### Лекция 3

# Почему суперкомпьютеры имеют столь высокую производительность?

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

# Два вида параллельной обработки данных

## Параллелизм

Есть одно устройство, с некоторой производительностью. Если нам не хватает этой производительности, то мы берём 3 таких же устройства, они распределяют работу между собой, и у нас есть надежда, что общая работа будет выполнена в 3 раза быстрее. Если и этого не хватает, то берём 5 таких устройств, 10, 15, итд.

## Конвейерность

Есть какая-то макрооперация, мы разбиваем её на более мелкие операции и выполняем их последовательно одну за другой. Очень многие операции допускают такое разбиение (например, даже сложение двух вещественных чисел. Если есть несколько пар аргументов, можно постепенно загружать конвейер, и одновременно будут обрабатываться по несколько пар аргументов.

**Длина конвейера** L -- число ступеней конвейера.

Каждая ступень срабатывает за 1 такт, **темп выдачи результатов** -- каждый такт.

**Время обработки всего набора из N входных данных**: T = L + (N - 1), L -- время (число тактов) для заполнения всего конвейера (одновременно обработан первый входной аргумент), (N - 1) -- время (число тактов) для обработки оставшегося набора из N-1 входных аргументов.

**???** Если каждая ступень срабатывает за Ti тактов, то за max(Ti) тактов будут выдаваться результаты в установившемся режиме. Эта ячейка, которая срабатывает за max(Ti) тактов будет узким местом.

**Зацепление функциональных устройств** (часто происходит на практике) -- такой режим обработки, при котором выход одного функционального устройства подаётся на вход другого.

L1 -- длина конвейера сложения

L2 -- длина конвейера умножения

Нужно выполнить операцию Ai = Bi + Ci \* d, i = 1, …, N

Время обработки в режиме без зацепления: T = L2 + (N - 1) + L1 + (N - 1)

Время обработки в режиме с зацеплением: T = L2 + L1 + (N - 1)

# Иерархия памяти

* регистры
* кэш-память 1 уровня
* кэш-память 2 уровня
* оперативная память
* дисковые устройства
* ленточные устройства
* ...

Чем выше уровень в иерархии, тем выше скорость выборки, тем выше стоимость и тем меньше объём

# Свойства работы программ с памятью

## Локальность

* Локальность вычислений
* Локальность использования данных

Чем меньше блуждаем по уровням, чем меньше обращаемся к нижним уровням иерархии, тем быстрее работает программа.

Ключевое понятие -- **локальность**: насколько близко расположены данные/команды друг относительно друга.

* **Локальность данных**
* **Пространственная** локальность данных (отражает среднее расстояние по памяти между запрашиваемыми данными)
* **ВременнАя** локальность данных (показывает среднюю частоту обращений по одному адресу в память за время выполнения программы)
* **Локальность команд**

## Характеристики параллельных программ

f -- **доля последовательных операций** (0 <= f <= 1)

p -- **число процессоров** (ядер, вычислительных узлов)

T1 -- **время работы программы на одном процессоре**

Tp -- **время работы программы на системе из p процессоров**

S = T1 / Tp -- **ускорение работы программы при переходе с одного процессора на систему из p процессоров**

## Закон Амдала

Ускорение работы программы при переходе с одного процессора на систему из p процессоров можно оценить следующим образом:

**S <=**

На практике S < p, идеальный случай -- линейное ускорение S = p.

## Следствие закона Амдала

* При большом числе процессоров S ~ 1/f  
  На практике: если доля последовательных операций в некоторой программе равна 0.1, значит, что **вне зависимости от числа используемых процессоров** ускорение не превысит 10.
* В теории: для того, чтобы ускорить программу в q раз, необходимо ускорить не менее, чем в q раз, не менее, чем (1 - 1/q)-ю часть программы.

На практике: Нужно ускорить работу программы в 100 раз. Значит, необходимо ускорить не менее, чем в 100 раз, не менее, чем 99% этой программы.

# Производительность компьютера

**Пиковая производительность компьютера Rpeak**

Определяется по архитектуре компьютера в предположении, что все устройства работают с полной загрузкой, без остановок, необходимые аргументы всегда готовы, а передача данных выполняется мгновенно. Это теоретически максимальная производительность компьютера.

**Реальная производительность компьютера Rmax**

Индивидуальна для каждой программы и/или каждого компьютера. Определяется по параметрам работы программы на компьютере:

Rmax = число операций в программе / время выполнения программы на компьютере

**Пиковая производительность недостижима на практике: Rmax < Rpeak**

На тестах и простых примерах: Rmax < Rpeak

В реальности: Rmax << Rpeak

**Эффективность** = Rmax / Rpeak

Эффективность показывает, в какой степени (насколько полно) используются вычислительные возможности компьютера.

### Лекция 4

# Основные показатели эффективности и масштабируемости параллельных программ

## Основные определения и обозначения

p -- число процессоров (процессорных ядер или каких-то вычислителей которые производят параллельные вычисления) [Какие-то вычислительные единицы, в общем. Далее процессоры.]

T1 -- время работы программы на одном процессоре

Tp -- время работы программы на p процессорах

**Ускорение (speedup)** S = T1/Tp, где T1 -- время исполнения распараллеленной проги на p процессорах, T1 -- время исполнения исходной программы

В идеальном случае (отсутствие накладных расходов на организацию параллелизма) получаем S = p -- **линейное ускорение** (равное числу процессоров).

**Суперлинейное ускорение** S > p

Самая частая причина суперлинейного ускорения -- более эффективное использование кэш-памяти. Исходную расчётную область делим на много-много процессоров (например, на 64), каждому процессору достаётся по чуть-чуть, этот объём очень легко может уложиться в кэш-памяти.

**Эффективность реализации** программы Rmax/Rpeak определяется как отношение реальной производительности Rmax к пиковой производительности Rpeak.

Пиковая производительность -- теоретическая характеристика, к тому же недостижима на практике (чаще всего). Тут выясняем, насколько близко удаётся подобраться к этой теоретической производительности.

Эффективность реализации программы всегда <= 1 (чаще всего строго меньше 1, поскольку к пиковой производительности можно только приблизиться в том или ином смысле)

Чем ближе этот показатель к 1, тем лучше для пользователя, поскольку говорит о том, что более эффективно задействованы ресурсы компьютера.

Этот показатель используют много где, например, в списке Top-500 (производительность на Linpack к пиковой производительности)

**Эффективность распараллеливания** E = S/p определяет среднюю долю времени выполнения параллельного алгоритма, в течение которого процессоры реально используются для решения задачи.

(получили на p процессорах ускорение S и делим его на p, чтобы понять какая доля этого ускорения приходится на каждый процессор)

Оценка качества распараллеливания предполагает получение наилучших (максимальных) значений ускорения и эффективности распараллеливания.

Получение большого ускорения за счёт большого числа процессоров зачастую приводит к снижению эффективности распараллеливания (*см. накладные расходы*).

**Стоимость (cost) вычислений** C = p \* Tp

(Суммарное время работы всех процессоров)

**Суммарные накладные расходы (total overhead)** T0 = p \* Tp - T1

(Стоимость минус время выполнения программы на одном процессоре)

Накладные расходы, как правило, > 0. При увеличении числа процессоров p значение, как правило, возрастает (в первую очередь из-за дополнительных накладных расходов на коммуникацию, как на самые долгие операции в современных компьютерах)

**Формулы для ускорения и эффективности**:

Tp = (T1 + T0) / p

S = T1 / Tp = p \* T1 / (T1 + T0)

E = S / p = T1 / (T1 + T0) = 1 / (1 + T0 / T1)

Если T1 фиксировано (например, фиксируем число операций, выполняемых этой программой), то при увеличении числа процессоров p эффективность распараллеливания E, как правило. уменьшается за счёт роста накладных расходов T0.

Если фиксировано число процессоров p (например, фиксируем T0?..), то эффективность распараллеливания E можно увеличить, увеличивая сложность решаемой задачи T1 (масштаб решаемой программы).

**Масштабируемость (scalability)** -- способность системы увеличивать свою производительность при добавлении ресурсов (обычно аппаратных).

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

Масштабируемость можно оценить через отношение прироста производительности системы к приросту используемых ею ресурсов.

Чем ближе это соотношение к 1, тем масштабируемость лучше.

Масштабируемость:

* компьютера или его компонент (например, коммуникационной сети)
* алгоритмов **без**относительно к компьютеру
* параллельных программ относительно данного компьютера

## Масштабируемость компьютеров

**Вертикальная масштабируемость (масштабируемость вглубь, scale up)** -- возможность замены платформы, в которой функционирует система, на новую, обладающую большей производительностью.

Например: взяли компьютер, заменили в нём процессоры на более новые, более мощные, получили бОльшую производительность, оцениваем её и в результате масштабируемость этого компьютера.

**Горизонтальная масштабируемость (масштабируемость вширь, scale out)** -- возможность увеличения производительности системы за счёт добавления дополнительных программных или аппаратных средств.

Например: взяли компьютер, добавили к нему таких же узлов, какие в нём были, и оцениваем общий рост (чаще всего рост) производительности.

## Масштабируемость программ или алгоритмов

**Вычислительная сложность задачи** W -- количество основных вычислительных шагов лучшего последовательного алгоритма, необходимость для решения задачи на одном процессоре.

(Берём лучший последовательный алгоритм для решаемой задачи и смотрим, сколько вычислительных шагов нужно сделать, чтобы этот алгоритм реализовать на однопроцессорной системе. Вычислительные шаги могут быть совершенно разные, от этого может зависимость оценка W. Во многих случаях здесь можно достаточно сильно огрублять, считая вычислительные шаги все примерно одинаковыми, даже единичными для простоты)

W -- некоторая функция от размера входных данных.

Если для простоты предположить, что каждый основной вычислительный шаг выполняется за единицу времени, то получим W = T1.

Примеры:

* Для сложения N чисел W = N - 1 (сложений N - 1 штука)
* Для скалярного произведения векторов W = 2N - 1
* Для перемножения матриц W = N2 \* (2N - 1)

**Масштабируемость параллельной программы** определяется **относительно конкретного компьютера** и показывает, как изменяются динамические характеристики (не обязательно только производительность) данной программы при использовании бОльших вычислительных ресурсов.

**Сильная масштабируемость (strong scaling)**  -- зависимость производительности R от количества процессоров p при фиксированной вычислительной сложности задачи (W = const).

(Фиксируем задачу и запускаем её на разном числе процессоров, смотрим, как изменяется производительность при изменении числа процессоров)

**Масштабируемость вширь (wide scaling)** -- зависимость производительности R от вычислительной сложности задачи W при фиксированном числе процессоров (p = const).

(Мы фиксируем систему, на которой решаем задачу, и начинаем изменять вычислительную сложность, т.е. масштаб решаемой задачи, и смотрим, как меняется производительность этой программы при изменении выч.сложности. Часто при увеличении выч.сложности получаем рост производительности, хотя это может зависеть от конкретной задачи)

**Слабая масштабируемость (weak scaling)** -- зависимость производительность R от количества процессоров p при фиксированной вычислительной сложности задачи в пересчёте на один процессор (W / p = const).

(Фиксируем то, какая часть вычислительной сложности приходится на один процессор. Если увеличиваем число процессоров, то увеличиваем и сложность задачи, решаем более сложную задачу на более мощной системе. И смотрим, как меняется производительность компьютера).

Многие компьютеры, не обладающие сильной масштабируемостью, обладают хотя бы слабой масштабируемостью. Но это не так плохо, многие программы пишутся так, чтобы они обладали хотя бы слабой масштабируемостью, это считается хорошим результатом.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Для того, чтобы численно оценивать масштабируемость, нужна какая-то метрика масштабируемости.

Для многих задач при увеличении вычислительной сложности задачи W (а, следовательно, и времени T1) эффективность распараллеливания E растёт.

Если при одновременном увеличении числа процессоров p и вычислительной сложности W эффективность распараллеливания E остаётся прежней, то **данную задачу на данном компьютере можно считать масштабируемой**.

**Функция изоэффективности (isoefficiency function)**

Определим, в какой степени должна увеличиваться вычислительная сложность задачи W в зависимости от числа процессоров p, чтобы эффективность распараллеливания E оставалась постоянной. Эта характеристика будет показывать масштабируемость конкретной вычислительной системы.

Чем меньше необходимая степень роста W для поддержания нужного уровня эффективности распараллеливания, тем более масштабируемой является система.

Считая, что каждый вычислительный шаг == 1:

E = 1 / (1 + T0 / T1) = 1 / (1 + T0 / W)

W = E / (1 - E) \* T0 = K \* T0, где K = E / (1 - E)

Получившаяся зависимость размера задачи, необходимой для достижения заданной эффективности распараллеливания, от числа процессоров (спряталось в T0) -- **функция изоэффективности**.

Пример:

Каскадная схема, суммирование методом сдваивания (каждый процессор суммирует свою локальную часть массива, далее у каждого процессора по числу, процессоры разбиваются на пары, в каждой паре назначается процессор-сумматор, ему второй посылает своё число, он суммирует пару элементов, одновременно суммируются остальные пары элементов. Итак, с каждым шагом число элементов уменьшается в два раза, число работающих процессоров -- тоже. И в итоге сходимся к одному числу -- сумме.)

Пусть для сложения n чисел используется p процессоров, тогда:

W = T1 ~ n

Tp ~ n / p + 2 log(p) [грубая оценка, пересылка + суммирование оценены двойкой, хотя в реальности это должно быть гораздо большее число, поскольку на пересылку требуется далеко не единичное время]

S = T1 / Tp = n / (n / p + 2 \* log(p)) = p / (1 + 2 \* p \* log(p) / n)

E = S / p = 1 / (1 + 2 \* p \* log(p) / n)

C = p \* T0 = p \* (n / p + 2 \* log(p)) = n + 2 \* p \* log(p)

T0 = p \* Tp - T1 = 2 \* p \* log(p)

W = K \* T0 = 2 \* K \* p \* log(p) = O(p \* log(p))

При увеличении числа процессоров от p до p’ для поддержания постоянной эффективности распараллеливания E необходимо увеличить размер задачи в (p’ \* log(p’)) / (p \* log(p)) раз.

Пусть E = 0.5, тогда K = E / (1 - E) = 1

Пусть p = 16, тогда W = 2 \* K \* p \* log(p) = 128

Пусть p = 64, тогда W = 768

Пусть p = 1024, тогда W = 20480.

## Основные помехи масштабируемости параллельных программ:

* Закон Амдала (последовательные части программы)
* Накладные расходы на коммуникации (латентность, пропускная способность)
* Неравномерность загрузки (load balancing) процессоров

*Часть процессоров загружены, часть процессоров простаивает. Чем больше таких процессоров, тем хуже общая масштабируемость программы.*

* Предел декомпозиции данных

# Архитектура параллельных вычислительных систем

## Классификация Флинна

Базируется на понятии потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором.

Итак, Флинн выделяет 4 класса архитектур:

* **SISD (одиночный поток команд и одиночный поток данных)**

К этому классу относятся 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 скорость обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка.

Сейчас таких компьютеров практически нет, если только совсем простых вычислительных систем.

УУ ⇔ ПР ⇔ ПД

* **SIMD (одиночный поток команд и множественный поток данных)**

В архитектурах подобного рода сохраняется один поток команд, включающий, в отличие от SISD, 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.

Такие машины всё-таки иногда встречаются, но всё ещё редко

УУ ⇔ ПР {⇔ ... ⇔} ПД

* **MISD (множественный поток инструкций и одиночный поток данных)**

Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных.

Таких компьютеров так и не придумали, класс пуст.  
УУ {⇔ ... ⇔} ПР ⇔ ПД

* **MIMD (множественный поток инструкций и множественный поток данных)**

Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединённых в единый комплекс и работающих каждое со своим потоком команд и данных.

Практически все современные большие компьютеры попадают в этот класс.

УУ {⇔ ... ⇔} ПР {⇔ ... ⇔} ПД

Есть много других классификаций архитектур, но мы их обсуждать не будем.

**Компьютеры с общей памятью** (**SMP-компьютеры**: либо Shared Memory Processors, либо Symmetric MultiProcessor)

В SMP-компьютерах всё, кроме процессоров, в одном экземпляре: образ ОС, память, подсистема ввода-вывода…

* относительная простота параллельного программирования, т.к не возникает понятия локальной памяти, вся память общая
* сложность увеличения числа процессоров (роста производительности), т.е их сложно масштабировать, обычно <1024 процессоров

**Компьютеры с распределённой памятью**

Компьютеры с распределённой 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.

* сложность параллельного программирования, нужно общение через коммуникационную среду, технологии доступа в разные локальные памяти
* относительная простота увеличения числа процессоров (роста производительности)

## Две задачи параллельных вычислений

1. построение вычислительных систем с максимальной производительностью (это компы с распределённой памятью)
2. эффективное программирование параллельных вычислительных систем (это компы с общей памятью)

## Почему компьютеры с распределённой памятью программировать сложнее, чем компьютеры с общей памятью?

Плохая идея засунуть все данные в память одного процессора -- остальные не смогут работать, пока им не будет что-то послано, неэффективно.

Всё распихать по разным процессорам случайно -- тоже не очень. Может получиться, что много процессоров не смогут ничего со своими данными сделать (потому что не нужно, а им, например, придётся взаимодействовать между собой)

Можно ещё в память каждого процессора скопировать все данные. Тогда зависимостей между процессорами нет, и каждый может работать. Но для каких=то задач это может быть неуместным..

* Распределение вычислений обязательно должно быть согласовано с распределением данных**!**
* 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

* Для **эффективного программирования компьютеров с распределённой 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

# Компьютеры с общей памятью

Иногда говорят, что классические SMP-компьютеры обладают архитектурой UMA (Uniform Memory Access), обеспечивая одинаковый доступ любого процессора к любому модулю памяти.

## Как сохранить единое адресное пространство и одновременно сохранить возможность для увеличения числа процессоров?

Например, отказаться от однородности доступа к памяти, перейти от архитектуры SMP к архитектуре NUMA (Non Uniform Memory Access)

Сохраняется общее адресное пространство, но доступ к памяти неоднородный, поскольку физически память является распределённой.

## Архитектура NUMA

(Конец 70 годов 20 века)

Контроллер памяти по старшим разрядам определяет, в каком модуле хранятся нужные данные. Запрос выставляется либо на локальную шину, либо на межкластерную шину.

**Межкластерная шина** -- узкое место данной архитектуры.

Она с достаточно хорошей пропускной способностью, но всё равно, если навешивается оч много кластеров и если они активно общаются, то всё оч грустно, и даже непредсказуемо, всё зависит от нагрузки.

Примеры компов с NUMA:

* **Cm\*** [схема в лекции]
* **BBN** Butterfly (Бабочка)

Тут межкластерную шину заменили на коммутатор Butterfly [схема в лекции], через который и осуществлялась связь с удалёнными блоками данных

Либо обращение к локальной памяти, либо обращается к чужой памяти через 2 переключателя (можем попасть в память любого блока)

Максимальная конфигурация -- 512 процессоров

Локальные ссылки -- около 2 мкс, удалённые -- около 6 мкс

## Архитектура ccNUMA

В процессорах первых NUMA-компьютеров кэш-памяти не было, а потом все процессоры стали оснащаться блоками кэш-памяти. Появилась **проблема когерентности кэшей (cache coherence problem)**: данные, которые зачитываются в локальный кэш, могут быть устаревшими. Например, если процессор P1 сохранил значение x в ячейке q, а затем процессор p2 читает значение ячейки q, то если z не вытеснено из кэш-памяти в оперативную память, то он получит старое значение (тк обращается к оперативе, а туда новые данные по x ещё не подгрузились).

Для решения этой проблемы появилась модификация NUMA-архитектуры -- **ccNUMA (cache coherent Non Uniform Memory Access)**

Проблема когерентности кэшей решается на аппаратном уровне.

По такой архитектуре построены почти все современные компьютеры с общим адресным пространством.

**Но насколько неоднородна архитектура NUMA?**

Если обращение к памяти другого узла требует на 5-10% больше времени, чем обращение к своей памяти, то к такой системе в большинстве случаев можно относиться, как к UMA (SMP).

Но чаще всего разница составляет 200-700% и это существенно влияет на производительность. И тогда программисту нужно думать о том, где и как разлеглись его данные по блокам локальной памяти, чтобы более часто используемое хранить ближе.

## Пример компьютера с ccNUMA

**Hewlett-Packard Superdome**

* Выпускается с 2000 г. Крайне успешный -- в 2001 году Top500 занимал 147 позиций (столько компов этой линейки были в Top500)
* В стандартной комплектации объединяет от 2 до 64 процессоров с возможностью расширения системы
* До 256 Гбайт оперативной памяти
* Используемые процессоры:
* PA-8600, PA-8700, PA-8900
* Intel IA/64: Itanium, Itanium 2

**В вычислительной ячейке (cell)**:

* [Схема в лекции]
* До 4 процессоров
* До 16 Гбайт оперативной памяти, разделённой на 2 банка
* Контроллер ячейки (24 миллиона транзисторов) реализует интерфейсные функции, когерентность кэшей
* Канал процессор ⇔ контроллер -- 2 Гбайт/сек
* Канал банк памяти ⇔ контроллер -- 2 Гбайт/сек
* Канал контроллер ⇔ внешний коммутатор -- 8 Гбайт/сек

**Базовая конфигурация** (то, что строится на основе вычислительных ячеек):

* [Схема в лекции]
* 2 стойки
* В каждой стойке -- 2 восьмипортовых коммутатора
* Все порты коммутаторов работают со скоростью 9 Гбайт/сек
* К каждому коммутатору подключено 4 ячейки
* 3 других порта для связи с коммутаторами
* 8-ой порт для формирования многоузловой конфигурации

**Виды задержек при обращении в память**, являющихся своего рода платой за высокую масштабируемость системы в целом:

* процессор и память располагаются в одной ячейке -- 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, причём 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 только в 1.6 раза. Это очень немного, поэтому компьютер считался вполне успешным.

**Конфигурации**

Компьютер HP Superdome может быть классическим единым компьютером с общей памятью. Однако его можно сконфигурировать и таким образом, что он будет являться совокупностью независимых разделов (nPartitions), работающих под различными операционными системами, в частности, под HP UX, Linux и Windows. Также предусмотрены организация эффективной работы с большим числом внешних устройств, возможности горячей замены всех основных компонент аппаратуры, резервирование, мониторинг базовых параметров.

**Производительность**

Процессор PA-8700 имеет тактовую частоту 750 MГц, может выполнять 4 операции за такт, что даёт пиковую производительность 3 Гфлопс на процессор. Следовательно, пиковая производительность базовой 64-процессорной конфигурации -- 192 Гфлопс.

## Причины снижения производительности компьютеров с общей памятью

(Указанные ниже “работают всегда, за ними всегда надо следить”)

* **Закон Амдала**

Если в программе 20% всех операций должны выполняться строго последовательно, то ускорения больше 5 получить нельзя вне зависимости от числа используемых процессоров.

При использовании моделей параллельных программ с общей памятью возникают дополнительные участки последовательного кода, связанные с синхронизацией доступа к общим данным, например, критические секции. Реально эти фрагменты будут последовательными участками кода.

* **ccNUMA (неоднородность доступа к памяти)** -- нужно думать, как раскладывать данные
* **ccNUMA (необходимость согласования кэш-памяти)** -- программисту думать не надо, но надо понимать, что это реализовано аппаратно, на это такты тратятся и из-за этого снижается производительность
* **Конфликты при обращении в память**
* **Сбалансированность вычислительной нагрузки процессоров** -- нужно загружать процессоры примерно равномерно, чтобы не было простаивающих процессоров, иначе снижается производительность

В случае систем с общей памятью ситуация упрощается тем, что практически всегда системы являются однородными, поэтому о сложной стратегии распределения работы речь, как правило, не идёт

В случае распределённой памяти может быть проблематично поделить равномерно область вычислений между процессорами (например, треугольная матрица, выколотые точки и т.п.)

* **Производительность отдельных процессоров**
* ...

Причин много, все они в той или иной мере проявляются в **каждой** программе. Их нужно ясно себе представлять, чтобы выбрать правильный способ решения.

Более распространены компьютеры с распределённой памятью, особенно если говорить о суперкомпьютерах.

# Компьютеры с распределённой памятью

## Чем компьютеры с распределённой памятью друг от друга отличаются?

* Используемые процессоры
* Коммуникационная сеть

Примеры

* Intel Paragon: Intel i860, двумерная прямоугольная решётка
* IBM SP1/SP2: IBM Power, высокоскоростной коммутатор каждый-с-каждым
* Ломоносов : Intel Xeon + NVIDIA Tesla, QDR Infiniband, топология “толстое дерево”

## Топологии

**Топология** -- конфигурация графа сети. Определяет взаимное расположение узлов и каналов связи.

В архитектуру помимо топологии входят также **алгоритмы маршрутизации (routing)** и **управления потоками (flow control)**.

**Примеры коммуникационных топологий**

* **линейка** -- для системы из p узлов нужно p-1 соединение, средняя длина пути между двумя узлами равна p/3

**--**  На практике используется редко, невыгодно

* **кольцо** -- для построения системы из p узлов, нужно p соединений, средняя длина пути между двумя узлами равна p/6
* увеличивается отказоустойчивость за счёт того, что передача сообщений может идти по двумя направлениям
* **звезда** -- общение только через центральный узел, p-1 соединение, длина пути между концевыми узлами равна 2

**--** большая нагрузка на центральный узел

* **двумерная решётка, двумерный тор, 3-мерные решётки и торы, 4-мерные решётки и торы, и т.д.** -- многомерные аналоги линейки и кольца (тор от решётки отличается тем, что соединены ещё и противоположные узлы -- узлы, находящиеся на противоположных гранях)

Начиная с 3-мерных -- часто используются на практике, обладают хорошими характеристиками

* **полносвязная топология** -- каждый узел имеет связь с каждым другим узлом, для соединения p узлов требуется p(p-1)/2 связей

**--** дорого, встречается редко

* **двоичный гиперкуб** -- в n-мерном пространстве поместим p = 2^n узлов системы в вершины единичного n-мерного куба, каждый узел соединим с соседом вдоль каждого из n измерений -- всего log(p) соединений, максимальное расстояние между вершинами n-мерного гиперкуба равно n

Достаточно хорошая, но на практике не слишком распространена

**Самые распространённые на практике топологии**

* **дерево, толстое дерево**
* **сети Клоза**
* **бабочки (k-ary n-flies), flattened butterfly (сейчас очень распространена, например, в Ломоносов-2)**
* **dragonfly (тоже очень распространена)**

[Схемы в лекции]

## Параметры коммуникационных сетей

**Длина критического пути (диаметр)** -- минимальное количество элементарных связей, которые нужно пройти для коммуникации двух самых удалённых процессоров

**Связность** -- количество элементарных связей, которые нужно удалить, чтобы схема распалась на две несвязные части

**Сложность** -- общее количество необходимых элементарных связей

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Схема коммутации** | **Длина критического пути** | **Связность** | **Сложность** | **Комментарий** |
| линейка | p - 1 | 1 | p - 1 |  |
| кольцо | округление вниз  (p / 2) | 2 | p |  |
| звезда | 2 | 1 | p - 1 | но большая нагрузка на центральный узел |
| двумерная решётка | 2 (sqrt(p) - 1) | 2 | 2 (p - sqrt(p)) | норм характеристики |
| двумерный тор | 2 округление вниз (sqrt(p) / 2) | 4 | 2p | чаще используются 3,5,6-мерные |
| полносвязная топология | 1 | p - 1 | p (p - 1) / 2 |  |
| двоичный гиперкуб | log(p) | log(p) | p log(p) / 2 | хорошие взвешенные характеристики, используются на практике либо сами, либо как части более сложных топологий |

* **Количество портов маршрутизаторов (radix)**: low-radix/high-radix
* **Масштабируемость** -- насколько большие системы можно строить с использованием этих коммуникационных сетей
* **Расширяемость**
* **Простота эксплуатации**
* **Устойчивость к отказам**
* **Многообразие путей** -- насколько разными путями могут идти данные от узла до узла
* …

## Примеры компьютеров с распределённой памятью

**Семейство Cray XT**

**Семейство**: T2D, T3E, XT3, XT4, XT5, XT6, XE6

* **Cray T3D** -- начало 90 годов
* **Cray XT5** -- на ноябрь 2010 г. второе место в Top500

**Используемые процессоры**

* T3D/T3e -- **DEC ALPHA**
* XT3/XT4/XT5/XT6 -- **AMD Opteron** (остался и по сей день)
* XE6 -- добавили **NVIDIA Tesla** в качестве ускорителя

**Коммуникационная сеть**

* **Топология** -- трёхмерный тор (ОТЛИЧИТЕЛЬНАЯ ОСОБЕННОСТЬ ЭТОЙ ЛИНЕЙКИ)
* отказоустойчивость, возможность выбора альтернативного маршрута для обхода повреждённых связей
* сокращение расстояния между узлами
* У каждого узла всегда 6 **непосредственных соседей**
* **Между узлами** -- два однонаправленных канала передачи данных, что допускает одновременный обмен данными в противоположных направлениях
* **Маршрутизация между вершинами A и B**:

(X - вправо, Y - вперёд, Z - вверх)

Сначала смещение по измерению X до тех пор, пока координата очередного транзитного узла и вершины B по измерению X не станут равными. Затем аналогичная процедура по измерению Y, а в конце -- по Z. Смещение может быть как положительным, так и отрицательным, минимизируя число перемещений по сети и обходя повреждённые связи.

Параллельный транзит данных по каждому из измерений X, Y, Z может выполняться одновременно.

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

**Узлы суперкомпьютера**

* **пользовательские**: компиляция, командные файлы, однопроцессоные задачи
* **операционной системы**: выполнение многих системных сервисных функций ОС, в частности, работа с файловой системой
* **вычислительные**: выполнение программ пользователя в монопольном режиме

Реальные конфигурации Cray T3E: 24/16/576 или 7/5/260 (пользовательские узлы/ОС/вычислительные)

**Вычислительные узлы**

процессор, локальная память, … (в начальных конфигурациях было по 2 процессора на вычислительный узел, далее -- по одному)

+ сетевой интерфейс (он же -- контроллер прямого асинхронного доступа в память)

Любой процессор через свой сетевой интерфейс может обращаться к памяти любого другого процессора, не прерывая его работы

**Аппаратная поддержка барьерной синхронизации**

**Барьер** -- точка в программе, при достижении которой каждый процесс должен ждать, пока остальные процессы также не дойдут до барьера. После этого все процессы могут продолжать работу дальше.

Несколько входных и выходных регистров синхронизации. Каждый разряд этих регистров соединён со своей независимой цепью реализации барьера. Двоичное дерево на основе двух типов устройств: логическое умножение (&) и дублирование входа на два выхода (1) [схема в лекции]

Эта система впервые использовалась в Cray XT, оказалась очень удобной и используется во многих современных компьютерах.

## Вычислительные кластеры

**Вычислительный кластер** -- это совокупность компьютеров, объединённых в рамках некоторой коммуникационной сети для решения одной задачи

* Доступные процессорные платформы (грубо говоря, с рынка. Тупо покупают готовые, а не разрабатывают специально)
* Стандартные сетевые технологии (т.е. снова ничего такого не разрабатывается)
* Свободное ПО: Linux, MPI, GNU, …
* Невысокая стоимость (из-за первых трёх пунктов так получается)

474 компьютера из Top500 мира -- кластеры

Возможность формирования архитектуры кластера с учётом характерных особенностей решаемых задач.

**Вычислительный узел**: небольшое количество процессоров (например, 4), необходимый объём оперативной памяти (варьируется в зависимости от нужд задачи), сетевой адаптер, жёсткий диск (необязательно). И вычислительные узлы объединяются некоторой коммуникационной сетью.

## Основные параметры коммуникационной сети кластеров

**Латентность** -- время начальной задержки при посылке сообщений (время с момента посылки сообщения до его ухода в коммуникационную сеть).

**Пропускная способность** сети определяется скоростью передачи информации по каналам связи и измеряется объёмом передаваемой информации в единицу времени (биты, байты,... в секунду).

**Время на передачу сообщения по коммуникационной сети** вычисляется по следующей формуле:

tN = t0 + N / S,

где t0 -- латентность, N - объём передаваемых данных, S - пропускная способность сети

* Если в программе много маленьких сообщений, то сильно скажется латентность
* Если сообщения в программе передаются большими порциями, то важна высокая пропускная способность каналов связи
* Наличие латентности определяет и тот факт, что максимальная скорость коммуникаций не может быть достигнута на сообщениях с небольшой длиной.

На практике пользователям не столь важны заявляемые производителем пиковые характеристики, сколько реальные показатели, достигаемые на уровне приложений, например, из программ, использующих технологию MPI.

**Измерение латентности на практике**

* процесс p1 замеряет время t1 (начальный момент времени)
* процесс p1 посылает сообщение длины 0 процессу p2 (это не значит, что это 0 байт, с сообщением всегда идёт какая-то служебная информация)
* процесс p2 принимает сообщение от процесса p1 и сразу посылает сообщение длины 0 обратно процессу p1
* процесс p1 принимает сообщение от процесса p2
* процесс p1 замеряет время t2
* под латентностью понимают величину t0 = (t2 - t1) / 2 (не та латентность, что выше)

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

Важна также **стоимость коммуникационной сети**, которая складывается из:

* количества и стоимости маршрутизаторов
* длины и типа кабелей

## Причины снижения производительности компьютеров с распределённой памятью

* **Закон Амдала** (чем больше процессоров, тем меньшее ускорение можем получить)
* **Латентность передачи по сети**
* **Пропускная способность каналов передачи данных**
* **Особенности использования SMP-узлов**
* **Балансировка вычислительной нагрузки** (более важный момент, чем для компьютеров с общей памятью)
* **Возможность асинхронного счёта и передачи данных** (особенность Cray, когда один узел мог получить данных от другого узла, не прерывая его счёта. Если такого нет, то это по сути аналог последовательных операций)
* **Особенности топологии коммуникационной сети**
* **Производительность отдельных процессоров**
* ...

### Лекция 6

# Суперкомпьютерный комплекс МГУ имени М.В.Ломоносова

## История появления суперкомпьютеров в комплексе МГУ

* **Стрела** [1956] 2000 оп/сек
* **БЭСМ-6** [1968] 1 млн.оп/сек
* **SCI** [2000] 12 Гфлоп/сек
* **Чебышёв** [2007] 60 Тфлоп/сек
* **Ломоносов** [2009] 1.7 Пфлоп/сек
* **Ломоносов-2** [2014] 5.5 Пфлоп/сек

## Какие суперкомпьютеры в комплексе МГУ сейчас

* Ломоносов-2 -- 5.5 Пфлоп/сек
* Ломоносов -- 1.7 Пфлоп/сек (вывели из эксплуатации)
* Чебышёв -- 60 Тфлоп/сек (вывели из эксплуатации)
* Polus -- 56 Тфлоп/сек
* IBM Blue Gene/P -- 27 Тфлоп/сек

## СКИФ МГУ Чебышёв

**Общая информация**

* 1 место в 8-ой редакции списка Топ50 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 740000х740000
* 625 вычислительных узлов, 1250 процессоров, 5000 процессорных ядер
* 42 стойки: 14 вычислительных стоек, 28 инфраструктурных стоек
* Помещение 98 м2
* 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

**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 (толстое дерево)
* SilverStorm 9120 – базовые коммутаторы
* Flextronix F-X430046 – листовые коммутаторы
* Характеристики
* 1.3 – 1.95 µs латентность
* 1.7 ГБ/с пропускная способность

[Схема топологии толстое дерево в лекции]

**Чем отличается толстое дерево от обычного дерева?**

1. Чем выше по уровню забираешься, тем толще связи (на картинке между коммутаторами 1ого и 2ого уровней толстые линки, по 2 обычных порта используется вместо одного).

Для чего это? Чем выше уровень коммутатора, тем потенциально большее число вычислительных узлов может через них общаться и тем больше требования к пропускной способности каналов.

1. Нет единого корня в отличие от обычного дерева. Это позволяет разводить маршруты передачи сообщений (в случае Чебышёва как минимум 6 вариантов всегда есть). Коммутаторы второго уровня связаны со всеми коммутаторами 1ого уровня.

**Вспомогательные сети**

* Gigabit Ethernet: коммутаторы Force10 C300 и Force10 S2410
* Управляющая сеть (две) -- ServNet + IPMI

**Хранилище данных**

* 60 ТБ распределённое отказоустойчивое сетевое хранилище T-Platforms ReadyStorage ActiveScale Cluster
* 15 ТБ локальных дисков на узлах
* Ленточное хранилище Quantum Scalar i500 (для бэкапов в основном)

**Система охлаждения**

* 8 кондиционеров APC InfraStruXure ACR502, уровень резервирования N+2
* Холодильные машины Liebert-Hiross SLH 023, одновременно работают 2 из3

**Горячий коридор**

Строится коридор, отделяющий часть помещения, только в эту часть помещения выдувается горячий воздух, нагретый вычислительными узлами, и только этот объём воздуха из горячего коридора прогоняется через систему охлаждения.

* Меньший объём охлаждаемой части помещения
* Более тесная компоновка
* Встречные воздушные потоки

[Схема компоновки системы и горячего коридора в отдельности в лекции]

Здесь один в отличие от Ломоносова, там несколько

**Система пожаротушения**

* Возможность ручного отключения всего комплекса
* Инертный газ
* 3 месяца тестирования на ложные срабатывания
* При входе в помещение автоматическая система отключается

**Электрическое оборудование**

* 1-ый и 4-ый ряды стоек
* PDU: APC AP9565
* UPS: APC Symmetra PX
* Мониторинг: ISX Manager
* Уровень резервирования N+1 (т.е. запасная охладительная машина, например, или другие компоненты суперкомпьютера)

## Суперкомпьютер Ломоносов

**Основная информация**

* **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** -- 2 130
* **Число процессорных ядер (x86)** -- 52 168
* **Число процессорных ядер (ГПУ)** -- 954 240
* **Оперативная память** -- 92 ТБайт
* **Коммуникационная сеть** -- QDR Infiniband / 10 GE (топология толстое дерево, но сложнее, чем в Чебышёве)
* **Система хранения данных** -- 1.75 ПБайт, Lustre, NFS, …
* **Операционная система** -- Clustrx T-Platforms Edition
* **Занимаемая площадь (вычислитель)** -- 252 м2
* **Энергопотребление (вычислитель)** -- 2.8 МВт

[Логическая схема Ломоносова в лекции]

[Схема построения толстого дерева в суперкомпьютере Ломоносов в лекции, вычислительные узлы не нарисованы, должны быть где-то там внизу]

**Система охлаждения**

Горячих коридоров несколько в отличие от Чебышёва (5 штук)

Воздушные кондиционеры засасывают воздух из горячих коридоров и производят теплообмен. Как?

Горячий воздух ⇔ вода (хранится на инженерном уровне, на этаж ниже, в больших баллонах)  
Эту воду надо охлаждать, на улице стоят чиллеры, но напрямую воду туда выводить неправильно (вода в трубах зимой может замёрзнуть, а трубы полопаться). Поэтому есть ещё один уровень теплообмена: вода ⇔ этиленгликоль (аналог незамерзайки из машин, температура замерзания гораздо ниже наших зимних морозов, его уже можно выводить в трубах на улицу)

Этиленгликоль ⇔ чиллеры

Итого три контура охлаждения.

Всего в системе 10 т этиленгликоля и 40 т воды.

Одна из холодильных машин на улице находится в резерве -- иногда подключают летом, если большая жара.

**В случае перебоя в работе электроэнергии** есть 10 минут, система понимает, что что-то не так, посылает сигналы на каждую стойку, за 10 минут все вычислительные узлы корректно завершают свою работу, выгружаются, комп выключается.

**Вес оборудования машзала** - 57т

**Вес системы бесперебойного электропитания (СБЭ)** -- 92 т

**Длина кабелей** -- больше 80 км

**Ломоносов в Top500 самых мощных суперкомпьютеров мира**

|  |  |  |
| --- | --- | --- |
| **Дата** | **Производительность** | **Место** |
| Ноябрь 2009 | 0.4 Пфлоп/сек | 12 |
| Ноябрь 2018 | 1.7 Пфлоп/сек | 485 |
| Июнь 2020 | 1.7 Пфлоп/сек | * (остановили) |

## Суперкомпьютер Ломоносов-2

1 стойка = 256 узлов

7 стоек = 5.5 Пфлоп/сек

intel Xeon + NVIDIA K40 + NVIDIA P100 + NVIDIAV100 (последнее добавилось этим летом)

**Top50 СНГ -- 2 место**

**Общая информация**

* **Пиковая производительность** -- 5 505 TFlop/s
* **Производительность (Linpack)** -- 2 478 TFlop/s
* **Эффективность** -- 50%
* **Вычислительных узлов** -- 1 722
* **Центральное процессоры** -- Intel Haswell-EP E5-2697v3, 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 (топология -- flatten butterfly)
* **Система хранения данных** -- 1392 ТБайт, Lustre, Panasas
* **Операционная система** -- CentOS 7

**Про коммуникационную сеть**

* **Тип связи** -- FDR InfiniBand (56 Gbit/s)
* **Топология** -- Flattened Butterfly
* **Число портов коммутатора** -- 36
* **Размерность** -- 8x8x8x4
* **Относительная ширина связей** -- 1x1x1x2
* **Число вычислительных узлов, подключенных к одному коммутатору** -- 8
* **Максимальное количество узлов** -- 16 384

[Схема flatten butterfly в лекции]

**Ломоносов-2 в Top500 самых мощных суперкомпьютеров мира**

|  |  |  |
| --- | --- | --- |
| **Дата** | **Производительность** | **Место** |
| Ноябрь 2014 | 2.6 Пфлоп/сек | 22 |
| Июнь 2020 | 4.9 Пфлоп/сек | 130 |

**Типы задач**

* Химики
* Биологи
* Физики
* ...

### Лекция 7

# Векторно-конвейерные компьютеры

Могут быть как компьютерами с общей, так и с распределённой памятью.

Век их популярности прошёл, таких компьютеров сейчас достаточно мало. Но элементы векторности постепенно проникают во все процессоры.

**Скалярный** -- работающий с одним элементом

**Функциональное устройство:**

* скалярное
* конвейерное (операция делится на несколько микроопераций, количество микроопераций определяет число ступеней конвейера)

**Команда:**

* скалярная (все аргументы - скаляры);
* векторная (например, сложить все элементы массива x с числом b)

**Компьютер:**

* скалярный;
* векторный (основная работа идёт с векторами. Векторностью может управлять и пользователь);
* конвейерный

Процесс поиска подходящих фрагментов в программе и их замена

векторными командами -- **векторизация программы**.

**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 фрагмент программы удалось заменить векторными

командами, то говорят о его **полной векторизации**. В противном

случае мы имеем дело с частичной векторизацией или

невозможностью векторизации фрагмента вовсе.

**Истоки векторности**:

* линейная организация памяти,
* использование массивов.

**Условия векторизации**

* наличие векторов-аргументов;
* над всеми элементами векторов должны выполняться одинаковые, независимые операции, для которых существуют аналогичные векторные команды в системе команд компьютера.

**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;
* диагональ квадратной матрицы;
* многомерный массив целиком.

**Не вектор** - поддиагональная часть двумерной матрицы**!!!**

**В векторизуемом фрагменте могут использоваться и простые**

**переменные.**

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

Основные кандидаты для векторизации - **самые внутренние циклы**

всех циклических конструкций программы.

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

**Необходимо отсутствие информационной зависимости между**

**операциями!**

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

Неизвестно, какая операция соответствует вызову функции f, и,

следовательно, на какие векторные команды можно заменить

данный фрагмент. Вообще говоря, этот кусочек векторизовать нельза.

**Как понять можно ли векторизовать (если есть возможность производить межпроцедурный анализ)?**

* inline-подстановка (на место f подставляется операция, которую она реализует, а затем выполняется анализ, можно ли производить векторизацию)
* бывает более сложный межпроцедурный анализ, в который включаются все вызываемые функции

## Векторно-конвейерный компьютер Cray C90

[Схема в лекции]

**Cray C90 -- 1991г**

**Многопроцессорный** (до 16 процессоров) компьютер **над общей памятью**

Структура одного из процессоров (справа на картинке):

1. Векторные операции проводятся над данными, которые загружаются в векторные регистры.
2. Есть 4 группы функциональных устройств (ФУ):

* **векторные ФУ** (выполняют векторные операции над целыми числами)
* **ФУ обработки вещественных чисел** (работают как в векторном, так и в скалярном режиме)
* **скалярные ФУ**
* **адресные ФУ** (для вычисления адресов)

1. В векторный регистр данные загружаются непосредственно из оперативной памяти

В скалярные (S) и адресные(A) регистры данные могут подгружаться как из оперативной памяти, так и из промежуточных регистров (T-регистры и B-регистры -- некоторый аналог кэш-памяти)

1. Векторные команды могут выполняться как над полными векторами, так и над элементами векторов. Для обозначения элементов вектора, которые участвуют в операции, существует **регистр маски вектора** -- битовый регистр (из 0 и 1). Участвуют те элементы в позициях которых в этом регистре единичка.
2. В векторных операциях могут участвовать векторы разной длины. Соответствующая длина векторной операции хранится в **регистре длины вектора**.

## Функциональные устройства

* Все функциональные устройства **конвейерные**.
* **Число ступеней различно,** однако **каждая ступень каждого устройства** всегда **срабатывает за один такт**.
* Все функциональные устройства **независимы и могут работать одновременно друг с другом**.
* Функциональные устройства данного компьютера делятся на четыре группы:
* **адресные** – операции над 32-разрядными целыми числами;
* **скалярные** – операции над 64-разрядными целыми числами;
* **для векторных команд** – операции над целочисленными векторами;
* **функциональные устройства для выполнения вещественной арифметики** – операции над 64-разрядными числами в форме с плавающей запятой (как в векторном, так и в скалярном режиме).

## Время разгона конвейера и секционирование векторных команд

Время разгона конвейера влияет на производительность и особенно сказывается на коротких векторах (но само от длины вектора не зависит, это характеристика конвейера)

|  |  |
| --- | --- |
| **Длина вектора** | **Производительность Cray C90 на операции a[i] = b[i] \* s + c[i]** |
| 1 | 7.0 |
| 4 | 27.6 |
| 32 | 181.9 |
| 128 | 433.7 |
| 129 | 364.3 |
| 256 | 548.0 |
| 257 | 491.0 |
| 8192 | 802.0 |

Секционирование векторных команд также влияет на производительность. Тут длина векторных регистров == 128 элементам. Поэтому если длина вектора > 128, то вектор делится на части и подгружаются сначала первые 128 элементов, над ними производится операция, а потом нам нужно ещё время на подгрузку следующих 128 элементов, и т.д. Поэтому на границе 128 (256, ...) элементов у нас производительнсть проседает. Получаем частично растущую функцию. В целом получили, что чем больше длина вектора, тем больше производительность, но на границах, кратных 128 элементам, будет немного проседать производительность.

## Дублирование и зацепление векторных устройств

В этом компьютере все конвейеры, которые выполняют векторные операции, продублированы. То есть если мы делаем какую-то векторную операцию сложения, то она будет выполняться одновременно на двух конвейерах. Например, чётные элементы отправляются на один конвейер, а нечётные -- на второй, т.е производительность увеличивается автоматически в 2 раза.

Дублируются векторные устройства и устройства для вещественной арифметики.

**Зацепление векторных устройств** -- режим, когда выход одного конвейера поступает сразу на вход другого конвейера.

**Зацепление векторных операций**

Пусть:

* операция “х” выполняется за l1 ступеней
* операция “+” выполняется за l2 ступеней

Без зацепления: (l1 + n-1) + (l2 + n-1) = l1 + l2 + 2n - 2 тактов

С зацеплением: l1 + l2 + n-1 тактов

Если n -- большое, то получим ускорение примерно в 2 раза.

За счёт чего? С какого-то момента эти два конвейера работают одновременно, выполняя единый конвейер.

## Конфликты в памяти

Характерны для самых разных компьютеров. Показываем на Cray C90, поскольку знаем, как устроена память.

В Cray C90 память устроена иерархически. [схема в лекции: 8 секций, каждая делится на 8 подсекция, каждая подсекция делится на 16 банков]

Доступ к каждой секции выполняется независимо, к разным -- может осуществляться одновременно.

**Когда возникает проблема?**

* При одновременном обращении к одной и той же секции возникает конфликт, который разрешается за 1 такт. В этом случае один из запросов продолжает обрабатываться, а другой просто блокируется на один такт.
* Если происходит одновременное обращение к одной и той же подсекции одной секции, то время на разрешение конфликта уже может достигать 6 тактов.

Данные стараются распределить так, чтобы элементы, к которым обращаются наиболее часто подряд, лежали в максимально разных секциях и разных подсекциях. Т.е если идём по какому-то массиву подряд, элементы хорошо распределить так, чтобы они последовательно находились в максимально разных секциях и подсекциях.

**Последовательные адреса идут с чередованием в Cray C90:**

адрес 0 — в 0-й секции, 0-й подсекции, 0-м банке;

адрес 1 — в 1-й секции, 0-й подсекции, 0-м банке;

адрес 2 — в 2-й секции, 0-й подсекции, 0-м банке;

...

адрес 8 — в 0-й секции, 1-й подсекции, 0-м банке;

адрес 9 — в 1-й секции, 1-й подсекции, 0-м банке;

...

адрес 63 — в 7-й секции, 7-й подсекции, 0-м банке;

адрес 64 — в 0-й секции, 0-й подсекции, 1-м банке;

адрес 65 — в 1-й секции, 0-й подсекции, 1-м банке;

...

Т.е. сначала сменяется номер секции, потом подсекции, потом банка.

Максимальное число конфликтов будет происходить при постоянном обращении к одной и той же подсекции одной и той же секции. Это заведомо произойдет, например, при выполнении процессором векторной операции чтения данных, расположенных с шагом, кратным 64

**При этом:**

* Операции чтения/записи последовательно расположенных данных проходят без возникновения конфликтов. В частности, все одномерные массивы будут обрабатываться именно так.
* При выборке данных с любым нечетным шагом конфликтов также не возникнет.
* Чем б**о**льшая степень двойки присутствует в качестве сомножителя в шаге выборки данных, тем больше времени требуется на разрешение возникающих конфликтов

**Как конфликты сказываются на производительности?**

for(i = 0; i < n\*k; i +=k)

a[i] = b[i]+c[i]\*d;

Специально идём с шагом k

|  |  |
| --- | --- |
| **Шаг по памяти** | **Производительность Cray C90 на операции a[i] = b[i] + c[i] \* d, n = 1000** |
| 1 | 705.2 |
| 2 | 444.6 |
| 4 | 274.6 |
| 8 | 142.8 |
| 16 | 84.5 |
| 32 | 44.3 |
| 64 | 22.7 |
| 128 | 22.6 |

Результаты соответствуют написанному выше, конфликты в памяти очень сильно влияют на производительность.

## Каналы процессор-память

Данные между процессором и памятью передаются по трём каналам передачи данных, два из которых могут работать на чтение из памяти, а третий на запись.

|  |  |
| --- | --- |
| **Длина вектора** | **Производительность Cray C90 на операции a[i] = b[i] \* c[i] + d[i]** |
| 10 | 57.0 |
| 100 | 278.3 |
| 1000 | 435.3 |
| 12801 | 445.0 |

Получаем не оч много производительности (~50% от пиковой), потому что не хватает каналов -- не можем прокачать 3 вектора одновременно, только 2 канала.

Таким образом, количество каналов тоже сказывается на производительности.

## Операции чтения/записи в векторные регистры, ограниченное число векторных регистров

Раскрутка цикла:

[глубина раскрутки == 1]

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

Вектор p[i][j-1] подкачивается лишний раз, но уже есть в регистрах, устраним это, будем идти с шагом 2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[глубина раскрутки == 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

Зависимость производительности Cray C90 от глубины раскрутки

|  |  |
| --- | --- |
| **Глубина раскрутки** | **Производительность Cray C90** |
| 1 | 612.9 |
| 2 | 731.6 |
| 3 | 780.7 |
| 4 | 807.7 |

В какой-то момент мы поймём, что регистров не хватает (т.к с увеличение глубины раскрутки требуется всё больше регистров), производительность сначала перестанет расти, потом будет уменьшаться.

Но в целом техника хорошая -- экономим на числе операций чтения/записи.

## Несбалансированное использование устройств, отсутствие операции деления

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Длина вектора** | **Производительность Cray C90 на различных операциях** | | | |
| **a[i] = b[i] + c[i]** | **a[i] = b[i] / c[i]** | **a[i] = s / b[i] + t** | **a[i] = s / b[i] \* t** |
| 10 | 35.5 | 24.8 | 49.7 | 46.1 |
| 100 | 202.9 | 88.4 | 197.4 | 166.5 |
| 1000 | 343.8 | 117.2 | 283.8 | 215.9 |

Вместо деления берут обратное к c и умножают на b → 2 операции вместо одной

В общем, все эти вещи неплохо так влияют на производительность Cray C90

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Предсказывалось, что векторные компьютеры устареют к 1990 году.
* Основная причина – быстрый прогресс в производительности микропроцессоров. Векторные инструкции в микропроцессорах не использовались.
* Что ещё привело к такому положению дел?

1. Сокращение государственных расходов США на суперкомпьютеры (после 1989 года).
2. Распространение микропроцессоров Intel и AMD -> большие объёмы продаж -> много средств на исследование и разработку.

В настоящее время векторных компьютеров мало, зато процессоры это используют.

* Возвращение векторности по причине эволюции микропроцессоров и появления в них векторных регистров.
* MMX Instruction set (1997) – 8 64-битных регистров, только операции с целыми числами.
* SSE (Streaming SIMD Extension, 1999) – 8 128-битных регистров, операции с плавающей точкой. Получила развитие в виде SSE2, SSE3 и.т.д. (добавление новых инструкций).
* AVX (2008) – 16 256-битных регистров, инструкции с 3 операндами: a = a + b -> c = a + b.
* AVX2 (2013) – имеются специальные инструкции для целых чисел. Добавлены инструции для операций вида a = b + c \* d.
* AVX-512 – 32 512-битных регистра. Поддерживается в Xeon Phi KNC, KNL и Intel Skylake-X.
* AltiVec / Velocity Engine / VMX – в своё время прямой конкурент SSE.

1. Регистры можно загружать и сохранять только в память.
2. Больший (по сравнению с SSE) набор допустимых типов данных и операций.
3. 32 128-битных регистра, инструкции с тремя операндами.

* ARM SVE – расширение для набора инструкций А64.

1. Длина вектора от 128 до 2048 бит, кратна 128. Предпочитаемой длины вектора нет, всё зависит от конкретного «железа».
2. Приложения подстраиваются под имеющийся объём.
3. Не нужно ничего перекомпилировать / переписывать.
4. Ориентация на борьбу с проблемами автовекторизации (современные компиляторы часто не в состоянии провести векторизацию из-за зависимостей внутри вектора).
5. Предназначен для решения научных задач с помощью HPC.

## NEC SX-Aurora TSUBASA

[Система векторных и SIMD операций → NEC Vector -- сочетание векторных и SIMD операций]

**Ядро** -- Vector Engine (VE) в виде карты PCI, включает в себя 8 ядер, состоящих из скалярного и векторного блоков.

**Векторная часть:**

В каждом ядре содержится по 64 векторных регистра длиной по 256 64-разрядных элементов. Суммарная ёмкость регистров составляет 128 Кбайт.

Над векторными регистрами выполняются операции с помощью трёх устройств FMA (Fused Multiply Add) -- выполняют операцию a \* b + c над векторными регистрами, каждый такт 32 результата. Суммарно 32x3x2=192 флоп/такт

При тактовой частоте 1.6 ГГц получаем 307.2 Гфлоп/cек на ядро или 2.4576 Тфлоп/сек 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-Aurora, используют в качестве коммуникационной сети Infiniband.

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

Диапазон платформ TSUBASA от рабочих станций A100 до стоечных суперкомпьютеров A500 с пиковой производительностью 157.3 TFlop/s

## Основные особенности архитектуры, дающие заметный вклад в ускорение выполнения программ

* Конвейеризация выполнения команд.
* Независимость функциональных устройств.
* Векторная обработка.
* Дублирование конвейеров векторных устройств и устройств для вещественной арифметики. (конкретно для Cray C90)
* Зацепление функциональных устройств.
* Многопроцессорная обработка.

## Что снижает производительность векторно-конвейерных компьютеров?

* Закон Амдала (если есть последовательные части в программе, хорошего ускорения мы на ней не получим)
* Время разгона конвейера (особенно влияет для коротких векторов)
* Секционирование векторных команд (связано с длиной векторных регистров)
* Конфликты в памяти (зависит от устройства оперативной памяти, но всё равно чем чаще в один банк памяти попадём, тем больше просядет производительность)
* Каналы процессор-память
* Операции чтения/записи в векторные регистры (то, что на это тоже тратится время)
* Ограниченное число векторных регистров
* Несбалансированное использование устройств (если часть простаивает, хорошей производительности не будет)
* Отсутствие операции деления (конкретно для Cray C90, деление -- 2 операции)
* Перезагрузка буферов команд (если ходим по программе с большим шагом, подкачек может понадобиться много → проседает производительность)
* ...

Эти факторы все работают одновременно → если большинство не убрать, то будет грустно.

# Распределённые вычислительные среды

Когда из компьютеров распределённых физически по всему земному шару, строят , так называемый, мета-компьютер для решения какой-то задачи.

## Чем отличаются вычислительные среды от классических компьютеров с распределённой памятью?

* Эти системы могут объединять в себя колоссальные ресурсы (можно объединить хоть все системы, соединённые интернетом, если они могут подключиться к системе)
* Распределённость и удалённость (ваще где угодно могут находиться части, требуется время на пересылку данных)
* Динамичность конфигурации (к ним в любой момент могут присоединиться/от них могут отсоединиться в любой момент вычислительные узлы)
* Неоднородность конфигурации (самые разные компьютеры объединяются, с разными ОС, разной степенью загрузки и т.д.)
* Различная административная принадлежность (части находятся в разных странах, с разной политикой, какие-то части доступны только по ночам, какие-то в свободное время, какие-то постоянно, где-то можно решать какие-то задачи, где-то нет)

## Примеры

* **Система метакомпьютинга X-COM** (разрабатывалась у нас в МГУ)

Решалась **задача проектирования новых лекарств**:

НИВЦ МГУ, Гематологический центр РАМН

**Время расчета**: 31 дек. 2005 – 11 янв. 2006,

**Процессорное время**:42774 часа = 1782 дня = > 4.5 года работы одного процессора

**Общее число работавших процессоров**: 273,

**Максимально процессоров одновременно**: 244,

**Пиковая производительность**: >1 Тфлопс

**Вычислительная среда**: кластеры МГУ + кластер ЮУрГУ (г.Челябинск) + учебный класс НИВЦ МГУ + компьютеры в лабораториях

## Использование вычислительных сред

Свойства вычислительных сред должны учитываться при

* выборе тех задач, которые будут решаться с их помощью
* формировании структуры процесса вычислений
* программировании вычислительных сред
* выполнении распределённых программ

Смежное направление - **проведение вычислений по требованию (on-demand computing).** Для многих организаций экономически невыгодно держать у себя высокопроизводительные компьютеры, поскольку потребности в проведении больших расчетов возникают лишь эпизодически, а стоимость приобретения и сопровождения такого оборудования велика.

Другой популярный сейчас термин **облачные вычисления (cloud computing)** - это технология распределённой 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, игровых приставках, ноутбуках, смартфонах и т.д.

**Многопоточность (мультитредовость, multithreading)** — свойство платформы или приложения, состоящее в том, что процесс, порождённый 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.

Очень часто сейчас используется

**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 определённых 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).

Пользователь не должен писать как параллельную программу, распараллеливанием занимается либо аппаратура, либо ПО (обычно компилятор).

**Выгоды параллелизма на уровне машинных команд**:

* отсутствие у пользователя необходимости в специальном параллельном программировании;
* переносимость.

**Суперскалярная архитектура** - архитектура, использующая несколько дешифраторов команд, которые могут нагружать работой множество исполнительных блоков. Если в процессе работы команды, обрабатываемые конвейером, не противоречат друг другу, и одна не зависит от результата другой, то такое устройство может распараллелить выполнение команд.

Суперскалярность не предполагает, что программа в терминах машинных команд будет включать в себя какую-либо информацию о содержащемся в ней параллелизме. **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 ни природу аппаратного обеспечения, на котором он будет реализован, ни точного временного порядка, в котором будут выполняться команды

Успех зависит от того, насколько хорошо аппаратура (именно аппаратура!) умеет отыскивать эти операции. Много где применяется.

**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. Команда VLIW-процессора состоит из набора полей, каждое из которых отвечает за свою операцию. Если какая-то часть процессора на данном этапе не востребована, то соответствующее поле команды не задействуется.

В процессорах VLIW задача распределения решается во время компиляции и в инструкциях явно указано, какое вычислительное устройство должно выполнять какую 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, какие регистры какие операнды будут содержать и т.д.

Успех зависит от того, насколько хорошо написан компилятор, насколько хорошо умеет планировать программу в форме этих больших командных слов.

**EPIC (Explicitly 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 архитектуре EPIC используется концепция длинного командного слова и добавляются следующие черты:

* Спекулятивная загрузка данных - вынесение команд загрузки данных далеко вперёд инструкций, использующих эти данные. Это позволяет лучше использовать иерархию памяти. Если произведена запись, затрагивающая загружаемое содержимое, то производится вызов кода восстановления, перезагружающего значение.
* 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, если количество команд в ветвях условного оператора невелико; уменьшить нагрузку на устройство предсказания переходов.

## Принципы построения высокопроизводительных компьютерных систем

1. Выбирается процессорная основа (на базе каких процессоров будет строиться система)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

FPGAs GP-GPU Cell … Векторные Многоядерные Классические

|  |  |  |
| --- | --- | --- |
| Эффективность (% от пика)  Производительность на ватт  Производительность на объём | ⇐ … ⇒ | Простота освоения  Переносимость программ Скорость программирования |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. На основе выбранных процессоров строим узлы с общей памятью (SMP)
2. Выбираем количество узлов, которое нам необходимо, объединяем некоторой коммуникационной сетью, получаем компьютер с распределённой памятью
3. Если и эта производительность не устроила, то строим распределённые вычислительные среды

### Лекция 8

# Методы оценки производительности

**Понятие производительности не до конца честное**.

Например

Вычислительное ядро: A = B + C

Производительность на Cray T90: X Мфлопс

Пользователь: Хочу больше, чем X

Специалист: Без проблем ⇒ A = B + C \* 1 ⇒ Производительность возросла почти в 2 раза (из-за дополнительной операции), но время-то осталось неизменным (или чуть-чуть совсем увеличится), обманули :(

**Какой компьютер выбрать?**

В идеале было бы компьютеру однозначно сопоставить некое число.

**Пиковая производительность**: вычисляется просто и однозначно, но нет связи с реальной задачей пользователя. Даёт нижнюю оценку времени выполнения программы.

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

**Синтетические (или искусственные) тесты** не имеют отношения к реальным приложениям; предназначены для создания стрессовой нагрузки на отдельные подсистемы компьютера.

**Реальные тесты** выполняют реальные задачи над реальными данными.

**Основные требования к тестам производительности**:

* **Непротиворечивость и понятность результатов** -- понятно, что тест выдаёт
* **Легкость в использовании** -- легко взять, развернуть, запустить, получить результат
* **Масштабируемость** -- можно прогонять на компьютерах разных размеров, от обычных ПК до суперкомпьютеров
* **Переносимость** -- чтобы можно было запускать на разных ОС, в разных средах, с разными библиотеками
* **Репрезентативность** -- чтобы тест из себя что-то представлял, не был какой-то абстрактностью из непонятно какой области, а скорее людям интересно, чтобы тест отвечал какой-то реальной задаче
* **Доступность теста и его исходного кода** -- большинство тестов свободно распространяемы, но не всегда (например, SPEC нужно покупать за деньги, чтобы использовать)
* **Воспроизводимость** -- чтобы не было такого, что запустили один раз, потом ещё один, и результаты совершенно друг другу не соответствуют

**Наиболее известные тесты (бенчмарки)**

* Linpack
* STREAM
* Ливерморские циклы
* Perfect Club Benchmarks
* SPEC
* HINT
* NAS Parallel Benchmarks (NPB)
* HPC Challenge
* Graph500
* HPCG

## Тест Linpack

* Cоздан Джеком Донгаррой (Jack Dongarra) и его коллегами в 1979 году.
* Используется для формирования списков Top500 и Топ50.
* Решение больших систем линейных алгебраических уравнений с плотной квадратной матрицей методом LU-разложения.
* Число операций с плавающей точкой оценивается по формуле 2n^3 /3 + 2n^2 , где n – линейный размер матрицы. Т.е основной порядок здесь -- n^3 операций (ну или ⅔n^3)

Если замерим время выполнения программы, то можем узнать производительность, поделив число операций на время.

* Сначала Linpack 100×100, запрет изменений текста. Но размер матрицы оч быстро стал маленьким, объём оперативной памяти компьютеров рос очень быстро, производительность тоже росла очень быстро, СЛАУ с такими матрицами решались очень быстро и перестали показывать адекватную производительность больших компьютеров. Стали увеличивать размер матрицы.
* Далее - Linpack 1000×1000, стандартная головная часть программы. Ну и такой размер матриц стал очень маленьким. Решили отказаться от фиксированного размера матриц.
* Сейчас – произвольный (максимально возможный, поскольку стараются как можно больше увеличить показатели производительности. В общем стараются задать матрицу размера максимально близкого к объёму оперативной памяти) размер матрицы, возможность вносить любые изменения в текст. На больших машинах тест Linpack может считаться достаточно долго, часами.

Некоторые производители переписывают Linpack специально для своих компьютеров, используя какие-нибудь архитектурные особенности, чтобы тест показывал максимальную производительность.

**High Performance Linpack (HPL)**

Одна из свободных реализаций теста Linpack

* <http://www.netlib.org/benchmark/hpl/>
* Наиболее популярная реализация теста Linpack на языке Си.
* Обмены между процессорами выполняются через процедуры MPI.
* Для оптимизации вычисления на каждом процессоре основываются на низкоуровневой библиотеке базовых функций линейной алгебры BLAS (Basic Linear Algebra Subprograms) или VSIPL(Vector Signal Image Processing Library).

По сути ассемблерные вставки, которые позволяют быстрее решать СЛАУ, **благодаря ним этот тест показывает хорошие результаты по производительности**, иначе были бы резы как на любой программе, написанной на языке высокого уровня

* 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.

## STREAM

**STREAM (Sustainable Memory Bandwidth in High Performance Computers)**

* <http://www.cs.virginia.edu/stream/>
* Синтетический тест, оценивающий скорость работы с памятью с простой арифметикой и без.
* **STREAM**. Производительность на операциях:
* 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.
* **STREAM2**. Производительность на операциях:
* a(i)=q;
* a(i)=b(i);
* a(i)=a(i)+q\*b(i);
* sum = sum + a(i);
* Размер массивов задаётся достаточным, чтобы выйти за размеры кэш-памяти (чтобы данные не хранились целиком в кэше). Отношение пиковой производительности к скорости передачи данных почти всегда больше 1.Чем отношение больше, тем больше несбалансированность компьютера, тем больше он нацелен на арифметику, передача данных отстаёт.
* Параллельные версии с использованием OpenMP и MPI.

Простота и переносимость тестов вызывает недоверие, появляются более сложные тесты, что вызывает проблемы с переносимостью и т.д. Поэтому создаются наборы тестов.

## Набор тестов NAS Parallel Benchmarks

**Оценка более-менее реальной производительности компьютеров**

* <http://www.nas.nasa.gov/Software/NPB/>
* Набор тестов производительности, разработанных в NASA Advanced Supercomputing (NAS) Division (ранее NASA Numerical Aerodynamic Simulation Program). Используется активно.
* Существуют 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: (вводится, чтобы тест можно было использовать на компьютерах разного масштаба)
* S, W (для тестовых прогонов);
* A, B, C (каждый в среднем в 4 раза больше предыдущего);
* D, E, F (каждый в среднем в 16 раз больше предыдущего)
* Набор бенчмарков состоит из нескольких приложений.
* **5 вычислительных ядер** (взяты из кодов NASA)
* IS (Integer Sort);
* EP (Embarrassingly Parallel);
* CG (Conjugate Gradient);
* MG (MultiGrid);
* FT (Fast Fourier Transform).
* **3 модельных приложения** (написанные целиком и взятые целиком как тесты)
* BT (Block Tri-diagonal solver);
* SP (Scalar Penta-diagonal solver);
* LU (Lower-Upper Gauss-Seidel solver).
* **3 дополнительных теста** (специально написанные дополнительные тесты)
* UA (Unstructured Adaptive mesh);
* DC (Data Cube);
* DT (Data Traffic).

Наборы этих тестов позволяют неплохо приблизиться к тем реальным задачам, которые у людей встречаются, чтобы можно было оценить производительность компьютера на чём-то достаточно похожем.

## Набор тестов HPC Challenge

Постарались включить как можно больше известных бенчмарков, протестировать компьютер со всех сторон.

* <http://www.hpcchallenge.org/>
* Включает в себя 7 тестов:
* HPL (Linpack);
* DGEMM (вычисляет производительность перемножения матриц);
* STREAM;
* PTRANS (транспонирование матрицы) -- нет вычислительных операций, только перегонка данных;
* RandomAccess (вычисляет скорость случайных обращений к памяти);
* FFTE (реализация одномерного дискретного преобразования Фурье);
* Communication bandwidth and latency (скорость передачи данных и латентность) -- тест на коммуникационную сеть

## 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, варианты 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.

## HPCG

* <http://hpcg-benchmark.org/>
* High Performance Conjugate Gradient – решение системы линейных алгебраических уравнений с разрежённой квадратной положительно определённой симметричной матрицей (такие матрицы хранятся в специальном сжатом виде без нулевых элементов)
* Оценивает не только скорость самих вычислений, но и нерегулярных обращений в память.
* Разработчики: Jack Dongarra, Michael Heroux, Piotr Luszczek (разработчики Linpack, по сути тест -- его модификация)
* Анонсирован в 2013 году
* На данный момент последняя версия теста 3.1.

На этом тесте производительность ощутимо меньше, чем на Linpack, и совсем не всегда лучши комп на Linpack будет лучшим на HPCG. Считается, что этот тест ближе к реальным задачам, чем Linpack.

## Необходимость комплексного тестирования программно-аппаратной 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.

Хорошая производительность достигается тогда. когда на всех этих уровнях удаётся найти достаточно эффективное решение.

# Технологии параллельного программирования

## Модели параллельного программирования

**Модели параллельного программирования**

* **SPMD (Single Program, Multiple Data, единая программа, множество данных)** – на всех процессорах выполняется одна и та же программа над своим множеством входных данных. Как это осуществляется, зависит от конкретной технологии параллельного программирования. В большинство технологий либо предусматривают такую модель параллельного программирования, либо для некоторых технологий это единственная возможная модель параллельного программирования. Самая популярная модель.
* **Master-Workers или Master-Slave (мастер-рабочие)** – один из процессов (называемый главным, master) получает входные данные задачи, разделяет их на части и передаёт другим процессам (рабочим, workers) для обработки, а затем получает результаты и проводит финальную обработку. По-разному может быть реализовано, сам мастер может также служить обработчиком, т.е. рабочим.

В общем, такая модель предполагает всё-таки, что есть какой-то выделенный процесс, особый, который выполняет сервисные функции, а вычислительные функции выполняют все остальные, либо все остальные и мастер тоже.

* **Message passing (модель передачи сообщений).** Соответствует в первую очередь технологии MPI, но и в других технологиях тоже используется. Программа порождает несколько задач (процессов) 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. Т.е. для общения не требуется пересылка сообщений, они могут напрямую по адресу обратиться к любому элементу данных. Требуется разграничение доступа к общим данным.

Как правило, привязана к компьютерам с общей памятью или к тем системам, в которых общая память как-то эмулируется.

**Критерии выбора подходящей технологии программирования**

1. **Возможность создания эффективных программ** (если какая-то технология на конкретном компьютере используется неэффективно, то её использовать скорее всего не будут)
2. **Возможность быстрого создания параллельных программ (продуктивность программирования)**. Наиболее эффективная программа на любом компьютере -- что-то типа ассемблера, но в реальности большие серьёзные программы на ассемблере писать очень сложно и редко это делается, тем более если речь о параллельных программах.
3. **Переносимость программ, гарантии сохранения эффектевности**

Т.е. не просто переносимость с одной платформы на другую, но и чтобы в рамках одной и той же модели программирования сохранялась некоторая гарантия эффективности реализации конкретной пользовательской программы

1. **Стоимость** (некоторые распространяются бесплатно, а некоторые нужно покупать, входят, например, в состав компиляторов, которые нужно покупать, или продаются отдельно)

## Программирование систем с общей и распределённой памятью, что нужно учитывать обязательно?

**Компьютеры с общей памятью:**

* параллелизм вычислений
* синхронизация доступа к общим данным.

**Компьютеры с распределённой памятью:**

* параллелизм вычислений;
* распределение данных;
* согласование параллелизма вычислений и распределения данных;
* организация пересылок данных

## Иерархия различных технологий параллельного программирования

**0. Распараллеливающий компилятор**

Когда стали появляться первые параллельные компьютеры, появилась такая идея.

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

Для пользователя идея замечательная, но оказалось, что компиляторы с этой ролью плохо справляются, их структура становится очень сложной, в каких-то простых случаях справляется хорошо, в случаях посложнее -- уже не очень. Но полностью от этой идеи не отказались, сейчас компиляторы всё-таки что-то такое сами умеют делать (иногда).

**1. Спецкомментарии**

Это комментарии специального вида в программе, которые помогают компилятору в той или иной степени.

* В чистом виде - **компиляторы компьютеров CRAY**

Пример команды: CDIR$ NODEPCHK. Означает no dependence check, ставился перед циклами, чтобы показать, что этот цикл не содержит информационных зависимостей и его можно распараллеливать. Компилятор, видя комментарий не проверял это, верил пользователю и разбрасывал итерации цикла по вычислительным устройствам, выполняя его в параллельном режиме.

Не все компиляторы могли понять этот комментарий, но те, кто могут, понимают, что это указания для них.

В наше время идея продолжает использоваться в некоторых технологиях параллельного программирования, хотя, может, и не в столь чистом виде:

* **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 (систем, которые используют как ЦП, так и ускорители), задействующих как центральный, так и графический процессор. Последняя версия стандарта – OpenACC 2.7. https://www.openacc.org

**2. Расширения существующих языков программирования**

Берётся некоторый существующий (последовательный) язык программирования и к нему добавляются некоторые конструкции. Поскольку это некоторое расширение ЯП, то переписывается компилятор, чтобы он мог воспринимать новые конструкции и умел реализовывать на соответствующем компьютере. Таких расширений ЯП достаточно много.

* **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. Большая нагрузка легла на компилятор, он усложнялся и усложнялся, создать переносимый компилятор для разных систем оказалось очень сложно.

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 по этой топологии и необходимые пересылки данных. Посылка сообщений организована с использованием интерфейса MPI. Информация, извлечённая из описания параллельного алгоритма, вместе с данными о реальной производительности процессоров и коммуникационных каналов, помогает системе программирования mpC найти эффективный способ отображения процессов 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-препроцессор переводит программу в Си (Фортран), расширенный функциями системы поддержки;
* для организации межпроцессного взаимодействия может использоваться одна из коммуникационных технологий (MPI, PVM, Router), что обеспечивает хорошую переносимость;
* директивы DVM: распределение данных, распределение вычислений, спецификация удалённых данных, редукционные операции.
* **CAF (Co-Array Fortran):**
* небольшое расширение языка Fortran, достаточное для разработки эффективных параллельных программ;
* модель SPMD;
* понятие co-array (распределённого массива), компоненты которого распределены по процессам; доступ к распределённым компонентам осуществляется по правилам работы с обычными массивами.
* **UPC (Unified Parallel C):**
* расширение языка программирования Си;
* модель SPMD;
* модель PGAS (Partitioned Global Address Space, глобальное разделённое 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** (http://opencl.org/ ) - расширения Си-подобных языков для программирования графических ускорителей.
* **Linda** - параллельный язык программирования. Берётся ЯП, к нему добавляются 4 новые операции и тогда на этом языке можно реализовать практически любые схемы параллельных вычислений.

Программа рассматривается как совокупность процессов, которые могут обмениваться данным 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 (по сути формулами);
* 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).
* Другие языки параллельного программирования: **Chapel**, **X10**, **Fortress**…

**4. Библиотеки и интерфейсы, поддерживающие взаимодействие параллельных процессов**

* **MPI (Message Passing Interface)** - хорошо стандартизованный механизм для построения программ по модели обмена сообщениями. Существуют стандартные «привязки» MPI к языкам С 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 своего 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». Пересылка сообщений не двусторонняя (один посылающий и один принимающий процесс, и оба должны активно использоваться в пересылке данных), а активен один процесс (либо посылающий, либо принимающий), т.е. либо один процесс посылает сообщение и вписывает его в область памяти принимающего процесса, принимающий только должен заранее разрешить это возможность, а активно он в этой операции не участвует, поэтому может не прерывать свою работу. Либо принимающий процесс активен и забирает сообщение из области памяти посылающего процесса, не прерывая его работу. Сначала эта идея появилась в Shmem, потом перекочевала в MPI.

**5. Средства и технологии для поддержки метакомпьютерных и распределённых вычислений**

Если нужна производительность больше производительности одного компьютера.

* 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.
* Библиотеки есть и в других областях, не только такие. Далеко не всегда нужно писать что-то с нуля, можно использовать и уже написанное :)

**7. Специализированные пакеты и программные комплексы**

При использовании таких пакетов пользователь не пишет программы, его задача правильно сконфигурировать начальные данные для этого пакета

* **FlowVision**, **ANSYS** для инженерный расчётов;
* **GAMESS**, **GAUSSIAN**, **PRIRODA**, **GROMACS**, **CHARMM** для квантовохимических расчётов;
* **OpenFOAM** для вычислительной гидродинамики;

### Лекция 9

# Технология программирования OpenMP

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

Один вариант программы для параллельного и последовательного выполнения. Значительная часть функциональности реализует идеологию, когда можно написать последовательную программу и из неё сделать параллельную на OpenMP путём добавления специальных комментариев.

Любой процесс состоит из нескольких нитей управления, которые имеют общее адресное пространство, но разные потоки команд и раздельные стеки. Нити -- основной элемент OpenMP. Поскольку у нитей нет общего адресного пространства, то их относительно легко, дёшево и быстро порождать и уничтожать. Поэтому в OpenMP есть идеология, что нити порождаются по необходимости, и когда необходимость отпадает, то они уничтожаются, т.е. нити существуют только в параллельных частях программы, когда они необходимы. Но всё равно этим увлекаться не надо, т.к. всё-таки есть некоторые накладные расходы, связанные с порождением и уничтожением нити.

## Как понять, поддерживает ли конкретный компилятор технологию OpenMP

Макрос **\_OPENMP** определён в формате yyyymm, где yyyy и mm – цифры года и месяца, когда был принят поддерживаемый стандарт OpenMP.

Условная компиляция:

**#include <stdio.h>**

**int main() {**

**#ifdef \_OPENMP**

**printf("OpenMP is supported!\n");**

**#endif**

**}**

Если хотим проверить, что поддерживается последняя версия, то нужно сравнивать значение макроса со значением 201811 (ноябрь 2018 года)

## Распараллеливание в OpenMP

Распараллеливание в OpenMP -- вставка в текст программы специальных директив, а также вызов вспомогательных функций.

**SPMD-модель (Single Program Multiple Data)** параллельного программирования -- для всех параллельных нитей используется один и тот же код (но каждая нить выполняет его над своими данными).

## Структура программы

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

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

Любое количество параллельных и последовательных областей. Параллельные области могут быть вложенными друг в друга.

## Переменные в параллельных областях

В OpenMP переменные в параллельных областях программы разделяются на два основных класса:

* **shared** (**общие**; все нити видят одну и ту же переменную -- если одна нить переменную изменяет, то значение изменяется и для всех нитей);
* **private** (**локальные**, приватные; каждая нить видит свой экземпляр данной переменной).

По умолчанию, все переменные, порождённые вне параллельной области, при входе в неё остаются общими. Исключение составляют переменные, являющиеся счётчиками итераций в цикле. Переменные, порождённые внутри параллельной области, по умолчанию являются локальными.

## Задание директив OpenMP

Директивы OpenMP в языке Си задаются указаниями препроцессору, начинающимися с **#pragma omp**, (опции могут разделяться запятыми, а могут и пробелами, как кому удобнее)

**#pragma omp directive-name**

**[опция[[,] опция]...]**

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

## Использование функций OpenMP

Чтобы задействовать **функции библиотеки OpenMP периода выполнения** (исполняющей среды), в программу нужно включить заголовочный файл **omp.h** (для программ на языке Фортран – файл **omp\_lib.h** или модуль **omp\_lib**).

Нужно задать количество нитей, выполняющих параллельные области программы, определив значение переменной среды **OMP\_NUM\_THREADS**:

**export OMP\_NUM\_THREADS=n**

(это **ЗАДАНИЕ КОЛИЧЕСТВА НИТЕЙ ПО УМОЛЧАНИЮ ПЕРЕД ЗАПУСКОМ ПРОГРАММЫ**)

## Функции для работы с системным таймером

* **double omp\_get\_wtime(void);**

Возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом. Гарантируется, что за время работы программы это время в прошлом не изменится. Результат работы функции не используется, используется разница между двумя замерами времени. Например, если хотим узнать, сколько времени работает участок программы, то вставляем вызов этой функции до этого участка, после этого участка, вычисляем разницу и получаем нужное время в секундах. Возвращаемое значение -- double, точности должно хватать, но вообще точность определяется точностью самого таймера. В реальных система таймеры, которые используются в OpenMP имеют точность порядка 10^(-5), 10^(-6) секунд обычно

* **double omp\_get\_wtick(void);**

Возвращает в вызвавшей нити разрешение таймера в секундах, т.е. то, с какой максимальной точностью может работать таймер, т.е. время между двумя тиками этого таймера, с бОльшей точностью таймер работать не сможет.

## Директива задания параллельной области

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

Порождаются новые **OMP\_NUM\_THREADS-1** нитей, каждая нить получает свой уникальный номер, причём порождающая нить получает номер 0 и становится основной нитью группы («мастером»).

Все нити начинают выполнять один и тот же код (оператор или блок, который идёт непосредственно после этой директивы).

При выходе из параллельной области производится неявная синхронизация (т.е. все нити ждут, пока все из них не завершат выполнение своего кода) и уничтожаются все нити, кроме породившей.

## Опции директивы задания параллельной области

* **if(условие)** – выполнение параллельной области по условию; Если условие истинно, то нити порождаются и код будет выполняться в параллельном режиме. Если условие ложно, то нити порождаться не будут и код будет выполнен одной нитью -- нитью мастером -- так, как она выполнялась бы и в последовательном режиме.

Может быть полезно, чтобы, например, не распараллеливать работу, если у нас слишком мало операций в коде, чтобы распараллеливать их между нитями

* **num\_threads (целочисленное выражение)** – явное задание количества нитей, которые будут выполнять параллельную область;

Даже если было задано значение **OMP\_NUM\_THREADS**, то эта директива имеет бОльший приоритет, именно она задаст количество порождаемых нитей

* **default(shared|none)** – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс **shared**;

**none** – всем переменным класс назначается явно, это запретит использовать переменные, которым класс явно не назначен;

Тоже удобная вещь. Если написать **default none**, то компилятор заставит явно написать классы всех переменных, которые будут в этой области использоваться, тогда не забудется это сделать :)

* **private(список)** – переменные, для которых порождается локальная копия в каждой нити;

Начальное значение не определено, даже если в последовательной области, в нити-мастере, она была чем-то определена;

* **firstprivate(список)** – переменные, для которых порождается локальная копия в каждой нити;

Локальные копии инициализируются значениями этих переменных в нити-мастере;

Как **private**, но есть инициализация значениями этих переменных из нити-мастера. Но на это может быть уйдёт чуть большее время, чтобы эти копирования все выполнить

* **shared(список)** – переменные, общие для всех нитей;

У этих переменных одна копия, и если одна нить меняет её значение, то оно будет изменено и для всех остальных нитей

* **reduction(оператор:список)** – позволяет после выполнения параллельной области выполнить “редукционную операцию”, задаёт оператор и список общих переменных, над которыми этот оператор будет выполняться;

Для каждой переменной создаются локальные копии в каждой нити (они становятся типа **private**); 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 Си – **+, \* , -, &, |, ^ , &&, ||**;

порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску (в каком порядке переменные будут суммироваться или что-то другое), т.к. это может приводить к различным ошибкам округления.

**Пример 1**

**#include <stdio.h>**

**int main(int argc, char \*argv[]) {**

//этот принт выполняется нитью-мастером, один раз

**printf("Последовательная область 1\n");**

**#pragma omp parallel {**

//этот принт будет выполнен столько раз, сколько нитей порождается для параллельной области. Явно не выписывается, значит, зависит от значение переменной OMP\_NUM\_THREADS.

**printf("Параллельная область\n");**

**}**

//этот принт тоже выполняется один раз, нитью-мастером

**printf("Последовательная область 2\n");**

**}**

**Пример 2**

**#include <stdio.h>**

**int main(int argc, char \*argv[]) {**

**int count = 0;**

**#pragma omp parallel reduction (+: count) {**

//оператор + аддитивен → count инициализируется нулём в каждой из порождённых нитей, затем в каждой нити значение count увеличивается на единичку следующей строчкой

**count++;**

//для каждой нити распечатывается единичка

**printf("Текущее значение count: %d\n", count);**

**}**

//параллельная область заканчивается и все единички суммируются в reduction (по сути получаем число нитей, порождённых для параллельной области), а потом единственной существующей теперь нитью-мастером печатается вычисленное значение count

**printf("Число нитей: %d\n", count);**

**}**

## Сокращение кода

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**.

## Порождение нитей в последующих параллельных секциях

**void omp\_set\_num\_threads(int num);** -- позволяет задать число нитей в параллельных областях программы из самой программы, имеет бОльший приоритет чем задача переменной среды **OMP\_NUM\_THREADS**, но меньший, чем **num\_threads(n)** перед любой последующей параллельной областью

**Пример**

**#include <omp.h>**

**#include <stdio.h>**

**int main(int argc, char \*argv[]) {**

**omp\_set\_num\_threads(2);** //определяем число нитей значением 2

**#pragma omp parallel num\_threads(3) {** //в этой параллельной области число нитей задаётся значением 3, перекрывается предыдущее заданное значение

//этих принтов будет 3 штуки

**printf("Параллельная область 1\n");**

**}**

**#pragma omp parallel {**

//а этих принтов будет 2, поскольку работает директива для всех параллельных областей

**printf("Параллельная область 2\n");**

**}**

**}**

## Вложенные параллельные области

* По умолчанию вложенная параллельная область выполняется одной нитью. Это управляется установкой переменной среды **OMP\_NESTED**. Если значение OMP\_NESTED установлено в false, то вложенные параллельные области будут игнорироваться и выполняться каждой отдельной нитью основной параллельной области, как любой другой код этой параллельной области

**export OMP\_NESTED=true** //это означает, что в параллельной области каждая вложенная параллельная область порождает свои нити

**!!! Так может очень быстро расти количество нитей, за этим надо следить !!!**

* Управлять этим можно и из программы с помощью функции **void omp\_set\_nested(int nested);** -- изменяет значение переменной OMP\_NESTED
* **int omp\_get\_nested(void);**  -- позволяет узнать значение переменной OMP\_NESTED

## Находимся ли мы в параллельной области?

**int omp\_in\_parallel(void);** -- позволяет узнать, находимся мы в параллельном участке программы или нет

Эта функция возвращает 1, если она была вызвана из активной параллельной области программы.

Может быть полезно, если мы хотим понять, надо нам новые нити создавать или нет (если последовательной, то почему нет, запараллелимся, а если из параллельной -- стоит подумать, надо ли оно нам)

**Пример**

**void mode(void){**

**if(omp\_in\_parallel()) printf("Параллельная область\n");**

**else printf("Последовательная область\n");**

**}** //функция, которая пишет “параллельная область”, если была вызвана из параллельной области, и пишет “последовательная область”, если была вызвана из последовательной области

**int main(int argc, char \*argv[]){**

**mode();** //будет напечатано “последовательная область”

**#pragma omp parallel**

**mode();** //будет напечатано “параллельная область”

**}**

## А что, если хотим, чтобы какой-то код выполняла только одна нить?

Если в параллельной области какой-либо участок кода должен быть выполнен лишь один раз, то его нужно выделить директивами **single:**

**#pragma omp single [опция [[,] опция]...]** -- кусок кода (оператор или блок после директивы) выполнит одна нить (какая из всех -- OpenMP не специфицирует, скорее всего это будет первая дошедшая нить, хотя это зависит от реализации), все остальные его просто проигнорируют

**Опции для директивы single:**

* **private(список);** -- с этой опцией уже знакомились
* **firstprivate(список);** -- с этой опцией уже знакомились
* **copyprivate(список)** – после выполнения блока кода некоторой нитью, которая его выполняла (т.е. она там что-то насчитала), значение этих переменных может стать известно всем остальным нитям -- новые значения переменных списка будут скопированы в локальные копии этих переменных всех остальных нитей;

опция не может использоваться совместно с опцией **nowait**;

переменные списка не должны быть перечислены в опциях **private** и **firstprivate** данной директивы **single**;

* **nowait** – после выполнения участка происходит неявная барьерная синхронизация нитей: их дальнейшее выполнение происходит только тогда, когда все достигнут данной точки; если в этом нет необходимости, опция nowait позволяет нитям, дошедшим до конца участка, продолжить выполнение без синхронизации.

**#pragma omp parallel {**

**printf("Сообщение 1\n");** //этих принтов будет столько, сколько породилось нитей для этой параллельной области

**#pragma omp single nowait {**

**printf("Одна нить\n");** //этот принт будет напечатан только один раз конкретной нитью

**}**

**printf("Сообщение 2\n");** //тоже будет напечатан всеми нитями, включая отставшую на выполнение своего блока

**}**

Директива **#pragma omp master** выделяет участок кода, который будет выполнен только нитью-мастером. Остальные нити пропускают данный участок и продолжают работу с оператора, расположенного следом. Неявной синхронизации не предполагает.

**Пример**

**#pragma omp parallel private(n) {**

**n=1;**

**#pragma omp master**

**n=2;** //это будет только у нити-мастера

**printf("значение n: %d\n", n);** //все нити, кроме нити-мастера напечатают единички, нить-мастер напечатает двойку

**}**

## Распределение участков кода по номеру процесса

Все нити в параллельной области нумеруются последовательными целыми числами от **0** до **N-1**, где **N** — количество нитей, выполняющих данную область.

* **int omp\_get\_thread\_num(void);** -- позволяет нити получить свой уникальный номер в текущей параллельной области.

Потом можно написать что-то типа “если номер нити равен 0, то делай такой код, если номер нити равен 1, то делай другой код”, и так далее строить распараллеливание, но это не в стиле OpenMP (это так, замечание, никто не мешает этим пользоваться)

* **int omp\_get\_num\_threads(void);** -- позволяет нити получить количество нитей в текущей параллельной области.

**Пример 1**

**#include <stdio.h>**

**#include <omp.h>**

**int main(int argc, char \*argv[]) {**

**int n=1;** //локальная переменная

**printf("в посл. области (1): %d\n", n);**

**#pragma omp parallel private(n) {**

**printf("Значение n на нити (1): %d\n", n);** //в большинстве нитей, кроме нити-мастера, будет напечатано какое-то мусорное значение, поскольку переменная private

**/\* Присвоим переменной n номер текущей нити \*/ n=omp\_get\_thread\_num();**

**printf("Значение n на нити (2): %d\n", n);** //теперь каждая нить напечатает свой уникальный номер, включая нить-мастера (она напечатает 0)

**}**

**printf("в посл. области (2): %d\n", n);** //тут осталась только нить-мастер, напечатает 0

**}**

**Пример 2 (private поменяли на firstprivate)**

**#include <stdio.h>**

**#include <omp.h>**

**int main(int argc, char \*argv[]) {**

**int n=1;** //локальная переменная

**printf("в посл. области (1): %d\n", n);**

**#pragma omp parallel *first*private(n) {**

**printf("Значение n на нити (1): %d\n", n);** //все нити напечатают единичку, поскольку переменная private

**/\* Присвоим переменной n номер текущей нити \*/ n=omp\_get\_thread\_num();**

**printf("Значение n на нити (2): %d\n", n);** //теперь каждая нить напечатает свой уникальный номер, включая нить-мастера (она напечатает 0)

**}**

**printf("в посл. области (2): %d\n", n);** //тут осталась только нить-мастер, напечатает 0

## Циклы

Если в параллельной области встретился оператор цикла, то, согласно общему правилу, он будет выполнен всеми нитями текущей группы, то есть, каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву **for**.

**#pragma omp for [опция [[,] опция]...]** -- эта директива относится к идущему следом за данной директивой блоку, включающему операторы for.

**Опции директивы** **for:**

* **private(список);** -- ага
* **firstprivate(список);** -- ага
* **lastprivate(список)** – переменным, перечисленным в списке, присваивается результат с последнего витка цикла (последний в последовательном смысле);
* **reduction(оператор:список);** -- ага
* **schedule(type[, chunk])** – опция задаёт, каким образом итерации цикла распределяются между нитями; **type** -- тип распределения, **chunk** (необязательный) -- количество нитей в блоке, если распределение блочное
* **collapse(n)** — опция указывает, что n последовательных тесновложенных циклов ассоциируется с данной директивой (т.е. если хотим распараллелить не один цикл **for**, а целое циклическое гнездо из **n** циклов **for**);

для циклов образуется общее пространство итераций, которое делится между нитями по общей схеме;

если опция collapse не задана, то директива относится только к одному непосредственно следующему за ней циклу;

* **ordered** – опция, говорящая о том, что в цикле могут встречаться директивы ordered (т.е. внутри кода есть последовательные куски);

в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле;

* **nowait**

## Ограничения на вид параллельных циклов

На вид параллельных циклов накладываются достаточно жёсткие ограничения. В частности, предполагается, что:

* корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит
* нельзя использовать побочный выход из параллельного цикла
* размер блока итераций, указанный в опции **schedule**, не должен изменяться в рамках цикла.

**Формат параллельных циклов** упрощённо можно представить следующим образом: **for([целочисленный тип] i = инвариант цикла; i {<,>,=,<=,>=} инвариант цикла; i {+,-}= инвариант цикла)**

Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поскольку у каждой нити должен быть свой счётчик итераций, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции lastprivate.

**Пример**

**#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);** //благодаря этому принту поймём какая итерация в какую нить распределилась

**}**

**}**

**}**

## Типы распределения итераций параллельного цикла

В опции **schedule** параметр **type** задаёт 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. до последней нити, затем распределение снова начинается с нулевой нити. Если значение **chunk** не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между нитями.

**#include <stdio.h>**

**#include <omp.h>**

**int main(int argc, char \*argv[]){**

**int i;**

**#pragma omp parallel private(i)**

**{**

**#pragma omp for schedule (static, 2)**

//тут не знаем, сколько нитей порождается. Если нитей >= 5 все итерации распределятся сразу для выполнения, каким-то, может, не достанется итераций вообще

**for (i=0; i<10; i++){**

**printf("Нить %d выполнила итерацию %d\n",**

**omp\_get\_thread\_num(), i);**

**sleep(1);**

**}**

**}**

**}**

* **dynamic** – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает chunk итераций (по умолчанию **chunk=1**), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из **chunk** итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные. Эта неравномерность в случае больших значений объёма порции может быть существенной ⇒ чтобы это сгладить, придумали следующий тип распределения:
* **guided** – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины **chunk** (по умолчанию **chunk=1**) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения **chunk**. В этом случае неравномерность распределения будет сглажена по сравнению с **dynamic**
* **auto** – способ распределения итераций выбирается компилятором и/или системой выполнения. Параметр **chunk** при этом не задаётся.
* **runtime** – способ распределения итераций выбирается во время работы программы по значению переменной среды **OMP\_SCHEDULE**. Параметр chunk при этом не задаётся.

Значение по умолчанию переменной **OMP\_SCHEDULE** зависит от реализации. Если переменная задана неправильно, то поведение программы при задании опции **runtime** также зависит от реализации.

Задать значение переменной **OMP\_SCHEDULE** в Linux в командной оболочке bash можно при помощи команды следующего вида: **export OMP\_SCHEDULE="dynamic,1"**

**void omp\_set\_schedule(omp\_sched\_t type, int chunk);** -- позволяет изменить значение переменной **OMP\_SCHEDULE** из программы

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;**

**void omp\_get\_schedule(omp\_sched\_t\* type, int\* chunk);** -- позволяет узнать текущее значение переменной **OMP\_SCHEDULE**.

**!!!** При распараллеливании цикла нужно убедиться в том, что итерации данного цикла не имеют информационных зависимостей. В этом случае его итерации можно выполнять в любом порядке, в том числе параллельно. Компилятор это не проверяет. Если дать указание компилятору распараллелить цикл, содержащий зависимости, результат работы может оказаться некорректным. **!!!**

## Независимые секции кода

Директива **#pragma omp sections [опция [[,] опция]...]** определяет набор независимых секций кода, каждая из которых выполняется своей нитью.

* **private(список);**
* **firstprivate(список);**
* **lastprivate(список)** – переменным присваивается результат из последней секции;
* **reduction(оператор:список);**
* **nowait**

Директива **#pragma omp section** задаёт участок кода внутри секции **sections** для выполнения одной нитью.

Перед первым участком кода в блоке **sections** директива **section** не обязательна. Какие нити будут задействованы для выполнения какой секции, не специфицируется. Если количество нитей больше количества секций, то часть нитей для выполнения данного блока секций не будет задействована. Если количество нитей меньше количества секций, то некоторым (или всем) нитям достанется более одной секции.

**Пример 1**

**int n;**

**#pragma omp parallel private(n) {**

**n=omp\_get\_thread\_num();**

**#pragma omp sections {**

**#pragma omp section** //этой директивы могло бы не быть, она необязательная

**printf("Первая секция, процесс %d\n", n);** //опции nowait нет, сначала напечатаются все вот эти принты, а принты параллельной области будут напечатаны строго после секций

**#pragma omp section**

**printf("Вторая секция, процесс %d\n", n);**

**}**

**printf("Параллельная область, процесс %d\n", n);**

**}**

**Пример 2**

**int n=0;**

**#pragma omp parallel {**

**#pragma omp sections lastprivate(n) {**

**#pragma omp section**

**n=1;**

**#pragma omp section**

**n=2;**

**}**

//так как lastprivate, то возьмётся 2 и скопируется во все локальные переменные n всех нитей

**printf("Значение n на нити %d: %d\n", omp\_get\_thread\_num(), n);** //то есть тут все нити напечатают 2

**}**

**printf("Значение n в конце: %d\n", n);** //тут тоже, наверно, 2, т.к. в нулевую нить же тоже скопируется 2

## Независимые задачи

Директива **#pragma omp task [опция [[,] опция]...]** применяется для выделения отдельной независимой задачи.

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. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией.

**Опции директивы task**

* **if(условие)** — порождение новой задачи только при выполнении некоторого условия; если условие не выполняется, то задача (ну по сути задача не создаётся, рассматривается как обычный кусок кода) 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;
* **default(shared|none);**
* **private(список);**
* **firstprivate(список);**
* **shared(список).**

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

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

## Синхронизация

Самый распространенный способ синхронизации в OpenMP – барьер.

Он оформляется с помощью директивы **#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. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые нити завершили все порождённые ими задачи (**task**) (т.е. внутри директивы **barrier** находится ещё и директива **taskwait**)

**Пример**

**#include <stdio.h>**

**#include <omp.h>**

**int main(int argc, char \*argv[]) {**

**#pragma omp parallel {**

**printf("Сообщение 1\n");**

**printf("Сообщение 2\n");**

//все нити сначала напечатают первые два принта, все подождут, а потом будут печатать третий принт ⇒ все тройки напечатаются заведомо после все двоек и единиц, иначе могли бы и перемешаться

**#pragma omp barrier**

**printf("Сообщение 3\n");**

**}**

**}**

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

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. Может использоваться, например, для упорядочения вывода от параллельных нитей.

**#pragma omp parallel private (i, n) {**

**n=omp\_get\_thread\_num();**

**#pragma omp for ordered**

**for (i=0; i<5; i++) {**

**printf("Нить %d, итерация %d\n", n, i);** //эти принты могут быть сделаны в любом порядке, параллельные

**#pragma omp ordered {**

**printf("ordered: Нить %d, итерация %d\n", n, i);** //а вот эти будут строго по возрастанию i

**}**

**}**

**}**

**}**

С помощью директивы **#pragma omp critical [(<имя\_критической\_секции>)]** оформляется критическая секция программы.

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. Если на входе стояло несколько нитей, то случайным образом выбирается одна из них, а остальные продолжают ожидание.

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.

**Пример**

**int n;**

**#pragma omp parallel {**

**#pragma omp critical {**

//если бы не было critical, то, так как n при входе в параллельную часть общая, каждая нить хотела бы присвоить ей своё значение, и в каком порядке это было бы выполнено -- вообще неизвестно. В итоге тоже будет неизвестно какой порядок, но всё чинно-мирно и по очереди выполнится, без толкотни (аналогичный результат получили бы, если бы сделали n локальной переменной при входе в параллельную область, например)

**n=omp\_get\_thread\_num();**

**printf("Нить %d\n", n);**

**}**

**}**

Директива **#pragma omp atomic** 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. На время выполнения оператора блокируется доступ к данной переменной всем запущенным в данный момент нитям, кроме нити, выполняющей операцию. Атомарной является только работа с переменной в левой части оператора присваивания, при этом вычисления в правой части не обязаны быть атомарными.

**#include <stdio.h>**

**#include <omp.h>**

**int main(int argc, char \*argv[]) {**

**int count = 0;**

**#pragma omp parallel {**

**#pragma omp atomic**

//count++ будет выполняться по очереди каждой нитью атомарно (иначе несколько нитей могли бы одновременно пытаться её увеличивать, какой результат бы получился -- непонятно)

**count++;**

**}**

**printf("Число нитей: %d\n", count);**

**}**

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**
* **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.

Есть два типа замков:

* **простые замки**
* **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.

**void omp\_init\_lock(omp\_lock\_t \*lock);**

**void omp\_init\_nest\_lock(omp\_nest\_lock\_t \*lock)** -- для инициализации простого или множественного замка соответственно.

После выполнения функции замок переводится в разблокированное состояние. Для множественного замка коэффициент захваченности устанавливается в ноль.

**void omp\_destroy\_lock(omp\_lock\_t \*lock);**

**void omp\_destroy\_nest\_lock (omp\_nest\_lock\_t \*lock);** -- используются для переведения простого или множественного замка в неинициализированное состояние.

**void omp\_set\_lock(omp\_lock\_t \*lock);**

**void omp\_set\_nest\_lock (omp\_nest\_lock\_t \*lock);** -- функции для захватывания замка

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.

**void omp\_unset\_lock(omp\_lock\_t \*lock);**

**void omp\_unset\_nest\_lock(omp\_lock\_t \*lock);** -- функции для освобождения замка.

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.

**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);**

**int omp\_test\_lock(omp\_lock\_t \*lock);**

**int omp\_test\_nest\_lock(omp\_lock\_t \*lock);**  -- функции для неблокирующей попытки захвата замка.

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.

**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);**

### Лекции 10 и 11

# Технология программирования MPI

## Общие понятия

* MPI - Message Passing Interface, интерфейс передачи сообщений.
* Стандарт MPI 3.1 (4 июня 2015 года). (только для C и Фортрана, раньше был и для C++)
* Более 450 процедур.
* SPMD (Single Program Multiple Data) - основная модель параллельного программирования (но MPI разрешает писать и не для SPMD модели, т.е. можно написать и для каждого процессора свою программу, но так происходит действительно редко. У нас рассматриваются примеры для SPMD)
* Предназначена в основном для компьютеров с распределённой памятью, но может использоваться и на любых других архитектурах
* Префикс MPI\_

**#include “mpi.h”** (**mpif.h** для языка Фортран)

* **Процессы** (понятие, схожее с процессом в UNIX), посылка сообщений.

Процессы со своим адресным пространством в отличие от OpenMP, у них нет никаких общих данных. Всё, что хотим передать кому-то ещё -- через посылку сообщений.

* **Сообщение** – массив однотипных данных, расположенных в последовательных ячейках памяти.
* **Группа** – упорядоченное множество процессов. Это понятие локальное, существует в памяти конкретного процесса, т.е. если мы что-то делаем с группой, то в памяти других процессов ничего не меняется, поэтому все операции с группами быстрые, не требуют межпроцессного взаимодействия. Группы могут быть вложенными, пересекающимися. Любой процесс может входить в любое количество групп.
* Обычно (в большинстве реализаций) программа в MPI запускается сразу на многих процессах, при запуске указывается, сколько процессов приложения мы запускаем. Так в большинстве реализаций, но стандарт говорит, что в принципе можно реализовать так, что процессы будут порождаться только при заходе в параллельную часть и уничтожаться при выходе из неё

## Коммуникаторы

**Коммуникатор** – контекст обмена группы. Это объект глобальный, он существует не в памяти какого-то конкретного процесса. Это нечто объединяющее различные процессы. Чтобы процессы общались между собой, им нужен коммуникатор.

**В операциях обмена используются только коммуникаторы!** Процесс обмена -- долгий, поэтому процессы с коммуникаторами достаточно долго, их нужно по возможности избегать, иначе программа очень сильно замедлится.

Коммуникаторы имеют в языке Си предопределённый тип **MPI\_Comm** (в языке Фортран – тип **INTEGER**).

**Предопределённые коммуникаторы**

**MPI\_COMM\_WORLD** – коммуникатор для всех процессов приложения. (нетривиальный такой)

**MPI\_COMM\_SELF** – коммуникатор, включающий только текущий процесс.

**MPI\_COMM\_NULL** – коммуникатор, не содержащий ни одного процесса

Каждый процесс может одновременно входить в разные коммуникаторы.

**Идентификация процесса**

Два основных атрибута процесса:

* коммуникатор (группа)
* номер процесса в коммуникаторе (группе).

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

**Сообщения**

* Сообщение — набор данных некоторого типа. Вместе с сообщением посылается некоторый набор атрибутов.
* Атрибуты сообщения:
* номер процесса-отправителя
* номер процесса-получателя
* идентификатор сообщения
* коммуникатор
* **Идентификатор сообщения** -- целое неотрицательное число в диапазоне от **0** до **MPI\_TAG\_UB** (не меньше **32767**). Системе всё равно, какое вы там число посылаете, ей важно, чтобы в посылающем и в принимающем процессах этот идентификатор совпал. Если не совпадёт, то пересылка просто не состоится. Т.е. если вам не нужно идентифицировать сообщения, то можно всё время использовать один и тот же идентификатор
* Для работы с атрибутами сообщений введена структура **MPI\_Status**.

## Результат работы функций MPI

Возвращаемым значением (в Фортране – последним аргументом) для большинства процедур **MPI** является информация об успешности завершения. (два исключения -- функции работы с таймером (см. далее))

В случае успешного выполнения процедура вернёт значение **MPI\_SUCCESS**, иначе - код ошибки. Этот код можно проверять и смотреть, что за ошибка произошла. Предопределённые значения, соответствующие различным ошибочным ситуациям, перечислены в файле **mpi.h**

## Общие процедуры MPI

**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.

**int MPI\_Finalize(void)**

Завершение параллельной части приложения. 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.

**Пример** (Простейшая параллельная программа)

**#include <stdio.h>**

**#include "mpi.h"**

**int main(int argc, char \*\*argv) {**

**printf("Before MPI\_INIT\n");** //так как все процессы запущены сразу и с момента запуска программы выполняют один и тот же код, то этот принт будет напечатан всеми процессами

**MPI\_Init(&argc, &argv);**

**printf("Parallel section\n");** //и этот

**MPI\_Finalize();**

**printf("After MPI\_FINALIZE\n");** //и этот

**}**

## Функции, которые могут быть использованы вне параллельной секции

**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. Эти процедуры можно вызвать до **MPI\_Init** и после **MPI\_Finalize**.

Может быть полезно, чтобы проверить, нужно ли нам делать **MPI\_Init** (или **MPI\_Finalized**) он уже был вызван (если дважды его вызвать, то будет ошибка)

## Вспомогательные функции

**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**.

**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;** //раскидали итерации цикла по процессам

**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();**

**}**

**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**.

## Работа с таймером

Исключения из правила, что результат работы функций -- успешность их завешения

**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.

Используем на практике разницу между двумя замерами.

**double MPI\_Wtick(void)**

Возвращает разрешение (точность) таймера в секундах.

**Пример** (Определение 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 приём сообщений с блокировкой

**Для отправляющего процесса**

**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**.

**Блокировка** в том смысле, что посылающий процесс блокируется (приостанавливается) до тех пор, пока сообщение, начиная с buff, куда-то не ушло.

Есть несколько вариантов:

* принимающий процесс готов принять это сообщение. Тогда сообщение напрямую копируется из buff в буфер приёма процесса dest
* принимающий пока не готов принять сообщение. Тогда это сообщение надо пока что скопировать в какой-нибудь буфер (есть специальные внутренние буферы MPI, их размер настраиваемый). Если размер промежуточного буфера позволяет сохранить там сообщение, то сообщение туда копируется, а отправляющий процесс разблокируется и продолжает работу. Если внутренних буферов MPI не хватает, то процесс стоит и ждёт, пока принимающий сможет принять сообщение.

**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**.

Модификации процедуры **MPI\_Send**: (по аргументам не отличаются)

* **MPI\_Bsend** — передача сообщения с буферизацией (пользователь сам, явно в программе заводит специальный буфер для промежуточного копирования посылаемых данных, для этого есть специальная функция).
* процесс может долго не простаивать на пересылке
* придётся ручками что-то там задавать
* копирование в промежуточный буфер может и не понадобиться, а производиться будет (можно и проиграть на этом)
* **MPI\_Ssend** — передача сообщения с синхронизацией (в этом случае никакие внутренние буфера не используются, а процесс стоит и ждёт, пока его сообщение заберут). Скорее для отладки используются, посмотреть до куда дошли процессы.
* **MPI\_Rsend** — передача сообщения по готовности. Почти всегда самый быстрый (никаких лишних копирований, проверок), но самый опасный способ. Отправляется сообщения напрямую в буфер принимающего, не зная, готов он принять или нет. В этом случае пользователь должен гарантировать, что принимающий готов. Если вдруг будет не готов, то будет ошибка

**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**. Ни для каких других целей этот буфер использовать нельзя.

**int MPI\_Buffer\_detach(buf, size)**

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

**Для принимающего процесса**

**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**.

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.

Пока нужное сообщение не появится, принимающий процесс будет стоять и ждать его.

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.

**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)**

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.

## Тупиковые ситуации (deadlock)

|  |  |  |
| --- | --- | --- |
| **процесс 0:**  **Recv(1)**  **Send(1)** | **процесс 1:**  **Recv(0)**  **Send(0)** | Процесс 0 доходит до блокирующего Recv, стоит и ждёт, у процесса 1 аналогичная ситуация. Ни тот, ни другой процесс до Send не доходят, так и ждут бесконечно. Программа повиснет в любом случае |
| **процесс 0:**  **Send(1)**  **Recv(1)** | **процесс 1:**  **Send(0)**  **Recv(0)** | Оба процесса посылают что-то друг другу, но тут есть буфера, в который могут откладываться сообщения, посылающий разблокируется и может получить сообщение от другого. Ну и второй потом разблокируется и получит сообщение из буфера.  Этот кусок может сработать несколько раз, а на какой-нибудь другой не сработать, если буферов не хватит вдруг, например. Этот случай хуже в плане отлаживаемости (ну типа иногда работает, иногда нет -- странно, долго думать) |

**Разрешение тупиковых ситуаций:**

1. **процесс 0**: | **процесс 1**:

**Send(1)** | **Recv(0)**

**Recv(1)** | **Send(0)**

Этот вариант в простых ситуациях точно не повиснет, независимо от внутренних буферов коммуникация произойдёт

1. Использование неблокирующих операций (для более сложного кода удобнее)
2. Использование функции **MPI\_Sendrecv**

**Пример** (Обмен сообщениями чётных и нечётных процессов)

**#include "mpi.h"**

**#include <stdio.h>**

**int main(int argc, char \*\*argv) {**

**int size, rank, a, b;**

**MPI\_Status status;**

**MPI\_Init(&argc, &argv);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

**a = rank;**

**b = -1;**

**if((rank%2) == 0){**

**if(rank<size-1)**

**MPI\_Send(&a, 1, MPI\_INT, rank+1, 5, MPI\_COMM\_WORLD);**

**} else**

**MPI\_Recv(&b, 1, MPI\_INT, rank-1, 5, MPI\_COMM\_WORLD,**

**&status);**

**printf("process %d a = %d, b = %d\n", rank, a, b);**

**MPI\_Finalize();**

**}**

## Передача и приём сообщений без блокировки

**int MPI\_Isend(void \*buf, int count, MPI\_Datatype datatype, int dest, int msgtag, MPI\_Comm comm, MPI\_Request \*request)**

Неблокирующая посылка сообщения. Возврат из функции происходит сразу после инициализации передачи. Переменная **request** идентифицирует пересылку.

**Важно, что если процесс идёт считать дальше, не блокируясь, и изменит что-то в массиве buf до приёма сообщения другим процессом, то получим недетерминированную программу**

**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**.

**!!! До завершения неблокирующей операции не следует записывать в используемый массив данных !!!**

**Переменная request помогает понять, произошла соответствующая операция или ещё нет вот так:**

**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.

**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**.

**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). Если завершилось несколько из ожидаемых операций, то массив структур заполняется для них всех.

**Пример** (Обмен по кольцевой топологии при помощи неблокирующих операций)

**#include <stdio.h>**

**#include "mpi.h"**

**int main(int argc, char \*\*argv) {**

**int rank, size, prev, next; int buf[2];**

**MPI\_Request reqs[4];**

**MPI\_Status stats[4];**

**MPI\_Init(&argc,&argv);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

**prev = rank - 1;**

**next = rank + 1;**

**if(rank==0)**

**prev = size - 1;**

**if(rank==size - 1)**

**next = 0;**

**MPI\_Irecv(&buf[0], 1, MPI\_INT, prev, 5, MPI\_COMM\_WORLD, &reqs[0]);**

**MPI\_Irecv(&buf[1], 1, MPI\_INT, next, 6, MPI\_COMM\_WORLD, &reqs[1]);**

**MPI\_Isend(&rank, 1, MPI\_INT, prev, 6, MPI\_COMM\_WORLD, &reqs[2]);**

**MPI\_Isend(&rank, 1, MPI\_INT, next, 5, MPI\_COMM\_WORLD, &reqs[3]);**

**MPI\_Waitall(4, reqs, stats);**

**printf("process %d prev = %d next=%d\n", rank, buf[0], buf[1]);**

**MPI\_Finalize();**

**}**

**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.

**int MPI\_Testall(int count, MPI\_Request \*requests, int \*flag, MPI\_Status \*statuses)**

Проверка завершённости **count** асинхронных операций, ассоциированных с идентификаторами **requests**.

**int MPI\_Testany(int count, MPI\_Request \*requests, int \*index, int \*flag, MPI\_Status \*status)**

В параметре **flag** возвращается значение **1**, если хотя бы одна из операций асинхронного обмена завершена.

**int MPI\_Testsome(int incount, MPI\_Request \*requests, int \*outcount, int \*indexes, MPI\_Status \*statuses)**

Аналог **MPI\_Waitsome**, но возврат происходит немедленно. Если ни одна из операций не завершилась, то значение **outcount** будет равно нулю.

**int MPI\_Testsome(int incount, MPI\_Request \*requests, int \*outcount, int \*indexes, MPI\_Status \*statuses)**

Аналог **MPI\_Waitsome**, но возврат происходит немедленно. Если ни одна из операций не завершилась, то значение **outcount** будет равно нулю.

## Отложенные запросы на взаимодействие

**int MPI\_Send\_init(void \*buf, int count, MPI\_Datatype datatype, int dest, int msgtag, MPI\_Comm comm, MPI\_Request \*request)**

Формирование отложенного запроса на посылку сообщения. Сама операция пересылки не начинается!

Модификации функции **MPI\_Send\_init**:

* **MPI\_Bsend\_init** — формирование запроса на передачу сообщения с буферизацией;
* **MPI\_Ssend\_init** — формирование запроса на передачу сообщения с синхронизацией;
* **MPI\_Rsend\_init** — формирование запроса на передачу сообщения по готовности.

**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. Если система умеет положить несколько мелких пересылок в один буфер и потом раскидать, то можем выиграть. Если будет много огромных пересылок, то система может виснуть, т.к. это огромная нагрузка

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!

Отложенные операции нужны для того, чтобы один раз подготовить необходимую пересылку, а потом её много-много раз использовать

**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.

**Пример** (Схема итерационного метода с обменом по кольцевой топологии при помощи отложенных запросов)

**prev = rank – 1;**

**next = rank + 1;**

**if (rank == 0)**

**prev = size – 1;**

**if (rank == (size – 1))**

**next = 0;**

**MPI\_Recv\_init(&rbuf[0], 1, MPI\_FLOAT, prev, tag1, MPI\_COMM\_WORLD, &reqs[0]);**

**MPI\_Recv\_init(&rbuf[1], 1, MPI\_FLOAT, next, tag2, MPI\_COMM\_WORLD, &reqs[1]);**

**MPI\_Send\_init(&sbuf[0], 1, MPI\_FLOAT, prev, tag2, MPI\_COMM\_WORLD, &reqs[2]);**

**MPI\_Send\_init(&sbuf[1], 1, MPI\_FLOAT, next, tag1, MPI\_COMM\_WORLD, &reqs[3]);**

**for(…){**

**sbuf[0]=…;**

**sbuf[1]=…;**

**MPI\_Startall(4, reqs);**

**...**

**MPI\_Waitall(4, reqs, stats);**

**...**

**}**

**MPI\_Request\_free(&reqs[0]);**

**MPI\_Request\_free(&reqs[1]);**

**MPI\_Request\_free(&reqs[2]);**

**MPI\_Request\_free(&reqs[3]);**

## Совмещённые приём и передача сообщений

**int MPI\_Sendrecv(void \*sbuf, int scount, MPI\_Datatype stype, int dest, int stag, void \*rbuf, int rcount, MPI\_Datatype rtype, int source, 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**.

**Пример** (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**

**next = rank + 1;**

**if(rank == (size - 1))**

**next=MPI\_PROC\_NULL;**

**MPI\_Send (&buf, 1, MPI\_FLOAT, next, tag, MPI\_COMM\_WORLD)**

## 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.

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

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).

**int** **MPI\_Barrier**(**MPI\_Comm** **comm**)

Работа процессов блокируется до тех пор, пока все оставшиеся процессы коммуникатора **comm** не выполнят эту процедуру. Все процессы должны вызвать **MPI\_Barrier**, хотя реально исполненные различными процессами коммуникатора вызовы могут быть расположены в разных местах программы.

**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.

**ОБЯЗАТЕЛЬНО ВСЕ ПРОЦЕССЫ ДОЛЖНЫ ВЫЗВАТЬ ИМЕННО ФУНКЦИЮ MPI\_ BCAST**

данные -- по горизонтали, процессы -- по вертикали

|  |  |  |  |
| --- | --- | --- | --- |
| A0 |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

\*broadcast\*

|  |  |  |  |
| --- | --- | --- | --- |
| A0 |  |  |  |
| A0 |  |  |  |
| A0 |  |  |  |
| A0 |  |  |  |

**Пример**

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);**

**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.

**Чтобы не происходило лишних копирований для процесса root**: 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**.

|  |  |  |  |
| --- | --- | --- | --- |
| A0 | A1 | A2 | A3 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

\*scatter\* (gather -- в другую сторону, т.е. из второй таблички получим первую)

|  |  |  |  |
| --- | --- | --- | --- |
| A0 |  |  |  |
| A1 |  |  |  |
| A2 |  |  |  |
| A3 |  |  |  |

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);**

**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).

**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.

**Чтобы не происходило лишних копирований для процесса root**

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.

**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. (по сути это как сборка данных, а потом броадкаст)

**Чтобы не было лишних копирований для процесса root**

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**.

|  |  |  |  |
| --- | --- | --- | --- |
| A0 |  |  |  |
| B0 |  |  |  |
| C0 |  |  |  |
| D0 |  |  |  |

\*allgather\*

|  |  |  |  |
| --- | --- | --- | --- |
| A0 | B0 | C0 | D0 |
| A0 | B0 | C0 | D0 |
| A0 | B0 | C0 | D0 |
| A0 | B0 | C0 | D0 |

**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. (такое транспонирование по сути)

**Чтобы не было лишних копирований для процесса root**

Если для посылки и приема 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**.

|  |  |  |  |
| --- | --- | --- | --- |
| A0 | A1 | A2 | A3 |
| B0 | B1 | B2 | B3 |
| C0 | C1 | C2 | C3 |
| D0 | D1 | D2 | D3 |

\*alltoall\*

|  |  |  |  |
| --- | --- | --- | --- |
| A0 | B0 | C0 | D0 |
| A1 | B1 | C1 | D1 |
| A2 | B2 | C2 | D2 |
| A3 | B3 | C3 | D3 |

**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. (короче, тут не только количество разное разным процессам, но можем ещё и типы разные кидать. Ну и сдвиги поэтому не в элементах массива, а в байтах, так можем для любого типа сдвиги указать)

**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 iом элементе массива **rbuf** процесса **root**.(что-то похожее было и в OpenMP)

**Типы предопределённых глобальных операций:**

* **MPI\_MAX**, **MPI\_MIN** – 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”

**Чтобы не было лишних копирований для root**

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.

**Пример**

**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)**;

**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**.

(Короче, reduce для одного процесса, сам вызвал, сам получил, порадовался. Никаких межпроцессных взаимодействий тут нет)

**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 ∑**rcounts**(i) независимых глобальных операций **op** над соответствующими элементами массивов **sbuf**. (как предыдущая, просто неравными блоками результаты делим между процессами)

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…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.

**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.

**typedef void MPI\_User\_function (void \*invec, void \*inoutvec, int \*len, MPI\_Datatype type)**

Интерфейс пользовательской функции. 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 **MPI\_OP\_NULL**.

**#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;**

**}** //функция суммирования векторов по модулю 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

**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 (если одинаковый **key**, то система их там сама каким-то неопределённым образом перенумерует). Сколько различных значений color было указано, столько коммуникаторов в итоге и получим.

Процессы, которые не должны войти в новые группы, указывают в качестве **color** константу **MPI\_UNDEFINED**. Им в параметре **newcomm** вернется значение **MPI\_COMM\_NULL**.

**MPI\_Comm\_split(MPI\_COMM\_WORLD, rank%3, rank, new\_comm)**

//разбиение на 3 новых коммуникатора

## 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.

**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**.

**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)}**

**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 **d**ispls.

**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

**for(i=0; i<n; i++){**

**blocklens[i]=n-i;**

**displs[i]=(n+1)\*i;**

**}**

**MPI\_Type\_indexed(n, blocklens, displs, MPI\_DOUBLE, &newtype)**

Создание нового типа данных для описания верхнетреугольной матрицы.

**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.

**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);**

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)}**

**int MPI\_Type\_create\_subarray(int ndims, int sizes[], int subsizes[], int starts[], int order, MPI\_Datatype type, MPI\_Datatype \*newtype)**

* **newtype** задаёт **ndims–мерный** подмассив исходного **ndims–мерного** массива.
* **sizes** задает размеры по каждому измерению исходного массива,
* **subsizes** – размеры по каждому измерению выделяемого подмассива.
* **starts** задаёт стартовые координаты каждого измерения выделяемого подмассива в исходном массиве.
* Все массивы индексируются с **0**. Задаваемые значения не должны выводить подмассив за пределы исходного массива ни по одному из измерений.
* order задаёт порядок хранения элементов многомерного массива: **MPI\_ORDER\_C** (по строкам), **MPI\_ORDER\_FORTRAN** (по столбцам).
* **type** задаёт тип элементов массива.

**MPI\_Datatype newtype;**

**int sizes[2], subsizes[2], starts[2];**

**int rank;**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

**sizes[0] = 100;**

**sizes[1] = 100;**

**subsizes[0] = 100;**

**subsizes[1] = 25;**

**starts[0] = 0;**

**starts[1] = rank\*subsizes[1];**

**MPI\_Type\_create\_subarray(2, sizes, subsizes, starts, MPI\_ORDER\_C, MPI\_DOUBLE, &newtype);**

**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.

**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.

**int MPI\_Type\_dup(MPI\_Datatype type, MPI\_Datatype \*newtype)**

Создает новый тип данных **newtype**, аналогичный типу данных **type**. При этом производится копирование всех ассоциированных с типом **type** атрибутов. Если исходный тип данных уже был зарегистрирован, то автоматически будет зарегистрирован и создаваемый тип данных.

**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**.

**blocklens[0] = 1;**

**blocklens[1] = 1;**

**types[0] = MPI\_DOUBLE;**

**types[1] = MPI\_CHAR;**

**MPI\_Get\_address(dat1, &displs[0]);**

**MPI\_Get\_address(dat2, &displs[1]);**

**MPI\_Type\_create\_struct(2, blocklens, displs, types, &newtype);**

**MPI\_Type\_commit(&newtype);**

**MPI\_Send(MPI\_BOTTOM, 1, newtype, dest, tag, MPI\_COMM\_WORLD);**

**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

**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**.

**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.

**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.

**#include <stdio.h>**

**#include "mpi.h"**

**int main(int argc, char \*\*argv) {**

**int size, rank, position, i;**

**float a[10];**

**char b[10], buf[100];**

**MPI\_Init(&argc, &argv);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

**for(i = 0; i<10; i++){**

**a[i] = rank + 1.0;**

**if(rank==0)**

**b[i]='a';**

**else**

**b[i] = 'b';**

**}**

**position=0;**

**if(rank==0){**

**MPI\_Pack(a, 10, MPI\_FLOAT, buf, 100, &position,**

**MPI\_COMM\_WORLD);**

**MPI\_Pack(b, 10, MPI\_CHAR, buf, 100, &position,**

**MPI\_COMM\_WORLD);**

**MPI\_Bcast(buf, 100, MPI\_PACKED, 0, MPI\_COMM\_WORLD);**

**} else {**

**MPI\_Bcast(buf, 100, MPI\_PACKED, 0, MPI\_COMM\_WORLD);**

**MPI\_Unpack(buf, 100, &position, a, 10, MPI\_FLOAT, MPI\_COMM\_WORLD);**

**MPI\_Unpack(buf, 100, &position, b, 10, MPI\_CHAR, MPI\_COMM\_WORLD);**

**}**

**for(i = 0; i<10; i++)**

**printf("process %d a=%f b=%c\n", rank, a[i], b[i]);**

**MPI\_Finalize();**

**}**

## Плюсы и минусы производных типов данных и упаковки 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 (не оч написанная)

# Компоненты и системное программное обеспечение суперкомпьютеров

Весь суперкомпьютер состоит из двух основных частей:

1. Инфраструктура (вряд ли придётся сталкиваться, но знать, что она есть надо) -- поддерживает жизнеспособность вашего суперкомпьютера

* Помещение:
* площадь (не только для стоек, но и для нормального техобслуживания: чтобы можно было достать, заменить, перекоммутировать и т.д. -- это достаточно большие дополнительные площади)
* шумность
* доступ для техобслуживания
* противопожарное оборудование (оно должно соответствовать наличию суперкомпов, например, водой тушить нельзя прям совсем, а после порошкового тушения оборудование всё придётся с большой вероятностью менять)
* контроль доступа (жёсткий контроль должен быть, никаких общих ключей для нескольких человек, где непонятно, кто ключом воспользовался + видеонаблюдение оч желательно)
* Электричество (суперкомпы потребляют очень много электричества, в обычную розетку наверняка не будете включать суперкомплюхтер)
* источники питания, проводка (качество важно), автоматы
* ИБП (источники беспроводного питания, учитывая, насколько важно обеспечение бесперебойной работы + как часто бывают проблемы с электричеством), PDU, место
* резервирование питания (если есть критичное оборудование (сетевые файловые системы, коммутаторы), то надо чтобы у них было резервное питание, плюс критическое оборудования подключать к другим источникам бесперебойного питания, нежели основное оборудование)
* расчётное время работы от батарей
* проводка: фальшпол, короба, организаторы
* заземление, статика (+ учитывать влажность, при низком накапливается статическое электричество)
* Охлаждение
* мощность кондиционеров: потребляемая и реальная (ещё смотреть и считать, какой объём тепла должен быть охлаждён и сколько могут и на деле охлаждают ваши кондеи)
* влажность (<50% не круто)
* центральное отопление (если оно в помещении суперкомпа есть, то его оч желательно отключить или как-то отрегулировать, потому что обычно суперкомпы сами выделяют достаточно тепла, а так ещё лишнего охлаждать придётся), солнце (может нагревать стены. Их надо специальным образом обработать снаружи)
* зимнее время (обратить внимание на режим работы кондиционера в зимнее время. Если осталось центральное отопление, хоть и выключенное, посмотреть, чтобы вода была слита, чтобы трубы не разорвало из-за мороза вдруг)
* “открытое окно” (не надо окна открывать, крч. Во-первых, пыли поналетит снаружи, компам оно не надо, во-вторых, управлять эффективностью такого охлаждения сложно)

1. Вычислитель

* Сети: (для небольших кластеров сеть вообще одна и выполняет все функции, но для больших кластеров, конечно, имеет смысл разделять их)
* вычислительная (для вычислений)
* управляющая
* хранения данных
* сервисная
* другие
* Узлы:
* вычислительные
* узлы доступа
* сетевая файловая система
* служебные узлы
* система архивирования

[схема устройства суперкомпьютера в лекции]

## Служебные узлы

Какие службы могут быть на вычислительных узлах (всё перечисленное не всегда обязательно, в зависимости от устройства кластера могут быть ньюансы). Их можно запускать все на одном сервере, но лучше разделять на несколько, а ещё лучше установить их в виртуальные машины и запускать уже в виде виртуальных машин, чтобы в случае сбоев можно было поднять этим машины на резервном сервере. Практически все эти службы можно запускать на виртуалках, но с разными ограничениями.

* xCAT -- управление образами
* DHCP -- управление ip-адресами
* TFTP -- загрузка, установка
* NFS -- загрузка, root-fs
* DNS -- управление dns-именами
* NTP -- синхронизация времени
* LDAP -- управление учётными записями
* Обязательно должен быть управляющий сервер (на него будут заходить пользователи, с него будут запускаться задачи итд)
* Отдельно есть серверы, отвечающие за планирование и запуск задач (там будут работать планировщики, менеджеры ресурсов итд)
* Сервер лицензии (flexlm) -- сразу советуют предусмотреть такой сервер, а лучше несколько, особенно, если предполагается использование коммерческого софта, и лучше пусть это будет виртуальная машина, поскольку всё лицензионное прикрепляется к машине по MAC-адресу (крч, если присобачить к чему-то невиртуальному, а у этого чего-то сдохнет сетевая карта или материнская плата, это придётся заменить и сервер лицензии работать больше не будет, т.к. поменяется MAC-адрес. Ну, некоторые карты позволяют прописать его вручную, можно будет вернуть, а некоторые -- не позволяют, и всё, и капец, лицензию заново придётся покупать (а они бывают дорогие), а доказать компании сложно, что у тебя была лицензия, просто сервер сломался)
* Мониторинг (не обязательно, но желательно) -- смотреть, как там суперкомпьютер, какие сбои происходят, насколько загружены ресурсы и пр
* Статистика (не обязательно, но желательно) -- сбор статистики по задачам и данным
* Визуализация (не обязательно, но желательно)
* Копирование данных (не обязательно, но если специфика задачи предполагает большие объёмы данных) -- у этого сервера должен быть хороший доступ в сетевое хранилище, к нему надо мочь подключать внешние диски, чтобы напрямую качать эти объёмы данных, чтобы не делать это через сервер доступа

## Загрузка ОС

[на схеме в лекции это нужно для?

* вычислительных узлов,
* служебных серверов
* управляющего сервера]

**ОС на всех ваших серверах должна быть так или иначе загружена.**

* Может быть установлена локально на жёстком диске
* быстрая загрузка
* нет перегруза сети
* не занимает лишнего места в памяти
* долгая установка (хоть и есть способы ускорить) и обновление
* рассинхронизация настроек и версий
* Может загружаться по сети каждый раз в оперативную память (есть какой-то сервер для загрузки, и каждый сервер при старте получает данные, необходимые для загрузки образа ОС и просто их загружает в оперативную память)
* Единый образ для всех узлов группы
* Возможность быстро обновить всё
* Нагрузка на сеть
* Много места в памяти
* Нет возможности сохранить изменения

**Установка ОС на диск (варианты)**

* на один сервер и после этого просто склонировать жёсткий диск на ещё сколько-то и потом поставить их в другие серверы (может быть достаточно долго, поскольку объёмы дисков достаточно большие, хоть и есть штуки, которые позволяют это дело чуть-чуть оптимизировать и сделать чуть быстрее)
* автоматизированная установка (один раз ставите автоматизированную систему на один из ваших серверов)
* готовые стеки (такой, одноразовый вариант)
* использовать xCAT (уже было выше, рекомендуется)

**Автоматизированная установка ОС на диск**

Linux -- стандарт де-факто в мире ОС для суперкомпьютеров

В зависимости от дистрибутива после установки будет создаваться файлик в каталоге root, в нём будут фактически записаны все ответы на вопросы, которые вам задавал инсталлятор

**RH:** anaconda-ks.cfg

**SuSE**: autoyast.xml

**Debian:** preseed.file

Можно взять этот файл, скачать на местный http-сервер либо на nfs, и дальше, при установке (например, с флешки) на следующий узел можно указать выделелнные ниже аргументы инсталлятору (ну, для каждого дистрибутива свой)

linux **ks=http://путь-к-файлу.cfg**

linux **autoyast=http://путь-к-файлу.xml**

linux **auto preseed=http://путь-к-файлу**

После этого инсталлятор увидит этот файл, не будет больше ничего у вас спрашивать, установит так же, как записано в файлике.

**Готовые стеки**

Есть уже готовые дистрибутивы, которые позволяют автоматизировать эту работу. Вообще говоря, они предназначены для того, чтобы быстренько развернуть кластер и что-то там посчитать. Долговременно их использовать может быть не оч удобно, тк в них не встроены средства управления, которые позволяли бы гибко настраивать то, что вы развернули.

* **ROCKS**
* **PelicanHPC / Parallel Knoppics**

**xCAT (рекомендовано)**

Всё сделает за вас, для установки использует либо загрузку ОС в память, либо сам формирует нужные файлики, делает установку в виртуальном окружении ОС и сам формирует образ, который нужно будет развернуть на ваших вычислительных узлах

* использует anaconda/autoyast
* управляет DNS, DHCP, TFTP (думать об их настройке не придётся, получите заранее настроенные адреса, параметры загрузки по сети)
* поддерживает удалённое управление питанием и консоль (через IPMI, ILO, ...)
* Имеет встроенные “массовые” команды

**Виды установки ОС с помощью xCAT**

* Локально (state**full**)
* По сети в память (state**less**)
* По сети, в памяти только изменения (state**lite**)

**Объекты:** site, node, group, network, osdistro, osimage, route...

**Таблицы:** domain, hosts, ipmi, litetree, mac, nodegroup, nodelist, nodetype, postscripts,... 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

chtab node=compute hosts.ip='|\D+(\d+)|10.0.0.(10+$1)|'

## Файловые системы

[на схеме в лекции это нужно? для хранилища данных]

Понятно, что есть какая-то локальная ФС, но для нормальной работы вычислителя нужна сетевая ФС

* **NFS (Network File System)** -- одна из первых сетевых ФС, которая до сих пор неплохо себя чувствует (в отличие от меня, которую уже тошнит от этого всего)

Есть один сервер, к нему подключён жёсткий диск, на нём обычная ФС, и часть этой системы может раздаваться по протококлу NFS сразу многим вычислительным узлам

* узкое место между основным сервером и дисковым хранилищем (сильно зависит от хранилища)
* узкое место -- сам вот этот сервер в качестве сетевого, сложно будет организовать параллельную работу сетевых интерфейсов (если их вообще несколько)
* **PanFS** -- параллельная ФС (есть набор специальных штук (я вам не третий поток, вообще не втыкаю, как это называется), в каждой из которых есть собственные жёсткие диски и собственная ФС, юнит (?) которой установлен на каждом из узлов. И узлы работают параллельно со всеми этими штуками с жёсткими дисками
* **Lustre** -- самое старое решение. Есть один или несклоько серверов метаданных (MetaDataServer(s)), к ним (**единовременно только к одному из них!!!**) подключается некоторое хранилище (MetaDataTarget). При этом если сервер, к которому подключены, выходит из строя, то мы можем вручную или автоматически (как сделаем), перемонтировать на другой сервер метаданных. Все данные размазываются (каким-то образом, каким MDS решит) по Object Storage Targetам, эти OSS могут подключаться к нескольким (вообще говоря) Object Storage Serverам, но одновременно только к одному (при выходе из строя аналогичные действия, как выше). Если OST выходит из строя, то это автоматически означает, что часть данных мы потеряли, так как OST не дублируются. Если потеряли MDT, то потеряли вообще всю файловую систему. Поэтому создатели Lustre советую дублировать инфу на уровне дисков (т.е подключать какие-нибудь RAID-массивы как MDT, так и скорость можно увеличить и надёжность хранения данных повысить). Крч, **СОХРАННОСТЬ ДАННЫХ НЕ ГАРАНТИРУЕТСЯ**

## Контроль ресурсов (управление задачами)

[по схеме это относится к служебным и управляющему серверам]

Для этого используются **менеджеры ресурсов**:

* **Slurm** (бесплатненый)
* **Torque** (бесплатный)
* **OpenPBS**
* **LSF**, **BrightManager**, **Moab**, ...

Все эти менеджеры устроены по примерно одинаковой схеме. Разбор на примере Slurm

Есть сервис **slurmctld** -- сам менеджер ресурсов (рядом с ним может ещё работать отдельный планировщик и сервис для эккаунтинга). По сети с ним связываются команды, которые запускают пользователи и администраторы (**sbatch**, **sinfo**, **squeue**, **scontrol**, ...), с их помощью можно ставить задачи, смотреть, что там происходит, менять настройки итд. Если Slurm решает, что нужно запустить какие-то задачи на каких-то узлах, он должен с ними связаться как-то и им об этом сообщить. Для этого на вычислительных узлах работают его агенты (**slurmd**), и если такой агент запущен и он рапортует, что узел в нормальном состоянии, то только тогда **slurmctld** уверен, что на нём можно запускать задания. И для старта от отдаст нужные команды агентам. Если задача скомпилирована с поддержкой Slurmа (большинство реализаций MPI имеют встроенную поддержку Slurmа), то демоны **slurmd**, запускают отдельные демоны **slurmstepd**, и каждому из них запускается отдельный процесс вашей параллельной задачи. Это делается так, чтобы если будете менять настройки и перезапускать slurmd, вы не потеряли контроль над задачей.

Настройки Slurm лежат в файле **/etc/slurm/slurm.conf**

Основная программа для управления демоном -- **scontrol** (позволяет на ходу менять параметры, которые могут быть прописаны и даже которые не прописаны в файле настроек)

* show node | job | partition |... (если создать новый узел, то после перезагрузки он не сохранится. Да и вообще, всё, что поменялось на ходу. Сохранится только то, что есть изначально в файле настроек)
* create ...
* delete ...
* update ...

## Хранение и синхронизация учётных записей

* passwd (можно синхронизировать все файлы, в которых инфа об учётных записях: passwd, shadow, … Но это долго и сложновато, если узлов много)
* nis+ (сетевое решение) -- ограничение до 70 узлов (иначе захлёбываться начинает)
* LDAP (сетевое решение) -- ограничение до 70 узлов
* Гибридный метод -- храненение данных в LDAP-сервере, а при любом существенном изменении заново формируются файлы passwd, shadow, …, и вычислительные узлы забирают эти файлы к себе

**passwd** | root:x:0:0:root of evil:/root:/bin/bash | логин:x:UID:GID:инфо:/дом:/шелл шелл **shadow** | root:\*:16219:0:99999:7::: | логин:пароль:...

## Лицензии

* Локальные (это файлик, который кладётся в специальное место, и при запуске программа должна его там найти. Но такое бывает редко) и сетевые (вот таких много-много вариантов)
* Привязка:
* MAC-адрес (чаще всего)
* Серийный номер материнской платы
* Файл
* Тип:
* Список пользователей/серверов (фиксируется, на них выписывается, больше никто не могёт пользоваться лицензией)
* Число пользователей (network) (тут подводный камень -- задержка между волнами максимального числа пользователей, 15 минут, например)
* Число одновременных запусков (floating)

### Лекция 13

## Решение задачи на компьютере

Задача ⇒ Метод ⇒ Алгоритм ⇒ Технологии программирования ⇒ Программа ⇒

Системное ПО ⇒ Компьютер

## Почему важно понимать, как написаны программы?

**Нужно выполнить:** A[i][j][k] = A[i-1][j][k] + B[j][k] + B[j][k], i=1,40; j=1,40; k=1,1000

Cray C90, пиковая производительность **960** Mflop/s

**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)**

**Производительность на этой программе**: **20** Mflop/s на Cray C90

**Почему? Не используются особенности архитектуры. Как переписать?**

1. Cray C90 -- векторная машина, все операции внутреннего цикла, которые можно заменить на векторные операции, должны быть информационно независимы. Тут A[i] вычисляется от A[i-1] ⇒ зависимость
2. Используются конвейерные функциональные устройства, т.е. короткие векторные операции вычисляются медленнее, чем длинные (внутренний цикл содержит всего 40 итераций)
3. сумма двух одинаковых слагаемых, вместо умножения на 2 (используется только устройство сложения, а можно задействовать ещё и устройство умножения)

**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)**

**Производительность на изменённой программе: 700** Mflop/s на Cray C90

## Почему важно знать, как устроены алгоритмы?

**Исходный код (порядок циклов: ( i, j, k))**

**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]**

**Возможен ли порядок?**

( i, k, j) - ДА

( k, i, j) - ДА

( k, j, i) - ДА

( j, i, k) - ДА

( j, k, i) - ДА

**Почему возможен другой порядок?** (ответ на это в конце лекций по алгоритмам будет?..)

**А зачем нужен другой порядок?**

Получаем разную производительность в зависимости от процессора и выбора порядка циклов (по-разному хранятся данные, эффективно или не очень эффективно используем 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…

## Вершины: итерации циклов.

**for( i = 0; i < n; ++i) {**

**A[i] = A[i – 1] + 2;**

**B[i] = B[i] + A[i];**

**}**

Будет n вершин, каждая вершина соответствует двум операциям (телу цикла), выполняемым на одной итерации цикла

## Вершины: срабатывания операторов.

**for( i = 0; i < n; ++i) {**

**A[i] = A[i – 1] + 2;**

**B[i] = B[i] + A[i];**

**}**

Получим 2n (?) вершин, каждая из которых соответствует одному из двух операторов тела данного цикла, выполненному на некоторой итерации

**Дуги:** отражают связь (отношение) между вершинами.

Выделяют два **типа отношений**:

* **операционное отношение** -- две вершины A и B соединяются направленной дугой ⇔ вершина B может быть выполнена сразу после вершины A (**Операционное отношение = отношение по передаче управления.**)

**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)**

**Граф:** (1) → (2) → (3) → (4)

* **информационное отношение** -- две вершины A и B соединяются направленной дугой ⇔ вершина B использует в качестве аргумента некоторое значение, полученное в A

(**Информационное отношение = отношение по передаче данных.**)

**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)**

## Четыре основные модели программ

**Граф управления программы**

**Вершины:** операторы

**Дуги:** операционное отношение

**for( i = 0; i < n; ++i) {**

**A[i] = A[i – 1] + 2; (1)**

**B[i] = B[i] + A[i]; (2)**

**}**



**Информационный граф программы**

**Вершины:** операторы

**Дуги:** информационное отношение

**for( i = 0; i < n; ++i) {**

**A[i] = A[i – 1] + 2; (1)**

**B[i] = B[i] + A[i]; (2)**

**}**

**Операционная история программы**

**Вершины:** срабатывания операторов

**Дуги:** операционное отношение

**for( i = 0; i < n; ++i) {**

**A[i] = A[i – 1] + 2; (1)**

**B[i] = B[i] + A[i]; (2)**

**}**



**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**

**Вершины:** срабатывания операторов

**Дуги:** информационное отношение

**for( i = 0; i < n; ++i) {**

**A[i] = A[i – 1] + 2; (1)**

**B[i] = B[i] + A[i]; (2)**

**}**



**Свойства информационной истории:**

* ациклический граф
* нет кратных дуг

**Несколько вопросов**

**???** Может ли информационная история некоторого фрагмента программы иметь 100 вершин и ни одной дуги?

**!!!** ДА (это идеальный для распараллеливания кусочек, такие и надо искать для распараллеливания)

**for( i = 0; i < 100; ++i)**

**A[i] = B[i] + C[i]\*x;**

**???** Может ли информационная история некоторого фрагмента программы иметь 67 вершин и 3 дуги?

**!!!** ДА

**for( i = 0; i < 63; ++i)**

**A[i] = B[i] + C[i]\*x;**

**x1 = 10;**

**x2 = x1+1;**

**x3 = x2+2;**

**x4 = x3+3;**

**???** Может ли информационная история некоторого фрагмента программы иметь 20 вершин и 200 дуг?

**!!!** НЕТ. Вспомним свойства информационной истории:

* ациклический граф,
* нет кратных дуг.

Макс.число дуг: (n-1) + (n-2) + (n-3) + … + 2 + 1 = n\*(n-1)/2

**???** Может ли граф управления некоторого фрагмента программы состоять из нескольких компонент связности?

**!!!** ДА.

**x1 = 10;**

**x2 = x1+1;**

**goto A;**

**B: x3 = x2+2;**

**goto B;**

**A: x4 = x2+3;**

**???** Модель некоторого фрагмента программы в качестве подграфа содержит следующий граф:



Какой моделью могла бы быть исходная модель?

ГУ ИГ ОИ ИИ

ГУ может быть, например, условие if,else

**!!!** Не может быть только ОИ (в ней не может быть ветвлений, это последовательный граф)

**Множество графовых моделей программ**

|  |  |  |
| --- | --- | --- |
|  | **Компактные модели** | **Истории** |
| **Операционные модели** | Граф Управления | Операционная История |
| **Информационные модели** | Информационный Граф | Информационная История |

## Какое отношение выбрать для описания свойств программ?

**Информационная структура** – это основа анализа свойств программ и алгоритмов.

**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)**

Показывает участки, которые могут/не могут выполняться параллельно. (1) и (2) могут исполняться только последовательно, а вот (3) и (4) можно выполнять параллельно.

**Информационная зависимость** определяет критерий эквивалентности преобразований программ. (Если поменяли программу, но граф остался неизменный, то результат будет таким же с точностью до ошибок округления)

**Информационная независимость** определяет ресурс параллелизма программы.

В общем, информационные модели нам нравятся больше, выбираем их. А какую брать: компактную или историю?

**Аргументы для выбора степени компактности модели:**

* компактность описания, | (компактность +)
* информативность, | (история + )

(одному информационному графу могут соответствовать несколько информационных историй)

* сложность построения. | (компактность +)

Непонятно, что выбирать, хочется найти компромисс...

**Граф алгоритма** – это параметризованная информационная история:

* компактность описания за счет параметризации,
* имеет информативность истории,
* разработана методика построения графа алгоритма по исходному тексту программ.

**Общая схема анализа и преобразования структуры программ**

Исходная программа → Построение графа алгоритма → Исследование графа алгоритма → Преобразование графа алгоритма → Преобразованная программа

## Основатели теории анализа структуры программ и алгоритмов

**Ершов Андрей Петрович**, академик, создатель сибирской школы системного и теоретического программирования. Многие его работы посвящены методам изучения свойств и структуры программ. Еще в 60-х годах он рассматривал задачу преобразования схем программ над общей и распределенной памятью, изучал фундаментальные основы графовых моделей программ.

**Воеводин Валентин Васильевич**, академик, создатель математической теории информационной структуры программ и алгоритмов. Разработал методы нахождения и описания информационной структуры программ по их исходному тексту, методы определения потенциала параллелизма и эквивалентного преобразования программ.

## Теорема о построении графа алгоритма

[В скобочках -- простыми словами]

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 [аргументы операторов] каждого оператора фрагмента будет указано конечное множество троек вида

( N, ∆(N), F(∆, N) )k , где:

N – линейный выпуклый многогранник в пространстве внешних переменных фрагмента [входные данные программы],

∆(N) – линейный выпуклый многогранник в пространстве итераций фрагмента [срабатывания операторов],

F(∆, N) – линейная векторная функция, описывающая входящие дуги [информационное отношение].

### Лекция 14

# Как описать ресурс параллелизма программ и алгоритмов?

## Ярусно-параллельная форма графа алгоритма

Нет циклов, нет кратных дуг, и его можно сделать направленным

Причешем граф: перенумеровав вершины, сделаем так, чтобы все дуги были направлены в одну сторону: расположим вершины на ярусах, все дуги начинаются на ярусах с меньшим номером и заканчиваются вершиной на ярусе с бОльшим номером, между вершинами, расположенными на одном ярусе, не может быть дуг. Получили ярусно-параллельную форму графа алгоритма.

Все операции одного яруса можем выполнять параллельно (между ними нет дуг, а значит, и зависимостей)

Эта форма показывает способ параллельного исполнения программы (сначала выполняем все операции на 1 ярусе выполняются, потом на втором ярусе и так далее)

**Высота ЯПФ** -- число ярусов

**Ширина яруса** -- число вершин, расположенных на ярусе

**Ширина ЯПФ** -- максимальная ширина ярусов ЯПФ

**Высота ЯПФ** -- сложность параллельной реализации алгоритма/программы

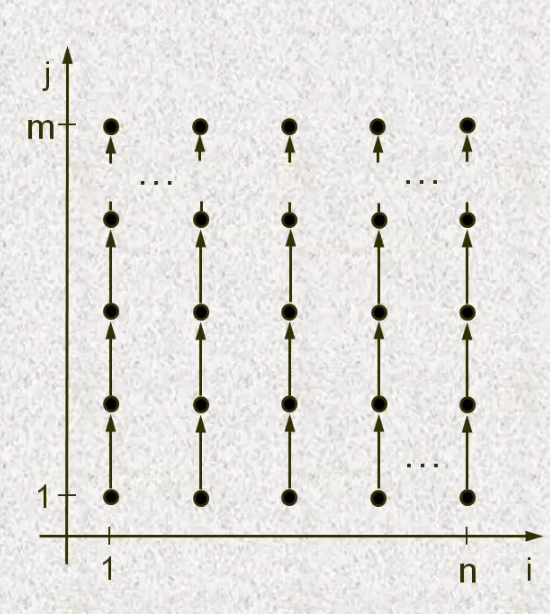
ЯПФ не единственную форму имеет

**Канонический вид ярусно-параллельной формы**

Ярусно-параллельная форма называется канонической, если у любой вершины, кроме вершин первого яруса, есть входная дуга, идущая с предыдущего яруса.

(Стараемся сжимать ЯПФ, уменьшать кол-во ярусов)

**Высота канонической ЯПФ** = длине критического пути + 1.

**Критический путь в ориентированном ациклическом графе** – это путь максимальной длины.(За сколько шагов мы можем выполнить программу?)

**Каков у фрагмента ресурс параллелизма?**

**for( i = 0; i < n; ++i)**

**for( j = 0; j < m; ++j)**

**A[i][j] = A[i][j–1] + C[i][j]\*x;**

Чему, согласно закону Амдала, равно максимальное ускорение, которое можно получить при исполнении данного фрагмента на параллельной вычислительной системе?

S ~ 1/a,

a = число последовательный операций / общее число операций = m / (m \* n) = 1 / n

S ~ n

## Виды параллелизма в программах

**Конечный параллелизм** определяется информационной независимостью некоторых фрагментов в тексте программы. (Какие-то фрагменты независимы друг от друга и не зависят от внешних параметров (от n не зависят))

**Массовый параллелизм** определяется информационной независимостью отдельных итераций циклов программы.

**Конечный параллелизм (пример)**

Три куска между линиями можно выполнять строго параллельно друг с другом: (можем ускорить максимум в 3 раза)

**cout << "N=" << N << endl;**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**cycleTestWithUnroll\_KJI("k");**

**cycleTestWithUnroll\_KJI("j");**

**cycleTestWithUnroll\_KJI("i");**

**cycleTestWithUnroll\_KJI\_3("k");**

**cycleTestWithUnroll\_KJI\_3("j");**

**cycleTestWithUnroll\_KJI\_3("i");**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**cycleTest("j,i,k");**

**cycleTest("i,k,j");**

**cycleTest("k,j,i");**

**cycleTest("i,j,k");**

**cycleTest("k,i,j");**

**cycleTest("j,k,i");**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**float \*\*\*a12=new float\*\*[N];**

**for (i=0;i<N;i++) {**

**a12[i]=new float\*[N];**

**for (j=0;j<N;j++) {**

**a12[i][j]=new float[N];**

**for (k=0;k<N;k++) {**

**a12[i][j][k]=(float)1/(i+j+k+1);**

**}**

**}**

**}**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**for (i=1;i<N;i++)**

**for (j=1;j<N;j++)**

**for (k=1;k<N;k++) {**

**testee[i][k] = testee[i][k] + S[k]\*A[k][j][i] + P[i][j]\*A[k][j][i-1] +**

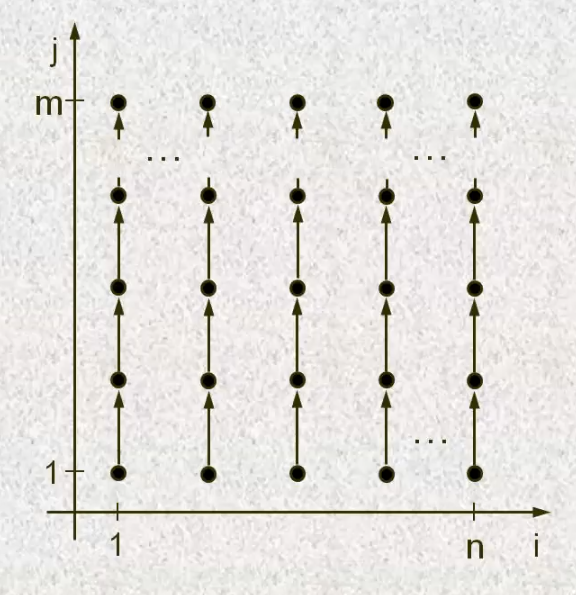
**P[i][k]\*A[k][j-1][i] + P[j][k]\*A[k-1][j][i];**

**}**

**Массовый параллелизм (пример)**

Тут параллелизм зависит от n

**for( i = 0; i < n; ++i)**

**A[i] = B[i] + C[i]\*x;**

**Массовый параллелизм:**

* **координатный**
* **скошенный**

**Координатный параллелизм (пример)**

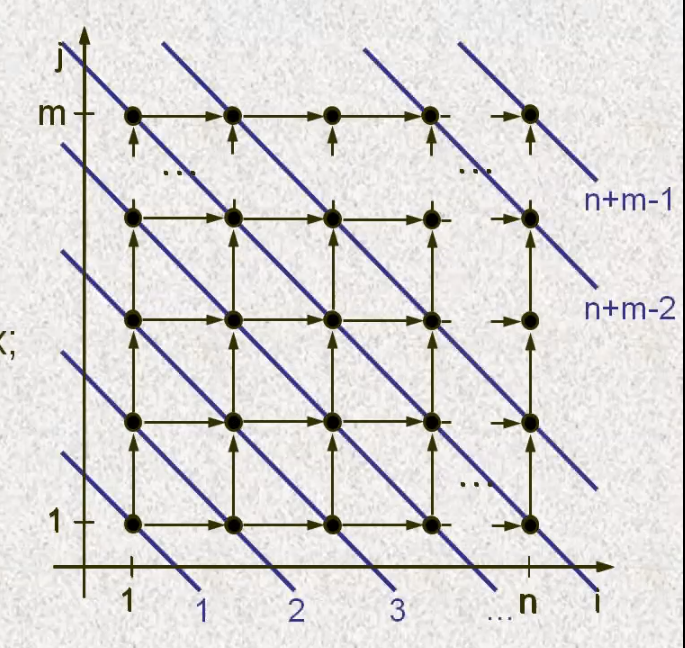
#pragma omp parallel for

for( i = 0; i < n; ++i)

for( j = 0; j < m; ++j)

A[i][j] = A[i][j–1] + C[i][j]\*x;

Утверждение: для того чтобы цикл был параллельным необходимо и достаточно, чтобы для любой тройки графа алгоритма данного цикла включение было верным, где — это многогранник из тройки, = {f1 = i1 , i1 — это параметр анализируемого цикла, f1 — это первая компонента векторной функции Fi из тройки. i Gi i Gi



**Скошенный параллелизм:**

for( i = 0; i < n; ++i)

for( j = 0; j < m; ++j)

A[i][j] = A[i][j–1] + A[i-1][j]\*x;

Слева Информационная История

По оси i и по оси j нет координатного параллелизма

Параллельная сложность = Длина критического пути = n + m

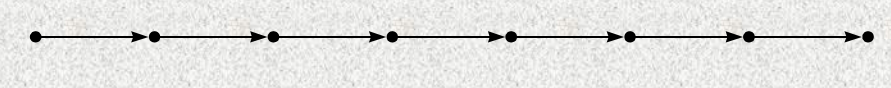
Параллелизм есть - он расположен по диагонали. Каждая диагональ - это ярус

Выполняются операции на диагоналях

Но есть сложность - неудобный параллелизм.

1. В начале мало независимых операций, которых становится больше к середине и снова уменьшается
2. Выразить параллелизм сложно. Не можем ни одну pragma перед циклом написать. Нужно менять сам цикл

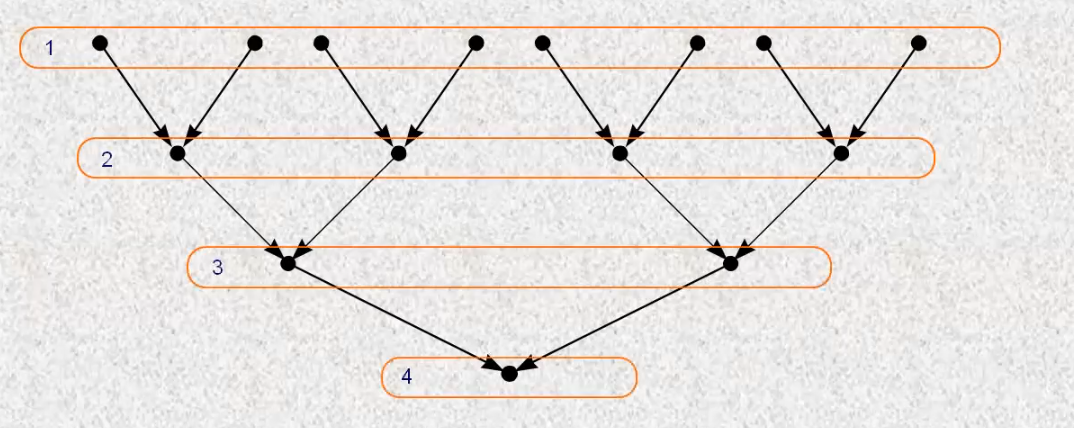
**Эквивалентные преобразования программ (суммирование элементов массива)**

s = 0.0; 

for ( i = 0; i < n; ++i )

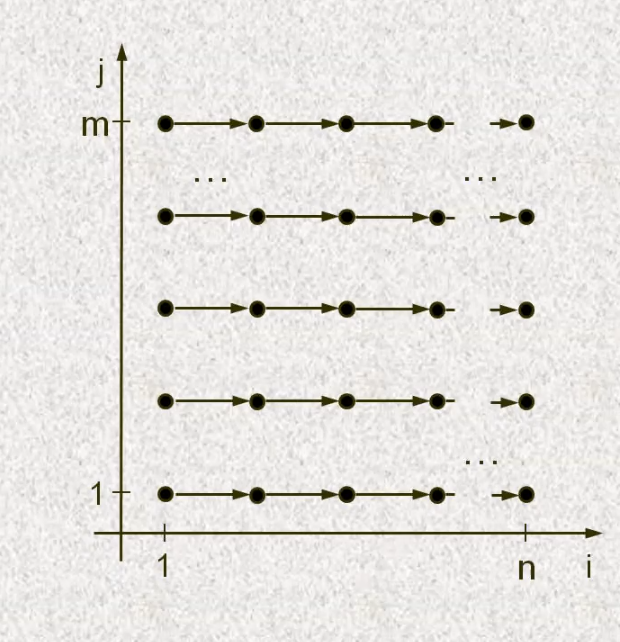
s = s + A[ i ];

Здесь нет параллелизма. Но мы можем перейти к другому алгоритму - к алгоритму сдваивания

Складываем пары, потом частичные суммы

Результат может не совпадать с последовательным решением

**Элементарные преобразования:**

В основном параллелизм выполняет компилятор, но иногда он может их пропустить, поэтому лучше самому это делать

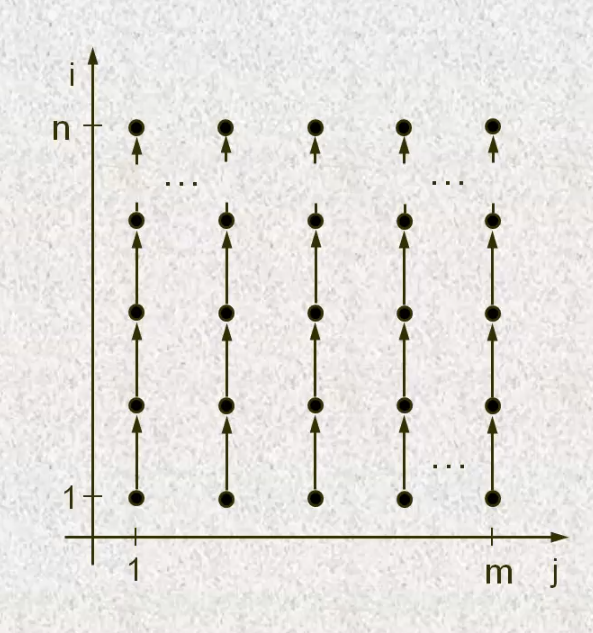
**Перестановка циклов:**

for( i = 0; i < n; ++i)

#pragma omp parallel for (вот здесь каждый раз будут создаваться нити, а это очень дорогостоящая операция)

for( j = 0; j < m; ++j)

A[i][j] = A[i-1][j] + C[i][j]\*x;

Можем pragma выставить перед вторым циклом, но это дорогостоящая операция. С каждым выполнением большого цикла, нити будут заново создаваться

Поменяем местами циклы

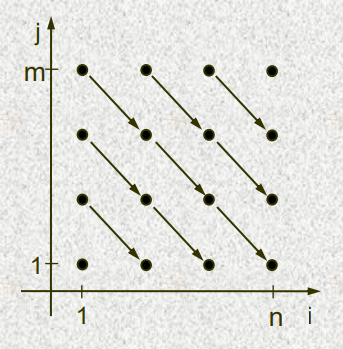
#pragma omp parallel for

for( j = 0; j < m; ++j)

for( i = 0; i < n; ++i)

A[i][j] = A[i-1][j] + C[i][j]\*x;

Не всегда 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;

for( i = 0; i < n; ++i)

for( j = 0; j < m; ++j)

A[i][j] = A[i–1][j+1] + C[i][j]\*x;

Получим совершенно другой результат, если поменяем местами циклы

**Распределение циклов:**

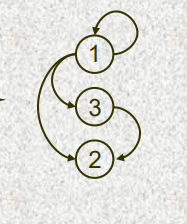
for( i = 1; i < n; ++i) {

A[i] = A[i–1]\*p + q;

C[i] = (A[i] + B[i–1])\*s;

B[i] = (A[i] – B[i])\*t;

}



Разобьем на три 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;

Первый цикл нельзя распараллеливать(есть зависимость от предыдущего значения), он должен выполнятся последовательно. Остальные можно параллельно

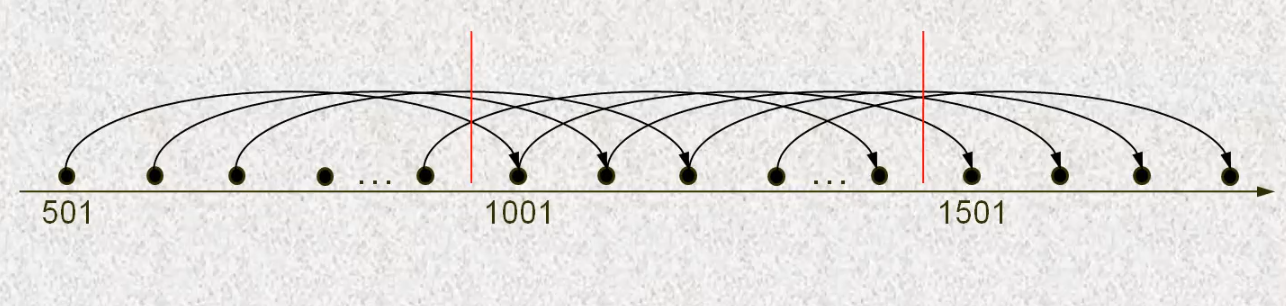
**Утверждение**: для того чтобы можно было выполнить распределение цикла необходимо и достаточно, чтобы распределяемые части находились в разных компонентах сильной связности информационного графа тела данного цикла

**Расщепление циклов.**

for( i = 501; i <= 2000; ++i)

A[i] = A[i] + A[i–500];

Нужно построить Информационный Граф



Разбили на три части. Итог:

#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];

**Эквивалентные ли преобразования?**

for( i = 0; i < n; ++i) (1)

D[i] = D[i] \* F[i];

if( m == 3 )

for( i = 0; i < n; ++i)

R[i] = P[i] + D[i];

else

for( i = 0; i < n; ++i)

R[i] = Q[i] – D[i];

if( m == 3 )

for( i = 0; i < n; ++i) (2)

R[i] = P[i] + D[i] \* F[i];

else

for( i = 0; i < n; ++i)

R[i] = Q[i] – D[i] \* F[i];

Нет, преобразования не эквиваленты. Потому, что по выходу из первого цикла (1) массив D вычислен, а во втором (2) массив D остался неизменным. То есть если массив D потом дальше не используется, то всё хорошо, иначе значения могут быть у нас различны. Но мы увеличили производительность в два раза

**Малый размер программы не означает простоту её структуры**

Пример:

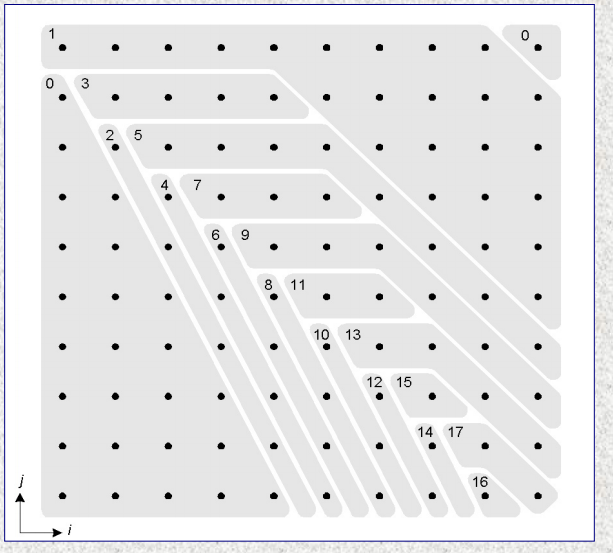
DO i = 1, n

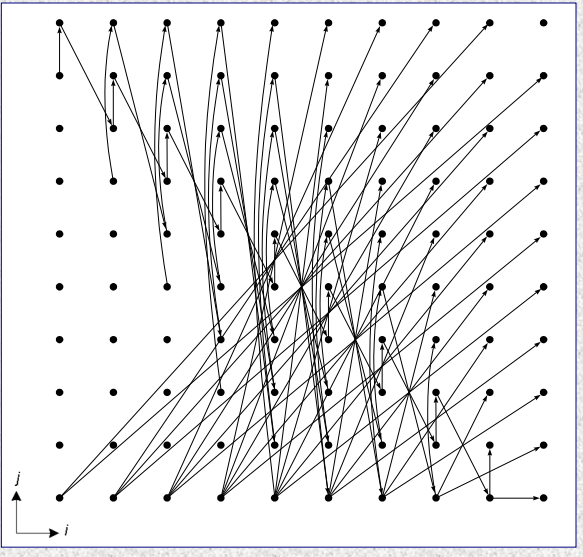
DO j = 1, n

U( i + j ) = U( 2\*n – i – j + 1)\*q + p

EndDO

EndDO

ИИ её и ЯПФ. Красотища какая



Видно, что параллелизма очень много

Можно сделать преобразование, чтобы ускорить программу

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

Внешний цикл делает перебор по ярусам, а два внутренних цикла являются параллельными

**Решение СЛАУ методом Гаусса:**

Примерная схема программы

do i = n, 1, -1

s = 0

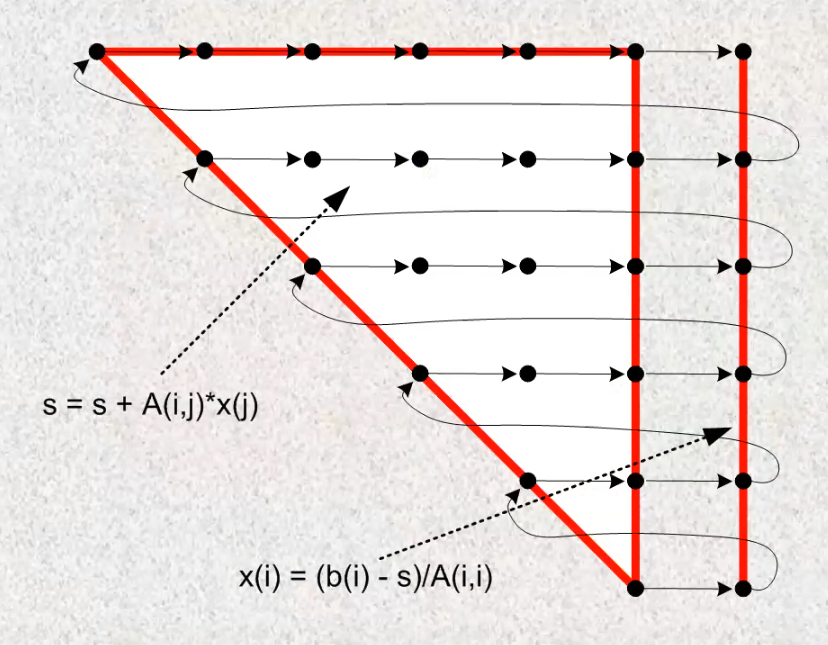
do j = i+1, n

s = s + A(i,j)\*x(j)

end do

x(i) = (b(i) - s)/A(i,i)

end do



Двигаемся по треугольнику вверх, ищем частичную сумму.

Внутри треугольника - это накопление очередной суммы

Фрагмент:

do j = i+1, n

s = s + A(i,j)\*x(j)

end do

Отдельно вынесены вычисление x(i)

Фрагмент:

x(i) = (b(i) - s)/A(i,i)

Где используется x(i)? Она будет сразу же использована в цикле

do j = i+1, n

s = s + A(i,j)\*x(j)

end do

при первом проходе, поэтому имеем длинную дугу

Поэтому критический путь проходит через все вершины и параллелизма никакого нет

Решение - перейти к другому алгоритму

Мы можем складывать во внутреннем цикле не по увеличение индекса, а по уменьшению

do i = n, 1, -1

s = 0

do j = n, i+1, -1

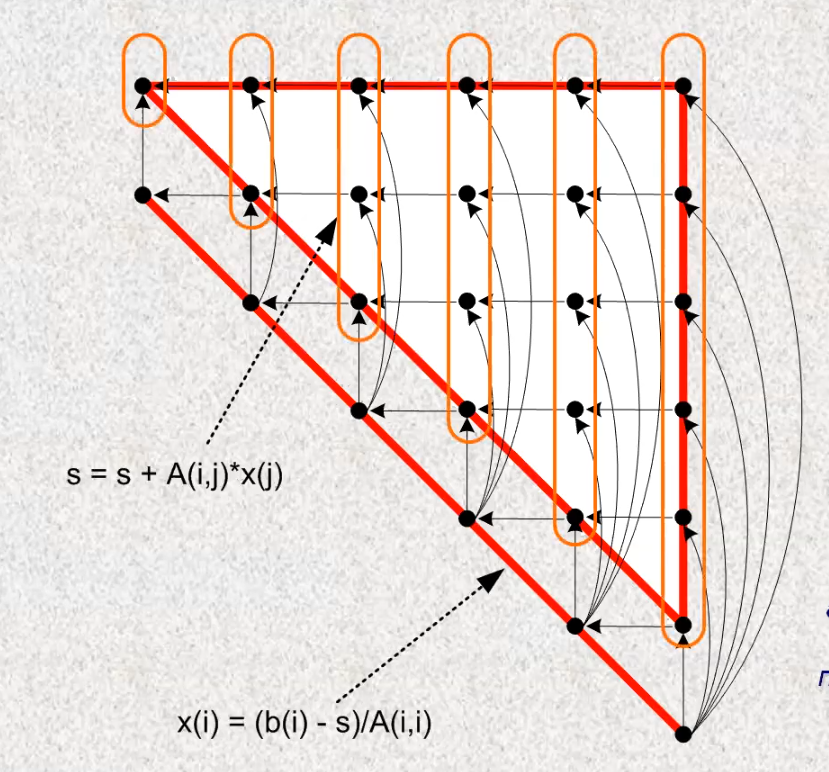
s = s + A(i,j)\*x(j)

end do

x(i) = (b(i) - s)/A(i,i)

end do

ИИ:



Вычислили x(n)(самая нижняя точка) и оно будет использоваться во всех вершинах!

Потом делаем последовательный шаг, вычисляем x(n-1), делаем последовательный шаг и так далее.

Значит, мы можем распараллелить программу

Критический путь графа алгоритма имеет длину o(n), следовательно данная программа обладает хорошим ресурсом параллелизма!

Оранжевое - это ярусы

Алгоритмы - это круто