МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №1
по курсу «Параллельная обработка данных»

Message Passing Interface (MPI)

Выполнил: Симонов С.Я.
Группа: 8О-406Б-18
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Цель работы: Знакомство с технологией MPI. Реализация метода Якоби.
Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода.

Вариант 2. обмен граничными слоями через bsend, контроль сходимости allgather

Программное и аппаратное обеспечение
GPU:

Compute capability : 6.1
Name : GeForce GTX 1050
Total Global Memory : 2096103424
Shared memory per block : 49152
Registers per block : 65536
Max threads per block : (1024, 1024, 64)
Max block : (2147483647, 65535, 65535)
Total constant memory : 65536
Multiprocessors count : 5

Сведения о системе:

Операционная система: Ubuntu 14.04 LTS 64-x битная
Рабочая среда: nano
Компилятор: mpic++

Метод решения

За основу был взят код из лекции, решающий двумерную задачу. Основной частью реализации стало добавление третьего измерения, доведение итерационной формулы до конца и добавление следующей формулы:

$$max_{i,j,k}\left|u_{i,j,k}^{(n+1)} - u_{i,j,k}^{(n)}\right| < \epsilon$$

Функция обмена граничными слоями попалась такая же как и в лекции, так что ее менять не пришлось.Сходимость процесса отслеживалась при помощи функции MPI_Allgather, которая складывает вычисленные значения в буффер. Причем если хотя бы одно значение больше нашей заданной точности, то итерационный процесс продолжается.

Описание программы

Всю программу можно условно разделить на три части:

1) Обмен граничными слоями между процессами.
2) Обновление значений.
3) Вычисление выше приведенной формулы.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <string>
#include <fstream>
#include <cmath>
#include "mpi.h"

#define _i(i, j, k) (((k) + 1) * (y + 2) * (x + 2) + ((j) + 1) * (x + 2) + (i) + 1)
#define _ibx(id) (((id) % (dim1 * dim2) ) % dim1)
#define _iby(id) (((id) % (dim1 * dim2) ) / dim1)
#define _ibz(id) ((id) / (dim1 * dim2) )
#define _ib(i, j, k) ((k) * (dim1 * dim2) + (j) * dim1 + (i))

// double abs(double a, double b) {
//	if (a - b < 0) {
//		return b - a;
//	}
//	return a - b;
// }

int main(int argc, char *argv[]) {
	int numproc, id, i, j, k, dim1, dim2, dim3, x, y, z, ib, jb, kb, max;
	std::string fi;
	double eps, l_x, l_y, l_z, u_down, u_up, u_left, u_right, u_front, u_back, u_0, hx, hy,
hz, check = 0.0;
	double *data, *temp, *next, *buff;
	MPI_Status status;
	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &numproc);
	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Barrier(MPI_COMM_WORLD);
	if (id == 0) {
		std::cin >> dim1 >> dim2 >> dim3 >> x >> y >> z;
		// std::cerr << dim1 << " " << dim2 << " " << dim3 << " " << x << " " << y << " "
<< z << " ";
		std::cin >> fi;
		// std::cerr << fi << " ";
		std::cin >> eps;
		// std::cerr << eps << " ";
		std::cin >> l_x >> l_y >> l_z;
		// std::cerr << l_x << " " << l_y << " " << l_z << " ";
		std::cin >> u_down >> u_up >> u_left >> u_right >> u_front >> u_back >>
u_0;
		// std::cerr << u_down << " " << u_up << " " << u_left << " " << u_right << " "
<< u_front << " " << u_back << " " << u_0 << " ";
```

```
	}
	MPI_Bcast(&dim1, 1, MPI_INT, 0, MPI_COMM_WORLD); // считанные переменные пересылаем всем остальным процессам
	MPI_Bcast(&dim2, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&dim3, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&y, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&z, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&l_x, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&l_y, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&l_z, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_down, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_up, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_left, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_right, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_front, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_back, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_0, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	ib = _ibx(id);
	jb = _iby(id);
	kb = _ibz(id);
	hx = l_x / (double)(dim1 * x);
	hy = l_y / (double)(dim2 * y);
	hz = l_z / (double)(dim3 * z);
	data = (double*)malloc(sizeof(double) * (x + 2) * (y + 2) * (z + 2)); // n + 2 -- для того чтобы организовать фиктивные ячейки
	next = (double*)malloc(sizeof(double) * (x + 2) * (y + 2) * (z + 2));
	// if (x >= y && x >= z) {
	//	max = x;
	// } else if (y >= x && y >= z) {
	//	max = y;
	// } else if (z >= y && z >= x) {
	//	max = z;
	// }
	max = std::max(std::max(x ,y), z);
	buff = (double*)malloc(sizeof(double) * (max) * (max));
	double* check_mpi = (double*)malloc(sizeof(double) * dim1 * dim2 * dim3);
	int buffer_size;
	buffer_size = 6 * (sizeof(double) * (max) * (max) + MPI_BSEND_OVERHEAD);
	double *buffer = (double *)malloc(buffer_size);
	MPI_Buffer_attach(buffer, buffer_size);
	for (i = 0; i < x; ++i) { // инициализация блока
		for (j = 0; j < y; ++j) {
			for (k = 0; k < z; ++k) {
				data[_i(i, j, k)] = u_0;
			}
		}
```

```
	}
	// for (int it = 0; it < 500; it++) {
	for (;;) {
		MPI_Barrier(MPI_COMM_WORLD); // синхронизация всех потоков
		if (ib + 1 < dim1) { // если наш поток не крайний
			for (j = 0; j < y; ++j) { //
				for (k = 0; k < z; ++k) {
					buff[k + j * z] = data[_i(x - 1, j, k)]; // тогда копируем
одну границу нашей области в буффер
				}
			}
			MPI_Bsend(buff, y * z, MPI_DOUBLE, _ib(ib + 1, jb, kb), id,
MPI_COMM_WORLD); // и отправляем враво наши данные, процессу с номером ib + 1
		}
		if (jb + 1 < dim2) { // отправна наверх наших данных
			for (i = 0; i < x; ++i) { //
				for (k = 0; k < z; ++k) {
					buff[k + i * z] = data[_i(i, y - 1, k)]; //
				}
			}
			MPI_Bsend(buff, x * z, MPI_DOUBLE, _ib(ib, jb + 1, kb), id,
MPI_COMM_WORLD); //
		}
		if (kb + 1 < dim3) { // отправна наверх наших данных
			for (i = 0; i < x; ++i) { //
				for (j = 0; j < y; ++j) {
					buff[j + i * y] = data[_i(i, j, z - 1)]; //
				}
			}
			MPI_Bsend(buff, x * y, MPI_DOUBLE, _ib(ib, jb, kb + 1), id,
MPI_COMM_WORLD); //
		}
		if (ib > 0) {
			for (j = 0; j < y; ++j) { //
				for (k = 0; k < z; ++k) {
					buff[k + j * z] = data[_i(0, j, k)];
				}
			}
			MPI_Bsend(buff, y * z, MPI_DOUBLE, _ib(ib - 1, jb, kb), id,
MPI_COMM_WORLD);
		}
		if (jb > 0) {
			for (i = 0; i < x; ++i) { //
				for (k = 0; k < z; ++k) {
					buff[k + i * z] = data[_i(i, 0, k)]; //
				}
			}
```

```
                MPI_Bsend(buff, x * z, MPI_DOUBLE, _ib(ib, jb - 1, kb), id,
MPI_COMM_WORLD); //
        }
        if (kb > 0) {
                for (i = 0; i < x; ++i) { //
                        for (j = 0; j < y; ++j) {
                                buff[j + i * y] = data[_i(i, j, 0)]; //
                        }
                }
                MPI_Bsend(buff, x * y, MPI_DOUBLE, _ib(ib, jb, kb - 1), id,
MPI_COMM_WORLD); //
        }
        MPI_Barrier(MPI_COMM_WORLD);
        // поскольку Bsend не подвисающая операция мы сразу можем
принимать данные

        if (ib > 0) { // если мы можем принимать данные
                MPI_Recv(buff, y * z, MPI_DOUBLE, _ib(ib - 1, jb, kb), _ib(ib - 1, jb,
kb), MPI_COMM_WORLD, &status);// то мы принимаем данные
                for (j = 0; j < y; ++j) {
                        for (k = 0; k < z; ++k) {
                                data[_i(-1, j, k)] = buff[k + j * z]; // копируем данные
                        }
                }
        } else { // если граничные элементы то задаем как границу
                for (j = 0; j < y; ++j) {
                        for (k = 0; k < z; ++k) {
                                data[_i(-1, j, k)] = u_left;
                        }
                }
        }
        if (jb > 0) { // если мы можем принимать данные
                MPI_Recv(buff, x * z, MPI_DOUBLE, _ib(ib, jb - 1, kb), _ib(ib, jb - 1,
kb), MPI_COMM_WORLD, &status);// то мы принимаем данные
                for (i = 0; i < x; ++i) {
                        for (k = 0; k < z; ++k) {
                                data[_i(i, -1, k)] = buff[k + i * z]; //
                        }
                }
        } else { // если граничные элементы то задаем как границу
                for (i = 0; i < x; ++i) {
                        for (k = 0; k < z; ++k) {
                                data[_i(i, -1, k)] = u_front;
                        }
                }
        }
        if (kb > 0) { // если мы можем принимать данные
```

```
                        MPI_Recv(buff, x * y, MPI_DOUBLE, _ib(ib, jb, kb - 1), _ib(ib, jb, kb -
1), MPI_COMM_WORLD, &status);// то мы принимаем данные
                        for (i = 0; i < x; ++i) {
                                for (j = 0; j < y; ++j) {
                                        data[_i(i, j, -1)] = buff[j + i * y]; //
                                }
                        }
                } else { // если граничные элементы то задаем как границу
                        for (i = 0; i < x; ++i) {
                                for (j = 0; j < y; ++j) {
                                        data[_i(i, j, -1)] = u_down;
                                }
                        }
                }
                if (ib + 1 < dim1) { // если мы можем принимать данные
                        MPI_Recv(buff, y * z, MPI_DOUBLE, _ib(ib + 1, jb, kb), _ib(ib + 1, jb,
kb), MPI_COMM_WORLD, &status);// то мы принимаем данные
                        for (j = 0; j < y; ++j) {
                                for (k = 0; k < z; ++k) {
                                        data[_i(x, j, k)] = buff[k + j * z]; //
                                }
                        }
                } else { // если граничные элементы то задаем как границу
                        for (j = 0; j < y; ++j) {
                                for (k = 0; k < z; ++k) {
                                        data[_i(x, j, k)] = u_right;
                                }
                        }
                }
                if (jb + 1 < dim2) { // если мы можем принимать данные
                        MPI_Recv(buff, x * z, MPI_DOUBLE, _ib(ib, jb + 1, kb), _ib(ib, jb + 1,
kb), MPI_COMM_WORLD, &status);// то мы принимаем данные
                        for (i = 0; i < x; ++i) {
                                for (k = 0; k < z; ++k) {
                                        data[_i(i, y, k)] = buff[k + i * z]; // копируем данные
                                }
                        }
                } else { // если граничные элементы то задаем как границу
                        for (i = 0; i < x; ++i) {
                                for (k = 0; k < z; ++k) {
                                        data[_i(i, y, k)] = u_back;
                                }
                        }
                }
                if (kb + 1 < dim3) { // если мы можем принимать данные
                        MPI_Recv(buff, x * y, MPI_DOUBLE, _ib(ib, jb, kb + 1), _ib(ib, jb, kb +
1), MPI_COMM_WORLD, &status);// то мы принимаем данные
                        for (i = 0; i < x; ++i) {
```

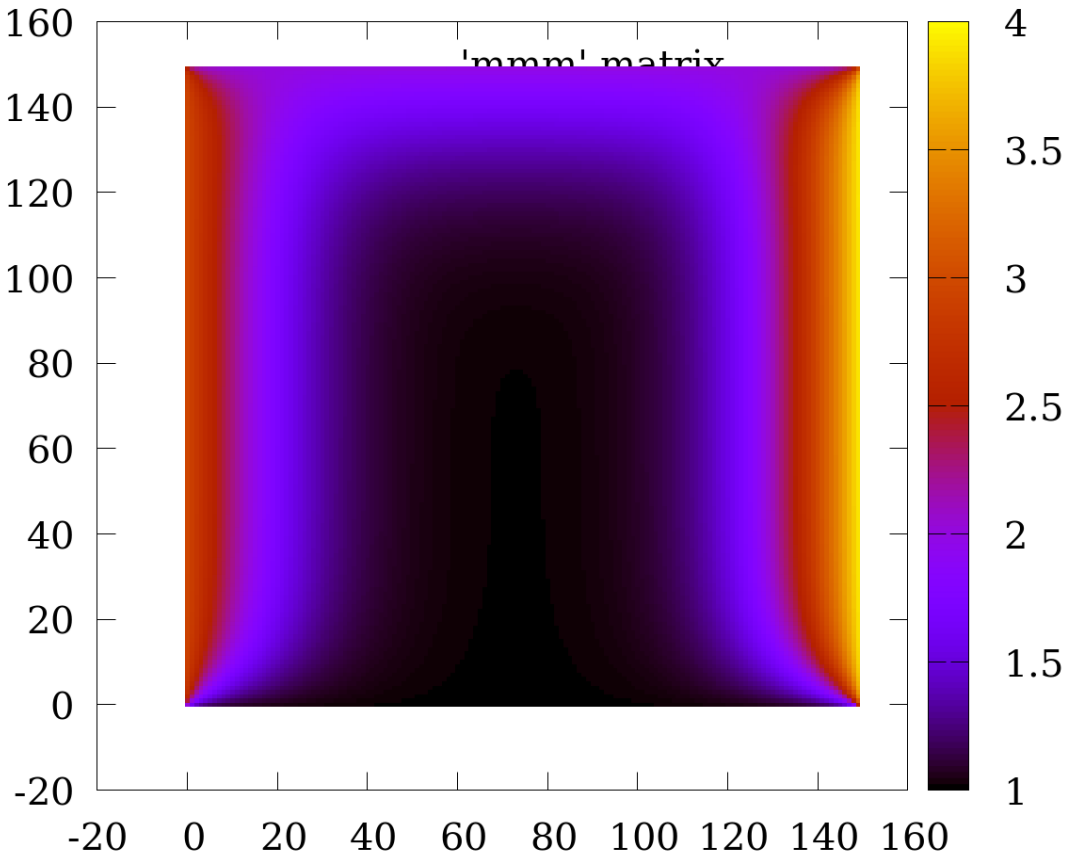
```
                        for (j = 0; j < y; ++j) {
                                data[_i(i, j, z)] = buff[j + i * y]; // копируем данные
                        }
                }
        } else { // если граничные элементы то задаем как границу
                for (i = 0; i < x; ++i) {
                        for (j = 0; j < y; ++j) {
                                data[_i(i, j, z)] = u_up;
                        }
                }
        }
        //дпльше оргнизуем основной вычислительный цикл
        MPI_Barrier(MPI_COMM_WORLD); // выполняем синзронизацию всех
процессов
        check = 0.0;
        for (i = 0; i < x; ++i) {
                for (j = 0; j < y; ++j) {
                        for (k = 0; k < z; ++k) {
                                next[_i(i, j, k)] = 0.5 * ((data[_i(i + 1, j, k)] + data[_i(i - 1,
j, k)]) / (hx * hx) + (data[_i(i, j + 1, k)] + data[_i(i, j - 1, k)]) / (hy * hy) + (data[_i(i, j, k + 1)] +
data[_i(i, j, k - 1)]) / (hz * hz)) / (1.0 / (hx * hx) + 1.0 / (hy * hy) + 1.0 / (hz * hz));
                                if (std::abs(next[_i(i, j, k)] - data[_i(i, j, k)]) > check) {
                                        check = std::abs(next[_i(i, j, k)] - data[_i(i, j, k)]);
                                }
                        }
                }
        }
        temp = next;
        next = data;
        data = temp;
        // if (id == 0) {
        //      fprintf(stderr, "it = %d\n", it);
        //      fflush(stderr);
        // }
        // double a[1];
        // a[0] = check;
        check_mpi[id] = check;
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Allgather(&check, 1, MPI_DOUBLE, check_mpi, 1, MPI_DOUBLE,
MPI_COMM_WORLD);
        check = 0.0;
        for (i = 0; i < dim1 * dim2 * dim3; ++i) {
                if (check_mpi[i] > check) {
                        check = check_mpi[i];
                }
        }
        if (check < eps) {
                break;
```

```
                }
        }
        MPI_Barrier(MPI_COMM_WORLD);
        if (id != 0) {
                for (k = 0; k < z; ++k) {
                        for (j = 0; j < y; ++j) {
                                for (i = 0; i < x; ++i)
                                        buff[i] = data[_i(i, j, k)];
                                MPI_Send(buff, x, MPI_DOUBLE, 0, id,
MPI_COMM_WORLD);
                        }
                }
        } else {
                std::ofstream file(fi);
                file.precision(6);
                file.setf(std::ios::scientific);
                for (kb = 0; kb < dim3; ++kb)
                        for (k = 0; k < z; ++k)
                                for (jb = 0; jb < dim2; ++jb)
                                        for (j = 0; j < y; ++j)
                                                for (ib = 0; ib < dim1; ++ib) {
                                                        if (_ib(ib, jb, kb) == 0)
                                                                for (i = 0; i < x; ++i)
                                                                        buff[i] = data[_i(i, j, k)];
                                                        else
                                                                MPI_Recv(buff, x,
MPI_DOUBLE, _ib(ib, jb, kb), _ib(ib, jb, kb), MPI_COMM_WORLD, &status);
                                                        for (i = 0; i < x; ++i)
                                                                file << buff[i] << " ";
                                                        if (ib + 1 == dim1) {
                                                                file << "\n";
                                                                if (j + 1 == y)
                                                                        file << "\n";
                                                        } else {
                                                                file << " ";
                                                        }
                                                }
                file.close();
        }
        MPI_Finalize();
        return 0;
}
```

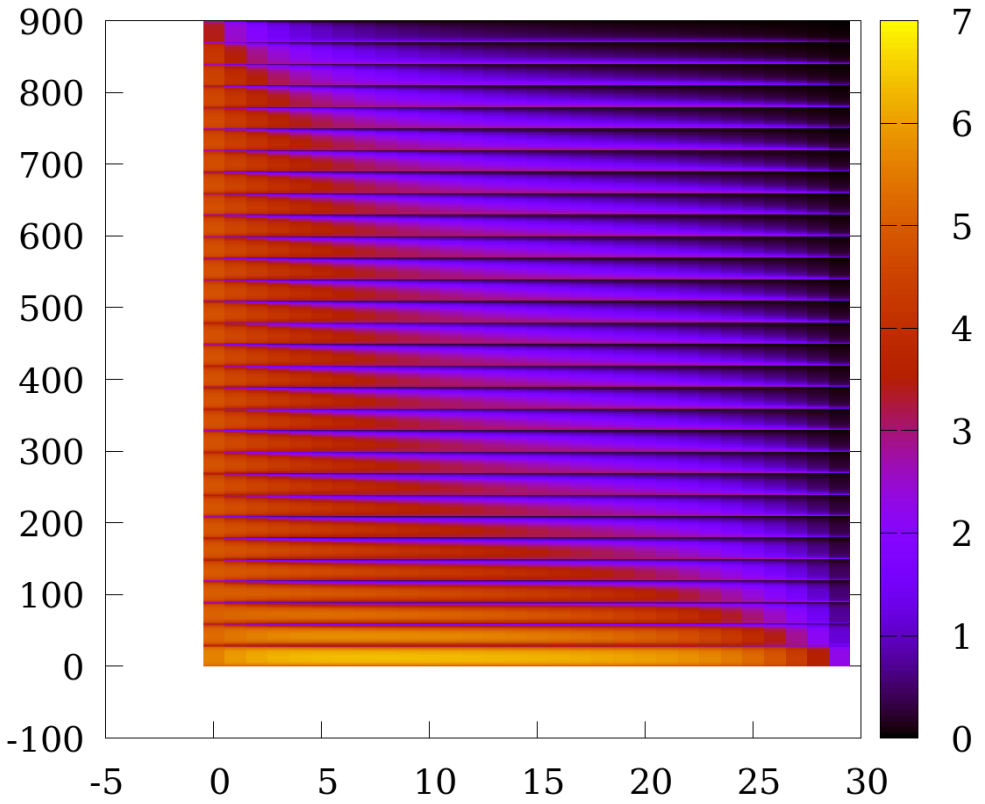
Результаты

Размер: 30*30*30

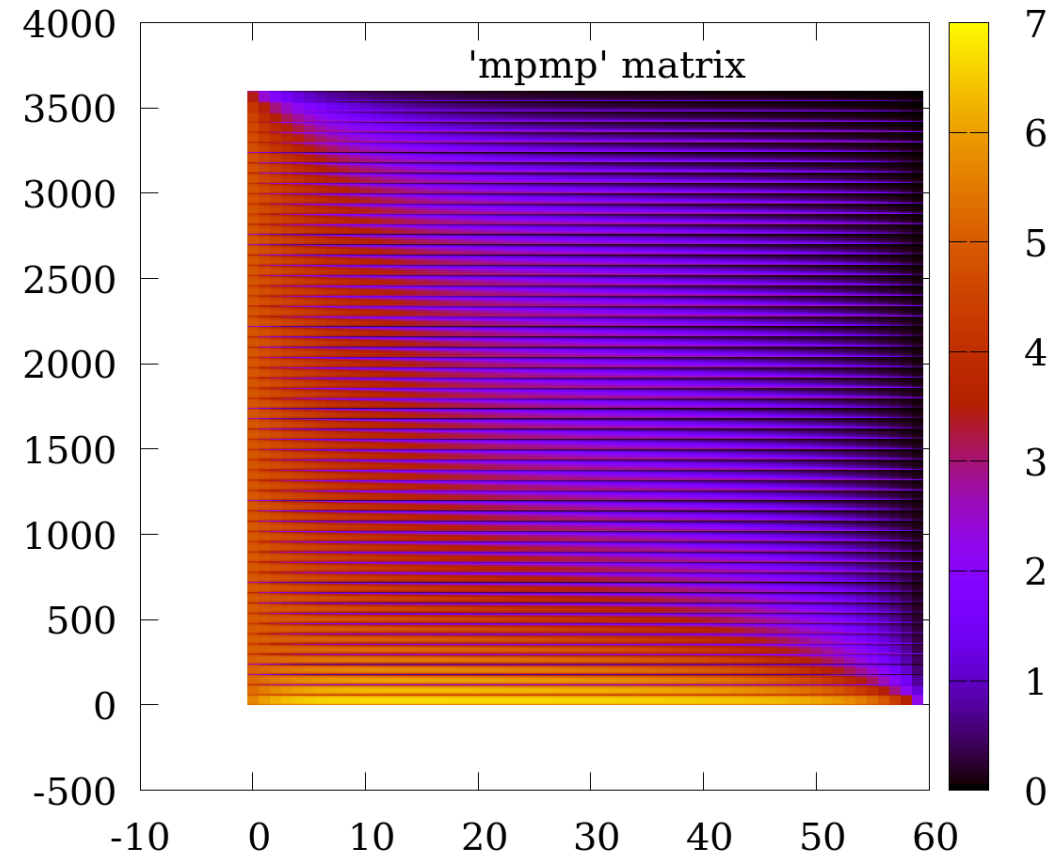
Время работы	dim1	dim2	dim3
4.63398sec	1	1	1
31.1725sec	2	2	2
100.352sec	2	3	2



30 * 30 * 30



30 * 30 * 30



Выводы

В ходе выполнения данной лабораторной работы я познакомился с технологией передачи информации между процессорами выполняющими одну и туже задачу - MPI. Сама лабораторная работа показалась гораздо проще, чем сортировки и метод Гаусса. Все стало предельно понятно и ясно после повторного просмотра лекции. По большей части время было потрачено на устранение ошибки допущенной по невнимательности. Ошибка была выявлена после совета Александра Юрьевича, по правильному тестированию программы.