллельных свойств),
- Параллельных программ (относительно конкретного компьютера).

Для нас данное понятие очень важно. У нас есть параллельная программа, система из p процессоров и мы хотим, чтобы с увеличением p время решения программы уменьшалось. В этом случае говорят, что программа масштабируется.

Различают два вида масштабируемости.

Вертикальная масштабируемость – это возможность эффективной замены платформы, на которой функционирует система, на новую, обладающую большей производительностью.

Горизонтальная масштабируемость – это возможность увеличения производительности системы за счет добавления дополнительных программных или аппаратных средств.

Теперь введем понятие **вычислительной сложности задачи (программы)** $\boldsymbol{W}$ – это число шагов (операций) последовательного алгоритма, необходимых для решения задачи на одном процессоре.

$\boldsymbol{W}$ является функцией от размера входных данных.

Если предположить, что каждый основной шаг выполняется за единицу времени, то $W = T_1$.

Рассмотрим примеры расчета вычислительной сложности:

- Последовательное сложение N чисел: $W = N - 1$,
- Скалярное произведение векторов размера N: $W = 2N - 1$,
- Перемножение квадратных матриц $(N * N)$: $W = N^2(2N - 1)$.

Рассмотрим три часто используемых на практике понятий масштабируемости.

Сильная масштабируемость – это зависимость реальной производительности R от числа процессоров p при фиксированной вычислительной сложности задачи ($W = const$).

Сильная масштабируемость показывает способность параллельной программы сохранять эффективность распараллеливания при увеличении числа процессоров для задачи фиксированной сложности.

Слабая масштабируемость – это зависимость реальной производительности R от числа процессоров p при фиксированной вычислительной сложности задачи в пересчете на один процессор $\left(\frac{W}{p} = const\right)$.

Слабая масштабируемость показывает способность параллельной программы сохранять эффективность распараллеливания при увеличении числа процессоров и одновременном сохранении объема работы, приходящейся на каждый процессор.

Масштабируемость вширь – это зависимость реальной производительности R от вычислительной сложности задачи W при фиксированном числе процессоров ($p = const$).

Масштабируемость вширь показывает способность параллельной программы сохранять эффективность распараллеливания на данной конфигурации вычислительной системы при увеличении сложности задачи.

Суммирование элементов массива. Решение задачи на компьютере

Необходимо сложить элементы массива N чисел. На Рис.3.5 показан классический вариант реализации последовательной программы.

```
s = 0.0;
for ( i = 0; i < n; ++i )
   s = s + A[ i ];
```


Рис.3.5. Пример программы сложения элементов массива.

Данный алгоритм обладает такими свойствами, которые не очень удобны для параллельной вычислительной системы.

Для корректной работы параллельного вычисления можно применить схему сдваивания, когда происходит независимое сложение пар элементов (Рис.3.6).

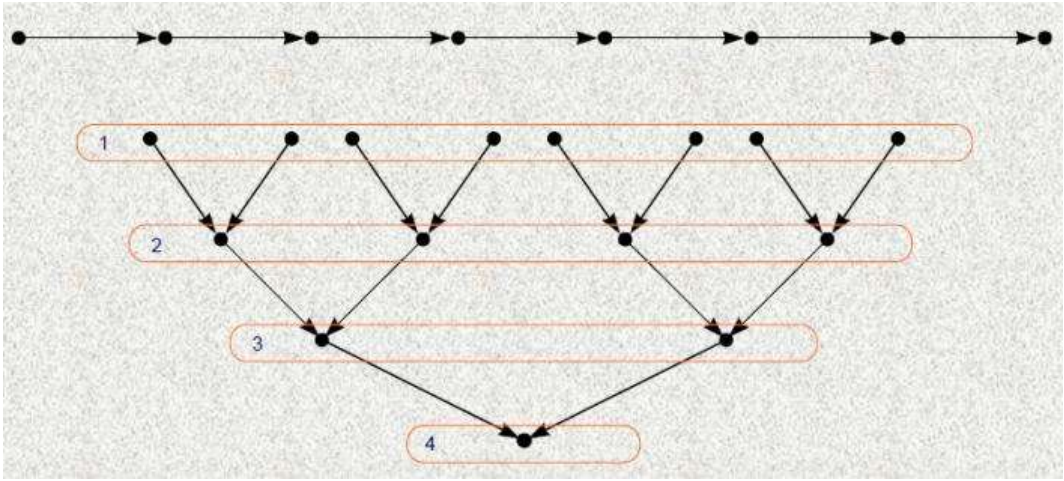


Рис.3.6. Суммирование элементов массива. Схема сдваивания.

Мы изменили алгоритм и тем самым изменили порядок исходного суммирования. Что, вообще говоря, может привести к неожиданным результатам.

Давайте теперь посмотрим на основные этапы, которые мы всегда проходим, если нам нужно решить задача на параллельной вычислительной системе (Рис.3.7).

Рис.3.7. Основные этапы решения задач.

Сначала формулируем некоторую задачу, например, нам нужно решить систему линейных уравнений. Для решения задачи выбираем метод – математику. После выбора метода выбираем алгоритм, то есть такой четкий набор элементарных действий и порядок их следования. Далее выбираем технологию программирования, пишем программу на конкретном ПО и компьютере.

Основная сложность заключается в том, что нынешний компьютер имеет высокую степень параллельности, то есть мы хотим получить большое ускорение в решении задачи, что может быть невозможно из-за того, что в каком-нибудь звене данной цепочки нет необходимой степени параллельности. Например, был выбран метод, в котором нет ресурсов параллельности.

Таким образом, для эффективного распараллеливания задачи во всей цепочке (Рис.3.7) не должно быть ни одного «узкого» места.

Было введено специальное понятие ***Co-Design***, которое расшифровывается как совместный дизайн (построение) всего того, что входит в цепочку – отображение программ и алгоритмов на архитектуру параллельных вычислительных систем.

Лекция 4. Классификация параллельных вычислительных систем. Часть 1

Показатели эффективности и масштабируемости параллельных программ

Понятие масштабируемости является важным в мире суперкомпьютеров, но хочется не просто говорить качественно масштабируется или нет какой-то алгоритм, какая-то программа, а хочется эту масштабируемость измерять или сравнивать. Поэтому часто говорят о введении **метрики масштабируемости**. Ее можно вводить по-разному и один из подходов будет рассмотрен в этой лекции.

В прошлой лекции были рассмотрены **две противоположные тенденции** (выводы из понятия эффективности): при увеличении числа процессоров, эффективность обычно падает, а при увеличении размера задачи или вычислительной сложности задачи, эффективность растет.

Для введения метрики масштабируемости используют такой подход: начинаем увеличивать одновременно с числом процессоров и вычислительную сложность решаемой задачи. То есть эффективность будет иметь тенденцию и к падению, и к росту.

Если мы зададимся каким-то конкретным уровнем эффективности распараллеливания E и будем стараться ее поддерживать за счет увеличения p и W, то получим некую зависимость вычислительной сложности W от числа процессоров p и эта зависимость в какой-то мере может считаться метрикой масштабируемости.

Зависимость W от p называют **функцией изоэффективности**. Для ее вычисления вспомним формулы для эффективности распараллеливания:

$$E = \frac{1}{1 + \frac{T_0}{T_1}} = \frac{1}{1 + \frac{T_0}{W}}$$

$$W = \frac{E}{(1 - E)T_0} = KT_0,$$

где $K = \frac{E}{1-E}$

Получившаяся зависимость размера задачи, необходимой для достижения заданной эффективности распараллеливания, от числа процессоров – **функция изоэффективности**.

Для примера снова обратимся к каскадной схеме (суммирование методом сдваивания), когда вектора делятся на кусочки, каждый процессор вычисляет сумму своего кусочка, а потом получившиеся суммы суммируются методом сдваивания.

Всего получаем $p/2$ таких сумм, которые могут выполняться одновременно и независимо друг от друга. В результате получаем $p/2$ чисел, которые можно суммировать методом сдваивания.

Для p процессоров требуется $\log p$ шагов.

Если мы производим суммирование методом сдваивания, то как для такого алгоритма можно записать функцию изоэффективности?

Для начала выпишем вычислительную сложность суммирования элементов вектора

$$W = T_1 \approx n$$

Теперь оценим время исполнения параллельного варианта программы как

$$T_p \approx \frac{n}{p} + \mathbf{2}\log_2 p$$

Здесь учтено, что сложность пересылки равна сложности арифметической операции.

Тогда ускорение равно

$$S = \frac{T_1}{T_p} = \frac{n}{\left(\frac{n}{p} + 2\log_2 p\right)} = \frac{p}{\left(1 + 2p\log_2 \frac{p}{n}\right)}$$

И эффективность имеет следующий вид

$$E = \frac{S}{p} = \frac{1}{\left(1 + 2p\log_2 \frac{p}{n}\right)}$$

Для оценки накладных расходов сначала найдем стоимость вычислений

$$C = pT_p = \frac{p}{\left(\frac{n}{p} + 2\log_2 p\right)} = n + 2p\log_2 p$$

$$T_0 = pT_p - T_1 = 2p\log_2 p$$

Подставляем значение в функцию изоэффективности и получаем

$$W = KT_0 = 2Kp\log_2 p = O(p\log_2 p)$$

Теперь если мы хотим увеличивать число процессоров от значения p до p' мы поймем, что для поддержания постоянного уровня эффективности распараллеливания E необходимо увеличивать вычислительную сложность задачи в

$$\frac{p'\log_2 p'}{p\log_2 p} \text{ раз.}$$

Посмотрим на это в конкретных цифрах.

Если мы хотим задаться уровнем $E = 0.5$, тогда $K = \frac{E}{1-E} = 1$.

- Пусть $p = 16$, тогда $W = 2Kp\log_2 p = 128$.
- Пусть $p = 64$, тогда $W = 2Kp\log_2 p = 768$.
- Пусть $p = 1024$, тогда $W = 2Kp\log_2 p = 20480$.

На Рис.4.1 показан график зависимости размера данных от числа процессора.

Видно, что если мы хотим добиться уровня распараллеливания близкого к единице ($E = 0.9$), то уровень роста требует больших значений векторов.

Был приведен один из способов оценки уровня масштабируемости с помощью функции изоэффективности. Он позволяет оценить **теоретический потенциал алгоритма**, исходя из соображений о его параллельном исполнении.

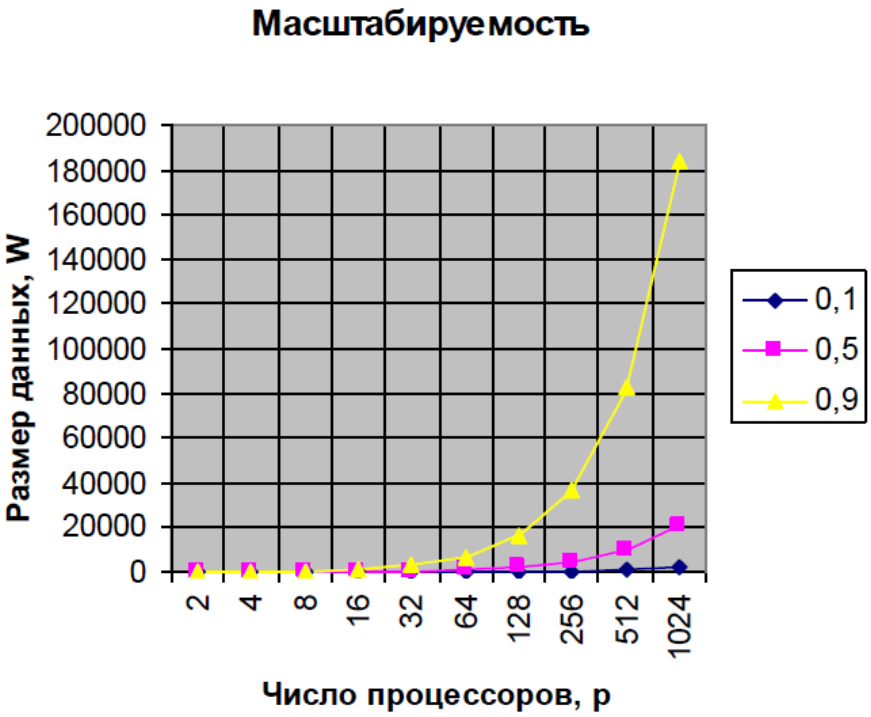


Рис.4.1. Масштабируемость параллельных операций.

Теперь отметим **основные факторы**, которые **мешают достижению хорошей масштабируемости** параллельных приложений.

- Закон Амдала (последовательные части программы).
- Накладные расходы на коммуникации (латентность, пропускная способность).
- Неравномерность загрузки (load balancing) процессоров.
- Предел декомпозиции данных.

Архитектура параллельных вычислительных систем. Классификация Флинна

Начнем с понятия классификации параллельных архитектур. По-видимому, самой ранней и наиболее известной является **классификация архитектур вычислительных систем**, которая была предложена в **1966 году М.Флинном**.

Данная классификация базируется на **понятии потока**, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных Флинн выделяет четыре класса архитектур:

$$SISD, MISD, SIMD, MIMD.$$

SISD (single instruction stream / single data stream) – одиночный поток команд и одиночный поток данных (Рис.4.2). К этому классу относятся, прежде всего, классические **последовательные машины**, или иначе, машины фон-неймановского типа. В таких машинах есть только один поток команд, все команды обрабатываются последовательно друг за другом и каждая команда инициирует одну операцию с одним

потоком данных. Для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка.

Далее на Рис.4.2-4.5, УУ – устройство управления, ПР – процессорный элемент, ПД – память данных.

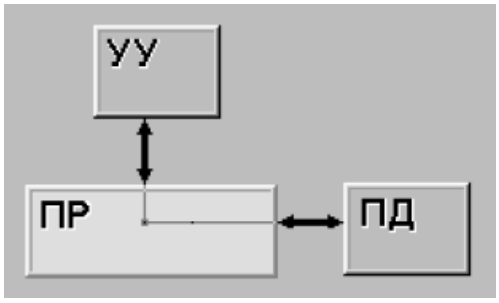


Рис.4.2. Класс SISD.

SIMD (single instruction stream / multiple data stream) – одиночный поток команд и множественный поток данных (Рис.4.3). В архитектурах подобного рода сохраняется один поток команд, включающий, в отличие от предыдущего класса, векторные команды. Это позволяет выполнять одну арифметическую операцию сразу над многими данными – элементами вектора. Способ выполнения векторных операций не оговаривается, поэтому обработка элементов вектора может производиться либо процессорной матрицей, либо с помощью конвейера.

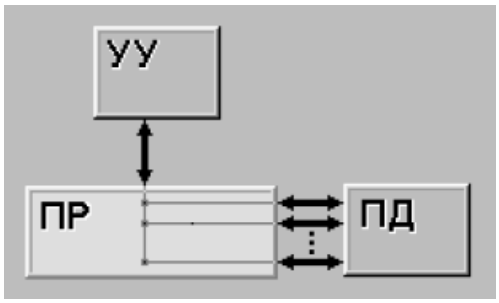


Рис.4.3. Класс SIMD.

MISD (multiple instruction stream / single data stream) – множественный поток команд и одиночный поток данных (Рис.4.4). Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Класс является пустым.

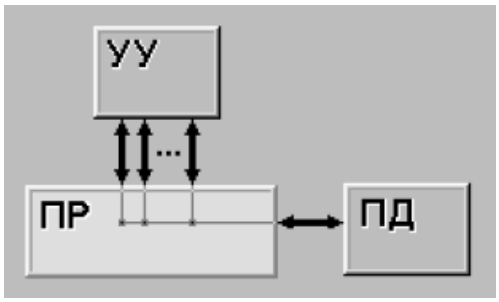


Рис.4.4. Класс MISD.

MIMD (multiple instruction stream / multiple data stream) – множественный поток команд и множественный поток данных (Рис.4.5). Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единых комплекс и работающих каждое со своим потоком команд и данных.

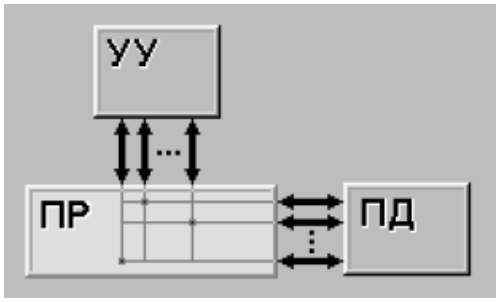


Рис.4.5. Класс MIMD.

Почти все суперкомпьютеры относятся именно к классу MIMD (да и вообще все современные компьютеры).

Компьютеры с общей и распределенной памятью

Компьютеры с **общей памятью** реализуются множеством каких-то обрабатывающих устройств или процессоров, которые представлены множеством экземпляров, а все остальное – в одном экземпляре (оперативная память, образ ОС, подсистемы ввода-вывода и т.д.) (Рис.4.6). Такие компьютеры принято называть ***SMP* – компьютеры**:

***S**hared **M**emory **P**rocessors* или ***S**ymmetric **M**ulti**P**rocessor*

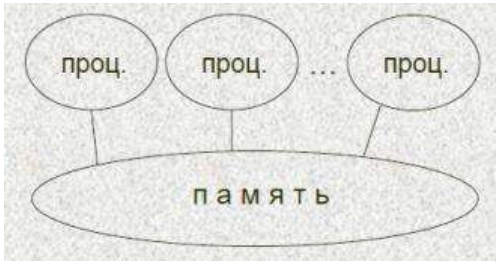


Рис.4.6. Компьютер с общей памятью.

А компьютеры с **распределенной памятью** состоят из вычислительных узлов, каждый из которых является полноценным компьютером со своей памятью, ОС, устройствами ввода-вывода и т.п., взаимодействующих друг с другом через коммуникационную среду (Рис.4.7).

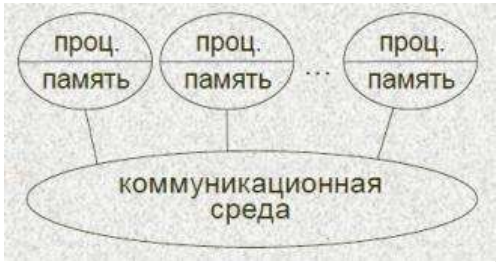


Рис.4.7. Компьютер с распределенной памятью.

Основные плюсы и минусы таких компьютеров:

Общая память.

+ Относительная простота параллельного программирования,
- Сложность увеличения числа процессоров (роста производительности).

Распределенная память.

+ Относительная простота увеличения числа процессоров (роста производительности),
- Сложность параллельного программирования.

Таким образом, если ставится задача построения вычислительной системы с максимально возможной производительностью, то нужны компьютеры с распределенной памятью.

А если требуется задача эффективного и простого использования, то больше для этого соответствуют компьютеры с общей памятью.

Зачастую сейчас строятся гибридные системы, когда компьютер состоит из отдельных узлов, каждый из которых строится со своей локальной памятью, а в целом в компьютере получается память распределенная.

Теперь рассмотрим более конкретно вопрос о том, почему же компьютеры с распределенной памятью программировать сложнее, чем компьютеры с общей памятью?

Для этого сначала будем использовать пример реализации вычислительного алгоритма на однопроцессорном компьютере с **общей памятью** (Рис.4.8):

$$A_i = B_i + C_i x, i = 1, \dots, n$$

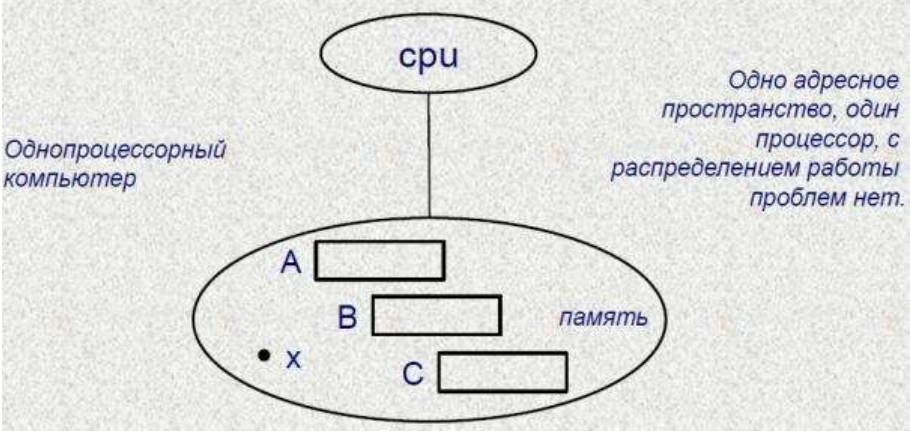


Рис.4.8. Реализация алгоритма на однопроцессорном компьютере.

Все данные находятся в одном блоке памяти. Не нужно распределять операции и сами данные.

В случае многопроцессорного компьютера с общей памятью мы делим операции алгоритма и раздаем их различным процессорам (Рис.4.9).

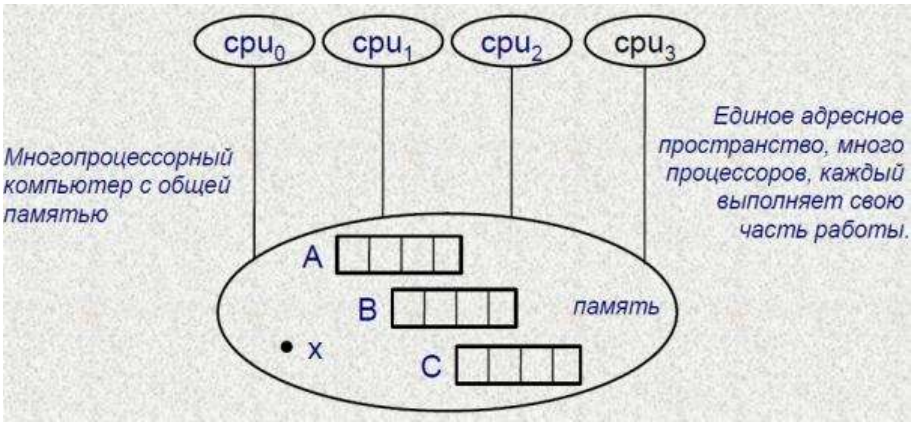


Рис.4.9. Многопроцессорный компьютер с общей памятью.

Теперь обратимся к компьютерам с **распределенной памятью** (Рис.4.10). В данном случае кроме распределения операций нам требуется распределить и данные. Пусть все данные алгоритма лежат в оперативной памяти одного процессора. Понятно, что это плохо, так как для начала выполнения алгоритма на других процессорах необходимо большое количество пересылок, что снижает общую производительность.

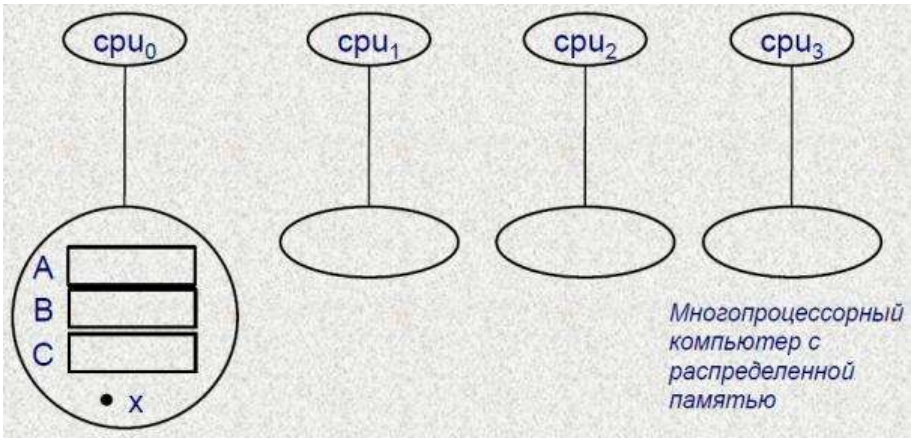


Рис.4.10. Случай, когда все данные лежат в оперативной памяти одного процессора.

Давайте распределим данные по-другому (Рис.4.11). Ничего хорошего мы снова не получим, так как любому процессору потребуются все элементы, которые раскиданы по памяти различных процессоров.

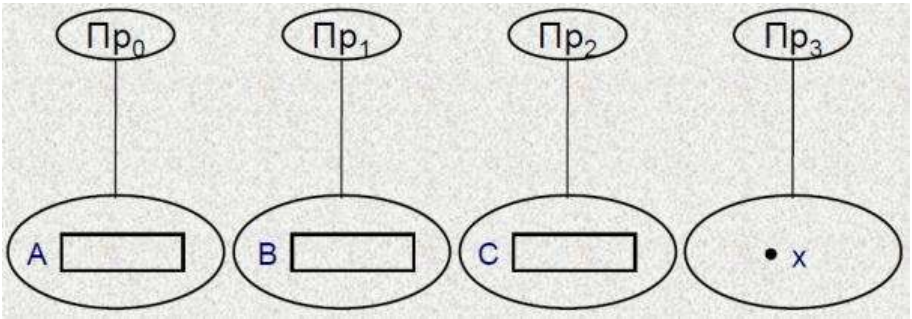


Рис.4.11. Хранение данных в разных частях памяти процессоров.

Другой способ распределения: разобьем вектора A, B и C на части, каждому процессору назначим свою часть векторов, причем распределим операции этого алгоритма согласовано с распределением данных, то есть тот процессор, кому попадется i – тая итерация и будет хранить i – тые элементы векторов A, B и C (Рис.4.12).

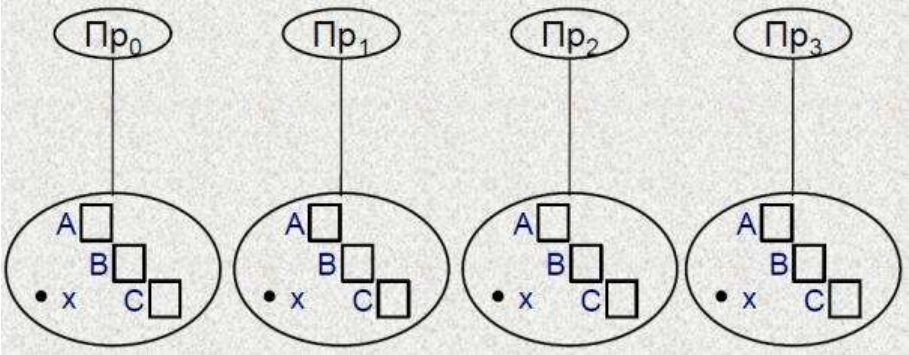


Рис.4.12. Разбиение i-тых элементов векторов по памяти процессоров.

То есть мы получаем, что каждый процессор может начать выполнять свою часть операций без пересылок, что обеспечивает высокую производительность работы системы.

Далее посмотрим на способ записи алгоритмов в виде кода на каком-либо языке программирования.

Например, мы имеем следующий последовательный алгоритм:

```
N=1000;
for (i=0; i<N; i++)
   A[i]=B[i]+C[i]*x;
for (i=0; i<N; ++i)
   A[i]=(A[i]+C[i])*B[N-i-1];
```

На последовательном компьютере все данные (массивы A, B, C и скаляр x) лежат в общей памяти, различий в классах переменных нет.

Теперь для того, чтобы распараллелить эту задачу, во-первых, нужно убедиться, что эти циклы являются параллельными, то есть их итерации не содержат информационных зависимостей (когда одна операция использует то, что было подсчитано другой операцией).

Во-вторых, если мы рассматриваем компьютер с **общей памятью**, то необходимо распределить итерации циклов. Способов для этого много. Сначала рассмотрим вариант **блочного распределения итераций**, когда все операции программы делим на имеющиеся число процессоров:

```
N=1000;
n_pr=4;
size=N/n_pr;
ibegin=proc_id*size;
iend=ibegin+size;
for(i=0; i<N; i++)
```

```
   A[i]=B[i]+C[i]*x;
for(i=0; i<N; ++i)
   A[i]=(A[i]+C[i])*B[N-i-1];
```

Возникает понятие **классов переменных**. В данном случае переменные $proc\_id$, i, $ibegin$, $iend$ должны быть локальными (своя копия на каждом процессоре), а переменные A, B, C, x должны быть распределенными (одна копия данных для всех процессоров).

Другой вариант – **циклическое распределение итераций**, когда нулевая итерация идет нулевому процессору, первая – первому и т.д. до последнего номера процессора. Потом начинаем снова циклически с нулевого и так до тех пор, пока все итерации не распределим:

```
N=1000;
n_pr=4;
for(i=proc_id; i<N; i+=n_pr)
   A[i]=B[i]+C[i]*x;
for(i=proc_id; i<N; i+=n_pr)
   A[i]=(A[i]+C[i])*B[N-i-1];
```

Переменные $proc\_id$, i должны быть локальными (своя копия на каждом процессоре), а переменные A, B, C, x должны быть распределенными (одна копия данных для всех процессоров).

Что касается написания программ на компьютерах с **распределенной памятью**, то помимо распределения операций требуется также распределение данных по локальным модулям памяти разных процессоров и организация обменов данными.

```
N=1000;
n_pr=4;
N1=N/n_pr;
<обмен 1> // рассылка частей массивов A, B, C и скаляра x
for(i=0; i<N1; ++i)
   A[i]=B[i]+C[i]*x;
<обмен 2> // обмен частями B: процессы 0<->3 и 1<->2
for(i=0; i<N1; ++i)
   A[i]=(A[i]+C[i])*B[N-i-1];
<обмен 3> // пересылка результирующего массива A
```

Таким образом, для эффективного программирования компьютеров с **общей памятью** необходимо:

1. Найти в программе ресурс параллелизма.
2. Распределить операции по исполнительным устройствам.
3. Разграничить доступ к данным в общей памяти.

А для эффективного программирования компьютеров с **распределенной памятью** необходимо:

1. Найти в программе ресурс параллелизма.

2. Распределить данные по модулям памяти вычислительных узлов.
3. Распределить операции по исполнительным устройствам.
4. Согласовать распределение данных с параллелизмом вычислений.
5. Организовать необходимые пересылки данных.

Лекция 5. Суперкомпьютерный комплекс МГУ

Суперкомпьютер «Чебышёв»

В этой лекции мы подробнее остановимся на суперкомпьютерном комплексе МГУ, посмотрим что в него входит.

На Рис.5.1 показана история появления вычислительной техники в МГУ. Все основные российские машины ставились на территории университета.

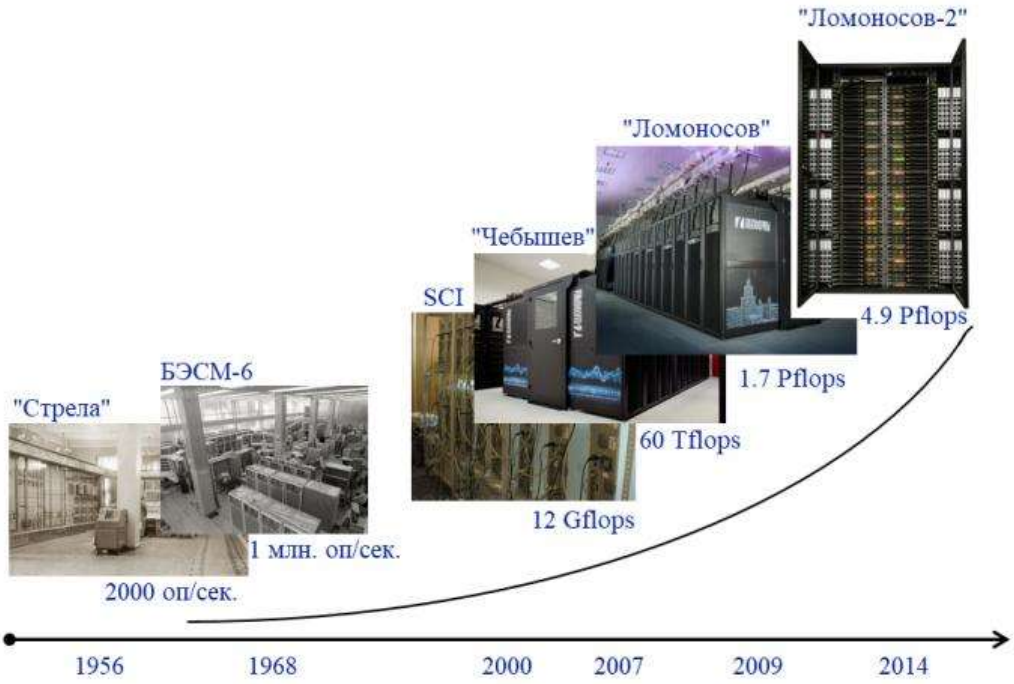


Рис.5.1. История появления вычислительной техники в МГУ.

Ниже приведен состав всех суперкомпьютеров

- Суперкомпьютер «Ломоносов-2», 4.9 $PFlop/s$
- Суперкомпьютер «Ломоносов», 1.7 $PFlop/s$
- Суперкомпьютер «Чебышёв», 60 $TFlop/s$
- Суперкомпьютер Polus, 56 $TFlop/s$
- Суперкомпьютер IBM Blue Gene/P, 27 $TFlop/s$

Первые два суперкомпьютера работают на весь университет и на другие организации, нижние две машины – факультетские машины, которые работают в основном на науку и учебу на факультете. А суперкомпьютер «Чебышёв» сейчас уже не работает. Он остался в этом списке, так как на его примере удобно рассказывать об устройстве суперкомпьютеров.

Начнем с характеристик и свойств суперкомпьютера **СКИФ МГУ «Чебышёв»**.

- 1 место в 8-ой редакции списка *Top50* наиболее мощных компьютеров СНГ (март 2008 года).
- 36 место в 31-ой редакции списка *Top500* наиболее мощных компьютеров мира (июнь 2008 года).

- Пиковая производительность: 60 $TFlop/s$
- Производительность на Linpack: 47.32 $TFlop/s$ (79% пиковой), матрица $740000 * 740000$
- 625 вычислительных узлов, 1250 процессоров, 5000 процессорных ядер
- 42 стойки: 14 вычислительных и 28 инфраструктурных
- Помещение 98 м^2
- Общий вес оборудования: более 30 тонн
- Энергопотребление вычислительной части 330 КВт, всего комплекса в пике до 720 КВт
- Система бесперебойного электропитания
- 10 минут автономной работы
- Система охлаждения
- Звукоизоляция
- Система автоматического газового пожаротушения

Теперь перейдем к начинке компьютера.

Вычислительные узлы:

- Процессоры:
  1250 Intel E5472 3.0 ГГц Harpertown
- Блэйд-шасси T-Blade («Т-Платформы»)
  Форм-фактор 5 U
  До 10 вычислительных узлов
- Оперативная память:
  529 x 8 ГБ, бездисковые
  64 x 8 ГБ, 160 ГБ HDD
  32 x 16 ГБ, 160 ГБ HDD
  8 x 32 ГБ, 160 ГБ HDD

Коммуникационная сеть:

- DDR InfiniBand
  Mellanox MT25418 NIC
  FatTree
  SilverStorm 9120 – базовые коммутаторы
  Flextronic F-X430046 – листовые коммутаторы
- Характеристики
  $1.3 - 1.95$ μs латентность
  1.7 ГБ/с пропускная способность

Далее рассмотрим способ построения коммуникационной сети ***Fat Tree*** (Рис.5.2). В суперкомпьютерах применяются специальных топологии. Во-первых, с точки зрения теории графов это вовсе не дерево, так как здесь есть циклы. Во-вторых, часть связей тут толще, чем другие. Внизу на 630 вычислительных узлов. 12 узлов

подключены к одному маленькому коммутатору на 24 порта, а от него вверх ведут линки к коммутатору верхнего уровня (двойные линки).

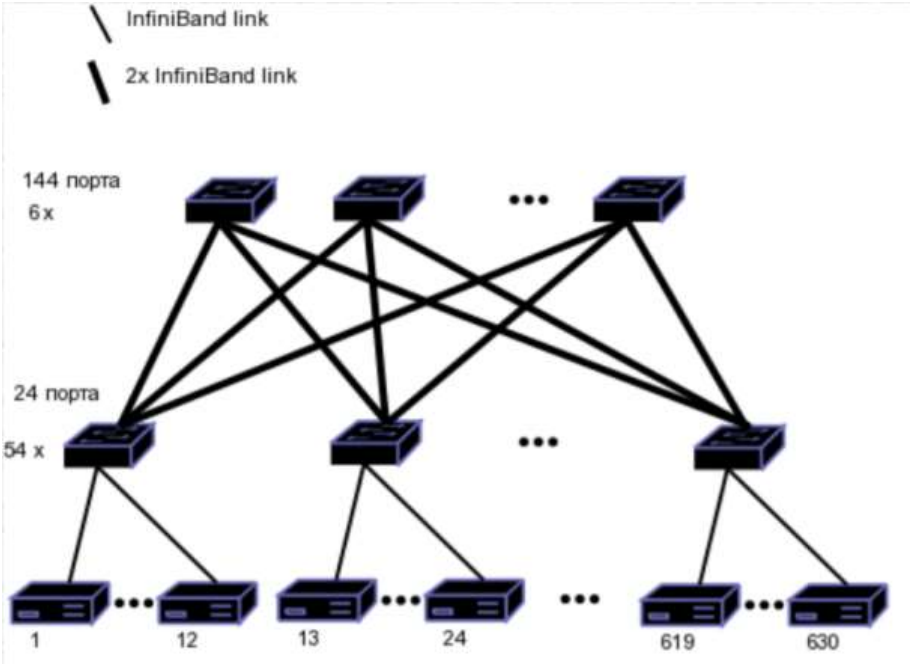


Рис.5.2. Коммуникационная сеть $Fat\ Tree$.

Такая конструкция позволяет разбить произвольным образом вычислительные узлы на пары и все эти пары между собой могут обмениваться на полной скорости. Это называется **полной бисекционной пропускной способностью**.

Кроме основной сети существуют и **вспомогательные сети**:

- $Gigabit\ Ethernet$: коммутаторы $Force10\ C300$ и $Force10\ S2410$
- Управляющая сеть $ServeNet + IPMI$

Также места **хранения данных**:

- 60 ТБ распределенное отказоустойчивое сетевое хранилище $T\text{-}Platforms\ ReadyStorage\ AvctiveScale\ Cluster$
- 15 ТБ локальных дисков на узлах
- Ленточное хранилище $Quantum\ Scalar\ i500$

Кроме того важная часть комплекса это **система охлаждения**, которая состояла из 8 кондиционеров $APC\ InfraStruXure\ ACR502$ и 3 холодильных машин $Liebert\text{-}Hiross\ SLH\ 023$.

И существует понятие **горячего коридора**, когда используются меньший объем охлаждаемой части помещения, более тесная компоновка системы и встречные воздушные потоки.

Теперь посмотрим на схему компоновки системы суперкомпьютера СКИФ МГУ «Чебышев» (Рис.5.3).

Рис.5.3. Компоновка системы СКИФ МГУ «Чебышев».

Что касается горячего коридора, то вычислительная техника берет холодный воздух снаружи, прогоняет его через себя, то есть охлаждается им, и потом отдает его внутрь коридора, где становится очень жарко. Далее кондиционеры забирают оттуда воздух и выбрасывают наружу (Рис.5.4).

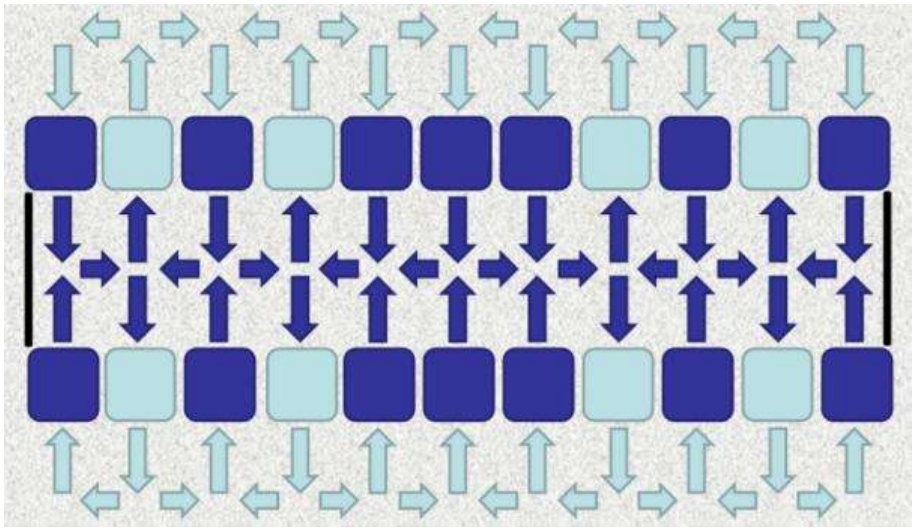


Рис.5.4. Горячий коридор системы и потоки воздуха.

Иногда конструируют обратную систему, то есть помещение с холодным воздухом внутри, а с горячим снаружи. Это может быть нужно тогда, когда высокая температура требуется снаружи системы для правильного функционирования комплекса.

Суперкомпьютер «Ломоносов»

Теперь перейдем к суперкомпьютеру «Ломоносов». Сначала рассмотрим его основные характеристики.

- Пиковая производительность – $1700.21\ TFlop/s$

- Производительность (Linpack) – 901.90 $TFlop/s$
- Эффективность 53%
- Вычислительных узлов (Intel) – 5 104
- Вычислительных узлов (ГПУ) – 1 065
- Вычислительных узлов (PowerXCell) – 30
- Процессоры Intel Xeon 5570, 5670 – 12 346
- NVIDIA Tesla X2070 – 2130
- Число процессорных ядер (x86) – 52 168
- Число процессорных ядер (ГПУ) – 954 240
- Оперативная память – 92 Тбайт
- Коммуникационная сеть – $QDR\ Infiniband$ / 10 GE
- Система хранения данных – 1.75 ПБайт, $Lustre, NFS, \ldots$
- Операционная система – $Clustrx\ T\text{-}Platforms\ Edition$
- Занимаемая площадь (вычислитель) – 252 м^2
- Энергопотребление (вычислитель) – 2.8 МВт

С помощью блочной схемы можно представить это так (Рис.5.5):

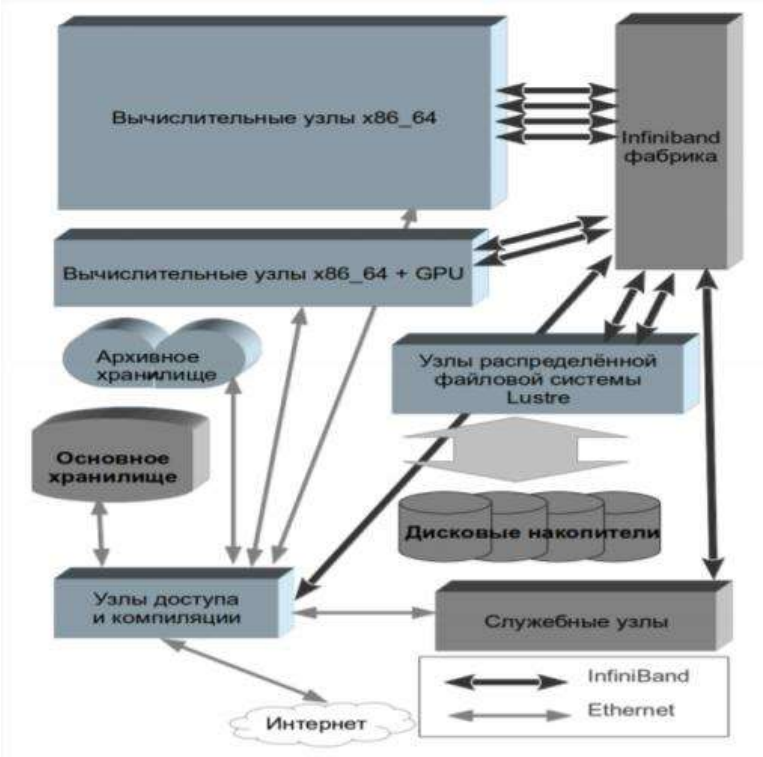


Рис.5.5. Схема устройства суперкомпьютера «Ломоносов».

Коммуникационная сеть у «Ломоносова» также построена по системе $Fat\ Tree$. Но суперкомпьютер строился в несколько приемов, поэтому можно четко видеть первую

очередь с системой $Fat\,Tree$. Потом был доделан 2-й раздел с графическими процессорами и появилось второе «толстое дерево» (Рис.5.6).

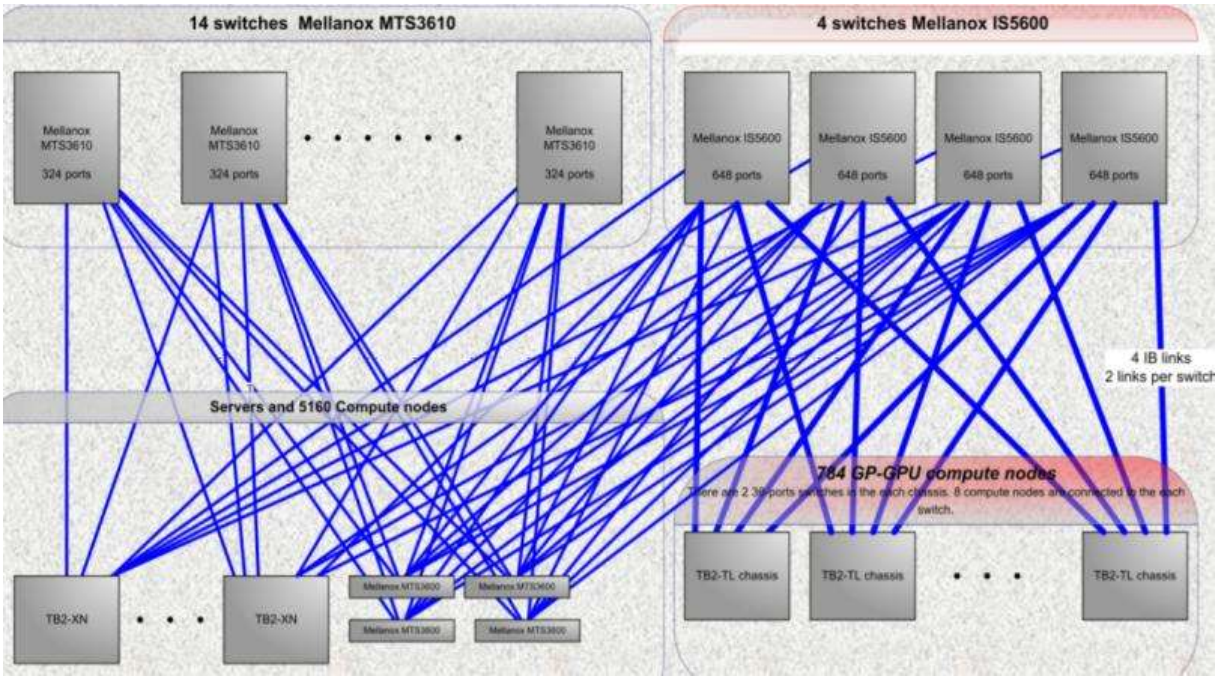


Рис.5.6. Коммуникационная сеть «Ломоносова».

Достаточно сложно понять, что происходит на Рис.5.6, однако принципы остаются такими же: сколько в коммутатор связи приходит снизу, столько же в сумме связей должно уходить наверх.

Суперкомпьютер «Ломоносов-2»

Перейдем к флагману суперкомпьютерного комплекса МГУ «Ломоносову-2». По традиции представим основные характеристики суперкомпьютера (на 2020 год).

- Пиковая производительность – 5 505 $TFlop/s$
- Производительность (Linpack) – 2 478 $TFlop/s$
- Эффективность 50%
- Вычислительных узлов – 1 722
- Центральные процессоры – $Intel\ Haswell\text{-}EP\ E5\text{-}2697v3$,
  $Intel\ Xeon\ Gold\ 6126$,
  $Intel\ Xeon\ Gold\ 6142$,
  $Intel\ Xeon\ Gold\ 6240$
- Ускорители – $Nvidia\ Tesla\ K40M$,
  $Nvidia\ P100$,
  $Nvidia\ V100$
- Оперативная память – 114 ТБайт
- Коммуникационная сеть – $FDR\ Infiniband$ / 10 GE
- Система хранения данных – 1392 ТБайт, $Lustre, Panasas$
- Операционная система – $CentOS\ 7$

«Ломоносов-2» занимает около $5000\ м^2$. Такая большая площадь объясняется тем, что рассчитывается добавление новых узлов в систему со временем (максимум можно добавить до 16000 вычислительных узлов).

Система охлаждения у суперкомпьютера **содержит воду**, то есть на процессор ставится пластина со входами, через которые течет вода. Мы помним, что рабочая температура процессора порядка 70 градусов Цельсия, значит мы можем охлаждать его водой с температурой 40 градусов Цельсия.

Конечно, в системе имеются и холодильные машины, правда их не очень много.

Теперь о **вычислительной сети** суперкомпьютера «Ломоносов-2».

- Тип связи – $FDR\ InfiniBand\ (56\ Gbit/s)$
- Топология – $Flattened\ Butterfly$
- Число портов коммутатора – 36
- Размерность – 8 x 8 x 8 x 4
- Относительная ширина связей – 1 x 1 x 1 x 2
- Число вычислительных узлов, подключенных к одному коммутатору – 8
- Максимальное количество узлов – 16 384

Здесь выбран такой тип топологии, на котором возможно относительно дешево наращивать вычислительные мощности.

Давайте рассмотрим детальнее технологию $Flattened\ Butterfly$.

Есть коммутаторы, которые соединены все со всеми. Из каждого коммутатора выходят три связи (зеленые линии). Теперь размножим эту конструкцию. И вводим двумерную конфигурацию размерами 4 x 4 (Рис.5.7). Такая конструкция хороша тем, что любые два коммутатора доступны за два шага (добавление синих линии по второй координате).

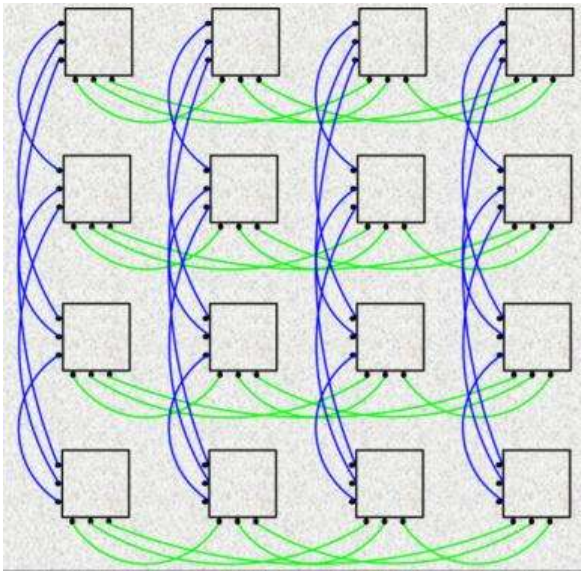


Рис.5.7. Технология 2D Flattened Butterfly.

Более того, данная конструкция легко масштабируется.

На сегодняшний день в рейтинге $Top50$ машин СНГ суперкомпьютер «Ломоносов-2» занимает второе место, уступая лишь «Кристофари» от компании СберБанка.

Что касается мирового рейтинга $Top500$, то позиции университетских суперкомпьютеров следующие (Рис.5.8):

«Ломоносов»

Дата	Производительность	Место
Ноябрь 2009 г.	0.4 PFlop/s	12
Ноябрь 2018 г.	1.7 PFlop/s	485
Июнь 2020 г.	1.7 PFlop/s	-

«Ломоносов-2»

Дата	Производительность	Место
Ноябрь 2014 г.	2.6 PFlop/s	22
Июнь 2020 г.	4.9 PFlop/s	130

Рис.5.8. Рейтинг Top500 для «Ломоносова» и «Ломоносова-2».

Подводя итоги, посмотрим на те области, где сейчас задействован суперкомпьютер «Ломоносов-2» (Рис.5.9).

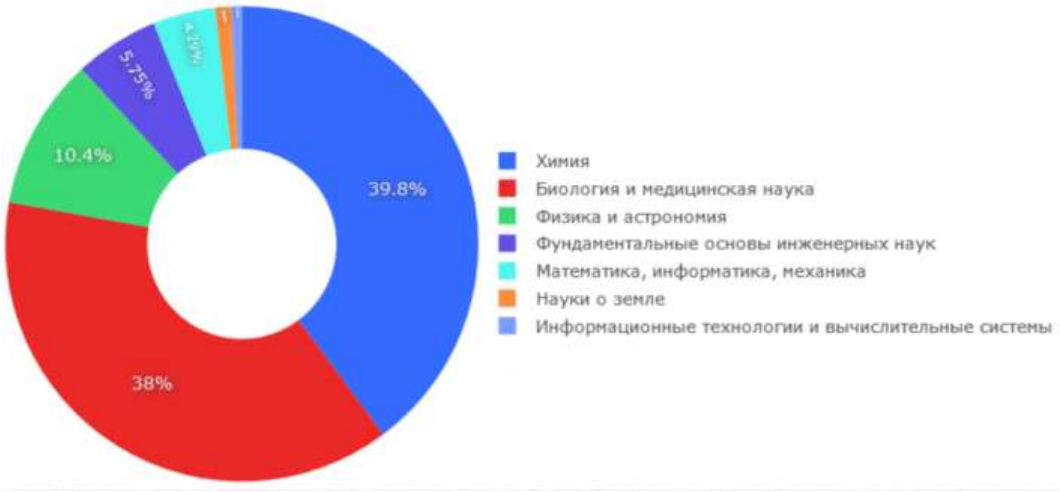


Рис.5.9. Распределение областей исследования на суперкомпьютере «Ломоносов-2».

Лекция 6. Классификация параллельных вычислительных систем. Часть 2

Архитектура SMP и NUMA

В начале этой лекции продолжим разговор про компьютеры с общей памятью. Иногда говорят, что классические SMP – компьютеры обладают архитектурой $\boldsymbol{UMA}$ ($\boldsymbol{Uniform\ Memory\ Access}$), обеспечивая одинаковый доступ любого процессора к любому модулю памяти.

Как мы помним, одним из основных недостатков компьютеров с общей памятью является физическая невозможность присоединения к общей памяти большого числа процессоров. Для того, чтобы получить некий компромисс, то есть, с одной стороны, сохранить общее адресное пространство, а с другой стороны, попытаться в это пространство подсоединить как можно большее число вычислительных узлов, пришли к видоизменению классической UMA архитектуры путем **отказа от однородности доступа к памяти,** то есть произвели переход к **архитектуре $\boldsymbol{NUMA}$.**

$$\boldsymbol{Non\ Uniform\ Memory\ Access}$$

Исторически первым таким компьютером был компьютер $\boldsymbol{Cm^*}$ в конце 70-х годов 20-го века. Он состоял из набора вычислительных узлов (кластеров) (Рис.6.1).

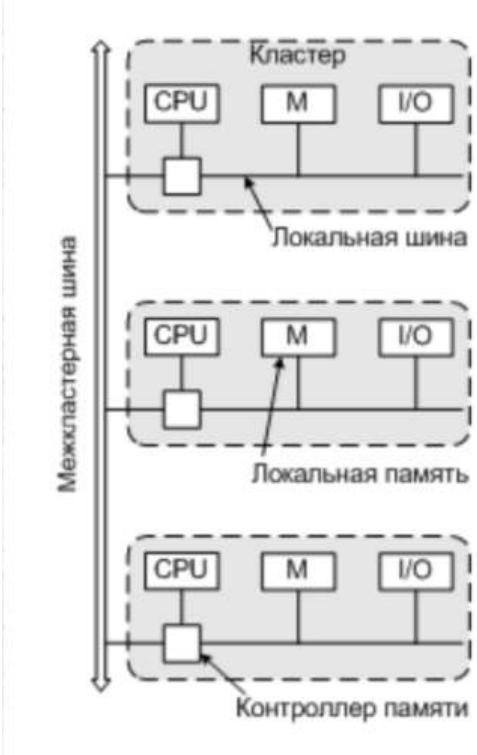


Рис.6.1. Архитектура NUMA, компьютер Cm.*

Главным при общении кластеров оказывался контроллер памяти, который по старшим разрядам адреса определял, в каком модуле памяти хранятся нужные данные. Далее запрос выставляется либо на локальную шину, либо на межкластерную шину.

Причем у данной архитектуры межкластерная шина была узким местом. В случае достаточно большого количества различных кластеров получается большая нагрузка на межкластерную шину, что не позволяет наращивать число процессоров для эффективного выполнения программ.

Поэтому были предложены другие виды архитектуры $NUMA$, где вместо межкластерной шины использовался набор коммутаторов или переключателей. Компьютер $\boldsymbol{BBN\ Butterfly}$ имел архитектуру как показано на Рис.6.2. Общение процессоров с блоками памяти осуществляется через иерархию переключателей из двух групп. К каждому переключателю подсоединены 4 процессора и каждый переключатель подсоединен к 4 переключателям второго уровня.

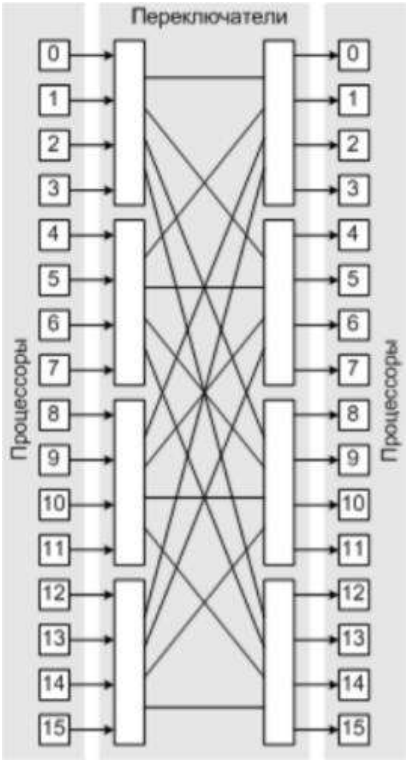


Рис.6.2. Архитектура BBN Butterfly.

В данном компьютере неоднородность архитектуры $NUMA$ выливалась в то, что доступ к локальным данным осуществлялся за $2\ \mu s$, а к удаленным – около $6\ \mu s$.

Все было хорошо, пока не появилась **кэш-память**. Проблема появилась из-за того, что если процессор работает с данными в своей кэш-памяти, он изменяет значения переменных, и потом возвращает обратно их в общую память, и возможны случаи, когда какой-то другой процессор в это же время может начать работу с устаревшими данными из общей памяти, куда еще не успели «подтянуться» новые данные. Название этой проблемы – $\boldsymbol{cache\ coherence\ problem}$.

Для борьбы с ней стали реализовываться аппаратные особенности процессоров, которые делают аппаратные согласования кэшей. Компьютеры, где была разрешена эта проблема на аппаратном уровне, называются по архитектуре $\boldsymbol{ccNUMA}$:

$$\boldsymbol{cache\ conerent\ Non\ Uniform\ Memory\ Access}$$

Таким образом, сейчас строятся компьютеры по $ccNUMA$ архитектуре. И когда их используют отвечают на вопрос о том, **насколько дольше обращение к удаленным данным, чем к локальным?** Если обращение к удаленным данным на какие-то проценты больше, чем к данным в локальной памяти, то это нестрашно. Однако, в реальности это не так. Чаще всего разница во времени составляет в $2-7$ раз, что существенно влияет на производительность.

В качестве примера компьютера на $ccNUMA$ архитектуре рассмотрим компьютер ***Hewlett-Packard Superdome***, линейка которого выпускается с 2000 года. Стандартная конфигурация этих компьютеров объединяется от 2 до 64 процессоров с возможностью расширения системы. В компьютере до 256 Гбайт оперативной памяти и использовались два варианта процессоров:

- $PA\text{-}8600, PA\text{-}8700, PA\text{-}8900$
- $Intel\ IA/64: Itanium, Itanium\ 2$

Структура этих компьютеров состояла на основе базового понятия вычислительной ячейки (cells) (Рис.6.3). Она состояла из 4 процессоров. Основным компонентом ячейки являлась сложная схема контроллера ячейки для общения всех со всеми.

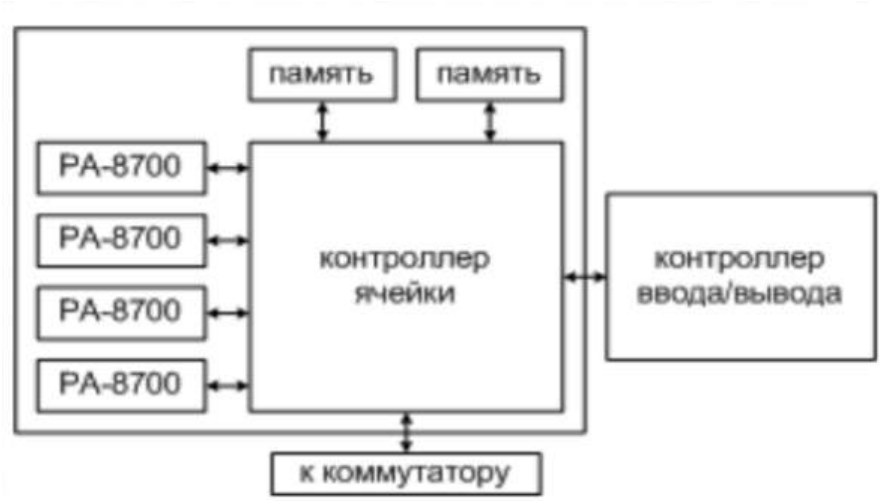


Рис.6.3. Структура компьютера Hewlett-Packard Superdome.

Пропускная способность канала процессора-контроллера по 2 Гбайта в секунду, как и канала банка памяти-контроллера, а канала контроллера-внешнего коммутатора – 8 Гбайт/сек.

На базе этих ячеек строится базовая конфигурация, которая состоит из 2 стоек, в каждой из которых два коммутатора (Рис.6.4). То есть в базовой конфигурации у нас 4 коммутатора, 16 ячеек и 64 процессора. Каждый коммутатор имеет 8 портов: 4 порта для связи с ячейками, 3 порта для связи с коммутаторами и 8-ой порт остается свободным.

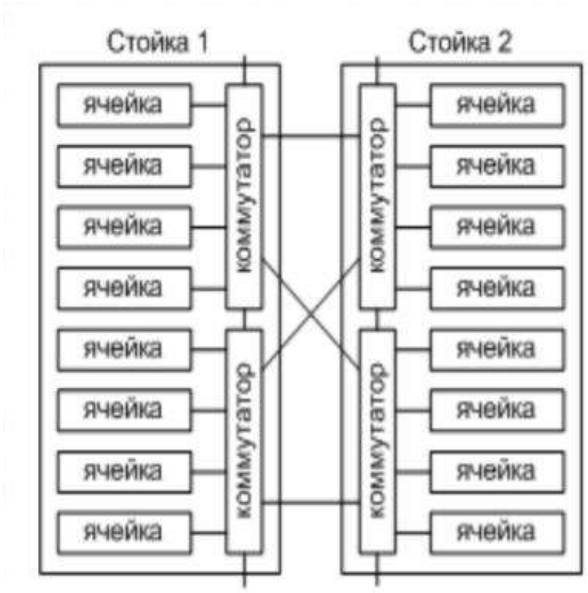


Рис.6.4. Базовая конфигурация компьютера.

В этом компьютере получались **три вида задержек** при обращении процессора к памяти:

- Процессор и память располагаются в одной ячейки – в этом случае задержка минимальна
- Процессор и память располагаются в разных ячейках, но обе эти ячейки подсоединены к одному и тому же коммутатору
- Процессор и память располагаются в разных ячейках, причем обе эти ячейки подсоединены к разным коммутаторам – в этом случае запрос должен пройти через два коммутатора, и задержки будут максимальными.

В $HP\ Superdome$ средняя задержка при переходе от 4 до 64 процессоров возрастает всего в 1.6 раза, что является очень хорошим показателем.

Оценим производительность этих компьютеров на процессоре $PA\text{-}8700$, который имеет тактовую частоту 750 МГц и может выполнять 4 операции за такт, что дает пиковую производительность 3 Гфлопс. Следовательно, пиковая производительность 64 процессорной конфигурации - 192 Гфлопс.

Суммируя результаты компьютеров с **общей памятью**, перечислим основные **причины снижения производительности компьютеров** с данной архитектурой:

1. Закон Амдала.
2. $cc\boldsymbol{NUMA}$ (неоднородность доступа к памяти).
3. $\boldsymbol{cc}NUMA$ (необходимость согласования кэш-памяти).
4. Конфликты при обращении в память.
5. Сбалансированность вычислительной нагрузки процессоров.
6. Производительность отдельных процессоров.
7. …

Далее поговорим про закон Амдала, касательно компьютеров с общей памятью. Напомним, что закон определяется такой формулой:

$$S \leq \frac{1}{\alpha + \frac{1-\alpha'}{p}},$$

где $\alpha -$ доля последовательных операций, $p -$ число процессоров в системе.

Время выполнения программы определяется долей последовательный операций программы. При использовании моделей параллельных программ с общей памятью **возникают дополнительные участки последовательного кода**, связанные с синхронизацией доступа к общим данным, например, **критические секции**.

И что касается **сбалансированности вычислительной нагрузки процессоров,** то мы имеем следующую ситуацию: чем больше используется процессоров, тем меньше область вычисления для каждого процессора. Тогда можно прийти к тому, что каждому процессору достанется слишком маленькая доля операций и больше времени будет тратится на общение процессоров, чем на вычисление в своей области.

В случае прямоугольной и треугольной областях можно добиться сбалансированности вычислительных нагрузок процессоров.

Компьютеры с распределенной памятью

Компьютеры с распределенной памятью строятся на основе каких-то процессоров и при помощи каких-то коммуникационных сетей. Для примера посмотрим на следующие компьютеры с распределенной памятью:

- $Intel\ Paragon$: $Intel\ i860$, двумерная прямоугольная решетка.
- $IBM\ SP1/SP2$: $IBM\ Power$, высокоскоростной коммутатор каждый-с-каждым.
- «Ломоносов»: $Intel\ Xeon + NVIDIA\ Tesla.$ $QDR\ Infiniband$, топология «толстое дерево».

Одна из характеристик коммуникационной сети – **топология**. Под ней понимается конфигурация графа этой сети и она определяет взаимное расположение узлов и каналов связи этого компьютера.

В архитектуру сети помимо топологии входят также **алгоритмы маршрутизации и управления потоками**.

Рассмотри несколько простых примеров возможный топологий (Рис.6.5). Первая топология – **линейка**. Все узлы выстраиваются в линейку, каждый узел соединен со следующим узлом. Для системы из p узлов нужно $p - 1$ соединение, средняя длина пути между двумя узлами равна $p/3$.

Далее – **кольцо**. Для построения системы из p узлов нужно p соединений, средняя длина пути между двумя узлами равна $p/6$. Увеличивается отказоустойчивость за счет того, что передача сообщений может идти по двум направлениям.

Следующая топология – **звезда**. Общение только через центральный узел, $p - 1$ соединение, длина пути между концевыми узлами равна 2. Характерна большая нагрузка на центральный узел.

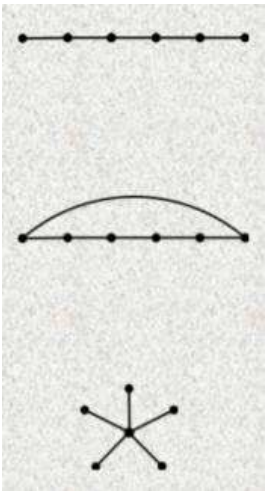


Рис.6.5. Топологии сверху вниз: линейка, кольцо и звезда.

Перейдем от одномерных вариантов к двумерным (Рис.6.6). Таким как **двумерные решетки, торы** и т.д. На практике используются тороидальные топологии за счет добавления связей между границами. А в реальных суперкомпьютерах используют трехмерные, четырех, пяти и шестимерные торы.

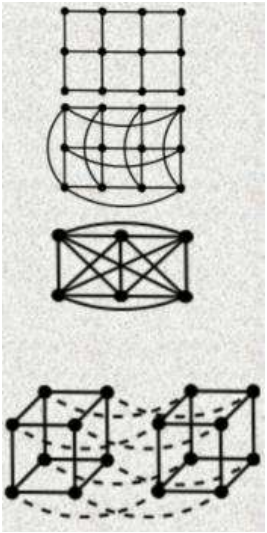


Рис.6.6. Топологии сверху вниз: двумерная решетка и тор, полносвязная топология и два двоичных гиперкуба.

Также можно отметить **полносвязную топологию** – каждый узел имеет связь с каждым другим узлом; для соединения p узлов требуется $p(p-1)/2$ связей. Обычно

такие топологии используются в небольших компьютерах или в составе более сложных топологий.

Следующий вариант – **двоичный гиперкуб**, когда в n-мерном пространстве помещается $p = 2^n$ узлов в вершины единичного n-мерного куба. Каждый узел соединен с соседом вдоль каждого из n измерений – всего $log_2 p$ соединений; максимальное расстояние между вершинами n-мерного гиперкуба равно n. Связанные два двоичных гиперкуба представляют **четырех-мерный двоичный гиперкуб**.

Более сложные варианты, которые встречаются в современных суперкомпьютерах, приведены на Рис.6.7.

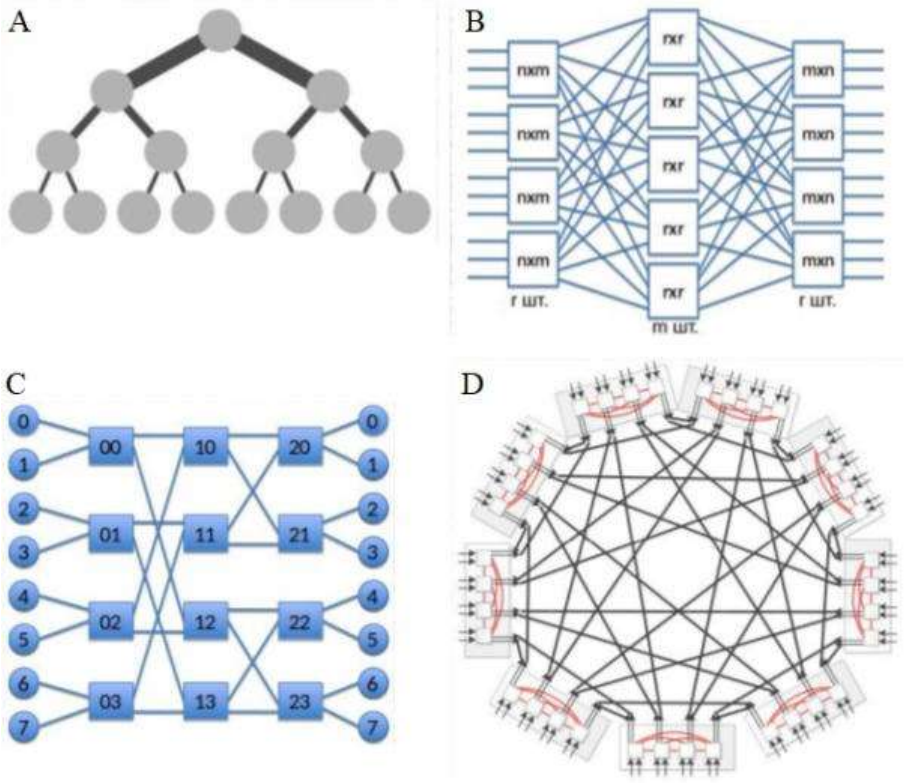


Рис.6.7. ***A*** *– толстое дерево,* ***B*** *– сети Клоза,* ***C*** *– бабочки (flattened butterfly),* ***D*** *– dragonfly.*

Существует набор параметров, по которым можно сравнивать коммуникационные сети.

Длина критического пути (диаметр) – минимальное количество элементарных связей, которые нужно пройти для коммутации двух самых удаленных процессоров.

Связность – количество элементарных связей, которые нужно удалить, чтобы схема распалась на две несвязные части.

Сложность – общее количество необходимых элементарных связей.

Давайте посмотрим на все эти параметры для приведенных выше топологий (Рис.6.8).

Схема коммутации	Длина критического пути	Связность	Сложность
линейка	p-1	1	p-1
кольцо	$\lfloor p/2 \rfloor$	2	p
звезда	2	1	p-1
двумерная решётка	$2(\sqrt{p}-1)$	2	$2(p-\sqrt{p})$
двумерный тор	$2\lfloor \sqrt{p}/2 \rfloor$	4	2p
полносвязная топология	1	p-1	p(p-1)/2
двоичный гиперкуб	$\log_2 p$	$\log_2 p$	$p\log_2 p/2$

Рис.6.8. Параметры топологий.

Можно заметить, что из всех представленных топологий сбалансированной является двоичный гиперкуб.

Кроме того, можно производить сравнение и по следующим параметрам сетей:

- Количество портов маршрутизаторов (radix): low-radix / high-radix.
- Масштабируемость.
- Расширяемость.
- Простота эксплуатации.
- Устойчивость к отказам.
- Многообразие путей.
- …

В качестве примера **компьютера с распределенной памятью** рассмотрим вариант классической линейки компьютеров фирмы $Cray$, семейства $\boldsymbol{Cray\ XT}$.

В качестве коммуникационной сети используется **трехмерный тор**.

Узлы этого компьютера делятся на три варианта:

- **Пользовательские**: компиляция, командные файлы, однопроцессорные задачи;
- **Операционной системы**: выполнение многих системных сервисных функций ОС
- **Вычислительные**: выполнение программ пользователя в монопольном режиме.

В реальных конфигурациях $\boldsymbol{Cray\ T3E}$: 24/16/576 или 7/5/260.

А **вычислительный узел** состоит из: процессора, локальной памяти + сетевой интерфейс. Любой процессор через свой сетевой интерфейс может обратиться к памяти любого другого процессора, не прерывая его работы.

Так как в коммуникационной сети используется **трехмерный тор**, то у каждого узла всегда есть шесть непосредственных соседей. Между узлами – два однонаправленных канала передачи данных, что допускает одновременный обмен данными в противоположных направлениях.

В этом компьютеры был принят следующий **принцип маршрутизации**: если должны пообщаться узлы A и B, то все измерения тора нумеровались (X, Y, Z) и

маршрутизация велась всегда в одном и том же порядке. Сначала выравнивание координат вдоль оси X, потом Y и Z (Рис.6.9).

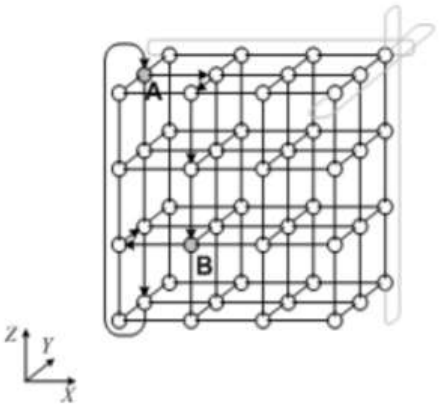


Рис.6.9. Принцип маршрутизации компьютера $Cray$.

В случае, если есть обрыв связи на маршруте, то сообщение может пойти вдоль той же оси, но в противоположном направлении.

На физическом уровне **узлы** были **расположены с чередованием** для того, чтобы уменьшить длину самой длинной связи.

Также в этом компьютере была впервые использована **аппаратная схема барьерной синхронизации. Барьер** – точка в программе, при достижении которой каждый процесс должен ждать, пока остальные процессы также не дойдут до барьера. Для реализации этой схемы используются специальные регистры. Если процессор дошел до точки синхронизации, то он выставляет в регистре «1». Дальше они переходят в схему логического «И», пока на выходе не будет две единички. Далее эта схема соединяется в двоичное дерево до тех пор, пока все процессоры дошли до барьера (Рис.6.10).

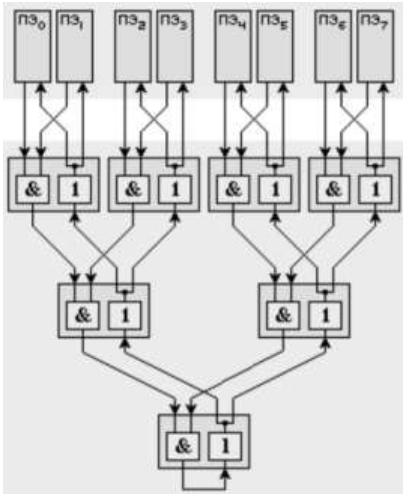


Рис.6.10. Аппаратная схема барьерной синхронизации (логика «И»).

Вычислительные кластеры

Вычислительный кластер – это совокупность компьютеров, объединенных в рамках некоторой коммуникационной сети для решения одной задачи.

Обычно кластеры строятся на основе доступных процессорных платформ, стандартных сетевых технологий и свободных ПО. Все это обеспечивает невысокую стоимость кластерных систем.

На момент июня 2020 года 474 компьютера из списка $Top500$ мира – кластеры.

Если говорить про кластеры, про компьютеры с распределенной памятью, то многое для них определяют характеристики коммуникационной сети.

Латентность – время начальной задержки при посылки сообщений.

Пропускная способность сети определяется скоростью передачи информации по каналам связи и измеряется объемом передаваемой информации в единицу времени.

Причем время на передачу сообщения вычисляется так:

$$t_N = t_0 + \frac{N}{S},$$

Где t_0 – латентность, N – объем передаваемых данных, S – пропускная способность сети.

На практике требуется померить реальные значения латентности. Можно воспользоваться следующим алгоритмом: процессор p_1 замеряет время t_1, процессор p_1 посылает сообщение длины 0 процессору p_2, который сразу же после приема посылает обратно это сообщение p_1. Теперь процессор p_1 замеряет время t_2. Тогда под латентностью понимаем величину $t_0 = (t_2 - t_1)/2$.

Пропускная способность и латентность сильно зависят от алгоритмов маршрутизации и топологии коммуникационной сети.

И в заключении приведем те факторы, которые будут снижать производительность компьютеров с распределенной памятью:

1. Закон Амдала
2. Латентность передачи по сети
3. Пропускная способность каналов передачи данных
4. Особенности использования SMP-узлов
5. Балансировка вычислительной нагрузки
6. Возможность асинхронного счета и передачи данных
7. Особенности топологии коммуникационной сети
8. Производительность отдельных процессоров
9. …

Лекция 7. Классификация параллельных вычислительных систем. Часть 3

Векторно-конвейерные компьютеры

В данной лекции будет рассмотрена тема векторизации, конвейерная обработка и векторно-конвейерные компьютеры.

На сегодняшний день векторная обработка проникает в микропроцессоры, где существуют методы векторизации.

Для начала рассмотрим отношение понятий, которые нам потребуются (Рис.7.1).

Рис.7.1. Взаимоотношение понятий.

Функциональное устройство:

- Скалярное;
- Конвейерное (операция делится на несколько микроопераций)

Команда:

- Скалярная (все аргументы – скаляры)
- Векторная (например, сложить все элементы массива x с числом b)

Компьютер:

- Скалярный;
- Векторный;
- Конвейерный.

Векторизация программы – процесс поиска подходящих фрагментов в программе и их замена векторными командами.

Пример векторизуемого фрагмента:

```
for (i=0;i<n;i++) c[i]=a[i]+b[i];
```

Компилятор сгенерирует последовательность векторных команд:

- Загрузка векторов a и b из памяти в векторные регистры,
- Векторная операция сложения,
- Запись содержимого векторного регистра в память.

Если весь рассматриваемый фрагмент программы может быть заменен на последовательность векторных операций, то говорят о **полной векторизации.**

Истоки векторности: линейная организация памяти и использование массивов.

Для векторизации какого-либо объекта необходимо:

- Наличие векторов-аргументов

- Над всеми элементами векторов должны выполняться одинаковые, независимые операции, для которых существуют аналогичные векторные команды в системе команд компьютеры.

Вектор – упорядоченный набор однотипных данных, все элементы которого размещены в памяти компьютера с одинаковым смещением друг относительно друга.

Простейшие примеры векторов:

- Одномерные массивы;
- Строки и столбцы матриц;
- Диагональные квадратные матрицы;
- Многомерные массивы целиком.

А пример **не вектора** – поддиагональная часть двумерной матрицы.

В векторизуемом фрагменте необязательно все данные должны быть массивами или векторами. Там могут встречаться и простые переменные.

```
for (i=0;i<n;i++) b[i]=a[i]+s;
```

Кандидаты для векторизации обычно – **самые внутренние циклы**.

Однако **не все фрагменты могут быть векторизованы**, так как они могут содержать **информационные зависимости**:

```
for (i=0;i<n;i++) a[i]=a[i-1]+b[i];
```

Если фрагмент программы следующий,

```
for (i=0;i<n;i++) a[i]=f(a[i],b[i]);
```

когда явно не выписана операция, а идет **вызов функции**, то обычный компилятор **векторизовать его не будет**. Необходим межпроцедурный анализ данной функции для поиска информационных зависимостей.

Далее рассмотрим пример классического векторно-конвейерного компьютера ***CRAY C90*** (1991 г.) (Рис.7.2).

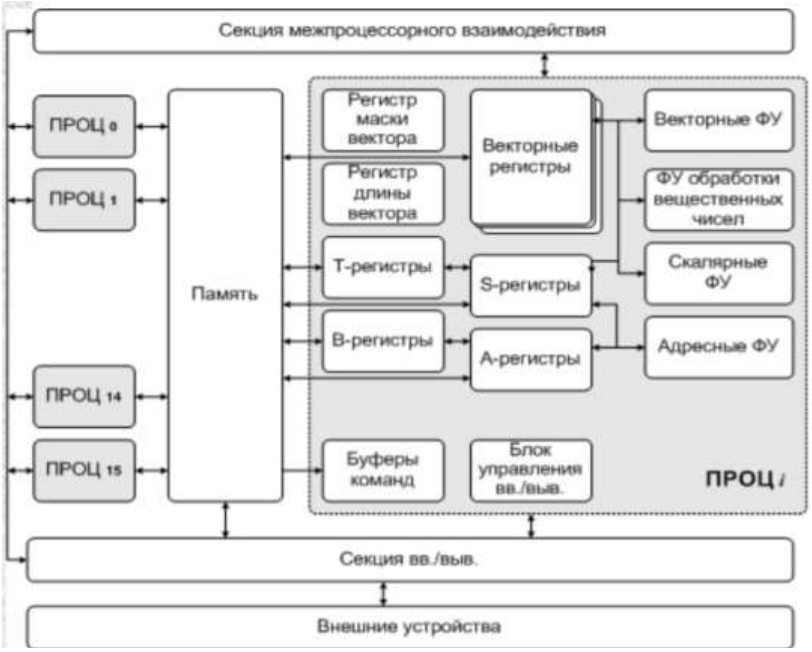


Рис.7.2. Общая схема суперкомпьютера CRAY C90.

Этот компьютер представляет собой до 16 векторно-конвейерных процессоров на общей памяти. Для взаимодействия процессоров используется секция межпроцессорного взаимодействия. Также имеется секция общего ввода-вывода для общения с внешними устройствами.

На правой половине Рис.7.2 видна структура одного из 16 процессоров компьютера (надпись ПРОЦ$_i$). Главное в ней это функциональные устройства (ФУ) для обработки данных: векторные ФУ, ФУ для обработки вещественных чисел, скалярные ФУ и адресные ФУ. Все ФУ реализованы в виде конвейеров и работают исключительно с регистрами.

В данном компьютере 8 векторных регистров по 128 элементов. Если в векторных операциях участвуют не все элементы, то используются регистры маски векторов. Эти регистры заполняется «0» или «1» в зависимости от участия элемента в операции.

Существуют также буферные T- и B-регистры, которые играют роль современной кэш-памяти для предвыборки элементов из памяти.

Теперь посмотрим влияние некоторых факторов на производительность этого компьютера.

Напомним понятие **времени разгона конвейера** – это время, которое необходимо для заполнения конвейера.

На маленьких векторах производительность низкая. А с ростом вектора растет и производительность (Рис.7.3). Для примера снова обратимся к циклу

```
for (i=0;i<n;i++) b[i]=a[i]*s+c[i];
```

Длина вектора	Производительность (Mflops)
1	7.0
4	27.6
32	181.9
128	433.7
129	364.3
256	548.0
257	491.0
8192	802.0

Рис.7.3. Зависимость производительности от длины вектора.

И второй фактор заключается в том, что когда мы проходим границу, кратную 128 элементам, существует некая деградация производительности так как следующий элемент необходимо подкачивать из памяти.

Также нужно отметить, что все конвейеры, которые делают векторную обработку в компьютере $CRAY\ C90$, физически продублированы. Отсюда возникает особенность **дублирования**, которая автоматически увеличивает пиковую производительность в 2 раза.

Еще одно свойство, которое присуще векторно-конвейерным компьютерам, – **зацепление векторных устройств**. То есть такой режим, в котором выход одного ФУ сразу подается на вход другого ФУ. При этом мы не только экономим время записи результата в память, но самое главное происходит экономия времени из-за образования одного длинного конвейера для операций. Продемонстрируем это на примере.

Пусть длина конвейера умножения l_1 ступень, а сложения l_2 ступеней, тогда в режиме **без зацепления**: $(l_1 + n - 1) + (l_2 + n - 1) = l_1 + l_2 + 2 * n - 2$ тактов, а **с зацеплением**: $l_1 + l_2 + n - 1$ тактов.

Далее перейдем к фактору **конфликтов в памяти**. Обычно память современных компьютеров разделяется на блоки: секции, банки и т.д. Разбиение памяти нужно для ускорение доступа к данным. И в компьютере $CRAY\ C90$ вся оперативная память (ОП) разбивается на 8 секций, далее каждая из секций разбивается на 8 подсекций и каждая на 16 банков. То есть суммарно 1024 банка на всю ОП.

В случае обращения к одной и той же секции возникает конфликт, который разрешается за 1 такт. Если происходит одновременное обращение к одной и той же подсекции одной секции, то время на разрешение конфликта достигает 6 тактов.

Если мы обращаемся к данным последовательно, то конфликтов не возникает вообще, но стоит пойти по массиву, например, с шагом кратным 2 или 64, то будет получен конфликт и необходимы 6 тактов для его разрешения.

Экспериментально это было подтверждено следующим фрагментом (Рис.7.4):

```
for (i=0;i<n*k;i+=k) a[i]=b[i]+c[i]*d;
```

Шаг по памяти	Производительность (Mflops)
1	705.2
2	444.6
4	274.6
8	142.8
16	84.5
32	44.3
64	22.7
128	22.6

Рис.7.4. Зависимость производительности от шага по памяти при n=1000.

Конфликты в памяти могут возникать и в достаточно сложных фрагментах:

```
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)
```

Мы идем по циклам с шагом «1». Если бы это был одномерный массив, конфликтов бы не было. Но здесь они возникают. В фортране массивы хранятся «по столбцам»:

```
X(N1,N2,N3)
X(i,j,k)≈X(i+1,j,k)     расстояние 1,
X(i,j,k)≈X(i,j+1,k)     расстояние N1,
X(i,j,k)≈X(i,j,k+1)     расстояние N1*N2.
```

Если, к примеру, массив описан не очень удачно:

```
X(40,40,N), то расстояние 40*40=1600=64*25.
```

Мы получили самый плохой случай, значит будет самая низкая производительность.

Конечно, для решения проблемы можно переописать массив как $X(41,41,N)$, тогда **шаг нечетный – конфликтов нет**.

Еще один фактор, влияющий на производительность – **ограниченное количество каналов между памятью и процессором.**

Конкретно в этом компьютере есть три канала передачи данных: два на чтение из памяти, а третий на запись.

Это приводит к тому, что если в операции больше, чем два векторных аргумента, производительность начинает проседать (в виду наличия только трех каналов) (Рис.7.5).

Длина вектора	Производительность (Mflops)
10	57.0
100	278.3
1000	435.3
12801	445.0

Рис.7.5. Зависимость производительности от длины вектора при операциях с более двумя векторными аргументами.

Более фундаментальное свойство, которое характерно многим типам архитектур – **раскрутка циклов**.

На примере рассмотрим два вложенных цикла:

```
for(j=1;j<=120;j++)
   for(i=1;i<=n;i++)
      d[i]=d[i]+s*p[i][j-1]+t*p[i][j];
```

```
120*3=360 операций чтения
120 операций записи
```

Стоит заметить, что в этом фрагменте мы используем вектор $p[i][j-1]$, который был использован нами же на предыдущей итерации по j. Поэтому можно сэкономить на этом путем расписывания внешнего цикла с каким-то шагом (в данном случае 2):

```
for(j=1;j<=120;j+=2)
   for(i=1;i<=n;i++)
      d[i]=d[i]+s*p[i][j-1]+t*p[i][j]+s*p[i][j]+t*p[i][j+1];

60*4=240 операций чтения
60 операций записи
```

Этот шаг еще называют **глубиной раскрутки**.

На реальных результатах производительность ведет себя так (Рис.7.6):

Глубина раскрутки	Производительность (Mflops)
1	612.9
2	731.6
3	780.7
4	807.7

Рис.7.6. Зависимость производительности от глубины раскрутки.

Еще один фактор компьютера $CRAY\ C90$ – **несбалансированное использование устройств, отсутствие операций деления.** То есть деление реализуется через две операции, поэтому программы с делением теряют в производительности (Рис.7.7).

Длина вектора	Производительность (Mflops)			
	$a_i=b_i+c_i$	$a_i=b_i/c_i$	$a_i=s/b_i+t$	$a_i=s/b_i\times t$
10	35.5	24.8	49.7	46.1
100	202.9	88.4	197.4	166.5
1000	343.8	117.2	283.8	215.9

Рис.7.7. Производительность операций с делением в зависимости от длины вектора.

Таким образом, большое количество факторов влияет на скорость выполнения программ. Еще один пример приведен ниже. Нам нужно реализовать следующую операцию:

$$A_{ijk} = A_{i-1jk} + B_{jk} + B_{jk}, \qquad i = 1{,}40;\ j = 1{,}40;\ k = 1{,}1000$$

```
Do k=1,1000
```

```
   Do j=1,40
      Do i=1,40
         A(i,j,k)=A(i-1,j,k)+B(j,k)+B(j,k)
```

Решение в лоб с производительностью всего 20 $Mflop/s$.

Теперь после обдумывания и учета некоторых факторов (раскрутка, конфликты в памяти и т.д.), напишем более приличный код:

```
Do i=1,40,2
   Do j=1,40
      Do k=1,1000
         A(i,j,k)=A(i-1,j,k)+2*B(j,k)
         A(i+1,j,k)=A(i,j,k)+2*B(j,k)
```

И производительность возрастет до 700 $Mflop/s$.

То есть для программистов такой тип архитектуры является очень удобным из-за того, что при правильном программировании можно добиваться высоких значений реальной производительности. Однако векторно-конвейерные компьютеры на сегодняшний день являются очень дорогими и они уже уступают по производительности параллельным архитектурам компьютеров.

Но возвращение к векторности происходит по причине эволюции микропроцессоров и появления в них векторных регистров.

Здесь можно отметить векторный процессор ***NEC SX-Aurora TSUBASA*** 2017 года. И его некоторые характеристики:

- $Vector\ Engine$ (VE) в виде карты $PCIe$ включает в себя 8 ядер, состоящих из скалярного и векторного блоков.
- В каждом ядре содержится по 64 векторных регистра длиной по 256 64-разрядных элементов. Суммарная ёмкость регистров составляет 128 Кбайт.
- 3 устройства FMA $(Fused\ Multiply\ Add)$ выполняют операцию $a * b + c$ над векторными регистрами – каждый такт 32 результата. Суммарно 32x3x2 = 192 $Flop$/такт. При тактовой частоте 1.16 ГГц получаем 307.2 $GFlop/s$ на ядро или 2.4576 $TFlop/s$ на процессор.

В отличие от ускорителей, программа целиком выполняется на VE. $Vector\ Host$ (VH) используется для компиляции приложений и реализации функций ОС, таких как выделение ресурсов, взаимодействие с файловой системой и т.д.

В рамках одного узла разные VE могут связываться друг с другом через $PCIe$. Большие параллельные системы, созданные с помощью $SX\text{-}Aurora$, используют в качестве коммуникационной сети $Infiniband$.

Операционная система VE называется $VEOS$ и полностью выгружена в хост-систему, векторный хост (VH).

Теперь перейдем к суммированию основных особенностей архитектуры, дающих заметный вклад в **ускорение выполнения программ**:

1. Конвейеризация выполнения команд.

2. Независимость функциональных устройств.
3. Векторная обработка.
4. Дублирование конвейеров векторных устройств и устройств для вещественной арифметики.
5. Зацепление функциональных устройств.
6. Многопроцессорная обработка.

А к **снижению производительности** векторно-конвейерных компьютеров приводят следующие факторы:

1. Закон Амдала
2. Время разгона конвейера
3. Секционирование векторных команд
4. Конфликты в памяти
5. Каналы процессор-память
6. Операции чтения/записи в векторные регистры
7. Ограниченное число векторных регистров
8. Несбалансированное использование устройств
9. Отсутствие операции деления
10. Перезагрузка буферов команд
11. …

Распределенные вычислительные среды

Далее будем говорить о таких компьютерах, которые строятся для решения определенных задач из частей, которые физически распределены по земному шару и соединены посредством сетей Интернет (Рис.7.8).

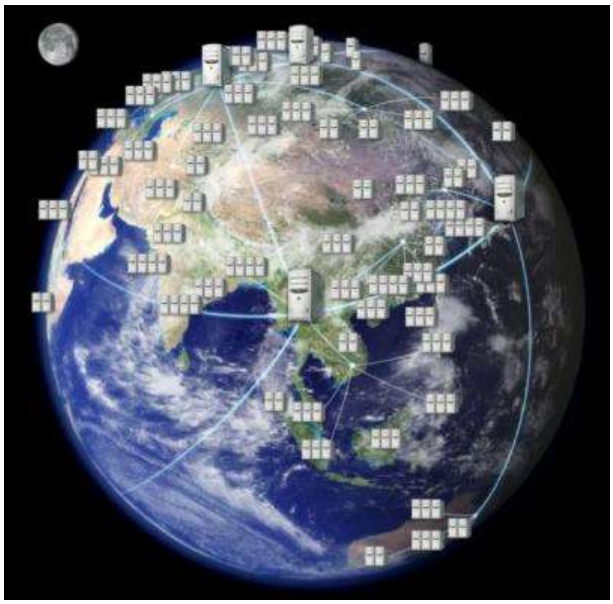


Рис.7.8. Схема распределенных вычислительных сред.

Для **эффективного использования** таких **распределенных сред** нужно учитывать следующие факторы:

- Колоссальные ресурсы (решение многих задач)
- Распределенность и удаленность (зависимость скорости передачи данных от качества каналов связи)
- Динамичность конфигурации (корректная обработка ситуаций при отключении от сети)
- Неоднородность конфигурации (в сети входят различные компьютеры)
- Различная административная принадлежность (разные организации, компании, страны)

В качества примера приведем распределенную метакомпьютерную среду $X\text{-}COM$, которая разработана в НИВЦ МГУ. Главная ее особенность была в том, что она легка в установке и использовании.

В 2005-2006 годах эта среда использовалась для проектирования новых лекарств. В нее входили: кластеры МГУ + кластер ЮУрГУ (г.Челябинск) + учебный класс НИВЦ МГУ + компьютеры в лабораториях.

Пиковая производительность > 1 ТФлопс. И за 11-12 дней удалось решить поставленную задачу.

При программировании распределенных вычислительных сред нужно учитывать их особенности на всех этапах: класса и свойств задач, структуры процесса вычислений, программирования вычислительных сред и выполнения распределенных программ.

Далее затронем некоторые понятия, которые являются смежными к нашей теме.

Проведение вычислений по требованию (*on-demand computing*). Ситуация, когда организация нуждается в расчетах, но собственного суперкомпьютера у нее нет. Тогда эта компания покупает процессорное время у других компаний.

Облачные вычисления (*cloud computing*) – это технология распределенной обработки данных, в которой компьютерные ресурсы и мощности предоставляются пользователю как Интернет-сервис. Облачная обработка данных – парадигма, в рамках которой информация постоянно хранится на серверах в Интернет и временно кэшируется на клиентской стороне, например, на персональных компьютерах и т.д.

Многопоточность (мультитредовость, *multithreading*) – свойство платформы или приложения, состоящее в том, что процесс, порожденный в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

Гипертрейдинг (гиперпоточность, *hyper-threading*) – технология, разработанная компанией *Intel* для процессоров на микроархитектуре *NetBurst*. Реализует идею «одновременной мультипоточности». После включения гипертрейдинга один физический процессор (одно физическое ядро) определяется операционной системой как два отдельных процессора (два логических ядра). При определенных

рабочих нагрузках позволяет увеличить производительность процессора. Суть технологии: передача полезной работы бездействующим исполнительным устройствам.

Параллелизм на уровне машинных команд

Параллелизм на уровне машинных команд – способность процессора исполнять несколько независимых машинных команд одновременно в рамках одного программного потока (треда).

Выгоды параллелизма на уровне машинных команд: отсутствие у пользователя необходимости в специальном параллельном программировании; переносимость.

В этом направлении выделяются два понятия:

Суперскалярная архитектура – архитектура, использующая несколько дешифраторов команд, которые могут нагружать работой множество исполнительных блоков.

А задача обнаружения параллелизма в машинном коде возлагается на аппаратуру, она же и строит соответствующую последовательность исполнения команд. В этом смысле код для суперскалярных процессоров не отражает точно не природу аппаратного обеспечения, на котором он будет реализован, ни точного временного порядка, в котором будут выполняться команды.

***VLIW* (*Very long instruction word*, «очень длинное командное слово»)** – архитектура процессоров с несколькими вычислительными устройствами. Характеризуется тем, что одна инструкция процессора содержит несколько операций, которые должны выполняться параллельно.

В процессорах $VLIW$ задача распределения решается во время компиляции и в инструкциях явно указанно, какое вычислительное устройство должно выполнять эту команду. **Компилятор сам выявляет параллелизм в программе** и явно сообщает аппаратуре, какие операции не зависят друг от друга. Код для $VLIW$-процессоров содержит точный план того, как процессор будет выполнять программу: когда будет выполнена каждая операция, какие функциональные устройства будут работать и т.д.

***EPIC* (*Explicity Parallel Instruction Computing*)** – вычисления с явным (заданным) параллелизмом на уровне команд. В этой технологии компилятор явным образом говорит процессору, какие команды можно исполнить параллельно, а какие зависят от других команд.

В архитектура $EPIC$ используется концепция длинного командного слова и добавляются следующие черты:

- **Спекулятивная загрузка данных** – вынесение команд загрузки данных далеко вперед инструкций, использующих эти данные. Это позволяет лучше использовать иерархию памяти.
- **Предикатное выполнение команд** – операции выполняются условно, в зависимости от параметра, имеющего булево значение – предиката, связанного с базовым блоком, содержащим операцию. Это позволяет избежать излишних инструкций ветвления и уменьшает нагрузку на устройство предсказания переходов.

Лекция 8. Оценка производительности. Технологии параллельного программирования

Методы оценки производительности суперкомпьютеров

В первой половине этой лекции речь пойдет об оценке производительности компьютеров. Вообще говоря, понятие производительности довольно-таки лукавое. Лукавство покажем на примере, взятом из классической статьи **«Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers».**

Пусть нужно провести операцию сложения

$$A = B + C$$

Причем производительность такой операции на суперкомпьютере будет равна X. Но нам необходимо больше, чем X. Тогда используем один из простых способов добавления операции умножения:

$$A = B + C * 1$$

Тогда производительность сразу же увеличивается в 2 раза и будет равна $2X$! А вот время выполнения никак не сократилось, возможно даже увеличилось.

Когда хотят выбрать компьютер для своих задач пользователи в идеале нуждаются в однозначном сопоставлении компьютеру некоторого числа, чтобы можно узнать какой компьютер лучше.

Таким естественным числом является **пиковая производительность компьютера**: вычисляется просто и однозначно, но нет связи с реальной задачей пользователя.

Поэтому полезнее для пользователя оценка эффективности программно-аппаратной среды на некоторых задачах или наборе задач.

Могу быть сконструированы **синтетические (или искусственные) тесты**, которые не имеют отношения к реальным приложениям, а предназначены для создания стрессовой нагрузки на отдельные подсистемы компьютера.

Основные требования к тестам производительности:

- Непротиворечивость и понятность результатов
- Легкость в использовании
- Масштабируемость
- Переносимость
- Репрезентативность
- Доступность теста и его исходного кода
- Воспроизводимость

Наиболее **известные тесты** (бенчмарки):

- *Linpack*
- *STREAM*
- Ливерморские циклы
- *Perfect Club Benchmarks*

- $SPEC$
- $HINT$
- $NAS\ Parallel\ Benchmarks\ (NPB)$
- $HPC\ Challenge$
- $Graph500$
- $HPCG$

Разберем подробнее наиболее популярный тест $\boldsymbol{Linpack}$. Он был создан Джеком Донгаррой ($Jack\ Dongarra$) и его коллегами в 1979 году.

- Используется для формирования списков $Top500$ и $Top50$.
- Решение больших систем линейных алгебраических уравнений с плотной квадратной матрицей метод LU-разложения.
- Число операций с плавающей точкой оценивается по формуле $\frac{2n^3}{3} + 2n^2$, где n – линейный размер матрицы.
- Изначально в тесте использовались маленькие матрицы 100x100, также был фиксированный текст теста.
- Потом матрицы увеличили до 1000x1000 и разрешили вносить изменения в тест, но стандартная головная часть должна была быть фиксированной.
- Сейчас – произвольный (максимально возможный) размер матрицы, возможность вносить любые изменения в текст.

Теперь рассмотрим тест $\boldsymbol{High\ Performance\ Linpack}$ ($\boldsymbol{HPL}$).

- http://www.netlib.org/benchmark/hpl/
- Наиболее популярная реализация теста $Linpack$ на языке Си.
- Обмены между процессорами выполняются через процедуру MPI.
- Вычисления на каждом процессоре основывается на низкоуровневой библиотеке базовых функций линейной алгебры $BLAS\ (Basic\ Linear\ Algebra\ Subprograms)$ или $VSIPL\ (Vector\ Signal\ Image\ Processing\ Library)$.
- Варианты $BLAS$:
  - ✓ $ACML\ (AMD\ Core\ Math\ Library)$ для процессоров $AMD\ Athlon$ и $Opteron$.
  - ✓ $Intel\ MKL\ (Intel\ Math\ Kernel\ Library)$ для процессоров $Intel$.
  - ✓ $cuBLAS$ в составе $NVIDIA\ CUDA\ SDK$ для видеокарт и ускорителей.
  - ✓ $ESSL\ (Engineering\ and\ Scientific\ Subroutine\ Library)$ для процессоров $PowerPC$.
  - ✓ $ATLAS\ (Automatically\ Tuned\ Linear\ Algebra\ Software)$, реализация интерфейса $BLAS$ с открытым исходным кодом.
  - ✓ $uBLAS$, часть библиотеки $Boost$.

В вычислительном центре МГУ создана версия мобильного $Linpack$'а, подробную информацию можно найти на сайте http://linpack.hpc.msu.ru/

Далее рассмотрим еще несколько тестов. Например, тест ***STREAM*** (***Sustainable Memory Bandwidth in High Performance Computers***) появился достаточно давно, он тестирует скорость работы некоторых арифметических операций и скорость работы с памятью компьютера. В первой версии $STREAM$ были следующие операции:

- ✓ $a(i) = b(i)$;
- ✓ $a(i) = q * b(i)$;
- ✓ $a(i) = b(i) + c(i)$;
- ✓ $a(i) = b(i) + q * c(i)$;
- ✓ Замер скорости передачи данных.

В версии $STREAM2$ были добавлены еще некоторые операции:

- ✓ $a(i) = q$;
- ✓ $a(i) = b(i)$;
- ✓ $a(i) = a(i) + q * b(i)$;
- ✓ $sum = sum + a(i)$;

Причем размер массивов задается достаточным, чтобы выйти за размеры кэш-памяти. Отношение пиковой производительности к скорости передачи данных почти всегда больше 1. Чем отношение больше, тем больше несбалансированность компьютера.

Параллельные версии этого теста сделаны с использованием технологий $OpenMP$ и MPI.

Но данный тест слишком простой и вызывает недоверие из-за отдаленности от реальных задач.

Поэтому появляются более сложные тесты, на основе которых создают наборы тестов. Один из наиболее известных таких наборов является ***NAS Parallel Benchmarks***.

- http://www.nas.nasa.gov/Software/NPB/
- Набор тестов производительности, разработанных в $NASA\ Advanced\ Supercomputing\ (NAS)\ Division$ (ранее $NASA\ Numerical\ Aerodynamic\ Simulation\ Program$).
- Существует последовательная реализация, параллельные реализации с использованием $MPI, OpenMP, MPI + OpenMP$, вариант на $JAVA$, версия для $Grid$ на базе $Globus$.
- Последняя на данный момент версия NPB 3.4
- Размер задачи – классы:
  - ✓ S, W (для тестовых прогнозов);
  - ✓ A, B, C (каждый в среднем в 4 раза больше предыдущего)
  - ✓ D, E, F (каждый в среднем в 16 раз больше предыдущего)
- 5 вычислительных ядер:
  - ✓ $IS\ (Integer\ Sort)$;

- ✓ EP $(Embarrassingly\ Parallel)$;
- ✓ CG $(Conjugate\ Gradient)$;
- ✓ MG $(MultiGrid)$;
- ✓ FT $(Fast\ Fourier\ Transform)$.

- 3 модельных приложения:
  - ✓ BT $(Block\ Tri\text{-}diagonal\ solver)$;
  - ✓ SP $(Scalar\ Penta\text{-}diagonal\ spolver)$;
  - ✓ LU $(Lower\text{-}Upper\ Gauss\text{-}Seidel\ solver)$.
- 3 дополнительных теста:
  - ✓ UA $(Unstructured\ Adaptive\ mesh)$;
  - ✓ DC $(Data\ Cube)$;
  - ✓ DT $(Data\ Traffic)$.

Набор таких тестов позволяет в некоторых случаях достаточно неплохо приблизиться к реальным задачам для оценки производительности компьютера. Поэтому тесты $NASA$ достаточно популярны в данный момент.

Другой набор тестов, который появился не так давно, называется ***HPC Challenge***. Включает в себя 7 тестов:

- HPL $(Linpack)$;
- $DGEMM$ (вычисляет производительность перемножения матриц);
- $STREAM$;
- $PTRANS$ (транспонирование матрицы);
- $RandomAccess$ (вычисляет скорость случайных обращений к памяти);
- $FFTE$ (реализация одномерного дискретного преобразования Фурье);
- $Communication\ bandwidth\ and\ latency$ (скорость передачи данных и латентность).

Пик популярности этого набора тестов прошел, так как, возможно, людям необходимо иметь какую-то одну точную оценку производительности системы, чем много оценок, составленных с разных сторон.

Еще один недавний тест – ***Graph500***.

- http://www.graph500.org/
- Анонсирован в 2010 году, составляется свой список наиболее производительных компьютеров.
- На данный момент последняя версия теста 3.0.0.
- Включает последовательный вариант, вариант на $OpenMP, MPI$ и вариант для $Cray\ XMT$.
- Реализует поиск в ширину (BFS) и поиск кратчайшего пути от одной вершины $(SSSP)$ в большом ненаправленном графе.
- Используется для измерения пропускной способности сетевой системы суперкомпьютеров.

- Производительность алгоритмов измеряется количеством пройденных дуг графа в секунду ($Traversed\ Edges\ Per\ Second, TEPS$). Используются обозначения ME/s и GE/s – миллионы и миллиарды пройденных дуг в секунду соответственно.

Самый новый из приведенных здесь тестов – $\boldsymbol{HPCG}$. Он был анонсирован в 2013 году и имеет на данный момент версию 3.1. Ниже приведены основные его характеристики:

- http://hpcg-benchmark.org/
- $High\ Performance\ Conjugate\ Gradient$ – решение системы линейных алгебраических уравнений с разреженной квадратной положительно определенной симметричной матрицей.
- Оценивает не только скорость самих вычислений, но и нерегулярных обращений в память.
- Разработчики: $Jack\ Dongarra, Michael\ Heroux, Piotr\ Luszczek$.

На данный момент тест $HPCG$ лучше характеризует производительность компьютеров с точки зрения близости к реальным задачам.

Для того чтобы оценить производительность суперкомпьютера нужно понимать, что производительность вносит свой вклад в самые разные уровни среды:

- Базовый уровень ПО (ОС, компилятор, системы программирования);
- Базовый уровень аппаратуры (элементарные операции, иерархия памяти);
- Уровень операций ввода/вывода;
- Базовый коммуникационный уровень;
- Коммуникационный уровень приложений;
- Уровень модельных приложений;
- Уровень реальных приложений.

Технологии параллельного программирования

Сначала обсудим вопрос, который связан с понятием моделей параллельного программирования. Ниже перечислим некоторые их виды.

- $\boldsymbol{SPMD}$ ($\boldsymbol{Single\ Program, Multiple\ Data}$, единая программа, множество данных) – на всех процессорах выполняется одна и та же программа над своим множеством входных данных.
- $\boldsymbol{Master\text{-}Workers}$ или $\boldsymbol{Master\text{-}Slave}$ (мастер-рабочие) – один из процессов (называемый главным, $master$) получает входные данные задачи, разделяет их на части и передает другим процессам (рабочим, $workers$) для обработки, а затем получает результаты и проводит финальную обработку.
- $\boldsymbol{Message\ passing}$ (модель передачи сообщений). Программа порождает несколько задач с уникальными идентификаторами. Взаимодействие осуществляется посредством отправки и приема сообщений. Новые задачи могут создаваться во время выполнения параллельной программы, несколько задач могут выполняться на одном процессоре.

- ***Data parallel*** (модель параллелизма данных). Одна операция применяется к множеству элементов структуры данных. Программа содержит последовательность таких операций. Распределяемыми данными обычно являются массивы. Как правило, языки программирования, поддерживающие данную модель, допускают операции над массивами, позволяют использовать в выражениях целые массивы, вырезки из массивов. Каждый процессор отвечает за то подмножество элементов массива, которое расположено в его локальной памяти.
- ***Shared memory*** (модель общей памяти). Задачи обращаются к общей памяти, имея общее адресное пространство. Требуется разграничение доступа к общим данным.

Технологий параллельного программирования много. И **выбор конкретной технологии** можно основывать **на следующих критериях**:

- Возможность создания эффективных программ.
- Возможность быстрого создания параллельных программ (продуктивность программирования).
- Переносимость программ, гарантии сохранения эффективности.
- Стоимость.

При программировании систем с общей и распределенной памятью нужно учитывать разные факторы.

Компьютеры с общей памятью:

- Параллелизм вычислений;
- Синхронизация доступа к общим данным.

Компьютеры с распределенной памятью:

- Параллелизм вычислений;
- Распределение данных;
- Согласование параллелизма вычислений и распределения данных;
- Организация пересылок данных.

Теперь перейдем к **иерархии различных технологий параллельного программирования.** От самых первых идей, базовых вариантов, до современных, которые используются сегодня.

0. Распараллеливающий компилятор.

Идея состоит в том, что компилятор достаточно интеллектуальный для того, чтобы мог сам извлекать параллелизм программ и отображать на архитектуру целевого параллельного компьютера.

Но возникали проблемы с тем, что в сложных случаях компилятор не справлялся с распараллеливанием. И необходимо было усложнить структуру самого компилятора, что не хорошо.

1. Спецкомментарии

Это комментарии специального типа, которые помогают компилятору.

- В чистом виде – компиляторы $CRAY$ ($CDIR\$\ NODEPCHK$).
- ***OpenMP*** – стандарт для программирования на масштабируемых SMP-системах в модели общей памяти. Кроме спецкомментариев есть и языковые конструкции (вызов функций).
- ***OpenACC*** – стандарт, описывающий набор директив для написания гетерогенных программ, задействующих как центральный, так и графический процессоры. Последняя версия стандарта - $OpenACC$ 2.7 https://www.openacc.org

2. Расширения существующих языков программирования.

HPF (***High Performance Fortran***): ориентирован на создание переносимых программ. Отображение массив – массив-шаблон – виртуальный процессорный массив – физические процессоры. Есть и спецкомментарии, и языковые конструкции, например, оператор $FORALL$ для параллельных циклов. Необходимые коммуникации и синхронизации реализуются компилятором. **Сложность конструкций *HPF*** оказалась **непреодолимым препятствием** для создания по-настоящему эффективных компиляторов. Есть система $HPF\ Adaptor$, переводящая программу с HPF на $Fortran$ с MPI.

mpC **(ИСП РАН):** язык программирования, основанный на языке C, предоставляющий средства создания параллельных программ для компьютеров с распределенной памятью. **Ориентирован на неоднородные системы.** Пользователь может задать топологию сети, распределение данных и вычислений и необходимые пересылки данных. Посылка сообщений организована с использованием интерфейса MPI. Информация, извлеченная из описания параллельного алгоритма, вместе с данными о реальной производительности процессоров и коммуникационных каналов, помогают системе программирования mpC найти эффективный способ отображения процессоров mpC-программы на компьютеры сети.

Языки *Fortran-DVM* и *C-DVM* (ИПМ РАН).

DVM-система состоит из **5 основных компонентов**: компиляторы, система поддержки выполнения параллельных программ, отладчик параллельных программ (сравнение промежуточных результатов), анализатор производительности, предсказатель производительности (предиктор).

- Высокоуровневая модель программирования;
- Спецкомментарии, один вариант программы для параллельного и последовательного выполнения;
- Основная работа по распределению вычислений и данных выполняется динамически системой поддержки;
- DVM-процессор переводит программу в Cu (Фортран), расширенный функциями системы поддержки;
- Для организации межпроцессорного взаимодействия может использоваться одна из коммуникационных технологий ($MPI, PVM, Router$), что обеспечивает хорошую переносимость;

- Директивы DVM: распределение данных, распределение вычислений, спецификация удаленных данных, редукционные операции.

***CAF* (*Co-Array Fortran*):**

- Небольшое расширение языка $Fortran$, достаточное для разработки эффективных параллельных программ;
- Модель $SPMD$;
- Понятие $co\text{-}array$ (распределенного массива), компоненты которого распределены по процессам; доступ к распределенным компонентам осуществляется по правилам работы с обычными массивами.

***UPC* (*Unified Parallel C*):**

- Расширение языка программирования Cu;
- Модель $SPMD$;
- Модель $PGAS$ ($Partitioned\ Global\ Address\ Space$, глобальное разделенное адресное пространство) – адресуемая глобальная память в виде логических разделов, причем каждый из разделов локален для каждого из процессоров;
- Синхронизация и обеспечение консистентности памяти.
- https://upc-lang.org/

CUDA (https://developer.nvidia.com) и ***OpenCL*** (https://opencl.org/) – расширение Cu-подобных языков для программирования графических ускорителей.

Linda – параллельный язык программирования. Программа рассматривается как совокупность процессов, которые могут обмениваться данными через пространство кортежей. Используется совместно с другими языками высокого уровня как средство общения параллельных процессов.

3. Специальные языки программирования.

Occam – язык параллельного программирования, ориентированный в первую очередь на написание программ для транспьютерных систем. Позволяет описать любую ЯПФ с помощью инструкций par, seq, отмечающих участки параллельных и последовательных процессов.

НОРМА (ИПМ РАН):

- Декларативный язык, предназначенный для описания решения вычислительных задач сеточными методами;
- Описание задач в нотации, близкой к исходной постановке проблемы математиком;
- Не содержит традиционные конструкции языков программирования, фиксирующие порядок вычисления и/или иным образом «скрывающие/ограничивающие» параллелизм;
- Язык с однократным присваиванием, нет конструкций вида $X = X + 1$;
- Распараллеливанием занимается компилятор, результат получается на Фортран + MPI, Фортран + PVM и др.

Colamo (НИИ МВС ЮФУ) – язык структурно-процедурного программирования высокого уровня реконфигурируемых вычислительных систем (на основе $FPGA$).

Также есть и другие развивающиеся языки программирования: ***Chapel***, ***X10***, ***Fortress*** …

4. Библиотеки и интерфейсы, поддерживающие взаимодействие параллельных процессов.

MPI (***Message Passing Interface***) – хорошо стандартизированный механизм для построения программ по модели обмена сообщениями. Существуют стандартные привязки MPI к языкам C и $Fortran$. Существуют бесплатные и коммерческие реализации почти для всех суперкомпьютерных платформ, а также для сетей рабочих станций $UNIX$ и $Windows$. В настоящее время MPI – наиболее широко используемый и динамично развивающийся интерфейс из всего класса.

PVM (***Parallel Virtual Machine***) – общедоступная библиотека, предоставляющая возможности управления процессами с помощью механизма передачи сообщений (предшественник MPI).

Shmem реализует схему работы над общей памятью с помощью операций Put/Get. $Shmem$ включена в стандарт MPI 2.0 в качестве раздела «односторонние коммуникации».

5. Средства и технологии для поддержки метакомпьютерных и распределенных вычислений.

Globus, gLite, UNICORE, Condor, BOINC, X-Com, Map/Reduce …

6. Параллельные предметные библиотеки.

PBLAS – параллельные версии базовых процедур линейной алгебры ($BLAS$), уровней 1,2,3. Библиотека разработана в рамках проекта $ScaLAPACK$.

ScaLAPACK включает подмножество процедур $LAPACK$, переработанных для использования на параллельных компьютерах, включая: решение систем линейных уравнений, обращение матриц, ортогональные преобразования, поиск собственных значений и др.

ATLAS – оптимизированная библиотека, включающая реализацию процедур $BLAS$ и часть функциональности $LAPACK$.

MKL – оптимизированная реализация $BLAS$ от $Intel$.

FFTW, DFFTPack – быстрое преобразование Фурье.

PETSc – набор процедур и структур данных для параллельного решения научных задач с моделями, описываемыми в виде дифференциальных уравнений с частными производными.

7. Специализированные пакеты и программные комплексы.

С помощью специально разработанных пакетов и комплексов (***FlowVision, ANSYS, GAMESS, GAUSSIAN, CHARMM, GROMACS, OpenFOAM***) пользователь не занимается процессом написания программы, а только подбирает начальные данные.

Лекция 9. Технология программирования *OpenMP*

Параллельные и последовательные области

OpenMP – технология параллельного программирования для компьютеров с **общей памятью**. Стандарт 5.0 принят в ноябре 2018 года.

Данная технология предполагает один вариант программы для параллельного и последовательного выполнения.

Любой процесс состоит из нескольких **нитей управления** (основного элемента управления *OpenMP*), которые имеют общее адресное пространство, но разные потоки команд и раздельные стеки. То есть идеология заключается в том, что нити рождаются по необходимости, и когда эта необходимость пропадает, нити уничтожаются.

Если мы хотим понять поддерживает ли наш компилятор технологию *OpenMP*, то нужно использовать специальный **макрос *_OPENMP,*** который определяется в формате ***yyyymm,*** где ***yyyy*** и ***mm*** – цифры года и месяца, когда был принят поддерживаемый стандарт *OpenMP*.

Ниже приведем пример на определенность данного макроса (без конкретной версии):

```
#include <stdio.h>
int main(){
#ifdef _OPENMP
 printf("OpenMP is supported!\n");
#endif
}
```

Основа распараллеливания этой технологии – **вставка** в текст программы **специальных директив**, а также **вызов** вспомогательных **функций**.

OpenMP подразумевает только ***SPMD*-модель** (*Single Program Multiple Data*) параллельного программирования: для всех параллельных нитей используется один и тот же код.

Программа всегда начинается с **последовательной области** – работает одна нить, при входе в **параллельную область** порождается еще несколько нитей, между которыми распределяются части кода.

И по завершении параллельной области все нити, кроме одной (нити-мастера), завершаются.

Таких параллельных и последовательных областей может быть любое количество в программе, в том числе они могут быть вложенными друг в друга.

В *OpenMP* есть **два основных класса переменных**:

- ***shared*** (*общие*; все нити видят одну и ту же переменную);
- ***private*** (*локальные*, приватные; каждая нить видит свой экземпляр данной переменной).

По умолчанию, все переменные, порожденные вне параллельной области, при входе в нее остаются общими. Исключение составляют переменные, являющиеся

счетчиками итераций в цикле. Переменные, порожденные внутри параллельной области, по умолчанию являются локальными.

Основная функциональность $OpenMP$ задается с помощью **директив**. Для языка $Cи$ это указания препроцессору, начинающимися с

```
#pragma omp

#pragma omp directive-name
[опция[[,] опция]...]
```

Объектом действия большинства директив является один оператор или блок, перед которым расположена директива в исходном тексте программы.

Чтобы задействовать **функции библиотеки *OpenMP***, то в программу необходимо включить заголовочный **файл *omp.h*** (для программ на языке Фортран – файл $omp\_lib.h$ или модуль omp_lib).

Можно задать количество нитей, которые будут выполнять параллельные области программы, определив значение переменной среды $OMP\_NUM\_THREADS$:

```
export OMP_NUM_THREADS=n
```

Опишем также несколько **вспомогательных функций**. Во-первых, существует функция для работы с системным таймером:

```
double omp_get_wtime(void);
```

Данная функция возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом. Гарантируется, что за время работы программы этот момент в прошлом не изменится. Есть смысл только в вычислении разницы времен вызовов этой функции. В реальных системах у таймеров достаточная точность.

Во-вторых, следующая функция возвращает разрешение таймера в секундах, то есть время между двумя «тиками» этого таймера:

```
double omp_get_wtick(void);
```

Первой будем рассматривать **директиву *parallel***, которая задает параллельную область:

```
#pragma omp parallel [опция[[,]опция]...]
```

Когда эта директива встречается в программе, порождаются новые ***OMP_NUM_THREADS* − 1 нитей**, каждая получает свой уникальный номер, причем порождающая нить получает номер 0 и становится нитью группы («мастером»). Все порожденные нити начинают выполнять один и тот же код, который идет непосредственно после этой директивы. При выходе из параллельной области производится неявная синхронизация, то есть все нити дожидаются, пока все остальные порожденные нити не выполнят заданный участок. И после этого все нити, кроме нити-мастера, уничтожаются.

Теперь познакомимся с **опциями** данной **директивы**.

```
if(условие) - выполнение параллельной области по условию;
```

Определяет выполнение параллельной области только в случае, если **условие истинно**.

Данная опция может быть полезна, когда не обязательно производить распараллеливание в случае малого количества операций.

num_threads (целочисленное выражение) – явное задание количества нитей, которые будут выполнять параллельную область;
default(shared|none) – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс shared; none – всем переменным класс назначается явно;
private(список) – переменные, для которых порождается локальная копия в каждой нити; **начальное значение не определено**;
firstprivate(список) – переменные, для которых порождается локальная копия в каждой нити; локальные копии инициализируются значениями этих переменных в нити-мастере;
shared(список) – переменные, общие для всех нитей;

Еще одной опцией является ***reduction*** **(оператор : список)**, которая задает оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; они инициализируются соответственно типу оператора (для **аддитивных – 0** или аналоги, для **мультипликативных – 1** или аналоги); после выполнения всех операторов параллельной области выполняется заданный оператор;

оператор это: для языка Cu $-+$, $*$, $-$, $\&$, $|$, $\hat{}$, $\&\&$, $||$; **порядок выполнения** операторов **не определен**, поэтому результат может отличаться от запуска к запуску.

Далее приведен простой пример программы на $OpenMP$, где показано, что до параллельной области $printf$ выполняется одной нитью-мастера, а в параллельной области $printf$ будет сделан столько раз, сколько будет порождено нитей.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
 printf("Последовательная область 1\n");
#pragma omp parallel
 {
 printf("Параллельная область\n");
 }
 printf("Последовательная область 2\n");
}
```

Теперь более сложный пример. Здесь присутствует параллельная область с **опцией *reduction***. У каждой нити возникает своя локальная копия переменной $count$. Происходит инкрементация и распечатывание результатов. В момент, когда все операторы параллельной области выполнены, выполняется операция $reduction$ (суммирование количества единиц, то есть количества нитей).

```
#include <stdio.h>
int main(int argc, char *argv[])
```

```
{
 int count = 0;
#pragma omp parallel reduction (+: count)
 {
 count++;
 printf("Текущее значение count: %d\n",
count);
 }
 printf("Число нитей: %d\n", count);
}
```

Если **внутри параллельной области** содержится **только один параллельный цикл**, одна конструкция ***sections*** или одна конструкция ***workshare***, то можно использовать укороченную запись:

```
parallel for, parallel sections или parallel workshare.
```

OpenMP позволяет задать число нитей в паралелльных областях программы из самой программы с помощью специальной функции:

```
void omp_set_num_threads(int num);
```

Параллельные области могут быть **вложенными** друг в друга. По умолчанию вложенная параллельная область выполняется одной нитью. Это управляется установкой переменной среды ***OMP_NESTED***.

```
export OMP_NESTED=true
```

Изменить значение переменной ***OMP_NESTED*** можно с помощью вызова функции ***omp_set_nested***().

```
void omp_set_nested(int nested);
```

Узнать значение переменной ***OMP_NESTED*** можно при помощи функции ***omp_get_nested***().

```
int omp_get_nested(void);
```

Еще одна полезная функция – ***omp_in_parallel***(), которая возвращает 1, если она была вызвана из активной параллельной области программы.

```
int omp_in_parallel(void);
```

И небольшой пример, где показана функция ***mode***(), которая печатает заданные значения в зависимости от области ее вызова.

```
void mode(void){
 if(omp_in_parallel()) printf("Параллельная область\n");
 else printf("Последовательная область\n");
}
int main(int argc, char *argv[]){
 mode();
#pragma omp parallel
 mode();
}
```

Теперь переходим к **директивам**, которые задают **распределение операций** программ.

Первая директива – ***single***, которая позволяет выделить участок кода, который должен быть выполнен только 1 раз.

```
#pragma omp single [опция [[,]опция]...]
```

И возможные опции:

```
private(список);
firstprivate(список);
copyprivate(список) - новые значения переменных списка будут доступны всем одноименным частным переменным (private и firstprivate); опция не может использоваться совместно с опцией nowait; переменные списка не должны быть перечислены в опциях private и firstprivate данной директивы single;
```

Еще одна опция – ***nowait.*** После выполнения участка после директивы происходит неявная барьерная синхронизация нитей. Если не нужно ждать синхронизации, то эта опция позволяет продолжить работу дальше дошедшим до конца нитям.

```
#pragma omp parallel
 {
 printf("Сообщение 1\n");
#pragma omp single nowait
 {
 printf("Одна нить\n");
 }
 printf("Сообщение 2\n");
 }
```

Еще одна директива, которая позволяет выделить код, который будет выполнен только нитью-мастером, является **директива *master***. Остальные нити пропускают данный участок.

```
#pragma omp master
#pragma omp parallel private(n)
 {
 n=1;
#pragma omp master
 n=2;
 printf("значение n: %d\n", n);
 }
```

В $OpenMP$ есть возможность распределять участки кода по номеру нити. Все нити нумеруются от 0 до $N-1$, где N – количество нитей. Для получения номера нити используют функцию ***omp_get_thread_num***().

```
int omp_get_thread_num(void);
```

Однако это далеко не главный способ распараллеливания.

Также есть функция, которая помогает определить количество нитей в текущей параллельной области ***omp_get_num_threads***().

```
int omp_get_num_threads(void);
```

Распараллеливание циклов в *OpenMP*

Основной ресурс параллелизма заключается **в циклах**. Если в параллельной области встретится оператор цикла, то по умолчанию он будет выполняться всеми нитями. Но нам нужно распределить операции цикла различным нитям. Для этого предусмотрена директива ***for***, которую нужно подставить перед блоком с оператором ***for***.

```
#pragma omp for [опция [[,]опция]...]
```

Для нее используются следующие опции:

```
private(список);
firstprivate(список);
lastprivate(список) - переменным, перечисленным в списке, присваивается результат с последнего витка цикла;
reduction(оператор:список);
schedule(type[, chunk]) - опция задает, каким образом итерации цикла распределяются между нитями;
collapse(n) - опция указывает, что n последовательных тесновложенных циклов ассоциируется с данной директивой для циклов образуется общее пространство итераций, которое делится между нитями.
ordered - в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле.
```

На вид параллельных циклов накладываются достаточно жесткие ограничения. Запрещены побочные выходы из параллельного цикла. Если упростить, то **формат цикла** должен выглядеть так:

```
for([целочисленный тип] i = инвариант цикла;
i {<,>,=,<=,>=} инвариант цикла;
i {+,-}= инвариант цикла)
```

Приведем пример параллельного цикла.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int A[10], B[10], C[10], i, n;
for (i=0; i<10; i++){A[i]=i; B[i]=2*i; C[i]=0;}
#pragma omp parallel shared(A,B,C) private(i,n)
{
```

```
 n=omp_get_thread_num();
#pragma omp for
 for (i=0; i<10; i++){
 C[i]=A[i]+B[i];
 printf("Нить %d сложила элементы %d\n",n,i);
 }
}
}
```

В операции ***schedule*** параметр ***type*** задает следующий тип распределения итераций:

static – блочно-циклическое распределение итераций цикла; размер блока – ***chunk***. Первый блок из *chunk* итераций выполняет нулевая нить, второй блок – следующая и т.д.

dynamic – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает *chunk* итераций (по умолчанию $chunk = 1$), та нить, которая заканчивает выполнение своей порций итераций, получает первую свободную порцию из *chunk* итераций.

guided – динамическое распределение итераций, при котором размер порций уменьшается с некоторого начального значения до величины *chunk* пропорционально количеству еще не распределенных итераций, деленному на количество нитей, выполняющих цикл.

auto – способ реализации итераций выбирается компилятором и/или системой выполнения. Параметр *chunk* при этом не задается.

runtime – способ распределения итераций выбирается во время работы программы по значению переменной ***OMP_SCHEDULE***. Параметр *chunk* при этом не задается.

Изменить значения переменной ***OMP_SCHEDULE*** из программы можно с помощью вызова функции ***omp_set_schedule***().

```
void omp_set_schedule(omp_sched_t type, int chunk);
```

Допустимые значения констант описаны в файле *omp.h*. Как минимум, следующие варианты:

```
typedef enum omp_sched_t {
omp_sched_static = 1,
omp_sched_dynamic = 2,
omp_sched_guided = 3,
omp_sched_auto = 4
} omp_sched_t;
```

При распределении итераций цикла нужно убедиться, что итерации данного цикла не имеют информационных зависимостей.

Директива ***sections*** определяет набор независимых секций кода, каждая из которых выполняется своей нитью.

```
#pragma omp sections [опция [[,]опция]...]
private(список);
firstprivate(список);
lastprivate(список) – переменным присваивается результат из последней секции;
reduction(оператор:список);
nowait.
```

Директива *section* задает участок кода внутри секций ***sections*** для выполнения одной нитью. Перед первым участком кода в блоке ***sections*** директива ***section*** необязательна.

Директива *task* применяется для выделения отдельной независимой задачи.

```
#pragma omp task [опция [[,]опция]...]
```

Текущая нить выделяет в качестве задачи ассоциированный с директивой блок операторов. Задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям.

***if* (условие)** – порождение новой задачи только при выполнении некоторого условия; если условие не выполняется, то задача будет выполнена текущей нитью и немедленно.

untied — опция означает, что в случае откладывания задача может быть продолжена любой нитью из числа выполняющих данную параллельную область; если данная опция не указана, то задача может быть продолжена только породившей её нитью.

Для гарантированного завершения в точке вызова всех запущенных задач используется директива ***taskwait***.

```
#pragma omp taskwait
```

Нить, выполнившая данную директиву, приостанавливается до тех пор, пока не будут завершены все ранее запущенные данной нитью независимые задачи.

Способы синхронизаций в $OpenMP$

Самый распространенный способ синхронизации в $OpenMP$ – **барьер**. Он оформляется с помощью **директивы *barrier***.

```
#pragma omp barrier
```

Нити, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все нити не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше.

Директива ***ordered*** определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле.

```
#pragma omp ordered
```

Относится к самому внутреннему из объемлющих циклов, а в параллельном цикле должна быть задана опция ***ordered***. Нить, выполняющая первую итерацию цикла, выполняет операции данного блока. Нить, выполняющая любую следующую

итерацию, должна сначала дождаться выполнения всех операций блока всеми нитями, выполняющими предыдущие итерации.

С помощью директивы ***critical*** оформляется критическая секция программы.

```
#pragma omp critical [(<имя_критической_секции>)]
```

В каждый момент времени в критической секции может находиться не более одной нити. Все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение. Как только работавшая нить выйдет из критической секции, одна из заблокированных нитей войдет в неё.

Все **неименованные критические секции** условно ассоциируются с одним и тем же именем. Имеющие одно и тоже имя рассматриваются единой секцией, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

```
int n;
#pragma omp parallel
 {
#pragma omp critical
 {
 n=omp_get_thread_num();
 printf("Нить %d\n", n);
 }
 }
}
```

Еще одна **директива *atomic***

```
#pragma omp atomic
```

Данная директива относится к идущему непосредственно за ней оператору присваивания (на используемые в котором конструкции накладываются достаточно понятные ограничения), гарантируя корректную работу с общей переменной, стоящей в его левой части. Атомарной является только работа с переменной в левой части оператора присваивания, при этом вычисления в правой части не обязаны быть атомарными.

Еще один из **вариантов синхронизации** в *OpenMP* реализуется через **механизм замков (*locks*)**. В качестве замков используются общие переменные. Данные переменные должны использоваться только как параметры примитивов синхронизации. Замок может находиться в одном из трёх состояний: *неинициализированный*, *разблокированный* или *заблокированный*. Разблокированный замок может быть захвачен некоторой нитью. При этом он переходит в заблокированное состояние. Нить, захватившая замок, и только она может его освободить, после чего замок возвращается в разблокированное состояние.

Существует **два типа замков**: **простые замки** и **множественные замки**. Множественный замок может многократно захватываться одной нитью перед его

освобождением, в то время как простой замок может быть захвачен только однажды. Для множественного замка вводится понятие **коэффициента захваченности (*nesting count*).** Изначально он устанавливается в ноль, при каждом следующем захватывании увеличивается на единицу, а при каждом освобождении уменьшается на единицу. Множественный замок считается разблокированным, если его коэффициент захваченности равен нулю.

Для инициализации простого или множественного замка используются соответственно функции

```
omp_init_lock() и
omp_init_nest_lock().
void omp_init_lock(omp_lock_t*lock);
void omp_init_nest_lock(omp_nest_lock_t*lock);
```

После выполнения функции замок переводится в разблокированное состояние.

Функции ***omp_destroy_lock***() и ***omp_destroy_nest_lock***() используются для переведения простого или множественного замка в неинициализированное состояние.

```
void omp_destroy_lock(omp_lock_t*lock);
void omp_destroy_nest_lock (omp_nest_lock_t *lock);
```

Для захватывания замка используются функции ***omp_set_lock***() и ***omp_set_nest_lock***().

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock (omp_nest_lock_t *lock);
```

Вызвавшая эту функцию нить дожидается освобождения замка, а затем захватывает его. Замок при этом переводится в заблокированное состояние. Если множественный замок уже захвачен данной нитью, то нить не блокируется, а коэффициент захваченности увеличивается на единицу.

Для освобождения замка используются функции ***omp_unset_lock***() и ***omp_unset_nest_lock***().

```
void omp_unset_lock(omp_lock_t*lock);
void omp_unset_nest_lock(omp_lock_t*lock);
```

Вызов этой функции освобождает простой замок, если он был захвачен вызвавшей нитью. Для множественного замка уменьшает на единицу коэффициент захваченности. Если коэффициент станет равен 0, замок освобождается. Если после освобождения есть нити, заблокированные на операции, захватывающей данный замок, он будет сразу захвачен одной из ожидающих нитей.

Пример использования замка приведен ниже:

```
omp_lock_t lock;
 int n;
 omp_init_lock(&lock);
#pragma omp parallel private (n)
 {
 n=omp_get_thread_num();
```

```
 omp_set_lock(&lock);
 printf("Начало закрытой секции, %d\n", n);
 sleep(5);
 printf("Конец закрытой секции, %d\n", n);
 omp_unset_lock(&lock);
 }
 omp_destroy_lock(&lock);
```

Какая-то нить захватывает замок и выполняет программу дальше, а другие ждут. После выполнения первой нити секции, она освобождает замок и туда заходит следующая нить и т.д.

Для неблокирующей попытки захвата замка используются функции ***omp_test_lock***() и ***omp_test_nest_lock***().

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_lock_t*lock);
```

Данная функция пробует захватить указанный замок. Если это удалось, то для простого замка функция возвращает 1, а для множественного замка – новый коэффициент захваченности. Если замок захватить не удалось, в обоих случаях возвращается 0.

Данная логика реализована в следующем примере:

```
omp_lock_t lock;
 int n;
 omp_init_lock(&lock);
#pragma omp parallel private (n)
 {
 n=omp_get_thread_num();
 while (!omp_test_lock (&lock)){
 printf("Секция закрыта, %d\n", n);
 sleep(2);
 }
 printf("Начало закрытой секции, %d\n", n);
 sleep(5);
 printf("Конец закрытой секции, %d\n", n);
 omp_unset_lock(&lock);
 }
 omp_destroy_lock(&lock);
```

Опять здесь инициализация простого замка. В цикле каждая нить пытается захватить замок. Если это не удается, то она остается внутри цикла ***while***. А та нить, которая захватывает замок, она идет работать дальше и после выполнения секции освобождает замок. То есть пример похож на предыдущий, однако нити здесь не простаивают, а выполняют некоторую полезную работу.

Лекция 10. Технология программирования MPI. Часть 1

Стандарт MPI. Общие процедуры MPI

Технология MPI является технологией №1 в мире параллельного программирования. Предназначена она в первую очередь для компьютеров с **распределенной памятью**, но может использоваться и на других архитектурах.

$\boldsymbol{MPI}$ – $\boldsymbol{Message\ Passing\ Interface}$, интерфейс передачи сообщений.

Стандарт MPI 3.1 (4 июня 2015 года). В нем более 450 различных процедур и функций. Основная модель программирования – $\boldsymbol{SPMD}$ ($Single\ Program\ Multiple\ Data$) – **основная модель** параллельного программирования.

Официальные стандарты утверждены для двух языков: C и $Fortran$.

Префикс $\boldsymbol{MPI\_}$.

```
#include "mpi.h"
```

($mpif.h$ для языка Фортран)

Основное, что есть в MPI – **процессы** (посылка сообщений).

Сообщение – массив однотипных данных, расположенных в последовательных ячейках памяти.

Процессы также объединяются в **группы** – упорядоченные множества процессов.

От понятия группы нужно отличать понятие **коммуникатора** – контекста обмена группы. Это глобальный объект. Он объединяет различные процессы. Поэтому все операции с коммуникатором достаточно долгие, в отличие от операций с группами. В операциях обмена используются только коммуникаторы!

Коммуникаторы имеют в языке Cu предопределённый тип $\boldsymbol{MPI\_Comm}$ (в языке Фортран – тип $\boldsymbol{INTEGER}$).

При запуске любого приложения существуют три предопределенный коммуникатора:

$\boldsymbol{MPI\_COMM\_WORLD}$ – коммуникатор для всех процессов приложения.

$\boldsymbol{MPI\_COMM\_SELF}$ – коммуникатор, включающий только текущий процесс.

$\boldsymbol{MPI\_COMM\_NULL}$ – коммуникатор, не содержащий ни одного процесса.

Каждый процесс может одновременно входить в разные коммуникаторы. Для идентификации процесса нужно указать *коммуникатор (группа)* и *номер процесса в коммуникаторе (группе).*

Если коммуникатор содержит n процессов, то номера процессов в нём лежат в пределах от 0 до $n-1$.

Вместе с сообщением по сети также посылается некий набор **атрибутов**: *номер процессаотправителя, номер процесса-получателя, идентификатор сообщения, коммуниктор*.

Идентификатор сообщения - целое неотрицательное число в диапазоне от 0 до $\boldsymbol{MPI\_TAG\_UB}$ (не меньше 32767). Идентификатор важен только программисту, системе главное, чтобы он совпал при пересылке.

Для работы с атрибутами сообщений введена структура $\boldsymbol{MPI\_Status}$.

Возвращаемым значением (в Фортране – последним аргументом) для большинства процедур *MPI* является информация об успешности завершения. В случае успешного выполнения процедура вернёт значение ***MPI_SUCCESS***, иначе - *код ошибки*.

Предопределённые значения, соответствующие различным ошибочным ситуациям, перечислены в файле *mpi.h*.

Далее перейдем к **функциям *MPI*.**

Для использования функций необходимо вызвать **инициализацию**

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

Инициализация параллельной части программы. Почти все другие процедуры *MPI* могут быть вызваны только после вызова ***MPI_Init***. Инициализация параллельной части для каждого приложения должна выполняться только один раз.

А завершат параллельную часть приложения функция

```
int MPI_Finalize(void)
```

Все последующие обращения к большинству процедур *MPI*, в том числе к ***MPI_Init***, запрещены. К моменту вызова ***MPI_Finalize*** каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Простейшая параллельная программа имеет такой вид:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
 printf("Before MPI_INIT\n");
 MPI_Init(&argc, &argv);
 printf("Parallel section\n");
 MPI_Finalize();
 printf("After MPI_FINALIZE\n");
}
```

Первый *printf* будет выполнен всеми процессами, потом начинается параллельная часть – все процессы напечатают "*Parallel section*". И последний *printf* тоже напечатают все процессы приложения.

Также есть два исключения, **две функции**, которые можно использовать до вызова ***MPI_Init*** и после вызова ***MPI_Finalize*.**

```
int MPI_Initialized(int *flag)
```

В аргументе *flag* возвращает 1, если вызвана после процедуры ***MPI_Init***, и 0 в противном случае.

```
int MPI_Finalized(int *flag)
```

В аргументе *flag* возвращает 1, если вызвана после процедуры ***MPI_Finalize*** , и 0 в противном случае.

Еще **две вспомогательные функции**, которые есть в практически любой программе:

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

В аргументе *size* возвращает число параллельных процессов в коммуникаторе *comm*.

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

В аргументе *rank* возвращает номер процесса в коммуникаторе *comm* в диапазоне от 0 до $size - 1$.

Ниже приведем небольшой пример

```
#include <stdio.h>
#include "mpi.h"
#define MAX 100
int main(int argc, char **argv)
{
 int rank, size, n, i, ibeg, iend;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 n=(MAX-1)/size+1;
 ibeg=rank*n+1; iend=(rank+1)*n;
 for(i=ibeg; i<=((iend>MAX)?MAX:iend); i++)
      printf ("process %d, %d^2=%d\n", rank, i, i*i);
 MPI_Finalize();
}
```

Здесь происходит разделение диапазона вычислений на части, когда каждый процесс будет выполнять свою часть (от *ibeg* до *iend*).

Еще две функции, которые **возвращают секунды, а не информацию об успешности процесса:**

```
double MPI_Wtime(void)
```

Возвращает для каждого вызвавшего процесса астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Момент времени, используемый в качестве точки отсчёта, не будет изменён за время существования процесса.

```
double MPI_Wtick(void)
```

Возвращает разрешение таймера в секундах.

```
int MPI_Get_processor_name(char *name, int *len)
```

Возвращает в строке *name* имя узла, на котором запущен вызвавший процесс. В переменной *len* возвращается количество символов в имени, не превышающее константы ***MPI_MAX_PROCESSOR_NAME***.

Рассмотрим пример определения характеристик системного таймера:

```
#include <stdio.h>
#include "mpi.h"
#define NTIMES 100
int main(int argc, char **argv)
```

```
{
 double time_start, time_finish, tick;
 int rank, i;
 int len;
 char *name;
 name =
(char*)malloc(MPI_MAX_PROCESSOR_NAME*sizeof(char));
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Get_processor_name(name, &len);
 tick = MPI_Wtick();
 time_start = MPI_Wtime();
 for (i = 0; i<NTIMES; i++)
     time_finish = MPI_Wtime();
 printf ("node %s, process %d: tick= %lf, time= %lf\n",
name, rank, tick, (time_finish-time_start)/NTIMES);
 MPI_Finalize();
}
```

Передача и прием сообщений с блокировкой

Начнем с передачи и приема сообщений с блокировкой типа точка-точка. Один посылающий и один принимающий процессы.

Первая **базовая функция**:

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

Блокирующая посылка массива buf с идентификатором $msgtag$, состоящего из $count$ элементов типа $datatype$, процессу с номером $dest$ в коммуникаторе $comm$.

Функция ***MPI_Send*** является **блокирующей.** То есть текущий процесс приостанавливается до того момента, пока сообщение из буфера buf куда-то не ушло. Можно **разблокировать** эту функцию, если скопировать данные в некие буфера. И пересылка будет осуществлена позже.

Типы данных:

- ***MPI_INT*** – int
- ***MPI_SHORT*** – $short$
- ***MPI_LONG*** – $long$
- ***MPI_FLOAT*** – $float$
- ***MPI_DOUBLE*** – $double$
- ***MPI_CHAR*** – $char$
- ***MPI_BYTE*** – 8 бит
- ***MPI_PACKED*** – тип для упакованных данных.

Все типы данных перечислены в файле $mpi.h$.

Есть **три** основные **модификации** функции ***MPI_Send*:**

- ***MPI_Bsend*** – передача сообщения с буферизацией (создаются специальные буферы для хранения данных).
- ***MPI_Ssend*** – передача сообщения с синхронизацией (здесь процесс будет **разблокирован** только тогда, когда **данные** будут уже **приняты принимающим процессом**).
- ***MPI_Rsend*** – передача сообщения по готовности (почти всегда самый быстрый вариант, нет проверки на готовность принятия данных).

Функция для **создания буфера** приведена ниже

```
int MPI_Buffer_attach(void* buf, int size)
```

Назначение массива buf размера $size$ для использования при посылке сообщений с буферизацией. В каждом процессе может быть только один такой буфер. Размер массива, выделяемого для буферизации, должен превосходить общий размер сообщения как минимум на величину, определяемую константой ***MPI_BSEND_OVERHEAD***.

После того, как буфер становится не нужным, то мы освобождаем память

```
int MPI_Buffer_detach(buf, size)
```

Освобождение массива buf для других целей. Процесс блокируется до того момента, когда все сообщения уйдут из данного буфера.

Также опишем функцию **приема сообщений**

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

Блокирующий приём в буфер buf не более $count$ элементов сообщения типа $datatype$ с идентификатором $msgtag$ от процесса с номером $source$ в коммуникаторе $comm$ с заполнением структуры атрибутов приходящего сообщения $status$.

Также вместо аргументов $source$ и $msgtag$ можно использовать константы:

- ***MPI_ANY_SOURCE*** – признак того, что подходит сообщение от любого процесса;
- ***MPI_ANY_TAG*** – признак того, что подходит сообщение с любым идентификатором.

Параметры принятого сообщения всегда можно определить по соответствующим элементам структуры $status$:

- ***status.MPI_SOURCE*** – номер процесса-отправителя;
- ***status.MPI_TAG*** – идентификатор сообщения;
- ***status.MPI_ERROR*** – код ошибки.

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

Если два сообщения были одновременно отправлены разными процессами, то порядок их получения принимающим процессом заранее не определён.

Если мы хотим понять, сколько элементов данных пришло реально, то используем функцию

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

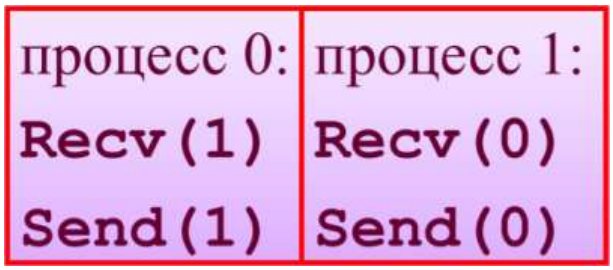
По значению параметра *status* определяет число *count* уже принятых (после обращения к ***MPI_Recv***) или принимаемых (после обращения к ***MPI_Probe*** или ***MPI_Iprobe***) элементов сообщения типа *datatype*.

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

Данная функция **не принимает сообщение**, а **заполняет структуру данных**. Получение в параметре *status* информации о структуре ожидаемого сообщения с блокировкой. Возврата не произойдёт, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения.

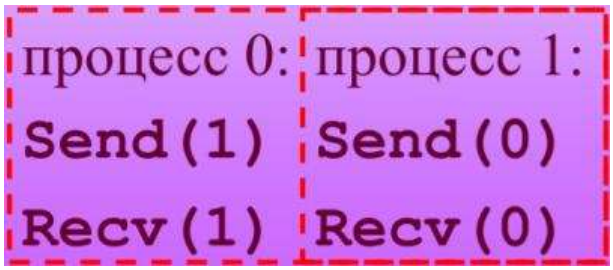
При использовании блокирующих функций возникают **тупиковые ситуации (*deadlock*):**

Первый вариант, когда два процесса должны обменяться сообщениями: процесс 0 что-то послать процессу 1 и наоборот. Тогда напишем следующее



И мы получим, что процесс 0 дойдет до ***Recv*** **(блокирующая функция)** и будет ждать. Точно также с процессором 1. Процессы будут ждать друг друга бесконечно.

Теперь **попробуем написать программу наоборот**:



Такой вариант становится даже хуже. Здесь может случиться так, что внутренних буферов хватает на посылаемые сообщения, тогда каждый процесс доходит до своего ***Send***, сообщение из буфера по ссылке будет скопировано во внутренние буфера *MPI*. Каждый процесс разблокируется и доходит до ***Recv***. То есть произойдет внутренняя коммуникация от ***Send*** к ***Recv***. Также и со вторым процессом. Программа работает, все

хорошо, **но** если запустим, например, ее 10 раз, то 9 раз она сработала, а на десятый программе не хватает внутренних буферов. То есть оба процесса могут повиснуть.

Разрешить тупиковые ситуации в простых случаях довольно легко:

1. процесс 0: процесс 1:
   Send **(1)** ***Recv*** **(0)**
   Recv **(1)** ***Send*** **(0)**
2. Использование неблокирующих операций
3. Использование функций ***MPI_Sendrecv***

Передача и прием сообщений без блокировки

Такие функции необходимы для того, чтобы избегать тупиковых ситуаций и иметь возможность совмещать пересылку со счетом, чтобы не простаивать на блокировках функций.

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

Неблокирующая посылка сообщения. То есть процесс выполняется на фоне. Возврат из функции происходит сразу после инициализации передачи. Переменная $request$ идентифицирует пересылку.

Однако при использовании неблокирующих операций существует опасность того, что мы можем испортить данные во время пересылки, то есть нужно всегда убедиться перед отправкой данных, что все операции с ними были завершены.

Рассмотрим модификации функции ***MPI_Isend***:

- ***MPI_Ibsend*** — передача сообщения с буферизацией;
- ***MPI_Issend*** — передача сообщения с синхронизацией;
- ***MPI_Irsend*** — передача сообщения по готовности.

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

Неблокирующий приём сообщения. Возврат из функции происходит сразу после инициализации передачи. Переменная $request$ идентифицирует пересылку.

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

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

Получение информации о структуре ожидаемого сообщения без блокировки. В аргументе $flag$ возвращается значение 1, если сообщение с подходящими атрибутами уже может быть принято.

Для того, чтобы дождаться, что неблокирующие операции завершились, есть группа функций

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

Ожидание завершения асинхронной операции, ассоциированной с идентификатором $request$. Для неблокирующего приёма определяется параметр $status$. $request$ устанавливается в значение ***MPI_REQUEST_NULL***.

Также с помощью следующей функции можно дождаться завершения нескольких блокирующих операций

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

Ожидание завершения $count$ асинхронных операций, ассоциированных с идентификаторами $requests$. Для неблокирующих приёмов определяются параметры в массиве $statuses$.

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

Ожидание завершения одной из $count$ асинхронных операций, ассоциированных с идентификаторами $requests$. Для неблокирующего приёма определяется параметр $status$.

Если к моменту вызова завершились несколько из ожидаемых операций, то случайным образом будет выбрана одна из них. Параметр $index$ содержит номер элемента в массиве $requests$, содержащего идентификатор завершённой операции.

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

Ожидание завершения хотя бы одной из $incount$ асинхронных операций, ассоциированных с идентификаторами $requests$.

Параметр $outcount$ содержит число завершённых операций, а первые $outcount$ элементов массива $indexes$ содержат номера элементов массива $requests$ с их идентификаторами. Первые $outcount$ элементов массива $statuses$ содержат параметры завершённых операций (для неблокирующих приёмов).

Точно такие же модификации есть для функции типа ***Test***, которая позволяет не ждать завершение соответствующих неблокирующих операций, а определять, завершились они или нет.

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

Проверка завершённости асинхронной операции, ассоциированной с идентификатором $request$. В параметре $flag$ возвращается значение 1, если операция завершена.

Отложенные запросы на взаимодействие

Такие операции отличаются от неблокирующих тем, что в этом случае когда функция встречается в программе, то **подготавливается все что нужно** к пересылке, **но она не начинается!**

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

Такие же модификации ***MPI_Bsend_init***, ***MPI_Ssend_init*** и ***MPI_Rsend_init.***

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

Формирование отложенного запроса на приём сообщения. Сама операция приёма не начинается!

```
int MPI_Start(MPI_Request *request)
```

Инициализация отложенного запроса на выполнение операции обмена, соответствующей значению параметра *request*. Операция запускается как неблокирующая.

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

Инициализация *count* отложенных запросов на выполнение операций обмена, соответствующих значениям первых *count* элементов массива *requests*. Операции запускаются как неблокирующие.

Сообщение, отправленное при помощи отложенного запроса, может быть принято любой из процедур ***MPI_Recv*** и ***MPI_Irecv***, и наоборот. По завершении отложенного запроса значение параметра ***request*** (*requests*) **сохраняется и может использоваться в дальнейшем!**

```
int MPI_Request_free(MPI_Request *request)
```

Удаляет структуры данных, связанные с параметром *request*. *request* устанавливается в значение ***MPI_REQUEST_NULL***. Если операция, связанная с этим запросом, уже выполняется, то она завершится.

Отложенные взаимодействия используются в итерационных схемах.

Теперь рассмотрим еще одну функцию, которая помогает в борьбе с *deadlock*.

```
int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype stype, int
dest, int stag, void *rbuf, int rcount, MPI_Datatype rtype, int
source, int rtag, MPI_Comm comm, MPI_Status *status)
```

Совмещённые приём и передача сообщений с блокировкой. Буферы передачи и приёма не должны пересекаться. **Тупиковой ситуации не возникает!** Сообщение, отправленное операцией ***MPI_Sendrecv***, может быть принято обычным образом, и операция ***MPI_Sendrecv*** может принять сообщение, отправленное обычной операцией.

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype
datatype, int dest, int stag, int source, int rtag, MPI_comm
comm, MPI_Status *status)
```

Совмещённые приём и передача сообщений с блокировкой через общий буфер *buf*.

Обмен по кольцевой топологии при помощи процедуры ***MPI_Sendrecv***:

```
prev = rank - 1;
next = rank + 1;
if (rank == 0) prev = size - 1;
if (rank == (size - 1)) next = 0;
MPI_Sendrecv(&sbuf[0], 1, MPI_FLOAT, prev,
```

```
tag2, &rbuf[0], 1, MPI_FLOAT, next, tag2,
MPI_COMM_WORLD, &status1);
MPI_Sendrecv(&sbuf[1], 1, MPI_FLOAT, next,
tag1, &rbuf[1], 1, MPI_FLOAT, prev, tag1,
MPI_COMM_WORLD, &status2);
```

Специальное значение ***MPI_PROC_NULL*** для несуществующего процесса. Операции с таким процессом завершаются немедленно с кодом завершения ***MPI_SUCCESS***.

Пример следующий

```
next = rank + 1;
if(rank == (size - 1)) next=MPI_PROC_NULL;
MPI_Send (&buf, 1, MPI_FLOAT, next, tag,
MPI_COMM_WORLD);
```

Лекция 11. Технология программирования *MPI*. Часть 2

Коллективные взаимодействия процессов

В операциях коллективного взаимодействия процессов **участвуют все процессы коммуникатора!** Как и для блокирующих процедур, возврат означает то, что разрешён свободный доступ к буферу приёма или посылки. Сообщения, вызванные коллективными операциями, не пересекутся с другими сообщениями.

Нельзя рассчитывать на синхронизацию процессов с помощью коллективных операций (кроме процедуры ***MPI_Barrier***).

Если какой-то процесс завершил свое участие в коллективной операции, то это не означает ни того, что данная операция завершена другими процессами коммуникатора, ни даже того, что она ими начата (если это возможно по смыслу операции).

В коллективных операциях **не используются идентификаторы сообщений (теги)**. Таким образом, коллективные операции строго упорядочены согласно их появлению в тексте программы. Несоблюдение такого порядка может привести к возникновению тупиковых ситуаций.

Теперь перейдем к конкретным операциям коллективного взаимодействия.

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

Эта функция должна быть вызвана всеми процессами коммуникатора и работа процессов блокируется до тех пор, пока все оставшиеся процессы коммуникатора $comm$ не выполнят эту процедуру. Главное, чтобы все процессы дошли до барьера.

Следующая функция осуществляет пересылку данных.

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

Рассылка $count$ элементов данных типа $datatype$ из массива buf от процесса root всем процессам данного коммуникатора $comm$, включая сам рассылающий процесс. Значения параметров $count$, $datatype$, $root$ и $comm$ должны быть одинаковыми у всех процессов.

Демонстрировать процессы коллективного взаимодействия будем с помощью схем (Рис.11.1):

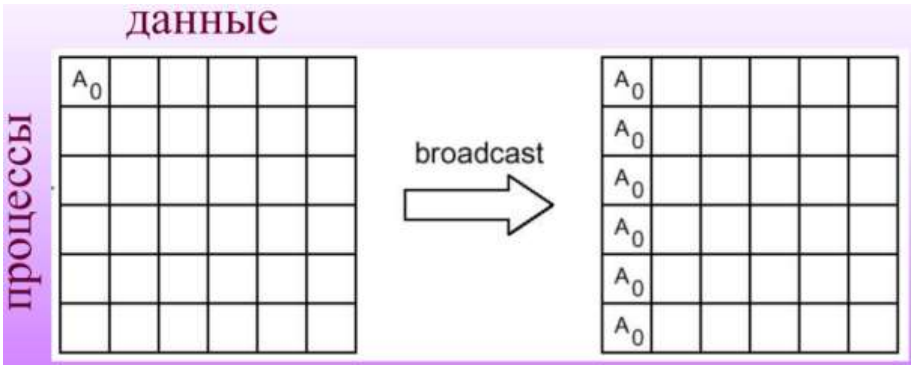


Рис.11.1. Слева блок данных до вызова процедуры $broadcast$.
Справа результат рассылки по всем процессам.

Например, для пересылки от процесса 2 всем остальным процессам приложения массива buf из 100 целочисленных элементов, нужно, чтобы во всех процессах встретился следующий вызов:

```
MPI_Bcast(buf, 100, MPI_INT, 2, MPI_COMM_WORLD);
```

Следующая функция осуществляет сбору данных со всех процессов на один.

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

Сборка $scount$ элементов данных типа $stype$ из массивов $sbuf$ со всех процессов коммуникатора $comm$ в буфер $rbuf$ процесса $root$. Данные сохраняются в порядке возрастания номеров процессов.

На процессе $root$ существенными являются значения всех параметров, а на остальных процессах - только значения параметров $sbuf$, $scount$, $stype$, $root$ и $comm$. Значения параметров $root$ и $comm$ должны быть одинаковыми у всех процессов. Параметр $rcount$ у процесса $root$ обозначает число элементов типа $rtype$, принимаемых от каждого процесса.

Если для посылки и приема данных должен использоваться один и тот же буфер, то на месте аргумента $sbuf$ процесса $root$ можно указать значение ***MPI_IN_PLACE***. В этом случае аргументы $scount$ и $stype$ игнорируются, и предполагается, что порция данных процесса $root$ уже расположена в соответствующем месте буфера приема $rbuf$.

Ниже покажем как действуют функции ***scatter*** и ***gather*** (Рис.11.2).

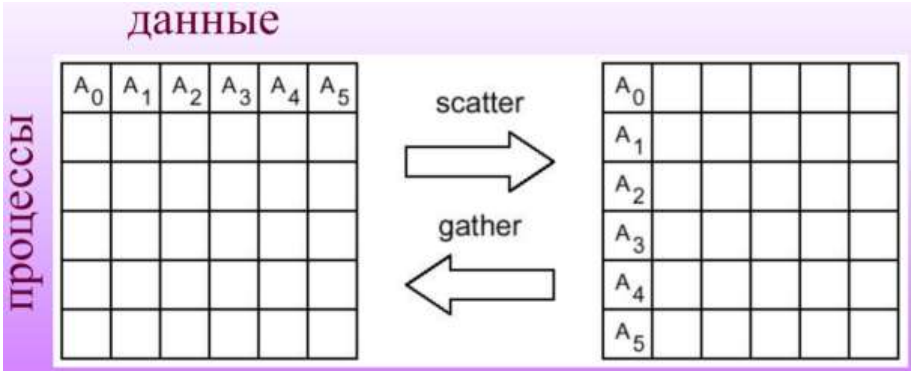


Рис.11.2. Результаты работ операций scatter и gather.

Например, чтобы процесс 2 собрал в массив $rbuf$ по 10 целочисленных элементов массивов buf со всех процессов приложения, нужно, чтобы во всех процессах встретился следующий вызов:

```
MPI_Gather(buf, 10, MPI_INT, rbuf, 10, MPI_INT, 2,
MPI_COMM_WORLD);
```

Еще одна модификация функций

```
int MPI_Gatherv(void *sbuf, int scount, MPI_Datatype stype, void
*rbuf, int *rcounts, int *displs, MPI_Datatype rtype, int root,
MPI_Comm comm)
```

Сборка различного количества данных из массивов $sbuf$. Порядок расположения данных в результирующем буфере $rbuf$ задаёт массив $displs$.

rcounts – целочисленный массив, содержащий количество элементов, передаваемых от каждого процесса (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

displs – целочисленный массив, содержащий смещения относительно начала массива *rbuf* (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

И обратная функция *scatter*, когда происходит рассылка данных. Каждому процессору **отсылается своя порция** данных.

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

Рассылка по *scount* элементов данных типа *stype* из массива *sbuf* процесса *root* в массивы *rbuf* всех процессов коммуникатора *comm*, включая сам процесс *root*. Данные рассылаются в порядке возрастания номеров процессов.

На процессе *root* существенными являются значения всех параметров, а на всех остальных процессах — только значения параметров *rbuf*, *rcount*, *rtype*, *source* и comm. Значения параметров source и comm должны быть одинаковыми у всех процессов.

Если для посылки и приема данных должен использоваться один буфер, то на месте аргумента *rbuf* процесса *root* можно указать значение ***MPI_IN_PLACE***. В этом случае аргументы *rcount* и *rtype* игнорируются, и предполагается, что порция данных процесса *root* не пересылается.

Ниже небольшой пример, где используется *scatter*

```
float sbuf[SIZE][SIZE], rbuf[SIZE];
if(rank == 0)
 for(i=0; i<SIZE; i++)
 for (j=0; j<SIZE; j++)
 sbuf[i][j]=…;
if (numtasks == SIZE)
 MPI_Scatter(sbuf, SIZE, MPI_FLOAT, rbuf,
SIZE, MPI_FLOAT, 0, MPI_COMM_WORLD);

int MPI_Scatterv(void *sbuf, int *scounts, int *displs,
MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype,
int root, MPI_Comm comm)
```

Рассылка различного количества данных из массива *sbuf*. Начало рассылаемых порций задает массив *displs*.

scounts – целочисленный массив, содержащий количество элементов, передаваемых каждому процессу (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

displs – целочисленный массив, содержащий смещения относительно начала массива *sbuf* (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

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

Сборка данных из массивов $sbuf$ со всех процессов коммуникатора $comm$ в буфере $rbuf$ каждого процесса. Данные сохраняются в порядке возрастания номеров процессов (Рис.11.3).

Если для посылки и приема данных должен использоваться один буфер, то на месте аргумента $sbuf$ всех процессов можно указать значение ***MPI_IN_PLACE***. В этом случае аргументы scount и $stype$ игнорируются, и предполагается, что порции исходных данных всех процессов уже расположены в соответствующих местах буферов приема $rbuf$.

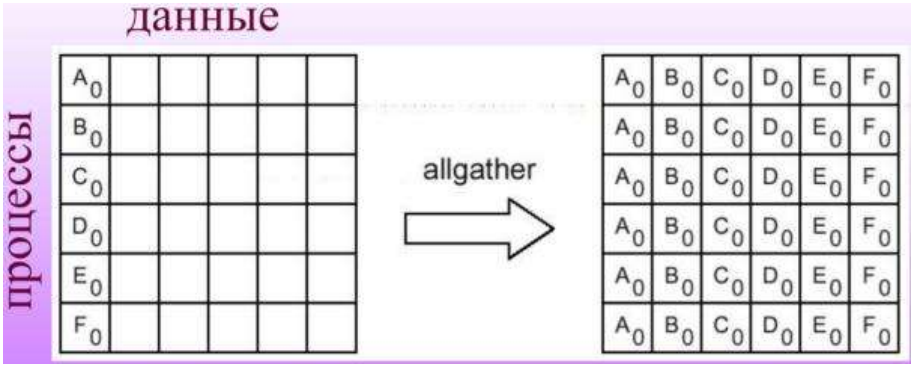


Рис.11.3. Результат работы процесса allgather.

```
int MPI_Allgatherv(void *sbuf, int scount, MPI_Datatype stype,
void *rbuf, int *rcounts, int *displs, MPI_Datatype rtype,
MPI_Comm comm)
```

Сборка на всех процессах различного количества данных из $sbuf$. Порядок расположения данных в массиве $rbuf$ задаёт массив $displs$.

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

Рассылка каждым процессом коммуникатора $comm$ различных порций данных всем другим процессам. j-й блок массива $sbuf$ процесса i попадает в i-й блок массива $rbuf$ процесса j (Рис.11.4).

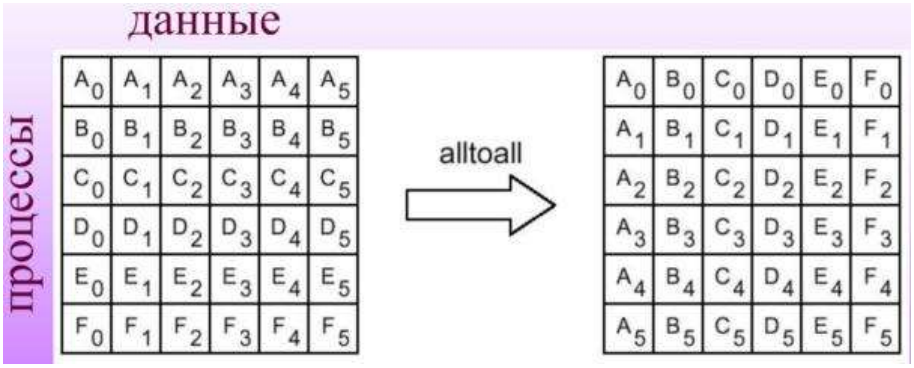


Рис.11.4. Операция alltoall.

```
int MPI_Alltoallv(void* sbuf, int *scounts, int *sdispls,
MPI_Datatype stype, void* rbuf, int *rcounts, int *rdispls,
MPI_Datatype rtype, MPI_Comm comm)
```

Рассылка со всех процессов коммуникатора $comm$ различного количества данных всем другим процессам. Размещение данных в буфере $sbuf$ отсылающего процесса определяется массивом $sdispls$, а в буфере $rbuf$ принимающего процесса – массивом $rdispls$.

```
int MPI_Alltoallw(void *sbuf, int scounts[], int sdispls[],
MPI_Datatype stypes[], void *rbuf, int rcounts[], int rdispls[],
MPI_Datatype rtypes[], MPI_Comm comm)
```

Процедура позволяет осуществить наиболее общий обмен данными с заданием для каждой порции своего размера, типа данных и размещения в буфере. Смещения в массивах $sdispls$ и $rdispls$ задаются в байтах.

Несколько функций также позволяют осуществлять операции над данными.

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

Выполнение $count$ независимых глобальных операций op над соответствующими элементами массивов $sbuf$. Результат операции над i-ми элементами массивов $sbuf$ получается в i-ом элементе массива $rbuf$ процесса $root$.

Типы предопределённых глобальных операций:

- ***MPI_MAX, MPI_MI***N – максимальное и минимальное значения;
- ***MPI_SUM, MPI_PROD*** – глобальная сумма и глобальное произведение;
- ***MPI_LAND, MPI_LOR, MPI_LXOR*** – логические “И”, “ИЛИ”, искл. “ИЛИ”;
- ***MPI_BAND, MPI_BOR, MPI_BXOR*** – побитовые “И”, “ИЛИ”, искл. “ИЛИ”.

Если для посылки и приема данных должен использоваться один буфер, то на месте аргумента $sbuf$ процесса $root$ можно указать значение ***MPI_IN_PLACE***. В этом случае входные данные процесса root берутся из буфера результата $rbuf$.

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

Выполнение $count$ независимых глобальных операций op над соответствующими элементами массивов $sbuf$. Результат получается в массиве $rbuf$ каждого процесса.

Небольшой пример ниже

```
for(i=0; i<n; i++) s[i]=0.0;
for(i=0; i<n; i++)
 for(j=0; j<m; j++)
 s[i]=s[i]+a[i][j];
MPI_Allreduce(s, r, n, MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);
```

Также есть функция, которая выполняется в оперативной памяти одного процессора и аналогичная ***MPI_Reduce.***

```
int MPI_Reduce_local(void* inbuf, void* inoutbuf, int count,
MPI_Datatype datatype, MPI_Op op)
```

Процедура на вызвавшем процессе поэлементно выполняет op операций над $count$ элементами массивов $inbuf$ и $inoutbuf$. Результат помещается в массив $inoutbuf$.

```
int MPI_Reduce_scatter_block( void* sbuf, void* rbuf, int
rcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Выполнение $n * rcount$ (n – число процессов) независимых глобальных операций op над соответствующими элементами массивов $sbuf$ всех процессов.

Сначала выполняются глобальные операции, а затем результат рассылается по процессам.

При этом i-ый процесс получает $(i+1)$-ую порцию результатов из $rcount$ элементов и помещает ее в массив $rbuf$. Значение rcount должно быть одинаковым на всех процессах коммуникатора $comm$.

```
int MPI_Reduce_scatter(void *sbuf, void *rbuf, int *rcounts,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Выполнение $\sum_i \text{rcounts(i)}$ независимых глобальных операций op над соответствующими элементами массивов $sbuf$.

Сначала выполняются глобальные операции, затем результат рассылается по процессам.

i-ый процесс получает $rcounts(i)$ значений результата и помещает в массив $rbuf$.

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

Выполнение $count$ независимых частичных глобальных операций op над соответствующими элементами массивов $sbuf$.

i-ый процесс выполняет глобальную операцию над соответствующими элементами массива $sbuf$ процессов $0 \ldots i$ и помещает результат в массив $rbuf$.

Окончательный результат глобальной операции получается в массиве $rbuf$ последнего процесса.

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

Выполнение $count$ независимых частичных глобальных операций op над соответствующими элементами массивов $sbuf$.

i-ый процесс $(i > 0)$ выполняет $count$ глобальных операций над соответствующими элементами массива $sbuf$ процессов с номерами от 0 до $i-1$ включительно и помещает полученный результат в массив $rbuf$. На процессе 0 массив $rbuf$ не задействуется.

Также есть возможность создавать пользовательские операции.

```
int MPI_Op_create (MPI_User_function *func, int commute, MPI_Op
*op)
```

Создание пользовательской глобальной операции op, которая будет вычисляться функцией $func$. Если $commute = 1$, то операция должна быть коммутативной и ассоциативной. Иначе порядок фиксируется по увеличению номеров процессов.

Теперь приведем интерфейс функции:

```
typedef void MPI_User_function (void *invec, void *inoutvec, int
*len, MPI_Datatype type)
```

Первый аргумент $invec$, второй аргумент – $inoutvec$, результат –$inoutvec$. len задает количество элементов входного и выходного массивов, а $type$ – тип данных. В пользовательской функции не должны производиться никакие обмены.

```
int MPI_Op_free(MPI_Op *op)
```

Уничтожение пользовательской глобальной операции. После выполнения процедуры переменной op присваивается значение $\boldsymbol{MPI\_OP\_NULL}$.

Приведем пример с пользовательской функцией, которая выполняет суммирование двух элементов векторов по модулю 5.

```
#include <stdio.h>
#include "mpi.h"
#define n 1000
void smod5(void *in, void *inout, int *l, MPI_Datatype
*type){
 int i;
 for(i=0; i<*l; i++) ((int*)inout)[i] = (((int*)in)[i]
+ ((int*)inout)[i])%5;
}
int main(int argc, char **argv)
{
 int rank, size, i;
 int a[n];
 int b[n];
 MPI_Op op;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 for(i=0; i<n; i++) a[i] = i + rank +1;
 printf("process %d a[0] = %d\n", rank, a[0]);
 MPI_Op_create(&smod5, 1, &op);
 MPI_Reduce(a, b, n, MPI_INT, op, 0, MPI_COMM_WORLD);
 MPI_Op_free(&op);
 if(rank==0) printf("b[0] = %d\n", b[0]);
 MPI_Finalize();
}
```

Группы и коммуникаторы

Рассмотрим функцию, которая позволяет из какого-то существующего коммуникатора получать ряд других коммуникаторов.

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

Разбиение коммуникатора $comm$ на несколько по числу значений параметра $color$. В одну подгруппу попадают процессы с одним значением $color$. Процессы с бóльшим значением key получат больший ранг.

Процессы, которые не должны войти в новые группы, указывают в качестве $color$ константу $\boldsymbol{MPI\_UNDEFINED}$. Им в параметре $newcomm$ вернется значение $\boldsymbol{MPI\_COMM\_NULL}$.

```
MPI_Comm_split(MPI_COMM_WORLD, rank%3, rank, new_comm)
```

Пересылка разнотипных данных

Сообщение – массив однотипных данных, расположенных в последовательных ячейках памяти.

Для пересылки **разнотипных данных** можно использовать:

- Производные типы данных
- Упаковку данных

Производные типы данных

Производные типы данных создаются во время выполнения программы с помощью подпрограмм-конструкторов.

Создание типа:

- Конструирование типа
- Регистрация типа

Производный тип данных характеризуется последовательностью базовых типов и набором значений смещения относительно начала буфера обмена. Смещения могут быть как положительными, так и отрицательными, не требуется их упорядоченность.

Первый конструктор следующий:

```
int MPI_Type_contiguous(int count, MPI_Datatype type, MPI_Datatype *newtype)
```

Создание нового типа данных $newtype$, состоящего из $count$ последовательно расположенных элементов базового типа данных $type$.

```
MPI_Type_contiguous(5, MPI_INT, &newtype);
```

Следующий конструктор

```
int MPI_Type_vector(int count, int blocklen, int stride, MPI_Datatype type, MPI_Datatype *newtype)
```

Создание нового типа данных $newtype$, состоящего из $count$ блоков по $blocklen$ элементов базового типа данных $type$. Следующий блок начинается через $stride$ элементов после начала предыдущего.

```
count=2;
blocklen=3;
```

```
stride=5;
MPI_Type_vector(count, blocklen, stride,
MPI_DOUBLE, &newtype);
```

Создание нового типа данных (тип элемента, количество элементов от начала буфера):

```
{(MPI_DOUBLE, 0), (MPI_DOUBLE, 1), (MPI_DOUBLE, 2),
(MPI_DOUBLE, 5), (MPI_DOUBLE, 6), (MPI_DOUBLE, 7)}
```

Здесь видно, что блоки начинаются с 0 и 5, причем состоят они из 3 элементов. Отсутствуют 3,4 элементы. То есть у нас появились новые типы данных, в которые входят элементы базового типа с промежутками.

```
int MPI_Type_create_hvector(int count, int blocklen, MPI_Aint
stride, MPI_Datatype type, MPI_Datatype *newtype)
```

Создание нового типа данных $newtype$, состоящего из $count$ блоков по $blocklen$ элементов базового типа данных $type$. Следующий блок начинается через $stride$ байт после начала предыдущего.

```
int MPI_Type_create_indexed_block( int count, int blocklen, int
displs[], MPI_Datatype type, MPI_Datatype *newtype)
```

Создание нового типа данных $newtype$, состоящего из $count$ блоков по $blocklen$ элементов базового типа данных $type$. Смещения блоков с начала буфера посылки в количестве элементов базового типа данных $type$ задаются в массиве $displs$.

```
int MPI_Type_indexed(int count, int *blocklens, int *displs,
MPI_Datatype type, MPI_Datatype *newtype)
```

Создание нового типа данных $newtype$, состоящего из $count$ блоков по $blocklens[i]$ элементов базового типа данных. i-ый блок начинается через $displs[i]$ элементов с начала буфера.

```
int MPI_Type_create_hindexed(int count, int *blocklens, MPI_Aint
*displs, MPI_Datatype type, MPI_Datatype *newtype)
```

Создание нового типа данных $newtype$, состоящего из $count$ блоков по $blocklens[i]$ элементов базового типа данных. i-ый блок начинается через $displs[i]$ байт с начала буфера.

```
int MPI_Type_create_struct(int count, int *blocklens, MPI_Aint
*displs, MPI_Datatype *types, MPI_Datatype *newtype)
```

Создание структурного типа данных из $count$ блоков по $blocklens[i]$ элементов типа $types[i]$. i-ый блок начинается через $displs[i]$ байт с начала буфера.

Как пример рассмотрим следующий код:

```
blocklens[0]=3;
blocklens[1]=2;
types[0]=MPI_DOUBLE;
types[1]=MPI_CHAR;
displs[0]=0;
displs[1]=24;
```

```
MPI_Type_create_struct(2, blocklens, displs,
types, &newtype);
```

Здесь описываем структуру данных из 2-х блоков. Сдвиг для нулевого блока 0, а для первого 24.

Создание нового типа данных (тип элемента, количество байт от начала буфера):

```
{(MPI_DOUBLE,   0),   (MPI_DOUBLE,   8),   (MPI_DOUBLE,   16),
(MPI_CHAR, 24), (MPI_CHAR, 25)}
```

А регистрация типов данных производится с помощью

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

Регистрация созданного производного типа данных *datatype*. После регистрации этот тип данных можно использовать в операциях обмена. Предопределенные типы данных регистрировать не нужно.

И когда типы нам уже больше не нужны, хорошо бы их уничтожить

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Аннулирование производного типа данных *datatype*. *datatype* устанавливается в значение ***MPI_DATATYPE_NULL.***

Производные от *datatype* типы данных остаются. Предопределённые типы данных не могут быть аннулированы.

Также можно создавать копии типов данных

```
int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)
```

Некоторой проблемой для *MPI* является правильное расположение описанных данных. Первое решение заключается в том, что можно **описать структуру**, когда данные лежат в памяти подряд. А второй способ связан со взятием явных адресов объектов. Для этого предусмотрена следующая функция (больше для Фортрана, так как в Си можно взять амперсанд):

```
int MPI_Get_address(void *location, MPI_Aint *address)
```

Определение абсолютного байт-адреса *address* размещения массива *location* в оперативной памяти компьютера. Адрес отсчитывается от некоторого базового адреса, значение которого содержится в системной константе ***MPI_BOTTOM***.

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

Определение размера типа *datatype* в байтах (объёма памяти, занимаемого одним элементом данного типа).

```
int MPI_Type_get_extent( MPI_Datatype datatype, MPI_Aint *lb,
MPI_Aint *extent)
```

Для элемента типа данных *datatype* определяет смещение от начала буфера данных нижней границы *lb* и диапазон *extent* (разницу между верхней и нижней границами) в байтах.

Упаковка данных

Для упаковки данных используется следующая функция

```
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype,
void *outbuf, int outsize, int *position, MPI_Comm comm)
```

Упаковка $incount$ элементов типа $datatype$ из массива $inbuf$ в массив $outbuf$ со сдвигом $position$ байт. $outbuf$ должен содержать хотя бы $outsize$ байт.

Параметр $position$ увеличивается на число байт, равное размеру записи. Параметр $comm$ указывает на коммуникатор, в котором в дальнейшем будет пересылаться сообщение.

Для пересылки упакованных данных используется тип данных ***MPI_PACKED***.

А при распаковки данных, соответственно

```
int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

Распаковка из массива $inbuf$ со сдвигом $position$ байт в массив $outbuf$ $outcount$ элементов типа $datatype$. Массив $inbuf$ имеет размер не менее $insize$ байт.

Для того, чтобы понять размер байт-массива, правильно использовать функцию

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
```

Определение необходимого объёма памяти (в байтах) для упаковки $incount$ элементов типа $datatype$. Необходимый для упаковки размер может превышать сумму размеров пакуемых элементов данных.

Таким образом, сравним два этих способа пересылки разнотипных данных.

Производные типы данных:

- \- Сложнее использовать;
- \+ Не нужны дополнительные буфера;
- \+ Нет дополнительных копирований.

Упаковка данных:

- \+ Проще использовать;
- \- Требуется дополнительный буфер;
- \- Требуются дополнительные копирования.

Лекция 12. Компоненты суперкомпьютеров

Инфраструктура суперкомпьютера

В данной лекции речь пойдет о структуре суперкомпьютеров и о том, как они выглядят «изнутри».

Любой суперкомпьютер состоит из **двух больших частей**:

- **Инфраструктура**
  - Помещение
  - Электричество
  - Охлаждение
- **Вычислитель**
  - Сети
  - Узлы

Далее подробнее посмотрим на каждую из этих составляющих.

Инфраструктура: помещение

При конструировании суперкомпьютера нужно обратить внимание на

- Площадь.
- Шумность.
- Доступ для техобслуживания.
- Противопожарное оборудование (водой нельзя, порошком – плохо, лучше всего газом).
- Контроль доступа.

Инфраструктура: электричество

Здесь крайне важны следующие факторы:

- Источники питания, проводка, автоматы.
- ИБП (источники бесперебойного питания), PDU ($Power\ Distribution\ Unit$), место.
- Резервирование (лучше иметь разные каналы для резервов).
- Расчетное время работы от батарей.
- Проводка: фальшпол, короба, организаторы.
- Заземление, статика (важна влажность воздуха внутри комплекса).

Инфраструктура: охлаждение

- Мощность: потребляемая и реальная.
- Влажность.
- Центральное отопление, солнце (желательно отрегулировать отопление).
- Зимнее время (нужно обратить внимание на режим работы кондиционеров).
- «Открытое окно» (нельзя охлаждать систему уличным воздухом, через окна, так как с улицы приходит и пыль).

Структура кластера

- Сети: вычислительная, управляющая, хранения данных, сервисная, другие.
- Вычислительные узлы.

- Узлы доступа.
- Сетевая файловая система.
- Служебные узлы.
- Система архивирования.

Ниже представлена схема устройства суперкомпьютера (Рис.12.1).

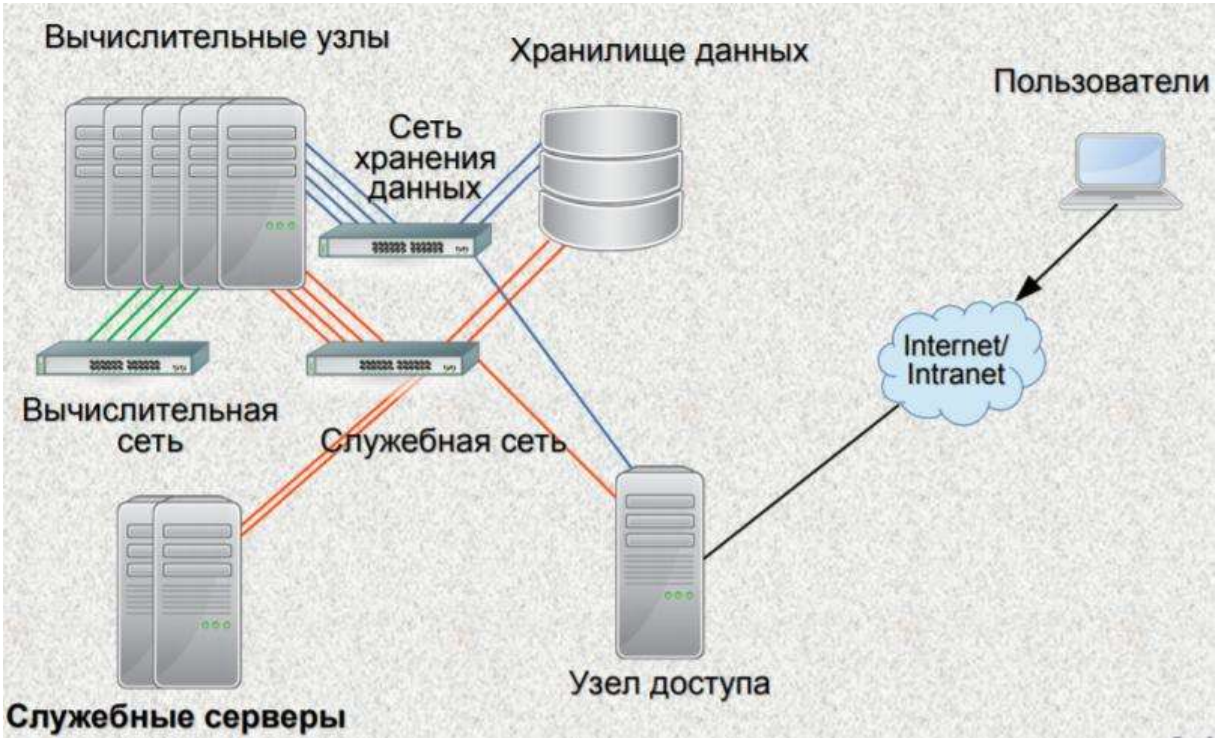


Рис.12.1. Схема устройства суперкомпьютера.

Теперь перейдем к **службам**, которые должны работать на вычислительных узлах (в зависимости от задач):

- $xCAT$ — управление образами т.п
- $DHCP$ — управление ip-адресами
- $TFTP$ — загрузка, установка
- NFS — загрузка, $root\text{-}fs$
- DNS — управление dns-именами
- NTP — синхронизация времени
- $LDAP$ — управление учётными записями

Лучше данные службы разделять на несколько серверов. Или установить их в виртуальные машины и запускать оттуда. Это сэкономит время их запуска при сбоях.

Служебные узлы осуществляют следующие **функции**:

- Доступ (управляющий)
- Управление задачами
- Лицензии ($flexlm$) (лучше иметь виртуальные машины, чтобы при сбоях не терять mac-адреса лицензионных софтов).

- Мониторинг
- Статистика
- Визуализация
- Копирование данных

После размещения железа необходимо будет поставить операционную систему и настроить все серверы на нормальную работу. Загрузка ОС производится как показано ниже причем двумя вариантами: установка локально или загрузка по сети (Рис.12.2).

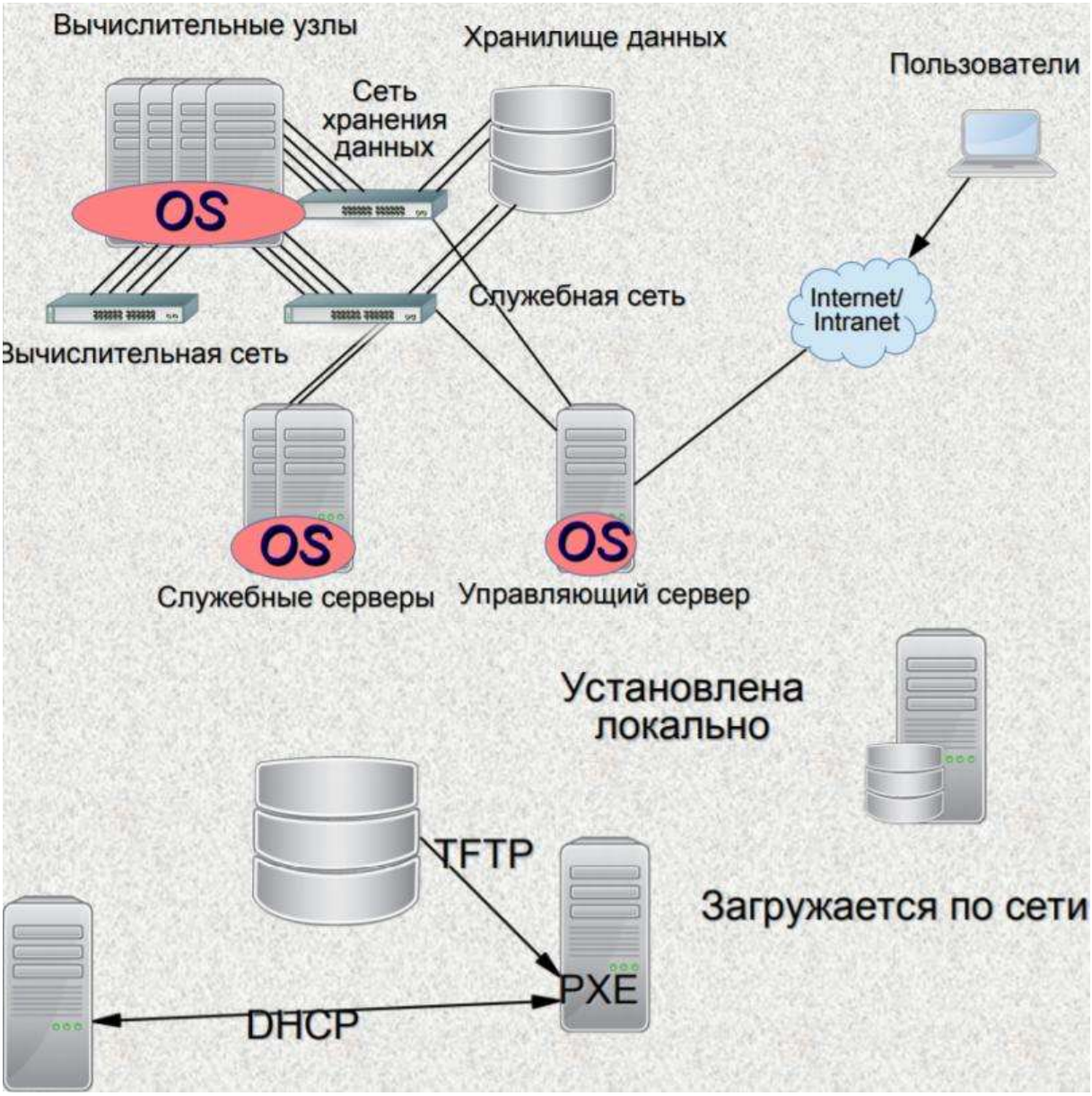


Рис.12.2. Загрузка ОС.

Для **локальной установки** следующие **плюсы:**

\+ Быстрая загрузка
\+ Нет перегруза сети

\+ Не занимает лишнего места в памяти

Однако есть и **минусы:**

\- Долгая установка и обновление

\- Рассинхронизация настроек и версий

А у **загрузки по сети** другие характеристики. **Плюсы:**

\+ Единый образ для всех узлов группы

\+ Возможность быстро обновить всё

И минусы:

\- Нагрузка на сеть

\- Много места в памяти

\- Нет возможности сохранить изменения

Если мы решили **ставить ОС на жесткий диск**, то можно

- Установить на один сервер и клонировать образ (долго).
- Автоматизировать установку.
- Воспользоваться готовыми решениями (стеки).
- Или использовать $xCAT$.

Автоматизированная установка подразумевает установку ОС один раз на сервер. Далее будем говорить про $Linux$.

После установки в каталоге $root$ будет создан файл: $\boldsymbol{anaconda\text{-}ks.cfg}$. Можно взять этот файл, скопировать его, например, на местный $\boldsymbol{http}$ сервер. И при установке на следующий узел можно указать такой аргумент

$$\boldsymbol{linux\ ks} = \boldsymbol{http}:// \text{ путь-к-файлу}.\boldsymbol{cfg}$$

После этого инсталлятор увидит этот файл и будет ставить систему по файлу.

Существуют **готовые дистрибутивы**, которые позволяют автоматизировать эту работу:

- $ROCKS$
- $PelicanHPC/Parallel\ Knoppics$

Следующий способ – **использование $\boldsymbol{xCAT}$.**

- Использует $anaconda/autoyast$
- Управляет $DNS, DHCP, TFTP$
- Поддерживает удалённое управление питанием и консоль (через $IPMI, ILO, \ldots$)
- Имеет встроенные «массовые» команды

И есть **три вида установки ОС** с использованием $\boldsymbol{xCAT}$:

1. Локально ($\boldsymbol{statefull}$).
2. По сети в память ($\boldsymbol{stateless}$).
3. По сети, в памяти только изменения ($\boldsymbol{statelite}$).

Система $\boldsymbol{xCAT}$ оперирует **объектами:** $site, \boldsymbol{node}, group,\ \boldsymbol{network}, osdistro, osimage, route. \ldots$

И **таблицами:** $domain, hosts, ipmi, litetree, mac,\ nodegroup, nodelist, nodetype, postscripts, \ldots$

Все объекты «размазаны» по таблицам. В таблицах можно быстрее настроить некоторую информацию, чем через объекты.

Ниже приведем пример создания объекта:

```
mkdef -t node node-1 groups=all,compute arch=x86_64 \
 bmc=node-1-ipmi bmcusername=ADMIN bmcpassword=admin \
 mac=xx:xx:xx:xx:xx:xx mgt=ipmi netboot=pxe \
 provmethod=centos67-x86_64-statelite-compute
```

И второй пример изменения записи в таблице:

```
chtab node=compute hosts.ip='|\D+(\d+)|10.0.0.(10+$1)|'
```

Здесь уже оперируем узлами, для которых просим записать *ip*-адрес.

Файловые системы

Теперь разберемся с файловой системой компьютера. Ее место положение приведено на Рис.12.3.

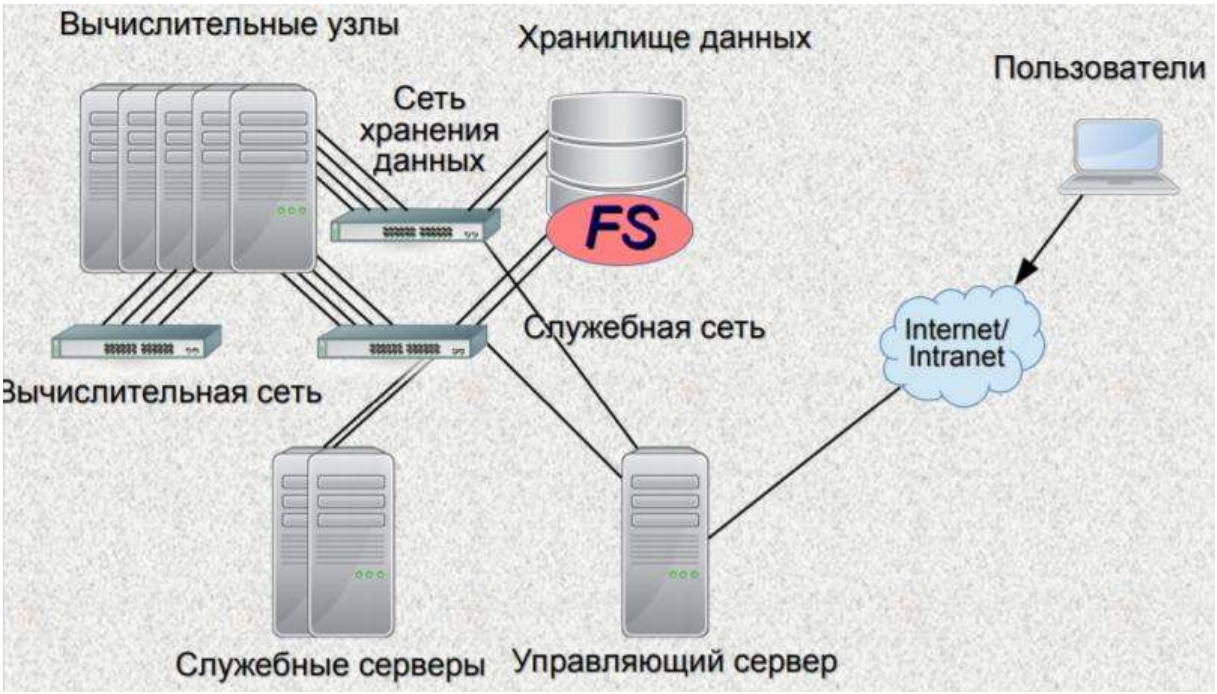


Рис.12.3. Файловая система на общей схеме суперкомпьютера.

Существуют следующие **варианты файловых систем:**

1. ***NFS***

   Есть один сервер, к которому подключен жесткий диск. Там обычная файловая система. И часть ее может раздаваться по протоколу *NFS* сразу многим вычислительными узлам. Но **минусы** такого **подхода** в том, что само место связи

с NFS может быть ограничено. И второе, сам сервер в качестве сетевого достаточно плох (Рис.12.4).

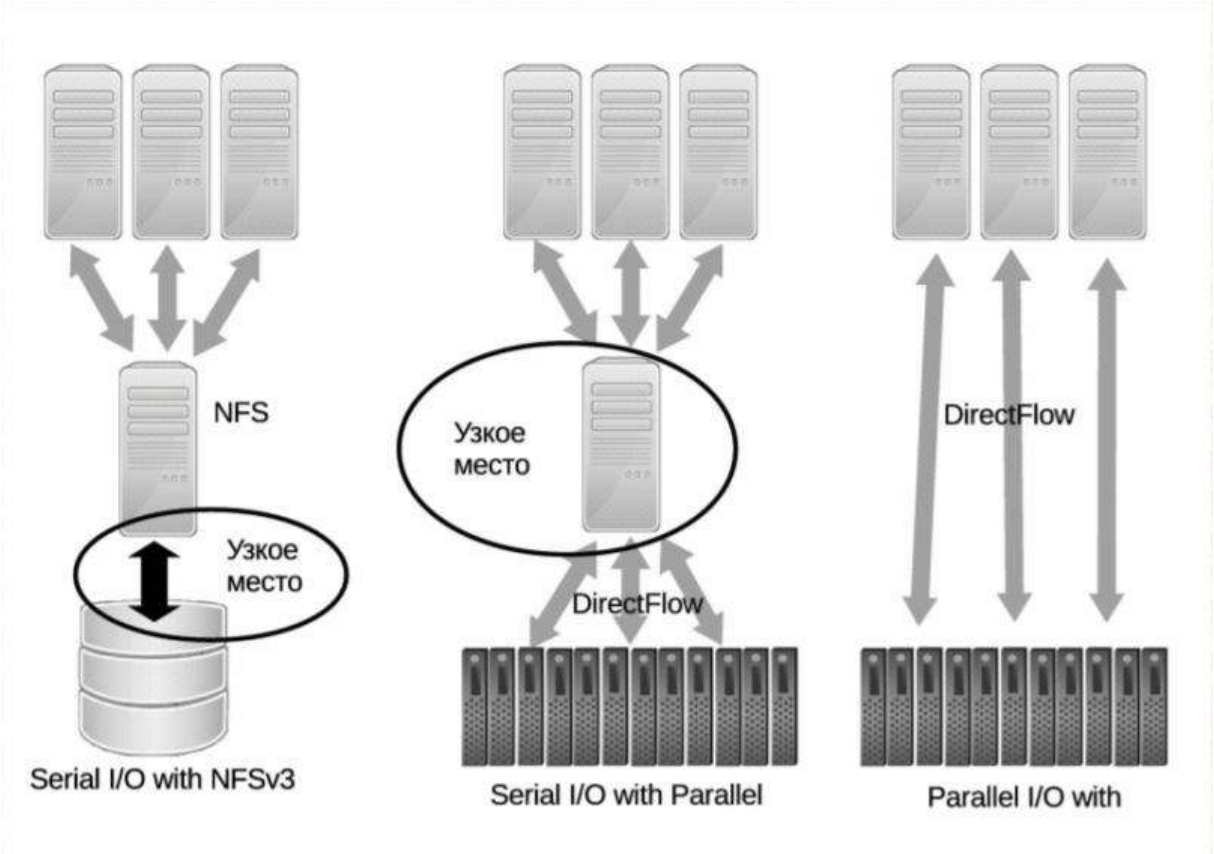


Рис.12.4. NFS (слева) и PanFS (справа).

2. ***PanFS***
   Есть набор блэйдов, в каждом из которых есть жесткие диски и специальная файловая система. Узлы работают параллельно со всеми блэйдами. Каждый узел выясняет, где находится часть того файла, с кем он хочет работать. Сеть используется равномерно и нагрузка снижается (однако все это дорого).
3. ***Lustre***
   «Люстра» устроена следующим образом (Рис.12.5). Есть один или несколько серверов метаданных (MDS). К ним подключается некоторое хранилище (MDT). Одновременно одно хранилище может быть подключено к одному серверу метаданных. Но если сервер сломался, то хранилище можно перекинуть на другой сервер. Все данные «размазываются» каким-то образом (без резерва и дублирования). Они раскидываются по объектам хранения (OST). Такой таргет может быть подключен только к одному серверу (OSS). При этом, если какой-то таргет полностью ломается, то вы автоматически теряете часть данных. А если вы теряете сервер метаданных, то вы теряете всю файловую систему.

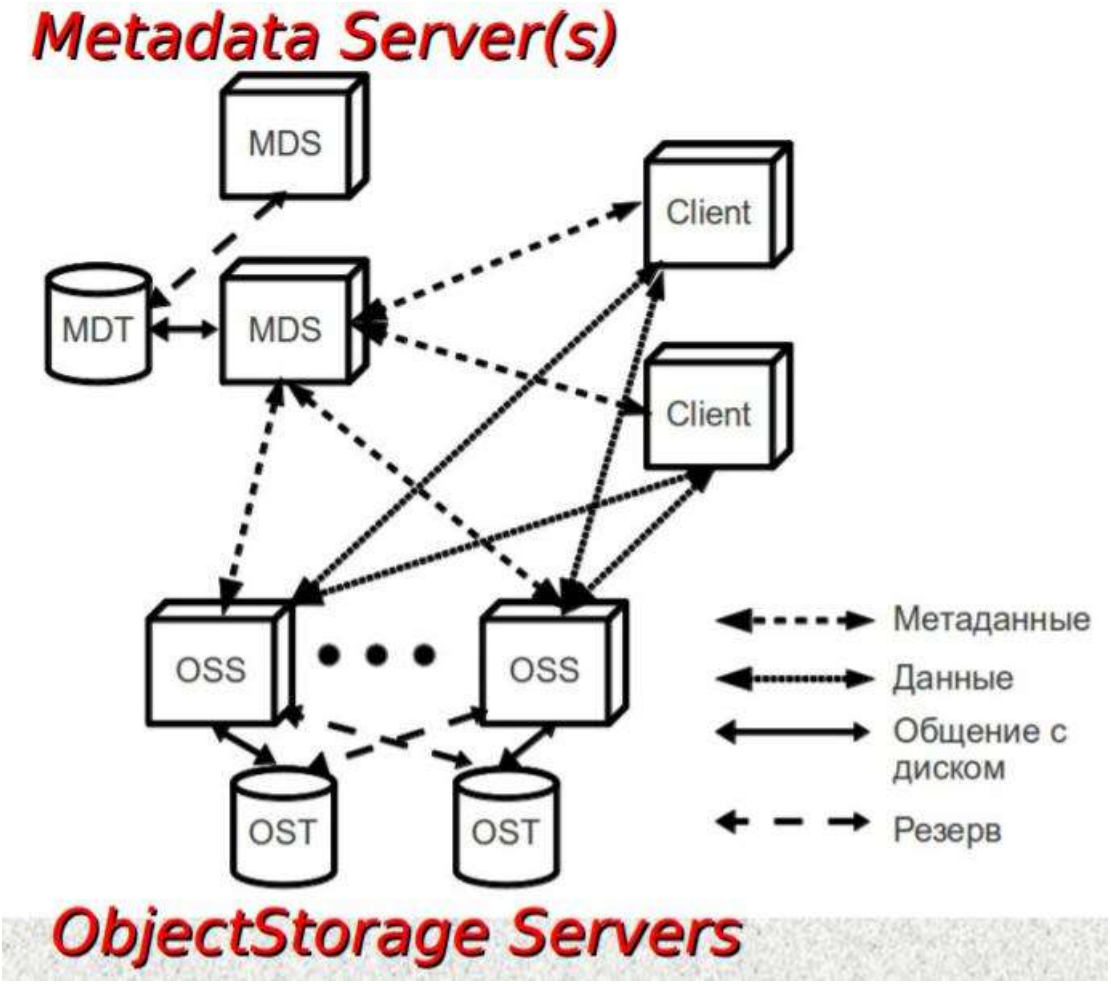


Рис.12.5. Вид файловой системы Lustre.

Контроль ресурсов

Пусть ОС у нас есть, сетевую файловую систему мы поставили. Нам нужно упорядочить запуск вычислительных задач. Для этого используются специальные менеджеры ресурсов, например:

- $Slurm$
- $Torque$
- $OpenPBS$
- $LSF, BrightManager, Moab, \ldots$

Все эти системы устроены примерно по одной схеме (Рис.12.6). У нас есть сервис $slurmctld$ (менеджер ресурсов), рядом с ним могут работать планировщики и т.д. По сети с ним связываются команды ($sbatch, sinfo, squeue,$ ...). С их помощью можно ставить задачи, менять настройки и т.д. Также на вычислительных узлах работают агенты ($slurmd$), которые сообщают о готовности этих узлов к работе. Если задача запускается с поддержкой $slurm$, то происходит запуск $slurmstepd$, которые запускают один процесс задачи.

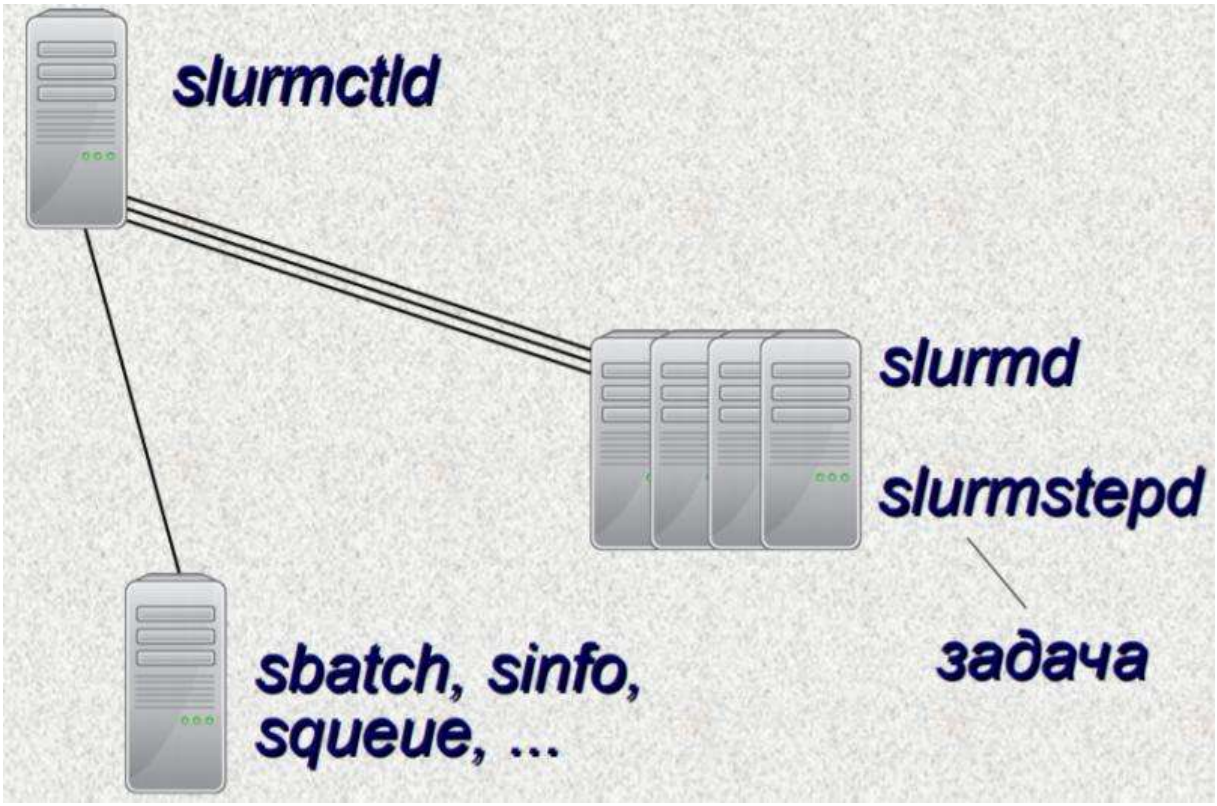


Рис.12.6. Устройство системы управления задачами.

Основная программа для управления демоном ***Slurm*** – ***scontrol,*** которая помогает на ходу менять какие-то параметры.

- $show\ node \mid job \mid partition \mid$
- $create$ …
- $delete$ …
- $update$ …

Хранение учетных записей

Для хранения учетных записей можно сделать следующее:

- Синхронизировать файлы, где хранятся учетные записи (проблематично).
- Использовать сетевые решения ($nis+$ или $LDAP$) (только до 50-70 узлов).
- Применять гибридные методы хранения (храним данные в $LDAP$ сервере, но при изменении формируем заново файлы $passwd$).

Лицензии

- Локальные и сетевые
- Привязка:
  - MAC-адрес
  - Серийный номер материнской платы
  - Файл

- Тип:
  - Список пользователей/серверов (жесткий список).
  - Число пользователей ($network$) (по количеству одновременно работающих пользователей). Такая лицензия обычно дается на какое-то время, что может стать проблемой.
  - Число одновременных запусков ($floating$)

Лекция 13. Введение в теорию анализа. Структуры программ и алгоритмов - 1

Структуры программ и алгоритмов

В заключительных лекциях будем разбираться с тем, как правильно подстраивать алгоритмы под архитектуру вычислительных систем и что нужно знать для эффективного написания программ.

С точки зрения эффективности нас не очень волнует язык программирования. Нам важно понимать, как свойства программы соответствуют свойствам архитектуры, на которую мы рассчитываем.

Вспомним пример реализации алгоритма, который мы приводили в 7 лекции.

$$A_{ijk} = A_{i-1jk} + B_{jk} + B_{jk}, \qquad i = 1{,}40;\ j = 1{,}40;\ k = 1{,}1000$$

Мы будем писать программу под машину $CRAY\ C90$, у которой пиковая производительность равна $960\ Mflop/s$.

Получаем простой код:

```
do k=1,1000
   do j=1,40
      do i=1,40
         A(i,j,k)=A(i-1,j,k)+B(j,k)+B(j,k)
```

Реальная производительность – $20\ Mflop/s$.

Тогда чтобы увеличить производительность мы начинаем подстраивать свою программу под архитектуру векторно-конвейерной машины. И после преобразования получим:

```
do i=1,40,2
   do j=1,40
      do k=1,1000
         A(i,j,k)=A(i-1,j,k)+2*B(j,k)
         A(i+1,j,k)=A(i,j,k)+2*B(j,k)
```

И сразу производительность увеличится до $700\ Mflop/s$.

Теперь перейдем к важности **устройства алгоритмов**. Для этого рассмотрим пример умножения матриц.

Фрагмент исходного текста:

```
for( i = 0; i < n; ++i)
 for( j = 0; j < n; ++j)
 for( k = 0; k < n; ++k)
 A[i][j] = A[i][j] + B[i][k]*C[k][j]
```

Порядок циклов следующий: (i, j, k).

А возможны ли другие порядки, при которых сохраняется данный результат? Оказывается, что да! И вот они:

$$(i,k,j), \qquad (k,i,j), \qquad (k,j,i), \qquad (j,i,k), \qquad (j,k,i).$$

А зачем нам нужны другие порядки, когда можно всегда использовать один? Ответ на этот вопрос можно наблюдать на Рис.13.1.

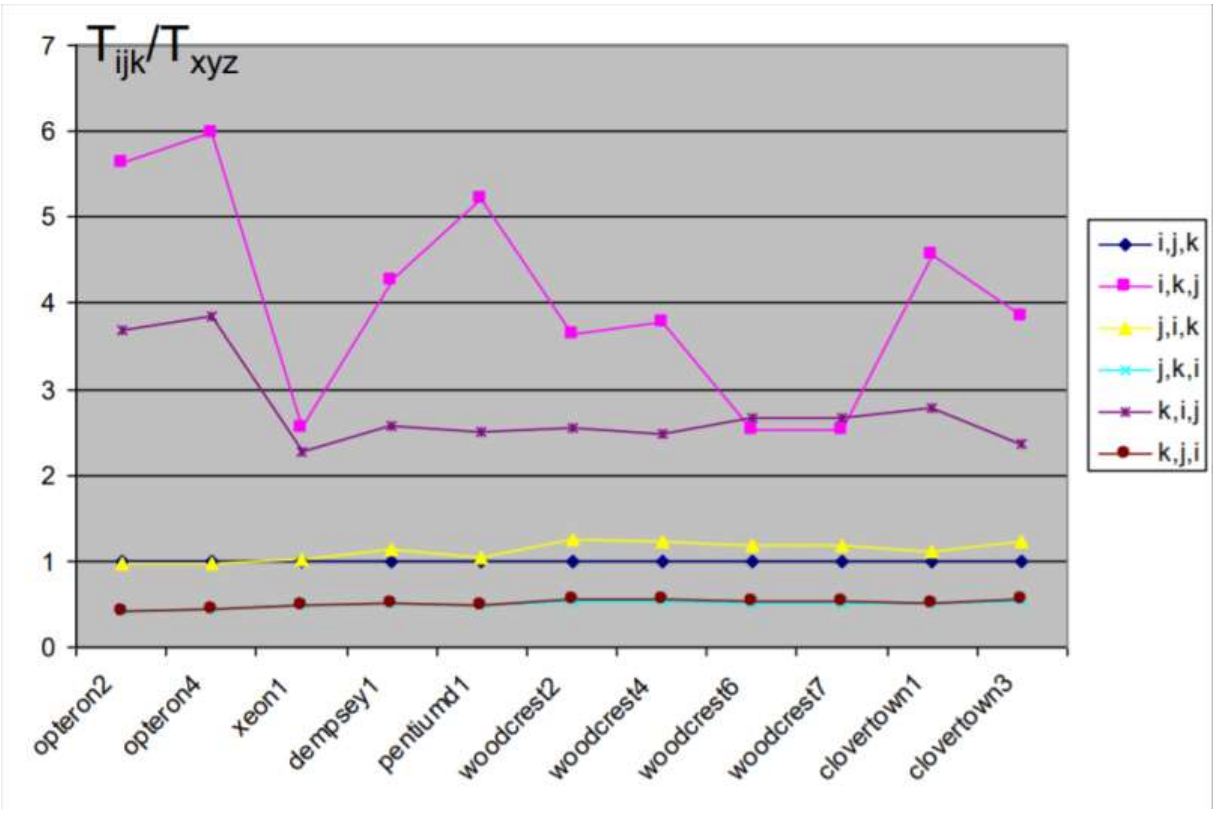


Рис.13.1. График зависимости отношения времени работы порядков цикла от разных типов процессоров.

Мы видим, что время исполнения алгоритма сильно зависит от порядка расположения циклов в программе. Эта зависимость объясняется размещением данных в памяти. То есть насколько эффективно используется кэш-память.

Графовые модели программ

Будем представлять программы с помощью **графов:** набор вершин и множество соединяющих их направленных дуг.

Вершины: процедуры, циклы, линейные участки, операторы, итерации циклов, срабатывания операторов…

Например, в качестве вершин рассмотрим **итерации циклов:**

```
for( i = 0; i < n; ++i) {
 A[i] = A[i - 1] + 2;
 B[i] = B[i] + A[i];
}
```

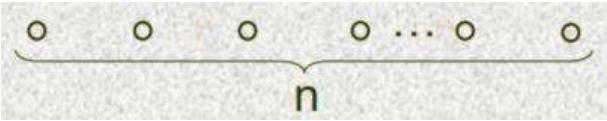


Рис.13.2. Множество вершин итераций циклов.

Каждая вершина соответствует двум операторам (телу цикла), выполненным на одной и той же итерации цикла.

Также в качестве вершин можно рассматривать **срабатывание операторов.**

Пример тот же самый, тогда множество вершин имеет вид

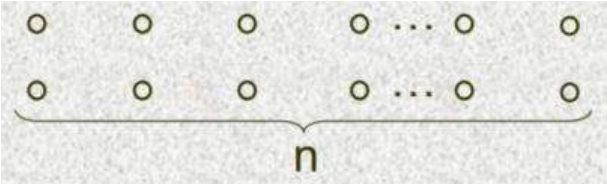


Рис.13.3. Множество вершин при срабатывании операторов.

Каждая вершина соответствует одному из двух операторов тела данного цикла, выполненному на некоторой итерации.

Теперь перейдем к **дугам.** Они отражают связь (отношение) между вершинами.

Выделяют **два типа отношений**:

- операционное отношение,
- информационное отношение.

Операционное отношение:

Две вершины A и B соединяются направленной дугой тогда и только тогда, когда вершина B может быть выполнена сразу после вершины A (Рис.13.2).

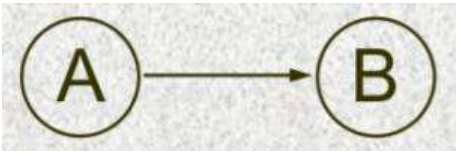


Рис.13.4. Операционное отношение.

Операционное отношение = **отношение по передаче управления.** Давайте рассмотрим пример:

```
x(i) = a + b(i)      (1)
y(i) = 2*x(i) - 3    (2)
t1 = y(i)*y(i) + 1   (3)
t2 = b(i) - y(i)*a   (4)
```

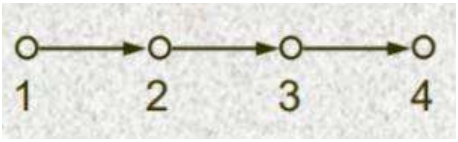


Рис.13.5. Вид дуг при операционном отношении.

Информационное отношение:

Две вершины A и B соединяются направленной дугой тогда и только тогда, когда вершина B использует в качестве аргумента некоторое значение, полученное в вершине A.

Информационное отношение = **отношение по передаче данных.** И на том же примере получим совершенно другой порядок дуг.

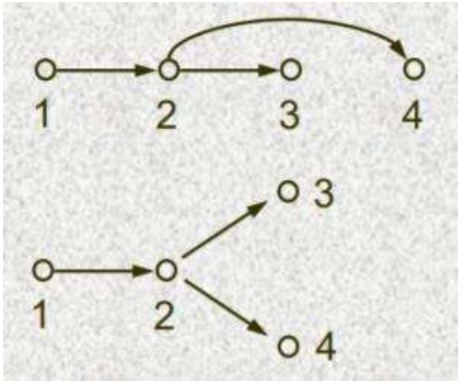


Рис.13.6. Вид дуг при информационном отношении.

Далее рассмотрим **четыре основные модели программ**, которые используются на практике.

Граф управления программы (Рис.13.7).

Вершины: операторы.

Дуги: операционные отношения.

Рассмотрим данную и все последующие модели не следующем **фрагменте кода:**

```
for( i = 0; i < n; ++i) {
 A[i] = A[i - 1] + 2;    (1)
 B[i] = B[i] + A[i];     (2)
}
```

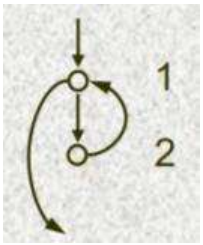


Рис.13.7. Граф управления.

Информационный граф программы (Рис.13.8).

Вершины: операторы.

Дуги: информационное отношение.

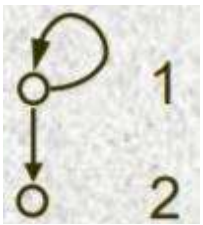


Рис.13.8. Информационный граф.

Операционная история программы (Рис.13.9).

Вершины: срабатывания операторов.

Дуги: операционное отношение.

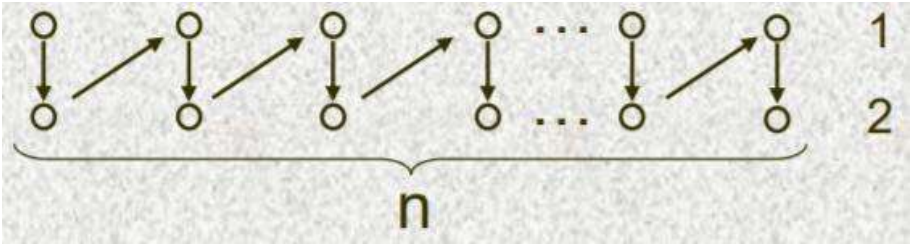


Рис.13.9. Операционная история программы.

Свойства операционной истории:

- одна начальная вершина, у которой нет входящей дуги,
- одна конечная вершина, у которой нет исходящей дуги,
- у всех остальных вершин есть ровно одна входящая дуга и одна исходящая дуга.

Информационная история программы (Рис.13.10).

Вершины: срабатывания операторов.

Дуги: информационное отношение.

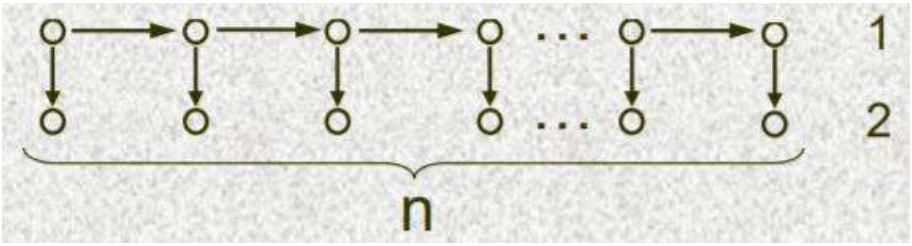


Рис.13.10. Информационная история программы.

Свойства информационной истории:

- ациклический граф,
- нет кратных дуг.

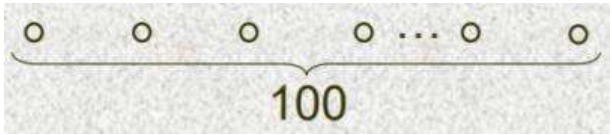
Теперь перейдем к решение несложных задач.

Задача 1.

Может ли информационная история некоторого фрагмента программы иметь 100 вершин и ни одной дуги?

Решение:

```
for( i = 0; i < 100; ++i)
 A[i] = B[i] + C[i]*x;
```

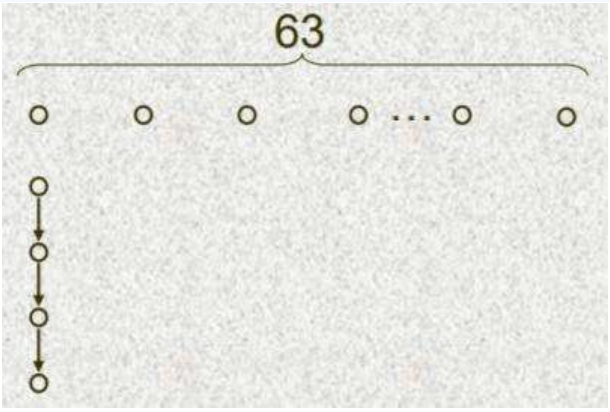


Задача 2.

Может ли информационная история некоторого фрагмента программы иметь 67 вершин и 3 дуги?

Решение:

```
for( i = 0; i < 63; ++i)
 A[i] = B[i] + C[i]*x;
x1 = 10;
x2 = x1+1;
x3 = x2+2;
x4 = x3+3;
```



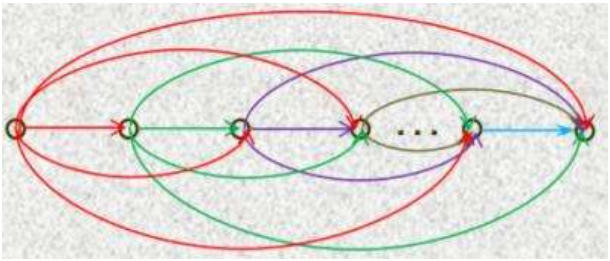
Задача 3.

Может ли информационная история некоторого фрагмента программы иметь 20 вершин и 200 дуг?

Решение:

Для решения этой задачи вспомним свойства информационной истории. Программа представляется в виде **ациклического графа**, у которого **нет кратных дуг**. Тогда максимальное число дуг будет равно:

$$(n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = n * (n-1)/2$$



Задача 4.

Может ли граф управления некоторого фрагмента программы состоять из нескольких компонент связности?

Решение:

```
x1 = 10;
x2 = x1+1; goto A;
B: x3 = x2+2; goto B;
A: x4 = x2+3;
```

Задача 5.

Есть модель некоторого фрагмента программы, которая в качестве подграфа содержит следующий граф (Рис.13.11):

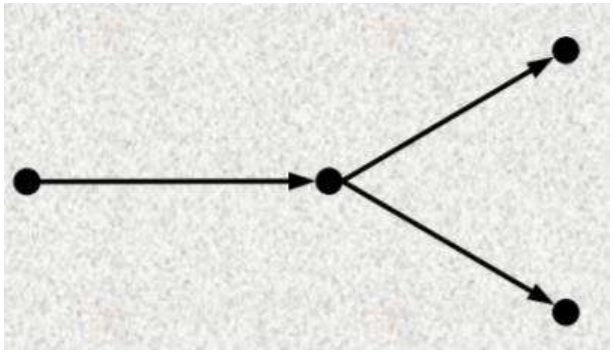


Рис.13.11. Вид графа к задаче 5.

Какой моделью могла бы быть исходная модель?

Решение:

Она могла быть графом управления, информационном графом и информационной историей. Но операционной историей она быть не могла, так как ветвлений там быть не может.

Теперь давайте посмотрим на модели программ с точки зрения параллельной реализации. Какие виды лучше всего использовать и как правильно подобрать критерии для наших задач.

Нужно ли нам операционное отношение?

```
x(i) = a + b(i)     (1)
y(i) = 2*x(i) - 3   (2)
t1 = y(i)*y(i) + 1  (3)
t2 = b(i) - y(i)*a  (4)
```

Кажется, что оно нам никак не поможет.

Тогда рассмотрим **информационную структуру** – основу анализа свойств программ и алгоритмов.

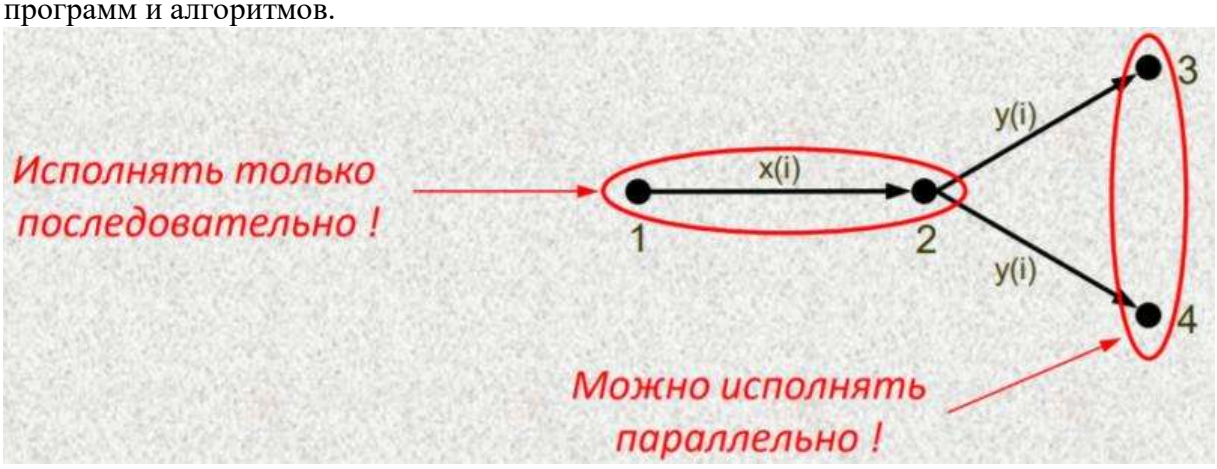


Рис.13.12. Информационная структура.

Информационная зависимость определяет критерий эквивалентности преобразований программ.

Информационная независимость определяет ресурс параллелизма программы.

После выбора информационных моделей нужно определиться с уровнем модели, то есть выбрать компактную модель или историю.

Давайте перечислим аргументы для выбора степени компактности модели:

- компактность описания (компактные +),
- информативность (истории +),
- сложность построения (компактные +).

Мы видим, что компактные модели и истории имеют свои плюсы и минусы. И на помощь приходит другая модель, которая чудным образом сочетает все преимущества других моделей в себе.

Граф алгоритма – это параметризованная информационная история:

- компактность описания за счет параметризации,
- имеет информативность истории,
- разработана методика построения графа алгоритма по исходному тексту программ.

Тогда общая схема анализа и преобразования структуры программ выглядит так (Рис.13.13):

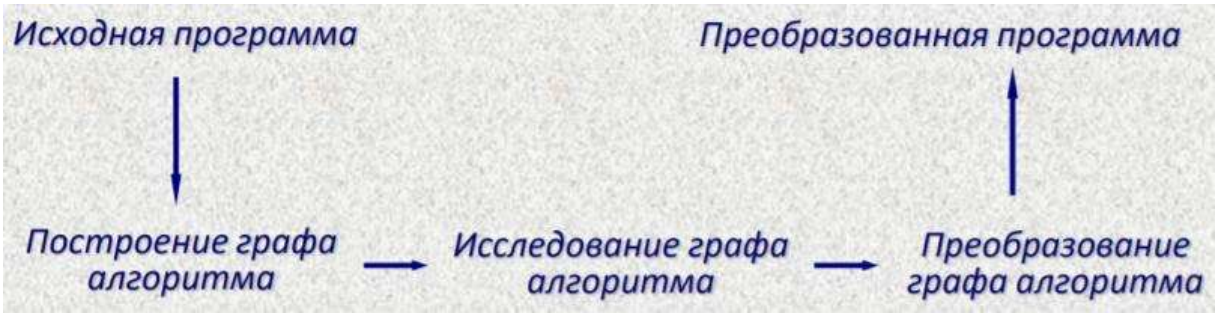


Рис.13.13. Общая схема преобразования структур программ.

Первый, кто начал заниматься подобной проблематикой, был **Андрей Петрович Ершов**. Еще в 60-х годах он рассматривал задачу преобразования схем программ над общей и распределенной памятью, изучал фундаментальные основы графовых моделей программ.

А создателем теории информационных структур программ и алгоритмов является **Воеводин Валентин Васильевич**. Он также разработал методы нахождения и описания информационной структуры программ по их исходному тексту, методы определения потенциала параллелизма и эквивалентного преобразования программ.

Далее приведем **теорему о построении графа алгоритма.**

Теорема. Если фрагмент принадлежит к линейному классу программ, то на основе статического анализа можно построить компактное описание его графа алгоритма в следующем виде: для каждого входа каждого оператора фрагмента будет указано конечное множество троек вида

$$(\mathrm{N}, \Delta(\mathrm{N}), \mathrm{F}(\Delta, \mathrm{N}))_k ,$$

где:

N – линейный выпуклый многогранник в пространстве внешних переменных фрагмента,

$\Delta(N)$ – линейный выпуклый многогранник в пространстве итераций фрагмента,

$F(\Delta, N)$ – линейная векторная функция, описывающая входящие дуги.

Давайте разберем формулировку теоремы и заменим некоторые выражения на известные и понятные для нас понятия:

«для каждого входа», здесь «вход» - аргументы операторов.

«внешних переменных фрагмента» - входные данные программы.

«пространстве итераций» - срабатывания операторов.

«входящие дуги» - информационное отношение.

На примере посмотрим на фрагмент кода, который реализует суммирование элементов матрицы.

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

Посмотрим на информационную структуру данного фрагмента (Рис.13.14).

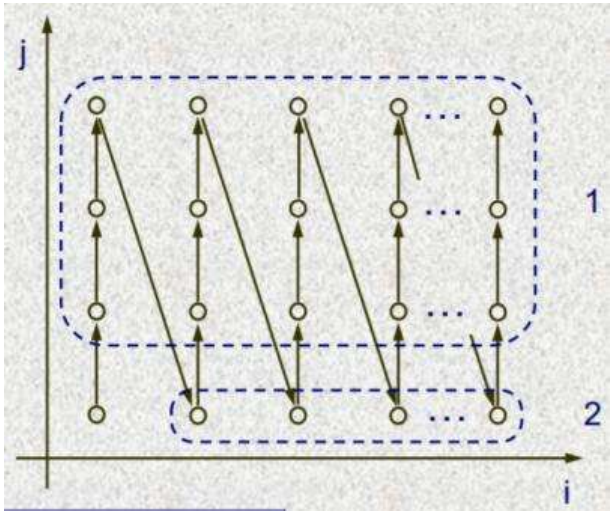


Рис.13.14. Структура алгоритма.

В данном случае рассматривается только один вход s, так как он постоянно пересчитывается. Тогда для него запишем (Рис.13.15):

$$N_1 = \begin{cases} n \geq 1 \\ m \geq 2 \end{cases} \quad I_1 = \begin{cases} 1 \leq i \leq n \\ 2 \leq j \leq m \end{cases} \quad F_1 = \begin{cases} i' = i \\ j' = j - 1 \end{cases}$$

$$N_2 = \begin{cases} n \geq 2 \\ m \geq 1 \end{cases} \quad I_2 = \begin{cases} 2 \leq i \leq n \\ j = 1 \end{cases} \quad F_2 = \begin{cases} i' = i - 1 \\ j' = m \end{cases}$$

Рис.13.15. Первая тройка (1) описывает все вершины с короткими дугами. Вторая тройка (2) – описание длинных дуг.

Рассмотрим еще один фрагмент

```
s = 0          (1)
Do i = 1, n
 s = s + 1     (2)
Do i = 1, m
 s = s + 1     (3)
s = s + 1      (4)
```

И граф алгоритма для входа *s* из (4) оператора выглядит так (Рис.13.16):

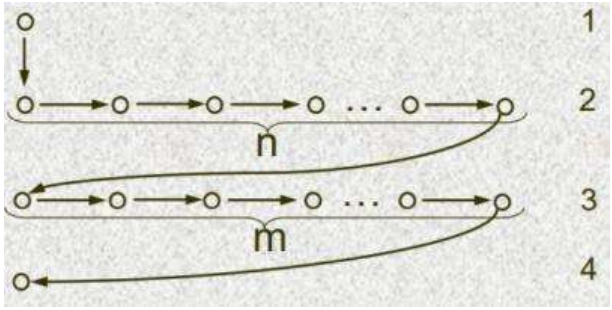


Рис.13.16. Граф алгоритм.

А его описание задается тремя тройками (Рис.13.17):

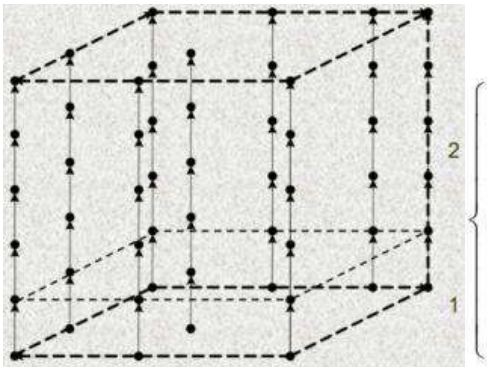
$$\begin{cases} m \ge 1 \\ j_1 = m \\ из\ 3 \end{cases} \qquad \begin{cases} m < 1 \\ n \ge 1 \\ j_1 = n \\ из\ 2 \end{cases} \qquad \begin{cases} m < 1 \\ n < 1 \\ из\ 1 \end{cases}$$

Рис.13.17. Описание граф алгоритма.

После изучения теории граф алгоритма вернемся к примеру с умножением матриц, где не важен порядок написания циклов.

```
Do i = 1, n
 Do j = 1, n
 A(i,j) = 0
 Do k = 1, n
 A(i,j) = A(i,j) + B(i,k)*C(k,j)
```

Тогда сам граф и его описание будут иметь вид (Рис.13.18):



$$\begin{cases} 1 \le i \le n \\ 1 \le j \le n \\ k = 1 \\ \begin{cases} i_1 = i \\ j_1 = j \end{cases} \\ из\ (1) \end{cases} \qquad \begin{cases} 1 \le i \le n \\ 1 \le j \le n \\ 2 \le k \le n \\ \begin{cases} i_1 = i \\ j_1 = j \\ k_1 = k - 1 \end{cases} \\ из\ (2) \end{cases}$$

Рис.13.18. Вид и описание графа алгоритма умножения матриц.

Лекция 14. Введение в теорию анализа. Структуры программ и алгоритмов - 2

В данной лекции поговорим о **способах описания ресурса** параллелизма программ и алгоритмов.

Ярусно-параллельная форма графа алгоритма

Информационная история не имеет циклов, кратных дуг и его можно сделать направленным. Расположим вершины на ярусах, которые перенумеруем и сделаем его направленным в одну сторону (Рис.14.1).

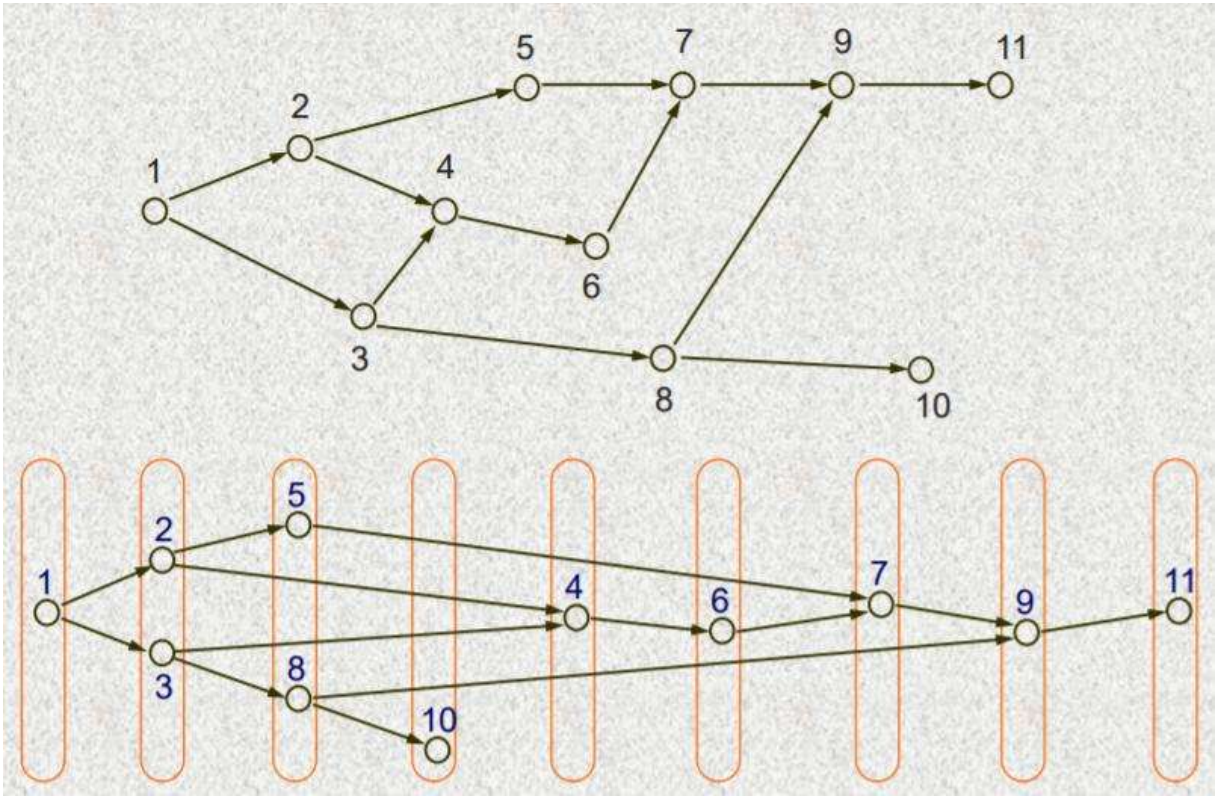


Рис.14.1. Сверху вид графа, а внизу его ярусно-параллельная форма (ЯПФ).

Каждый набор множеств – **ярусы**. И у каждой дуги очень простые свойства:

- начальная вершина каждой дуги расположена на ярусе с номером меньшим, чем номер яруса конечной вершины,
- между вершинами, расположенными на одном ярусе, не может быть дуг.

Полезность ЯПФ заключается в том, что она показывает способ параллельного исполнения алгоритма или программы. **Высота ЯПФ** – это число ярусов, **ширина яруса** – число вершин, расположенных на ярусе, **ширина ЯПФ** – это максимальная ширина ярусов в ЯПФ.

Высота ярусно-параллельной формы - это **сложность параллельной реализации** алгоритма/программы.

ЯПФ определяется неоднозначно.

Рассмотрим **каноническую ЯПФ.** Ярусно-параллельная форма называется **канонической**, если у любой вершины, кроме вершин первого яруса, есть входная дуга, идущая с предыдущего яруса (Рис.14.2).

Высота канонической ЯПФ = длине критического пути + 1.

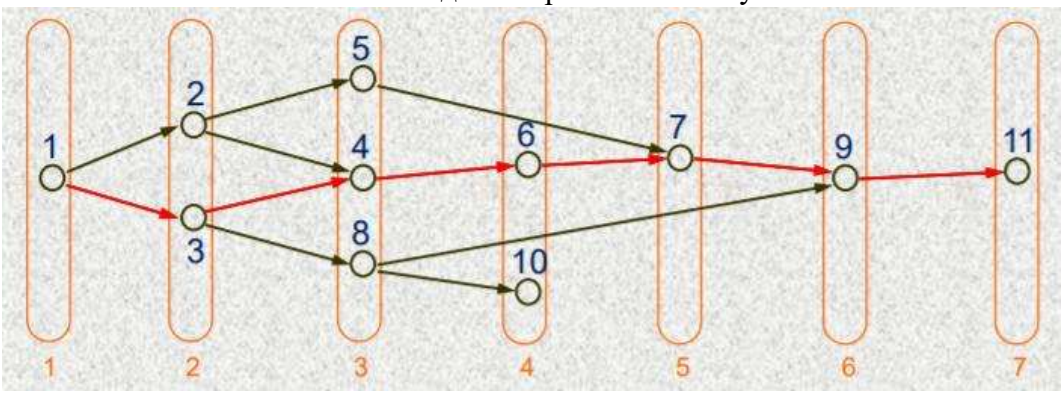


Рис.14.2. Каноническая ЯПФ.

Рассмотрим данное применение на практике.

```
for( i = 0; i < n; ++i)
 for( j = 0; j < m; ++j)
 A[i][j] = A[i][j-1] + C[i][j]*x;
```

Нам требуется **ответить на вопрос:** Чему, согласно закону Амдала, равно максимальное ускорение, которое можно получить при исполнении данного фрагмента на параллельной вычислительной системе?

Для ответа воспользуемся формулой закона Амдала.

$$S \approx \frac{1}{\alpha}, \qquad \alpha = \frac{\text{число послед. операций}}{\text{общее число операций}} = \frac{m}{n * m} = \frac{1}{n}$$

Где α мы нашли с помощью информационной структуры данного алгоритма (Рис.14.3).

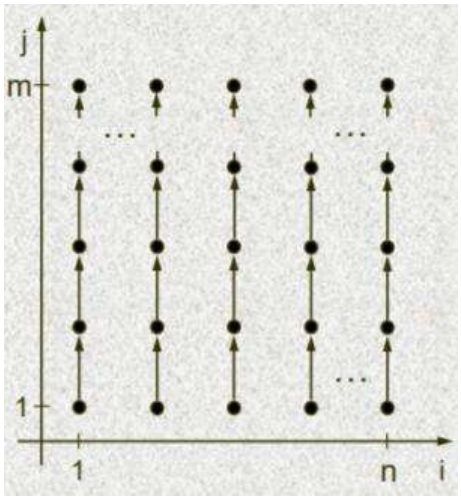


Рис.14.3. Информационная структура алгоритма.

Тогда получили, что $S \approx n$

Виды параллелизма в алгоритмах и программах

Параллелизм делится на два больших класса: **конечный и массовый параллелизмы.**

Конечный параллелизм определяется информационной независимостью некоторых фрагментов в тексте программы (Рис.14.4).

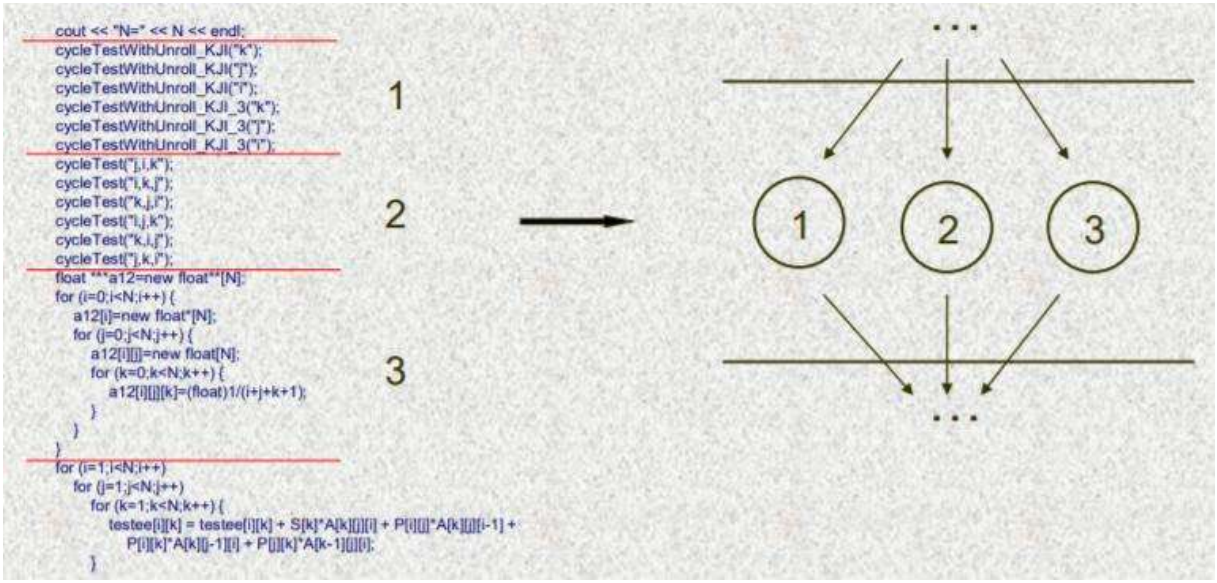


Рис.14.4. Конечный параллелизм.

Массовый параллелизм определяется информационной независимостью итераций циклов программы (Рис.14.5).

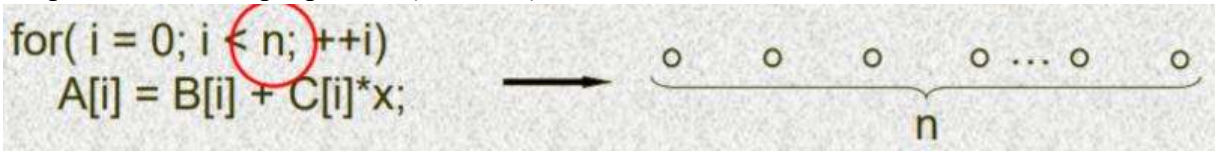


Рис.14.5. Массовый параллелизм.

Массовый параллелизм также делится на два класса: **координатный и скошенный.**

Координатный появляется в том случае, когда множество независимых операций лежат на гиперплоскостях, параллельных из координатных осей. В таком случае выразить такой параллелизм просто: поставим $\#pragma\ omp\ parallel$. Был последовательный фрагмент, стал параллельным.

Утверждение: для того чтобы цикл был параллельным необходимо и достаточно, чтобы для любой тройки графа алгоритма данного цикла включение $\Delta_i \subset G_i$ было верным, где

Δ_i – это многогранник из тройки, $G_i = \{f_1 = i_1\}$,

i_1 — это параметр анализируемого цикла,

f_1 — это первая компонента векторной функции F_i из тройки.

Скошенный параллелизм.

Рассмотрим на примере

```
for( i = 0; i < n; ++i)
 for( j = 0; j < m; ++j)
 A[i][j] = A[i][j-1] + A[i-1][j]*x;
```

Дуги идут по двум координатным осям, причем ни по j, ни по i нет координатного параллелизма (Рис.14.6). Здесь длина критического пути равна $(n+m)^k$.

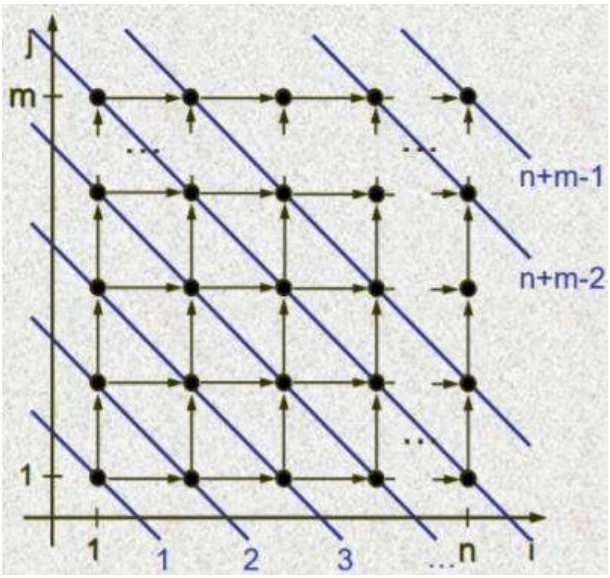


Рис.14.6. Структура скошенного параллелизма.

Есть сложности у такого параллелизма. Он очень неудобный, так как в начале на каждом ярусе довольно мало независимых операций. Кроме того, выразить его не так просто. Мы не можем поставить $\#pragma$ ни перед одним из циклов for. Придется переписать программу так, чтобы внешний цикл задавал переборы гиперплоскостей, а внутренний перебирал операции, лежащие на этих гиперплоскостях.

Элементарные преобразования циклов

Начнем с преобразования **перестановки циклов.** На следующем примере сделаем это

```
for( i = 0; i < n; ++i)
 for( j = 0; j < m; ++j)
    A[i][j] = A[i][j-1] + A[i-1][j]*x;
```

Цикл по i последовательный, по j – параллельный. Нет никакой зависимости. И для каждой итерации по i, множество вершин, лежащих на гиперплоскости независимо. Однако это очень неудобно.

Давайте поменяем структуру алгоритма:

```
#pragma omp parallel for
for( j = 0; j < m; ++j)
 for( i = 0; i < n; ++i)
    A[i][j] = A[i][j-1] + A[i-1][j]*x;
```

То есть мы получили новый обход вершин данного алгоритма. Происходит порождение параллельных ветвей один раз.

Теперь подумаем, всегда ли перестановка циклов является эквивалентным преобразованием?

```
for( i = 0; i < n; ++i)
 for( j = 0; j < m; ++j)
    A[i][j] = A[i+c1][j+c2] + C[i][j]*x;
```

И следующие значения констант c1 и c2 не дадут нам эквивалентного преобразования в случае перестановки циклов.

```
for( i = 0; i < n; ++i)
 for( j = 0; j < m; ++j)
    A[i][j] = A[i-1][j+1] + C[i][j]*x;
```

Еще одно интересное преобразование, которое позволяет вычленить максимум последовательного или параллельного из данного фрагмента. **Это распределение циклов.**

```
for( i = 1; i < n; ++i) {
 A[i] = A[i-1]*p + q;
 C[i] = (A[i] + B[i-1])*s;
 B[i] = (A[i] - B[i])*t;
}
```

Давайте поймем, как сделать этот фрагмент параллельным.

Утверждение: для того чтобы можно было выполнить распределение цикла необходимо и достаточно, чтобы распределяемые части находились в разных компонентах сильной связности информационного графа тела данного цикла.

```
for( i = 1; i < n; ++i)
 A[i] = A[i-1]*p + q;
#pragma omp parallel for
for( i = 1; i < n; ++i)
 B[i] = (A[i] - B[i])*t;
#pragma omp parallel for
for( i = 1; i < n; ++i)
 C[i] = (A[i] + B[i-1])*s;
```

В данном случае преобразование выглядит таким образом. Три цикла с нашими операторами.

Далее рассмотрим пример **расщепления циклов.**

```
for( i = 501; i <= 2000; ++i)
 A[i] = A[i] + A[i-500];
```

И после расщепления данного цикла на три части, получим уже фрагмент с параллельным исполнением:

```
#pragma omp parallel for
for( i = 501; i <= 1000; ++i)
```

```
 A[i] = A[i] + A[i-500];
#pragma omp parallel for
for( i = 1001; i <= 1500; ++i)
 A[i] = A[i] + A[i-500];
#pragma omp parallel for
for( i = 1501; i <= 2000; ++i)
 A[i] = A[i] + A[i-500];
```

Теперь ответим на вопрос: Означает ли малый размер программы простоту ее информационной структуры?

Для этого рассмотрим фрагмент программы

```
DO i = 1, n
 DO j = 1, n
   U( i + j ) = U( 2*n - i - j + 1)*q + p
 EndDO
EndDO
```

Здесь происходит перевычисление массива U. А вот информационная структура этого примера очень непростая (Рис.14.7).

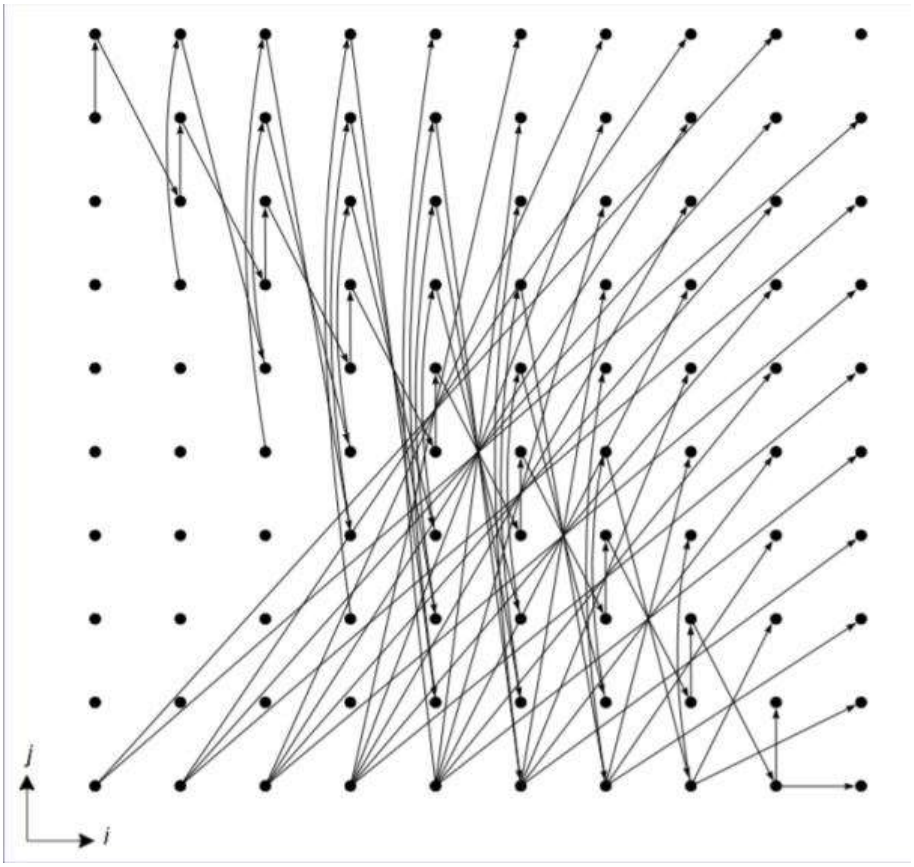


Рис.14.7. Информационная структура примера.

И после приведения данной структуры к ЯПФ, можно представить фрагмент кода с параллельной реализацией:

```
DO i = 1, n
```

```
 DO j = 1, n - i
   U( i + j ) = U( 2*n - i - j + 1)*q + p
 End DO
 DO j = n - i + 1, n
   U( i + j ) = U( 2*n - i - j + 1)*q + p
 End DO
End DO
```

Метод Гаусса

Давайте рассмотрим решение верхней треугольной системы **методом Гаусса.**

Примерная схема программы следующая:

```
do i = n, 1, -1
 s = 0
 do j = i+1, n
 s = s + A(i,j)*x(j)
 end do
 x(i) = (b(i) - s)/A(i,i)
end do
```

А информационная структура выглядит так (Рис.14.8).

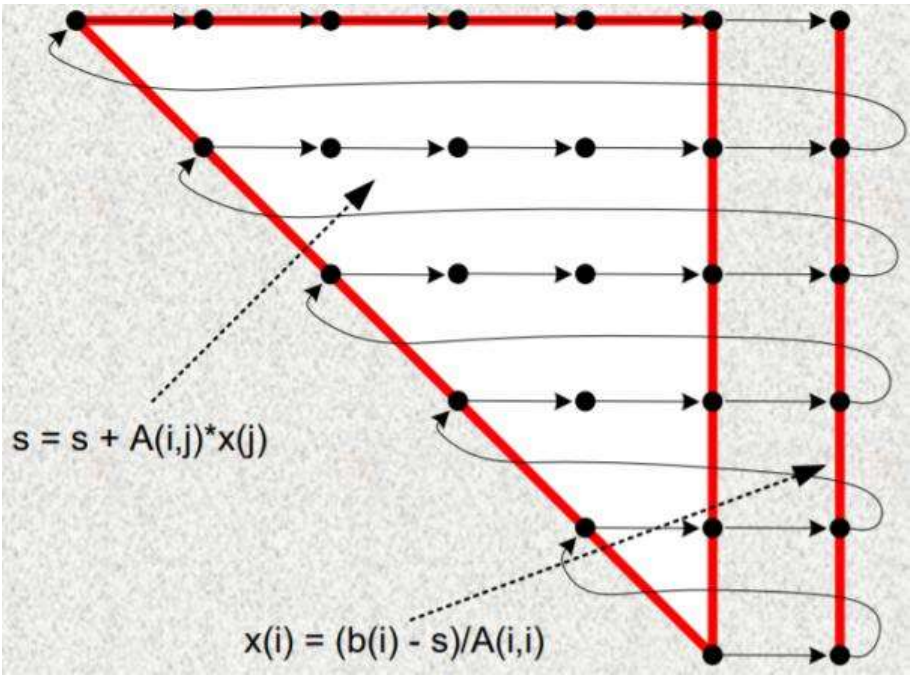


Рис.14.8. Информационная структура при решении треугольной системы методом Гаусса.

Критический путь графа алгоритма проходит через все вершины, следовательно такую программу нет смысла исполнять на параллельной вычислительной системе!

С точки зрения метода Гаусса не имеет никакого значения, в каком порядке выполнять итерации внутреннего цикла по j (суммирование).

Поэтому давайте Изменим в программе только этот порядок и определим информационную структуру (Рис.14.9).

```
do i = n, 1, -1
 s = 0
 do j = n, i+1, -1
 s = s + A(i,j)*x(j)
 end do
 x(i) = (b(i) - s)/A(i,i)
end do
```

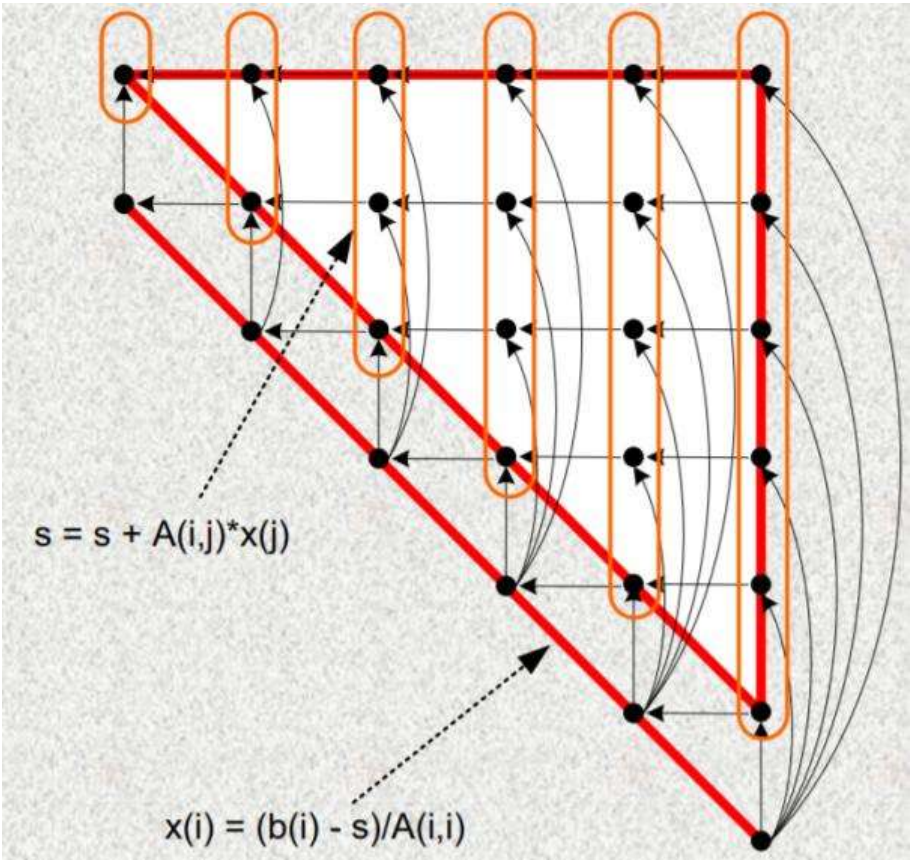


Рис.14.9. Информационная структура в случае изменения внутреннего цикла по j.

Критический путь графа алгоритма имеет длину $o(n)$, следовательно данная программа обладает хорошим ресурсом параллелизма!

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И
КИБЕРНЕТИКИ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА

teach-in

ЛЕКЦИИ УЧЕНЫХ МГУ