

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания

по выполнению лабораторных работ
по дисциплине «МНОГОПОТОЧНОЕ
ПРОГРАММИРОВАНИЕ» для студентов направления
подготовки 09.03.03 Прикладная информатика

(ЭЛЕКТРОННЫЙ ДОКУМЕНТ)

Ставрополь 2026

Оглавление	
Лабораторная работа №1. Основные функции MPI.....	3
Лабораторная работа №2. Виртуальные топологии.....	69
Лабораторная работа №3. Основные директивы OpenMP	74
Лабораторная работа №4. Решение систем линейных алгебраических уравнений итерационными методами.....	100
Лабораторная работа №5. Решение систем линейных алгебраических уравнений методом Гаусса.....	108
Лабораторная работа №6. Решение задачи Пуассона методом Зейделя в трехмерной области	118
Лабораторная работа №7. Решение систем линейных алгебраических уравнений итерационными методами.....	124
Лабораторная работа №8. Программирование многопоточных приложений. POSIX Threads.	131
Лабораторная работа №9. Редукционные операции. Решение СЛАУ методом простой итерации.....	145
Лабораторная работа №10. Решение СЛАУ методом Гаусса	156

Лабораторная работа №1. Основные функции MPI

1. Цель и содержание

Цель лабораторной работы: дать представление о построении простых параллельных программ на языке параллельного программирования MPI; представление о параллельных программах, настраиваемых на размер вычислительной системы, как на параметр; закрепить практическое освоение функций парных взаимодействий между ветвями параллельной программы.

2. Теоретическая часть

Система MPI является основным средством программирования таких современных высокопроизводительных мультимпьютеров, как Silicon Graphics Origin 2000, Cray T3D, Cray T3E, IBM SP2 и многих других. MPI работает на самых разных архитектурах мультимпьютеров, как с распределенной памятью, так и с разделяемой памятью. Кроме того, MPI работает на сетях компьютеров (кластерах) однородных и гетерогенных. Количество компьютеров в сети может быть от одного и более. Важнейшей особенностью MPI является то, что пользователь при написании своих параллельных программ не должен учитывать все эти архитектурные особенности конкретных мультимпьютеров, поскольку MPI предоставляет пользователю виртуальный мультимпьютер с распределенной памятью и с виртуальной сетью связи между виртуальными компьютерами. Пользователь заказывает количество компьютеров, необходимых для решения его задачи, и определяет топологию связей между этими компьютерами. MPI реализует этот заказ на конкретной физической системе. Ограничением является объем оперативной памяти физического мультимпьютера. Таким образом пользователь работает в виртуальной среде, что обеспечивает переносимость его параллельных программ. Система MPI представляет собой библиотеку средств параллельного программирования для языков C и Fortran 77.

Методы распараллеливания и модели программ, поддерживаемые MPI

Одной из целей, преследуемых при решении задач на вычислительных системах, в том числе и на параллельных, – является эффективность. Эффективность параллельной программы существенно зависит от соотношения времени вычислений ко времени коммуникаций между компьютерами (при обмене данными). И чем меньше в процентном отношении доля времени, затраченного на коммуникации, в общем времени вычислений, тем больше эффективность. Для параллельных систем с передачей сообщений оптимальное соотношение между вычислениями и коммуникациями обеспечивают методы крупнозернистого распараллеливания, когда параллельные алгоритмы строятся из крупных и редко взаимодействующих блоков [2–8]. Задачи линейной алгебры, задачи, решаемые сеточными методами и многие другие, достаточно эффективно *распараллеливаются крупнозернистыми методами*.

MPMD - модель вычислений. MPI - программа представляет собой совокупность автономных процессов, функционирующих под управлением своих собственных программ и взаимодействующих посредством стандартного набора библиотечных процедур для передачи и приема сообщений. Таким образом, в самом общем случае MPI - программа реализует MPMD - модель программирования (Multiple program - Multiple Data).

SPMD - модель вычислений. Все процессы исполняют в общем случае различные ветви одной и той же программы. Такой подход обусловлен тем обстоятельством, что задача может быть достаточно естественным образом разбита на подзадачи, решаемые по одному алгоритму. На практике чаще всего встречается именно эта модель программирования (Single program - Multiple Data) [1,12].

Последнюю модель иначе можно назвать моделью *распараллеливания по данным*. Кратко, суть этого способа заключается в следующем. Исходные данные задачи распределяются по процессам (ветвям параллельного алгоритма), а алгоритм является одним и тем же во всех процессах, но

действия этого алгоритма распределяются в соответствии с имеющимися в этих процессах *данными*. Распределение действий алгоритма заключается, например, в присвоении разных значений переменным одних и тех же циклов в разных ветвях, либо в исполнении в разных ветвях разного количества витков одних и тех же циклов и т.п. Другими словами, процесс в каждой ветви следует различными путями выполнения на той же самой программе.

Указанные модели, помимо поддержки на языковом уровне, поддерживаются архитектурами таких самых современных суперкомпьютеров как: ASCI RED (более 9000 Pentium PRO/200 объединены в единую систему, имеет быстродействие около 3,2 Tflops), и ASCI WAIT (8192 - , имеет быстродействие 12,2 Tflops), Cray T3D, Cray T3E, IBM SP2.

В первой части дано описание системы параллельного программирования MPI. Даны операторы компиляции и запуска C-программ, программные средства задания системных взаимодействий, виртуальные топологии. В каждом подразделе приводятся примеры программ, закрепляющие понимание и усвоение материала. Эти же примеры могут использоваться как образцы для написания новых программ.

Во второй части даны четыре лабораторных работы, построенные как последовательность шагов по изучению программных средств параллельного программирования и освоению основных навыков написания параллельных программ.

1. Виртуальные топологии

В этом пункте описывается механизм виртуальных топологий MPI. Под виртуальной топологией здесь понимается программно реализуемая топология в виде конкретного графа, например: кольцо, решетка, тор, звезда, дерево и вообще произвольно задаваемый граф на существующей физической топологии. Виртуальная топология обеспечивает очень удобный механизм наименования процессов, связанных коммутатором, и является мощным средством отображения процессов на оборудование системы.

Виртуальная топология в MPI может задаваться только в группе процессов, объединенных коммуникатором.

Нужно различать виртуальную топологию процессов и топологию основного, физического оборудования. Механизм виртуальных топологий значительно упрощает и облегчает написание параллельных программ, делает программы легко читаемыми и понятными. Пользователю при этом не нужно программировать схему физических связей процессоров, а только программировать схему виртуальных связей между процессами. Отображение виртуальных связей на физические связи осуществляет система, что делает параллельные программы машинно-независимыми и легко переносимыми. Функции в этой главе осуществляют только машинно-независимое отображение.

1.1. Функции создания декартовых топологий

Функция, конструирующая декартову топологию

MPI_CART_CREATE используется для описания декартовой структуры произвольного измерения. Для каждого направления координаты определяется, является ли структура процесса периодической или нет.

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder,
comm_cart)

IN comm_old входной (старый) коммуникатор

IN ndims количество измерений в декартовой топологии

IN dims целочисленный массив размером ndims,
определяющий количество процессов в каждом
измерении

IN periods массив размером ndims логических значений,
определяющих периодичность (true) или
нет (false) в каждом измерении

IN reorder ранги могут быть перенумерованы (true) или
нет (false)

OUT comm_cart коммуникатор новой (созданной) декартовой
топорлогии

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,  
int *periods, int reorder, MPI Comm *comm_cart)
```

MPI_CART_CREATE возвращает управление новому коммуникатору, к которому присоединена информация декартовой топологии. Эта функция коллективная. Как в случае с другими коллективными функциями, вызов этой функции должен быть синхронизован во всех процессах. Если reorder = false, тогда номер каждого процесса в новой группе идентичен ее номеру в старой группе. Иначе, функция может переупорядочивать процессы (возможно чтобы выбрать хорошее отображение виртуальной топологии на физическую топологию). Если полный размер декартовой сетки меньше чем размер группы comm_old, то некоторые процессы возвращают MPI_COMM_NULL, по аналогии с MPI_COMM_SPLIT. Запрос ошибочен, если он определяет сетку, которая является большей, чем размер группы comm_old.

0	1	2	3
0,0)	0,1)	0,2)	0,3)
4	5	6	7
1,0)	1,1)	1,2)	1,3)
8	9	10	11

2,0)	2,1)	2,2)	2,3)

Рис. 1.2. Связь между рангами и декартовыми координатами для 3x4 2D топологии. Верхний номер в каждой клетке - номер процесса, а нижнее значение (строка, столбец) - координаты.

Декартова функция задания решетки

Для декартовой топологии, функция `MPI_DIMS_CREATE` помогает пользователю выбрать сбалансированное распределение процессов по направлению координат, в зависимости от числа процессов в группе, и необязательных ограничений, которые могут быть определены пользователем. Одно возможное использование этой функции это разбиение всех процессов (размер группы `MPI_COMM_WORLD`) в N-мерную топологию.

`MPI_DIMS_CREATE(nnodes, ndims, dims)`

IN nnodes количество узлов в решетке

IN ndims размерность декартовой топологии

INOUT dims целочисленный массив, определяющий
количество узлов в каждой размерности

`int MPI_Dims_create(int nnodes, int ndims, int *dims)`

Элементы в массиве `dims` представляют описание декартовой решетки с размерностями `ndims` и общим количеством узлов `nnodes`. Размерности устанавливаются так, чтобы быть близко друг к другу насколько возможно, используя соответствующий алгоритм делимости. Пользователь может ограничивать действие этой функции, определяя элементы массива `dims`. Если в `dims[i]` уже записано число, то функция не будет изменять количество узлов в измерении `i`; функция модифицирует только нулевые элементы в массиве, т.е. где `dims[i] = 0`. Отрицательные значения элементов `dims(i)` ошибочны. Ошибка будет выдаваться, если `nnodes` не кратно $\prod \text{dims}[i]$.

(i,dims[i]≠0)

Для dims[i] установленных функцией, dims[i] будут упорядочены в монотонно уменьшающемся порядке. Массив dims подходит для использования, как вход к функции MPI_CART_CREATE. Функция MPI_DIMS_CREATE локальная. Отдельные типовые запросы показываются в примере 1.1.

ПРИМЕР 1.2

dim s пер ед вызовом	функции	dim s пос ле возврата
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3 ,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3 ,1)
(0,3 ,0)	MPI_DIMS_CREATE(6, 3, dims)	ош ибка

Декартовы информационные функции

Если декартова топология создана, может возникнуть необходимость запросить информацию относительно этой топологии. Эти функции даются ниже и все они локальные.

MPI_CARTDIM_GET(comm, ndims)

IN comm коммуникатор с декартовой топологией

OUT ndims размерность декартовой топологии

int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

MPI_CARTDIM_GET возвращает число измерений декартовой топологии, связанной с коммуникатором comm. Коммуникатор с топологией на рис. 1.2 возвратил бы ndims = 2.

MPI_CART_GET(comm, maxdims, dims, periods, coords)

IN comm коммуникатор с декартовой топологией

IN maxdims максимальный размер массивов dims, periods, и coords в вызывающей программе

OUT dims массив значений, указывающих количество процессов в каждом измерении

OUT periods массив значений, указывающих периодичность (true/false) в каждом измерении

OUT coords координаты вызвавшего процесса в декартовой топологии

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,  
                 int *periods, int *coords)
```

MPI_CART_GET возвращает информацию относительно декартовой топологии, связанной с коммуникатором comm. maxdims должен быть, по крайней мере, равен ndims, как возвращает MPI_CARTDIM_GET. Для примера на рис. 1.2, dims = (3,4), а в процессе 6 функция возвратит coords = (1,2).

Декартовы функции транслирования

Функции, приведенные в этом пункте переводят наиз ранга в декартовы координаты топологии. Эти запросы локальные.

MPI_CART_RANK(comm, coords, rank)

IN comm коммуникатор с декартовой топологией

IN coords целочисленный массив, определяющий координаты нужного процесса в декартовой топологии

OUT rank ранг нужного процесса

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

Для группы процессов с декартовой структурой функция MPI_CART_RANK переводит логические координаты процессов в номера. Эти номера процессы используют в парных взаимодействиях между процессами. coords - массив размером ndims как возвращает MPI_CARTDIM_GET. Для примера на рис. 1.2 процесс с coords = (1,2) возвратил бы rank = 6. Для измерения i с periods(i) = true, если координата, coords(i), находится вне диапазона, то есть coords(i) < 0 или coords(i) >= dims(i), она перемещается назад к интервалу 0 <= coords(i) < dims(i) автоматически. Если топология на рис. 1.2 периодическая в обеих размерностях, то процесс с coords = (4,6) также возвратил бы rank = 6. Для непериодических размерностей диапазон вне координат ошибочен.

MPI_CART_COORDS(comm, rank, maxdims, coords)

IN comm коммуникатор с декартовой топологией

IN rank ранг процесса в топологии comm

IN maxdims максимальный размер массивов dims, periods, и
coords в вызывающей программе

OUT coords целочисленный массив, определяющий координаты
нужного процесса в декартовой топологии

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int
*coords)

MPI_CART_COORDS – переводит номер процесса в координаты
процесса в топологии. Это обратное отображение MPI_CART_RANK.

maxdims можно взять, например, равным ndims, возвращенным MPI_CARTDIM_GET. Для примера на рис. 1.2, процесс с rank = 6 возвратил бы coords = (1,2).

Декартова функция смещения

Если в декартовой топологии, используется функция MPI_SENDRECV (см. далее п. 3.2) для смещения данных вдоль направления какой либо координаты, то входным аргументом MPI_SENDRECV берется номер процесса source (процесса источника) для приема данных, и номер процесса dest (процесса назначения) для передачи данных. Операция смещения в декартовой топологии определяется координатой смещения и размером шага смещения (положительным или отрицательным). Функция MPI_CART_SHIFT возвращает информацию для входных спецификаций, требуемых в вызове MPI_SENDRECV. Функция MPI_CART_SHIFT локальная.

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

IN comm коммуникатор с декартовой топологией

IN direction номер измерения (в топологии), где делается смещение

IN disp направление смещения (> 0: смещение в сторону увеличения номеров координаты direction, < 0: смещение в сторону уменьшения номеров координаты direction)

OUT rank_source ранг процесса источника

OUT rank_dest ранг процесса назначения

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,  
                  int *rank_source, int *rank_dest)
```

Аргумент direction указывает измерение, в котором осуществляется смещение данных. Измерения маркируются от 0 до ndims-1, где ndims -

число размерностей. `disp` указывает направление и величину смещения. Например, в топологии "линейка" или "кольцо" с $N \geq 4$ процессами для процесса с номером 1 при `disp = +1` `rank_source = 0`, а `rank_dest = 2`; при `disp = -1` `rank_source = 2`, а `rank_dest = 0`. Для этого же процесса 1 при `disp = +2` `rank_source = N-1` для "кольца" и `MPI_PROC_NULL` для "линейки", а `rank_dest = 3` для обоих структур; при `disp = -2` `rank_source = 3` для обоих структур, а `rank_dest = N-1` для "кольца" и `MPI_PROC_NULL` для "линейки".

В зависимости от периодичности декартовой топологии в указанном направлении координат, `MPI_CART_SHIFT` обеспечивает идентификаторы `rank_source` и `rank_dest` для кольцевого или не кольцевого смещения данных. Это имеющие силу входные аргументы к функции `MPI_SENDRECV`. Ни `MPI_CART_SHIFT`, ни `MPI_SENDRECV` не коллективные функции. Не требуется, чтобы все процессы в декартовой решетке одновременно вызвали `MPI_CART_SHIFT` с теми же самыми `direction` и `disp` аргументами, но только тот процесс, который посылает соответственно, получает в последующих запросах к `MPI_SENDRECV`.

Декартова функция разбиения

`MPI_CART_SUB(comm, remain_dims, newcomm)`

IN `comm` communicator для декартовой топологии

IN `remain_dims` `i`-й элемент `remain_dims` определяет
соответствующую `i`-ю размерность включаемую в
подрешетку (`true`) или не включаемую (`false`)

OUT `newcomm` communicator созданных подрешеток

`int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)`

Если декартова топология была создана функцией `MPI_CART_CREATE`, то может использоваться функция `MPI_CART_SUB` для разбиения группы, связанной коммуникатором, на подгруппы, которые

формируют декартовы подрешетки меньшей размерности, и строить для каждой такой подгруппы коммунитор, связанный с подрешеткой декартовой топологии. Этот запрос коллективный.

ПРИМЕР 1.4

Предположим, что `MPI_CART_CREATE(...,comm)` определяет (2 x 3 x 4) решетку. Допустим `remain_dims=(true, false, true)`. Тогда запрос к

`MPI_CART_SUB(comm, remain_dims, comm_new)`

создаст три коммунитора каждый с восьмью процессами 2 x 4 в декартовой топологии. Если `remain_dims=(false, false, true)`, то запрос к `MPI_CART_SUB(comm, remain_dims, comm_new)` создаст шесть непересекающихся коммуниторов, каждый с четырьмя процессами, в одномерной декартовой топологии.

ПРИМЕР 1.5

```
...
int rem[3], dims[3], period[3], reord, i, j;
MPI_Comm comm_3D, comm_2D[3], comm_1D[3];
...
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, period, reord,
&comm_3D);
...
for(i = 0; i < 3; i++)
{ for(j = 0; j < 3; j++)
    rem[j] = (i != j);
    MPI_Cart_sub(comm_3D, rem, &comm_2D[i]);
}
for(i = 0; i < 3; i++)
{ for(j = 0; j < 3; j++)
    rem[j] = (i == j);
    MPI_Cart_sub(comm_3D, rem, &comm_1D[i]);
}
```

1.2. Функции создания топологии графа

Функция построения графа

`MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder,`
`comm_graph)`

IN `comm_old` входной коммуникатор

IN `nnodes` количество узлов в графе

IN `index` целочисленный массив для описания узлов графа

IN `edges` целочисленный массив для описания ребер графа

IN `reorder` переупорядочить ранги (true) или нет (false)

OUT `comm_graph` коммуникатор с топологией построенного графа

`int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index,`
`int *edges, int reorder, MPI_Comm *comm_graph)`

`MPI_GRAPH_CREATE` возвращает новый коммуникатор, к которому присоединена информация топологии графа. Если `reorder=false`, то ранг каждого процесса в новой группе идентичен ее рангу в старой группе. Иначе, функция может переупорядочивать процессы. Если размер, `nnodes`, графа меньше чем размер группы `comm_old`, то некоторые процессы возвращают `MPI_COMM_NULL`. Запрос ошибочен, если он определяет граф, который является большим, чем размер группы определяемой `comm_old`. Эта функция коллективная.

Три параметра `nnodes`, `index` и `edges` определяют структуру графа. `nnodes` - число узлов графа. Узлы маркируются от 0 до `nnodes-1`. *i*-й элемент массива `index` хранит общее число соседей первых *i* узлов графа. Списки соседних узлов 0,1,...,`nnodes-1` хранятся в массиве `edges`. Массив `edges` - сжатое представление списков ребер. Общее число элементов в `index` равно `nnodes`, и общее число элементов в `edges` равно числу ребер графа.

Определения аргументов `nnodes`, `index`, и `edges` иллюстрируются ниже в примере 1.5.

ПРИМЕР 1.5

Предположим, что имеются четыре компьютера 0, 1, 2, 3 со следующей матрицей смежности:

мп.	Ко	Со	се
	ди		
	0	1, 3	
	1	0	
	2	3	
	3	0, 2	

Тогда, входные аргументы будут иметь следующие значения:

`nnodes = 4`

`index = (2, 3, 4, 6)`

`edges = (1, 3, 0, 3, 0, 2)`

Функции запроса графа

Если топология графа установлена, может быть необходимо запросить информацию относительно топологии. Эти функции даются ниже и все - локальные вызовы.

`MPI_GRAPHDIMS_GET(comm, nnodes, nedges)`

IN `comm` коммуникатор группы, связанной с графом

OUT `nnodes` количество узлов в графе

OUT `nedges` количество ребер в графе

`int MPI_Graphdims_get(MPI_Comm_comm, int *nnodes, int *nedges)`

`MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)`

INTEGER COMM, NNODES, NEDGES, IERROR

`MPI_GRAPHDIMS_GET` возвращает число узлов и число ребер в графе. Число узлов идентично размеру группы, связанному с `comm`. `nnodes` и

nedges используются, чтобы снабдить массивы надлежащего размера для index и edges, соответственно, в функции MPI_GRAPH_GET. MPI_GRAPHDIMS_GET возвратил бы nnodes = 4 и nedges = 6 в примере 1.5.

MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)

IN comm коммуникатор группы со структурой графа

IN maxindex длина вектора index в вызвавшей программе

IN maxedges длина вектора edges в вызвавшей программе

OUT index массив чисел, определяющих узлы графа

OUT edges массив чисел, определяющих узлы графа

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges,
 int *index, int *edges)

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX,
EDGES, IERROR)

INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*),
IERROR

MPI_GRAPH_GET возвращает index и edges какими они были в MPI_GRAPH_CREATE. maxindex и maxedges по крайней мере такие же как nnodes и nedges, соответственно, какие возвращаются функцией MPI_GRAPHDIMS_GET.

2. Парные взаимодействия

Существует много вариантов парных операций. Варианты операций необходимы для оптимизации взаимодействий в каждом конкретном случае при описании параллельного алгоритма задачи. Здесь рассматриваются *блокированные*, *неблокированные* и *синхронные* взаимодействия, реализуемые соответствующими функциями. Имеются все передающие функции, указанных выше типов. Принимающих функций только две: *блокированная* и *неблокированная*. Каждая принимающая функция из этих

типов совместима со всеми передающими функциями и наоборот. Блокированные и неблокированные взаимодействия являются асинхронными.

Блокированная передача/прием данных. Блокированная передающая функция не возвращает управление, пока данные сообщения не были безопасно сохранены в промежуточном буфере системы и посылающийся буфер не был снова свободен к доступу (чтению, записи данных). Блокированная принимающая функция возвращает управление только после того, как буфер приема данных получит соответствующее сообщение.

Неблокированная передача/прием данных. Неплокированные операции всегда имеют две части: функции передачи параметров системе, которые иницируют запрошенное действие и функции "проверки на завершение", которые допускают, чтобы прикладная программа обнаружила, закончилось ли запрошенное действие. Неплокированная передающая функция, указывает, что система может начинать копировать данные вне посылающегося буфера. После передачи параметров функции в систему, функция возвращает управление. После этого передающий процесс не должен иметь доступ (по записи и, в данном случае и, по чтению) к посылающемуся буферу. Т.е. система не гарантирует в этом случае сохранность посылаемых данных. Для проверки завершения рассматриваемой операции, используются функции `MPI_WAIT` и `MPI_TEST`. Под завершением операции здесь понимается, что данные сообщения были безопасно сохранены в промежуточном буфере системы и посылающийся буфер снова свободен к доступу. Неплокированная принимающая функция, указывает, что система может начинать писать данные в буфер приема. После передачи параметров функции в систему, функция возвращает управление. После этого принимающий процесс не должен иметь доступ (по записи и, в данном случае и, по чтению) к буферу приема. Т.е. система не гарантирует в этом случае сохранность принимаемых данных. Для проверки завершения рассматриваемой операции, используются те же функции

MPI_WAIT и MPI_TEST. Под завершением операции здесь понимается, что принимаемые данные находятся в буфере приема.

Синхронная блокированная/неблокированная передача данных. Синхронный способ взаимодействий имеет семантику реализации "rendezvous". Синхронная передача может быть начата только после того как соответствующий приемник готов к приему посылаемых данных, т.е. запустилась соответствующая принимающая функция. Таким образом, завершение синхронных передающих операций не только указывает, что посылающийся буфер может теперь использоваться, но также и указывает, что приемник достиг некоторого пункта в его выполнении, а именно, что он запустил выполнение соответствующей получающей функции. Синхронный способ обеспечивает семантику синхронной связи: связь не заканчивается с обоих концов перед обоюдным сближением процессов в связи. Для неблокированных операций функции MPI_WAIT и MPI_TEST проверяют, завершилась операция или нет.

2.1. Блокированные посылающая и получающая функции

Блокированная передача данных

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN buf адрес передаваемого буфера

IN count количество передаваемых элементов

IN datatype тип передаваемых элементов

IN dest номер процесса, которому осуществляется передача данных

IN tag тег сообщения

IN comm имя переключателя каналов (communicator)

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

IN - обозначаются входные параметры, OUT - выходные. Длина сообщения определяется в терминах количества *элементов*, а не количества

байтов. Это связано с тем, что одни и те же типы данных могут иметь разное количество байтов в представлении на ЭВМ с разными архитектурами. Основные типы данных MPI соответствуют основным типам данных базового языка.

Возможные значения для этого аргумента для C и соответствующие типы C внесены в список ниже.

MPI_datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGN	unsigned char
ED_CHAR	unsigned short
MPI_UNSIGN	int
ED_SHORT	unsigned int

MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	

Типы MPI_BYTE и MPI_PACKED не соответствуют типам Fortran или C. Значение типа MPI_BYTE состоит из байта (8 двоичных цифр). Байт неинтерпретируется и отличается от символа (знака).

Блокированный прием данных

MPI_RECV(buf, count, datatype, source, tag, comm, status)

OUT buf адрес буфера для приема данных
 IN count максимальное количество принимаемых элементов
 IN datatype тип принимаемых элементов
 IN source ранг (номер) передающего процесса
 IN tag тег сообщения
 IN comm имя переключателя каналов (communicator)
 OUT status статус (состояние) принимаемых данных

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

Длина сообщения определяется, как и для функции MPI_SEND, в терминах количества *элементов*, а не количества *байтов* и должна быть

меньше или равняться длине буфера для получения данных. Получаемые данные сопровождаются информацией об этих данных, которая записывается в status. На C статус полученных данных представляется структурой типа MPI_Status, которая содержит три поля, с именами: MPI_SOURCE, MPI_TAG, и MPI_ERROR, содержащие, соответственно, источник (source), тег (tag) и код ошибки, полученного сообщения. (Доступ к этим данным: status.MPI_SOURCE, status.MPI_TAG , status. MPI_ERROR). На Fortran(e) статус - это массив целых чисел длины MPI_STATUS_SIZE. Три константы: MPI_SOURCE, MPI_TAG и MPI_ERROR, которые хранят источник (source), тег (tag) и поле ошибки.

Статус имеет также аргумент, через который возвращается информация относительно длины полученного сообщения. Однако, эта информация непосредственно не доступна как поле переменной статуса, а доступна с помощью запроса к функции MPI_GET_COUNT.

MPI_GET_COUNT(status, datatype, count)

IN status статус принятых данных

IN datatype тип принятых элементов

OUT count количество принятых элементов

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

MPI_GET_COUNT берет как входной параметр status, соответствующего MPI_RECV и вычисляет число полученных элементов. Количество полученных элементов возвращено в переменной count. Аргумент datatype должен быть таким же, как и аргумент в получающей функции MPI_RECV, к статусу которой осуществляется доступ.

Соответствие типов для передаваемых и получаемых данных должно строго выполняться. Типы между посылающей и получающей функциями соответствуют, если они обе определяют идентичные имена типа. То есть

MPI_INT соответствует MPI_INT, MPI_DOUBLE соответствует MPI_DOUBLE, и так далее.

ПРИМЕР 1.7

Фрагмент программы для источника и приемника.

```
MPI_Comm_rank(comm, rank);  
if(rank == 0)  
    MPI_Send(a, 10, MPI_FLOAT, 1, tag, comm);  
elseif(rank == 1)  
    MPI_Recv(b, 15, MPI_FLOAT, 0, tag, comm, status);
```

MPI не делает преобразование типов данных, например, округляя double (вещественный) к int (целый) числу, но делает преобразование представления данных, изменяя двоичное представление значения данных в рамках одного и того же типа, например, изменяя значение байта, или изменяя 32-разрядное число с плавающей запятой к 64-разрядному числу с плавающей запятой.

2.2. Объединенная функция передачи/приема данных

MPI имеет специальную функцию (MPI_SENDRECV), объединяющую в одном запросе посылающее и получающее действия. Посылка одного сообщения по назначению и получение другого сообщения из источника. Источник и назначение, - возможно, те же самые. Такая функция полезна для моделей связи, где каждый процесс, и посылает и получает сообщения. Один пример - обмен данными между двумя процессами. Другой пример - смещение данных вдоль цепи процессов. Безопасная программа, которая осуществляет такое смещение, должна использовать одинаковый порядок связи между процессами. MPI_SENDRECV может использоваться вместе с функциями, описанными выше. Имеется совместимость между MPI_SENDRECV и функциями отправки и получения сообщений, описанными выше. Сообщение, посланное MPI_SENDRECV может быть получено обычной функцией приема сообщений и, наоборот,

MPI_SENDRECV может получать сообщение, посланное обычной функцией передачи сообщений.

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
recvcount, recvtype, source, recvtag, comm, status)

IN sendbuf адрес посылаемого буфера

IN sendcount количество, посылаемых элементов

IN sendtype тип элементов в посылаемом буфере

IN dest ранг (номер) процесса, которому осуществляется передача

IN sendtag тег посылаемого сообщения

OUT recvbuf адрес буфера для приема данных

IN recvcount максимальное количество принимаемых элементов

IN recvtype тип принимаемых элементов

IN source ранг (номер) передающего процесса

IN recvtag тег принимаемых данных

IN comm имя переключателя каналов (communicator)

OUT status статус полученного сообщения

int MPI_Sendrecv(void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,
int recvcount, MPI_Datatype recvtype, int source,
MPI_Datatype recvtag, MPI_Comm comm, MPI_Status *status)

MPI_SENDRECV выполняет блокированные посылку и получение данных. Обе посылающая и получающая функции используют тот же самый переключатель каналов. Посылающийся буфер и буфер для приема, не должны пересекаться, и могут иметь различные длины. Следующая функция аналогична предыдущей, но у нее посылаемый буфер совпадает с буфером для получения данных.

MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source,
recvtag, comm, status)

INOUT buf адрес посылаемого буфера и буфера для приема данных
 IN count количество элементов в посылаемом буфере и буфере
 для приема
 IN datatype тип элементов в посылаемом буфере и буфере для приема
 IN dest ранг (номер) процесса, которому осуществляется
 передача
 IN sendtag тег посылаемого сообщения
 IN source ранг (номер) передающего процесса
 IN recvtag тег принимаемых данных
 IN comm имя переключателя каналов (communicator)
 OUT status статус полученного сообщения

```

int MPI_Sendrecv_replace(void* buf, int count,
                          MPI_Datatype datatype, int dest, int sendtag, int source,
                          int recvtag, MPI_Comm comm, MPI_Status *status)
  
```

MPI_SENDRECV_REPLACE выполняется с блокированием посылки и получения. Используется тот же самый буфер, как для посылающей, так и для получающей функции. Посланное сообщение затем заменяется полученным сообщением.

ПРИМЕР 1.8

Имеется N компьютеров, объединенных в топологию кольцо с именем comm. В определенный момент все компьютеры пересылают свои данные соседним компьютерам с большим номером. (Фрагмент программы).

```

.....
MPI_Comm_rank(comm, rank);  /* rank - номер компьютера */
MPI_Comm_rank(comm, size);  /* size - размер системы */
.....
  
```

```
int MPI_Sendrecv(sbuf,scount, stype, MPI_INT, (rank+1)% size, stag,  
                rbuf, rcount, MPI_INT, (rank+size-1)%size, rtag, comm, &status)
```

.....

Аналогично можно использовать и функцию `MPI_Sendrecv_replace()`.

2.3. Неблокированные посылающая и получающая функции

Можно улучшить выполнение многих программ, выполняя обмен данными и вычисления с перекрытием, т.е. параллельно. Это особенно хорошо реализуется на системах, где связь может быть выполнена автономно от работы компьютера. Многоподпроцессный режим - один из механизмов для достижения такого перекрытия. В то время как один из подпроцессов заблокирован, ожидая завершения связи, другой подпроцесс может выполняться на том же самом процессоре. Этот механизм эффективен, если система поддерживает многоподпроцессный режим.

Неблокированные функции

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
```

IN buf	адрес посылаемого буфера
IN count	количество элементов в посылаемом буфере
IN datatype	тип элементов в передаваемом буфере
IN dest	номер принимающего процессора
IN tag	тег передаваемых данных
IN comm	имя коммуникатора связи (communicator)
OUT request	имя (заголовка) запроса

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

Неблокированная передача данных, инициализирует посылающее действие, но не заканчивает его. Функция возвратит управление прежде, чем сообщение скопировано вне посылающегося буфера. Неблокированная посылающая функция указывает, что система может начинать копировать данные вне посылающегося буфера. Посылающий процесс не должен иметь доступ к посылаемому буферу после того, как неблокированное посылающее действие инициировано, до тех пор, пока функция завершения не возвратит управление.

`MPI_Irecv(buf, count, datatype, source, tag, comm, request)`

OUT buf	адрес буфера приема данных
IN count	максимальное количество принимаемых элементов
IN datatype	тип принимаемых элементов
IN source	номер передающего процесса
IN tag	тег сообщения
IN comm	имя коммуникатора связи (communicator)
OUT request	имя (заголовка) запроса

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

Неблокированный прием данных, инициализирует получающее действие, но не заканчивает его. Функция возвратит управление прежде, чем сообщение записано в буфер приема данных. Неблокированная получающая функция указывает, что система может начинать писать данные буфер приема данных. Приемник не должен иметь доступ к буферу приема после того, как неблокированное получающее действие инициировано, до тех пор, пока функция завершения не возвратит управление.

Эти обе функции размещают данные в системном буфере, и возвращают заголовок этого запроса в request. request используется, чтобы опросить состояние связи.

Функции завершения неблокированных операций

Чтобы закончить неблокированные посылку и получение данных, используются *завершающие* функции MPI_WAIT и MPI_TEST. Завершение посылающего процесса указывает, что он теперь свободен к доступу посылающегося буфера. Завершение получающего процесса указывает, что буфер приема данных содержит сообщение, приемник свободен к его доступу.

MPI_WAIT(request, status)

INOUT request имя запроса

OUT status статус объекта

int MPI_Wait(MPI_Request *request, MPI_Status *status)

Запрос к MPI_WAIT возвращает управление, после того как операция, идентифицированная request, выполнялась. То есть это блокированная функция. Если объект системы, указанный request был первоначально создан неблокированными посылающей или получающей функциями, то этот объект освобождается функцией MPI_WAIT, и request устанавливается в MPI_REQUEST_NULL. *Статус* объекта содержит информацию относительно выполненной операции.

MPI_TEST(request, flag, status)

INOUT request имя запроса

OUT flag true, если операция выполнялась, иначе false

OUT status статус объекта

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

Запрос к MPI_TEST возвращает flag = true, если операция, идентифицированная request, выполнялась. В этом случае, *статус* состояния содержит информацию относительно законченной операции. Если объект

системы, указанный request был первоначально создан неблокированными посылающей или получающей функциями, то этот объект освобождается функцией MPI_TEST, и request устанавливается в MPI_REQUEST_NULL. Запрос возвращает flag = false, если операция не выполнялась. В этом случае, значение *статуса* состояния неопределено. То есть это не блокированная функция.

2.4. Синхронные посылающие функции

MPI_SSEND(buf, count, datatype, dest, tag, comm)

IN buf	адрес передаваемого буфера
IN count	количество передаваемых элементов
IN datatype	тип передаваемых элементов
IN dest	ранг приемника
IN tag	тег сообщения
IN comm	коммуникатор (communicator)

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,  
              int tag, MPI_Comm comm)
```

MPI_SSEND - блокированная, синхронная функция передачи данных.

MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)

IN buf	адрес передаваемого буфера
IN count	количество передаваемых элементов
IN datatype	тип передаваемых элементов
IN dest	ранг приемника
IN tag	тег сообщения
IN comm	коммуникатор (communicator)
OUT request	заголовок запроса

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

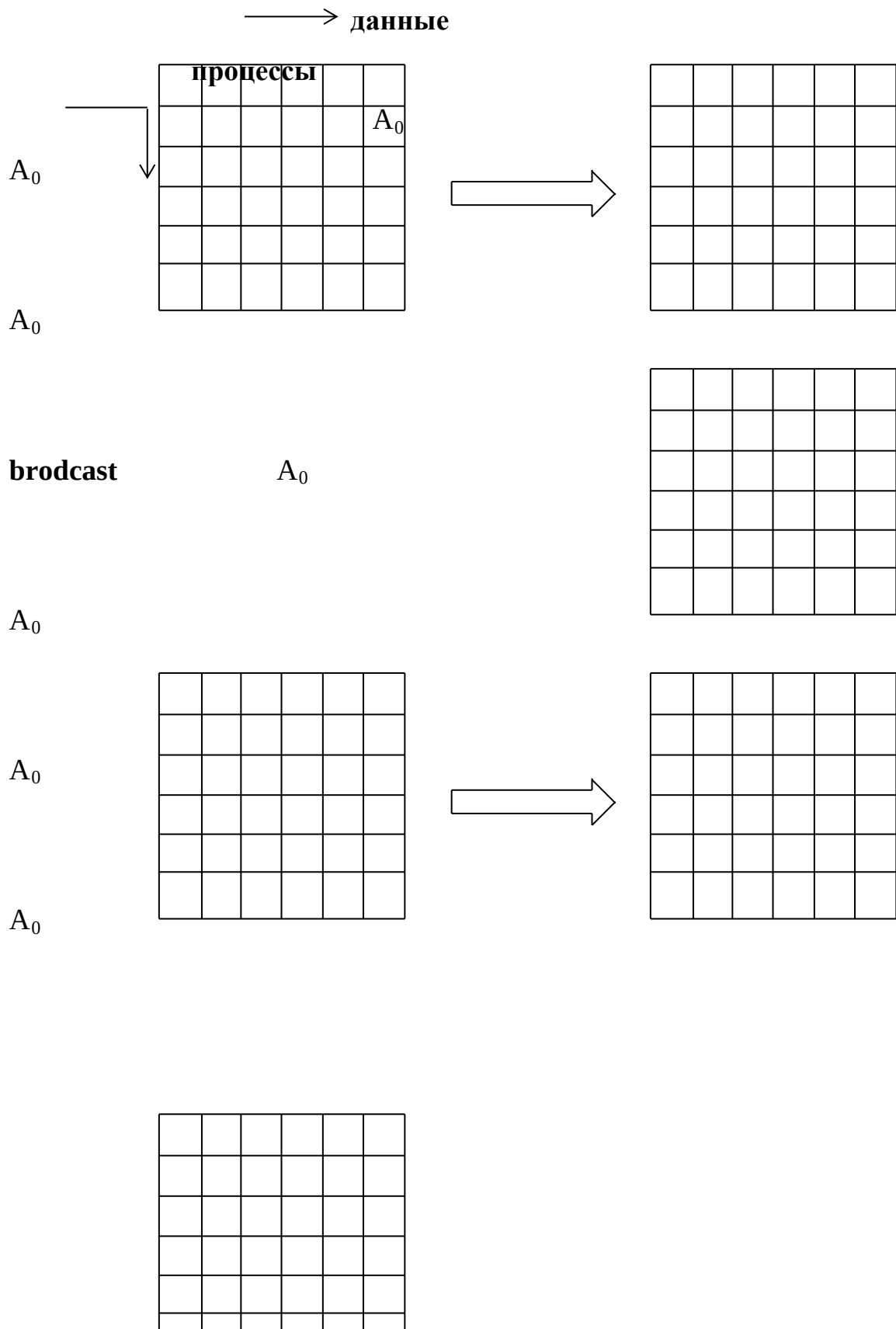
int dest, int tag, MPI_Comm comm, MPI_Request *request)

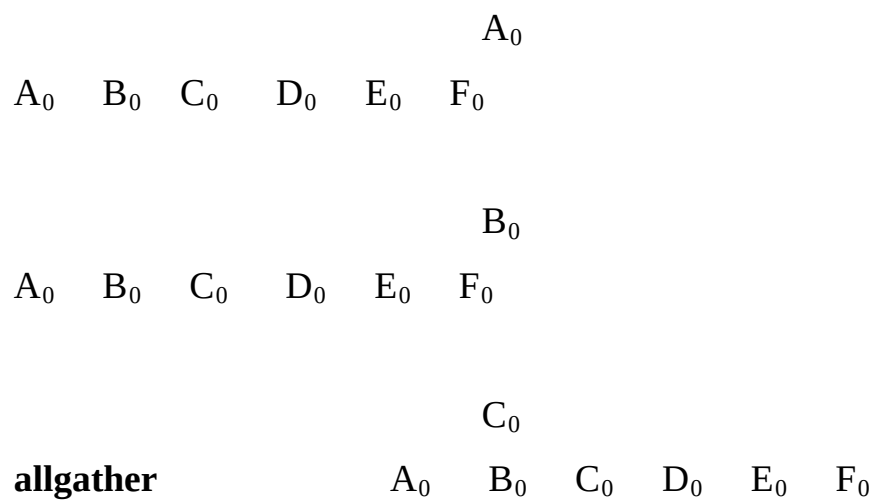
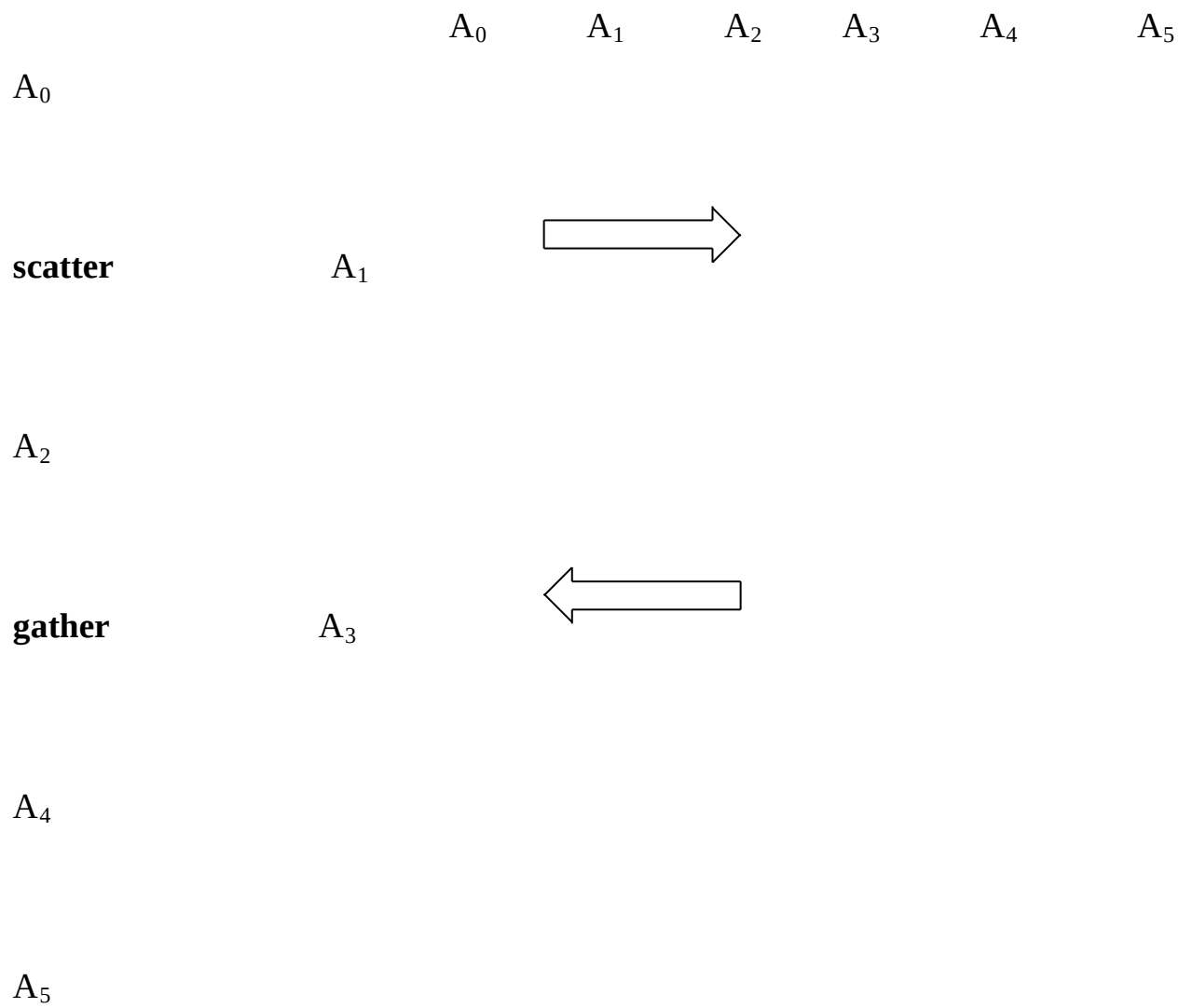
MPI_ISSEND - неблокированная, синхронная функция передачи данных. Если соответствующей принимающей функцией является неблокированная принимающая функция MPI_Irecv, то передающая функция MPI_ISSEND синхронизируется с переданными в систему параметрами соответствующей не блокированной принимающей функции. А функции Wait и Test со стороны передающей функции только проверяют наличие этих выставленных параметров со стороны неблокированной принимающей функции.

При синхронных взаимодействиях пересылаемый буфер передается в принимаемый буфер "напрямую" (память-память) минуя сохранение в промежуточных буферах.

3. Коллективные взаимодействия

Коллективная связь обеспечивает обмен данными среди всех процессов в группе, указанной аргументом коммуникатора.

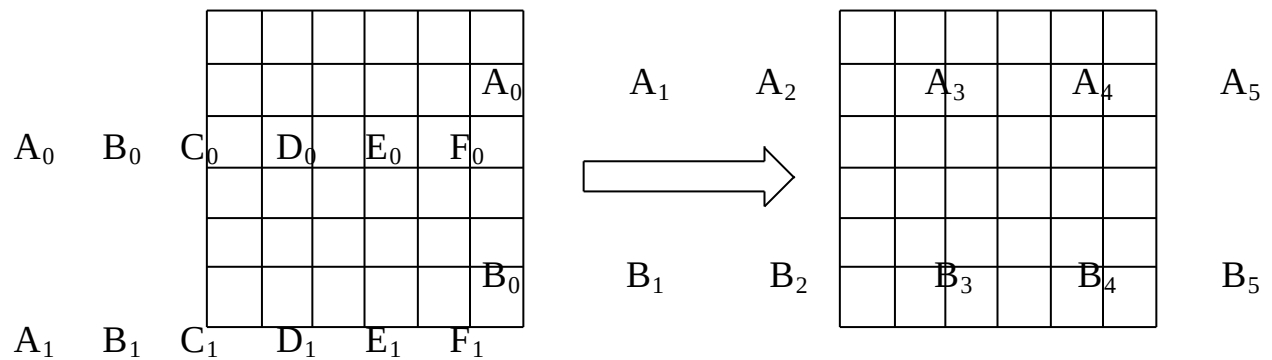




D_0
 $A_0 \quad B_0 \quad C_0 \quad D_0 \quad E_0 \quad F_0$

E_0
 $A_0 \quad B_0 \quad C_0 \quad D_0 \quad E_0 \quad F_0$

F_0
 $A_0 \quad B_0 \quad C_0 \quad D_0 \quad E_0 \quad F_0$



$C_0 \quad C_1 \quad C_2 \quad C_3 \quad C_4 \quad C_5$
alltoall $A_2 \quad B_2 \quad C_2 \quad D_2 \quad E_2 \quad F_2$

$D_0 \quad D_1 \quad D_2 \quad D_3 \quad D_4 \quad D_5$
 $A_3 \quad B_3 \quad C_3 \quad D_3 \quad E_3 \quad F_3$

$E_0 \quad E_1 \quad E_2 \quad E_3 \quad E_4 \quad E_5$
 $A_4 \quad B_4 \quad C_4 \quad D_4 \quad E_4 \quad F_4$

$F_0 \quad F_1 \quad F_2 \quad F_3 \quad F_4 \quad F_5$
 $A_5 \quad B_5 \quad C_5 \quad D_5 \quad E_5 \quad F_5$

Рис. 1.3 Иллюстрация коллективных передающих функций для группы из шести процессов. В каждой клетке представлены локальные данные в одном процессе. Например, в broadcast передает данные A_0 только первый процесс, а другие процессы принимают эти данные.

Коллективные сделаны более ограниченными чем point-to-point операции. В отличие от point-to-point операций, количество посланных данных в этих функциях должно быть точно согласовано с количеством данных, указанных приемником. Коллективные функции имеют только блокированные версии. Коллективные функции не используют аргумент тега. Аргумент типа данных должен быть одним и тем же во всех процессах, участвующих во взаимодействии. Внутри каждой области связи, коллективные запросы строго согласованы согласно порядку выполнения. В коллективном запросе к MPI_BCAST должны участвовать все процессы, объединенные коммунитором. Коллективные функции не согласуются с функциями парных взаимодействий.

3.1. Синхронизация

MPI_BARRIER(comm)

IN comm коммунитор

int MPI_Barrier(MPI_Comm comm)

MPI_BARRIER блокирует вызывающий оператор, пока все элементы группы не вызовут его. В любом процессе запрос возвращается только после того, как все элементы группы вошли в запрос.

3.2. Трансляционный обмен данными

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT buffer адрес буфера

IN count количество элементов в буфере

IN	datatype	тип данных
IN	root	ранг корневого процесса
IN	comm	коммуникатор

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,  
              int root, MPI_Comm comm)
```

MPI_BCAST передает сообщение из процесса с рангом root ко всем процессам группы. Аргументы корня и на всех других процессах должны иметь идентичные значения, и comm должна представлять ту же самую область связи. После возвращения, содержимое буфера buffer из корня копируется ко всем процессам в буфер buffer.

3.3. Сбор данных

```
MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,  
            root, comm)
```

IN	sendbuf	адрес передаваемого буфера
IN	sendcount	количество передаваемых элементов
IN	sendtype	тип передаваемых элементов
OUT	recvbuf	адрес буфера приема
IN	recvcount	количество принимаемых элементов в каждом процессе
IN	recvtype	тип принимаемых данных
IN	root	ранг принимающего процесса
IN	comm	коммуникатор

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
               void* recvbuf, int recvcount, MPI_Datatype recvtype,  
               int root, MPI_Comm comm)
```

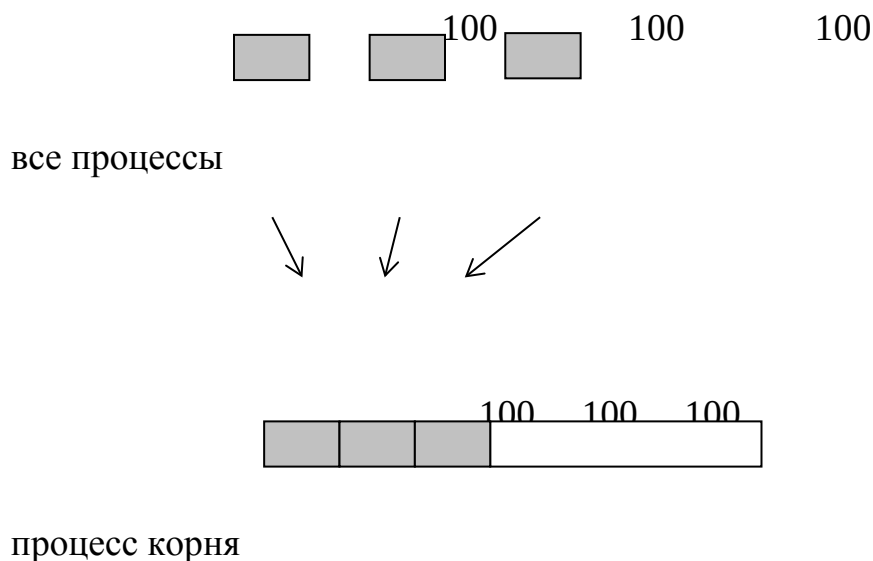
Каждый процесс (включая процесс корня) посылает содержимое его посылаемого буфера к процессу корня. Процесс корня получает сообщения и

хранит их в порядке рангов, посылающих процессов. Результат выглядит так, как будто каждый из n процессов в группе (включая процесс корня) выполнил запрос к `MPI_SEND(sendbuf,sendcount,sendtype,root, ...)`, и корень выполнил n запросов к `MPI_RECV(recvbuf+i*recvcount,recvcount,recvtype,i,...)`. Приемный буфер игнорируется для всех процессов не равных корню. Аргумент `recvcount` в корне указывает число элементов, которые получает корень от каждого процесса, а не общее число элементов, которые он получает всего.

ПРИМЕР 1.12

Прием корнем 100 элементов от каждого процесса группы. (рис. 1.4).

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf;
...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```



начинается в смещении от начала предыдущего пакета в `displs[j]` элементов (в терминах `recvtype`). Буфер приема игнорируется для всех процессов, не принадлежащих корню. Все аргументы в функции на корне процесса значимы, в то время как на других процессах, значимы только аргументы `sendbuf`, `sendcount`, `sendtype`, `root`, и `comm`. Аргументы должны иметь идентичные значения на всех процессах, и `comm` должна представлять ту же самую область связи.

ПРИМЕР 1.13

Каждый процесс посылают 100 элементов. В принимающем процессе пакеты (в 100 элементов) нужно разместить на расстоянии с некоторым шагом. Используется `MPI_GATHERV` и аргумент `displs`, чтобы достичь этого эффекта. Допустим шаг $\text{stride} \geq 100$. (рис. 1.5).

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf, stride;
int *displs, i, *rcounts;
...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for(i = 0; i < gsize; ++i)
    { displs[i] = i*stride;
      rcounts[i] = 100;
    }
MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
root, comm);
```

Программа ошибочна, если $-100 < \text{stride} < 100$.

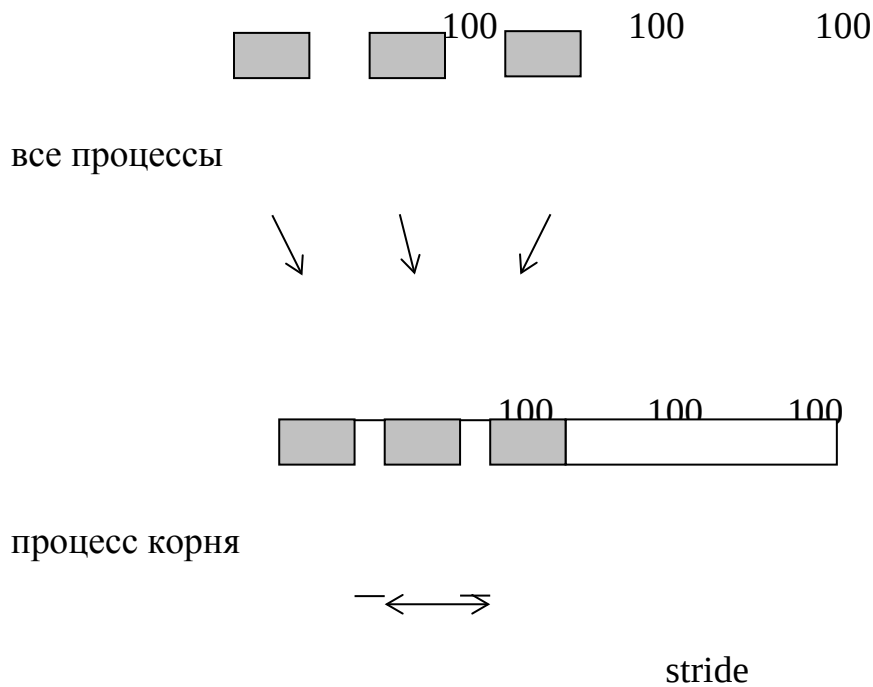


Рис. 1.5 Процесс корня принимает 100 элементов от каждого процесса в группе; каждый набор размещается с шагом `stride` элементов друг от друга.

ПРИМЕР 1.15

Тот же самый как пример 1.13 в посылающей стороне, но в получающей стороне делается разный шаг между полученными блоками (рис. 1.7).

```

MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, *stride, myrank, bufsize;
MPI_Datatype stype;
int *displs, i, *rcounts, offset;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
stride = (int *)malloc(gsize*sizeof(int));
...

```

```

/* Установлен stride[i] от i = 0 до gsize-1 */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for(i = 0; i < gsize; ++i)
{ displs[i] = offset;
  offset += stride[i];
  rcounts[i] = 100-i;
  scounts[i] = 100-myrank;
}
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, scounts, MPI_INT, rbuf, rcounts, displs, MPI_INT, root,
comm);

```

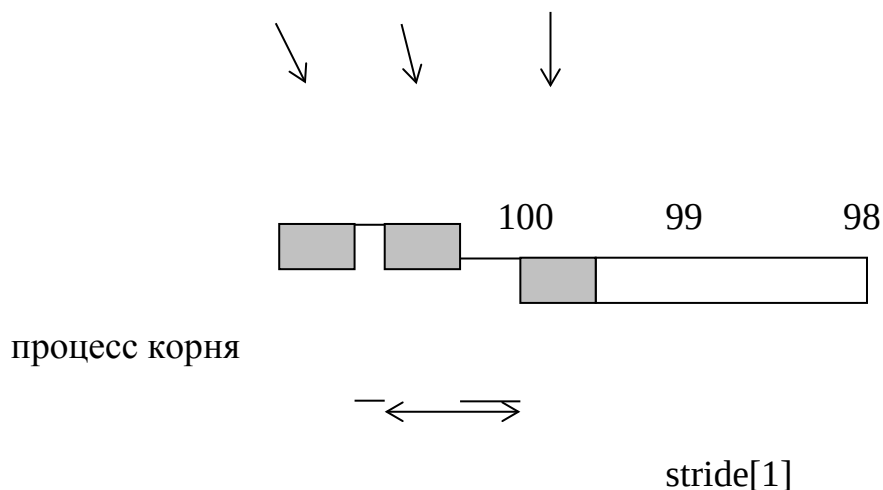
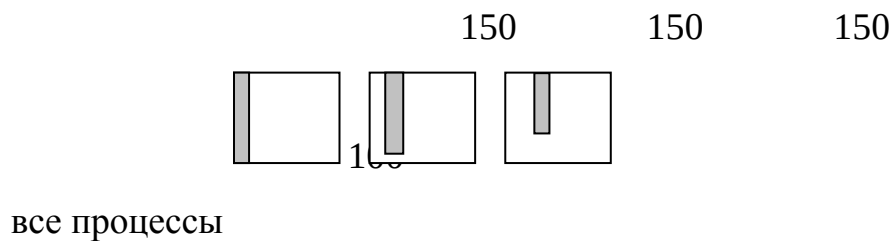


Рис. 1.7 Процесс корня принимает $100-i$ элементов из колонки i массива 100×150 , и каждый набор размещается с изменяющимся шагом $\text{stride}[1]$ элементов.

3.5. Разброс данных

`MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, root, comm)`

IN sendbuf адрес передаваемого буфера

IN sendcount количество передаваемых элементов каждому процессу

IN sendtype тип передаваемых данных

OUT recvbuf адрес буфера приема

IN recvcount количество принимаемых элементов

IN recvtype тип принимаемых данных

IN root ранг передающего процесса

IN comm коммуникатор (communicator)

`int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)`

`MPI_SCATTER` - обратная операция к `MPI_GATHER`. Результат выглядит так, как будто корень выполнил n посылающих операций `MPI_Send(sendbuf+i*sendcount *extent(sendtype), sendcount, sendtype, i, ...)`, $i=0$ до $n-1$. И каждый принимающий процесс выполнил функцию, `MPI_Recv(recvbuf, recvcount, recvtype, root, ...)`. Аргументы `sendcount` и `sendtype` в корне должны быть равны аргументам `recvcount` и `recvtype` во всех процессах. Это подразумевает, что количество посланных данных должно быть равно количеству полученных данных, попарно между каждым процессом и корнем. Все аргументы в функции значимы на корневом

процессе, в то время как на других процессах, значимы только аргументы `recvbuf`, `recvcount`, `recvtype`, `root`, `comm`.

ПРИМЕР 1.16

`MPI_Scatter` передает 100 элементов из корня каждому процессу в группе (рис. 1.8).

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

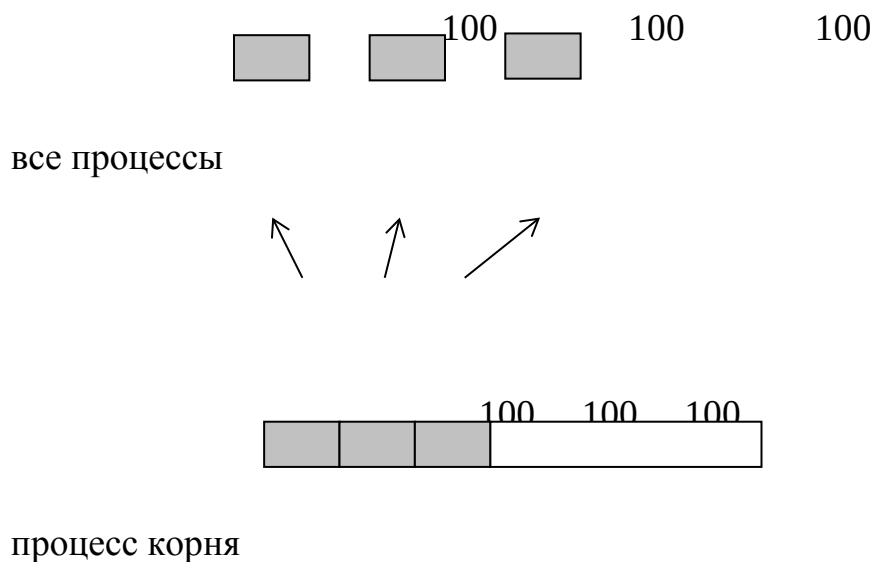


Рис. 1.8. Процесс корня передает наборы 100 элементов к каждому процессу в группе.

3.6. Разброс данных (векторный вариант)

`MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf,
recvcount, recvtype, root, comm)`

IN sendbuf адрес передаваемого буфера

IN sendcounts массив, содержащий количество передаваемых
элементов каждому процессу

IN displs целочисленный массив смещений пакетов
относительно друг друга

IN sendtype тип передаваемых данных

OUT recvbuf адрес буфера приема

IN recvcount количество принимаемых элементов

IN recvtype тип принимаемых данных

IN root ранг передающего процесса

IN comm коммуникатор (communicator)

`int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)`

`MPI_SCATTERV` - обратная операция к `MPI_GATHERV`.
`MPI_SCATTERV` расширяет функциональные возможности `MPI_SCATTER`,
позволяя посылать каждому процессу изменяющееся количество данных, так
как `sendcounts` - теперь массив. Она также допускает большее количество
относительно того, где данные взяты из корня, обеспечивая новый аргумент
`displs`.

Результат выглядит так, как будто корень выполнил `n` посылающих
операций

`MPI_Send(sendbuf+displs[i]*extent(sendtype),sendcounts[i],sendtype,i,...)`, `i=0` до
`n-1`, и каждый процесс выполнил функцию приема
`MPI_Recv(recvbuf,recvcount, recvtype,root,...)`. Все аргументы в функции

значимы на корневом процессе, в то время как на других процессах, значимы только аргументы `recvbuf`, `recvcount`, `recvtype`, `root`, `comm`.

ПРИМЕР 1.18

Обратный примеру 1.14. Передаваемые блоки в буфере корня расположены с изменяющимся шагом между блоками, получающая сторона принимает $100-i$ элементов в i -ю колонку массива 100×150 в процессе i (рис. 1.10).

```
MPI_Comm comm;
int gsize, recvarray[100][150], *rptr;
int root, *sendbuf, myrank, bufsize, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] для i=0 до gsize-1 установлены */
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for(i = 0; i < gsize; ++i)
{
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100-i;
    rcounts[i] = 100-i;
}
MPI_Type_commit(&rtype);
```

```

rptr = &recvarray[0][myrank];
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rptr, rcounts, MPI_INT,
root, comm);

```

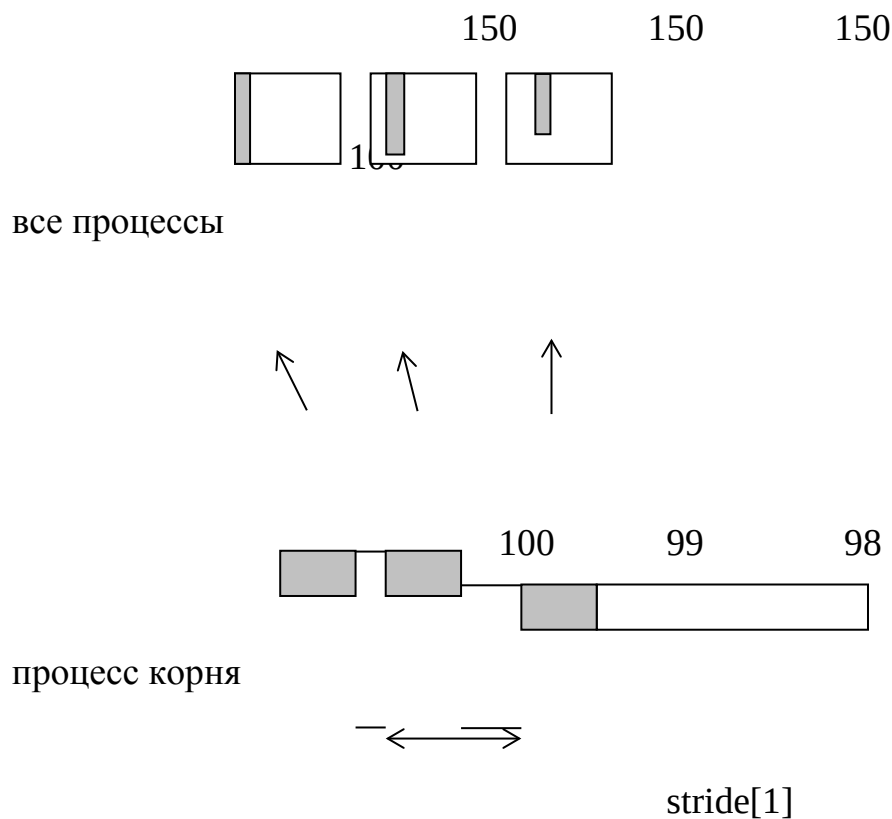


Рис. 1.10 Корень разбрасывает блоки по 100-и элементов в колонки i массива 100×150 . В корне блоки расположены в буфере с шагом $\text{stride}[i]$ элементов.

3.7. Сбор данных у всех процессов

```

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
comm)

```

IN sendbuf	адрес передаваемого буфера
IN sendcount	количество передаваемых элементов

IN sendtype	тип передаваемых данных
OUT recvbuf	адрес буфера приема
IN recvcount	количество принимаемых элементов от процессов
IN recvtype	тип принимаемых данных
IN comm	коммуникатор (communicator)

```
int MPI_Allgather(void* sendbuf, int sendcount,
                 MPI_Datatype sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, MPI_Comm comm)
```

MPI_ALLGATHER аналогична операции MPI_GATHER, за исключением того, что все процессы получают результат от всех процессов, вместо только одного корня. j-й блок данных, посланных из каждого процесса, получен каждым процессом и размещается в j-м блоке буфера recvbuf.

Аргументы sendcount и sendtype в процессе должны быть равны во всех процессах.

Результат запроса к MPI_ALLGATHER(...) выглядит так, как будто все процессы выполнили n запросов к MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm), для root=0,...,n-1.

ПРИМЕР 1.19

Версия примера 1.11. Используется MPI_ALLGATHER, принимается по 100 элементов из каждого процесса в группе в каждом процессе.

```
MPI_Comm comm;
int gsize, sendarray[100];
int *rbuf;
...
MPI_Comm_size(comm, &gsize);
```

```
rbuf = (int *)malloc(gsize*100*sizeof(int));  
MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

После запроса, каждый процесс имеет конкатенацию наборов данных в количестве размера группы.

3.8. Сбор данных у всех процессов (векторный вариант)

```
MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts,  
                displs, recvtype, comm)
```

IN sendbuf адрес передаваемого буфера

IN sendcount количество передаваемых элементов

IN sendtype тип передаваемых данных

OUT recvbuf адрес буфера приема

IN recvcounts массив, указывающий количество принимаемых
элементов

от процессов

IN displs целочисленный массив смещений пакетов данных друг
относительно друга

IN recvtype тип принимаемых данных

IN comm коммуникатор (communicator)

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype  
                  sendtype, void* recvbuf, int *recvcounts, int *displs,  
                  MPI_Datatype recvtype, MPI_Comm comm)
```

MPI_ALLGATHERV аналогична MPI_GATHERV, за исключением того, что все процессы получают результат, вместо только одного корня. j-й блок данных, посылается из каждого процесса, получается каждым процессом и размещается в j-м блоке буфера recvbuf. Не все эти блоки могут иметь тот же самый размер. Аргументы sendcount и sendtype в процессе j должны быть равны аргументам recvcounts[j] и recvtype в любом другом

процессе. Результат выглядит так, как будто все процессы выполнили запросы

к

MPI_GATHERV(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,
recvtype, root, comm), для root=0,..., n-1.

3.9. Разброс/сбор - все ко всем

MPI_ALLTOALL(sendbuf,sendcount,sendtype,recvbuf,recvcount,
recvtype,comm)

IN sendbuf адрес передаваемого буфера

IN sendcount количество передаваемых элементов к каждому процессу

IN sendtype тип передаваемых данных

OUT recvbuf адрес буфера приема

IN recvcount количество принимаемых элементов от каждого процесса

IN recvtype тип принимаемых данных

IN comm коммуникатор (communicator)

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLTOALL - расширение MPI_ALLGATHER для случая, когда каждый процесс посылает различные данные на каждый из приемников. j-й блок, посланный из процесса i получен процессом j и размещен в i-м блоке буфера recvbuf. Аргументы sendcount и sendtype в процессе должны быть равны аргументам recvcount и recvtype в любом другом процессе. Это подразумевает, что количество посланных данных должно быть равно количеству полученных данных, попарно между каждой парой процессов. Результат выглядит так, как будто каждый процесс выполнил посылающий к каждому процессу (включая себя) запрос MPI_Send(sendbuf+i*sendcount*extent(sendtype),

sendcount, sendtype,i,...), и получает из каждого другого процесса запросом к MPI_Recv(recvbuf+i*recvcount*extent(recvtype),recvcount,i,...), где i=0,...,n-1.

3.10. Все ко всем (векторный вариант)

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype,comm)

IN sendbuf адрес передаваемого буфера

IN sendcounts массив, содержащий количество передаваемых
элементов

каждому процессу

IN sdispls массив смещений передаваемых пакетов данных
относительно друг друга

IN sendtype тип передаваемых данных

OUT recvbuf адрес буфера приема

IN recvcounts массив, указывающий количество принимаемых
элементов от процессов

IN rdispls массив смещений принимаемых пакетов данных
относительно друг друга

IN recvtype тип принимаемых данных

IN comm коммуникатор (communicator)

int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
RECVBUF,

RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)

<type> SENDBUF(*), RECVBUF(*)

INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE,
RECVCOUNTS(*),
RDISPLS(*), RECVTYPE, COMM, IERROR

MPI_ALLTOALLV прибавляет гибкость к MPI_ALLTOALL в том, что адреса данных для посылающего процесса определена в массиве sdispls и адреса размещения данных в получающей стороне определена в массиве rdispls. j-й блок, посланный из процесса i получен процессом j и размещен в i-м блоке массива recvbuf. Эти блоки не все имеют тот же самый размер. Аргументы sendcount[j] и sendtype в процессе i должны быть равны аргументам recvcount[i] и recvtype в процессе j. Это подразумевает, что количество посланных данных должно быть равно количеству полученных данных, попарно между каждой парой процессов. Результат выглядит так, как будто каждый процесс послал сообщение процессу i функцией MPI_SEND(sendbuf+displs[i]*extent(sendtype), sendcounts[i],sendtype,i,...), и получил сообщение из процесса i запросом к MPI_RECV(recvbuf+displs[i]*extent(recvtype),recvcounts[i],recvtype, i,...), где i=0,...,n-1.

4. Определяемые пользователем типы данных

Часто желательно послать данные, которые не однородны, типа структуры, или это не смежные в памяти элементы, типа секции массива. MPI обеспечивает два механизма, чтобы достичь этого. Пользователь может определять производные типы (datatypes), которые определяют более общее расположение данных. Определяемый пользователем тип может использоваться в MPI-функциях связи, аналогично, как и базовый, предопределенный тип.

Производный тип данных строится из основных типа данных с использованием строителей типов. Строители могут применяться рекурсивно.

Производный тип данных - непрозрачный объект, который определяет два предмета:

- последовательность примитивных типов и,
- последовательность целого числа смещений (в количестве байт) значений этих типов от начального адреса.

Не требуется, чтобы смещения были положительными, различными, или в увеличивающемся порядке. Следовательно, порядок значений типов в списке производного типа не обязательно должен совпадать с их порядком в памяти, и значения типов могут появляться в списке больше чем один раз. Такая пара последовательностей (или последовательность пар) называется *отображением типа*. Последовательность примитивных типов данных (смещения игнорируются) - называется *сигнатурой типа* данных.

Предположим

$$\text{Typemap} = \{(\text{type}_0, \text{disp}_0), \dots, (\text{type}_{n-1}, \text{disp}_{n-1})\},$$

такое отображение типа, где type_i - примитивные типы, и disp_i - смещения значений этих типов относительно базового адреса.

Предположим

$$\text{Typesig} = \{\text{type}_0, \dots, \text{type}_{n-1}\},$$

соответствующая сигнатура типа. Это отображение типа, вместе с базовым адресом buf , определяет коммуникационный буфер: коммуникационный буфер, который состоит из n элементов, где i -й элемент стоит по адресу $\text{buf} + \text{disp}_i$ и имеет тип type_i . Сообщение, собранное из отдельных типов будет состоять из n значений типов, определенных Typesig .

Имя производного типа может появляться как аргумент в посылающей или получающей функции, вместо аргумента примитивного типа. Функция $\text{MPI_SEND}(\text{buf}, 1, \text{datatype}, \dots)$ использует адрес посылаемого буфера, как базовый адрес buf производного типа данных, и тип посылаемых данных, как производный тип, соответствующий datatype . Она генерирует сообщение с сигнатурой типа, определенной аргументом datatype . Функция $\text{MPI_RECV}(\text{buf}, 1, \text{datatype}, \dots)$ использует адрес буфера приема, как базовый

адрес buf производного типа, и тип принимаемых данных, как производный тип, соответствующий datatype.

Производные типы данных могут использоваться во всех посылающих и принимающих функциях, включая коллективные функции.

Диапазон (extent) типа данных определен, как поле памяти от первого байта до последнего байта, заполненного элементами этого типа данных, округляемыми в большую сторону, чтобы удовлетворить требования к точности совмещения.

ПРИМЕР 1.20

Предположим, что $Type = \{(double, 0), (char, 8)\}$ (double со смещением ноль, char со смещением восемь). Предположим, кроме того, что вещественные величины (double), должны строго выравниваться по адресам, кратным восьми байтам. Тогда, $lb(Type) = \min_j disp_j = 0$, $ub(Type) = \max_j (disp_j + sizeof(type_j)) + e = (8 + 1 + 7) = 16$, и диапазон этого типа равен 16 (8 байт (double) + 1 байт (char) = 9, которая округляется к следующему кратному 8, это есть 16). Отображение этого типа проиллюстрировано ниже на рис. 1.11.

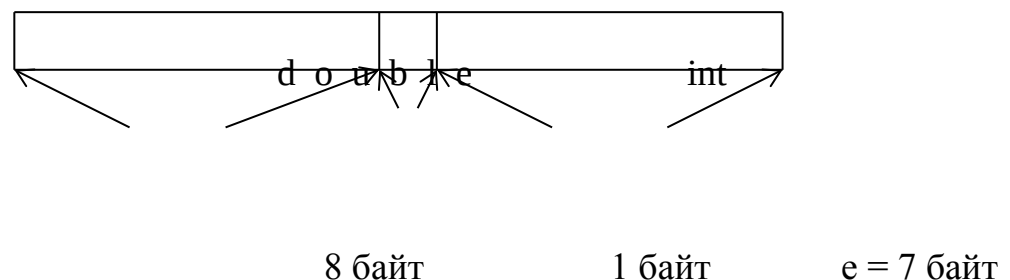


Рис. 1.11. Отображение типа $Type = \{(double, 0), (char, 8)\}$

Следующие функции возвращают информацию относительно типов данных.

`MPI_TYPE_EXTENT(datatype, extent)`

IN datatype тип данных

OUT extent диапазон типа datatype

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

MPI_TYPE_EXTENT возвращает диапазон типа datatype. В дополнение к его использованию с производным datatypes, она может использоваться, чтобы запросить относительно диапазона примитивного datatypes. Например, MPI_TYPE_EXTENT(MPI_INT, extent) возвратит в extent размер, в байтах, int - то же самое значение, которое было бы возвращено в C, функцией sizeof(int).

Имена типов данных в MPI - непрозрачные, поэтому нужно использовать функцию MPI_TYPE_EXTENT, чтобы определить размер ("size") типа. Нельзя использовать по аналогии, как в C функцию sizeof(datatype), например, sizeof(MPI_DOUBLE). Она возвратит размер непрозрачного заголовка, который является размером указателя, и, конечно же, отличается от значения sizeof(double).

```
MPI_TYPE_SIZE(datatype, size)
```

IN datatype тип данных

OUT size размер типа данных

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

MPI_TYPE_SIZE возвращает полный размер, в байтах, входов в сигнатуре типа, связанной с datatype; то есть полный размер данных в сообщении, которое было бы создано с этим datatype. Элементы, которые встречаются многократно в datatype, учитываются с их кратностью. Для примитивного datatypes, эта функция возвращает ту же самую информацию как MPI_TYPE_EXTENT.

ПРИМЕР 1.21

Допустим datatype имеет тип отображения Type, определенный в примере 1.21. Тогда запрос к MPI_TYPE_EXTENT(datatype,i) возвратит i=16; запрос к MPI_TYPE_SIZE(datatype,i) возвратит i=9 (8 байт (double) + 1 байт (char) = 9).

4.1. Строитель смежных типов данных CONTIGUOUS

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN count	количество копий
IN oldtype	старый тип данных
OUT newtype	новый (сконструированный) тип данных

int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_CONTIGUOUS самый простой строитель типа данных. Он конструирует новый тип данных путем размножения count копий исходного типа в смежные поля. Аргумент newtype – новый, полученный, тип, представляющий собой count смежных копий исходного типа oldtype. При сочленении копий, используется extent(oldtype) (диапазон исходного типа) как размер составных копий. Действие CONTIGUOUS-строителя представлено схематично на рис. 1.12.

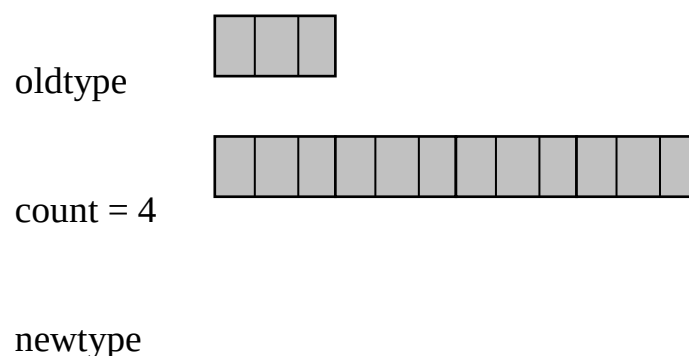


Рис. 1.12. Построение типа данных функцией строителя типов MPI_TYPE_CONTIGUOUS.

ПРИМЕР 1.22

Допустим oldtype имеет отображение типа $\{(double,0),(char,8)\}$; с диапазоном 16, и допустим count=3. Отображение типа datatype, возвращенного newtype есть:

$\{(double,0),(char,8),(double,16),(char,24),(double,32),(char,40)\}$,

то есть, чередуются double и char элементы, со смещениями 0, 8, 16, 24, 32, 40.

4.2. Векторный строитель типов данных VECTOR

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

IN count	количество конструируемых блоков
IN blocklength	количество элементов в каждом блоке
IN stride	расстояние между началами последовательных блоков, выраженное в количестве исходных элементов
IN oldtype	исходный тип данных
OUT newtype	новый тип данных

```
int MPI_Type_vector(int count, int blocklength, int stride,  
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

MPI_TYPE_VECTOR–строитель размножает копии исходного типа oldtype в поля, которые являются равноотстоящими друг от друга блоками. Каждый блок получен из заданного количества (blocklength) смежных копий исходного типа oldtype. Расстояние между блоками здесь задается как расстояние между началами двух соседних блоков. Это расстояние измеряется в единицах oldtype диапазона и одинаково для всех рядом стоящих блоков. В отображении типа смещения блоков даются относительно базового адреса. Например, смещение для нулевого в последовательности блока равно $0*stride$, для пятого блока равно $5*stride$, и т.д. Действие VECTOR-строителя представлено схематично на рис. 1.13.

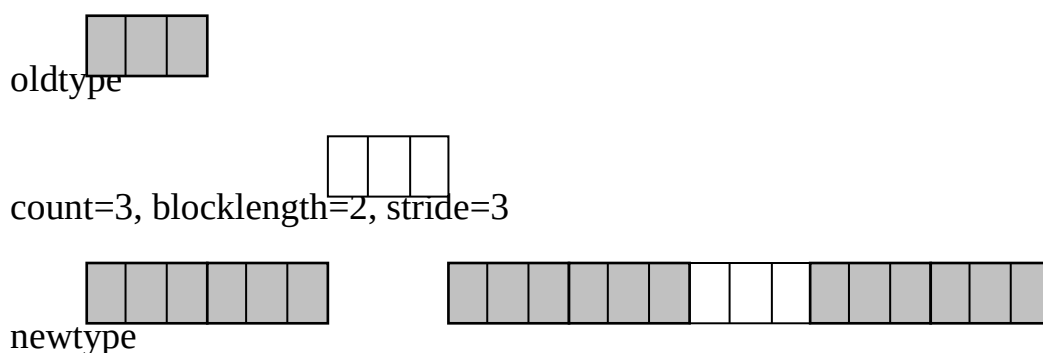


Рис. 1.13. Построение типа данных функцией строителя типов MPI_TYPE_VECTOR

4.3. Модифицированный Векторный строитель типов данных HVECTOR

Векторный строитель типа, описанный в предыдущем пункте, задает расстояние между началами соседних блоков в единицах oldtype диапазона. Иногда полезно ослабить это предположение и допускать расстояние, измеряемое в байтах. Строитель типа Hvector задает расстояния между началами соседних блоков в байтах. Использование Vector и Hvector - строителей иллюстрируется в примере 1.24.

MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)

IN count	количество конструируемых блоков
IN blocklength	количество элементов в каждом блоке
IN stride	расстояние между началами соседних блоков, выраженное в байтах
IN oldtype	исходный тип данных
OUT newtype	новый тип данных

```
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

MPI_TYPE_HVECTOR идентичен MPI_TYPE_VECTOR, за исключением того, что расстояние (stride) дается в байтах. (Н олицетворяет "heterogeneous" системы). Действие Hvector-строителя представлено схематично на рис. 1.14.

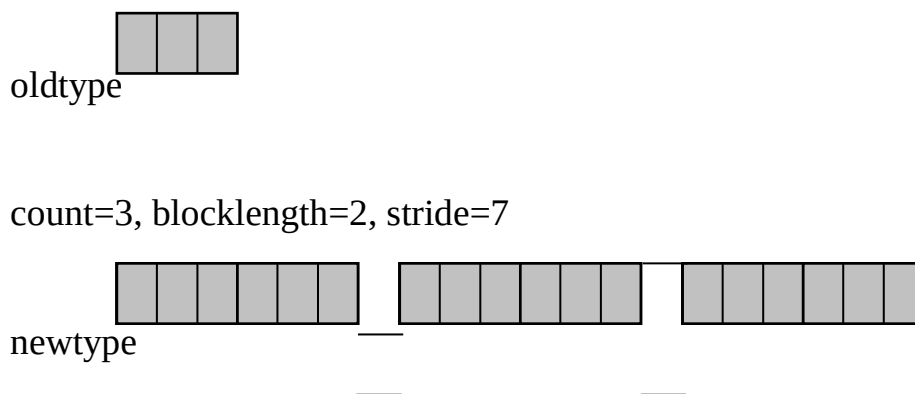


Рис. 1.14. Построение типа данных функцией строителя типов MPI_TYPE_HVECTOR

Пример 1.24 показывает, как определяемый пользователем тип данных используется при послыке верхней триангуляции матрицы. На рис. 1.15 показана диаграмма расположения в памяти, определяемого пользователем типа данных.

ПРИМЕР 1.24

Фрагмент программы, которая передает верхнюю триангуляцию матрицы.

```
double a[100][100], disp[100], blocklen[100], i;
MPI_Datatype upper;

/*Вычисление начала и размера каждой строки матрицы (начиная от
диагонали) */
for(i=0; i<100; ++i)
{ disp[i] = 100*i+i;
  blocklen[i] = 100-i;
```

```

    }
    /* Создание типа для верхней триангуляции матрицы */
    MPI_Type_indexed(100, blocklen, disp, MPI_DOUBLE, &upper);
    MPI_Type_commit(&upper);
    /* .. и его передача */
    MPI_Send(a, 1, upper, dest, tag, MPI_COMM_WORLD);

```

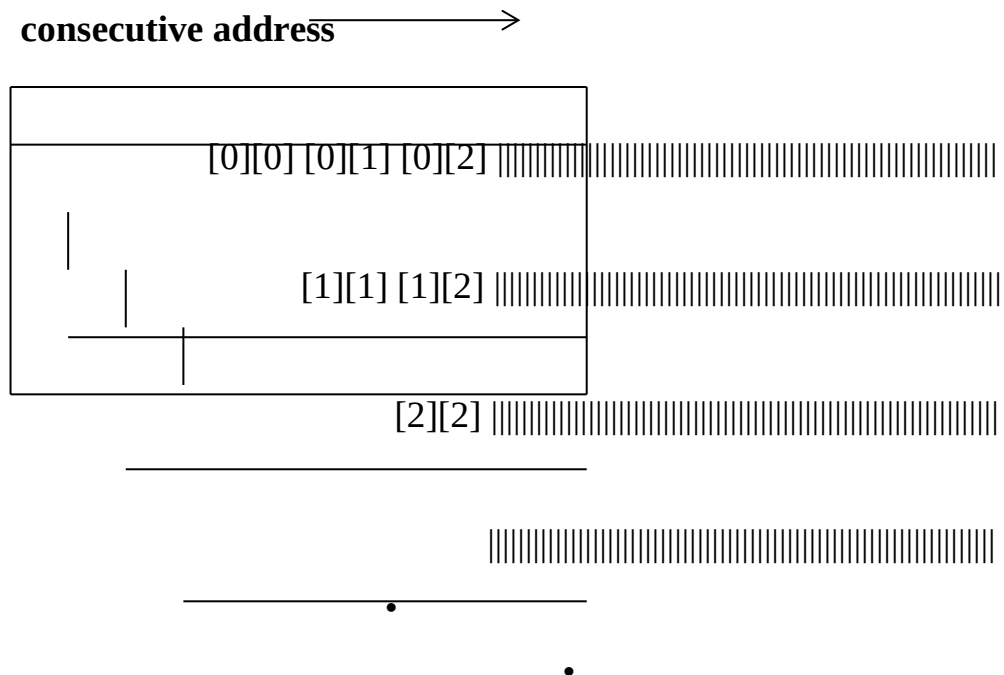


Рис. 1.15. Диаграмма ячеек памяти, представляющих определяемый пользователем тип данных `upper`. Заштрихованные ячейки - это элементы матрицы, которые будут посланы.

4.4. Индексированный строитель типов данных `INDEXED`

Индексированный строитель определяет расположение данных состоящих из нескольких несмежных блоков. Передающий процесс компанует все блоки в один пакет и посылает их в одном сообщении. Получающий процесс полученные в сообщении блоки раскомпановывает в соответствии с определением типа.

Рис. 1.16. Построение типа данных функцией строителя типов MPI_TYPE_INDEXED

ПРИМЕР 1.25

Допустим oldtype имеет отображение типа {(double,0),(char,8)} с диапазоном 16, B=(3,1) и D=(4,0). Функция MPI_TYPE_INDEXED(2,B,D,oldtype,newtype) возвратит следующее отображение типа:

{(double,64),(char,72),(double,80),(char,88),(double,96),(char,104),
(double,0), (char, 8)}.

То есть три копии исходного типа, со смещением от базового адреса $4 * 16 = 64$, и одной копии, со смещением 0.

4.5. Модифицированный индексированный строитель типов данных HINDEXED

Иногда удобно измерять смещения в однотипных элементах диапазона oldtype, но иногда необходимо учесть произвольные смещения. Hindexed-строитель удовлетворяет последнему требованию.

MPI_TYPE_HINDEXED(count,array_of_blocklengths,array_of_displacements,

oldtype, newtype)

IN count количество блоков

IN array_of_blocklengths количество элементов в каждом блоке

IN array_of_displacements смещения каждого блока, измеряемые в байтах

IN oldtype исходный тип данных

OUT newtype новый тип данных

int MPI_TYPE_hindexed(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements,MPI_Datatype oldtype,MPI_Datatype
*newtype)

`MPI_TYPE_HINDEXED` идентичен `MPI_TYPE_INDEXED`, за исключением того, что смещения блоков в массиве смещений определены в байтах, а не в однотипных элементах `oldtype` диапазона. Действие `Hindexed-`строителя представлено схематично на рис. 1.17.

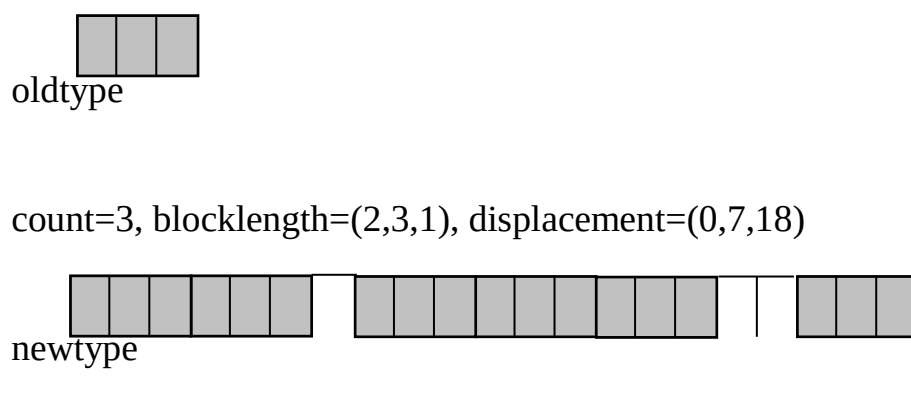


Рис. 1.17. Построение типа данных функцией строителя типов `MPI_TYPE_HINDEXED`

ПРИМЕР 1.26

Здесь используются те же самые аргументы функции `MPI_TYPE_INDEXED`, в примере 1.25. Таким образом, `oldtype` имеет отображение типа, `{{(double,0),(char,8)}}` с диапазоном 16, `B=(3,1)`, и `D=(4,0)`. Запрос к `MPI_TYPE_HINDEXED(2,B,D,oldtype,newtype)` возвращает следующее отображение типа:

`{{(double,4),(char,12),(double,20),(char,28),(double,36),(char,44),
(double,0),(char,8)}}`.

Частичное перекрытие между элементами типа `double` подразумевает, что тип ошибочен, если этот тип данных используется в посылающем действии. Чтобы получать тот же самый тип данных как в примере 1.25, нужно, что бы `D=(64,0)`.

4.6. Структурный строитель типов данных `STRUCT`

`MPI_TYPE_STRUCT(count, array_of_blocklengths,
array_of_displacements,`

array_of_types, newtype)

IN count	количество блоков
IN array_of_blocklengths	количество элементов в каждом блоке
IN array_of_displacements	смещения каждого блока, измеряемые в байтах
IN array_of_types	типы элементов в каждом блоке
OUT newtype	новый тип данных

```
int MPI_TYPE_struct(int count, int *array_of_blocklengths,
    MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
    MPI_Datatype *newtype)
```

MPI_TYPE_STRUCT - наиболее общий строитель типа. Он далее обобщает MPI_TYPE_HINDEXED, и допускает, чтобы каждый блок состоял из дублирований различного типа данных. Для этого имеется массив для описания типов элементов для каждого блока. Действие Struct-строителя представлено схематично на рис. 1.18.

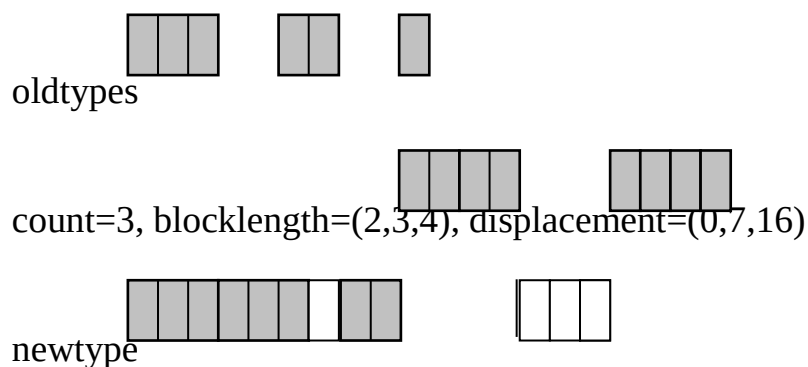


Рис. 1.18. Построение типа данных функцией строителя типов MPI_TYPE_STRUCT

ПРИМЕР 1.27

Допустим type1, имеет отображение типа {(double,0),(char,8)} с диапазоном 16. Допустим B=(2,1,3), D=(0,16,26), и

T=(MPI_FLOAT,type1,MPI_CHAR).

Тогда

ВЫЗОВ

MPI_TYPE_STRUCT(3,B,D,T,newtype) построит новый тип с отображением:

{(float,0),(float,4),(double,16),(char,24),(char,26),(char,27),(char,28)}.

Здесь две копии MPI_FLOAT имеют смещение 0, далее одна копия type1 имеет смещение 16, три копии MPI_CHAR, имеют смещение 26. (Здесь предполагается, что float занимает четыре байта)

4.7. Передача и освобождение типа

В данном случае производный тип данных (datatype) передается операционной системе, а не каким-то пользовательским процессам. Производный тип данных должен быть передан прежде, чем он будет использоваться при обмене данными. Переданный datatype может продолжать использоваться как аргумент входа в строителях типов (так, чтобы другой datatypes мог быть получен из переданного datatype). Примитивный тип данных передавать не нужно.

MPI_TYPE_COMMIT(datatype)

INOUT datatype тип данных, который передается в ОС

int MPI_TYPE_commit(MPI_Datatype *datatype)

MPI_TYPE_COMMIT передает производный тип данных datatype в ОС. Передача не подразумевает, что datatype привязан к текущему содержанию буфера связи. После того, как datatype был передан, он может неоднократно повторно использоваться, чтобы идентифицировать данные.

Объект datatype освобождается запросом к MPI_TYPE_FREE.

MPI_TYPE_FREE(datatype)

INOUT datatype тип данных, который освобождается

int MPI_Type_free(MPI_Datatype *datatype)

MPI_TYPE_FREE регистрирует объект типа данных, связанный с datatype для освобождения и устанавливает datatype к MPI_DATATYPE_NULL. Любая связь, которая в это время (постоянно) использует этот datatype, завершится обычно. Производные типы данных, которые были определены из освобожденного типа данных (datatype), не повреждаются.

1.6.8 Соответствие типов

Предположим, что посылающая функция MPI_Send(buf, count, datatype, dest, tag, comm) выполнена, где datatype имеет отображение типа

$$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},$$

с диапазоном extent. Функция посылает $n \cdot count$ элементов, где элемент (i, j) записан по адресу $addr_{i,j} = buf + extent \cdot i + disp_j$ и имеет тип $type_j$, для $i=0, \dots, count-1$ и $j=0, \dots, n-1$. Переменная, записанная по адресу $addr_{i,j}$ в вызывающей программе должна иметь тип, который соответствует $type_j$, где соответствующий тип определен, как в п. 5.4.1.

Точно так же предположим, что выполнена получающая функция MPI_Recv(buf, count, datatype, source, tag, comm, status). Получающее действие получает до $n \cdot count$ элементов, где элемент (i, j) записан по адресу $buf + extent \cdot i + disp_j$ и имеет тип $type_j$. Соответствующий тип определен согласно наименованию типа передачи datatypes, то есть как последовательность примитивных компонентов типа. Соответствующий тип не зависит от других аспектов определения datatype, типа смещений (расположение в памяти) или промежуточных типов, используемых, чтобы определить datatypes.

Для посылки, datatype может определяться с накладываемыми элементами. Это не верно для получения. Если datatype, используемый в получающем действии определяет накладываемые элементы, то запрос ошибочен.

ПРИМЕР 1.29

Этот пример показывает, что соответствующий тип определен только в терминах примитивных типов, которые составляют производный тип.

```
...
MPI_Type_contiguous(2, MPI_FLOAT, type2, ...);
MPI_Type_contiguous(4, MPI_FLOAT, type4, ...);
MPI_Type_contiguous(2, type2, type22, ...);
...
MPI_Send(a, 4, MPI_FLOAT, ...);
MPI_Send(a, 2, type2, ...);
MPI_Send(a, 1, type22, ...);
MPI_Send(a, 1, type4, ...); ...
MPI_Recv(a, 4, MPI_FLOAT, ...);
MPI_Recv(a, 2, type2, ...);
MPI_Recv(a, 1, type22, ...);
MPI_Recv(a, 1, type4, ...);
```

Любая посылающая функция соответствует любой получающей функции.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Методы распараллеливания и модели программ, поддерживаемые MPI.
2. Свойства MPI, обеспечивающие переносимость параллельных программ и независимость от вычислительных систем.
3. Пояснить основные принципы выполнения ниже указанных функций.
Парные взаимодействия:

- 3.1. MPI_Isend(...), MPI_Irecv(...), MPI_Wait(...), MPI_Test(...);
- 3.2. MPI_Send(...), MPI_Recv(...);

3.3. MPI_Ssend(...);

3.4. MPI_Sendrecv(...);

Коллективные взаимодействия:

3.5 MPI_Barrier(..);

3.6 MPI_Bcast(...);

3.7 MPI_Scatter(...), MPI_Scatterv(...);

3.8 MPI_Gather(...), MPI_Gathrv(...), MPI_Allgather(...);

4. Команда компиляции MPI-программ.

5. Команда запуска MPI-программ.

6. Действия системы по команде mpirun .

7. Задание

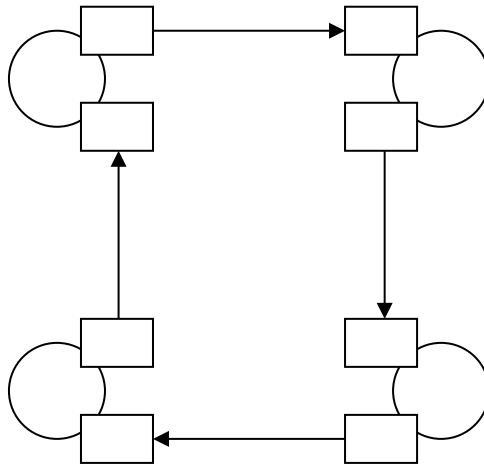
Задание 1. Программирование конвейера.

Ветвь 0 пересылает некоторые данные ветви 1, ветвь 1 принятые данные передает ветви 2, ветвь 2 принятые данные передает ветви 3 и т.д. по цепочке увеличения номеров. И, наконец, ветвь 0 принимает пересылаемые данные от ветви size-1 и выводит на экран свой номер и принятые данные. Т.е. пересылка информации по кольцу компьютеров. В этом примере программу алгоритма нужно написать настраиваемой автоматически на размер вычислительной системы, т.е. алгоритм не должен зависеть от числа компьютеров и программа должна запускаться на любом допустимом количестве компьютеров.

Задание 2. Программирование кольцевых сдвигов данных

Все ветви одновременно пересылают некоторые свои данные ветвям с номерами на единицу большими, чем передающие ветви. Т.е. пересылка информации вдоль кольца компьютеров, см. рисунок ниже. В этом примере программу алгоритма нужно написать настраиваемой автоматически на размер вычислительной системы, т.е. алгоритм не должен зависеть от числа

компьютеров и программа должна запускаться на любом допустимом количестве компьютеров.



8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

- 1) отвечает на контрольные вопросы;
- 2) поясняет ход выполнения индивидуального задания;
- 3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №2. Виртуальные топологии

1. Цель и содержание

Цель лабораторной работы: практическое освоение программирования параллельных процессов, исполняющихся на декартовой топологии связей и топологии "граф".

2. Теоретическая часть

}

```
MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, reord,  
                  &comm_gr);
```

```
/* Каждая ветвь определяет свой собственный номер: от 0 до (size-  
1) */
```

```
MPI_Comm_rank(comm_gr, &rank);
```

```
/* Каждая ветвь генерирует полосы исходных матриц A и B,  
полосы C обнуляет */
```

```
for(i = 0; i < M; j++)  
{ for(j = 0; j < N; i++)  
  { A[j][i] = 3.0;  
    C[j] = 0.0;  
  }  
  B[i] = 2.0;  
}
```

```
/* Засекаем начало умножения матриц */
```

```
t1 = MPI_Wtime();
```

```
/* Каждая ветвь производит умножение своих полос матрицы и  
вектора */
```

```
for(j = 0; j < N; j++)  
{ for(i = 0; i < M; i++)  
  C[j] += A[j][i] * B[i];
```

```

    }

    /* Соединяем части вектора C и записываем их в вектор B во все
компьютеры */

    MPI_Allgather(C, N, MPI_DOUBLE, B, N, MPI_DOUBLE, comm_gr);

    /* Каждая ветвь печатает время решения и нулевая ветвь печатает
вектор B */

    t2 = MPI_Wtime();
    rt = t2 - t1;
    printf("rank = %d Time = %d\n",rank,rt);
    if(rank == 0)
    { for(i = 0; i < M; i++)
        printf("B = %6.2f\n",B[i]);
    }
    MPI_Finalize();
    return(0);
}

```

Обратить внимание на функцию задания топологии граф: MPI_Graph_create, и функцию замера времени счета: MPI_Wtime().

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Для чего нужны виртуальные топологии?
2. Как задаются "декартовы" топологии в MPI?
3. Как задаются топологии "граф" в MPI?
4. Какой метод распараллеливания используется в параллельных алгоритмах умножения матрицы на вектор и матрицы на матрицу с использованием MPI?
5. Как распределяются элементы матрицы и вектора при умножении матрицы на вектор на топологии "кольцо"?
6. Как распределяются элементы обеих матриц при умножении

матрицы на матрицу на топологии "кольцо"?

7. Как лучше представить в памяти компьютера вторую матрицу, при умножении матрицы на матрицу, для ускорения времени решения задачи?

7. Задание

Задание 1. Проработка примеров из пункта "Примеры параллельных программ" этой же лабораторной.

Внимательно изучить примеры 2.1, 2.2, 2.3. Примеры 2.2 и 2.3 (Умножение матрицы на вектор на топологиях "кольцо" и "полный граф") откомпилировать и запустить на 4-х компьютерах.

Задание 2. Параллельное умножение матрицы на вектор на топологии "кольцо".

Разработать алгоритм написать и отладить параллельную программу умножения матрицы на вектор в топологии "кольцо": $A \times B = C$, при условии, что количество строк матрицы и элементов вектора нацело не делится на количество компьютеров. Например, матрица A размером $[22 \times 22]$ и вектор B размером $[22]$ на четырех компьютерах.

Задание 3. Параллельное умножение матрицы на матрицу на топологии "кольцо".

Вариант 1.

Разработать алгоритм написать и отладить параллельную программу умножения матрицы на матрицу в топологии "кольцо": $A \times B = C$, при условии, что количество строк матрицы A и столбцов матрицы B нацело делится на количество компьютеров. Например, матрица A размером $[20 \times 20]$ и матрица B размером $[20 \times 20]$ на четырех компьютерах.

Вариант 2.

Разработать алгоритм написать и отладить параллельную программу умножения матрицы на матрицу в топологии "кольцо": $A \times B = C$, при условии, что количество строк матрицы A и столбцов матрицы B нацело не делится на количество компьютеров. Например, матрица A размером $[22 \times 20]$ и матрица B размером $[20 \times 22]$ на четырех компьютерах.

Задание 4. Сравнительные временные характеристики двух алгоритмов умножения матрицы на вектор на топологиях "кольцо" и "полный граф".

В качестве параллельных программ взять примеры 2.2 и 2.3 (Умножение матрицы на вектор на топологиях "кольцо" и "полный граф"). Для обеих программ построить небольшие графики зависимости времени решения задачи от размеров матрицы и вектора. Обе программы запустить последовательно для матриц: $A = [300 \times 300]$, $[500 \times 500]$, $[700 \times 700]$, $[1000 \times 1000]$ и соответствующих этим матрицам векторов. Откомпилировать и запустить на 4-х компьютерах, засечь время и построить указанные графики.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

- 1) отвечает на контрольные вопросы;
- 2) поясняет ход выполнения индивидуального задания;
- 3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №3. Основные директивы OpenMP

1. Цель и содержание

Цель лабораторной работы: дать представление о построении простых параллельных программ на языке параллельного программирования OpenMP; закрепить практическое освоение директив языка.

2. Теоретическая часть

Что такое OpenMP?

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP, ccNUMA, etc.) в модели общей памяти (shared memory model). В

стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды. Примерами систем с общей памятью, масштабируемых до большого числа процессоров, могут служить суперкомпьютеры Cray Origin2000 (до 128 процессоров), HP 9000 V-class (до 32 процессоров в одном узле, а в конфигурации из 4 узлов - до 128 процессоров), Sun Starfire (до 64 процессоров).

Где найти информацию?

Основной источник информации - сервер <http://www.openmp.org/>. На сервере доступны спецификации, статьи, учебные материалы, ссылки.

Базовый потоковый параллелизм

Общедоступный процесс памяти может состоять из множественных потоков, несколько нитей управления, которые имеют общее адресное пространство, но разные потоки команд и отдельные стеки. В простейшем

случае, процесс состоит из одной нити. Нити иногда называют также потоками, легковесными процессами, LWP (light-weight processes).. OpenMP основан на существовании множественных потоков в общедоступной памяти, программирующей парадигму.

Явный Параллелизм

OpenMP имеет явную (не автоматический) модель программирования, предлагая программисту полное управление по распараллеливанию.

Модель Fork-Join (Ветвление – Объединение)

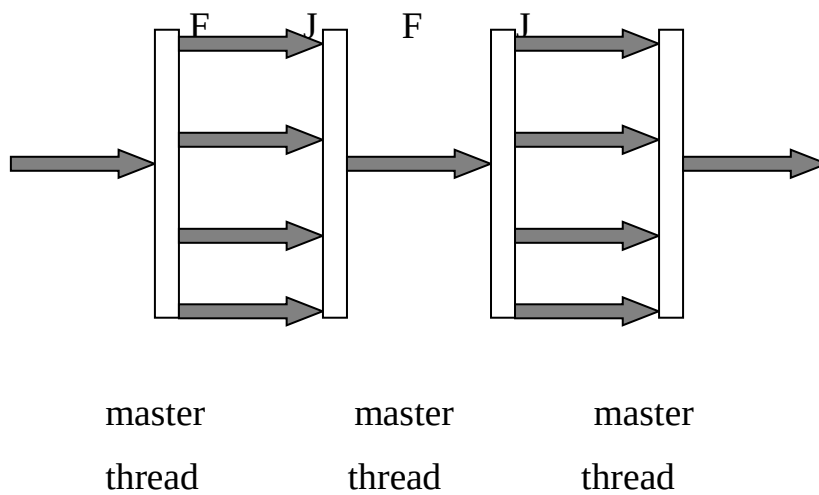
OpenMP использует Fork-Join модель параллельного выполнения:

Все программы OpenMP начинаются как единственный процесс: главный поток. Главный поток выполняется последовательно, пока не сталкиваются с первой областью параллельной конструкции.

Fork (ВЕТВЛЕНИЕ): главный поток создает группу параллельных потоков.

Инструкции в программе, которые включены параллельной конструкцией области{*региона*}, тогда выполнены параллельно среди различных потоков группы

Join (ОБЪЕДИНЕНИЕ): Когда потоки группы завершают инструкции в области параллельной конструкции, они синхронизируются и закрываются, оставляя только главный поток.



2. Директивы

Директивы OpenMP с точки зрения языка C являются командами препроцессора:

`#pragma omp`

Фортрана являются комментариями и начинаются с комбинации символов:

`!$OMP.`

Формат директивы на C/C++:

<команда для препроцессора> <имя директивы>
 <предложение(klausu)>

`#pragma omp parallel [clause clause ...]`
`{`

```
.  
.   
.   
}
```

Директивы можно разделить на 3 категории:

- определение параллельной секции,
- разделение работы,
- синхронизация.

Каждая директива может иметь несколько дополнительных атрибутов -
клауз. Отдельно специфицируются клаузы для назначения классов
переменных, которые могут быть атрибутами различных директив.

Порождение нитей

#pragma omp parallel [clause clause ...]

```
{  
.   
.   
.   
}
```

!\$OMP PARALLEL ... END PARALLEL

clause: if(условие) – выполнение параллельной секции по условию;

private(list) – список переменных локальных в каждой нити;

shared(list) - список переменных общих для каждой нити;

firstprivate(list) - список переменных, которые становятся

локальными

в каждой нити со значениями, ранее присвоенными

этим переменным;
copyin(list) - список переменных (массивов), которые определены
#pragma omp threadprivate(list), и которые создаются
в каждой нити;
reduction(operator:list) - список переменных, с которыми
выполняются
операции обобщенно по всем нитям.

list: - список переменных;
operator: - +,-,*,.,. . . .

Определяет параллельную область программы. При входе в эту область порождаются новые (N-1), образуется "команда" из N нитей, а порождающая нить получает номер 0 и становится основной нитью команды (т.н. "master thread"). При выходе из параллельной области основная нить дожидается завершения остальных нитей, и продолжает выполнение в одном экземпляре. Предполагается, что в SMP-системе нити

будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы).

Каким образом между порожденными нитями распределяется работа - определяется директивами DO,SECTIONS и SINGLE. Возможно также явное управление распределением

работы (а-ля MPI) с помощью функций, возвращающих номер текущей нити и общее число нитей. По умолчанию (вне этих директив), код внутри PARALLEL выполняется всеми нитями одинаково.

Параллельные области могут динамически вложенными. По умолчанию (если вложенный параллелизм не разрешен явно), внутренняя параллельная область выполняется одной нитью.

Разделение работы (work-sharing constructs)

Параллельные циклы

```
#pragma omp parallel [clause clause ...]
```

```
{ ...
```

```
...
}
```

```
#pragma omp for [clause clause ...]
```

```
{ ...
```

```
...
}
```

```
#pragma omp for [clause clause ...]
```

```
{ ...
```

```
}
```

```
...
```

```
...
```

```
}
```

```
!$OMP DO ... [ENDDO]
```

```
clause:  schedule(type[,chink])
```

```
ordered
```

```
private(list)
```

```
shared(list)
```

```
firstprivate(list)
```

```
lastprivate(list)
```

```
reduction(operator:list)
```

```
nowait
```

Определяет параллельный цикл.

Клауза schedule определяет способ распределения итераций по нитям:

static,m - статически, блоками по m итераций

dynamic,m - динамически, блоками по m (каждая нить берет на выполнение первый

еще невзятый блок итераций)

guided,m - размер блока итераций уменьшается экспоненциально до величины m

runtime - выбирается во время выполнения.

ordered – (для циклов) реализуется последовательное выполнение витков цикла, как в последовательном алгоритме.

lastprivate(list) – переменным присваивается результат последнего витка цикла.

По умолчанию, в конце цикла происходит неявная синхронизация; эту синхронизацию

можно запретить с помощью nowait (ENDDO NOWAIT).

Параллельные секции

```
#pragma omp parallel [clause clause ...]
```

```
{ ...
```

```
...
```

```
#pragma omp sections [clause clause ...]
```

```
{ ...
```

```
...
```

```
#pragma omp section
```

```
{ ...
```

```
...
```

```

    }
#pragma omp section
    { ...
    ...
    }
} // – end sections
...
...
}

```

!\$OMP SECTIONS ... END SECTIONS

```

clause:  private(list)
         firstprivate(list)
         lastprivate(list)
         reduction(operator:list)
         nowait

```

Не-итеративная параллельная конструкция. Определяет набор независимых секций

кода (т.н., "конечный" параллелизм). Секции отделяются друг от друга директивой

section.

Примечание. Если внутри parallel содержится только одна конструкция for или

только одна конструкция section, то можно использовать укороченную запись:

parallel for или parallel section.

Исполнение одной нитью

```
#pragma omp parallel [clause clause ...]  
{ ...  
  ...  
#pragma omp single [clause clause ...]  
{ ...  
  ...  
#pragma omp single [clause clause ...]  
{ ...  
  ...  
}  
}
```

!\$OMP SECTIONS ... END SECTIONS

```
clause:  private(list)  
         firstprivate(list)  
         nowait
```

SINGLE ... END SINGLE

Определяет блок кода, который будет исполнен только одной нитью (первой, которая дойдет до этого блока).

Явное управление распределением работы

С помощью функций `omp_get_thread_num()` и `omp_get_num_threads()` нить может узнать свой номер и общее число нитей, а затем выполнять свою часть работы в зависимости от своего номера (этот подход широко используется в программах на базе интерфейса MPI).

Пример

Простой пример: вычисление числа "Пи". В последовательную программу вставлены две строчки, и она распараллелена!

C:

```
#include<omp.h>
#include<stdio.h>

double f(double y)
{ return(4.0/(1.0+y*y));
}

main()
{ double w,x,sum,pi;
  int i;
  int n = 1000;
  w = 1.0/n;
  sum = 0.0;
  #pragma omp parallel for schedule(static,n/2) private(i,x) \
  shared(w) reduction(+:sum)
  for(i=0; i < n; i++)
    { x = w*(i-0.5);
```

```

        sum = sum + f(x);
    }
    pi = w*sum;
    printf("pi = %f\n",pi);

return(0);
}

```

Здесь переменная `sum` не должна описываться в `private(i,x)`, но может быть описана в `shared(w)`.

Fortran:

```

program compute_pi
    parameter (n = 1000)
    integer i
    double precision w,x,sum,pi,f,a
    f(a) = 4.d0/(1.d0+a*a)
    w = 1.0d0/n
    sum = 0.0d0;
    !$OMP PARALLEL DO PRIVATE(x) SHARED(w) REDUCTION(+:sum)
    do i=1,n
        x = w*(i-0.5d0)
        sum = sum + f(x)
    enddo

    pi = w*sum
    print *, 'pi = ',pi
    stop
end

```

Директивы синхронизации

#pragma omp master

```
{  
.  
}
```

MASTER ... END MASTER

Определяет блок кода, который будет выполнен только master-ом (нулевой нитью).

Вход/выход из критического интервала запрещены.

#pragma omp critical[name]

```
{  
.  
}
```

CRITICAL ... END CRITICAL

Определяет критическую секцию, то есть блок кода, который не должен выполняться

одновременно двумя или более нитями.

name – имя критического интервала. Имя – глобальный идентификатор.

Различные критические интервала с одним именем обрабатываются как одна и та же область (критический интервал).

Вход/выход из критического интервала запрещены.

#pragma omp barrier

BARRIER

Определяет точку барьерной синхронизации, в которой каждая нить дожидается всех

остальных. Наименьшая инструкция, которая содержит barrier должна быть структурным блоком. barrier не может быть в for и sections.

Неверно:

```
for(x == 0)
    #pragma omp barrier
```

Верно:

```
for(x == 0)
{
    #pragma omp barrier
}
```

#pragma omp atomic

x binop = expr;

x++

++x

x--

--x

binop - +, *, -, /, &, ^, \, >, <, or;

expr – скалярное выражение.

ATOMIC

Определяет переменную в левой части оператора "атомарного" присваивания, которая должна корректно обновляться несколькими нитями.

#pragma omp ordered

ORDERED ... END ORDERED

Определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле. Может использоваться для упорядочения вывода от параллельных нитей.

#pragma omp flush(list)

FLUSH

Явно определяет точку, в которой реализация должна обеспечить одинаковый вид памяти для всех нитей. Неявно FLUSH присутствует в следующих директивах: BARRIER, CRITICAL, END CRITICAL, DO, END DO, PARALLEL, END PARALLEL, SECTIONS, END SECTIONS, SINGLE,

END SINGLE, ORDERED, END ORDERED. В целях синхронизации можно также пользоваться механизмом замков (locks).

3. Классы переменных

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

SHARED (общие; под именем A все нити видят одну переменную) и PRIVATE (приватные; под именем A каждая нить видит свою переменную).

Отдельные правила определяют поведение переменных при входе и выходе из параллельной области или параллельного цикла: REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYIN.

По умолчанию, все COMMON-блоки, а также переменные, порожденные вне параллельной

области, при входе в эту область остаются общими (SHARED). Исключение составляют

переменные - счетчики итераций в цикле, по очевидным причинам. Переменные, порожденные внутри параллельной области, являются приватными (PRIVATE). Явно назначить класс переменных по умолчанию можно с помощью клаузы DEFAULT.

shared(list)

SHARED

Применяется к переменным, которые необходимо сделать общими.

private(list)

PRIVATE

Применяется к переменным, которые необходимо сделать приватными. При входе в параллельную область для каждой нити создается отдельный экземпляр переменной, который не имеет никакой связи с оригинальной переменной вне параллельной области.

#pragma omp thredprivate(list)

THREADPRIVATE

Применяется к COMMON-блокам, которые необходимо сделать приватными. Директива должна применяться после каждой декларации COMMON-блока.

```
double A[[]],B[];
```

```
#pragma omp thredprivate(A,B)
```

firstprivate

FIRSTPRIVATE

Приватные копии переменной при входе в параллельную область инициализируются значением оригинальной переменной.

lastprivate

LASTPRIVATE

По окончании параллельно цикла или блока параллельных секций, нить, которая выполнила последнюю итерацию цикла или последнюю секцию блока, обновляет значение оригинальной переменной.

reduction(+:A)

REDUCTION(+:A)

Обозначает переменную, с которой в цикле производится reduction-операция (например, суммирование). При выходе из цикла, данная операция производится над копиями переменной во всех нитях, и результат присваивается оригинальной переменной. Переменные в этом списке не должны описываться в `private(list)`, но могут быть описаны в `shared(list)`.

copyin(list)

COPYIN

Применяется к COMMON-блокам, которые помечены как `threadprivate`. При входе в параллельную область приватные копии этих данных инициализируются оригинальными значениями.

4. Runtime-процедуры и переменные среды

В целях создания переносимой среды запуска параллельных программ, в OpenMP

определен ряд переменных среды, контролирующих поведение приложения.

В OpenMP предусмотрен также набор библиотечных процедур, которые позволяют:

- во время исполнения контролировать и запрашивать различные параметры, определяющие поведение приложения (такие как число нитей и процессоров, возможность вложенного параллелизма);
- процедуры назначения параметров имеют приоритет над соответствующими переменными среды.
- использовать синхронизацию на базе замков (locks).

Переменные среды

export OMP_SCHEDULE “guided,4”

export OMP_SCHEDULE “dynamic”

Определяет способ распределения итераций в цикле, если в директиве DO использована клауза SCHEDULE(RUNTIME).

export OMP_NUM_THREADS=n (= 4, setenv)

Определяет число нитей для исполнения параллельных областей приложения.

export OMP_DYNAMIC TRUE

Разрешает или запрещает динамическое изменение числа нитей.

export OMP_NESTED TRUE

Разрешает или запрещает вложенный параллелизм.

Процедуры для контроля/запроса параметров среды исполнения

omp_set_num_threads(int num)

OMP_SET_NUM_THREADS

Позволяет назначить максимальное число нитей для использования в следующей параллельной области (если это число разрешено менять динамически). Вызывается из последовательной области программы.

omp_get_max_threads()

OMP_GET_MAX_THREADS

Возвращает максимальное число нитей.

omp_get_num_threads()

OMP_GET_NUM_THREADS

Возвращает фактическое число нитей в параллельной области программы.

omp_get_num_procs()

OMP_GET_NUM_PROCS

Возвращает число процессоров, доступных приложению.

omp_in_parallel()

OMP_IN_PARALLEL

Возвращает .TRUE., если вызвана из параллельной области программы.

omp_set_dynamic(int dynamic_threads)/omp_get_dynamic()

OMP_SET_DYNAMIC / OMP_GET_DYNAMIC

Устанавливает/запрашивает состояние флага, разрешающего динамически изменять число нитей.

omp_set_nested(int nested)/omp_get_nested()

OMP_GET_NESTED / OMP_SET_NESTED

Устанавливает/запрашивает состояние флага, разрешающего вложенный параллелизм.

Процедуры для синхронизации на базе замков

В качестве замков используются общие переменные типа INTEGER (размер должен быть достаточным для хранения адреса). Данные переменные должны использоваться только как параметры примитивов синхронизации.

omp_init_lock(omp_lock_t *lock)/ omp_destroy_lock(omp_lock_t *lock)

OMP_INIT_LOCK(var) / OMP_DESTROY_LOCK(var)

Инициализирует замок, связанный с переменной var.

omp_set_lock(omp_lock_t *lock)

OMP_SET_LOCK(VAR)

Заставляет вызвавшую нить дождаться освобождения замка, а затем захватывает его.

omp_unset_lock(omp_lock_t *lock)

OMP_UNSET_LOCK(VAR)

Освобождает замок, если он был захвачен вызвавшей нитью.

omp_test_lock(omp_lock_t *lock)

OMP_TEST_LOCK(VAR)

Пробует захватить указанный замок. Если это невозможно, возвращает .FALSE.

5. Спецификация OpenMP для языков C/C++

Спецификация OpenMP для C/C++, выпущенная на год позже фортранной, содержит в

основном аналогичную функциональность.

Необходимо лишь отметить следующие моменты:

1) Вместо спецкомментариев используются директивы компилятора "#pragma omp".

2) Компилятор с поддержкой OpenMP определяет макрос "_OPENMP", который может

использоваться для условной компиляции отдельных блоков, характерных для

параллельной версии программы.

3) Распараллеливание применяется к for-циклам, для этого используется директива

"#pragma omp for". В параллельных циклах запрещается использовать оператор

break.

4) Статические (static) переменные, определенные в параллельной области

программы, являются общими (shared).

5) Память, выделенная с помощью malloc(), является общей (однако указатель на

нее может быть как общим, так и приватным).

6) Типы и функции OpenMP определены во включаемом файле <omp.h>.

7) Кроме обычных, возможны также "вложенные" (nested) замки - вместо логических

переменных используются целые числа, и нить, уже захватившая замок, при

повторном захвате может увеличить это число.

Пример распараллеливания for-цикла в C

```
#pragma omp parallel for private(i)
#pragma omp shared(x, y, n) reduction(+: a, b)
for (i=0; i<n; i++)
{
```

```
a = a + x[i];  
b = b + y[i];  
}
```

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.

4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Методы распараллеливания и модели программ, поддерживаемые OpenMP.

2. Пояснить основные принципы выполнения ниже указанных директив.

3. Как задаются параллельные процессы при умножении матрицы на вектор и матрицы на матрицу в OpenMP?

7. Задание

Разработать алгоритм написать и отладить параллельную программу умножения матрицы на вектор с использованием распределения работ для параллельных процессов директивой **sections**. Использовать алгоритм примера 3.5.

Вариант 2.

Разработать алгоритм написать и отладить параллельную программу умножения матрицы на вектор с использованием распределения работ для параллельных процессов с непосредственным заданием работы («вручную»). Использовать алгоритм примера 3.5.

Задание 3. Параллельное умножение матрицы на матрицу.

Вариант 1.

Разработать алгоритм написать и отладить параллельную программу умножения матрицы на матрицу с использованием директивы распараллеливания цикла по виткам.

Вариант 2.

Разработать алгоритм написать и отладить параллельную программу умножения матрицы на матрицу с использованием распределения работ для параллельных процессов с непосредственным заданием работы («вручную»).

Задание 4. Сравнительные временные характеристики двух алгоритмов умножения матрицы на вектор на языке MPI и языке OpenMP.

В качестве параллельных программ взять параллельные программы: «умножение матрицы на вектор в MPI на топологии "кольцо"» и «умножение матрицы на вектор в OpenMP (один из вариантов)». Для обеих программ построить небольшие графики зависимости времени решения задачи от размеров матрицы и вектора. Обе программы запустить последовательно для матриц: $A = [300 \times 300]$, $[500 \times 500]$, $[700 \times 700]$, $[1000 \times 1000]$ и соответствующих этим матрицам векторов. Откомпилировать и запустить на 2-х компьютерах, засечь время и построить указанные графики.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

- 1) отвечает на контрольные вопросы;
- 2) поясняет ход выполнения индивидуального задания;
- 3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №4. Решение систем линейных алгебраических уравнений итерационными методами

1. Цель и содержание

Цель лабораторной работы: дать представление о распараллеливании алгоритмов решения систем линейных алгебраических уравнений итерационными методами на языках MPI и OpenMP.

2. Теоретическая часть

Выбирается начальное приближение x_0 . $r_0 = f - Ax_0$; $z_0 = r_0$;

$\alpha_k = (r_{k-1}, r_{k-1}) / (Az_{k-1}, z_{k-1})$ - коэффициент (здесь k – это номер итерационного шага);

$x_k = x_{k-1} + \alpha_k z_{k-1}$ - вектор решений на k -й текущей итерации;

$r_k = r_{k-1} - \alpha_k Az_{k-1}$ - вектор невязки на k -й текущей итерации;

$\beta_k = (r_k, r_k) / (r_{k-1}, r_{k-1})$ - коэффициент;

$z_k = r_k + \beta_k z_{k-1}$ - вектор спуска на k -й текущей итерации (сопряженное направление).

Выход из итерационного процесса:

$$\frac{\|r_k\|}{\|f\|} < \varepsilon \quad (7)$$

Ниже приведена программа для последовательного алгоритма.

Обратите на то как вычисляются скалярные произведения векторов.

```
#include<stdio.h>
```

```
#include<time.h>
```

```
#include<sys/time.h>
```

```
#define M 100
```

```
#define E 0.00001
```

```
#define T 0.01
```

```
double A[M][M], F[M], Xk[M], Zk[M];
```

```
double Rk[M], Sz[M], alf, bet, mf;
```

```
double Spr, Spr1, Spz;
```

```
int main()
```

```
{ int i, j, v;
```

```
  struct timeval tv1, tv2;
```

```
  long int dt1;
```

```
/* Генерация данных. Здесь задается матрица с элементами равными 1,  
* по диагонали равными 2. Матрица берется хорошо обусловленной и  
* симметричной. При правой части равной M+1 все корни будут  
равными 1 */
```

```
  for(mf=0,i = 0; i < M; i++)
```

```
  { for(j = 0; j < M; j++)
```

```
    { if(i == j)
```

```
      A[i][j] = 2.0;
```

```
    else
```

```
      A[i][j] = 1.0;
```

```
    }
```

```
    F[i] = M + 1;
```

```
/* Сразу вычисляем сумму квадратов элементов вектора F,
```

```
* т.е. подкоренное выражение формулы (7). */
```

```
    mf += F[i] * F[i];
```

```
  }
```

/* Задаем начальное приближение корней. В X_k хранятся значения корней

* к-й итерации. */

for(i = 0; i < M; i++)

$X_k[i] = 0.2;$

/* Задаем начальное значение r_0 и z_0 . */

for(i = 0; i < M; i++)

{ for(Sz[i]=0, j = 0; j < M; j++)

$Sz[i] += A[i][j] * X_k[j];$

$Rk[i] = F[i] - Sz[i];$

$Zk[i] = Rk[i];$

}

/* Засекаем время начала вычислений. */

gettimeofday(&tv1, NULL);

do

{

/* Вычисляем числитель и знаменатель для коэффициента

* $\alpha_k = (r_{k-1}, r_{k-1}) / (A z_{k-1}, z_{k-1})$ */

$Spz = 0;$

$Spr = 0;$

for(i = 0; i < M; i++)

{ for(Sz[i]=0, j = 0; j < M; j++)

$Sz[i] += A[i][j] * Zk[j];$

$Spz += Sz[i] * Zk[i];$

$Spr += Rk[i] * Rk[i];$

}

```
alf = Spr/Spz;          /*  $\alpha_k$  */
```

```
/* Вычисляем вектор решения:  $x_k = x_{k-1} + \alpha_k z_{k-1}$ ,
```

```
вектор невязки:  $r_k = r_{k-1} - \alpha_k A z_{k-1}$  и числитель для  $\beta_k$  равный  $(r_k, r_k)$  */
```

```
Spr1 = 0;
```

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

```
{ Xk[i] += alf*Zk[i];
```

```
Rk[i] -= alf*Sz[i];
```

```
Spr1 += Rk[i]*Rk[i];
```

```
}
```

```
/* Вычисляем  $\beta_k$  */
```

```
bet = Spr1/Spr;
```

```
/* Вычисляем вектор спуска:  $z_k = r_k + \beta_k z_{k-1}$  */
```

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

```
Zk[i] = Rk[i] + bet*Zk[i];
```

```
}
```

```
/* Проверяем условие выхода из итерационного цикла */
```

```
while(Spr1/mf > E*E);
```

```
/* Засекаем время конца вычислений. Время решения задачи выводим  
на экран */
```

```
gettimeofday(&tv2, (struct timezone*)0);
```

```
dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec;
```

```
printf("Time = %d\n",dt1);
```

```
/* Для контроля выводим 8 первых корней */
```

```
printf("      %f      %f      %f      %f      %f      %f      %f      %f  
%f\n",Xk[0],Xk[1],Xk[2],Xk[3],Xk[4],Xk[5],  
Xk[6], Xk[7]);
```

```
return(0);  
}
```

ЗАДАНИЕ

Тщательно изучить программы примеров. Скомпилировать и запустить все программы примеров на одном компьютере.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Какой метод распараллеливания используется в параллельных алгоритмах решения СЛАУ методами простой итерации и сопряженных градиентов в MPI?
2. Как распределяются элементы матрицы коэффициентов и вектора решений при решении СЛАУ методом простой итерации в MPI?
3. Как распределяются элементы матрицы коэффициентов и вектора решений при решении СЛАУ методом сопряженных градиентов в MPI?
4. Как распараллеливаются алгоритмы решения СЛАУ методами простой итерации и сопряженных градиентов в OpenMP?

7. Задание

Задание 1. Проработка примеров из пункта "Примеры параллельных программ" этой же лабораторной.

Внимательно изучить примеры 4.1-4.2. Откомпилировать и запустить на 1-м процессоре.

Задание 2. Параллельный алгоритм решения СЛАУ методом простой итерации в MPI

Разработать параллельный алгоритм, написать и отладить параллельную программу решения СЛАУ методом простой итерации в MPI. Использовать программу для последовательного алгоритма в примере 4.1.

Задание 3. Параллельный алгоритм решения СЛАУ методом сопряженных градиентов в MPI

Разработать параллельный алгоритм, написать и отладить параллельную программу решения СЛАУ методом сопряженных градиентов в MPI. Использовать программу для последовательного алгоритма в примере 4.2.

Задание 4. Параллельный алгоритм решения СЛАУ методом простой итерации в OpenMP

Разработать параллельный алгоритм, написать и отладить параллельную программу решения СЛАУ методом простой итерации в OpenMP. Использовать программу для последовательного алгоритма в примере 4.1.

Задание 5. Параллельный алгоритм решения СЛАУ методом сопряженных градиентов в OpenMP

Разработать параллельный алгоритм, написать и отладить параллельную программу решения СЛАУ методом сопряженных градиентов в OpenMP. Использовать программу для последовательного алгоритма в примере 4.2.

Задание 6. Сравнительные временные характеристики двух алгоритмов решения СЛАУ методом простой итерации в языке MPI и языке OpenMP.

В качестве параллельных программ взять параллельные программы заданий 2 и 4. Для обеих программ построить небольшие графики зависимости времени решения задачи от размеров матрицы коэффициентов. Обе программы запустить последовательно для матриц: $A = [300 \times 300]$, $[500 \times 500]$, $[700 \times 700]$, $[1000 \times 1000]$. Откомпилировать и запустить на 2-х компьютерах, засечь время и построить указанные графики.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

- 1) отвечает на контрольные вопросы;
- 2) поясняет ход выполнения индивидуального задания;
- 3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №5. Решение систем линейных алгебраических уравнений методом Гаусса

1. Цель и содержание

Цель лабораторной работы: практическое освоение методов решения систем линейных алгебраических уравнений прямыми методами.

2. Теоретическая часть

Требуется найти решение системы линейных алгебраических уравнений:

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = f_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = f_2$$

.....

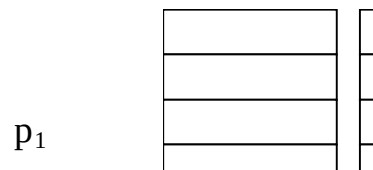
$$a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = f_n$$

Метод Гаусса основан на последовательном исключении неизвестных. Здесь рассматриваются два алгоритма решения СЛАУ методом Гаусса. Они связаны с разными способами представления *данных* (матриц коэффициентов и правых частей) в распределенной памяти мультимикрокомпьютера. Схемы распределения данных по микрокомпьютерам для обоих примеров приведены ниже. Хотя данные распределены в памяти мультимикрокомпьютера в каждом алгоритме по-разному, но оба они реализуются на одной и той же топологии связи микрокомпьютеров - "полный граф".

ПРИМЕР 5.1. Решение СЛАУ методом Гаусса. Алгоритм первый

В алгоритме, представленном в данном пункте, исходная матрица коэффициентов A и вектор правых частей F разрезаны горизонтальными полосами, как показано на рис. 5.1. Каждая полоса загружается в соответствующий микрокомпьютер: нулевая полоса – в нулевой микрокомпьютер, первая полоса – в первый микрокомпьютер, и т. д., последняя полоса – в $p-1$ микрокомпьютер.

Здесь, в примере, каждая ветвь генерирует свои части матрицы коэффициентов A и вектор правых частей F .



A F

Рис. 5.1. Разрезание *данных* для параллельного алгоритма 1 решения СЛАУ методом Гаусса

При прямом ходе матрица приводится к треугольному виду последовательно по компьютерам. Вначале к треугольному виду приводятся строки в нулевом компьютере, при этом нулевой компьютер последовательно, строка за строкой, передает свои строки остальным компьютерам, начиная с первого. Затем к треугольному виду приводятся строки в первом компьютере, передавая свои строки остальным компьютерам, начиная со второго, т.е. компьютерам с большими номерами, и т. д. Процесс деления строк на коэффициенты при x_i не требует информации от других компьютеров.

После прямого хода полосы матрицы A в каждом узле будут иметь вид (рис. 5.2)

0	1\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$	1	00001\$\$\$\$\$\$\$\$\$\$\$\$
	01\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$		000001\$\$\$\$\$\$\$\$\$\$\$\$
	001\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$		0000001\$\$\$\$\$\$\$\$\$\$\$\$
	0001\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$		00000001\$\$\$\$\$\$\$\$\$\$\$\$
2	000000001\$\$\$\$\$\$\$\$	3	00000000000001\$\$\$
	00000000001\$\$\$\$\$\$\$		000000000000001\$\$
	000000000001\$\$\$\$\$\$		0000000000000001\$
	0000000000001\$\$\$\$\$		00000000000000001

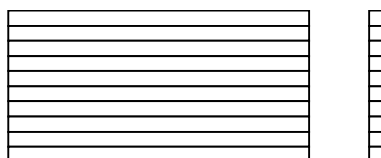
Рис. 5.2. Вид полос после прямого хода в алгоритме 1 решения СЛАУ методом Гаусса. Пример приведен для четырех узлов; \$ – вещественные числа.

Аналогично, последовательно по компьютерам, начиная с последнего по номеру компьютера, осуществляется обратный ход.

Особенностью этого алгоритма является то, что как при прямом, так и при обратном ходе, компьютеры, завершившие свою часть работы, переходят в состояние ожидания, пока не завершат эту работу другие компьютеры. Таким образом, вычислительная нагрузка распределяется по компьютерам неравномерно, не смотря на то, что *данные* изначально распределяются по компьютерам приблизительно одинаково. Простои компьютеров значительно уменьшаются при распределении матрицы циклическими горизонтальными полосами. Этот метод представлен в п. 3.4.2.

ПРИМЕР 5.2. Решение СЛАУ методом Гаусса. Алгоритм второй

В алгоритме, представленном здесь, исходная матрица коэффициентов распределяется по компьютерам циклическими горизонтальными полосами с шириной полосы в одну строку, как показано ниже на рис. 5.3.



А F

Рис. 5.3. Разрезание *данных* для параллельного алгоритма 2 решения СЛАУ методом Гаусса

Нулевая строка матрицы помещается в компьютер 0, первая строка – в компьютер 1, и т. д., (p_1-1) -я строка в компьютер p_1-1 (где p_1 количество компьютеров в системе). Затем, p_1 -я строка, снова помещается в компьютер 0, p_1+1 -я строка – в компьютер 1, и т. д.

При таком распределении данных, соответствующим этому распределению должен быть и алгоритм. Строку, которая вычитается из всех остальных строк (после предварительного деления на нужные коэффициенты), назовем текущей строкой. Алгоритм прямого хода заключается в следующем. Сначала текущей строкой является строка с индексом 0 в компьютере 0, затем строка с индексом 0 в компьютере 1 (здесь не нужно путать общую нумерацию строк во всей матрице и индексацию строк в каждом компьютере; в каждом компьютере индексация строк в массиве начинается с нуля) и т. д., и наконец, строка с индексом 0 в последнем по номеру компьютере. После чего цикл по компьютерам повторяется и текущей строкой становится строка с индексом 1 в компьютере 0, затем строка с индексом 1 в компьютере 1 и т. д. После прямого хода полосы матрицы в каждом компьютере будут иметь вид,

показанный на рис. 5.4. Пример приведен для четырех узлов; \$ – вещественные числа.

Аналогично, последовательно по узлам, начиная с последнего по номеру компьютера, осуществляется обратный ход.

Особенностью этого алгоритма является то, что как при прямом, так и при обратном ходе компьютеры являются более равномерно загруженными, чем в первом методе. Значит, и вычислительная нагрузка распределяется по компьютерам более равномерно, чем в первом методе. Например, нулевой компьютер, завершив обработку своих строк при прямом ходе, ожидает, пока другие компьютеры обработают только по одной, оставшейся у них не обработанной строке, а не полностью обработают полосы, как в первом алгоритме.

0	1 \$\$\$\$\$\$\$\$\$\$\$\$\$\$	1	0 1 \$\$\$\$\$\$\$\$\$\$\$\$\$\$
	0 0 0 0 1 \$\$\$\$\$\$\$\$\$\$		0 0 0 0 0 1 \$\$\$\$\$\$\$\$\$\$
	0 0 0 0 0 0 0 0 1 \$\$\$\$\$\$		0 0 0 0 0 0 0 0 0 1 \$\$\$\$\$\$
	0 0 0 0 0 0 0 0 0 0 0 1 \$\$\$		0 0 0 0 0 0 0 0 0 0 0 0 0 1 \$\$\$
2	0 0 1 \$\$\$\$\$\$\$\$\$\$\$\$\$\$	3	0 0 0 1 \$\$\$\$\$\$\$\$\$\$\$\$\$\$
	0 0 0 0 0 0 1 \$\$\$\$\$\$\$\$\$\$		0 0 0 0 0 0 0 1 \$\$\$\$\$\$\$\$\$\$
	0 0 0 0 0 0 0 0 0 1 \$\$\$\$\$\$		0 0 0 0 0 0 0 0 0 0 1 \$\$\$\$\$\$
	0 0 0 0 0 0 0 0 0 0 0 0 0 1 \$		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Рис. 5.4. Вид полос после прямого хода в алгоритме 2 решения СЛАУ методом Гаусса

Ниже приведена программа решения СЛАУ методом Гаусса для последовательного алгоритма:

```
#include<stdio.h>
#include<sys/time.h>

#define M 100
```

```
double MA[M][M+1], MAD;
```

```
int main()
```

```
{ int i, j, v, k;
```

```
/* Переменные для замера времени решения */
```

```
struct timeval tv1, tv2;
```

```
long int dt1;
```

```
/* Генерация данных */
```

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

```
{ for(j = 0; j < M; j++)
```

```
{ if(i == j)
```

```
    MA[i][j] = 2.0;
```

```
else
```

```
    MA[i][j] = 1.0;
```

```
}
```

```
MA[i][M] = 1.0*(M)+1.0;
```

```
}
```

```
gettimeofday(&tv1, NULL);
```

```
/* Прямой ход */
```

```
for(k = 0; k < M; k++)
```

```
{ MAD = 1.0/MA[k][k];
```

```
for(j = M; j >= k; j--)
```

```
    MA[k][j] *= MAD;
```

```
for(i = k+1; i < M; i++)
```

```

        for(j = M; j >= k; j--)
            MA[i][j] -= MA[i][k]*MA[k][j];

    }

/* Обратный ход */
    for(k = M-1; k >= 0; k--)
        { for(i = k-1; i >= 0; i--)
            MA[i][M] -= MA[k][M]*MA[i][k];
        }

/* Засечение времени и печать */
    gettimeofday(&tv2,NULL);
    dt1 = (tv2.tv_sec - tv1.tv_sec)*1000000 + tv2.tv_usec - tv1.tv_usec;
    printf("Time = %ld\n",dt1);

/* Печать первых восьми корней */
    printf(" %f %f %f %f\n",MA[0][M],MA[1][M],MA[2][M],MA[3][M]);
    printf(" %f %f %f %f\n",MA[4][M],MA[5][M],MA[6][M],MA[7][M]);

    return(0);
}

```

Обратите внимание на функцию замера времени gettimeofday(&tv,NULL). Эту функцию можно использовать и для параллельных программ.

ЗАДАНИЕ

Тщательно изучить программы примеров. Скомпилировать и запустить все программы примеров на одном компьютере.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Какой метод распараллеливания используется в параллельных алгоритмах решения СЛАУ методом Гаусса в MPI?
2. Как распределяются элементы матрицы коэффициентов и вектора правых частей при решении СЛАУ методом Гаусса алгоритмом № 1?
3. Как распределяются элементы матрицы коэффициентов и вектора правых частей при решении СЛАУ методом Гаусса алгоритмом № 2?
4. Какой из методов Гаусса является более быстродействующим?

7. Задание

Задание 1. Проработка примера из пункта "Примеры параллельных программ" этой же лабораторной.

Внимательно изучить пример 6.1. Откомпилировать и запустить на 1-м процессоре.

Задание 2. Построение параллельного алгоритма решения задачи Пуассона методом Зейделя в MPI

Разработать параллельный алгоритм, написать и отладить параллельную программу решения задачи Пуассона методом Зейделя в MPI.

Рекомендация: параллельный алгоритм реализовать на «линейке» компьютеров.

Использовать программу для последовательного алгоритма в примере 6.1.

Задание 3. Запуск параллельного алгоритма решения задачи Пуассона для разных функций

Разработанный и отлаженный параллельный алгоритм проверить, например, для функции $f = X^2 * Y^2 * Z^2$.

Задание 4. Сравнительные временные характеристики алгоритма решения задачи Пуассона на одном и нескольких компьютерах.

В качестве параллельной программы взять вами созданную параллельную программу. В качестве параллельной программы, взять последовательную программу из примера предыдущего пункта. Обе программы запустить и засечь время счета для одного и того же пространства размером $20*20*20$. Засечь время решения задачи на 1-м, 2-х и 4-х компьютерах.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

- 1) отвечает на контрольные вопросы;
- 2) поясняет ход выполнения индивидуального задания;
- 3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №6. Решение задачи Пуассона методом Зейделя в трехмерной области

1. Цель и содержание

Цель лабораторной работы: практическое освоение методов распараллеливание алгоритмов задач, решаемых сеточными методами на примере решения задачи Пуассона в трехмерной области.

2. Теоретическая часть

```
struct timeval tv1,tv2;
```

```
it = 0;
```

```
X = 2.0;
```

```
Y = 2.0;
```

```
Z = 2.0;
```

```
e = 0.00001;
```

```
/* Размеры шагов */
```

```
hx = X/in;
```

```
hy = Y/jn;
```

```
hz = Z/kn;
```

```
owx = pow(hx,2);
```

```
owy = pow(hy,2);
```

```
owz = pow(hz,2);
```

```
c = 2/owx + 2/owy + 2/owz + a;
```

```
gettimeofday(&tv1,(struct timezone*)0);
```

```
/* Инициализация границ пространства */
```

```
Inic();
```

```
/* Основной итерационный цикл */
```

```
do
```

```
{ f = 1;
```

```
  for(i = 1; i < in; i++)
```

```
    for(j = 1; j < jn; j++)
```

```
      { for(k = 1; k < kn; k++)
```

```
        { F1 = F[i][j][k];
```

```
          Fi = (F[i+1][j][k] + F[i-1][j][k]) / owx;
```

```
          Fj = (F[i][j+1][k] + F[i][j-1][k]) / owy;
```

```
          Fk = (F[i][j][k+1] + F[i][j][k-1]) / owz;
```

```
          F[i][j][k] = (Fi + Fj + Fk - Ro(i*hx,j*hy,k*hz)) / c;
```

```
          if(fabs(F[i][j][k] - F1) > e)
```

```
            f = 0;
```

```
        }
```

```
    }
```

```
  it++;
```

```
}
```

```
while(f == 0);
```

```
gettimeofday(&tv2,(struct timezone*)0);
```

```
osdt = (tv2.tv_sec - tv1.tv_sec)*1000000 + tv2.tv_usec-tv1.tv_usec;
```

```
printf("\n in = %d  iter = %d E = %f  T = %ld\n",in,it,e,osdt);
```

```
/*      Нахождение      максимального      расхождения      полученного  
приближенного решения
```

```
* и точного решения */
```

```
max = 0.0;
```

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

```

    { for(j = 1; j < jn; j++)
        { for(k = 1; k < kn; k++)
            { if((F1 = fabs(F[i][j][k] - Fresh(i*hx,j*hy,k*hz))) > max)
                { max = F1;
                  mi = i; mj = j; mk = k;
                }
            }
        }
    }

    printf("Max differ = %f\n in point(%d,%d,%d)\n",max,mi,mj,mk);
}

return(0);

}

```

ЗАДАНИЕ

Тщательно изучить программу примера. Скомпилировать и запустить её на одном компьютере.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Какой метод распараллеливания используется для решения задач, использующих в своей реализации сеточные методы?
2. Как можно распределить пространство вычислений 3D на распределенной вычислительной системе?
3. Какой из параллельных алгоритмов решения задачи Пуассона в 3D на ваш взгляд будет более быстродействующим: 1) на линейке компьютеров;

2) на двумерной решетке компьютеров; или 3) на трехмерной решетке компьютеров?

7. Задание

Задание 1. Проработка примера из пункта "Примеры параллельных программ" этой же лабораторной.

Внимательно изучить пример 6.1. Откомпилировать и запустить на 1-м процессоре.

Задание 2. Построение параллельного алгоритма решения задачи Пуассона методом Зейделя в MPI

Разработать параллельный алгоритм, написать и отладить параллельную программу решения задачи Пуассона методом Зейделя в MPI.

Рекомендация: параллельный алгоритм реализовать на «линейке» компьютеров.

Использовать программу для последовательного алгоритма в примере 6.1.

Задание 3. Запуск параллельного алгоритма решения задачи Пуассона для разных функций

Разработанный и отлаженный параллельный алгоритм проверить, например, для функции $f = X^2 * Y^2 * Z^2$.

Задание 4. Сравнительные временные характеристики алгоритма решения задачи Пуассона на одном и нескольких компьютерах.

В качестве параллельной программы взять вами созданную параллельную программу. В качестве последовательной программы, взять последовательную программу из примера предыдущего пункта. Обе программы запустить и засечь время счета для одного и того же пространства размером $20*20*20$. Засечь время решения задачи на 1-м, 2-х и 4-х компьютерах.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

- 1) отвечает на контрольные вопросы;
- 2) поясняет ход выполнения индивидуального задания;
- 3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №7. Решение систем линейных алгебраических уравнений итерационными методами

1. Цель и содержание

Цель лабораторной работы: практическое освоение методов решения систем линейных алгебраических уравнений итерационными методами.

2. Теоретическая часть

Ниже приведена программа для последовательного алгоритма.

Обратите на то как вычисляются скалярные произведения векторов.

```
#include<stdio.h>
#include<time.h>
#include<sys/time.h>
```

```
#define M 100
```

```
#define E 0.00001
```

```
#define T 0.01
```

```
double A[M][M], F[M], Xk[M], Zk[M];
double Rk[M], Sz[M], alf, bet, mf;
double Spr, Spr1, Spz;
```

```
int main()
{ int i, j, v;
  struct timeval tv1, tv2;
  long int dt1;
```

```
/* Генерация данных. Здесь задается матрица с элементами равными 1,
 * по диагонали равными 2. Матрица берется хорошо обусловленной и
```

* симметричной. При правой части равной $M+1$ все корни будут равными 1 */

```
for(mf=0,i = 0; i < M; i++)
```

```
{ for(j = 0; j < M; j++)
```

```
{ if(i == j)
```

```
    A[i][j] = 2.0;
```

```
    else
```

```
    A[i][j] = 1.0;
```

```
}
```

```
F[i] = M + 1;
```

/* Сразу вычисляем сумму квадратов элементов вектора F,

* т.е. подкоренное выражение формулы (7). */

```
mf += F[i] * F[i];
```

```
}
```

/* Задаем начальное приближение корней. В X_k хранятся значения корней

* к-й итерации. */

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

```
    Xk[i] = 0.2;
```

/* Задаем начальное значение r_0 и z_0 . */

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

```
{ for(Sz[i]=0,j = 0; j < M; j++)
```

```
    Sz[i] += A[i][j] * Xk[j];
```

```
    Rk[i] = F[i] - Sz[i];
```

```
    Zk[i] = Rk[i];
```

```
}
```

/* Засекаем время начала вычислений. */

```
gettimeofday(&tv1, NULL);
```

```
do
```

```
{
```

```
/* Вычисляем числитель и знаменатель для коэффициента
```

```
*  $\alpha_k = (r_{k-1}, r_{k-1}) / (Az_{k-1}, z_{k-1})$  */
```

```
    Spz = 0;
```

```
    Spr = 0;
```

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

```
    { for(Sz[i]=0, j = 0; j < M; j++)
```

```
        Sz[i] += A[i][j] * Zk[j];
```

```
        Spz += Sz[i] * Zk[i];
```

```
        Spr += Rk[i] * Rk[i];
```

```
    }
```

```
    alf = Spr/Spz;          /*  $\alpha_k$  */
```

```
/* Вычисляем вектор решения:  $x_k = x_{k-1} + \alpha_k z_{k-1}$ ,
```

```
вектор невязки:  $r_k = r_{k-1} - \alpha_k Az_{k-1}$  и числитель для  $\beta_k$  равный  $(r_k, r_k)$  */
```

```
    Spr1 = 0;
```

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

```
    { Xk[i] += alf*Zk[i];
```

```
      Rk[i] -= alf*Sz[i];
```

```
      Spr1 += Rk[i]*Rk[i];
```

```
    }
```

```
/* Вычисляем  $\beta_k$  */
```

```
    bet = Spr1/Spr;
```

```
/* Вычисляем вектор спуска:  $z_k = r_k + \beta_k z_{k-1}$  */
```

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

```

        Zk[i] = Rk[i] + bet*Zk[i];
    }
    /* Проверяем условие выхода из итерационного цикла */
    while(Spr1/mf > E*E);

    /* Засекаем время конца вычислений. Время решения задачи выводим
на экран */
    gettimeofday(&tv2, (struct timezone*)0);
    dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec;
    printf("Time = %d\n",dt1);

    /* Для контроля выводим 8 первых корней */
    printf("      %f      %f      %f      %f      %f      %f      %f      %f\n",
Xk[0],Xk[1],Xk[2],Xk[3],Xk[4],Xk[5],
Xk[6], Xk[7]);

    return(0);
}

```

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

7. Задание

Задание 1. Проработка примеров из пункта "Примеры параллельных программ" этой же лабораторной.

Внимательно изучить примеры 7.1-7.2. Откомпилировать и запустить на 1-м процессоре.

Задание 2. Сортировка множеств

Разработать параллельный алгоритм, написать и отладить параллельную программу сортировки множеств в MPI. (Метод сортировки

любой по желанию). Использовать программу для последовательного алгоритма в примере 7.1.

Задание 3. Параллельный алгоритм решения СЛАУ методом сопряженных градиентов локально-оптимальная схема в OpenMP

Разработать параллельный алгоритм, написать и отладить параллельную программу решения СЛАУ методом сопряженных градиентов локально-оптимальная схема в OpenMP. Использовать программу для последовательного алгоритма в примере 7.2.

Задание 4. Параллельный алгоритм решения СЛАУ методом сопряженных градиентов локально-оптимальная схема в MPI

Разработать параллельный алгоритм, написать и отладить параллельную программу решения СЛАУ методом сопряженных градиентов локально-оптимальная схема в MPI. Использовать программу для последовательного алгоритма в примере 7.2.

Задание 5. Сравнительные временные характеристики двух алгоритмов решения СЛАУ методом сопряженных градиентов локально-оптимальная схема в языке MPI и языке OpenMP.

В качестве параллельных программ взять параллельные программы заданий 3 и 4. Для обеих программ построить небольшие графики зависимости времени решения задачи от размеров матрицы коэффициентов. Обе программы запустить последовательно для матриц: $A = [300 \times 300]$, $[500 \times 500]$, $[700 \times 700]$, $[1000 \times 1000]$. Откомпилировать и запустить на 2-х компьютерах, засечь время и построить указанные графики.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

- 1) отвечает на контрольные вопросы;
- 2) поясняет ход выполнения индивидуального задания;
- 3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №8. Программирование многопоточных приложений. POSIX Threads.

1. Цель и содержание

Цель лабораторной работы: Освоить разработку многопоточных программ с использованием POSIX Threads API. Познакомиться с задачей динамического распределения работы между процессорами..

2. Теоретическая часть

Формулировка задачи

Есть список неделимых заданий, каждое из которых может быть выполнено независимо от другого. Как формируется задание, см. ниже. Задания могут иметь различный вычислительный вес, т.е. требовать при одних и тех же вычислительных ресурсах различного времени для выполнения. Считается, что этот вес нельзя узнать, пока задание не выполнено. После того, как *все* задания из списка выполнены, появляется новый список заданий. Необходимо организовать параллельную обработку заданий на нескольких компьютерах. Количество заданий существенно превосходит количество процессоров. Программа не должна зависеть от числа компьютеров.

Понятно, что для распараллеливания задачи задания из списка нужно распределять между компьютерами. Так как задания имеют различный вычислительный вес, а список обрабатывается итеративно, и требуется синхронизация перед каждой итерацией, то могут возникать ситуации, когда некоторые процессоры выполнили свою работу, а другие -- еще нет. Если ничего не предпринять, первые будут простаивать в ожидании последних. Так возникает задача динамического распределения работы. Для ее решения на каждом процессоре заведем несколько потоков. Как минимум, потоков должно быть 2:

- поток, который обрабатывает задания и, когда задания закончились, обращается к другим компьютерам за добавкой к работе,

- поток, ожидающий запросов о работе от других компьютеров

Полезно может быть завести третий поток, который возьмет на себя задачу подкачки работ на компьютер, при этом первый поток будет только обрабатывать задания. В таком случае третий поток, до того как кончатся задания (соответствующий момент времени определить самостоятельно), на фоне счета будет отсылать запросы о работе и добавлять к локальному списку пришедшие задания.

Сложность задачи заключается в

- разработке правильной политики взаимодействия между процессами, когда все послыки (send) запросов и данных и ожидания (receive) приема запросов и данных будут согласованы.
- организации корректной работы многих потоков с общими структурами данных. Необходимо обеспечивать взаимное исключение потоков при добавлении заданий в список, удалении задач, выборке заданий для выполнения. Кроме того, надо помнить, что могут быть некоторые неявно используемые общие данные, в частности, сокрытые в реализации подключаемых библиотек и в том числе MPI, см. о требованиях MPI к многопоточным программам. Существует понятие "потокобезопасный" ("thread-safe"). Этот термин может относиться к библиотеке, процедуре и т.п. Он означает, что если потоки одной программы будут одновременно пользоваться функциями этой библиотеки или процедурой, то корректность поведения программы не нарушится. Очевидно, в реализации потокобезопасного кода должно быть предусмотрено возможное параллельное использование этого кода несколькими потоками.

Использование потоков позволяет производить перераспределение заданий на фоне счета. Благодаря этому можно добиться гораздо более эффективного использования ресурсов, чем если бы процесс должен был прерывать обработку заданий на время принятия или отсылки части работы.

Инструменты

Для организации взаимодействия между компьютерами нужно использовать MPI. См. о требованиях MPI к многопоточным программам. Для организации потоков использовать POSIX Threads. См. описание API и пример программы с потоками .

Замечание

Для отправки запросов и данных к конкретным адресатам можно использовать, как обычно, MPI_Send с указанием номера (rank) процесса-адресата. Для приема запроса от *любого* процесса можно использовать функцию MPI_Recv, где в качестве параметра "источник сообщения" указать константу MPI_ANY_SOURCE.

Формирование списка заданий

Задание в данном случае пусть будет иметь совершенно модельный характер. Например, оно может быть таким: выполнить некоторые тратящие время процессора действия repeatNum раз. Различие в вычислительном весе заданий будет заключаться в том, что у каждого задания количество повторений repeatNum свое:

```
TaskList tl;
double globalRes = 0;
int iterCounter = 0;
...
while(true) //итерации обработки списков
{
    iterCounter++; // счетчик глобальных итераций
    for(всех task = номер задания из списка) // выборка заданий из списка
        for(int i = 0; i<tl[task].repeatNum; i++)
            globalRes += sqrt(i);
    ...
}
```

Вес задачи можно назначить случайным образом с использованием функции `rand()`, однако для экспериментов лучше задать некоторые осмысленные правила изменения загрузки на процессорах. Например, на каждой глобальной итерации `iterCounter` веса заданий на процессоре с номером `rank` установить пропорционально (или в некоторой другой зависимости от) `abs(rank-(iterCounter%size))`, где `size` -- количество процессоров. Это создаст "волну" загрузки, смещающуюся с ходом итераций от процессора с меньшим номером к процессору с большим номером (с перескакиванием на начало). Задания генерируются для каждой глобальной итерации заново!

Синхронизация процессоров между итерациями должна заключаться в том, что никто не начинает новую итерацию, пока не обработаны все задания с предыдущей.

Пример многопоточной программы. Параллельное умножение векторов.

В программе порождаются 4 потока, каждый из которых вычисляет часть скалярного произведения, обрабатывая свои четверти от векторов, а затем добавляет полученный результат к глобальному результату.

```
#include <pthread.h>
#include <stdio.h>

//переменная для сборки окончательного результата
int globalRes = 0;
//векторы, которые предстоит умножить
int v1[100], v2[100];
//
int ids[4] = {0,1,2,3};
//четыре объекта типа "описатель потока"
pthread_t thrs[4];
```

```

//мьютекс
pthread_mutex_t mutex;

//функция потока
void* prod(void* me)
{
    //узнали номер потока
    int offset = *((int*)me);
    //вычислили смещение в векторе до "своей" четверти
    offset *= 25;
    //вычисление частичного результата
    int res = 0;
    for(int i = offset; i<offset+25; i++)
        res += v1[i]*v2[i];
    //захват мьютекса
    pthread_mutex_lock(&mutex);
    //добавление к глобальному результату при исключительном владении
    глобальной переменной
    globalRes += res;
    //освобождение мьютекса
    pthread_mutex_unlock(&mutex);
}

int main()
{
    //тут где-то должна быть инициализация массивов

    //атрибуты потока
    pthread_attr_t attrs;

```

```

//инициализация атрибутов потока
if(0!=pthread_attr_init(&attrs))
{
    perror("Cannot initialize attributes");
    abort();
};
//установка атрибута "присоединенный"
if(0!=pthread_attr_setdetachstate(&attrs,
PTHREAD_CREATE_JOINABLE))
{
    perror("Error in setting attributes");
    abort();
}
//порождение четырех потоков
for(int i = 0; i<4; i++)
if(0!=pthread_create(&thrs[i], &attrs, prod, &ids[i]))
{
    perror("Cannot create a thread");
    abort();
}
//освобождение ресурсов, занимаемых описателем атрибутов
pthread_attr_destroy(&attrs);
//ожидание завершения порожденных потоков
for(int i = 0; i<4; i++)
if(0!=pthread_join(thrs[i], NULL))
{
    perror("Cannot join a thread");
    abort();
}

```

```
    return 0;
}
```

Описание POSIX Threads API

POSIX Threads API состоит из большого набора функции, и лишь необходимые для выполнения задания описаны здесь. Для более полного освоения материала см. [литературу](#).

Замечание

В описании параметров слово "IN" означает, что параметр используется для передачи значения в функцию, а "OUT" -- что через параметр функция возвращает результаты. Часто, параметрами функций будут указатели. В таком случае, "IN" говорит о том, что в области памяти, на которую указывает параметр, размещены потребляемые функцией данные, а "OUT" говорит о том, что в этой области будут размещены данные в результате работы функции. Нужно заметить, что в обоих случаях указатель инициализируется до вызова функции, т.е. функция не отвечает за размещение данных в памяти -- она либо читает оттуда, откуда указано, либо пишет туда, куда указано. Если в качестве фактического параметра типа "IN" передается NULL, то это означает, что функция должна поступать в соответствии с политикой по умолчанию. Если NULL указан вместо параметра типа "OUT", то это означает, что соответствующий результат программисту не нужен.

Порождение потока

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine)(void*), void *arg);
```

thread	O	В результате успешного срабатывания функции
	UT	порожденного потока.

attr	I	Атрибуты потока. Задают свойства потока. Может б
------	---	--

	N		
start_routine	I		Указатель на функцию потока. Выполнение потока с
	N		Указатель, который будет передан функции потока
arg	I		потока может менять содержимое памяти с использованием
	N/OUT		просто передает указатель функции потока. Функция по адресу.

Атрибуты

Объект задающий атрибуты потока имеет тип `pthread_attr_t`. Такой объект должен быть инициализирован с помощью функции

```
int pthread_attr_init(pthread_attr_t *attr);
```

В результате объект будет содержать набор свойств потока по умолчанию для данной реализации потоков. А ресурсы, которые могут использоваться в системе для хранения этих атрибутов освобождаются вызовом функции (после того, как объект был использован в вызове `pthread_create` и больше не нужен)

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Поток может быть "присоединяемым" (joinable) или "оторванным" (detached). Для установки этого свойства в атрибутах используется функция

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

где detachstate можно установить в `PTHREAD_CREATE_JOINABLE` или в `PTHREAD_CREATE_DETACHED` соответственно.

Присоединяемые и оторванные потоки

Для каждого присоединяемого потока, один из других потоков явно должен вызвать функцию

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Поток, вызвавший эту функцию, останавливается, пока не окончится выполнение потока thread. Если никто не вызывает pthread_join для присоединяемого потока, то завершившись, он не освобождает свои ресурсы, а это может служить причиной утечки памяти в программе. value_ptr (OUT) - это указатель на указатель, возвращенный функцией завершившегося потока.

Взаимное исключение потоков

Для организации взаимного исключения потоков при доступе к разделяемым данным используются мьютексы (mutex = mutual exclusion), объекты типа pthread_mutex_t. Мьютекс должен быть инициализирован перед использованием с помощью функции

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Параметр attr (IN) задает атрибуты мьютекса. Можно передать NULL для принятия атрибутов по умолчанию. Ресурсы, занимаемые мьютексом, могут быть освобождены функцией

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Для захвата мьютекса поток использует (см. пример) функцию

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

а для освобождения

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Условные переменные

Поясним использование условных переменных на примере. Например, поток номер 1 должен выполнить некоторые действия, когда значение некоторого глобального счетчика counter достигнет критического значения criticalVal, причем значение счетчика меняет поток номер 2. Тогда, чтобы первый поток не крутился в цикле, все время проверяя значение counter, можно использовать условную переменную, с помощью которой поток номер 1 устанавливается в состояние ожидания до тех пор, пока поток номер 2 не просигнализирует о наступлении нужного события:

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

```
pthread_mutex_t mutex; //условная переменная pthread_cond_t cond;  
//счетчик int counter = 0;
```

ПОТОК НОМЕР 1

```
//захват мьютекса pthread_mutex_lock(&mutex); //если значение не  
равно критическому if(counter!=criticalValue) //останавливаемся в ожидании  
этого события. //При этом мьютекс будет разблокирован. Как только событие  
наступит, //и мьютекс будет отпущен вторым потоком, //мьютекс снова  
захватывается и выполнение потока 1 продолжается  
pthread_cond_wait(&cond, &mutex); //обработка критического значения  
processCriticalValue(); //освобождение мьютекса pthread_mutex_unlock();
```

ПОТОК НОМЕР 2

```
do { ... //захват мьютекса pthread_mutex_lock(&mutex); //изменение  
значения счетчика doSomethingWith(&counter); //если событие наступило  
if(counter==criticalValue) //просигнализировать об этом ждущим на условной  
переменной cond pthread_cond_signal(&cond); //отпустить мьютекс  
pthread_mutex_unlock(&mutex); ... }while(...);
```

Условная переменная должна быть инициализирована функцией

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t  
*attr);
```

А ресурсы, занятые ей, могут быть освобождены функцией

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

MPI и потоки

Эта тема раскрывается в главе 8.7. MPI and Threads стандарта MPI-2.0. В ней указываются минимальные требования к реализации MPI, поддерживающей программы с многими потоками, и говорится о том, как должна инициализироваться (в смысле того, для чего существует MPI_Init) такая реализация. Здесь заметим следующее:

- Каждый поток в процессе может посылать и принимать сообщения. Адресуемой единицей является процесс, поэтому любой поток может принять любое сообщение адресованное его процессу.
- Для того, чтобы сообщения не путались между потоками, каждый поток, например, может использовать свой MPI-коммуникатор для обмена данными с другими процессами.
- Стандарт говорит, что из утверждения "реализация MPI поддерживает многопоточность" (она "thread-compliant") должно следовать, что все ее процедуры (функции) потокобезопасны, и любая блокирующая операция MPI блокирует только вызвавший эту операцию поток.
- Функция MPI_Finalize() должна вызываться только тем потоком, который инициализировал MPI, и только после того, как все потоки завершили выполнение MPI-операций. Поэтому все потоки должны быть присоединяемыми (joinable) и присоединены к этому главному потоку перед вызовом MPI_Finalize().
- Вместо MPI_Init в многопоточной программе может быть использована `int MPI_Init_thread(int *argc, char *((*argv)[]), int required, int *provided)`

`argc` `argv` `required` `provided`
 То же самое, что и в MPI_Init

`argc` `argv` `required` `provided`
 То же самое, что и в MPI_Init

Запрашиваемый уровень "поддержки потоков":

- o MPI_THREAD_SINGLE -- программист обещает, что программа будет иметь лишь один поток.
- o MPI_THREAD_FUNNELED -- программист обещает, что только тот

поток, который инициализировал MPI будет в дальнейшем вызывать функции MPI.

- o MPI_THREAD_SERIALIZED --
программист обещает, что потоки не будут вызывать функции MPI одновременно.
- o MPI_THREAD_MULTIPLE --
программист ничего не обещает. Любой поток может вызывать MPI-функции независимо от других потоков. Если реализация MPI поддерживает этот уровень, то это и есть thread-compliant реализация MPI.

provided	UT	(	Фактически предоставленный уровень
----------	----	---	------------------------------------

MPI_Init означает MPI_Init_thread с уровнем MPI_THREAD_SINGLE.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Задание

1. Повторите задание лабораторной работы реализовав его с использованием POSIX Threads.

7. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

- 1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №9. Редукционные операции. Решение СЛАУ
методом простой итерации

1. Цель и содержание

Цель лабораторной работы: получить знания и навыки по работе с редукционными операциями.

2. Теоретическая часть

В ниже приведенном примере переменная описана в теле цикла. Это, так называемая, /приватная/ переменная витка цикла, т.е. ее значение безразлично в начале витка и не используется по окончании витка. Использовать в цикле подобным образом переменные, описанные вне цикла, запрещается из-за двух потенциальных проблем для параллельного выполнения витков цикла: зависимость по данным между витками и не консистентное состояние переменной при выходе из цикла.

Пример. Объявление приватной переменной.

```
DVM(PARALLEL [i][j] ON A[i][j] )  
FOR(i, N)  
FOR(j, N)  
{float x; /* приватная переменная для каждой итерации (витка цикла)*/  
x = B[i][j] + C[i][j];  
A[i][j] = x;  
}
```

Т.е. в C-DVM приватные переменные описываются в теле цикла.

В F-DVM:

Приватные переменные в F-DVM специфицируются клаузой NEW.

Если использование переменной локализовано в пределах одного витка цикла, то ее необходимо указать в спецификации NEW.

NEW-переменные (приватные переменные) не могут быть распределенными массивами. Значение приватной переменной не определено в начале витка цикла и не используется после витка цикла, поэтому в каждом витке цикла может использоваться свой экземпляр приватной переменной.

Пример. Спецификация приватной переменной.

```
CDVM$ PARALLEL (I,J) ON A(I,J), NEW( X )
    DO 10 I = 1, N
    DO 10 J = 1, N
        X = B(I,J) + C(I,J)
        A(I,J) = X
10  CONTINUE
```

В спецификации NEW может стоять список приватных переменных: NEW(X,Y,Z).

2 Редукционные операции

В C-DVM:

Очень часто в программе встречаются циклы, в которых выполняются /редукционные операции/ - в некоторой переменной суммируются элементы массива или вычисляется максимальное (минимальное) значение. Витки таких циклов можно распределять, если указать спецификацию REDUCTION.

Редукционными переменными не могут быть распределенные массивы. Редукционные переменные вычисляются и используются только в операторах определенного вида - редукционных операторах.

Второй параметр функций MAXLOC и MINLOC -- это переменная, описывающая элемент с найденным максимальным (соответственно, минимальным) значением. Обычно это индекс элемента в одномерном массиве или структура, содержащая индексы в многомерном массиве.

Пример. Спецификация редукции.

```
S = 0;
X = A[0];
Y = A[0];
MINi = 0;
DVM(PARALLEL [i] ON A[i];
    REDUCTION SUM(S) MAX(X) MINLOC(Y,MINi))
FOR(i, N)
{
    S = S + A[i];
    X = max(X,A[i]);
    if(A[i] < Y) {
        Y = A[i];
        MINi = i;
    }
}
```

В F-DVM:

Редукционными переменными не могут быть распределенные массивы. Редукционные переменные вычисляются и используются только в операторах определенного вида - редукционных операторах.

Введем следующие обозначения

/rv/ - редукционная переменная

/L/ - одномерный массив целого типа

/n/ - число координат минимума или максимума

/er/ - выражение, не содержащее /rv/

/I/_/k/ - целая переменная

/op/ - одна из следующих операций языка Фортран: +, -, .OR., .AND., .EQV., .NEQV.

/ol/ - одна из следующих операций языка Фортран: .GE., .GT., .LE., .LT.

/f/ - функция max или min

В теле цикла редукционный оператор имеет один из следующих видов:

1) rv = rv op er

rv = er op rv

2) rv = f(rv, er)

rv = f(er, rv)

3) if(rv ol er) rv = er

if(er ol rv) rv = er

4) if(rv ol er) then

rv = er

L(1/_ / /) = e/_/1

```

. .
L( n/_/ / / ) = e/_/n
endif

if( er ol rv ) then
rv = er
L( 1/_/ / / )/_/ / /= e/_/1
. . .
L( n/_/ / / ) = e/_/n
endif/

```

Соответствие вида оператора, операции языка Фортран и имени редукции

FDVM приведено в следующей таблице.

Вид оператора операция Фортран имя редукции FDVM

1	+	SUM(rv)	
1	*	PRODUCT(rv)	
1	.AND.	AND(rv)	
1	.OR.	OR(rv)	
1	.EQV.	EQV(rv)	
1	.NEQV.	NEQV(rv)	
2,3		MAX(rv)	
		MIN(rv)	
4		MINLOC(rv, /L, n)	
		MAXLOC(rv, L, n)	

Операция MAXLOC (MINLOC) предполагает вычисление
максимального

(минимального) значения и определение его координат.

Пример. Спецификация редукции.

```
S = 0
Y = 1.E10
X = -1.
IMIN(1) = 0
CDVM$ PARALLEL ( I ) ON A( I ) ,
CDVM$ REDUCTION(SUM(S), MAX(X), MINLOC(Y,IMIN,1))
DO 10 I = 1, N
  S = S + A(I)
  X = MAX(X,A(I))
  IF(A(I) .LT. Y) THEN
    Y = A(I)
    IMIN(1) = I
  ENDIF
10  CONTINUE
```

2 Решение СЛАУ методом “Простой итерации”

Формулы, по которым осуществляются вычисления данным методом, приведены в лабораторной работе № 4.

В C-DVM:

```
#include<math.h>
```

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
#include<mpi.h>
```

```
#define DVM(dvmdir)
```

```
/* Задается макрос для заголовка цикла for */
```

```
#define DO(v,l,h,s) for(v=l; v<=h; v+=s)
```

```
#define M 20
```

```
#define ITMAX 100
```

```
/* Задается массив процессоров 1 x 4 */
```

```
DVM(PROCESSORS) void *R[1][4];
```

```
/* Специфицируется массив A и одновременно распределяется на  
массив
```

```
процессоров R */
```

```
DVM(DISTRIBUTE [BLOCK][ ] ONTO R) double A[M][M];
```

```
/* Специфицируются массивы F, S, X1, и X и одновременно  
распределяется на массив процессоров R через выравнивание на массив A */
```

```
DVM(ALIGN [i] WITH A[i][ ]) double F[M];
```

```
DVM(ALIGN [i] WITH A[i][ ]) double S[M];
```

```
DVM(ALIGN [i] WITH A[i][]) double X1[M];
```

```
DVM(ALIGN [i] WITH A[][i]) double X[M];
```

```
int main(int an, char **as)
```

```
{ int i, j, it;
```

```
double mf, msf, t, e;
```

```
t = 0.01;
```

```
e = 0.0001;
```

```
/* Инициализация данных */
```

```
DVM(PARALLEL [i][j] ON A[i][j])
```

```
DO(i,0,M-1,1)
```

```
DO(j,0,M-1,1)
```

```
{ if(i == j) A[i][j] = 2.0;
```

```
else A[i][j] = 1.0;
```

```
}
```

```
/* Вычисляется сумма квадратов правых частей */
```

```
DVM(PARALLEL [i] ON F[i]; REDUCTION SUM(mf))
```

```
DO(i,0,M-1,1)
```

```
{ F[i] = M+1;
```

```
mf += F[i] * F[i];
```

```
}
```

```
/* Задается начальное приближение */
```

```
DVM(PARALLEL [i] ON X1[i])
```

```
DO(i,0,M-1,1)
```

```
X1[i] = 0.6;
```

/* Основной итерационный цикл */

DO(it,1,ITMAX,1)

{

DVM(PARALLEL [i] ON X[i])

DO(i,0,M-1,1)

{ $X[i] = X1[i]$;

$S[i] = 0.0$;

}

/* Параллельное умножение матрицы коэффициентов на вектор
приближенных

решений */

DVM(PARALLEL [i][j] ON S[i])

DO(i,0,M-1,1)

DO(j,0,M-1,1)

$S[i] += A[i][j] * X[j]$;

/* Параллельное вычисление новой итерации корней и вычисление
суммы квадратов

элементов вектора невязки */

DVM(PARALLEL [i] ON X1[i]; REDUCTION SUM(msf))

DO(i,0,M-1,1)

{ $X1[i] = X[i] - t * (S[i] - F[i])$;

$msf += (S[i] - F[i]) * (S[i] - F[i])$;

}

/* Проверка условия выхода */

if($msf/mf < e * e$) break;

```

}

// -----

/* Выдача результата на экран */
/* Печать вычисленных корней */
DVM(PARALLEL [i] ON X1[i])
DO(i,0,M-1,1)
printf("X1[%d] = %lf\n", i, X1[i]);

return 0;
}

```

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники

безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Задание

Задание 1. Решение СЛАУ методом сопряженных градиентов.

Задание 2. Решение СЛАУ методом сопряженных градиентов.

7. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

- 1) отвечает на контрольные вопросы;
- 2) поясняет ход выполнения индивидуального задания;
- 3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №10. Решение СЛАУ методом Гаусса

1. Цель и содержание

Цель лабораторной работы: получить знания и навыки по решению СЛАУ методом Гаусса.

2. Теоретическая часть

1 Удаленные ссылки

Определение удаленных ссылок.

/Удаленными данными/ называются данные, используемые на данном процессоре, но размещенные на другом процессоре. Ссылки на такие данные называются /удаленными ссылками/. Рассмотрим обобщенный оператор

$$\text{if } (\dots A[\text{inda}] \dots) B[\text{indb}] = \dots C[\text{indc}] \dots$$

где

/A, B, C/ - распределенные массивы, /inda, indb, indc/ - индексные выражения.

В модели DVM этот оператор будет выполняться на процессоре, на котором размещен элемент /B(indb)/. Ссылки /A(inda)/ и /C(indc)/ не являются удаленными ссылками, если соответствующие им элементы массивов /A/ и /C/ размещены на том же процессоре. Единственной гарантией этого является выравнивание /A(inda)/, /B(indb)/ и /C(indc)/ в одну точку шаблона выравнивания. Если выравнивание невозможно или не было выполнено, то ссылки /A(inda)/ и/или /C(indc)/ необходимо специфицировать как удаленные ссылки. В случае многомерных массивов

данное правило применяется к каждому распределенному измерению.

По степени эффективности обработки удаленные ссылки разделены на два типа: SHADOW и REMOTE.

Если массивы $/B/$ и $/C/$ выровнены и $/inda/ = /indc/ \pm /d/ (/d/ - \text{положительная целочисленная константа})$, то удаленная ссылка $/C(indc)/$ принадлежит типу SHADOW. Удаленная ссылка на многомерный массив принадлежит типу SHADOW, если распределяемые измерения удовлетворяют определению типа SHADOW.

Удаленные ссылки, не принадлежащие типу SHADOW, составляют множество ссылок типа REMOTE.

В данной лабораторной рассмотрим удаленные ссылки типа REMOTE.

Директива REMOTE_ACCESS

Удаленные ссылки типа REMOTE специфицируются директивой REMOTE_ACCESS.

Директива REMOTE_ACCESS может быть отдельной директивой (область действия - следующий оператор) или дополнительной спецификацией в директиве PARALLEL (область действия – тело параллельного цикла).

Если удаленная ссылка задается как имя массива без списка индексов, то все ссылки на этот массив в параллельном цикле (операторе) являются удаленными ссылками типа REMOTE.

В пределах нижестоящего оператора или параллельного цикла компилятор заменяет все вхождения удаленной ссылки ссылкой на буфер. Пересылка удаленных данных производится перед выполнением оператора или цикла.

В C-DVM:

Пример. Спецификация удаленных ссылок типа REMOTE.

```
DVM(DISTRIBUTE [][BLOCK]) float A[100][100], B[100][100];
. . .
DVM(REMOTE_ACCESS A[50][50])    X = A[50][50];
. . .
DVM(REMOTE_ACCESS B[100][100]) A[1][1] = B[100][100];
. . .
DVM(PARALLEL[I][J] ON A[I][J]; REMOTE_ACCESS B[][N])
FOR(I, 100)
FOR(J, 100)
    A[I][J] = B[I][J] + B[I][N];
```

Первые две директивы REMOTE_ACCESS специфицируют удаленные ссылки для отдельных операторов. REMOTE_ACCESS в параллельном цикле специфицирует удаленные данные (столбец матрицы) для всех процессоров, на которые распределен массив /A/.

В F-DVM:

Рассмотрим удаленную ссылку на многомерный распределенный массив

$$A(\text{ind}_1, \text{ind}_2, \dots, \text{ind}_k)$$

Пусть ind_j – индексное выражение по j -ому измерению.

В директиве REMOTE_ACCESS индексное выражение указывается без изменений, если

j-ое измерение является распределенным измерением,

$ind_j = a * i + b$, где a и b не изменяются в процессе выполнения цикла (инварианты).

Во всех остальных случаях в директиве REMOTE_ACCESS вместо ind_j указывается “:” (все измерение).

Пример. Спецификация удаленных ссылок типа REMOTE.

```
DIMENSION A(100,100), B(100,100)
CDVM$ DISTRIBUTE (*,BLOCK) :: A
CDVM$ ALIGN B( I, J ) WITH A( I, J )
...
CDVM$ REMOTE_ACCESS ( A(50,50) )
C      замена ссылки A(50,50) ссылкой на буфер
C      рассылка значения A(50,50) по всем процессорам
1      X = A(50,50)
...
CDVM$ REMOTE_ACCESS ( B(100,100) )
C      пересылка значения B(100,100) в буфер процессора
own(A(1,1))
2      A(1,1) = B(100,100)
...
CDVM$ PARALLEL (I,J) ON A(I,J) , REMOTE_ACCESS ( B(:,N) )
C      рассылка значений B(:,N) по процессорам own(A(:,J))
3      DO 10 I = 1, 100
        DO 10 J = 1, 100
10     A(I,J) = B(I,J) + B(I,N)
```

Первые две директивы `REMOTE_ACCESS` специфицируют первые две директивы удаленные ссылки для отдельных операторов. `REMOTE_ACCESS` в параллельном цикле специфицирует удаленные данные, которые распределены на столбцы матрицы A для всех процессоров.

2 Решение СЛАУ методом Гаусса

Последовательный алгоритм, по которым осуществляются вычисления данным методом, приведены в лабораторной работе № 5.

В C-DVM:

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#define DVM(dvmdir)

/* Задается макрос для заголовка циклов for */
#define DO(v,l,h,s) for(v=l; v<=h; v+=s)

#define N 10 /*100*/

int main(int an, char **as)
{
    long i, j, k;
```

/* Задается массив процессоров 1 x 4 */

DVM(PROCESSORS) void *R[1][4];

/* Задается указатель на матрицу коэффициентов A и одновременно определяется

распределение массива A на массив процессоров R */

DVM(DISTRIBUTE [BLOCK] [] ONTO R) float (*A)[N+1];

/* Задается указатель на вектор правой части X и одновременно определяется

распределение X на массив процессоров R через выравнивание на массив A */

DVM(ALIGN [i] WITH A[i][]) float (*X);

/* Выделяется память для матрицы A и вектора X */

A = malloc(N*(N+1)*sizeof(float));

X = malloc(N*sizeof(float));

/* Инициализация матрицы A и вектора правой части A[][N+1] */

DVM(PARALLEL [i][j] ON A[i][j])

DO(i,0,N-1,1)

DO(j,0,N,1)

if (i==j || j==N) A[i][j] = 1.f;

else A[i][j]=0.f;

/* Прямой ход */

for (i=0; i<*/=*/N-1; i++)

{

DVM(PARALLEL [k][j] ON A[k][j]; REMOTE_ACCESS A[i][])

DO (k,i+1,N-1,1)

```

DO (j,i+1,N,1)
    A[k][j] = A[k][j]-A[k][i]*A[i][j]/A[i][i];
}
/* REMOTE_ACCESS A[i][] – ссылка на текущую обрабатываемую
строку */
/* Обратный ход */
X[N-1] = A[N-1][N]/A[N-1][N-1];
for (j=N-2; j>=0; j-=1)
{
DVM(PARALLEL [k] ON A[k][]; REMOTE_ACCESS X[j+1])
    DO (k,0,j,1)
        A[k][N] = A[k][N]-A[k][j+1]*X[j+1];
X[j]=A[j][N]/A[j][j];
/* REMOTE_ACCESS X[j+1] – ссылка на уже вычисленный корень */

/* Выдача на экран монитора результатов вычислений */
DVM(REMOTE_ACCESS X[j])
printf("j=%4li X[j]=%3.3E\n",j,X[j]);
}
return 0;
}

```

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Задание

1. Тщательно изучить приведенный материал.
2. Скомпилировать и запустить программу примера на 4-х процессорах.

7. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

- 1) отвечает на контрольные вопросы;
- 2) поясняет ход выполнения индивидуального задания;
- 3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.