Федеральное агентство связи

Федеральное государственное образовательное бюджетное учреждение
высшего профессионального образования
«Сибирский государственный университет телекоммуникации и информатики»
(ФГОБУ ВПО «СибГУТИ»)

А.Д. Рычков

ЛАБОРАТОРНЫЕ РАБОТЫ ПО ПАРАЛЛЕЛЬНЫМ ВЫЧИСЛИТЕЛЬНЫМ ТЕХНОЛОГИЯМ

Новосибирск
2013

ОГЛАВЛЕНИЕ

Общие рекомендации по выполнению лабораторных работ

1. Для выполнения лабораторных работ студент должен уметь компилировать и запускать на счет параллельные программы на узлах кластерной вычислительной системе Jet. Руководство по работе на кластере можно найти на сайте http://cpct.sibsutis.ru/index.php/Main/Jet.
2. Основное содержание лабораторных работ связано с разработкой параллельных программ для решения задач линейной алгебры, численного анализа, дифференциальных уравнений и с имитационным моделированием. Поэтому студент должен обладать компетенциями в данных предметных областях в рамках стандартного вузовского курса.
3. В данном методическом пособии приведены лишь необходимые для выполнения работ сведения по инструментальным средствам создания параллельных программ – MPI и OpenMP. Более подробную информацию можно найти в источниках, приведенных в списке литературы.

ОСНОВНЫЕ ПРОЦЕДУРЫ MPI

int **MPI_Init(** int* argc, char*** argv)
MPI_Init - инициализация параллельной части приложения.
Возвращает: в случае успешного выполнения - MPI_SUCCESS, иначе - код ошибки.

int **MPI_Finalize**(void)
MPI_Finalize - завершение параллельной части приложения.

int **MPI_Comm_size(MPI_Comm** comm, int* size)
Определение общего числа параллельных процессов в группе comm.
-- comm - идентификатор группы
-- size - размер группы

int **MPI_Comm_rank**(**MPI_Comm** comm, int* rank)
Определение номера процесса в группе comm.
Значение, возвращаемое по адресу &rank, лежит в диапазоне от 0 до size-1.
-- comm - идентификатор группы
-- rank - номер вызывающего процесса в группе comm

double **MPI_Wtime**(void)
Функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Гарантируется, что этот мо-

мент не будет изменен за время существования процесса.

Предопределенные типы

MPI_Status - структура; атрибуты сообщений; содержит три обязательных поля:

-- **MPI_Source** (номер процесса отправителя)
-- **MPI_Tag** (идентификатор сообщения)
-- **MPI_Error** (код ошибки)
-- **MPI_Comm** - системный тип; идентификатор группы (коммуникатора)
-- **MPI_COMM_WORLD** - зарезервированный идентификатор группы, состоящей из всех процессов приложения

Коммуникационные обмены «точка-точка»

Прием/передача сообщений с блокировкой

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

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

int **MPI_Recv(**void* buf, int count, MPI_Datatype datatype, int source, int msgtag,MPI_Comm comm, MPI_Status *status) -- buf - адрес начала буфера приема сообщения

-- count - максимальное число элементов в принимаемом сообщении
-- datatype - тип элементов принимаемого сообщения
-- source - номер процесса-отправителя
-- msgtag - идентификатор принимаемого сообщения
-- comm - идентификатор группы
-- status - параметры принятого сообщения

Для определения числа фактически полученных элементов сообщения необходимо использовать специальную *функцию MPI_Get_count*:

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

status - атрибуты принятого сообщения;
datatype - тип элементов принятого сообщения;
count - число полученных элементов.

Подпрограмма MPI_Get_count может быть вызвана после чтения сообщения (функциями MPI_Recv, MPI_Irecv).

Объединенные функции передачи-приема данных

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

-- sendbuf - адрес начала посылаемого буфера
-- sendcount - количество посылаемых элементов
-- senddatatype - тип передаваемых элементов
-- dest - ранг (номер) процесса, которому осуществляется передача
-- sendtag - тег посылаемого сообщения
-- recvbuf - адрес буфера для приема данных
-- recvcount - максимальное количество принимаемых элементов
-- recvtype - тип принимаемых элементов
-- source - ранг (номер) передающего процесса
-- recvtag - тег принимаемых данных
-- comm – имя переключателя каналов
-- status - статус полученного сообщения

Эта функция выполняет блокированную посылку и получение даных. Посылающий и приемный буферы не должны пересекаться.

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)

-- buf - адрес посылаемого буфера и буфера приема данных
-- count - количество элементов в посылаемом буфере и в буфере приема
-- datatype - тип передаваемых и получаемых элементов
-- dest - ранг (номер) процесса, которому осуществляется передача
-- sendtag - тег посылаемого сообщения
-- recvtype - тип принимаемых элементов
-- source - ранг (номер) передающего процесса
-- recvtag - тег принимаемых данных
-- comm – имя переключателя каналов
-- status - статус полученного сообщения

Операция выполняется с блокированием посылки и получения. Используется тот же самый буфер как для посылающего, так и для получающего оператора. Посланное сообщение заменяется затем полученным сообщением.

Прием/передача сообщений без блокировки

int **MPI_Isend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_COMM_WORLD, MPI_Request *request)

buf	- адрес начала расположения передаваемых данных;
count	- число посылаемых элементов;
datatype	- тип посылаемых элементов;
dest	- номер процесса-получателя;
tag	- идентификатор сообщения;
request	- "запрос обмена".

int **MPI_Irecv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_COMM_WORLD, MPI_Request *request)

buf	- адрес для принимаемых данных;
count	- максимальное число принимаемых элементов;
datatype	- тип элементов принимаемого сообщения;
source	- номер процесса-отправителя;
tag	- идентификатор сообщения;
request	- "запрос обмена".

Функция ожидания завершения неблокирующей операции MPI_Wait.

int MPI_Wait(MPI_Request *request, MPI_Status *status)

request	- запрос связи;
status	- атрибуты сообщения.

Функция проверки завершения неблокирующей операции MPI_Test.

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

request	-	запрос связи;
flag	-	признак завершенности проверяемой операции;
status	-	атрибуты сообщения, если операция завершилась.

Функция снятия запроса без ожидания завершения неблокирующей операции MPI_Request_free.

int MPI_Request_free(MPI_Request *request)

request - запрос связи.

Коллективные операции

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

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

Рассылка сообщения от процесса source всем процессам, включая рассылающий процесс.

int **MPI_Barrier**(**MPI_Comm** comm)

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

int **MPI_Reduce**(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

-- sendbuf - адрес передаваемого буфера;
-- recvbuf - адрес буфера приема;
-- count - количество передаваемых элементов;
-- datatype - тип посылаемых элементов;
-- op - операция редукции;
-- root - ранг корневого процесса;
-- comm - коммуникатор.

Имена некоторых операций:
MPI_MAX – определение максимального значения;
MPI_MIN – определение минимального значения;
MPI_SUM – суммирование;
MPI_PROD – произведение;

int **MPI_Gather**(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_COMM_WORLD)

sendbuf - адрес начала размещения посылаемых данных;
sendcount - число посылаемых элементов;

sendtype - тип посылаемых элементов;
recvbuf - адрес начала буфера приема (используется только в процессе-получателе root);
recvcount - число элементов, получаемых от каждого процесса (используется только в процессе-получателе root);
recvtype - тип получаемых элементов;
root - номер процесса-получателя;

Функция MP_Gather производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root. Длина блоков предполагается одинаковой. Объединение происходит в порядке увеличения номеров процессов-отправителей. То есть данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf процесса root. Длина массива, в который собираются данные, должна быть достаточной для их размещения.

int **MPI_Allgather**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_COMM_WORLD)

sendbuf - адрес начала буфера посылки;
sendcount - число посылаемых элементов;
sendtype - тип посылаемых элементов;
recvbuf - адрес начала буфера приема;
recvcount - число элементов, получаемых от каждого процесса;
recvtype - тип получаемых элементов;

Функция MPI_Allgather выполняется так же, как MPI_Gather, но получателями являются все процессы группы. Данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf каждого процесса. После завершения операции содержимое буферов приема recvbuf у всех процессов одинаково.

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

sendbuf - адрес начала размещения блоков распределяемых данных (используется только в процессе-отправителе root);
sendcount - число элементов, посылаемых каждому процессу;
sendtype - тип посылаемых элементов;
recvbuf - адрес начала буфера приема;

recvcount - число получаемых элементов;
recvtype - тип получаемых элементов;
root - номер процесса-отправителя;

Функция MPI_Scatter разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе). Процесс root использует оба буфера (посылки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы являются только получателями, поэтому для них параметры, специфицирующие буфер посылки, не существенны. Тип посылаемых элементов sendtype должен совпадать с типом recvtype получаемых элементов, а число посылаемых элементов sendcount должно равняться числу принимаемых recvcount.

Функции декартовых топологий

int **MPI_Cart_create**(MPI_com com_old, int ndims, int *dims, int *periods,
int reoerar, MPI_Comm *comm_cart)

-- comm_old входной (старый) коммуникатор
-- ndims количество измерений в декартовой топологии
-- dims целочисленный массив размером ndism, определяющий количество процессоров в каждом измерении
-- periods массив размером ndism логических значений, определяющ ий периодичность (true) или нет (false) в каждом измерении
-- reorder ранги могут быть пронумерованы (true) или нет (false)
-- comm_cart коммуникатор новой (созданной) декартовой топологии

Если reorder = false, тогда ранг каждого процессора в новой группе идентичен ее рангу в старой группе, иначе функция может переупорядочивать процессы.

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

-- nnodes количество узлов в решетке
-- ndims мерность декартовой топологии
-- dims целочисленный массив размером ndism, определяющий количество узлов в каждой размерности

Осуществляет разбиение всех процессоров группы в N-мерную топологию

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

-- comm коммуникатор с декартовой топологией
-- rank ранг процессора в топологии comm
-- maxdims максимальный размер массивов dims, periods и coords в вызывающей программе
-- coords целочисленный массив, определяющий координаты нужного процесса в декартовой топологии

Функция переводит ранг процесса в координаты процесса в топологии.

int **MPI_Cart_shift**(MPI_com comm, int direction, int disp, int *rank_source,
int *rank_dest)
-- comm коммуникатор с декартовой топологией
-- direction номер измерения (в топологии), где делается смещение
-- disp направление смещения (> 0 смещение в сторону увеличения номеров координаты direction, < 0 смещение в сторону уменьшения номеров координаты direction)
-- rank_source ранг процесса источника
-- rank_dest ранг процесса назначения

Координаты маркируются от 0 до ndims-1, где ndims – число размерностей.

Примеры использования этих функций для организации параллельных вычислений приведены в Приложении 2, 3.

Основные директивы OpenMP

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).

Поведение переменных при входе и выходе из параллельной области или параллельного цикла управляется дополнительными параметрами: ***reduction, firstprivate, lastprivate, copyin.***

Директивы

Директивы OpenMP на языке C начинаются с комбинации символов "#pragma и разделяютсяь на 3 категории: *определение параллельной секции, разделение работы, синхронизация.*

1. Директива parallel (основная директива OpenMP).

Формат директивы parallel

#pragma omp parallel [clause ...] newline
structured_block

Наиболее употребляемые параметры (clause*): private (list), firstprivate (list), shared (list), reduction (operator: list)*

Существует 3 директивы для распределения вычислений в параллельной области

for – распараллеливание циклов

sections – распараллеливание раздельных фрагментов кода (функциональное распараллеливание)

single – директива для указания последовательного выполнения кода

2. Директива **for** (распараллеливание циклов)

Формат директивы **for**

#pragma omp for [clause ...] newline

<оператор цикла>

Наиболее употребляемые параметры (clause): *private(list), firstprivate(list), lastprivate(list), reduction(operator: list), ordered, schedule(par1, par2), nowait.*

В клаузе *schedule par1* и *par2* 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.

3. Директива **sections** – распределение вычислений для раздельных фрагментов кода. Фрагменты выделяются при помощи директивы **section**. Каждый фрагмент выполняется однократно, разные фрагменты выполняются разными потоками.

Формат директивы **sections**

```
#pragma omp sections [clause ...] newline
{
#pragma omp section newline
structured_block
#pragma omp section newline
structured_block
}
```

Возможные параметры (clause): *private(list), firstprivate(list), lastprivate(list), reduction(operator: list), nowait*

4. Директива **single**

Формат директивы single

```
#pragma omp single [clause ...] newline
structured_block
```

Определяет блок, который будет исполнен только одной нитью (первой, которая дойдет до этого блока). Возможные параметры (clause): *private(list), firstprivate(list), nowait*

Директивы синхронизации

1. Директива **master** определяет фрагмент кода, который должен быть выполнен только основным потоком; все остальные потоки пропускают данный фрагмент кода.

```
#pragma omp master newline
structured_block
```

2. Директива **critical** определяет блок кода, который должен выполняться только одним потоком в каждый текущий момент времени, то есть блок кода, который не должен выполняться одновременно двумя или более нитями.

```
#pragma omp critical [name] newline
structured_block
```

3. Директива **barrier** – определяет точку барьерной синхронизации, в которой каждая нить дожидается всех остальных.

```
#pragma omp barrier newline
```

1. Лабораторная работа №1.
Организация приема и передачи данных в MPI и OpenMP

Цель работы: освоение базовых функций систем MPI и OpnMP.

1.1. Коммуникационные операции “точка-точка»

1. Изучите тестовую программу, приведенную в Приложении 1, в которой организована передача данных между двумя процессами с помощью процедур MPI_Send и MPI_Recv. Убедитесь, что она работает только при числе процессоров, равных двум. Вставьте в программу с помощью функции MPI_Wtime замеры времени работы процедур MPI_Send и MPI_Recv. Чтобы получить значимые времена их работы увеличьте длину передаваемого и принимаемого массивов.

2. На основе тестовой программы напишите свою программу, моделирующую соотношение между временем вычислительной нагрузкой на процессор и временами обмена в коммуникационных операциях. Фрагмент такой программы приведен ниже.

```
……….
n=10; m=1000; for(k=1; k<=n; k++){
if(rank==0){
for(i=0; i<1000; i++)bb[i]=i+0;
 t1=MPI_Wtime();
MPI_Send(bb, 1000, MPI_DOUBLE, 1,1, MPI_COMM_WORLD);
t1=MPI_Wtime()-t1;
% имитация вычислительной нагрузки
t2=MPI_Wtime();
for(i=1; i<=m; i++)a=sin(i)*cos(i)*pow((i+0.),0.75)*exp((i+0.)/m);
t2=MPI_Wtime()-t2;
printf(“время Send= %13.4e время нагрузки= %13.4e \n“, t1,t2);}
MPI_Barrier(MPI_COMM_WORLD);
if(rank==1){t3=MPI_Wtime();
MPI_Recv(bb, 1000, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &status );
t3=MPI_Wtime()-t3;printf(“время Recv= %13.4e\n“,t3);  }
MPI_Barrier(MPI_COMM_WORLD);
}
……
```

Осредните 10 полученных величин t1, t2, t3 для получения их более достоверных значений. Изменяя значение величины m, можно регулировать вычислительную нагрузку t2. Постройте графики величин t1/t2 и t3/t2 от параметра m.

3. Замените в программе коммуникационные операции с блокировкой на неблокирующие операции (используйте также функции MPI_Wait, MPI_Test), проведите расчеты и постройте аналогичные графики. Определите, начиная с какого значения m неблокирующие операции требуют меньшего времени на обмены.

1.2. Коллективные обмены

1. Модифицируйте предыдущую программу, включив в нее один одномерный (буфер) и один двумерный (рабочий) массивы. Длина буфера должна быть не меньше длины рабочего массива, умноженного на максимальное число запускаемых процессоров (ограничтесь восьмью).

Проделайте следующие действия:

1.1. В нулевом процессе заполните рабочий массив какими-либо числами и разошлите значение этого массива всем процессам с помощью процедуры MPI_Bcast. Проверьте в каждом процесса правильность полученных данных.

1.2. В каждом процессе заполните рабочий массив числами, различными в каждом процессоре, и с помощью процедуры MPI_Gather соберите их в буферном массиве нулевого процесса. Размер этого массива должен быть достаточным для приема данных. Проверьте правильность информации, записанной в буфер.

1.3. Поделайте обратную рассылку процедурой MPI_Scatter содержимого буфера нулевого процессора всем остальным процессорам и проверьте правильность полученной ими информации.

1.4. В каждом процессе заполните рабочий массив своими числами (например, номером процесса) и с помощью процедуры MPI_Allgather разошлите информацию из него всем процессам. Проверьте правильность информации, полученной всеми процессами. Длина приемного буфера должен быть достаточной для размещения в нем всех массивов.

1.5. Заполните рабочий массив информацией, различной для каждого процессора, и проанализируйте работу функции MPI_Reduce с типами операций MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, собирая результаты ее работы в нулевом процессоре.

1.6. Напишите программу на MPI вычисления скалярного произведения двух векторов, с использованием функции MPI_Reduce.

1.3. Виртуальные топологии

1. Изучите программу обмена данными между процессами с топологией «кольцо» из Приложения 1. Убедитесь, что программа работает правильно на разном числе процессоров. Сделайте в программе следующие изменения:

- смените ранг стартового процесса;

- измените направления передачи по кольцу на противоположное;
- увеличьте шаг сдвига по кольцу до двух и запустите программу на четном и нечетном числе процессоров.

Напишите краткий отчет по полученным результатам.

1.2. На основе исходной программы напишите программу скалярного умножения вектор-строки $\vec{a}$ длиной n, находящегося в нулевом процессе, на матрицу $B - n \times n$, столбцы которой по одному расположены в остальных ветвях, используя пересылку этих вектор-столбцов в нулевой процесс с помощью топологии «кольцо»: $c_i = \sum_{k=1}^{n} a_k b_{ki}$. Результат (вектор-строка $\vec{c}$) расположить в нулевом процессе. Число n положить равными числу процессоров size в задаче, значения элементов вектора и матрицы задайте произвольными.

2. Изучите работу программы обмена между процессами с топологией «линейка» из Приложения 2 на различном числе процессоров.

2.1. Реализуйте умножение вектор-строки на матрицу из предыдущего задания 1.2 с помощью топологии «линейка».

1.4. Программирование в MPI

1. Напишите параллельную программу на MPI для вычисления тройного интеграла

$$I = \int_0^1 \int_0^1 \int_0^1 x^4 y^4 z^4 dxdydz = 1/125$$

по формуле прямоугольников $I \approx \sum_{k=0}^{n_k-1} \sum_{j=0}^{n_j-1} \sum_{i=0}^{n_i-1} f(x_i, y_j, z_k) \Delta x \Delta y \Delta z$ на прямоугольной сетке, где $x_i = i \cdot \Delta x, y_j = j \cdot \Delta y, z_k = k \cdot \Delta z$ при $n_i = 300, n_j = 200, n_k = 400$. Для распределения вычисления по процессорам используйте свойство аддитивности интеграла.

2. Напишите параллельную программу для вычисления максимального и минимального значений среди элементов матрицы A – 400 x 300, заполненной случайными числами. Определите зависимость коэффициента ускорения от числа процессоров.

1.5. Программирование на OpenMP

1. Напишите программу вычисления скалярного произведения двух векторов, используя директивы распараллеливания циклов и редукции.

2. Напишите параллельные программы вычисления тройного интеграла и вычисления максимального значения среди элементов матрицы, аналогич-

ные задачам, приведенным в предыдущем задании. Определите зависимости коэффициента ускорения от числа процессоров и сравните полученные результаты с результатами, полученными на MPI.

2. Лабораторная работа № 2. Умножение двух матриц

Работа содержит два этапа. **На первом этапе** целью работы является написание параллельных программ на MPI и OpenMP для умножения двух прямоугольных матриц $A - M \times N$ и $B - N \times L$. Предполагается, что как исходные матрицы, так и результат умножения (матрица $C - M \times L$) целиком расположены в памяти каждого процессора. В нулевом процессоре матрицы A и B заполняются произвольными числами. Здесь же, для последующего контроля правильности работы программы, вычисляется их произведение $D = A \cdot B$. Далее, в MPI программе с помощью процедуры MPI_Bcast значения матриц A и B рассылаются во все остальные процессоры. Матрица A разрезается на M/n горизонтальных полос (n – число процессоров, M/n –целое число), и в каждом процессоре производится умножение своей полосы на столбцы матрицы B таким образом, чтобы получаемая матрица C в каждом процессоре тоже представляла собой набор из M/n горизонтальных полос. После их вычисления все полосы матрицы C собираются в нулевом процессоре с помощью команды MPI_Gather. Здесь же для контроля правильности работы программы собранная матрица C поэлементно сравнивается с ранее вычисленной матрицей D. Время счета можно определять с помощью процедуры MPI_Wtime как разность времен между первым обращением к ней в начале работы программы (после рассылки) и последним в конце работы программы. В OpenMP программе нет необходимости в разрезании матриц на полосы и нужно с помощью соответствующей директивы распараллелить самый внешний цикл алгоритма умножения матриц.

2.1. Первый этап работы

1. Напишите параллельную программу для решения описанной выше задачи на MPI. Число строк и столбцов во всех матрицах задайте одинаковым и равным 1600 (для удобства счета на 2, 4, 8 и 16 процессорах). Способ заполнения матриц числами – на ваше усмотрение. Вставьте в программу замеры времени с помощью процедур MPI_Wtime с целью определения общего времени работы программы (“время счета” t_c) и времени, которое потрачено на обмен в процедуре MPI_Gather (“накладные расходы” t_mpi).
2. Проведите расчеты на двух, четырех, восьми и шестнадцати процессорах и постройте зависимость коэффициента ускорения от числа процессоров. Постройте в виде таблицы зависимости t_c и t_mpi от числа процессоров и опре-

делите “оптимальное” число процессоров, поддерживающее баланс между временными затратами на счет и “накладными расходами”.

3. Для восьми процессоров проведите расчеты в случае заказа в паспорте задачи одного узла и восьми ядер и для случая заказа восьми узлов по одному ядру в каждом, оценив тем самым работу коммуникационных каналов связи между процессорами.

4. Напишите программу на OpenMP для решения указанной выше задачи и определите коэффициент ускорения вычислений в зависимости от числа используемых параллельных потоков (от числа ядер на одном вычислительном узле).

5. Сравните эффективность работы программ на VPI

Схема разбиения матрицы A на горизонтальные полосы представлена на рис 1.

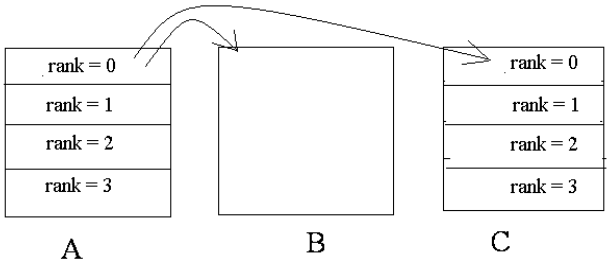

Рис.1. Схема умножения матриц первого этапа

1). Описываются массивы A[M][N], B[N][L], C[M][L], C_t[M][L]. 2). В нулевом процессоре массивы A и B заполняются произвольными числами. Здесь же для контроля в нулевом процессоре из матриц A и B вычисляется контрольная матрица $C\_t = A \cdot B$, компоненты которой записываются в массив C_t[M][L]. Содержимое матриц A и B рассылаются во все остальные процессоры. 3). В каждом процессоре своя часть строк матрицы A (полоска) умножается на все столбцы матрицы B, результат помещается в соответствующую часть матрицы C (также в полоску). После завершения вычислений всех полос матрицы C в нулевом процессоре из этих полос собирается итоговая матрица C, которая для контроля правильности сборки поэлементно сравнивается с матрицей C_t.

2.2. Второй этап работы

На втором этапе нужно написать MPI программу умножения двух квадратных матриц $A - N \times N$ и $B - N \times N$, в которой матрица A разбивается на n горизонтальных полос, а матрица B – на n вертикальных по числу процессоров n так, что в каждом процессоре содержится K=N/n (K –целое) строк матрицы A и K столбцов матрицы B. Результирующая матрица C в этом случае будет разрезана на аналогичные горизонтальные полосы и также будет распределена по процессорам. Для организации параллельных вычислений необходимо по очереди из каждого процессора с помощью топологии «кольцо» рассылать вертикальную полосу матрицы B всем остальным процессорам. После вычисления всех соответствующих полос матрицы C итоговая матрица C – N x N собирается в нулевом процессоре и поэлементно сравнивается с компонентами контрольной матрицы,

а также определяются все временные затраты аналогично предыдущему этапу работы.

В каждом процессоре описываются массивы A[N][N], B[N][N], C_t[N][N]. Массивы A и B в нулевом процессоре заполняются произвольными числами. Здесь же вычисляется контрольная матрица $C\_t = A \cdot B$. Далее, с помощью процедуры MPI_Bcast следует разослать из нулевого процессора всем остальным свои горизонтальные полосы матрицы A и вертикальные полосы матрицы B. После завершения всех обменов по «кольцу» нужно собрать полученную матрицу в нулевом процессоре и сравнить ее поэлементно с контрольной матрицей C_t.

Схема разбиения матрицы A на горизонтальные полосы, матрицы B на вертикальные полосы, формирование матрицы C и структура обменов по «кольцу» для четырех процессоров показана на рис 2.

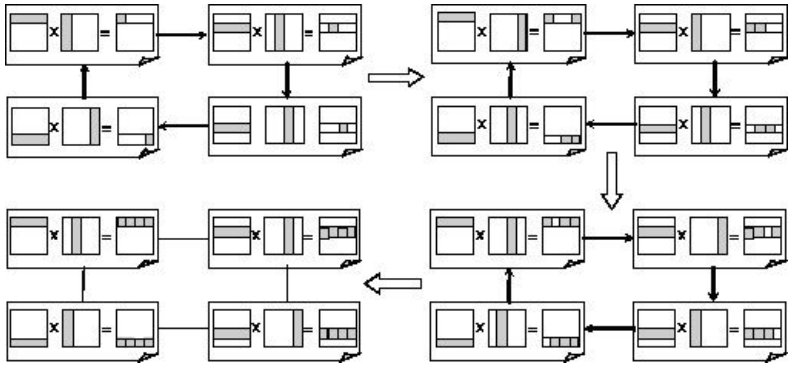

Рис.2. Схема обменов в топологии «кольцо»

Проделайте следующую работу. 1. Напишите параллельную программу для решения описанной выше задачи на MPI с использованием топологии «кольцо». Задайте N = 1600 (для удобства счета на 2, 4, 8 и 16 процессорах) и заполните матрицы A и B произвольными числами. Вставьте в программу замеры времени с помощью процедур MPI_Wtime с целью определения общего времени работы программы, т.е. нужно засечь время в момент окончания рассылки и время после завершения всех обменов по «кольцу».

2. Проведите расчеты на двух, четырех, восьми и шестнадцати процессорах и постройте зависимость коэффициента ускорения от числа процессоров.

3. Сравните эффективность работы программ первого и второго этапов.

3. Лабораторная работа № 3. Решение системы линейных алгебраических уравнений методом Гаусса

Суть классического метода Гаусса заключается в следующем. Пусть в системе уравнений

$$a_{11}^{(0)} x_1 + a_{12}^{(0)} x_2 + \ldots, + a_{1n}^{(0)} x_n = a_{1,n+1}^{(0)}$$

$$
\begin{array}{l}
a_{21}^{(0)}x_1 + a_{22}^{(0)}x_2 + \ldots, + a_{2n}^{(0)}x_n = a_{2,n+1}^{(0)} \\
\ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\
a_{n1}^{(0)}x_1 + a_{n2}^{(0)}x_2 + \ldots, + a_{nn}^{(0)}x_n = a_{n,n+1}^{(0)}
\end{array}
$$

первый элемент $a_{11}^{(0)} \neq 0$ и назовем его ведущим элементом первой строки. Поделим все элементы этой строки на $a_{11}^{(0)}$ и исключим x_1 из всех последующих строк, начиная со второй, путем вычитания первой (преобразованной), умноженной на коэффициент при x_1 в соответствующей строке. Получим

$$
\begin{array}{l}
x_1 + a_{12}^{(1)}x_2 + \ldots, + a_{1n}^{(1)}x_n = a_{1,n+1}^{(1)} \\
0 + a_{22}^{(1)}x_2 + \ldots, + a_{2n}^{(1)}x_n = a_{2,n+1}^{(1)} \\
\ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\
0 + a_{n2}^{(1)}x_2 + \ldots, + a_{nn}^{(1)}x_n = a_{n,n+1}^{(1)}.
\end{array}
$$

Если $a_{22}^{(1)} \neq 0$, то, продолжая аналогичное исключение, приходим к системе уравнений с верхней треугольной матрицей

$$
\begin{array}{l}
x_1 + a_{12}^{(1)}x_2 + \ldots, \quad + a_{1n}^{(1)}x_n = a_{1,n+1}^{(1)} \\
0 + \quad x_2 + \ldots, \quad + a_{2n}^{(2)}x_n = a_{2,n+1}^{(2)} \\
0 + \quad 0 + x_3 + \ldots, + a_{3n}^{(3)}x_n = a_{3,n+1}^{(3)} \\
\ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\
0 + \quad 0 + \ldots \quad \ldots, \quad + x_n = a_{n,n+1}^{(n)}.
\end{array}
$$

Далее из нее в обратном порядке аналогичным образом исключим с помощью последней строки все x_n во всех строках, расположенных выше последней. Продолжая этот процесс, получим на месте исходной матрицы единичную матрицу, а в столбце правых частей будут находиться значения искомых компонент решения x_i:

$$
\begin{array}{l}
x_n = a_{n,n+1}^{(n)} \\
x_{n-1} = a_{n-1,n}^{(n-1)} - a_{n-2,n}^{(n-1)}x_n \\
\ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\
x_1 = a_{1,n+1}^{(1)} - a_{1,2}^{(1)}x_2 - \quad \ldots \quad - a_{1,n}^{(1)}x_n.
\end{array}
$$

Процесс приведения исходной системы к системе с треугольной матрицей называется прямым ходом метода Гаусса, а нахождение неизвестных - обратным.

Рассмотрим один из вариантов организация параллельных вычислений при

решении системы уравнений $A\vec{x} = \vec{y}$ методом Гаусса, где A – квадратная матрица размерности M, $\vec{x}$ – вектор-столбец неизвестных, $\vec{y}$ – вектор-столбец правых частей системы. К матрице A добавляется вектор правых частей и эта расширенная матрица $\overline{A}$

$$\overline{A} = \begin{pmatrix} a_{11}\ a_{12} \ldots a_{1n}\ a_{1,n1} \\ a_{21}\ a_{22} \ldots a_{21}\ a_{2,n+1} \\ \ldots \quad \ldots \quad \ldots \quad \ldots \\ a_{n1}\ a_{n2} \ldots a_{nn}\ a_{n.n+1} \end{pmatrix}$$

разрезается на N полос равной ширины (для удобства выберем значение N кратным M), каждая из которых загружается в свой процессор. Далее в нулевом (активном) процессоре первая (ведущая) строка матрицы $\overline{A}$ делится на первый (ведущий) элемент и она передается всем остальным процессорам. Во всех процессорах с помощью этой строки элементарными преобразованиями все элементы первого столбца матрицы A (за исключением первой строки активного процессора) преобразовываются в нулевые. Затем в активном процессоре ведущей становится следующая строка, ведущим – ее первый ненулевой элемент и процесс продолжается до исчерпания строк в активном процессоре. После этого активным становится следующий (первый) процессор и все повторяется снова до тех пор, пока не исчерпаются все ведущие строки во всех процессорах. В результате в расширенной матрица $\overline{A}$ матрица A приведется к верхней треугольной матрице, распределенной по всем процессорам. На этом прямой ход метода Гаусса заканчивается. Далее активным становится (N-1) – й процессор и аналогичным образом организуется обратный ход, в результате которого матрица A приводится к единичной. На этом этапе каждая ведущая строка состоит только из одного ненулевого элемента и после приведения матрицы A к единичной на месте последнего столбца расширенной матрицы $\overline{A}$ оказываются значения вектора-столбца искомого решения $\vec{x}$.

3.1. Порядок выполнения работы

1. Изучите приведенную в Приложении 4 параллельную программу, написанную на MPI, для решения описанной выше задачи. Число строк в матрице A задайте равным 480 (для удобства счета на 2, 4, 8 и 16 процессорах) а значение N равным целому числу от деления M на число процессоров без остатка. Чтобы выбрать исходную матрицу, определитель которой отличен от нуля, можно поступить следующим образом. Нужно построить нижнюю (или верхнюю) треугольную матрицу D, элементы главной диагонали которой отличны от нуля, значения же остальных ее элементов можно задать произвольными. Тогда определитель матрицы $A = D \cdot D^T$ будет отличен от нуля. Далее, задайте про-

извольные значения вектора-столбца $\vec{x}$, тогда значения вектора-столбца $\vec{y}$ найдутся из соотношения $\vec{y} = A\vec{x}$. Решите прямую задачу $A\vec{x} = \vec{y}$ и сравните полученные значения вектора-столбца $\vec{x}$ (точнее, его части в каждом процессоре) с исходными.

2. Вставьте в программу замеры времени счетной части программы (t_c) в каждом процессоре и времени (t_mpi), которое занимают обмены с соседними процессорами ("накладные расходы"). Проведите расчеты на двух, четырех, восьми и шестнадцати процессорах и постройте зависимость времени счета от числа процессоров. Постройте в виде таблицы зависимости t_c и t_mpi от числа процессоров и определите "оптимальное" число процессоров, поддерживающее баланс между временными затратами на счет и "накладными расходами". Постройте зависимость коэффициента ускорения от числа процессоров.

3. Разберите приведенную в Приложении 5 параллельную программу на OpenMP, в которой полагается, что матрица $\overline{A}$ целиком располагается в памяти узла. Постройте зависимость коэффициента ускорения от числа ядер (потоков) на узле. Сравните эффективность работы обеих программ.

4. Лабораторная работа № 4. Вычисление определителя методом Гаусса

Способ вычисления определителя квадратной матрицы A – $n \times n$ с помощью метода Гаусса заключается в приведении матрицы A к верхней треугольной матрице с помощью тех же элементарных преобразований, которые использовались при решении систем линейных алгебраических уравнений (СЛАУ). Однако есть принципиальное отличие – нельзя делить ведущую строку на ее главный элемент. Итак, пусть дана матрица A

$$A = \begin{pmatrix} a_{11}\, a_{12} \ldots a_{1n} \\ a_{21}\, a_{22} \ldots a_{2n} \\ \ldots\ldots\ldots\ldots \\ a_{n1}\, a_{n2} \ldots a_{nn} \end{pmatrix}$$

и нужно вычислить значение ее определителя. Пусть $a_{11} \neq 0$. Этого всегда можно добиться путем перестановки строк или столбцов, учитывая при этом, что нечетное число таких перестановок меняет знак определителя. Далее, ум-

ножая последовательно первую строку на $-a_{11}/a_{i1}$ и складывая ее с i – той $(i = 1,2,\ldots,n)$, получим преобразованную матрицу

$$A_1 = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ 0 & a_{22}^{(1)} & \ldots & a_{2n}^{(1)} \\ \ldots & \ldots & \ldots & \ldots \\ 0 & a_{n2}^{(1)} & \ldots & a_{nn}^{(1)} \end{pmatrix},$$

определитель которой равен определителю исходной матрицы. Продолжая этот процесс дальше, приведем ее к верхней треугольной матрице

$$A_{n-1} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ 0 & a_{22}^{(1)} & \ldots & a_{2n}^{(1)} \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & a_{nn}^{(n-1)} \end{pmatrix},$$

значение определителя которой равен произведению элементов ее главной строки $\det(A) = (-1)^m a_{11} \cdot a_{22}^{(1)} \cdot \ldots \cdot a_{nn}^{(n-1)}$, где m – число перестановок строк и/или столбцов в процессе приведения матрицы к верхней треугольной.

Рассмотрим пример организации параллельных вычислений в этом методе. Пусть исходная матрица A размерности M, как и в предыдущей практической работе, разрезается на N полос равной ширины (N выбирается кратным M), каждая из которых загружается в свой процессор. Далее в нулевом (активном) процессоре первая (ведущая) строка передается всем остальным процессорам. Во всех процессорах с помощью этой строки элементарными преобразованиями все элементы первого столбца матрицы A (за исключением первой строки активного процессора) преобразовываются в нулевые. Затем в активном процессоре ведущей становится следующая строка и процесс продолжается до исчерпания строк в активном процессоре. После этого активным становится следующий (первый) процессор и все повторяется снова до тех пор, пока не исчерпаются все ведущие строки во всех процессорах. В результате матрица A приведется к верхней треугольной матрице, распределенной по всем процессорам. Далее, в каждом процессоре перемножаются элементы главной диагонали и результаты умножения собираются в нулевом процессоре, где умножением их и вычисляется окончательное значение определителя.

4.1. Последовательность выполнения работы

1. На основе программы решения СЛАУ методом Гаусса, написанную для MPI

(Приложение 4), напишите параллельную программу для решения описанной выше задачи. Число строк в матрице A задайте равным 480 (для удобства счета на 2, 4, 6 и 12 процессорах) а значение N равным делению M на число процессоров. В качестве тестового примера составьте произвольную верхнюю треугольную матрицу, элементы главной диагонали которой отличны от нуля, вычислите ее определитель и запомните его значение в нулевом процессоре. В конце программы сравните полученное значение с тестовым, чтобы убедиться в правильности работы вашей параллельной программы.

2. Вставьте в программу замеры времени счетной ее части (t_c) в каждом процессоре и времени (t_mpi), которое занимают обмены с соседними процессорами ("накладные расходы"). Проведите расчеты на двух, четырех, шести и двенадцати процессорах и постройте зависимость времени счета от числа процессоров. Постройте в виде таблицы зависимости t_c и t_mpi от числа процессоров и определите "оптимальное" число процессоров, поддерживающее баланс между временными затратами на счет и "накладными расходами".

3. На основе программы на OpenMP для решения СЛАУ методом Гаусса (Приложение 5) напишите параллельную программу для вычисления определителя и постройте зависимость коэффициента ускорения от числа ядер (потоков) на узле.

5. Лабораторная работа № 5. Вычисление обратной матрицы методом Гаусса

Для вычисления обратной матрицы можно также использовать метод Гаусса, обобщив его на решение матричных уравнений. Действительно, согласно определению, матрица A^{-1}, обратная к матрице A, должна удовлетворять следующему матричному уравнению $A \cdot A^{-1} = E$, если рассматривать элементы матрицы A^{-1} как неизвестные. Расширенная матрица этой системы размерности $n \times 2n$ имеет вид

$$\overline{A} = \begin{pmatrix} a_{11}\ a_{12} \ldots a_{1n} & 1\ 0\ 0 \ldots 0 \\ a_{21}\ a_{22} \ldots a_{2n} & 0\ 1\ 0 \ldots 0 \\ \ldots\ \ \ldots\ \ \ldots & \ldots\ \ \ldots\ \ \ldots \\ a_{n1}\ a_{n2} \ldots a_{nn} & 0\ 0\ 0 \ldots 1 \end{pmatrix}$$

и после прямого и обратного ходов метода Гаусса, как это описано в работе 3, в расширенной матрице на месте элементов исходной матрицы будут находить-

ся элементы единичной матрицы, а на месте элементов единичной матрицы – элементы обратной $A^{-1} = \{b_{ij}\}$:

$$\overline{A} = \begin{pmatrix} 1\ 0 \ldots 0\ b_{11}\ b_{12} \ldots b_{1n} \\ 0\ 1 \ldots 0\ b_{21}\ b_{22} \ldots b_{2n} \\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots \\ 0\ 0 \ldots 1\ b_{n1}\ b_{n2} \ldots b_{nn} \end{pmatrix}.$$

Для организации параллельных вычислений расширенная матрица, как и в работе 3, разрезается на N полос равной ширины (N выбирается кратным M), каждая из которых загружается в свой процессор. Далее все действия аналогичны действиям, описанным в работе 3. Обратная матрица будет также распределена по N полосам, находящимся в соответствующих процессорах. Вставьте проверку на возможность обращения определителя матрицы A в нуль с блокирующей остановкой программы в этом случае.

5.1. Порядок выполнения работы

1. На основе программы решения СЛАУ методом Гаусса, написанную для MPI (Приложение 4), напишите параллельную программу для решения описанной выше задачи. Число строк в матрице A задайте равным 480 (для удобства счета на 2, 4, 6 и 12 процессорах) а значение N равным делению M на число процессоров. В качестве тестового примера задайте произвольную матрицу A, определитель которой отличен от нуля и с помощью написанной программы вычислите для нее обратную. Для проверки правильности работы вашей программы воспользуйтесь известным соотношением AX = E, причем для умножения матриц используйте параллельную программу из лабораторной работы № 2.
2. Вставьте в программу замеры времени счетной ее части (t_c) в каждом процессоре и времени (t_mpi), которое занимают обмены с соседними процессорами (“накладные расходы”). Проведите расчеты на двух, четырех, шести и двенадцати процессорах и постройте зависимость времени счета от числа процессоров. Постройте в виде таблицы зависимости t_c и t_mpi от числа процессоров и определите “оптимальное” число процессоров, поддерживающее баланс между временными затратами на счет и “накладными расходами”.
3. На основе программы на OpenMP для решения СЛАУ методом Гаусса (Приложение 5) напишите параллельную программу для вычисления обратной матрицы и постройте зависимость коэффициента ускорения от числа ядер (потоков) на узле.

6. Лабораторная работа № 6.
Решение систем нелинейных алгебраических уравнений

Целью лабораторной работы является: 1). Написание параллельных программ на OpenMP (или на MPI) для решения системы нелинейных уравнений методами Ньютона и Якоби. 2). Получение оценок коэффициентов ускорения вычислений данными методами в зависимости от числа процессоров (нитей).
3). Сопоставление (в виде таблиц) времени решения указанными методами на разном числе процессоров (нитей).

Система замкнутых нелинейных алгебраических уравнений имеет следующий вид

$$\begin{aligned} &f_1(x_1, x_2, \ldots, x_n) = 0, \\ &f_2(x_1, x_2, \ldots, x_n) = 0, \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ &f_n(x_1, x_2, \ldots, x_n) = 0, \end{aligned} \tag{6.1}$$

где $f_i, (i = 1, 2, \ldots, n)$ - функции действительных переменных $x_1, x_2, \ldots, x_n$. В векторном виде она записывается в более компактной форме

$$\vec{f}(\vec{x}) = 0, \quad \vec{x} = (x_1, x_2, \ldots, x_n), \vec{f} = (f_1, f_2, \ldots, f_n)^T.$$

Для ее решения предлагается использовать два метода, которые достаточно эффективно распараллеливаются на кластерной системе – метод Ньютона и метод Якоби.

Метод Ньютона для решения системы (6.1) записывается в виде двух эквивалентных формул (верхний индекс есть номер итерации)

$$\vec{f}'(\vec{x}^n) \cdot (\vec{x}^{n+1} - \vec{x}^n) = -\vec{f}(\vec{x}^n), \quad n = 0, 1, 2, \ldots, \tag{6.2}$$

где $\vec{f}'(\vec{x}^n)$ есть матрицы Якоби, вычисляемая в точке $\vec{x}^n$. Или

$$\vec{x}^{n+1} = \vec{x}^n - [\vec{f}'(\vec{x}^n)]^{-1} \vec{f}(\vec{x}^n), \quad n = 0, 1, \ldots, \tag{6.3}$$

где

$$[\vec{f}'(\vec{x}^n)]^{-1} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \dots\dots & \dots\dots & & \dots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}^{-1}$$

есть матрица, обратная к матрице Якоби, также вычисляемая в точке $\vec{x}^n$.
Вторая формула более предпочтительна, поскольку обращать матрицу Якоби можно не на каждом итерационном шаге, что, хотя и снижает скорость сходимости итераций, но зато позволяет сократить время счета, поскольку операция обращения матрицы является достаточно трудоемкой. Условия сходимости метода Ньютона для (6.1) определяются теоремой Канторовича, однако воспользоваться этой теоремой при решении практических задач удается в довольно редких случаях. Объектами распараллеливания в методе Ньютона являются процедуры обращения матрицы Якоби и умножение полученной обратной матрицы на вектор.

Метод Якоби для решения системы (6.1) позволяет находить новое значение вектора $\vec{x}^{n+1}$ последовательно для всех его координат, причем первая координата этого вектора находится из решения первого уравнения системы, вторая – из второго и т.д. С этой целью система (6.1) записывается в следующем виде

$$f_i(x_1^n, x_2^n, \dots, x_{i-1}^n, x_i^{n+1}, x_{i+1}^n, \dots, x_i^n) = 0, \quad i = 1, 2, \dots, n \ .$$

Таким образом, в этом методе для нахождения нового значения каждой координаты x_i^{n+1} вектора $\vec{x}^{n+1}$ нужно решить одно скалярное нелинейное уравнение каким-либо из численных методов решения одного нелинейного уравнения, например, тем же методом Ньютона. Такая независимость вычислений позволяет эффективно реализовывать данный алгоритм на кластерной системе.

В качестве контроля за сходимостью итерационных процессов можно использовать условие малости нормы вектора $\left\|\vec{f}(\vec{x}^n)\right\| = \sqrt{\sum_{i=1}^{n} f_i^2(\vec{x}^n)} \le \varepsilon$, обычно называемое условием малости невязки, где ε – заданная малая величина.

6.1. Порядок выполнения работы

1. Для решения методом Ньютона заданной ниже системы нелинейных уравнений составить матрицу Якоби, для обращения которой можно использовать параллельный алгоритм метода Гаусса из лабораторной работы № 5. Параллельный алгоритм умножения матрицы на вектор напишите сами (за основу также можно взять алгоритм из лабораторной работы № 2).

$$\vec{f} = \begin{pmatrix} (1-x_1^3)+x_2^2 \\ x_1^2+(1-x_2^3)+x_3^2 \\ x_2^2+(1-x_3^3)+x_4^2 \\ \cdots\cdots\cdots\cdots\cdots \\ x_{j-1}^2+(1-x_j^3)+x_{j+1}^2 \\ \cdots\cdots\cdots\cdots\cdots \\ x_{n-1}^2+(1-x_n^3) \end{pmatrix}$$

2. Напишите параллельную программу для решения данной задачи. Число уравнений задайте равным 800 (n = 800)
3. Задайте максимальное значение вектора невязки $\varepsilon = 10^{-7}$. Вставьте в программу замеры общего времени счета и проведите расчеты на двух, четырех, шести и восьми процессорах. Постройте коэффициент ускорения вычислений в зависимости от числа процессоров. Сведите в таблицу времена счета при различном числе процессоров.
4. Напишите параллельную программу для решения данной системы уравнений методом Якоби при аналогичном значении вектора невязки. Для вычисления любой из координат x_i^{n+1} вектора $\vec{x}^{n+1}$ используйте метод Ньютона

$$x_i^{n+1} = x_i^n - f_i/(\partial f_i/\partial x_i), \quad i = 1,2,\ldots,n,$$

причем все вычисления на одном итерационном шаге для всех значений индекса i выполняются параллельно.
5. Проделайте для метода Якоби те же расчеты, что и в п.2 метода Ньютона.
6. Напишите общий отчет по лабораторной работе.

7. Лабораторная работа № 7. Решение систем жестких обыкновенных дифференциальных уравнений

Целью лабораторной работы является: 1). Написание параллельных программ на OpenMP (или на MPI) для решения задачи Коши системы жестких ОДУ явными и неявными методами. 2). Получение оценок коэффициентов ускорения вычислений данными методами в зависимости от числа процессоров (нитей).
3). Сопоставление (в виде таблиц) времени решения указанными методами на разном числе процессоров (нитей).

Рассматривается задача Коши для системы жестких ОДУ следующего вида

$$\frac{d\vec{y}}{dx} = A\vec{y}, \quad \text{при } x = 0 \quad \vec{y} = \vec{y}^0, \ (\vec{y}^0 \text{ - заданный вектор}), \tag{7.1}$$

где A – матрица, все собственные числа $\lambda(A)$ которой вещественные, отрицательные и имеют большой разброс по модулю, т.е. $\max|\lambda(A)| >> \min|\lambda(A)|$.

В этом случае при решении системы (7.1) явными методами Рунге-Кутта возникает ограничении на шаг интегрирования h вида $h \le 1/\max|\lambda(A)|$.

Так, **явный метод** Рунге-Кутта четвертого порядка точности для решения (7.1) на одном шаге интегрирования запишется

$$\vec{y}_{i+1} = (E + Ah + A^2h^2/2 + A^3h^3/6 + A^4h^4/24)\vec{y}_i \tag{7.2}$$

Свободный от ограничений на шаг интегрирования **неявный метод** тоже четвертого порядка точности имеет следующий вид:

$$\vec{y}_{i+1} = (E - Ah + A^2h^2/2 - A^3h^3/6 + A^4h^4/24)^{-1}\vec{y}_i \tag{7.3}$$

и для своей реализации требует уже обращения матрицы, что при больших размерностях уравнений (7.1) приводит к существенному увеличению его трудоемкости. Однако шаг интегрирования неявного метода может быть выбран много больше явного метода, что является несомненным преимуществом неявного метода, поскольку в этом случае для получения решения нужно сделать намного меньше шагов.

Получить значение максимального по модулю собственного числа матрицы A, необходимого для оценки допустимого значения шага интегрирования явного метода (7.2), можно с помощью следующего алгоритма.

Выбирается произвольный ненулевой вектор $\vec{x}_0 = (x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})$, где $x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)}$ – произвольные числа и организуется следующий итерационный процесс. Вычисляется $\vec{z}_0 = \dfrac{\vec{x}_0}{\|\vec{x}_0\|}$, а затем по рекуррентным соотношениям

$\vec{x}_k = A\vec{z}_{k-1}$, вычисляется $\lambda_{max}^{(k)} = (\vec{x}_k, \vec{z}_{k-1})$, и $\vec{z}_k = \dfrac{\vec{x}_k}{\|\vec{x}_k\|}$, где $k = 1, 2, \dots,$.

Итерации прекращаются при выполнении условия $\dfrac{|\lambda_{max}^{(k+1)} - \lambda_{max}^{(k)}|}{\lambda_{max}^{(k)}} \le \varepsilon$, где ε – заданная относительная точность вычисления λ_{max}. Обычно достаточно задать значение $\varepsilon = 10^{-5}$.

В приведенных методах распараллеливаются алгоритмы умножения матриц, умножения матрицы на вектор и обращения матрицы.

7.1. Порядок выполнения работы

1. Для системы (7.1) задается трех диагональная матрица A, элементы главной диагонали которой определены как $a_{ii} = -(n^2 + 1/(1+x))$, а элементы нижней и верхней диагоналей равны $a_{i,i+1} = a_{i.i-1} = n/(1+x)$. Такой способ задания матрицы гарантирует, что ее максимальное по модулю собственное число будет иметь наибольшее значение при x = 0, т.е. шаг интегрирования для метода (7.2) достаточно определить только в начальной точке. Поэтому сначала необходимо с помощью изложенного выше итерационного метода найти один раз при x = 0 значение $\max|\lambda(A)|$, вычислить $h_{expl} = 1/\max|\lambda(A)|$ и использовать его для данного метода. Шаг интегрирования для неявного метода выберите $h_{impl} = 0.01$, сравните значения этих шагов.

2. Напишите параллельные программы для решения данной задачи на отрезке $0 \le x \le 1$ обеими методами. Число уравнений задайте равным 800 (n = 800). Вставьте в программы замеры общего времени счета и проведите расчеты на двух, четырех, шести и восьми процессорах. Постройте коэффициент ускорения вычислений в зависимости от числа процессоров. Сведите в таблицу для обеих методов времена счета при различном числе процессоров.

3. Напишите общий отчет по лабораторной работе.

8. Лабораторная работа № 8. Решение двумерного уравнения теплопроводности с помощью явной разностной схемы

Целью лабораторной работы является: 1). Написание параллельной программы на MPI для решения двумерного уравнения теплопроводности в прямоугольной области с помощью явной конечно-разностной схемы. 2). Получение оценок коэффициентов ускорения вычислений в зависимости от числа процессоров. 3). Составление в виде таблицы времени решения задачи на различных разностных сетках при заданном числе процессоров.

Рассматривается начально-краевая задача для уравнения теплопроводности в прямоугольной области $D\{t \ge 0, 0 \le x \le 5, 0 \le y \le 2\}$

$$\frac{\partial T}{\partial t} - \sigma\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = 0, \qquad (8.1)$$

где: $T(t,x,y), \sigma = 10^{-3}$ - температура и коэффициент температуропроводности соответственно, с начальным условием $T(0,x,y) = 300$ и с краевыми условиями

первого рода:

$$T(t,x,0) = 400 + 10x; \quad T(t,0,y) = 400 + 50y; \quad T(t,x,2) = 500 - 10x^2; \quad T(t,5,y) = 450 - 50y^2. \tag{8.2}$$

Для решения (8.1), (8.2) используется явная разностная схема

$$\frac{y_{i.j}^{n+1} - y_{i,j}^{n}}{\tau} - \sigma[\frac{y_{i+1,j}^{n} - 2y_{i,j}^{n} + y_{i-1,j}^{n}}{h_x^2} + \frac{y_{i,j+1}^{n} - 2y_{i,j}^{n} - y_{i,j-1}^{n}}{h_y^2})] = 0, \tag{8.3}$$

где: $\tau, h_x = 5/N_x, h_y = 2/N_y$ - шаги сетки по времени и по пространственным переменным соответственно, N_x, N_y - число шагов сетки по осям координат. Схема имеет порядок аппроксимации $O(\tau + h_x^2 + h_y^2)$ и устойчива при выполнении условия

$$\tau \le \min(\frac{h_x^2}{2\sigma}, \frac{h_y^2}{2\sigma}) \ . \tag{8.4}$$

8.1. Порядок выполнения работы

1. Для проведения расчетов на MPI необходимо провести декомпозицию области решения по координате OX на подобласти по числу процессоров. В каждой из подобластей вводятся фиктивные дополнительные узлы разностной сетки, которые используются для реализации обменных краевых условий. Схема их реализации показана на рис. 3.

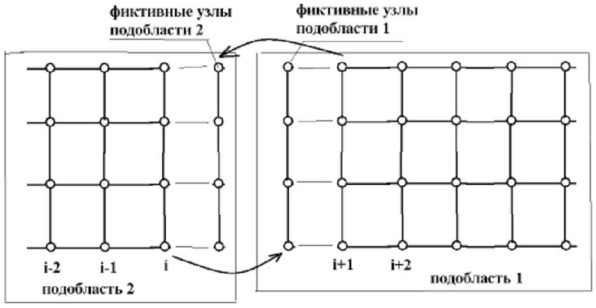

Рис. 3. Схема обменных краевых условий

2. Задайте размер разностной сетки $N_x = 1000; \quad N_y = 800$, шаг интегрирования по времени определите из условия (8.4). Напишите параллельную программу на MPI для реализации разностной схемы (8.3) с учетом декомпозиции области решения, значения сеточных функций на границах области определите из граничных условий (8.2).

3. Проведите расчеты на двух, четырех, восьми и двенадцати процессорах и постройте зависимость коэффициента ускорения от числа процессоров.

4. Напишите аналогичную программу на OpenMP. В этом случае декомпозиция области решения не требуется и достаточно только распараллелить соответствующие циклы. Проведите расчеты на двух, четырех и восьми нитях и постройте зависимость коэффициента ускорения от числа процессоров. Сравните поведение коэффициентов ускорения на MPI и OpenMP.

3. Напишите общий отчет по лабораторной работе.

9. Лабораторная работа № 9. Имитационное моделирование системы массового обслуживания (СМО)

Система состоит из узла обслуживания, имеющего N независимых исполнительных устройств (каналов) и устройства ввода. Поток входных данных (заявки) поступает во вводное устройство случайным образом с заданной плотностью распределения вероятности $p_{in}(t)$ появления заявки по времени. Если в момент ее поступления свободно хотя бы одно устройство, заявка начинает выполняться на этом устройстве, в противном случае заявки игнорируется (система обслуживания с отказами). Время выполнения заявки каждым i – тым каналом также является случайной величиной, плотность вероятности которой $p_i(t)$ считается известной. При окончании выполнения заявки i – тым каналом узел обслуживания проверяет, имеются ли заявки в устройстве ввода и если заявки нет, то i – тый канал простаивает. Каждый канал исполняет в данный момент времени только одну заявку и каждая заявка исполняется только одним каналом.

Целью исследования функционирования такой системы является определение среднего числа отказов на обслуживание в заданном интервале времени. Полагается, что поток заявок описывается законом распределения Пуассона. В этом случае функция распределения вероятности имеет вид $P(t) = 1 - \exp(-\lambda t), \quad 0 \le t < \infty$, а ее функция плотности распределения есть

$$p_{in}(t) = \lambda_{in} \exp(-\lambda_{in} t) \ , \qquad (1)$$

где $\lambda_{in} = \dfrac{1}{T_{in}}$ есть интенсивность поступления заявок, T_{in} – среднее значение интервала между поступлением очередных заявок. Время обслуживания одной заявки $t_{обсл}$ каждым каналом также является случайной величиной, которая подчиняется распределению Пуассона. В этом случае плотность вероятности для времени обслуживания i-го канала имеет аналогичный вид

$$p_i(t) = \lambda_i \exp(-\lambda_i t) \ , \qquad (3)$$

где $\lambda_i = \dfrac{1}{T_i}$ есть скорость обслуживания заявки, T_i – среднее время обслуживания заявки.

При расчете времени обслуживания и времени поступления очередной заявки в систему необходимо решать интегральные уравнения вида

$$\int_0^{\tau} p(t)dt = \gamma, \qquad (4)$$

где τ – искомый момент времени, p(t) – функция плотности вероятности, γ – случайная величина, равномерно распределенная на отрезке [0, 1]. Для распределения Пуассона интеграл (4) вычисляется аналитически и величина τ определяется следующим образом

$$\tau = -(1/\lambda)\ln(\gamma). \qquad (5)$$

Случайная величина γ вычисляется с помощью стандартных программ, которые имеются во всех языках программирования.

9.1. Общая схема алгоритма

Каналы есть вещественный массив K[N], где N – число каналов, каждая компонента которого содержит время обслуживания заявки, t – время; N_{ot} - число отказов, начальное значение которого равно нулю; N_{in} - число заявок; M - заданное число циклов моделирования (достаточно большое число). Число каналов N, значения λ_i для всех каналов и интенсивность потока заявок λ_{in} должна быть заданы.

Моделирование процесса обслуживания происходит на конечном интервале времени $t \in [t_0, t_k]$. Исходные значения параметров: N_{in}=0; $t_0 = 0$; $t_i = 0$ $(i = 1, 2, \ldots, N)$. Последовательность расчета одного цикла моделирования следующая:

1). Вычисляется $T_{min} = \underbrace{\min(t_i)}_{1 \le i \le N}$ и затем определяется момент времени поступления очередной заявки t_{in} по формуле (5): $t_{in} = t_{in-1} - (1/\lambda_{in})\ln(\gamma)$. К счетчику заявок N_{in} добавляется единица.

2). Проверяется, имеются ли свободные каналы. Для этого достаточно проверить выполнение условия

$$T_{min} \le t_{in}. \qquad (6)$$

3). Если условие (6) выполнено, то свободным является i – тый канал, для которого $t_i = T_{min}$ (если свободных каналов несколько, то выбирается канал с наименьшим номером i) и для него определяется время обслуживания заявки $t_i = t_{in} - (1/\lambda_i)\ln(\gamma)$, которое записывается в массив K[i].

4).Если условие (6) не выполнено, то свободных каналов нет и полагается, что система выдала отказ в обслуживании заявки. Число отказов N_{ot} увеличивается на единицу.

5). Проверяется выполнение условия $t_{in} < t_k$. Если оно выполняется, то все повторяется, начиная с п.1. Если нет, расчет текущего цикла моделирования прекращается, счетчик циклов моделирования увеличивается на единицу, все времена обнуляются: $t_0 = 0$; $t_i = 0$ $(i = 1, 2, \ldots, N)$ и цикл вычислений повторяется.

После выполнения M циклов определяется среднее число отказов на интервале времени $[t_0, t_k]$ $\hat{N}_{ot} = N_{ot} / N_{in}$, которое также является и вероятностью отказа в обслуживании заявок за указанное время.

9.2. Порядок выполнения работы

1. Поскольку каждый цикл моделирования выполняется независимо от других, то процесс моделирования СМО может выполняться на n процессорах при использовании MPI или на n нитях при использовании OpenMP. Тогда число циклов на каждом процессоре (нити) будет m = M/n, за счет чего и достигается ускорение вычислений. Вероятность отказа в обслуживании заявок будет равно сумме величин $\hat{N}_{ot}^{(i)}$, полученных в каждом процессоре (нити), осредненной по числу процессоров (нитей), т.е. $\hat{N}_{ot} = (\sum_{i=1}^{n} N_{ot}^{(i)}) / n$.

2. Напишите параллельные программы на MPI и OpenMP. Задайте число каналов N = 50, значения λ_i=0.3 для всех каналов и интенсивность потока заявок $\lambda_{in} = 10$, $t_k = 3600$. На основе расчетов на одном процессоре определите минимальное число циклов M, при котором имеет место статистически устойчивое значение величины $\hat{N}_{ot}$, т.е. когда вероятность $\hat{N}_{ot}$ уже практически не зависит от M.

3. Проведите расчеты на двух, четырех, восьми и шестнадцати процессорах на MPI и на двух, четырех, шести и восьми нитях на OpenMP. Постройте зависимость коэффициента ускорения вычислений от числа процессоров (нитей).

4. На наибольшем из числа процессоров проведите расчеты при значениях интенсивности потока заявок λ_{in} = 20, 30, 40 и постройте зависимость $\hat{N}_{ot}(\lambda_{in})$.

ЛИТЕРАТУРА

1. Рычков А.Д. Численные методы и параллельные вычисления: Учебное пособие/ СибГУТИ - Новосибирск, 2007. – 144с.
2. Рычков А.Д., Захарова Т.Э. Основы линейной алгебры и аналитической геометрии: Учебное пособие/ СибГУТИ - Новосибирск, 2007. – 112с.
3. Корнеев В.Д. Параллельное программирование в MPI – Москва – Ижевск: Ин-т компьютерных исследований, 2003. – 304с.
4. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления.- Спб:БХВ – Петербург, 2002. – 608с.
5. Бусленко Н.П. Моделирование сложных систем. М. Наука, 1968.
6. Климов Г.П. Стохастические системы обслуживания. М. Наука, 1966.
7. Кофман А., Крюон Р., Массовое обслуживание, теория и приложение. Мир, 1965.

ПРИЛОЖЕНИЕ

Приложение 1. Организация блокированных обменов в MPI

```
/* Простая передача-прием:  MPI_Send, MPI_Recv Завершение по ошибке:
MPI_Abort  */
  #include <mpi.h>
  #include <stdio.h>
/* Идентификаторы сообщений */
  #define tagFloatData  1
  #define tagDoubleData 2

int main(int argc, char **argv)
  {
    int size, rank, count,i;
    float floatData[10];
    double doubleData[20];
    MPI_Status status;
 /* Инициализация библиотеки MPI*/
    MPI_Init(&argc, &argv);
 /* Каждая ветвь узнает количество задач в стартовавшем приложении */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
 /* и свой собственный номер: от 0 до (size-1) */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 /* Пользователь должен запустить ровно две задачи, иначе ошибка */
    if(size != 2)
      {
     /* Задача с номером 0 сообщает пользователю об ошибке */
       if(rank == 0)
         printf("Error: two processes required instead of %d, abort\n",
                                                 size);

     /* Все ветви в области связи MPI_COMM_WORLD будут стоять, пока ветвь
0 не выведет сообщение.
      */
       MPI_Barrier(MPI_COMM_WORLD);
     /* Без точки синхронизации может оказаться, что одна из ветвей вызовет
MPI_Abort раньше, чем успеет отработать printf() в ветви 0, MPI_Abort немед-
ленно принудительно завершит  все ветви и сообщение выведено не будет. Все
задачи аварийно завершают работу */
       MPI_Abort(
         MPI_COMM_WORLD,  /* Описатель области связи, на которую
```

```
      распространяется действие ошибки */
    MPI_ERR_OTHER);  /* Целочисленный код ошибки */
   return -1;
  }

if(rank == 0)
  {
  for(i=0;i<10;i++)floatData[i]=i;for(i=0;i<20;i++)doubleData[i]=-i;
 /* Передача из ветви 0 в ветвь 1 */
   MPI_Send(
    floatData,       /* адрес передаваемого массива */
    5,              /* передано 5 элементов, т.е.
                       floatData[0]..floatData[4] */
    MPI_FLOAT,       /*  тип переданных элементов */
    1,              /*  кому: ветви 1 */
    tagFloatData,    /*  идентификатор сообщения */
    MPI_COMM_WORLD);  /*  описатель области связи, через
                       которую происходит передача */

 /* и еще одна передача: данные другого типа */

   MPI_Send(doubleData, 6, MPI_DOUBLE, 1, tagDoubleData,
                                MPI_COMM_WORLD);
  }
else
  {
 /* Ветвь 1 принимает  данные от ветви 0,
  дожидается сообщения и помещает пришедшие данные в буфер */
   MPI_Recv(
    floatData,         /*  адрес массива приемного буфера */
    10,   /*максимальная  длина приемного буфера */
    MPI_FLOAT,         /*  сообщаем MPI, что пришедшее
                       сообщение состоит из чисел
                       типа 'float' */
    0,               /* от кого: от ветви 0 */
    tagFloatData,      /* ожидаем сообщение с таким  идентификатором */
    MPI_COMM_WORLD,    /*  описатель области связи, через
                       которую ожидается приход сообщения */
    &status);          /*  сюда будет записан статус завершения
                       приема */

 /* Вычисляем фактически принятое количество данных */
```

```
      MPI_Get_count(
        &status,      /* статус завершения приема в MPI_Recv */
        MPI_FLOAT,    /* сообщаем MPI, что пришедшее сообщение
                         состоит из чисел типа 'float' */
        &count);      /* сюда будет записан результат */
        for(i=0;i<count;i++)if(floatData[i]!=i){printf("Error 1: i=%3d\n",i);}

    /* Выводим фактическую длину принятого на экран */
      printf("rank = %d Received %d elems float\n", rank, count);

    /* Аналогично принимаем сообщение с данными типа double
     */
      MPI_Recv(doubleData, 20, MPI_DOUBLE,
                0, tagDoubleData, MPI_COMM_WORLD, &status );
      MPI_Get_count(&status, MPI_DOUBLE, &count);
 for(i=0;i<count;i++)if(doubleData[i]!=-i){printf("Error 2: i=%3d\n",i);}
 printf("Received %d elems double\n",count);

 }
MPI_Barrier(MPI_COMM_WORLD);
 /* Обе ветви завершают выполнение */
   MPI_Finalize();
   return 0;
     }
```

Приложение 2. Обмен данными на системе компьютеров с топологией связи "кольцо"

```
/* Сдвиг данных на кольце компьютеров */
#include <mpi.h>
#include <stdio.h>
#define NUM_DlMS 1
int main(   int argc,   char** argv )
{
int     rank,   size,   l,   A,   B,  dims[NUM_DIMS];
int     periods [NUM_DIMS] ,  source,  dest;
int     reorder = 0;
MPI_Comm      comm_cart;
MPI_Status      status;
MPI_Init(&argc, &argv);
 /*  Каждая ветвь узнает обшее количество ветвей */
```

```
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
 /* и свой номер    от 0 до  (size-1)  */
A=rank;B=-1;
/* Обнуляем массив dims и заполняей насеяв periods для топологии "кольцо" */
for( i= 0; i < NUM_DIMS; i++) { dims[i] = 0; periods[i] = 1; }
/* Заполняем массив dims, где указываются размеры (одномерной) решетки */
MPI_Dims_create(size, NUM_DIMS, dims);
/* Создаем топологию "кольцо" с communicator(om) comm_cart */
MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, reorder,
&comm_cart);
/* Каждая ветвь находит своих соседей вдоль кольца, в направлении
*       больших значений рангов */
MPI_Cart_shift(comm_cart, 0, 1, &source, &dest);
/* 0-ветвь инициирует передачу данных (значение своего ранга) вдоль
*       кольца, и принимает это же значение от ветви size-1 */
if(rank == 0)
MPI_Send(&A, 1, MPI_lNT, dest, 12, comm_cart);
MPI_Recv(&B, 1, MPI_INT, source, 12, comm_cart, &status);
printf("rank= %d A=%d B=%d \n", rank, A, B); }
/* Работа всех остальных ветвей */
else{ MPI_Recv(&B, 1, MPI.INT, source, 12, comm_cart, &status);
MPI_Send(&B, 1, MPI.INT, dest, 12, comm_cart); }
/* Все ветви завершают системные процессы, связанные с топологией
comm_cart и завершаю выполнение программы */
MPI_Comm_free(&comm_cart);
MPI_Finalze(); return 0;}
```

Приложение 3. Обмен данными на системе компьютеров с топологией связи "линейка"

```
/*
Сдвиг данных на линейке процессов с использованием MPI_PROC_NULL про-
цессов
*/
#inlude <mpi.h>
#include <stdio.h>
int main(int argc,   char **argv)
{
int     rank,  size,  i,  A,  B,  dims[1] ;
int     periods[1],  new_coords[1];
```

```
int     souirceb,  destb,  sourcem,  destm;
int     reorder = 0;
MPI_Comm   comm_cart;
MPI_Status    status;
MPI_Init(&argc, &argv);
/* Каждая ветвь узнает количество ветвей */
MPl_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 /* и свой номер от 0 но (size-1) */
/* Обнуляем массив dims и заполняем массив periods для топологии "линейка"
*/
for(i = 0; i < 1; i++) { dims[i] = 0; periods[i] = 0; }
/* Заполняем массив dims, где указываются размеры (одномерной) решетки */
MPI_Dims_create(size, 1, dims);
 /* Создаем топологию "линейка" с communicator(ом) comm_cart */
MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, reorder,
&comm_cart);
 /* Отображаем ранги в координаты и выводим их */
MPI_Cart_coords(comm_cart, rank, 1, new_coords);
A = new_coords[0];B = -1;
 /* Каждая ветвь находит своих соседей вдоль линейки, в направлении больших
значений номеров компьютеров и в направлении меньших значений номеров.
Ветви с номером new_coords[0] == 0 не имеют соседей с меньшим номером, по-
этому с этого направления эти ветви принимают данные от несуществующих
ветвей, т.е. от ветвей sourcem = MPI_PROC_NULL, и, соответственно, передают
данные в этом направлении ветвям destm = MPI_PROC_NULL Аналогично оп-
ределяется соседство для ветвей с номером new_coords[0] == dims[0] - 1
*/
if(new_coords[0] == 0){sourcem = destm = MPI_PROC_NULL;}
else{ sourcem = destm = new_coords[0]-l;}
if(new_coords[0]  == dims[0]-l){destb = sourceb =NPI_PROC_NULL;}
else{ destb = sourceb = new_coords[0]+l;}
 /* Каждая ветвь передает своя данные (значение переменной A) своей
соседней ветви с большим номером и принимает данные в B от
соседней ветви с меньшим номером. Свой номер и номер,
принятый в B выводятся на печать
*/
MPI_Sendrecv(&A, 1, MPI_INT, destb, 12, &B, 1, MPI_INT, sourcem, 12,
comm_cart, &status);
printf ("new_coords[0] = %d B = %d\n", new_coords[0] , B);
```

```
/* Сдвиг данных в противоположную сторону и вывод соответствующих данных */
MPI_Sendrecv(&A, 1, MPI_INT, destm, 12, &B, 1, MPI_INT, sourceb, 12,
comm_cart, &status);
printf("new_coords[0] = %d B = %d\n", new_coords[0], B);
 /* Все ветви завершают системные процессы, связанные с топологией
comm_cart и завершают выполнение программы */
MPI_Comm_free(&comm_cart);
MPI_Finalize();
return 0;
```

Приложение 4. Решение СЛАУ методом Гаусса на MPI.

```
/* Программа на MPI, распределение  данных - горизонтальными полосами.
(Запуск задачи на 8-ми процессах).
*/
  #include<stdio.h>
  #include<mpi.h>
/* Каждая ветвь задает размеры своих полос матрицы MA и вектора правой
части.
(Предполагаем, что размеры данных делятся без остатка
на количество  компьютеров.) */
  #define M 400
  #define N 50

/* Описываем массивы для полос расширенной матрицы - MA и вектор V для
приема данных. В последнем столбце расширенной матрицы и получим результат решения системы. */

double MA[N][M+1], V[M+1], MAD, R;

int main(int args, char **argv)
  { int size, MyP, i, j,k, m, p;
double t0,dt,t1,t2,t3,t4,dt1,dt2,dt3,dt4;
   MPI_Status stat;
 /* Инициализация библиотеки */
   MPI_Init(&args, &argv);
 /* Каждая ветвь узнает размер системы */
   MPI_Comm_size(MPI_COMM_WORLD, &size);
 /* и свой номер (ранг) */
```

```
   MPI_Comm_rank(MPI_COMM_WORLD, &MyP);

/* Каждая ветвь генерирует свою полосу матрицы A и свой отрезок вектора
правой части, который присоединяется дополнительным столбцом к A. Нулевая
ветвь генерирует нулевую полосу, первая ветвь - первую полосу  и т.д. (По диа-
гонали исходной матрицы - числа = 2, остальные числа = 1).
*/

   for(i = 0; i < N; i++) for(j = 0; j < M; j++)
         if((N*MyP+i) == j) MA[i][j] = 2.0;
          else  MA[i][j] = 1.0;

     for(j=0;j<M;j++)V[j]=-(double)(j+1)/2.;
   for(i = 0; i < N; i++)
     { MA[i][M]=0;for(j = 0; j < M; j++)
         MA[i][M]+=MA[i][j]*V[j];}
     /* Каждая ветвь засекает начало вычислений и производит вычисления */
   t0=MPI_Wtime();
 /* Прямой ход */
 /* Цикл p - цикл по компьютерам. Все ветви, начиная с нулевой,
последовательно приводят к диагональному виду свои строки. Ветвь, приводя-
щая свои строки к диагональному виду, назовем активной, строка, с которой
производятся  вычисления, так же назовем активной. */
   for(p = 0; p < size; p++)
     {
    /* Цикл k - цикл по строкам. (Все ветви "крутят" этот цикл). */
       for(k = 0; k < N; k++)
        { if(MyP == p)
            {
 /* Активная ветвь с номером MyP == p приводит свои строки к  диагональному
виду.  Активная строка k передается ветвям, с номером большим чем MyP */
             MAD = 1.0/MA[k][N*p+k];
             for(j = M; j >= N*p+k; j--)
               MA[k][j] = MA[k][j] * MAD;
                  t1=MPI_Wtime();
             for(m = p+1; m < size; m++)
 MPI_Send(&MA[k][0], M+1, MPI_DOUBLE, m, 1,MPI_COMM_WORLD);
             dt1=MPI_Wtime()-t1;
             for(i = k+1; i < N; i++)
               { for(j = M; j >= N*p+k; j--)
                   MA[i][j] = MA[i][j]-MA[i][N*p+k]*MA[k][j];
               }
```

```
        }
      /* Работа принимающих ветвей с номерами MyP > p */
        else if(MyP > p)
         {t2=MPI_Wtime();
          MPI_Recv(V, M+1, MPI_DOUBLE, p, 1,MPI_COMM_WORLD, &stat);
         dt2=MPI_Wtime()-t2;
          for(i = 0; i < N; i++)
            { for(j = M; j >= N*p+k; j--)
               MA[i][j] = MA[i][j]-MA[i][N*p+k]*V[j];
            }
         }
       }       /* конец цикла по k */
    }          /* конец цикла по p */
                                          /* Обратный ход */
/* Циклы по p и k аноалогичны, как и при прямом ходе. */
  for(p = size-1; p >= 0; p--)
   { for(k = N-1; k >= 0; k--)
      {
      /* Работа активной ветви */
        if(MyP == p)
         {t3=MPI_Wtime();
          for(m = p-1; m >= 0; m--)
           MPI_Send(&MA[k][M], 1, MPI_DOUBLE, 1,MPI_COMM_WORLD);
        dt3=MPI_Wtime()-t3;
             for(i = k-1; i >= 0; i--)
           MA[i][M] -= MA[k][M]*MA[i][N*p+k];
         }
      /* Работа ветвей с номерами MyP < p */
        else
         { if(MyP < p)
            {t4=MPI_Wtime();
              MPI_Recv(&R, 1, MPI_DOUBLE, p, 1,MPI_COMM_WORLD, &stat);
           dt4=MPI_Wtime()-t4;
                for(i = N-1; i >= 0; i--)
                MA[i][M] -= R*MA[i][N*p+k];
            }
         }
       }             /* конец цикла по  k */
   }                 /* конец цикла по  p */
/* Все ветви засекают время и печатают его */
  dt = MPI_Wtime()-t0;t0=dt1+dt2+dt3+dt4;
  printf("MyP = %d Time = %13.4e los time=%13.4e\n", MyP, dt,t0);
```

```
   /* Все ветви печатают, для контроля, свои первые четыре значения корня */
printf("MyP = %d %13.4e %13.4e %13.4e %13.4e\n“,
MyP,MA[0][M],MA[1][M],MA[2][M],MA[3][M]);
 dt=(double)N*MyP;
   /* Печать точных первых четырех значений корня */
printf("toch =  %13.4e %13.4e %13.4e %13.4e\n",-(dt+1)/2,-(dt+2)/2,-(dt+3)/2,-
(dt+4)/2);
/* Все ветви завершают выполнение */
MPI_Barrier(MPI_COMM_WORLD);
   MPI_Finalize();
   return(0);
  }
```

Приложение 5. Решение СЛАУ методом Гаусса на OpenMP.

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

#define M 1600

/* Описываем массивы для расширенной матрицы - MA и вектор V - решение
системы. */

double MA[M][M + 1], V[M];

int main(int argc, char **argv)
{
int i, j, p;
double a;
struct timeval tv_start, tv_end;

gettimeofday(&tv_start, NULL);

/* Заполняется матрица системы  */
   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;
/* Задается решение системы. */
   for (j = 0; j < M; j++)
            V[j] = -(double)(j + 1) / 2.;

/* Вычисляется вектор правой части, который записывается в последний столбец
      расширенной матрицы */
   for(i = 0; i < M; i++) {
            MA[i][M] = 0.0;
for(j = 0; j < M; j++)
            MA[i][M] += MA[i][j] * V[j];
}

/* Прямой ход */
for(p = 0; p < M; p++) {
            a = MA[p][p];
            for(i = p; i <= M; i++)
                        MA[p][i] = MA[p][i] / a;

            /* Цикл k - цикл по строкам. (Все ветви "крутят" этот цикл). */
#pragma omp parallel for private (i, j, a)
       for(j = p + 1; j < M; j++) {
            a = MA[j][p];
            for(i = p; i <= M; i++)
                                    MA[j][i] = MA[j][i] - a * MA[p][i];
            }
}

/* Обратный ход */
   for(p = M - 1; p >= 0; p--) {
#pragma omp parallel for private (i, j)
            for(j = p - 1; j >=0; j--) {
                        for(i = M; i > j; i--)
                                    MA[j][i] = MA[j][i]- MA[j][p] * MA[p][i];
            }
}
gettimeofday(&tv_end, NULL);

/* Вычисленные значения решения расположены в последнем столбце
      расширенной матрицы MA */
```

```
   /* Выдаются для контроля первые четыре значения корня */
   printf("Вычисленные значения: %13.4e %13.4e %13.4e %13.4e \n", MA[0][M],
MA[1][M], MA[2][M], MA[3][M]);
   printf("Точные значения: %13.4e  %13.4e  %13.4e %13.4e\n", V[0], V[1], V[2],
V[3]);

printf("Время счета = %.6f sec.\n", (tv_end.tv_sec * 1E6 + tv_end.tv_usec -
         tv_start.tv_sec * 1E6 + tv_start.tv_usec) / 1E6);
   return 0;
}
```