МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа №3
по курсу «Параллельная обработка данных»

Технология MPI и технология OpenMP

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

Москва, 2021

Условие

Цель работы: Совместное использование технологии MPI и технологии OpenMP. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода.

Вариант 2. Распараллеливание в общем виде с разделением работы между нитями вручную (“в стиле CUDA”).

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: nvcc

Метод решения

Основная суть решения по отношению к предыдущим лабораторным работам не изменилась. Основными изменениями стало передача данных не при помощи буфера, а при помощи производных типов данных освоенных в лабораторной №2 по ПОД. Производные типы данных были реализованы в соответствии с вариантом задания по второй лабораторной работе (MPI_Type_create_subarray).
Помимо всего прочего основной вычислительный цикл был распараллелен при помощи технологии OPENMP. Запись в файл была взята из предыдущей лабораторной работы.

Описание программы

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

```
#include "mpi.h"
#include <omp.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))

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;
	char fi[128];
	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;
	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);
	MPI_Bcast(fi, 128, MPI_CHAR, 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 органиховать фиктивные ячейки
	next = (double*)malloc(sizeof(double) * (x + 2) * (y + 2) * (z + 2));
	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;
			}
		}
	}

	int sizes[3] = {x + 2, y + 2, z + 2};
	int subsizes[3], starts[3];

	MPI_Datatype bsend_yz_left, bsend_yz_right, recv_yz_left, recv_yz_right;
	subsizes[2] = z; subsizes[1] = y; subsizes[0] = 1;
	starts[2] = 1; starts[1] = 1; starts[0] = 1;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &bsend_yz_left);
	MPI_Type_commit(&bsend_yz_left);
	starts[2] = 1; starts[1] = 1; starts[0] = x;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &bsend_yz_right);
	MPI_Type_commit(&bsend_yz_right);
	starts[2] = 1; starts[1] = 1; starts[0] = 0;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &recv_yz_left);
	MPI_Type_commit(&recv_yz_left);
```

```
	starts[2] = 1; starts[1] = 1; starts[0] = x + 1;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &recv_yz_right);
	MPI_Type_commit(&recv_yz_right);
	MPI_Datatype bsend_xz_front, bsend_xz_back, recv_xz_front, recv_xz_back;
	subsizes[2] = z; subsizes[1] = 1; subsizes[0] = x;
	starts[2] = 1; starts[1] = 1; starts[0] = 1;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &bsend_xz_front);
	MPI_Type_commit(&bsend_xz_front);
	starts[2] = 1; starts[1] = y; starts[0] = 1;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &bsend_xz_back);
	MPI_Type_commit(&bsend_xz_back);
	starts[2] = 1; starts[1] = 0; starts[0] = 1;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &recv_xz_front);
	MPI_Type_commit(&recv_xz_front);
	starts[2] = 1; starts[1] = y + 1; starts[0] = 1;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &recv_xz_back);
	MPI_Type_commit(&recv_xz_back);
	MPI_Datatype bsend_xy_up, bsend_xy_down, recv_xy_up, recv_xy_down;
	subsizes[2] = 1; subsizes[1] = y; subsizes[0] = x;
	starts[2] = 1; starts[1] = 1; starts[0] = 1;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &bsend_xy_up);
	MPI_Type_commit(&bsend_xy_up);
	starts[2] = z; starts[1] = 1; starts[0] = 1;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &bsend_xy_down);
	MPI_Type_commit(&bsend_xy_down);
	starts[2] = 0; starts[1] = 1; starts[0] = 1;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &recv_xy_up);
	MPI_Type_commit(&recv_xy_up);
	starts[2] = z + 1; starts[1] = 1; starts[0] = 1;
	MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_FORTRAN,
MPI_DOUBLE, &recv_xy_down);
	MPI_Type_commit(&recv_xy_down);

	for (;;) {
		MPI_Barrier(MPI_COMM_WORLD); // синхронизация всех потоков
		if (ib + 1 < dim1) { // если наш поток не крайний
			MPI_Bsend(data, 1, bsend_yz_right, _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
```

```
                        MPI_Bsend(data, 1, bsend_xz_back, _ib(ib, jb + 1, kb), id,
MPI_COMM_WORLD); //
                }
                if (kb + 1 < dim3) { // отправна наверх наших данных
                        MPI_Bsend(data, 1, bsend_xy_down, _ib(ib, jb, kb + 1), id,
MPI_COMM_WORLD); //
                }
                if (ib > 0) {
                        MPI_Bsend(data, 1, bsend_yz_left, _ib(ib - 1, jb, kb), id,
MPI_COMM_WORLD);
                }
                if (jb > 0) {
                        MPI_Bsend(data, 1, bsend_xz_front, _ib(ib, jb - 1, kb), id,
MPI_COMM_WORLD); //
                }
                if (kb > 0) {
                        MPI_Bsend(data, 1, bsend_xy_up, _ib(ib, jb, kb - 1), id,
MPI_COMM_WORLD); //
                }
                // поскольку Bsend не подвисающая операция мы сразу можем
принимать данные

                if (ib > 0) { // если мы можем принимать данные
                        MPI_Recv(data, 1, recv_yz_left, _ib(ib - 1, jb, kb), _ib(ib - 1, jb, kb),
MPI_COMM_WORLD, &status);// то мы принимаем 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(data, 1, recv_xz_front, _ib(ib, jb - 1, kb), _ib(ib, jb - 1, kb),
MPI_COMM_WORLD, &status);// то мы принимаем данные
                } 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(data, 1, recv_xy_up, _ib(ib, jb, kb - 1), _ib(ib, jb, kb - 1),
MPI_COMM_WORLD, &status);// то мы принимаем 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, -1)] = u_down;
                                }
                        }
                }
                if (ib + 1 < dim1) { // если мы можем принимать данные
                        MPI_Recv(data, 1, recv_yz_right, _ib(ib + 1, jb, kb), _ib(ib + 1, jb, kb),
MPI_COMM_WORLD, &status);// то мы принимаем данные
                } 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(data, 1, recv_xz_back, _ib(ib, jb + 1, kb), _ib(ib, jb + 1, kb),
MPI_COMM_WORLD, &status);// то мы принимаем 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(data, 1, recv_xy_down, _ib(ib, jb, kb + 1), _ib(ib, jb, kb + 1),
MPI_COMM_WORLD, &status);// то мы принимаем 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;
                #pragma omp parallel private(i, j, k)
                {
                        int off = omp_get_num_threads();
                        int id = omp_get_thread_num();
                        for (i = id; i < x; i += off) {
                                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));
```

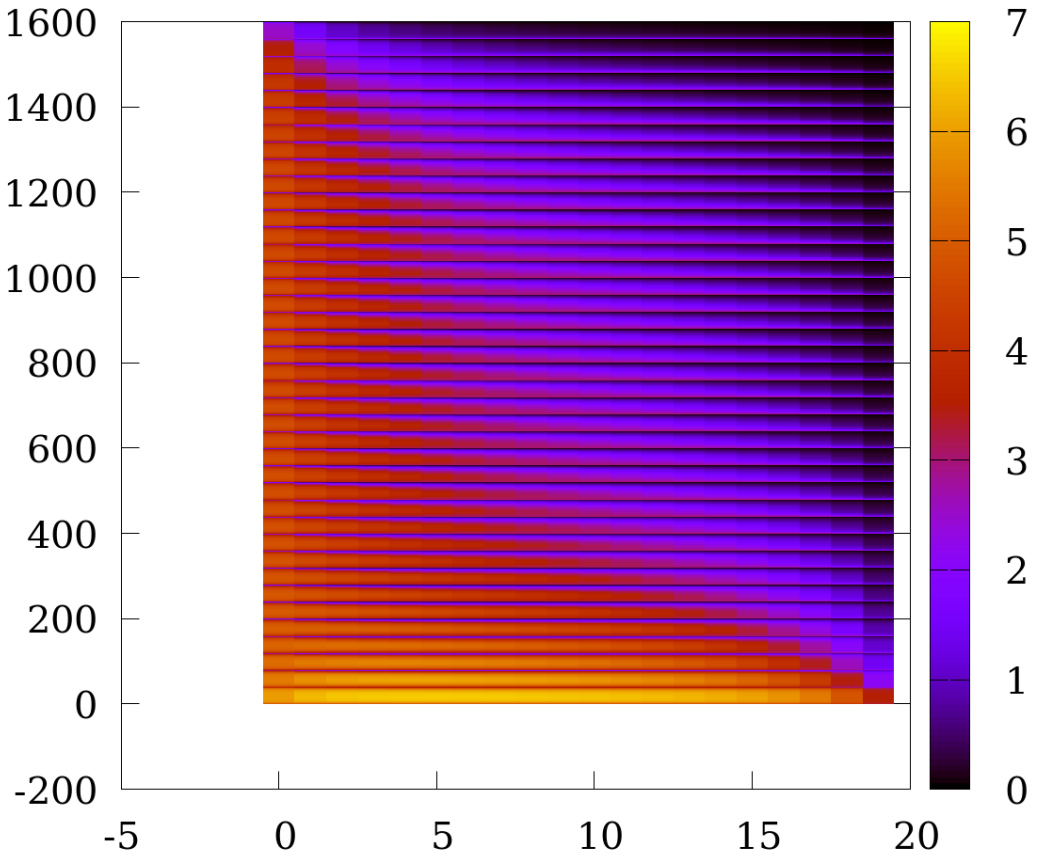
```
                                                if (std::abs(next[_i(i, j, k)] - data[_i(i, j, k)]) >
check) {
                                                        check = std::abs(next[_i(i, j, k)] -
data[_i(i, j, k)]);
                                                }
                                        }
                                }
                        }
                }
                temp = next;
                next = data;
                data = temp;
                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);
        int n_size = 30;
        char* bf = (char*)malloc(sizeof(char) * x * y * z * n_size);
        memset(bf, ' ', sizeof(char) * x * y * z * n_size);
        for (k = 0; k < z; k++) {
                for (j = 0; j < y; j++) {
                        for (i = 0; i < x; i++) {
                                sprintf(bf + ((k * x * y) + j * x + i) * n_size, "%.6e", data[_i(i, j,
k)]);
                        }
                        if (ib + 1 == dim1) {
                                bf[(k * x * y + j * x + i) * n_size - 1] = '\n';
                        }
                }
        }
        for (i = 0; i < x * y * z * n_size; i++)  {
                if (bf[i] == '\0') {
                        bf[i] = ' ';
                }
        }
        MPI_File fl;
        MPI_Datatype filetype;
        MPI_Type_contiguous(n_size, MPI_CHAR, &filetype);
```

```
        MPI_Type_commit(&filetype);
        MPI_Datatype sub, big;
        int sub_bigsizes[3] = {x, y, z};
        int sub_subsizes[3] = {x, y, z};
        int sub_starts[3] = {0, 0, 0};
        int big_bigsizes[3] = {x * dim1, y * dim2, z * dim3};
        int big_subsizes[3] = {x, y, z};
        int big_starts[3] = {ib * x, jb * y, kb * z};
        MPI_Type_create_subarray(3, sub_bigsizes, sub_subsizes, sub_starts,
MPI_ORDER_FORTRAN, filetype, &sub);
        MPI_Type_create_subarray(3, big_bigsizes, big_subsizes, big_starts,
MPI_ORDER_FORTRAN, filetype, &big);
        MPI_Type_commit(&sub);
        MPI_Type_commit(&big);
        MPI_File_delete(fi, MPI_INFO_NULL);
        MPI_File_open(MPI_COMM_WORLD, fi, MPI_MODE_CREATE |
MPI_MODE_WRONLY, MPI_INFO_NULL, &fl);
        MPI_File_set_view(fl, 0, MPI_CHAR, big, "native", MPI_INFO_NULL);
        MPI_File_write_all(fl, bf, 1, sub, MPI_STATUS_IGNORE);
        MPI_File_close(&fl);
        MPI_Finalize();
        return 0;
}
```

Результаты

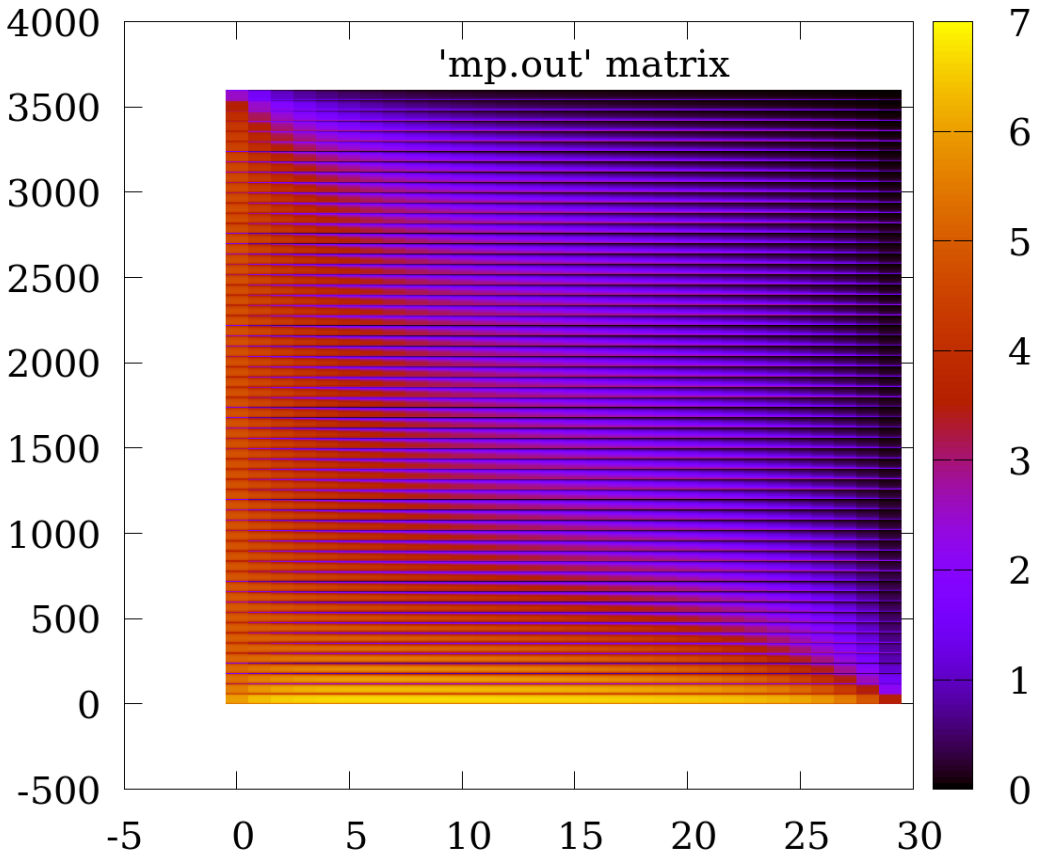
20*20*20

Время работы	dim1	dim2	dim3
0.139784sec	1	1	1
72.7262sec	1	1	2
362.159sec	1	2	2



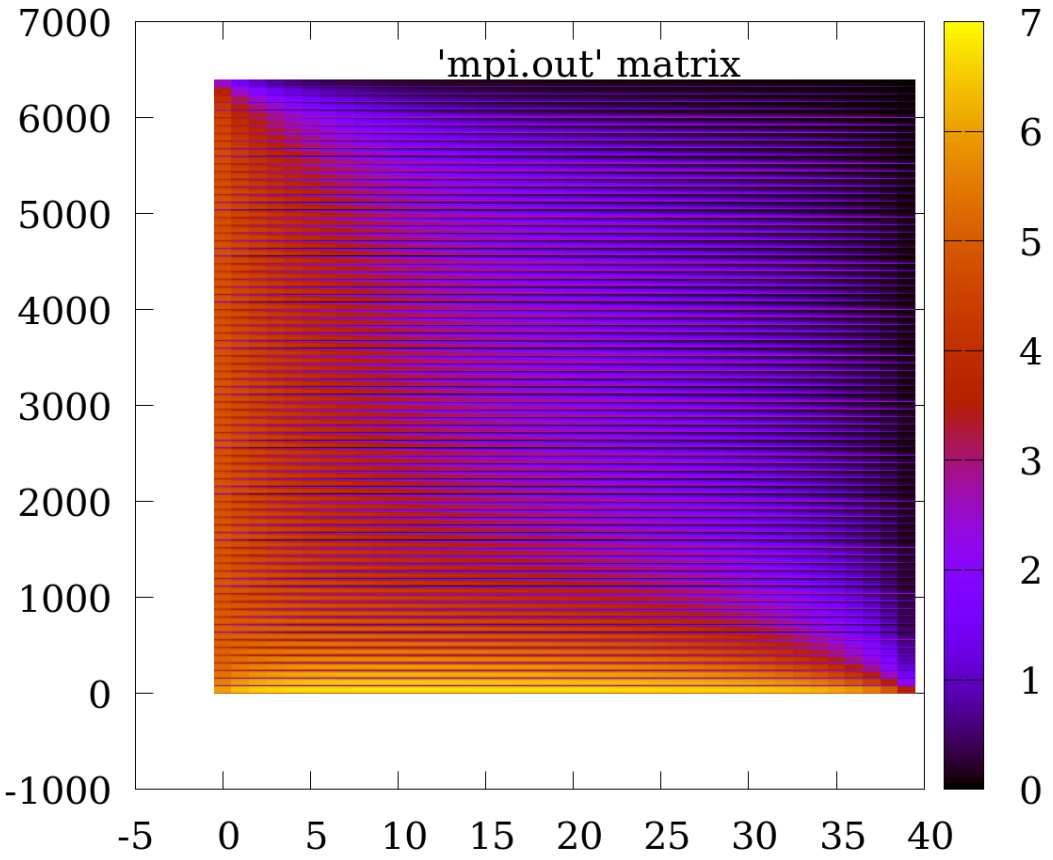
30*30*30

Время работы	dim1	dim2	dim3
0.86876sec	1	1	1
151.915sec	1	1	2
769.935sec	1	2	2



40*40*40

Время работы	dim1	dim2	dim3
2.93173sec	1	1	1
262.116sec	1	1	2
1331.28sec	1	2	2



Выводы

Выполнив данную лабораторную работу я применил технологию OPENMP. Также успешное выполнение данной лабораторной работы подразумевало под собой полное избавление от буферов(используемых для передачи данных), что я и сделал при помощи производных типов данных.
Сам процесс выполнения данной лабораторной работы не занял много времени. Если сравнивать результаты с предыдущими лабораторными работами, то можно заметить, что время работы данной программы значительно увеличилось. Скорее всего это связано с тем, что я проводил замеры на личной машине.