1. **Архитектура вычислительных систем (ВС). Многопроцессорные системы с общей и распределенной памятью. Классификация ВС. Топология коммуникационных сетей ВС.**

Классификация архитектур ВС

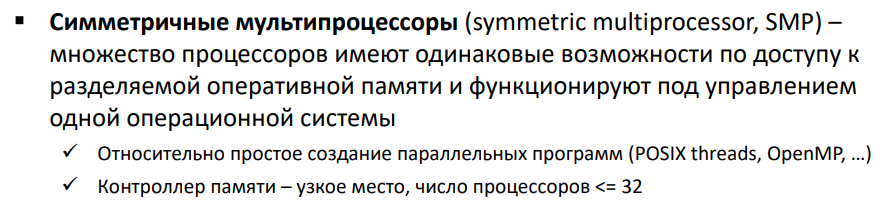
SISD – (Один поток данных, одна инструкция) последовательная ВС; одно устройство управления работает с одним потоком инструкций в памяти, выполняя их на последовательном процессоре (работает с одним потоком данных): первые процессоры.

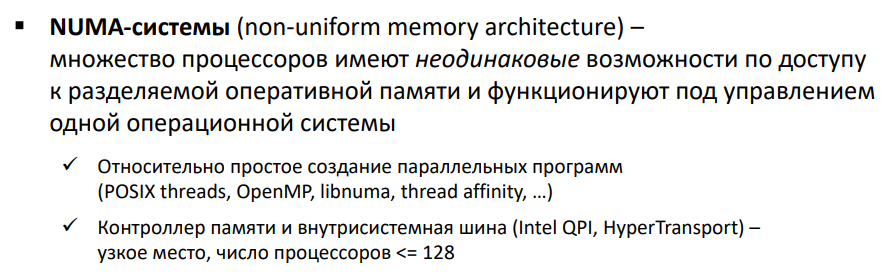
SIMD – (Много потоков данных, одна инструкция) вычислительная систем, в которой множество процессоров выполняют одну инструкцию над своими локальными данными: векторные ВС Cray, NEC; наборы векторных инструкций AVX, AltiVec, NEON SIMD; GPU.

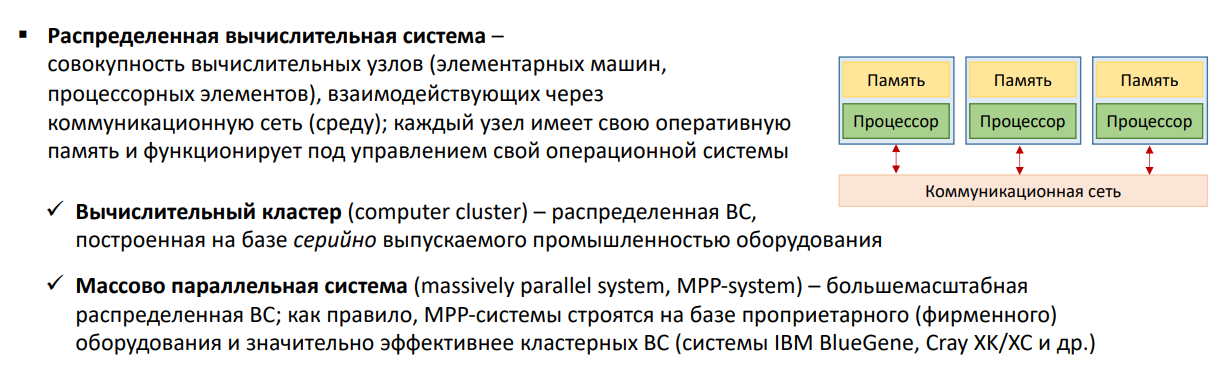
MISD – (Один поток данных, много инструкций) вычислительная система типа “много потоков команд – один поток данных”: конвейерные ВC (частично) и систолические ВС (systolic arrays, частично).

MIMD – (Много потоков данных, много инструкций) совокупность процессорных элементов, работающих со своими локальными потоками команд и данных: вычислительные кластеры, MPP-системы.

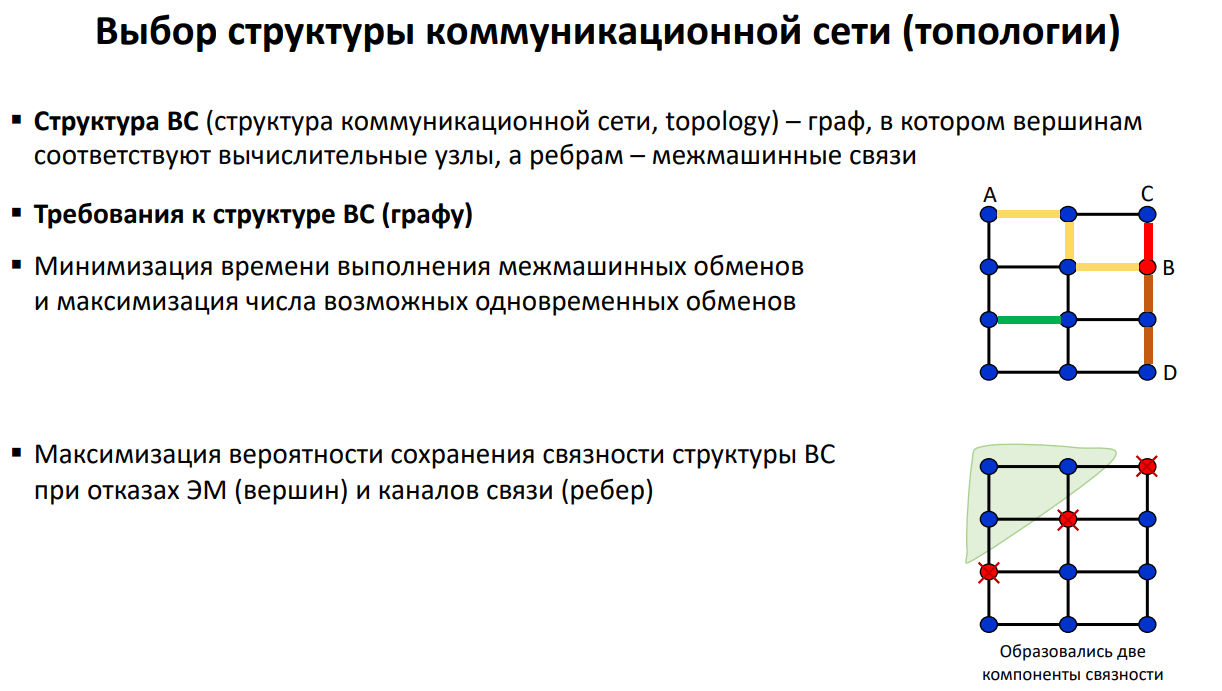
Классификация АВС

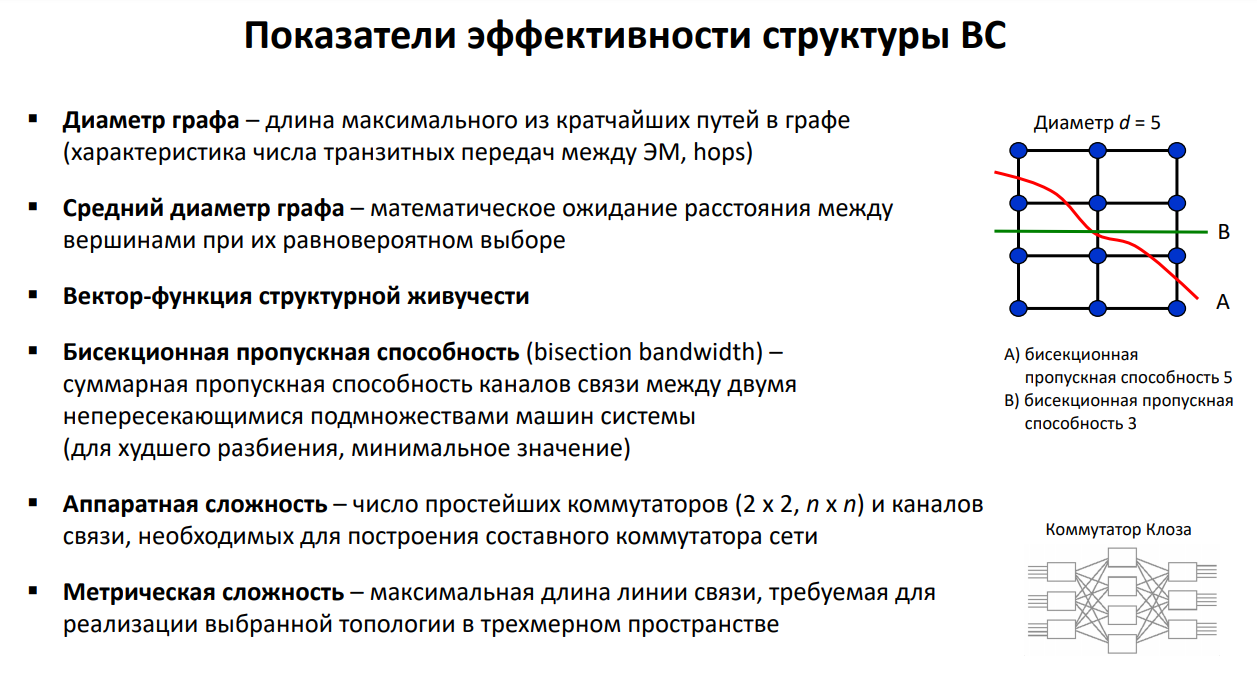






Топология ВС





1. **Показатели эффективности параллельных алгоритмов и программ. Коэффициенты ускорения и накладных расходов. Строгая и слабая масштабируемость параллельных программ. Законы Амдала и Густафсона-Барсиса.**

Показатели эффективности параллельных алгоритмов и программ.

1. Коэффициент ускорения (Speedup)
2. Коэффициент эффективности (Efficiency)
3. Коэффициент накладных расходов
4. Показатель равномерности загруженности параллельных ветвей (процессов, потоков)

**Коэффициент ускорения**

Показывает во сколько раз параллельная программа выполняется на *p* процессорах быстрее последовательной программы при обработке **одних и тех же входных данных размера *n***

***T***(***n***) – время выполнения последовательной программы (sequential program)

***Tp***(***n***) – время выполнения параллельной программы (parallel program) на *p* процессорах

Коэффициент **Sp(n)** ускорения параллельной программ (Speedup):

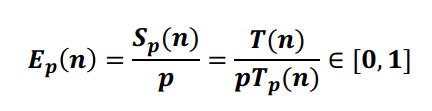
**Sp(n) = (*T***(***n***))/(***Tp***(***n***))

**Sp(n) <= p**

**ЦЕЛЬ РАСПАРАЛЛЕЛИВАНИЯ – ДОСТИЧЬ ЛИНЕЙНОГО УСКОРЕНИЯ, НА МАКСИМАЛЬНО БОЛЬШОМ КОЛИЧЕСТВЕ ПРОЦЕССОРОВ**

За время выполнения последовательной программы берем время лучшего известного алгоритма или алгоритма, который распараллеливаем.

За время выполнения параллельной программы берем время выполнения потока, завершившего работу последним.

Коэффициент эффективности – коэф ускорения, деленный на число проц.  


**Масштабируемость параллельной программы** – показывает, как изменяются показатели производительности программы при изменении числа процессов на конкретной ВС.

**Строгая/сильная масштабируемость** – зависимость коэффициента ускорения от числа p процессов при фиксированном размере n входных данных (n = const)

Показывает, как растут накладные расходы с увеличением p

Цель – минимизировать время решения задачи фиксированного размера

**Слабая масштабируемость (weak scaling)** – зависимость коэффициента ускорения параллельной программы от числа процессов при фиксированном размере входных данных на один процессор (n / p = const)

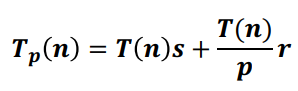
**Закон Амдала**

Пусть имеется последовательная программа c временем выполнения T(n)

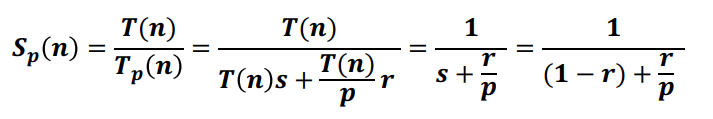
r ∈ [0, 1] – часть программы, которая может быть распараллелена

s = 1 − r – часть программы, которая не может быть распараллелена

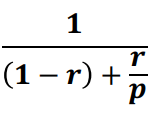
Время выполнения параллельной программы на p процессорах (время каждого потока) складывается из последовательной части s и параллельной r:

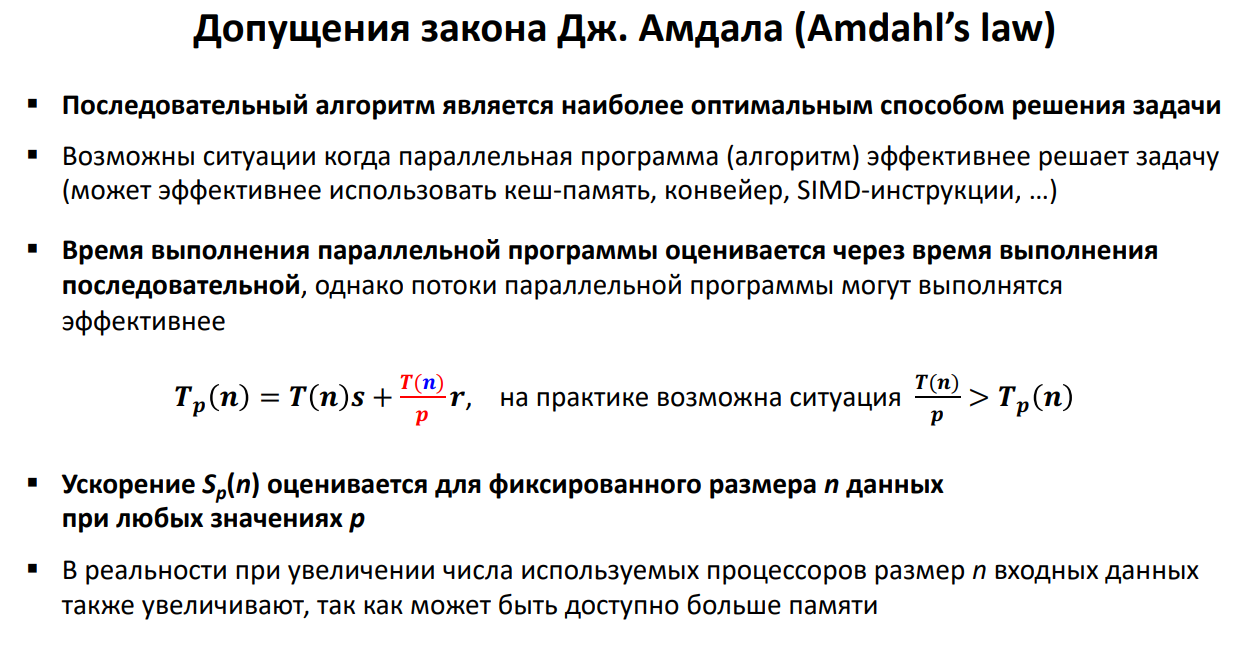
****

Вычислим значение коэффициент ускорения (по определению)

****

Максимальное ускорение Sp программы на p процессорах равняется



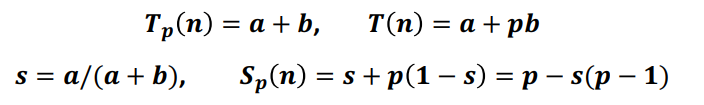


**Закон Густафсона**

Пусть имеется последовательная программа c временем выполнения T(n)

Обозначим S ∈ [0, 1] – часть параллельной программы, которая выполняется последовательно (purely sequential)

Масштабируемое ускорение Sp программы на p процессорах равняется

 Обоснование: пусть a – время последовательной части, b – время параллельной части. .

Время выполнения последовательной программы выражается через время выполнения параллельной.

1. **Стандарт MPI. Двусторонние обмены: блокирующие и неблокирующие. Режимы передачи сообщений. Постоянные запросы (persistent).**

MPI - это стандарт на программный интерфейс коммуникационных библиотек для создания параллельных программ в модели передачи сообщений.

Стандарт определяет интерфейс для языков программирования C и Fortran

Стандарт де-факто для систем с распределенной памятью.

**Двусторонние обмены**

Один процесс инициирует передачу сообщения (Send), другой его принимает (Receive).

Изменение памяти принимающего процесса происходит при его явном участии.

Обмен совмещен с синхронизацией процессов.

**Односторонние**

Только один процесс явно инициирует передачу/прием сообщения из памяти удаленного процесса.

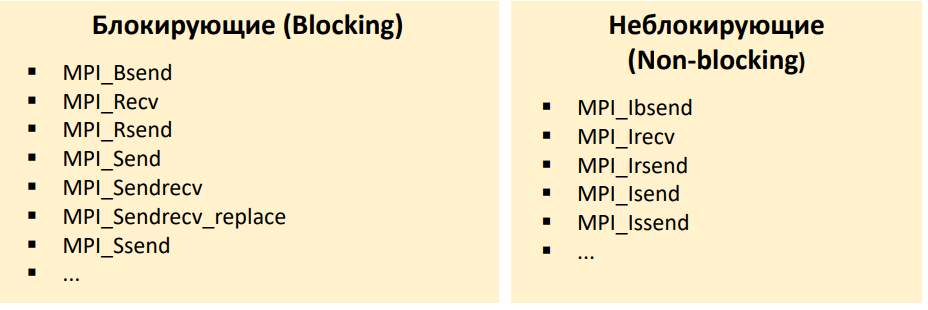
Синхронизация процессов отсутствует.

Двусторонние обмены– участвуют два процесса коммуникатора (send, recv)

Односторонние обмены– участвуют два процесса коммуникатора (без синхронизации процессов, put, get)

Коллективные обмены – участвуют все процессы коммуникатора (one-to-all broadcast, all-to-one gather, all-to-all broadcast)

Двусторонние обмены



Send – передача сообщения. выполняет посылку count элементов типа datatype сообщения с идентификатором tag процессу dest в области связи коммуникатора comm. Переменная buf - это, как правило, массив или скалярная переменная. В последнем случае значение count = 1.

Блокирующие – приостанавливают время выполнения процесса на время приема или передачи сообщения.  
Неблокирующие – выполнение процесса продолжается, а программа в нужный момент может запросить подтверждение завершения приема сообщения

**РЕЖИМЫ ПЕРЕДАЧИ**

Для трех дополнительных коммуникационных режимов используются три дополнительные функции передачи. Коммуникационный режим отмечается одной префиксной буквой: B - для буферизованного, S - для синхронного и R - для режима готовности.

Передача по готовности: сообщение посылается насколько возможно быстро.

Синхронная передача: отправитель посылает запрос на посылку сообщения. Получатель хранит этот запрос. Когда прием инициирован, получатель посылает отправителю разрешение на передачу сообщения и теперь отправитель посылает сообщение.

Стандартная передача: первый протокол может быть использован для коротких сообщений, второй - для длинных.

Буферизованная передача: отправитель копирует сообщение в буфер и затем посылает его с неблокирующим send (используя тот же протокол, что и для стандартного режима).  
  
Постоянные запросы

Постоянные функции привязывают аргументы к дескриптору запроса (persistent request), дальнейшие вызовы операции осуществляется по дескриптору запроса.

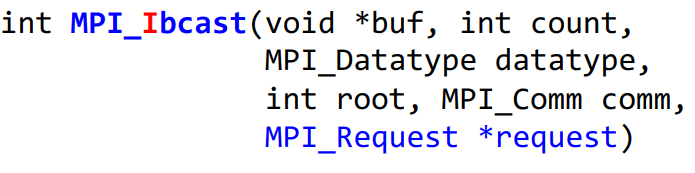
Позволяет сократить время выполнения запроса.

1. **Стандарт MPI. Коллективные операции: блокирующие и неблокирующие. Векторные версии коллективных операций.**

Неблокирующий коллективный обмен – коллективная операция, выход из которой осуществляется, не дожидаясь завершения операций обменов

Пользователю возвращается дескриптор запроса (request), который он может использовать для проверки состояния операции

Цель – обеспечить возможность совмещения вычислений и обменов информацией



Часто возникает потребность разослать некоторую переменную или массив из одного процессора всем остальным. Можно написать с использованием Send/Recv, но MPI\_Bcast удобнее и эффективнее. **Главное отличие** коллективных операций от операций типа точка-точка состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором.

Barrier, alltoall, bcast, gather, allgather, scatter, reduce, allreduce.

Barrier – останавливает работу процесса до тех пор, пока все процессы не сделают это.

Alltoall – каждый процесс делит данные из своего буфера передачи и рассылает фрагменты остальным процессам, одновременно собирая фрагменты посланные другими процессами в свой буфер приема

Bcast – рассылка информации от одного процесса всем остальным

Gather – сборка распределенного по процессам массива в один массив в данном процессе.

Allgather – сборка распределенного по процессам массива в один массив с передачей его всем процессам.

Scatter – разбиение массива и рассылка его фрагментов всем процессам

Reduce – глобальные вычислительные операции над данными расположенными в адресных пространствах разных процессов, с сохранением результата в данном процессе.

Allreduce – reduce с рассылкой результата всем процессам.

Векторные версии имеются у каждой перечисленной операции кроме Bcast. В конце имеют символ “v”.

Имеют более широкие возможности по организации коллективных коммуникаций, снимая ограничения, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов.

1. **Стандарт MPI. Производные типы данных: contiguous, vector, indexed, indexed\_block, strict.**

Основные типы данных: MPI\_

char, double, float, int, long, long\_double, long\_long\_int, short, unsigned, unsigned\_char, unsigned\_long, unsigned\_short, byte, packed. (К началу приделать MPI\_, все заглавными буквами).

Зная заданные здесь типы данных и их число, Вы можете отправлять сообщения с непрерывными (смежными) данными одного и того же типа.

Идея производных типы данных MPI должна обеспечить переносимый и эффективный способ передачи сообщения, состоящего из нескольких несмежных или смешанных типов данных. Производные типы данных MPI обеспечивают более простой, более чистый, более изящный и эффективный способ обработать этот тип данных, которые являются обычными в приложениях. Хотя Вы и можете прожить без производных типов данных, Вы не сможете сделать это легко.

Производные типы данных – это типы данных, которые построены из основных типов данных MPI.

Стандарт MPI определяет общий тип данных как объект, который определяет две вещи:

-последовательность основных типов данных

-последовательность целых (байт) смещений

производные типы данных MPI создаются во время выполнения через вызовы функций из библиотеки MPI.

Вызовы MPI\_Type\_contiguous создают новый тип данных посредством замены существующего типа данных в смежных расположениях.

int MPI\_Type\_contiguous(int count, MPI\_Datatype oldtype,

MPI\_Datatype \*newtype)

count – входная переменная, определяющая количество дублей

oldtype – входная переменная, определяющая указатель на старый тип данных

newtype – выходная переменная, определяющая указатель на новый тип данных

Вызовы MPI\_Type\_vector, подобно вызовам MPI\_Type\_contiguous, создают новый тип данных посредством дублирования существующего; тем не менее, MPI\_Type\_vector позволяет учитывает промежутки в смещении. Такие промежутки – множители степени существующего типа данных.

int MPI\_Type\_vector(int count, int blocklength, int stride,

MPI\_Datatype oldtype, MPI\_Datatype \*newtype)

count – входная переменная, определяющая число блоков

blocklength – входная переменная, определяющая число элементов в каждом блоке

stride – входная переменная, определяющая число элементов между началом последовательных блоков.

Вызовы MPI\_Type\_indexed копируют существующий тип данных в последовательность блоков, где каждый блок – конкатенация существующего типа данных. Каждый блок может содержать различное число копий и иметь различное смещение; однако, все блочные смещения – множители диапазона существующего типа данных

int MPI\_Type\_indexed(int count, int \*array\_of\_blocklengths,

int \*array\_of\_displacements, MPI\_Datatype oldtype,

MPI\_Datatype \*newtype)

count – входная переменная, определяющая число блоков

array\_of\_blocklengths – входная переменная, определяющая число элементов, приходящихся на блок

array\_of\_displacements – входная переменная, определяющая смещение каждого блока в множителях диапазона старого типа данных

1. **Стандарт MPI. Односторонние обмены (one-sided communications).**

Односторонние обмены основаны на механизме удалённого доступа к памяти и позволяют процессу, инициировавшему обмен, самостоятельно задать параметры обмена как для источника, так и для адресата сообщения.

Односторонние обмены используются в том случае, когда процесс "знает", какие данные другого процесса он должен модифицировать, а процесс-адресат сообщения этого не знает.

*Односторонний обмен* возможен, если процесс создаёт "окно" приема, доступное всем остальным процессам.



* base - адрес окна;
* size - размер окна в байтах;
* disp\_unit - масштабный множитель для вычисления смещений;
* info - информационный параметр;
* comm - коммуникатор.

Выходной параметр - win - окно.

Аннулировать окно можно вызовом подпрограммы:

int MPI\_Win\_free(MPI\_Win \*win) MPI\_Win\_free(win, ierror)

Три операции одностороннего обмена являются неблокирующими операциями:

1. MPI\_Put - передача данных от отправителя в окно;
2. MPI\_Get - передача данных из окна отправителю;
3. MPI\_Accumulate - обновление окна получателя.

При вызове подпрограммы MPI\_Get данные копируются в обратном направлении - из памяти адресата в память "источника".

1. **Стандарт MPI. Управление коммуникаторами. Виртуальные топологии процессов. Оптимизация информационных обменов между процессами MPI-программы.**

Коммуникатор – это механизм создания самостоятельного (замкнутого в себе) коммуникационного "мира". Право получить сообщение, отправленное с данным коммуникатором, имеет только процесс, указавший тот же самый коммуникатор.

Средства доступа коммуникатора

MPI\_Comm\_Size – возвращает число процессов в группе коммуникатора

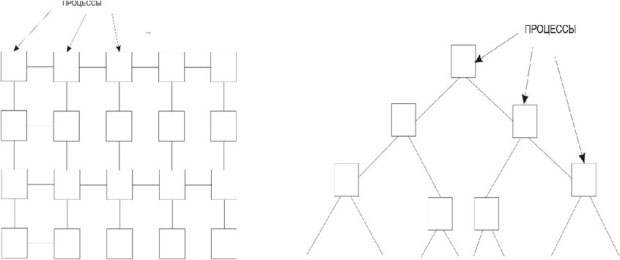
MPI\_Comm\_Rank – возвращает ранг вызываемого процесса в группе коммуникатора

MPI\_Comm\_Compare – сравнивает два коммуникатора

В *MPI* *топология* представляет собой механизм сопоставления процессам, принадлежащим группе, альтернативных по отношению к обычной схемам нумерации.

Топологией в данном случае называют структуру соединений - линий и узлов сети без учета характеристик самих этих узлов. Узлами здесь являются процессы, соединениями - каналы обмена сообщениями, а сетью мы, фактически, называем все процессы, входящие в состав параллельной программы.

В *MPI* существуют два типа топологии: **декартова** *топология* — прямоугольная решетка произвольной размерности и *топология* **графа**



Для того, чтобы связать структуру декартовой решетки с коммуникатором MPI\_COMM\_WORLD, необходимо задать следующие параметры:

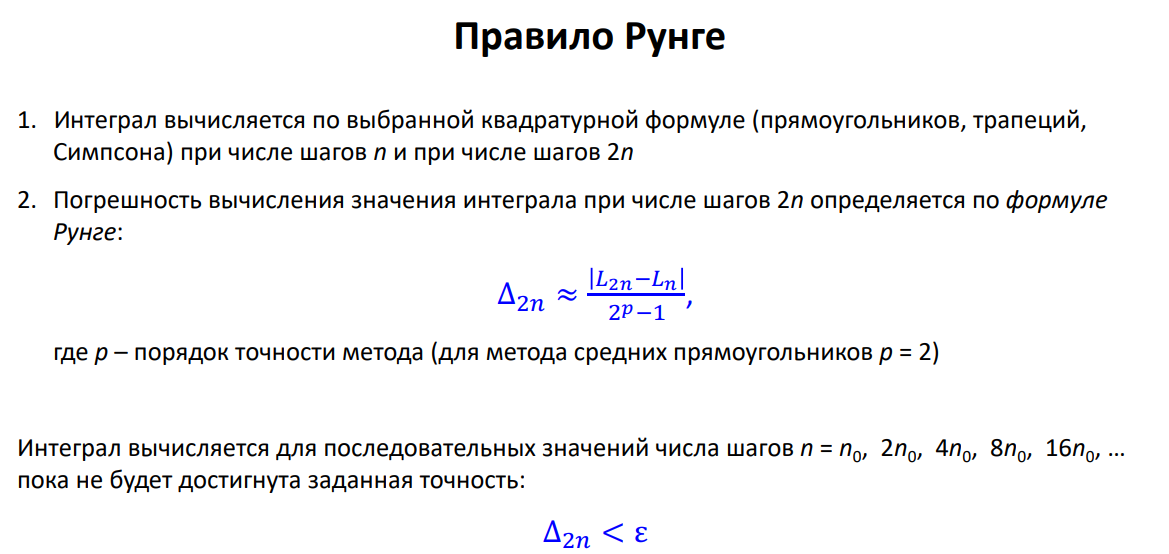
* **размерность** решетки (значение 2, например, соответствует плоской решетке);
* **размер** решетки вдоль каждого измерения (размеры {10, 5}, например, соответствуют прямоугольной плоской решетке, протяженность которой вдоль оси x составляет 10 узлов-процессов, а вдоль оси y - 5 узлов);
* **граничные условия** вдоль каждого измерения (решетка может быть периодической, если процессы, находящиеся на противоположных концах ряда, взаимодействуют между собой).

На основе анализа работы параллельной программы составляется информационный граф программы, который используется эвристическим алгоритмом для эффективного распределения её процессов по процессорным ядрам с целью минимизации суммарного времени выполнения обменов между ветвями MPI-программы. Для оптимизации отображения на первом этапе выполняется анализ информационного графа обмена сообщений между процессами данной MPI-программы. Идея метода оптимизации отображения параллельной программы заключается в разбиении информационного графа программы на непересекающиеся подмножества интенсивно обменивающихся процессов и привязке этих подмножеств к узлам/процессорам, соединённым наиболее быстрыми каналами связи. Разбиение выполняется для минимизации суммы рёбер, соединяющих разные подмножества разбиения. Разбиение рекурсивно выполняется сначала для уровня описывающего обмены между вычислительными узлами, а затем для уровня описывающего обмены внутри каждого из узлов в случае, если внутри узла установлено несколько процессоров. Целью такого разбиения является минимизация времени выполнения программы. Задача оптимального отображения процессов MPIпрограммы является NP-полной, так как её можно свести к задаче разбиения графов. Для её решения целесообразно использовать эвристические алгоритмы дающие решения, близкие к оптимальным.

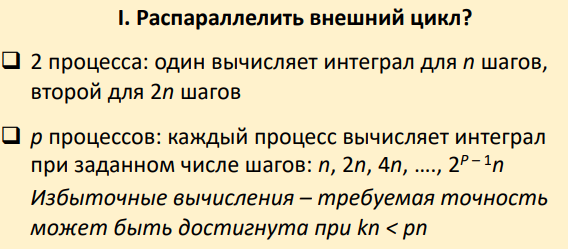
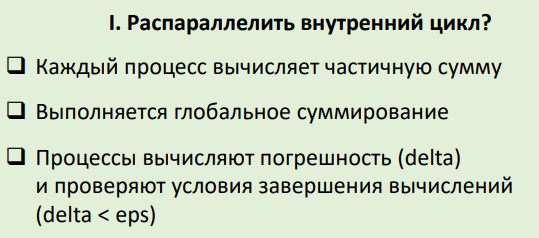
1. **Параллельное численное интегрирование. Параллельная MPI-реализация метода средних прямоугольников с апостериорной оценкой погрешности по правилу Рунге. Параллельная MPI-реализация метода Монте-Карло.**

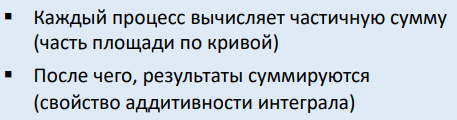
Численное интегрирование (numerical integration) – вычисление значения определенного интеграла

Можем использовать формулу средних прямоугольников и Монте-Карло

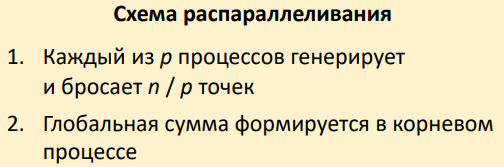
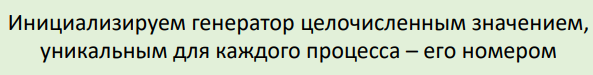


В 1 случае можем распараллелить внешний или внутренний цикл



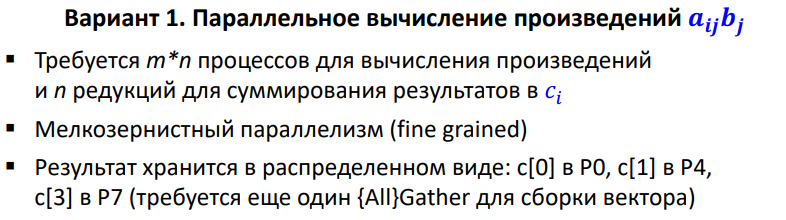
В 2 случае

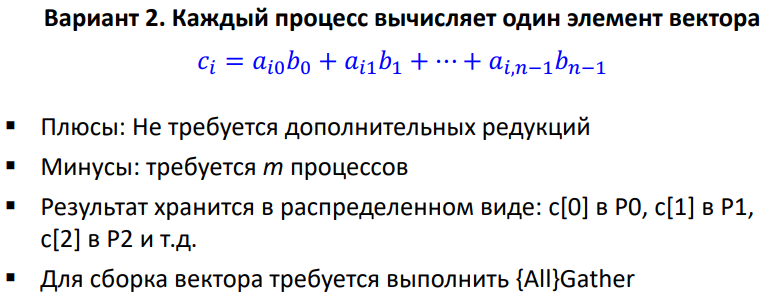
 

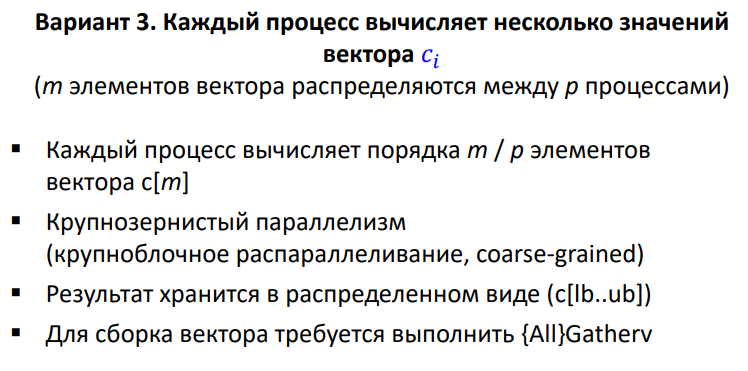
2 для того, чтобы у каждого процесса генерировались свои точки

1. **Матричные вычисления. Параллельная MPI-реализация умножения матрицы на вектор. Распределенное хранение матрицы в памяти вычислительных узлов. Анализ потребления памяти при различных схемах хранении матрицы в памяти параллельных процессов. Анализ ускорения.**

Массивы A, B, C – хранятся во всех процессах

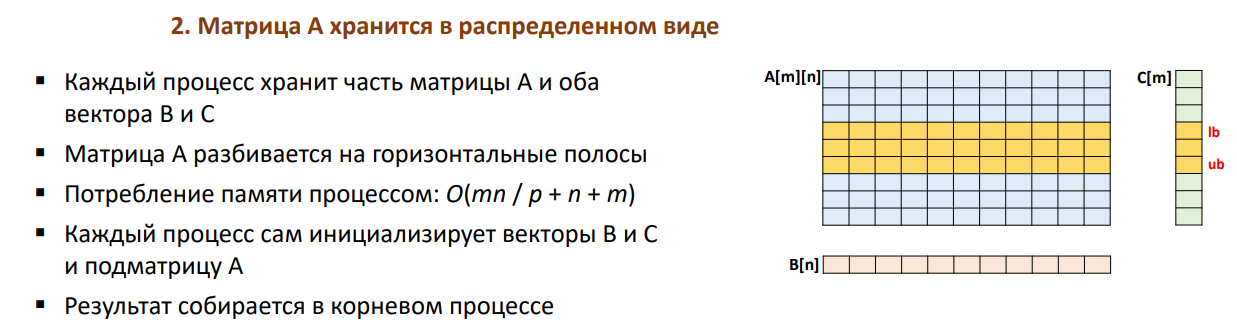




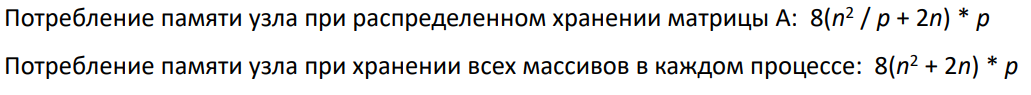


Матрица А хранится в распределенном виде





При распределенном хранении матрицы A[n][n] увеличивается эффективность использования узла (степень параллелизма узла)



1. **Прямые методы решения систем линейных алгебраических уравнений. Параллельная MPI-реализация метода Гаусса. Анализ схем декомпозиции матрицы коэффициентов: смежные горизонтальные полосы, циклическое распределение строк.**

Метод Гаусса (Gaussian elimination, row reduction) – метод последовательного исключения переменных § Шаги метода Гаусса:

1. Прямой ход (elimination) – СЛАУ приводится к треугольной форме путем элементарных преобразований (вычислительная сложность O(n3))

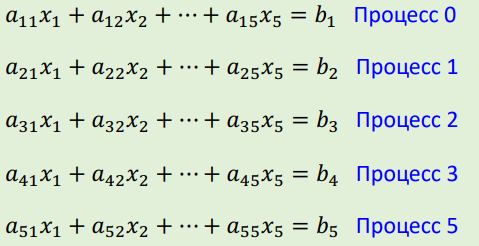
2. Обратный ход (back substitution) – начиная с последних, находятся все неизвестные системы (вычислительная сложность O(n2))

Параллельная версия в1

Каждый процесс хранит в своей памяти одну строку матрицы – одно уравнение

Требуется n процессов

Процессы неравномерно загружены вычислениями – после передачи строки они выбывают из вычислений

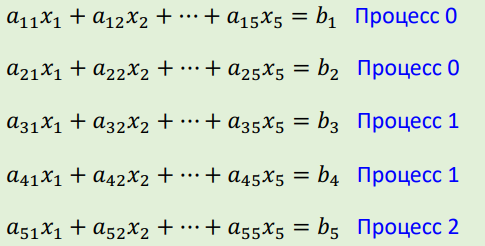


В2

Каждый процесс хранит в своей памяти горизонтальную полосу из n / P смежных строк

Требуется P процессов

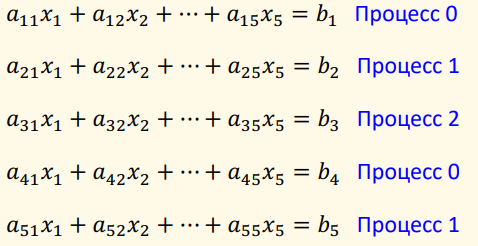
Процессы неравномерно загружены вычислениями – после передачи строки они выбывают из вычислений



В3 – циклическое распределение строк матрицы

Каждый процесс хранит в своей памяти порядка n / P строк, векторы b[n] и x[n]

Строки распределены по циклической схеме для выравнивания вычислительной загрузки процессов

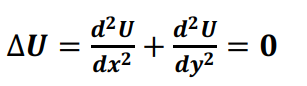


1. **Параллельные сеточные вычисления. Решение стационарного двумерного уравнения Лапласа. Метод Якоби. Анализ схем геометрической декомпозиции расчетной области: горизонтальные полосы (1D), подматрицы (2D). Реализация в MPI и анализ эффективности.**

С течением времени в теле устанавливается некоторое не зависящее от времени распределение температуры (тепловое состояние выходит на стационарный режим)

Распределение температуры в таком случае описывается уравнением теплопроводности

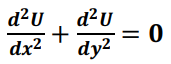
Стационарное двумерное уравнение Лапласа (Laplace equation)



Функция U(x, y) – неизвестный потенциал (теплота)

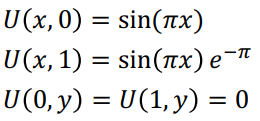
Задано уравнение (1), значения функции U(x, y) на границах расчетной 2D-области

Требуется найти значение функции U(x, y) во внутренних точках расчетной 2Dобласти



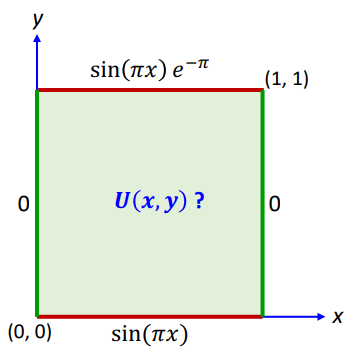
Расчётная область (domain) – квадрат [0, 1] x [0, 1]

Граничные условия (boundary conditions):



Для данной задачи известно аналитическое решение





Метод последовательных итераций Якоби (Jacobi)

Вычисляем новое значение в каждой точке [i, j] сетки – среднее из предыдущих значений четырех ее соседних точек (схема «крест»), результат записываем в новую сетку (массив)

На следующей итерации текущей делаем новую сетку предыдущей итерации

Заканчиваем итерационный процесс, если разность между каждым текущим и предыдущим значениями по модулю не больше EPSILON

**Параллельная версия метода Якоби (1D)**

Одномерная (1D) декомпозиция расчётной области на горизонтальные полосы § Каждому процессу назначается ny / p строк расчетной сетки § Сетка хранится в памяти в распределенном виде § **Проблема** – для расчета значений некоторых точек требуются данные соседних полос, которые находятся в памяти других процессов

На каждой итерации перед вычислением значений в точках обмениваемся пограничными строками с соседними процессами

Анализ времени выполнения информационных обменов при одномерной декомпозиции § Время передачи сообщения размером m байт: t(m) = a + bm § Пусть сетка квадратная и содержит n строк и столбцов § На каждой итерации выполняется два send и два recv:

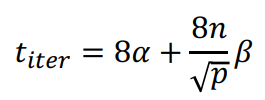


Минус: время обменов не зависит от числа p процессов – при любом числе процессов время будет одним и тем же (не будет убывать)

**Параллельная версия метода Якоби (2D)**

Двумерная (2D) декомпозиция расчётной области на прямоугольные области § Каждому процессу назначается подмассив [ny / p, nx / p] строк расчетной сетки § Сетка хранится в памяти в распределенном виде § Проблема – для расчета значений некоторых точек требуются данные соседних полос, которые находятся в памяти других процессов

Анализ времени выполнения информационных обменов при двумерной декомпозиции § Время передачи сообщения размером m байт: t(m) = a + bm § Пусть сетка квадратная и содержит n строк и столбцов, а число p процессов – степень числа 2 § На каждой итерации выполняется четыре send и четыре recv:



Плюс: время выполнения обменов уменьшается с ростом числа p процессов!

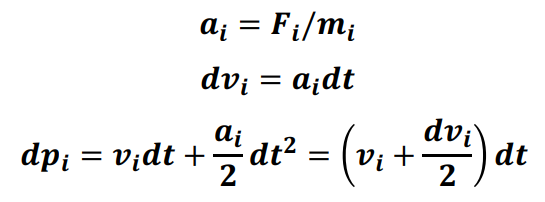
1. **Гравитационная задача N-тел (N-body). Прямой метод решения. Анализ способов распараллеливания прямого метода: шаблон «производитель-потребитель» (masterslave), схема «пульсации», конвейер.**

Задача N-тел (nbody)

§ Сила, действующая на тело 1, равна сумме сила действующих на тело 1 со стороны тел 2, 3 и 4 

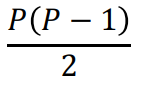
Гравитационные силы, действующие на тела, вызывают их ускорение и перемещение

Интегрирование уравнений движения (схема leapfrog):



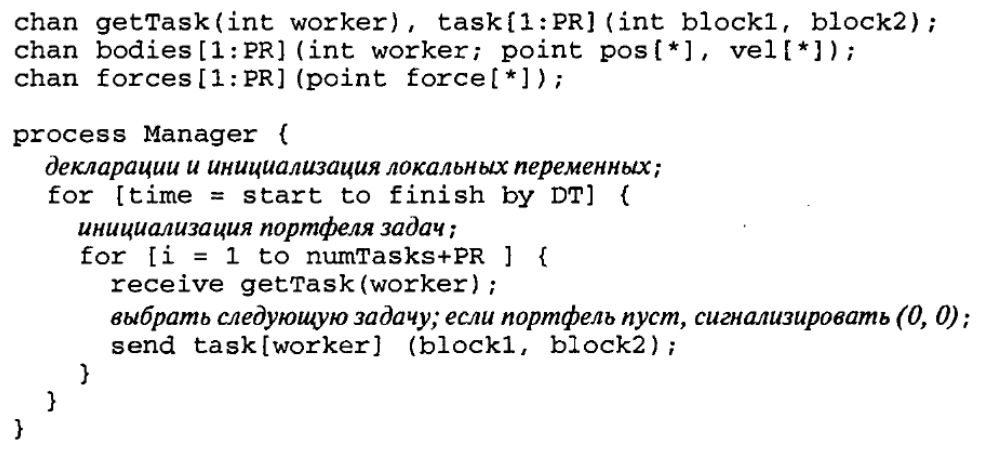
**Шаблон «производитель-потребитель»**

Имеется P процессов § Множество тел разбивается на P блоков, в каждом из которых n / P тел § Образуются пары (i, j) для всех возможных комбинация блоков § Число пар блоков

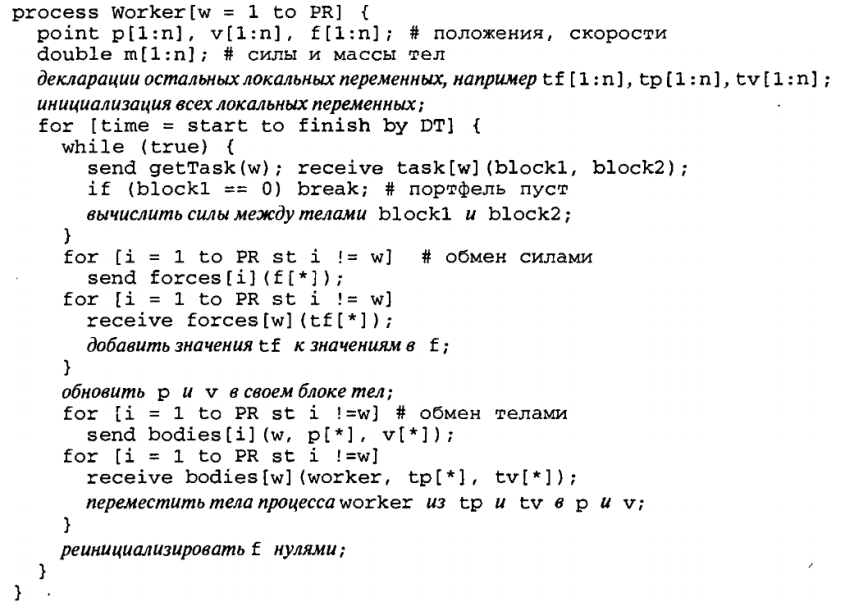
****

Каждая пара – это элементарная задача § Производитель (главный процесс, master process) – раздает элементарные задачи рабочим процессам § Потребители (рабочие процессы, slave) – запрашивают у производителя задачи и выполняют вычисление сил между телами блоков i и j § После вычисления сил, процессы обмениваются силами и перемещают тела

**Производитель**

****

**Потребитель**

****