МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Параллельное программирование

Методические указания
к лабораторной работе №3

САМАРА 2017

Составитель: доц., Козлова Е.С.

УДК 519.681

Методические указания к лабораторным работам

Самарский государственный аэрокосмический университет имени академика С.П.КОРОЛЁВА (национальный исследовательский университет)

Составитель: Е.С. Козлова

Самара, 2017. 12 с.

Методические указания предназначены для бакалавров направления 010400.62 «Прикладная математика и информатика»

Печатается по решению редакционно-издательского совета Самарского университета

Рецензент:

1 Стандарт OpenMP

1.1 Особенности выполнения

OpenMP используется модель параллельного выполнения “ветвление-слияние” (*fork-join*). Программа начинается выполнением одной нити, называемой начальной (*initial*) нитью. Начальная нить выполняется последовательно. Когда нить достигает директивы *parallel* она создает команду нитей, состоящую из неё самой и нуля или более дополнительных нитей, и становится хозяйкой (*master*) созданной команды. Все члены команды исполняют код структурной области, связанной с директивой *parallel* (параллельной области). В конце параллельной области размещается неявный барьер. Только нить-хозяйка продолжает выполнение после завершения параллельной области.

Число нитей в команде, выполняющихся параллельно, можно контролировать несколькими способами. Один из них – использование переменной окружения OMP_NUM_THREADS. Другой способ – вызов процедуры *omp_set_num_threads()*. Еще один способ – использование выражения *num_threads* в сочетании с директивой *parallel*.

1.2 Директива for

В C/C++ директивы OpenMP определяются конструкциями *#pragma*, предусматривающимися стандартами C и C++, и используемых для задания дополнительных указаний компилятору. Использование специальной ключевой директивы “omp” указывает на то, что команды относятся к OpenMP и для того, чтобы исключить случайные совпадения имён директив OpenMP с другими именами. Таким образом директивы *#pragma* для работы с OpenMP имеют следующий формат:

#pragma omp директива<> опция[[[,] опция]...]

Наиболее часто используемой директивой является директива для распараллеливания циклов, которая может совмещаться с директивой объявления параллельной области.

Директива *for* служит для распределения итераций цикла между различными нитями можно использовать директиву *for*:

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

Эта директива относится к идущему следом за данной директивой блоку, включающему оператор for. Если в параллельной области встретился оператор цикла без директивы, то, согласно общему правилу, он будет выполнен всеми нитями текущей группы, то есть каждая нить выполнит все итерации данного цикла.

Возможные опции:

private;

firstprivate;

lastprivate (список) – переменным, перечисленным в списке, присваивается результат с последнего витка цикла;

reduction;

schedule(type[, chunk]) – опция задаёт, каким образом итерации цикла распределяются между нитями;

collapse(n) – опция указывает, что n последовательных тесновложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между нитями; если опция *collapse* не задана, то директива относится только к одному непосредственно следующему за ней циклу;

ordered – опция, говорящая о том, что в цикле могут встречаться директивы ordered; в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле;

nowait – в конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция *nowait* позволяет нитям, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными.

На вид параллельных циклов накладываются достаточно жёсткие ограничения. В частности, предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит. Нельзя использовать побочный выход из параллельного цикла. Размер блока итераций, указанный в опции schedule, не должен изменяться в рамках цикла.

Эти требования введены для того, чтобы OpenMP мог при входе в цикл точно определить число итераций. Если директива параллельного выполнения стоит перед гнездом циклов, завершающихся одним оператором, то директива действует только на самый внешний цикл. Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции *lastprivate*.

Подробнее разберемся с планировщиком для цикла, формат работы которого задается опцией *schedule*.

Опция *schedule* управляет распределением работы между нитями в конструкции распределения работы цикла:

schedule mun([, chunk])

Опция задаёт, каким образом итерации цикла распределяются между нитями. Задается вид алгоритма планирования и, если необходимо, числовой параметр алгоритма (обычно размер блока пространства итераций).

В опции *schedule* параметр *type* задаёт следующий тип распределения итераций:

static – блочно-циклическое распределение итераций цикла; размер блока – *chunk*. Первый блок из *chunk* итераций выполняет нулевая нить, второй блок – следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение *chunk* не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между нитями.

dynamic – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает *chunk* итераций (по умолчанию *chunk*=1), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из *chunk* итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.

guided – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины *chunk* (по умолчанию *chunk*=1) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения *chunk*.

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

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

2 Стандарт MPI

2.1 Модель программирования MPI

Под параллельной программой в рамках MPI понимается множество одновременно выполняемых процессов. Все процессы порождаются один раз, образуя параллельную часть программы. Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода (модель SPMP - Single Programm Multiple Processes). Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI-программ и в ходе вычислений меняться не может. Все процессы программы последовательно перенумерованы от 0 до p-1, где p есть общее количество процессов. Номер процесса именуется рангом процесса. При этом для того, чтобы избежать идентичности вычислений на разных процессорах, можно, во-первых, подставлять разные данные для программы на разных процессорах, а во-вторых, с помощью средств для идентификации процесса, на котором выполняется программа, организовать различия в вычислениях в зависимости от используемого программой процессора. Это позволяет загружать ту или иную подзадачу в зависимости от “номера” процессора.

Основу MPI составляют операции передачи сообщений. Сообщение – это набор данных некоторого типа. Каждое сообщение имеет атрибуты, такие как: номер процесса – отправителя, номер процесса – получателя, идентификатор сообщения и другие. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом. Среди предусмотренных в составе MPI функций различаются парные (point-to-point) операции между двумя процессами и коллективные (collective) коммуникационные действия для одновременного взаимодействия нескольких процессов.

2.2 Коллективные операции MPI

Одной из наиболее привлекательных сторон MPI является наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий. Главное отличие коллективных операций от операций типа "точка-точка" состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Зачастую исходные данные в программе формируются на исходном нулевом процессе. Однако для обработки данные необходимо предоставить всем процессам. Использование обычной широковещательной рассылки *Bcast* бывает неоправданным в случае реализации некоторых алгоритмов, в связи с тем, что процессу не требуется весь набор исходных данных, а только их часть. Кроме того, декомпозиция данных позволяет

экономить затраты на хранение данных и их пересылку по сети. Для реализации подобного сценария используется семейство коллективных операций *Scatter.*

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):

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

где *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, *comm* – коммуникатор. Процесс *root* использует оба буфера (посылки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы с коммуникатором *comm* 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*. Следует отметить, что значение *sendcount* в вызове из процесса *root* – это число посылаемых каждому процессу элементов, а не общее их количество.

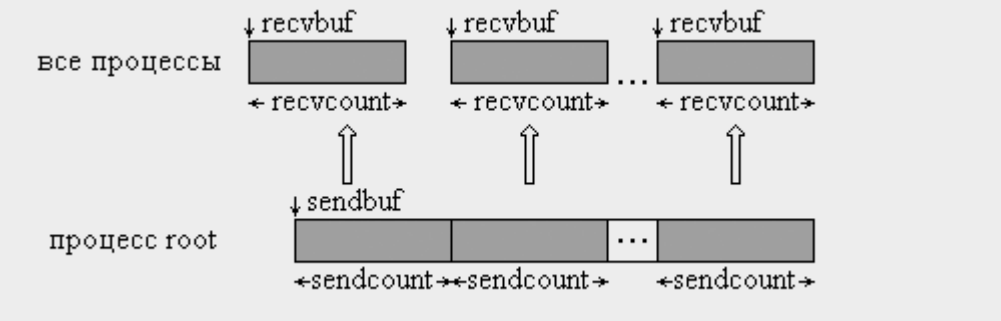

Рис. 1. Графическая интерпретация операции Scatter

Функция *MPI_Scatterv* является векторным вариантом функции MPI_*Scatter*, позволяющий посылать каждому процессу различное количество элементов:

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

где *sendbuf* – адрес начала буфера посылки (используется только в процессе-отправителе *root*), *sendcount*s – целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, посылаемых каждому процессу, *displs* – целочисленный массив (размер равен числу процессов в группе), *i*-ое значение определяет смещение относительно начала *sendbuf* для данных, посылаемых процессу *i*, *sendtype* – тип посылаемых элементов, *recvbuf* – адрес начала буфера приема, *recvcount*– число получаемых элементов, *recvtype* – тип получаемых элементов, *root* – номер процесса-отправителя, *comm* – коммуникатор. Начало расположения элементов блока, посылаемого *i*-му процессу, задается в массиве смещений *displs*, а число посылаемых элементов – в массиве *sendcount*s.

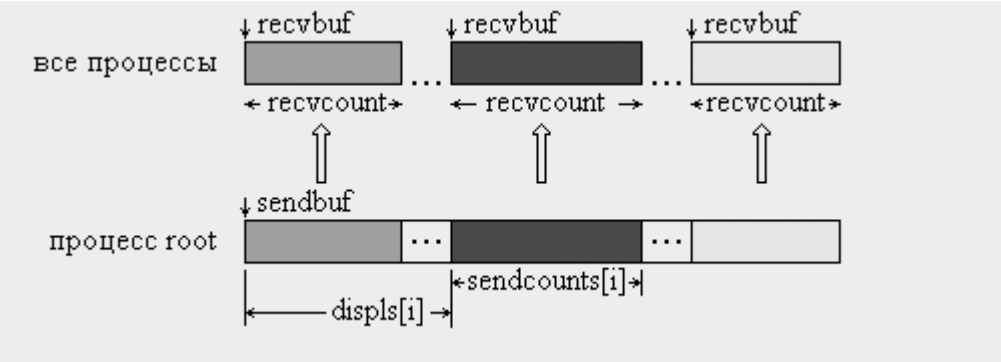

Рис. 2. Графическая интерпретация операции Scatterv

Логично, что при использованиии функций декомпозиции входных данных нам потребуются функции по сборке результата, которыми являются функции семейства *Gather*.

Функция *MPI_Gather* производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером *root*:

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

где *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, *comm* – коммуникатор.

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.

Тип посылаемых элементов *sendtype* должен совпадать с типом *recvtype* получаемых элементов, а число *sendcount* должно равняться числу *recvcount*. То есть, *recvcount* в вызове из процесса *root* – это число собираемых от каждого процесса элементов, а не общее количество собранных элементов.

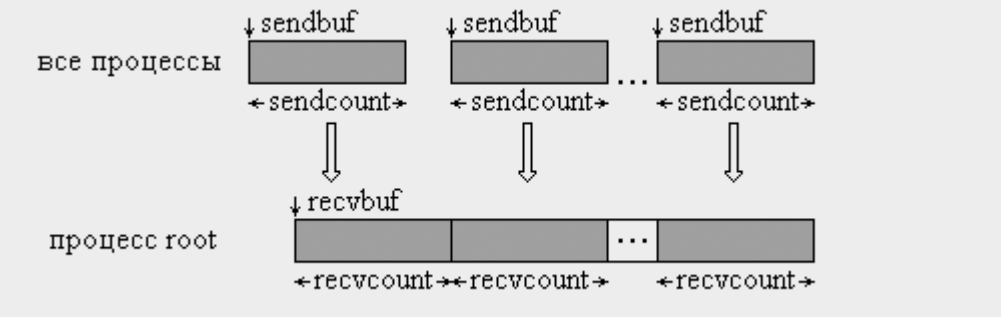

Рис. 3. Графическая интерпретация операции Gather

Функция *MPI_Gatherv* позволяет собирать блоки с разным числом элементов от каждого процесса:

int MPI_Gatherv(void sendbuf, int sendcount, MPI_Datatype sendtype, void* rbuf, int * recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)*

где *sendbuf* – адрес начала буфера передачи, *sendcount* – число посылаемых элементов, *sendtype* – тип посылаемых элементов, r*buf* – адрес начала буфера приема, *recvcount*s – целочисленный массив (размер равен числу процессов в группе), *i*-й элемент которого определяет число элементов, которое должно быть получено от процесса *i*, *displs* – целочисленный массив (размер равен числу процессов в группе), *i*-ое значение определяет смещение *i*-го блока данных относительно начала r*buf*, *recvtype* – тип получаемых элементов, *root* – номер процесса-получателя, *comm* – коммуникатор.

Количество элементов, принимаемых от каждого процесса, задается индивидуально с помощью массива *recvcount*s. Эта функция обеспечивает также большую гибкость при размещении данных в процессе-получателе, благодаря введению в качестве параметра массива смещений *displs*. Сообщения помещаются в буфер приема процесса *root* в соответствии с номерами посылающих процессов, а именно, данные, посланные процессом *i*, размещаются в адресном пространстве процесса *root*, начиная с адреса *rbuf* + *displs*[*i*]. представлена на рис 4.

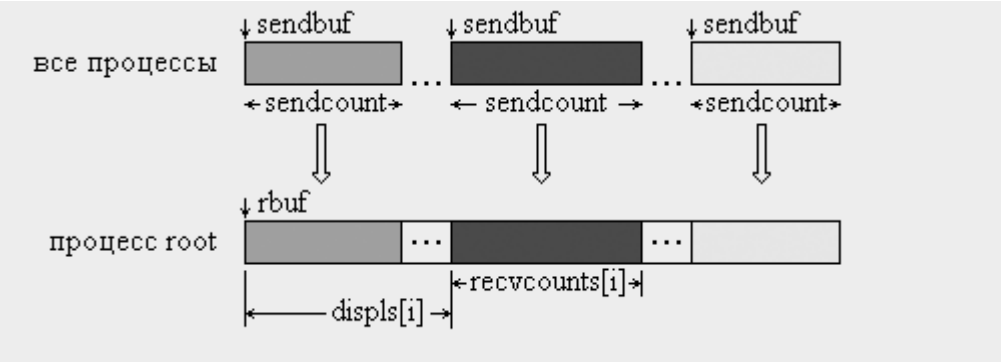

Рис. 4. Графическая интерпретация операции Gatherv

Функция *MPI_Allgather* производит сборку блоков данных, посылаемых всеми процессами группы, и рассылает всем процессам группы:

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

где *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, *comm* – коммуникатор.

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.

Функция *MPI_Allgatherv* позволяет собирать блоки с разным числом элементов от каждого процесса, является аналогом функции *MPI_Gatherv*, но сборка выполняется всеми процессами группы (поэтому в списке параметров отсутствует параметр *root*):

int *MPI_Allgatherv*(void* *sendbuf*, int *sendcount*, MPI_Data*type sendtype*, void* r*buf*, int **recvcount*s, int **displs*, MPI_Data*type recvtype*, MPI_*Comm comm*)

где *sendbuf* – адрес начала буфера передачи, *sendcount* – число посылаемых элементов, *sendtype* – тип посылаемых элементов, r*buf* – адрес начала буфера приема, *recvcount*s – целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, которое должно быть получено от каждого процесса, *displs* –

целочисленный массив (размер равен числу процессов в группе), *i*-ое значение определяет смещение относительно начала r*buf* *i*-го блока данных, *recvtype* – тип получаемых элементов, *comm* – коммуникатор.

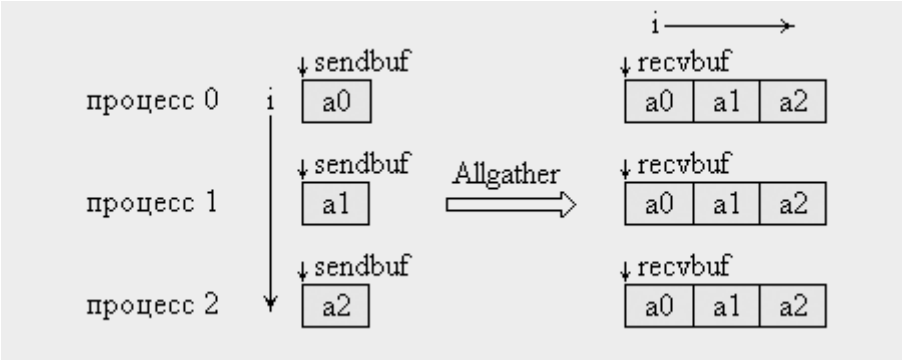

Рис. 5. Графическая интерпретация операции Allgater: ось Y - процессы группы, ось X - блоки данных

3 Задание на лабораторную работу

Лабораторная работа 3

Произвести запуск программ для поиска суммы векторов, которые используют технологии MPI и OpenMP, на различном количестве процессоров (потоков).

На основе технологии OpenMP реализуйте три варианта работы с использованием различных настроек планировщика задач

На основе технологии MPI реализуйте два варианта программы:

1) количество элементов массива кратно количеству процессов;
2) количество элементов массива не кратно количеству процессов.

В программе с использованием технологией MPI для используйте схему распределенного хранения данных: каждый поток хранит только ту часть исходных данных, которая необходима ему для расчетов. На 0-м процессе также хранятся полные копии обрабатываемых и результирующих векторов.

В ходе анализа работы программы оцените время ее выполнения на различном количестве исполняющих нитей (процессов). Оцените влияние различных функций и директив(опций) на скорость работы приложений.

Ниже приведен шаблоны программ для запуска.

Пример 1. Код OpenMP

```
#define CHUNK 100
#define NMAX 500000
#include<omp.h>
main ()
{
    int i;
    double a[NMAX], sum[NMAX], b[NMAX],s=0;
    int chunk,n;
    chunk=CHUNK;
    n=NMAX;
    omp_set_num_threads(16);
    //<инициализация данных>
    double st_time, end_time;
    st_time = omp_get_wtime();
    #pragma omp parallel for shared (a,b,sum,n) private (i)
     //суммирование векторов
    end_time = omp_get_wtime ();
    end_time = end_time - st_time;

    printf("\nTIME OF WORK IS %f ", s);

     return 0;
}
```

Пример 2. Код MPI

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

```
#include "mpi.h"

#define NMAX 5000
main(int argc, char* argv[])
{
     int * a, *b , *c ,s=0, j;
     int *a_loc, *b_loc, *c_loc;
     int ProcRank, ProcNum, N=NMAX,i;
     MPI_Status Status;
     double st_time, end_time;
     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
     MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
     int count=N/ProcNum;
     // инициализация 0-ым процессом исходных данных
     a_loc = (int *) malloc(count*sizeof(int));
     c_loc = (int *) malloc(count*sizeof(int));
     b_loc = (int *) malloc(count*sizeof(int));
     int time=0;

     st_time = MPI_Wtime();
     // рассылка 0-ым процессом по всем остальным
     // получение локальной суммы векторов
     // сборка результата 0-ым процессом
     end_time = MPI_Wtime();

     time = end_time - st_time;

     if ( ProcRank == 0 )
     {
          printf("\nTIME OF WORK IS %f ", end_time);
     }
     MPI_Finalize();
     return 0;
}
```