Федеральное государственное бюджетное образовательное учреждение высшего образования
"Уфимский государственный авиационный технический университет"

Кафедра Высокопроизводительных вычислительных технологий и систем

Дисциплина: Алгоритмы Параллельного Програмирования

Отчет по лабораторной работе № 2

Тема: «Параллельное вычисление произведения двух матриц»

Группа ПМ-353	Фамилия И.О.	Подпись	Дата	Оценка
Студент	Шамаев И.Р.			
Принял	Юлдашев А.В.			

Уфа 2022

Цель: научиться использовать принцип геометрической декомпозиции в параллельных алгоритмах и создавать параллельные программы для систем с распределенной памятью с использованием коллективных функций MPI на примере вычисления произведения двух матриц

Теоретический материал.

MPI (Message Passing Interface) – это стандартизованная библиотека функций, призванная обеспечить совместную работу параллельных процессов путем организации передачи сообщений между ними.

Интерфейс MPI:

- Модель программирования MPI.
- Базовые функции MPI.
- Функции парного взаимодействия.
- Функции коллективного взаимодействия.
- Функция определения времени
- Пользовательские типы данных.
- Управление областью взаимодействия и группой процессов.

Базовые понятия MPI

- MPI предназначен для написания программ для MIMD архитектур
- Каждая программа представляет собой совокупность одновременно работающих процессов, которые могут обмениваться сообщениями.
- Все процессы объединяются в группы.
- Обмен сообщениями возможен между процессами одной группы, которой поставлена в соответствие своя область связи.
- Идентификатор области связи называется коммуникатором.
- При запуске программы все доступные ей процессы объединяются в начальную группу с общей областью связи, имеющей коммуникатор MPI_COMM_WORLD.

Базовые операции стандарта MPI

- Операции межпроцессорного взаимодействия типа
- «точка-точка».
- Операции коллективного взаимодействия.
- Операции над группами процессов.
- Операции с областями коммуникации.
- Операции с топологией процессов.
- Функции ввода/вывода (появились в MPI-2.0)

Особенности реализации MPI для C/C++

1. Первая строка программы

#include “mpi.h”

2. В MPI принят ANSI C стандарт.
3. Нумерация массивов начинается с 0.
4. Массивы хранятся по строкам.

5. Логические переменные являются переменными типа integer со значением 0 в случае false и любым не нулевым значением, обозначающем true.

Базовые функции MPI

1. int ***MPI_Init***(int *argc, char **argv[]); – инициализация параллельной части программы.

Возвращает предопределенные константы

MPI_SUCCESS - возвращается в случае успешного выполнения,

MPI_ERR_ARG - ошибка неправильного задания аргумента,

MPI_ERR_INTERN - внутренняя ошибка (нехватка памяти),

MPI_ERR_UNKNOWN - неизвестная ошибка.

2. int ***MPI_Finalize***(void); завершение параллельной части программы.

3. int ***MPI_Comm_size***(MPI_Comm comm, int* size); определение числа процессов size в коммуникационной группе с коммуникатором comm.

4. int ***MPI_Comm_rank***(MPI_comm comm, int* rank); определение номера rank вызвавшего ее процесса, входящего в коммуникационную группу с коммуникатором comm

5. int ***MPI_Send***(void* sbuf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) – передача сообщений от одного процесса к другому

sbuf – адрес в памяти, начиная с которого размещаются передаваемые данные;

count – количество передаваемых элементов;

datatype – тип передаваемых элементов;

dest – номер процесса-получателя сообщения;

tag – метка передаваемого сообщения;

comm – коммуникатор.

6. int ***MPI_Recv***(void* rbuf, int count, MPI_Datatype datatype, int source, int tag, MPI_comm comm, MPI_Status *status) – прием сообщений от одного процесса к другому

Входные параметры:

count – количество получаемых элементов;

datatype – тип получаемых элементов;

source – номер процесса-отправителя сообщения;

tag – метка принимаемого сообщения;

comm – коммуникатор.

Выходные параметры:

rbuf – адрес в памяти, начиная с которого размещаются принимаемые данные;

status – структура, содержащая информацию о принятом сообщении.

Структура status имеет три поля.

Status.MPI_SOURCE - номер процесса-отправителя;

Status.MPI_TAG - метка принимаемого сообщения;

Status.MPI_ERROR - код завершения приема сообщения.

Функция совмещенного приема/передачи.

int ***MPI_Sendrecv***(void* sbuf, int scount, MPI_Datatype sdatatype, int dest, int stag, void* rbuf, int rcount, MPI_Datatype rdatatype, int source, int rtag, MPI_comm comm, MPI_Status *status);

Входные параметры:

sbuf – адрес в памяти, начиная с которого размещаются передаваемые данные;

scount – количество передаваемых элементов;

sdatatype – тип отправляемых элементов;

dest – номер процесса-получателя сообщения;

stag – метка отправляемого сообщения;

rcount – количество получаемых элементов;

rdatatype – тип получаемых элементов;

source – номер процесса-отправителя сообщения;

rtag – метка принимаемого сообщения;

comm – коммуникатор.

Выходные параметры:

rbuf – адрес в памяти, начиная с которого размещаются принимаемые данные;

status – структура, содержащая информацию о принятом сообщении.

Функции коллективного взаимодействия

1. int ***MPI_Bcast***(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

Функция предназначена для рассылки данных, хранящихся на одном процессе, всем остальным процессам группы.

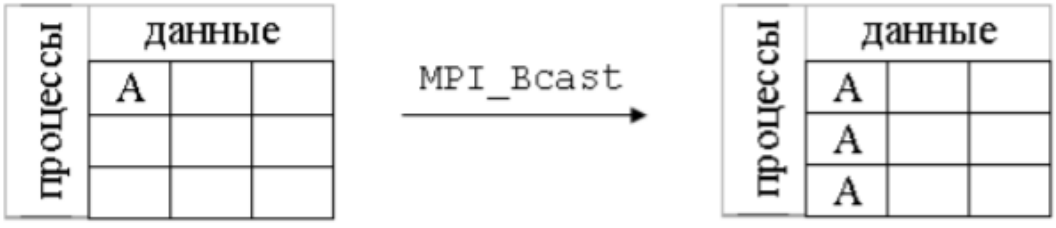

Рисунок 1. MPI_Bcast

Входные параметры:

buf – адрес в памяти, начиная с которого размещаются передаваемые данные;

count – количество рассылаемых элементов;

datatype – тип отправляемых данных;

root – номер процесса-отправителя сообщения;

comm – коммуникатор.

2. int ***MPI_Gather***(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm);

Функция предназначена для сбора данных со всех процессов на одном (так называемый, "совок“).

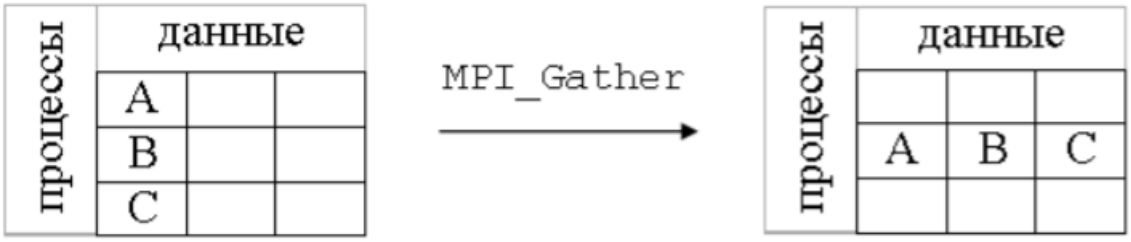

Рисунок 2. MPI_Gather

Входные параметры:

sbuf – адрес в памяти на каждом процессе, начиная с которого размещаются отправляемые данные;

scount – количество элементов, отправляемых с каждого процесса;

stype – тип отправляемых данных;

rcount – количество принимаемых элементов от каждого процесса;

rtype – тип принимаемых данных;

root – номер процесса, на котором осуществляется сборка сообщений;

comm – коммуникатор.

Выходные параметры:

rbuf – адрес в памяти на процессе с номером root, начиная с которого размещаются принимаемые сообщения.

3. Вариант функции с варьируемым кол-вом собираемых элементов:

int ***MPI_Gatherv***(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int *rcounts, int *displs, MPI_Datatype rtype, int root, MPI_Comm comm);

Характерные отличия:

rcounts – массив длин принимаемых от процессов сообщений;

displs – массив позиций в приемном буфере, по которым размещаются принимаемые сообщения.

4. int ***MPI_Scatter***(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm);

Функция предназначена для рассылки данных с одного процесса всем остальным процессам (так называемый, “разбрызгиватель”).

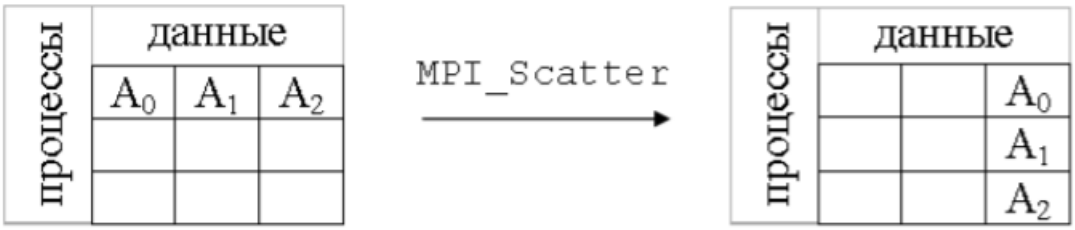

Рисунок 3. MPI_Scatter

Входные параметры:

sbuf – адрес в памяти на процессе-отправителе сообщения, начиная с которого размещаются отправляемые данные;

scount – количество элементов, отправляемых каждому процессу;

stype – тип отправляемых данных;

rcount – количество элементов, принимаемых каждым процессом (длина принимаемого сообщения);

rtype – тип принимаемых данных;

root – номер процесса-отправителя сообщения;

comm – коммуникатор.

Выходные параметры:

rbuf – адрес в памяти на каждом процессе, начиная с которого размещаются принимаемые сообщения.

5. Вариант функции с варьируемым кол-вом рассылаемых элементов:

int ***MPI_Scatterv***(void *sbuf, int *scounts, int *displs, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm);

Характерные отличия:

scounts – массив, содержащий количество элементов в каждой части, на которые разбивается сообщение;

displs – массив позиций, определяющий начальные положения каждой части сообщения.

6. int ***MPI_Reduce***(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

Функция проделывает операцию op над данными, хранящимися в sbuf в каждом процессе группы, результат которой записывается в rbuf в процесс с номером root.

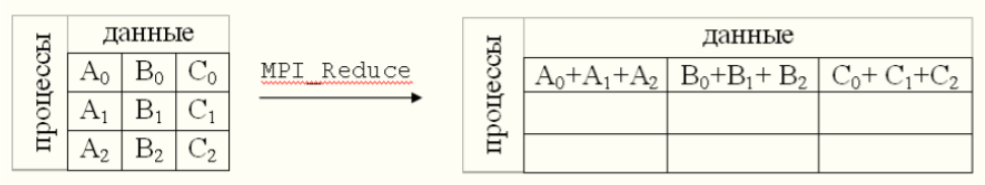

Рисунок 4. MPI_Reduce

Входные параметры:

sbuf – адрес в памяти на каждом процессе, по которому хранятся исходные данные для распределенной операции;

count – количество элементов в sbuf;

datatype – тип данных, над которыми производится распределенная операция;

root – номер процесса, на котором осуществляется размещение результата выполнения операции;

op – название распределенной операции;

comm – коммуникатор.

Выходные параметры:

rbuf – адрес в памяти, по которому размещаются результаты выполнения операции.

12 предопределенных операций:

MPI_MAX – поиск поэлементного максимума;

MPI_MIN – поиск поэлементного минимума;

MPI_SUM – вычисление суммы векторов;

MPI_PROD – вычисление поэлементного произведения векторов;

MPI_LAND – логическое “И”;

MPI_LOR – логическое “ИЛИ”;

MPI_LXOR – логическое исключающее “ИЛИ”;

MPI_BAND – бинарное “И”;

MPI_BOR – бинарное “ИЛИ”;

MPI_BXOR – бинарное исключающее ИЛИ;

MPI_MAXLOC – поиск индексированного максимума;

MPI_MINLOC – поиск индексированного минимума.

Практическая часть

В данной работе были протестированы две версии программы, обычное перемножение матриц (колонки 1050 и 2500) и перемножение матриц с предварительным транспонированием второй из них (колонки 1050 T и 2500 T, время транспонирования в расчет не бралось)

Таблица 1. Результаты замеров времени

Кол-во процессов	1050	2500	1050 T	2500 T
1	81,16	809,613	29,3	416,6
2	42,15	420,062	17,7	257,7
4	24,28	273,933	15,1	229,4
8	17,5	229,023	15,9	214,1
16	8,84	114,319	8	107
32	4,72	57,46	4,1	54,7
64	2,31	29,656	2,1	27,6
96	1,63	19,709	1,5	18,5

Вычислим ускорение и эффективность по формулам $S_p = \frac{T_1}{T_p}$, $E_p = \frac{S_p}{p}$:

Таблица 2. Вычисление ускорения для разного числа процессов

Кол-во процессов	1050	2500	1050 T	2500 T
1	1	1	1	1
2	1,925504152	1,927365484	1,655367	1,616608
4	3,342668863	2,95551467	1,940397	1,816042
8	4,637714286	3,535072897	1,842767	1,94582
16	9,180995475	7,082051103	3,6625	3,893458
32	17,19491525	14,09002785	7,146341	7,616088
64	35,13419913	27,30014162	13,95238	15,0942
96	49,79141104	41,07833984	19,53333	22,51892

Таблица 3. Вычисление эффективности для разного числа процессов

Кол-во процессов	1050	2500	1050 T	2500 T
1	1	1	1	1
2	0,962752076	0,963682742	0,82768362	0,808304
4	0,835667216	0,738878667	0,48509934	0,45401
8	0,579714286	0,441884112	0,23034591	0,243227
16	0,573812217	0,442628194	0,22890625	0,243341
32	0,537341102	0,44031337	0,22332317	0,238003
64	0,548971861	0,426564713	0,21800595	0,235847
96	0,518660532	0,427899373	0,20347222	0,234572

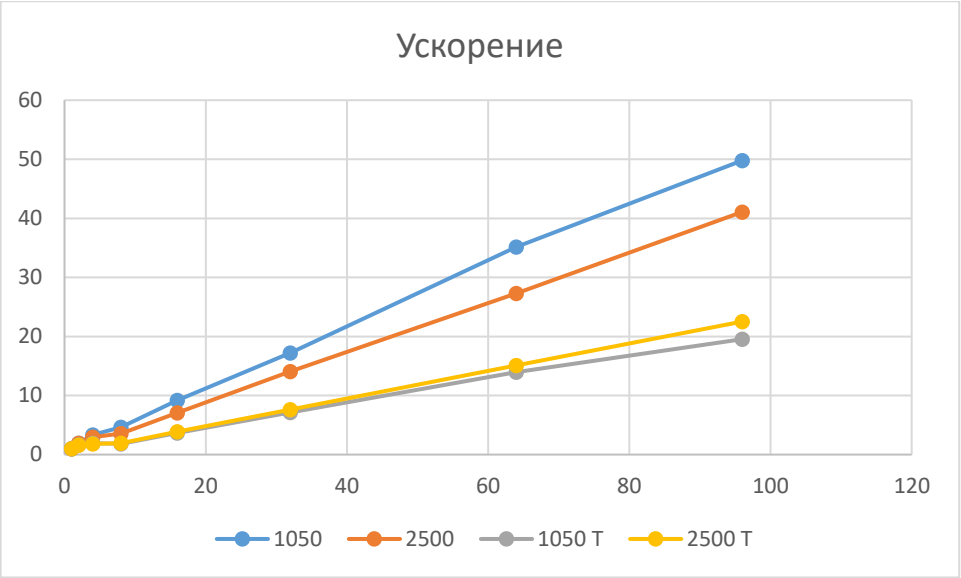

Рисунок 3. График зависимости ускорения от числа процессов

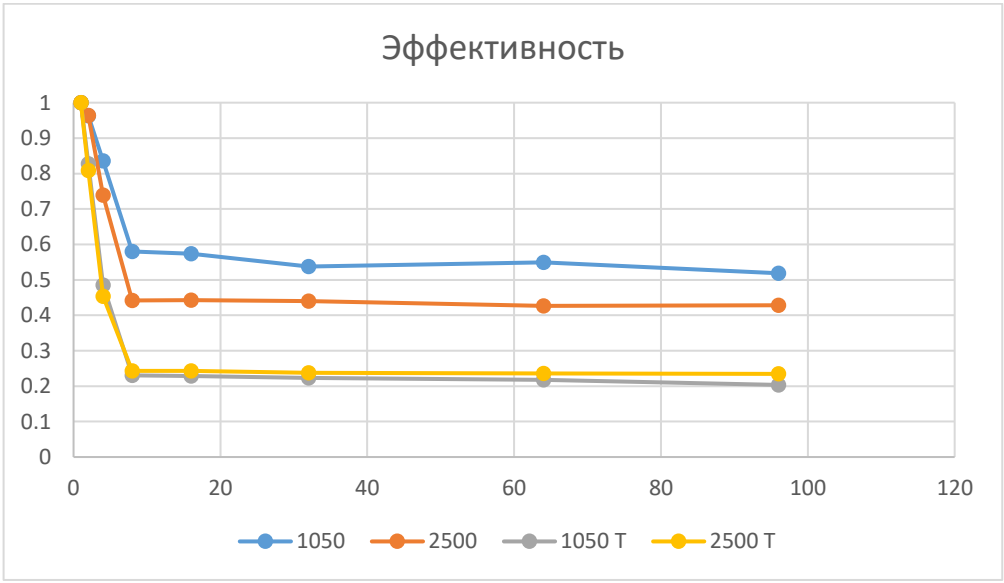

Рисунок 4. График зависимости эффективности от числа процессов

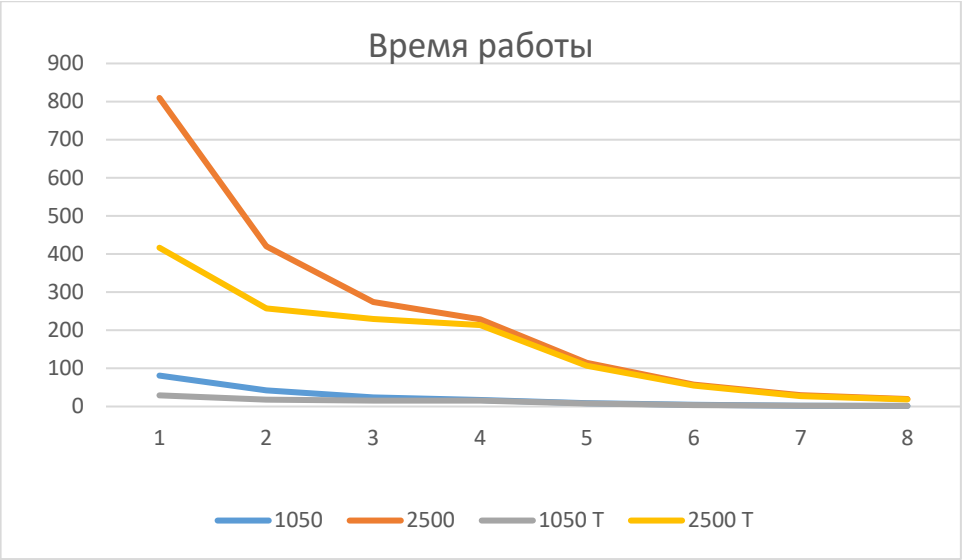

Рисунок 5. График времени работы в зависимости от числа процессов

Вывод: в ходе лабораторной работы на примере задачи параллельного умножения матриц научились реализовывать простейшие параллельные вычислительные алгоритмы и проводить анализ их эффективности.

Код программы.

```
#include "mpi.h"
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

using namespace std;

int main(int argc, char *argv[])
{
	srand((unsigned int)1);

	int MyID, NumProc, ierror;
	ierror = MPI_Init(&argc, &argv);//Функция инициализации

	if (ierror != MPI_SUCCESS)
		cout << "MPI Init error" << endl;

	MPI_Comm_size(MPI_COMM_WORLD, &NumProc); //Функция определения количества процессов в коммуникационной группе с коммуникатором comm
	MPI_Comm_rank(MPI_COMM_WORLD, &MyID); //Функция, возвращающая номер rank вызвавшего ее процесса, входящего в коммуникационную группу с коммуникатором comm


	if (argc != 2)
	{
		cout << "[!] 'N' needed" << endl;
		return -1;
	}

	long long L;
	L = atoi(argv[1]); преобразует строку string в целое значение типа int

	long long N = 10 * L, p = N / NumProc, o = N % NumProc;

	double *A = (double*)calloc(N*L, sizeof(double)),
*B = (double*)calloc(L*L, sizeof(double)),
*C = (double*)calloc(N*L, sizeof(double)),
*Ap = (double*)calloc((p + 1)*L, sizeof(double)),
*Cp = (double*)calloc((p + 1)*L, sizeof(double)),
*normid = (double*)calloc(1, sizeof(double)),
*Norma = (double*)calloc(1, sizeof(double));

	int *count = (int*)calloc(NumProc, sizeof(int)), *displs =
(int*)calloc(NumProc, sizeof(int)),
*end=(int*)calloc(NumProc, sizeof(int)),
ccount;

	double tstart, tfinish;
```

```
	if (MyID == 0)//матрицы  заполняются случайными элементами вещественного типа;
	{
		for (int i = 0; i < N*L; i++) A[i] = -0.5 + (double)rand() / RAND_MAX;;

		for (int i = 0; i < L*L; i++) B[i] = -0.5 + (double)rand() / RAND_MAX;;

		tstart = MPI_Wtime();
	}

	MPI_Barrier(MPI_COMM_WORLD); //Останавливает выполнение вызвавшей ее задачи до тех пор, пока не будет вызвана изо всех остальных задач, подсоединенных к указываемому коммуникатору

	if (MyID == 0)
	{
		for (int i = 0; i < NumProc; i++)
		{
			if (o > i)
			{
				count[i] = (p + 1)*L;// массив, содержащий количество элементов в каждой части, на которые разбивается сообщение;
				displs[i] = i * (p + 1)*L;
			}
			else
			{
				count[i] = p * L;// массив, содержащий количество элементов в каждой части, на которые разбивается сообщение;
				displs[i] = o * (p + 1)*L + (i - o)*p*L;// массив позиций, определяющий начальные положения каждой части сообщения.

		}
	}
	if (MyID < o)
	{
		ccount = (p + 1)*L;
		end[MyID] = p + 1;
	}
	else
	{
		end[MyID] = p;
		ccount = (p)*L;
	}

	MPI_Scatterv(A, count, displs, MPI_DOUBLE, Ap, ccount, MPI_DOUBLE, 0, MPI_COMM_WORLD);// Функция предназначена для рассылки данных с одного процесса всем остальным процессам
	MPI_Bcast(B, L*L, MPI_DOUBLE, 0, MPI_COMM_WORLD);//Функция предназначена для рассылки данных, хранящихся на одном процессе, всем остальным процессам группы.

	MPI_Barrier(MPI_COMM_WORLD);

	for (int i = 0; i < end[MyID]; i++)
	{
```

```
			for (int j = 0; j < L; j++)
			{
				Cp[L * (i)+j] = .0;
				for (int k = 0; k < L; k++)
				{
					Cp[L * (i)+j] += Ap[L * (i)+k] * B[L * (k)+j];
				}

				normid[0] += Cp[L * (i)+j] * Cp[L * (i)+j];
			}

		}

		MPI_Gatherv(Cp, ccount, MPI_DOUBLE, C, count, displs, MPI_DOUBLE, 0,
MPI_COMM_WORLD);// Функция предназначена для сбора данных со всех процессов
на одном. Вариант функции с варьируемым кол-вом собираемых элементов
		MPI_Reduce(normid, Norma, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);//Функция предназначена для рассылки данных с одного
процесса всем остальным процессам
		if (MyID == 0)
		{
			tfinish = MPI_Wtime() - tstart;
			cout << "> Norm = " << Norma[0] << endl;
			cout << "> Time = " << tfinish << endl;
		}
		free(A); free(C); free(Ap); free(B); free(Cp); free(normid); free(Norma);
free(count); free(displs); free(end);
		MPI_Finalize();//функция завершения работы с MPI
}
```