Отчёт по практической задаче №2 (неявная схема).
В рамках курса «Параллельные методы решения задач».

Выполнил: Волков Н. И. , академическая группа 523/1

1. Постановка задачи.

Производится численное решение двумерного нестационарного неоднородного уравнения теплопроводности в прямоугольной области (в некотором материале).
Такой перенос тепла описывается уравнением следующего вида

$$\rho * c * (dT/dt) = \lambda * (d^2T/dx^2 + d^2T/dy^2) + Q$$

где T – температура, зависит от x, y, t; Q – функция, характеризующая дополнительные источники тепловыделения, зависит в общем случае от x, y, t, T. X, Y – декартовы координаты, t – время. ρ, c, λ – параметры, характеризующие «материал» области, в которой рассматривается уравнение теплопроводности. При решении задаётся размер сетки и количество шагов по времени. При этом сам размер поля остаётся неизменным.

2. Описание метода решения и способа декомпозиции области.

Для численного решения уравнения область равномерно разбивается между процессами (сетка подбирается таким образом, чтобы обеспечить точное разбиение). Каждому процессу в итоге соответствует некоторый прямоугольник из заданной области, определяемый по координатам процесса в двумерной MPI решетке. Это допустимо, поскольку вычислительная сложность каждой ячейки сетки условно одинакова (в том смысле, что каждая ячейка сетки подлежит расчёту по одному и тому же алгоритму).

В процессе решения задачи используемые узлы формируют двумерную MPI решетку. Расчёт новых значений температуры на каждом шаге по времени производится в два этапа. На первом этапе каждый процесс построчно рассматривает свою расчётную область. Тепловой поток в направлении оси Y представляется в неявной форме (в момент времени t(i + 1)), тепловой поток в направлении оси X игнорируется. На втором полушаге всё происходит наоборот. При этом функция дополнительных источников тепла в обоих полушагах берется для момента времени t(i + ½), что заложено в код программы. Таким образом, получается две разностные схемы вида.

$\Phi_{mk} - \Psi_{mk} / \Delta t / 2 = v \Delta_{yy} \Phi_{mk} + Q \qquad \Omega_{mk} - \Phi_{mk} / \Delta t / 2 = v \Delta_{xx} \Omega_{mk} + Q$

В первой из них неявной частью является та, что соответствует Φ, во второй — та, что соответствует Ω.

3. Описание модели поведения материала и точного решения.

Для проведения тестов и верификации в качестве начального значения температуры берется функция sinx + siny. При этом размер рассматриваемой области берется кратным 2*π. Тогда на границах области начальная температура равна, соответственно sinx или siny. Дополнительные источники тепла функционируют одинаково во всей области по закону f = p * cos(t / q), то есть каждая точка области попеременно нагревается и охлаждается в определенной степени. В программу можно встроить и иные параметры. Например, если постановить f = 0 в любой момент времени в любой точке области, то будет известно точное решение $e^{-t} * \sin(x) + e^{-t} \sin(y)$, при условии, что граничные условия выражаются функциями $e^{-t} * \sin x$ и $e^{-t} \sin y$ в каждый момент времени. Это свойство будет применено в дальнейшем для верификации программы.

4. Описание используемой вычислительной системы.

Расчёты проводились на вычислительной системе IBM Blue Gene/P, установленной в суперкомпьютерном комплексе МГУ. Основные её свойства следующие:

1) 2048 вычислительных узлов.
2) 2048 процессоров (1 процессор на узел).
3) 8192 ядра (4 ядра на процессор).
4) интерконнект для операций точка-точка — имеет топологию трехмерного тора, пропускная способность одного соединения — 450 MB/s. Латентность - 0.1μc для пакета в 32 байта, 0.8μc для пакета в 256 байта, до ближайшего соседа.
5) интерконнект для глобальных операций — по 3 соединения между каждым узлом и I/O картой (1 карта на 128 узлов); пропускная способность одного соединения — 850 MB/s. Латентность - 3.0μc (полный обход).

Более подробные характеристики этой системы можно найти по адресу http://hpc.cmc.msu.ru/bgp . Для расчётов использовалось 256 узлов, т. е. 1024 ядра, а также 64 узла (256 ядер). Поскольку необходимо сравнивать эффективность

параллельной программы с эффективностью последовательной, пришлось использовать сетки небольших размеров: 1024 * 1024, 2048 * 2048, 4096 * 4096 итп, чтобы последовательной программе хватило памяти на узле для оперирования всей расчётной сеткой, а также, чтобы хватило лимита времени для возможных тестов с большим числом шагов по времени.

5. Верификация программы, сходимость, проверка корректности вычислений.

Как уже было сказано, установка функции генерации тепла в тестовый режим принципиально не влияет на вычисления (изменяются только значения конкретного числового коэффициента F). Проверка корректности вычислений производится в 2 этапа. Сначала показывается корректность последовательной версии алгоритма. Затем показывается соответствие результатов работы последовательной и параллельной версий алгоритма. Примечание: для верификации использовались области малого размера, иначе бы долго работал верификатор (это простая утилита, не предназначенная для запуска на кластерной вычислительной системе).

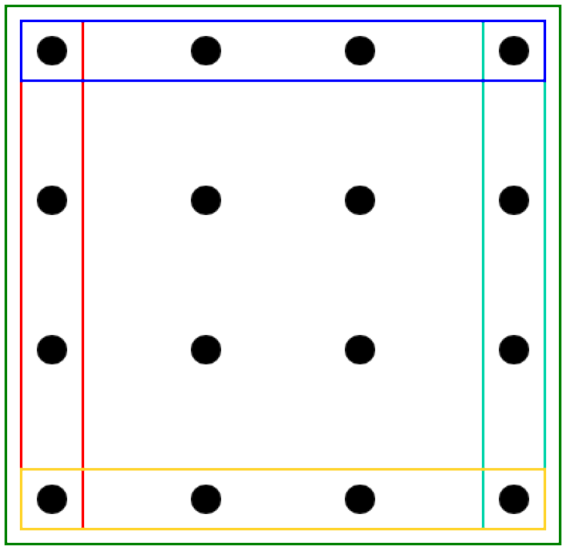

Рис 1: Первый вариант расчёта

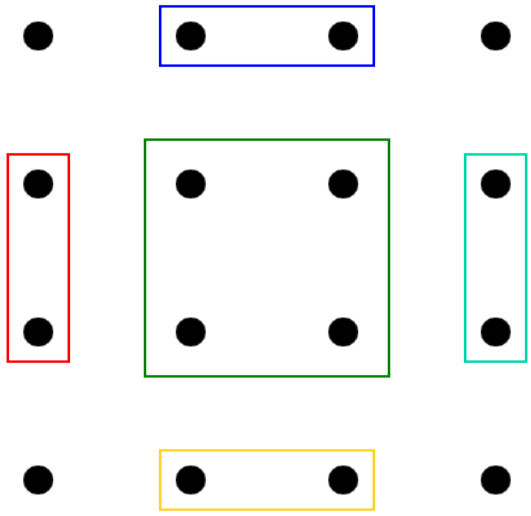

Рис 2: Второй вариант расчёта

Применяется два варианта расчётной сетки. Их суть изложена на Рис. 1 и Рис. 2 выше. Левый и правый столбцы, верхняя и нижняя строки соответствуют границе исследуемой области. Зеленый прямоугольник — область, для которой производится непосредственный расчёт (на изображениях параметр размера расчётной сетки в первом случае равен 4, во втором — 2). Все остальные

прямоугольники обозначают области, значения в которых рассматриваются как граничные условия. Полностью корректной является первая схема, однако вторая обладает интересными свойствами погрешности. Для проверки корректности достаточно взять значения в конкретных точках с конкретного шага по времени и сравнить их со значением точного решения при тех же параметрах. Коэффициенты ρ, c, λ, смысл которых заключается в физических характеристиках материала, в котором рассматривается теплопроводность, приравниваются к единице для более простой верификации. На Рис. 3 изображено отклонение от точного решения на 100-ом шаге по времени (параметр t = 0.01, то есть рассматривается момент времени 1.0) согласно результатам работы верификатора на сетках размера 128, 256, 384, 512, 640, 768, 1024, для первого и второго варианта расчётов. Погрешность вычисляется как среднее отклонение вычисленного решения от истинного значения в данной точке. *Эти значения также совпадают с точностью до машинного нуля для последовательной и параллельной версий алгоритма. Поэтому соответствующий график интереса не представляет и не приводится.* Интересный вывод заключается в том, что хотя погрешность 1 варианта уменьшается со сгущением сетки и уже на сетке 2048 * 2048 должна быть, по крайней мере, сопоставима с погрешностью второго варианта (которая почти не зависит от размера сетки), второй вариант обеспечивает хорошую

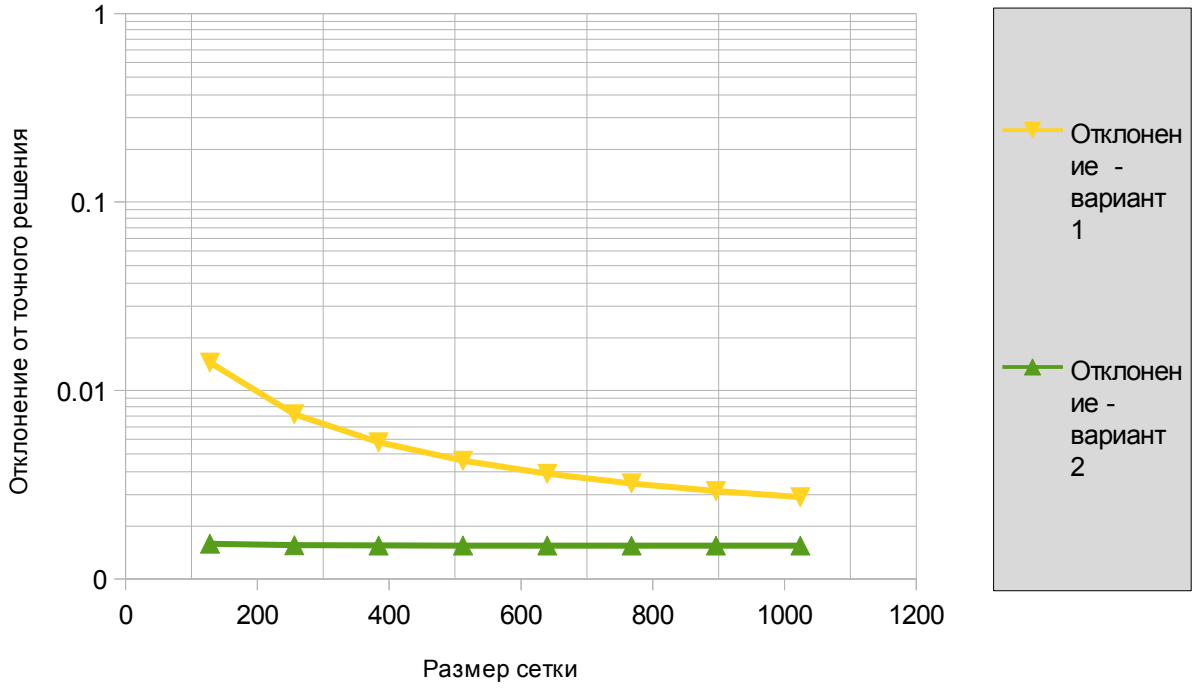

Рис 3: Отклонение от точного решения в зависимости от размера сетки на 100-ом шаге. Шкала логарифмическая.

точность на сетках малого размера. В дальнейших расчётах, однако, используется первый вариант.

6. Графики времени решения, эффективности распараллеливания итп.

К сожалению, последовательное решение на Blue Gene / P при более-менее значительных размерах сеток выполняется долго, вплоть до выхода за лимит времени. Поэтому последовательный алгоритм запускается для малого числа шагов по времени, а параллельный алгоритм — для большего числа шагов по времени. В качестве основной меры времени выполнения, эффективности итп. программы используется среднее время выполнения одного шага, получаемое простым арифметическим делением общего времени выполнения на количество шагов. Этот параметр рассматривается для квадратных сеток со стороной 1024, 2048, 3072, 4096, 5120, 6144, 7168, 8192. Во всех случаях рассматривается область одного размера (2 * π), то есть для все сеток шаг по x одинаков с шагом по y и равен (2 * π) / (data_size – 1). Шаг по времени во всех случаях равен 0.01. В каждом случае на Рис. 4 приведено среднее время выполнения одного шага для

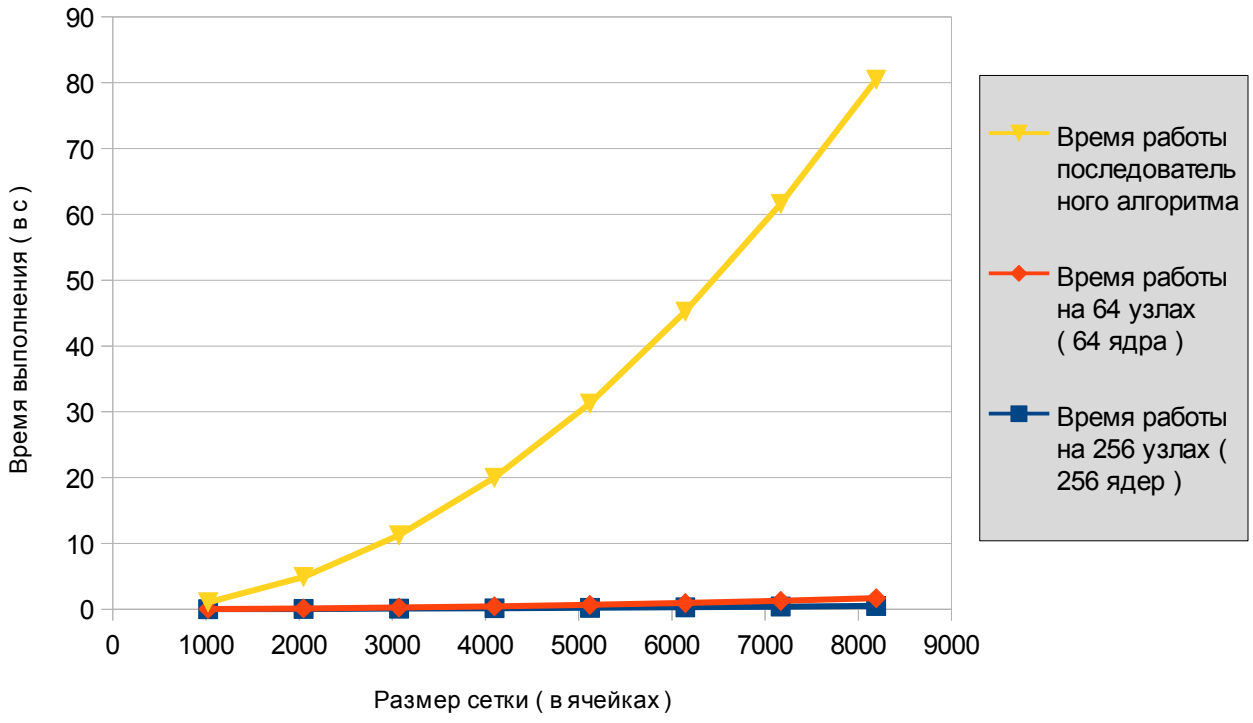

Рис 4: Время работы алгоритма без openmp на 1 узле, на 64 узлах, на 256 узлах

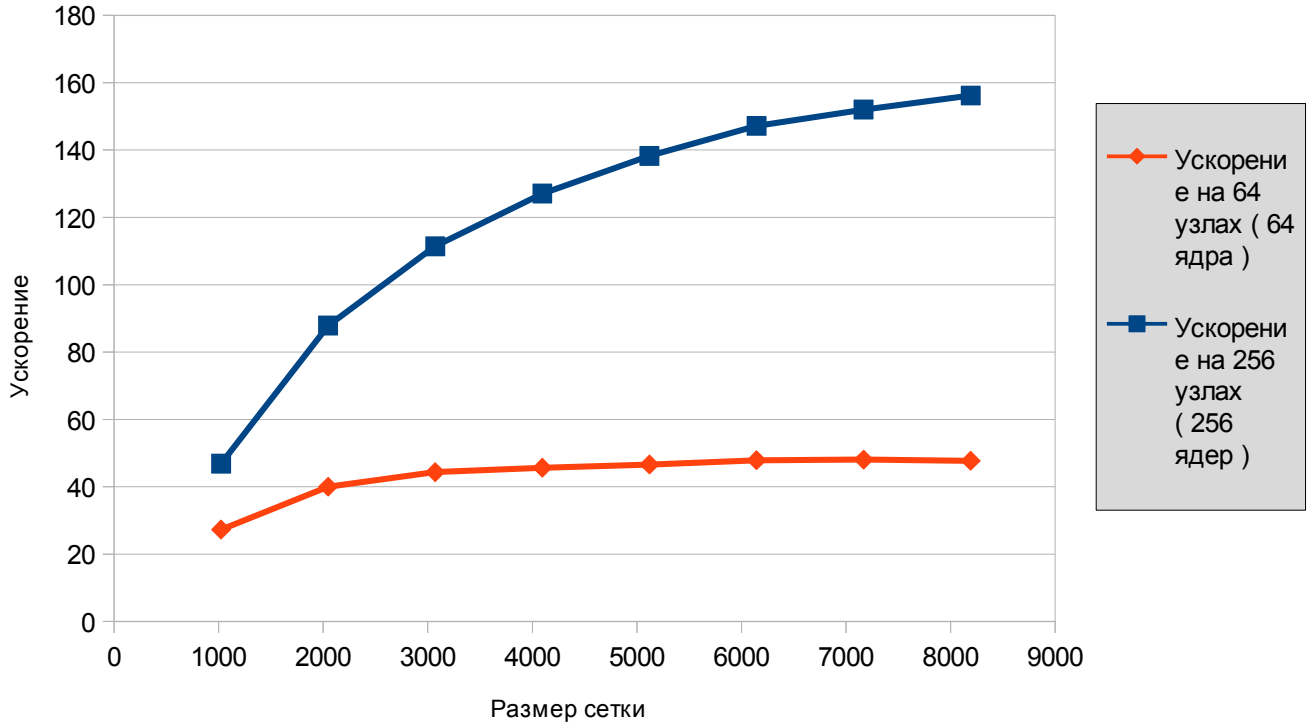

Рис 5: Ускорение алгоритма без opentp на 64 узлах, на 256 узлах по сравнению с 1 узлом

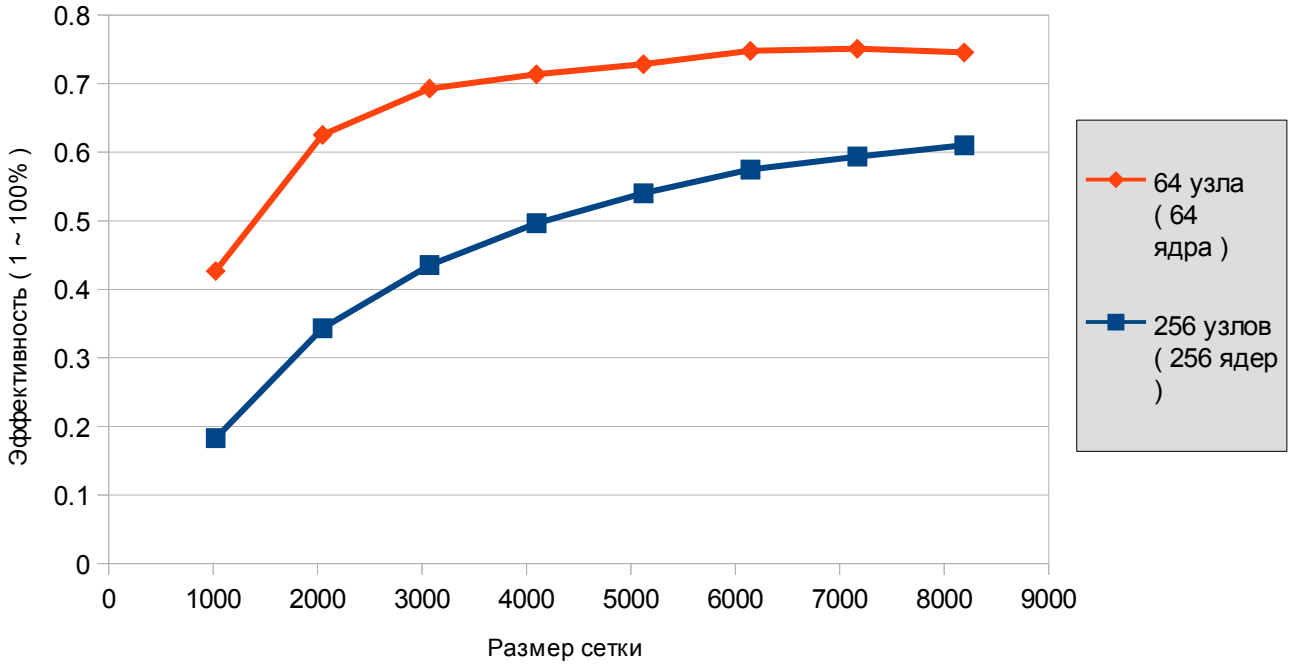

Рис 6: Эффективность алгоритма без opentp на 64 и 256 узлах

последовательного алгоритма на 1 узле, параллельного алгоритма на 64 узлах,

параллельного алгоритма на 256 узлах. На Рис. 5 показано ускорение на 64 узлах и на 256 узлах по сравнению с последовательным алгоритмом. Критерий сравнения не меняется, использованные размеры сетки также не меняются. На Рис. 6 указана эффективность алгоритма согласно определению эффективности (ускорение / число используемых узлов). Нужно помнить о том, что рассматриваемый алгоритм включает в себя параллельную блочную прогонку, эффективность которой не может превышать 50%.

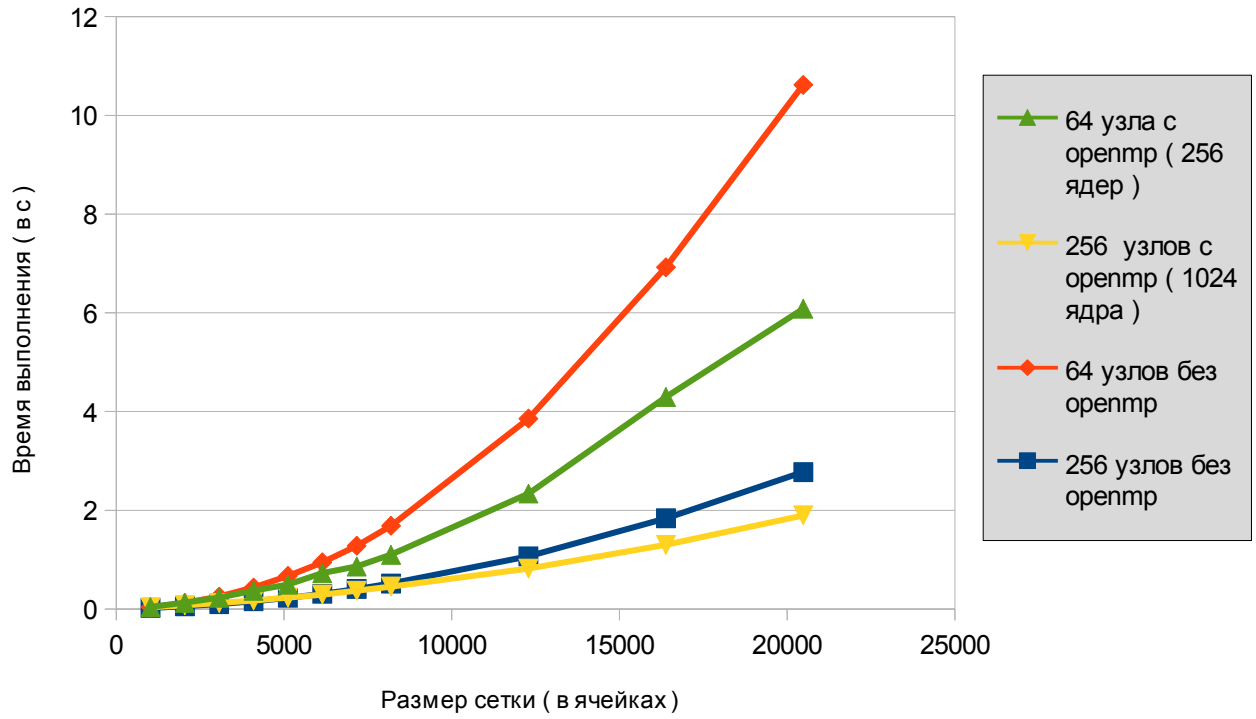

Рис 7: Сравнение времени работы программы с использованием opentp и без него

Параллельный алгоритм включает в себя использование openmp. Openmp используется следующим образом: Функция выполнения шага по времени имеет следующую структуру (outer_cycle_1 (inner_1_1) (inner_1_2) (INTERACT) (inner_1_3))(outer_cycle_1 (inner_1_1) (inner_1_2) (INTERACT) (inner_1_3)). Здесь каждый из внешних циклов обеспечивает обработку всей области, за которую отвечает процесс. Их два, поскольку каждый шаг по времени предполагает выполнение двух полушагов. Во внешних циклах использование openmp проблематично, поскольку их внутренняя структура включает в себя взаимодействие узлов. Первый внутренний цикл обеспечивает первоначальный расчёт коэффициентов и построение СЛАУ. В нём можно свободно использовать openmp. Второй внутренний цикл выполняет прямой ход метода прогонки в

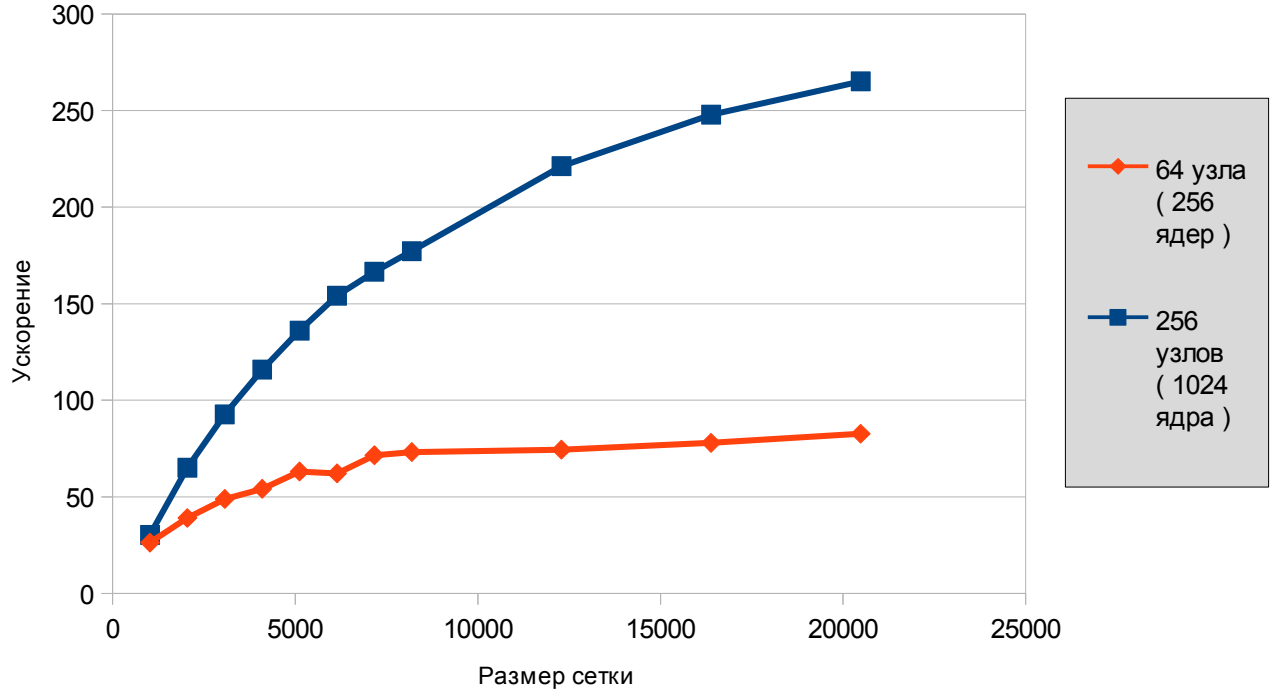

Рис 8: Ускорение алгоритма с opentp на 64, 256 узлах в сравнении с 1 узлом

локальной области. Здесь НЕЛЬЗЯ использовать openmp (например, для

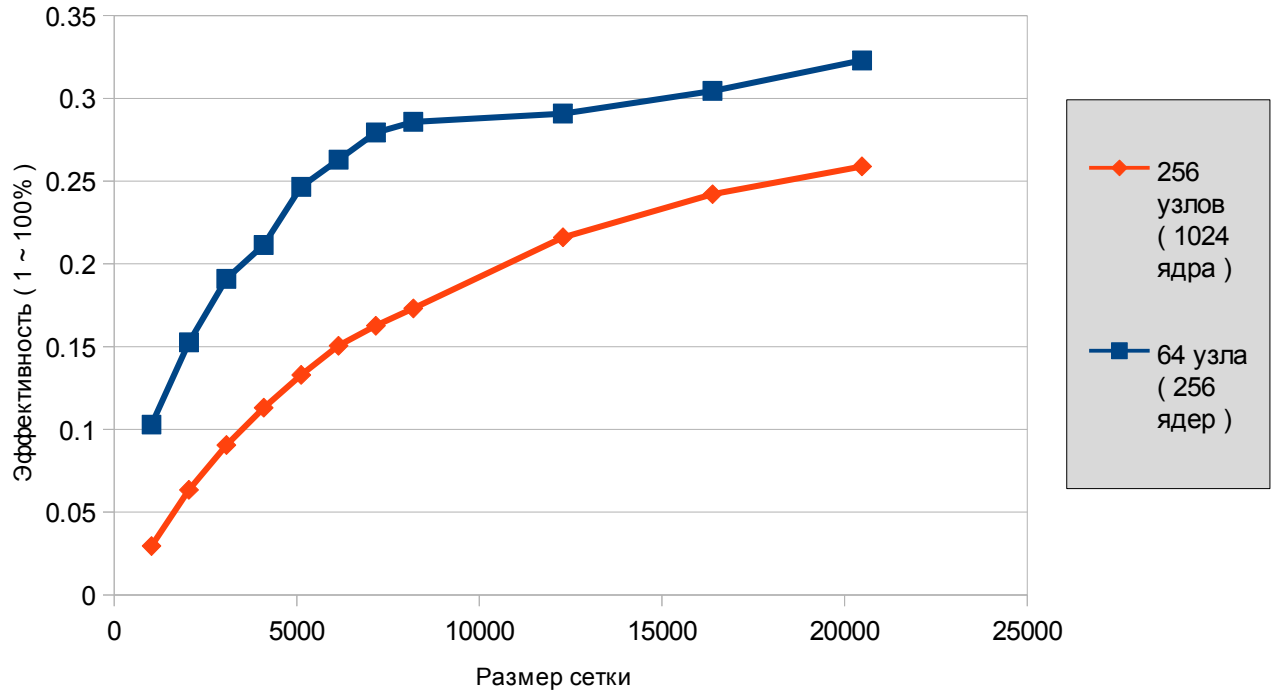

Рис 9: Эффективность алгоритма с opentp на 64, 256 узлах

организации встречной прогонки), поскольку мы можем выполнить только

прямой ход. Группа функций INTERACT отвечает за взаимодействие процессов (пересылку нижних строк матрицы. Здесь также нельзя использовать openmp. Однако openmp нужно и можно использовать для организации метода встречной прогонки в «главном» процессе каждого столбца на первом этапе и каждой строки на втором этапе, когда происходит решение трехдиагональной СЛАУ для матрицы из последних строк; что и делается.

Наконец, третий внутренний цикл представляет собой модифицированный обратный ход метода прогонки и также может выполняться лишь последовательно. На Рис. 7 приведено соотношение времени работы программы с использованием openmp (версия, использованная во всех остальных тестах) и программы, откуда использование openmp было удалено. Для последовательной версии алгоритма данные не приводятся, т. к. последовательная версия алгоритма не использует openmp по определению. На Рис. 8 и Рис. 9, однако, приводятся графики ускорения алгоритма без openmp на 64, 256 узлах по сравнению с последовательным вариантом (Для сеток больших размеров, которые не были рассчитаны последовательным алгоритмов, используется линейная интерполяция для установления возможного времени вычисления последовательным алгоритмов. Сетки такого размера в умещаются в оперативную память) а также их эффективность *относительно пиковой производительности системы*.

7. Анализ результатов и заключение.

Необходимо вкратце описать свойства реализованного алгоритма. Во-первых, часть с построением СЛАУ может выполняться одновременно и параллельно на всех узлах в процессе выполнения каждого шага по времени. Во-вторых, каждый узел при расчете очередного шага по времени производит пересылку 2 * (local_length + local_width) данных и принимает столько же посредством MPI. Между независимыми узлами пересылки также могут выполняться параллельно. Наконец, использован модифицированный вариант алгоритма блочной прогонки, в котором на обратном проходе зануляются наддиагональные элементы строк собственной полосы матрицы, начиная с *предпоследней* + последней строки с предыдущей полосы. Ускорение этого алгоритма составляет примерно $pn/(2n+p^2)$, где p – число блоков, n – число строк матрицы. Для систем с общей памятью n >> p, то есть ускорение должно быть близким к p / 2. Из этого можно сделать вывод о достаточно высокой эффективности представленной реализации алгоритма, т. к. на при достаточно большом размере сетки на p процессах он ускоряется приблизительно в p / 2 раз по сравнению с последовательным

вариантом. Фактическое значение ускорение по всему алгоритму превосходит p / 2 по нескольким причинам. Во-первых, один шаг по времени включает не только алгоритм блочной прогонки, и некоторые из операций (например, формирование СЛАУ) ускоряются именно в p раз. Во-вторых, планировщик задач Blue Gene / P не позволяет запускать на моём аккаунте задачи менее чем на 16 узлов. Отсюда могут происходить некоторые накладные расходы в последовательном алгоритме.
Если говорить более конкретно, то на обработку одного столбца на одном шаге по времени уходит ~N операций на формирование СЛАУ и ~3 * N операций на решение этой СЛАУ методом прогонки. Используя P процессоров, мы получим ~N/P + 6 * N / P операций, которые необходимо выполнить последовательно. Это означает 7 * N / P операций по сравнению с 4 * N для последовательного алгоритма. То есть ускорение последовательного алгоритма на 64 узлах составило бы 36.57142... . Реальное ускорение несколько больше, т. к. операции формирования СЛАУ имеют большую вычислительную сложность, особенно в сравнении с вычислением значений неизвестных на последнем этапе метода прогонки. Еще одним источником ускорения являются N ^ 2 операций по определению граничных условий в начале каждого шага по времени, которые также распараллеливаются на p процессоров.

Также нужно отметить, что использование openmp не дало существенного прироста производительности. Это вызвано в первую очередь ограниченностью использования openmp только для составления первоначальных СЛАУ и решения СЛАУ для «глобальной» матрицы, размеры которой невелики. **В случае с BG/P наилучшим способом видится использовать режим -VN (** http://hpc.cmc.msu.ru/bgp/jobs/modes **), который позволяет каждому ядру на узле функционировать как полноценный MPI процесс. Это средство поддерживается на аппаратном уровне, кроме того, в рамках MPI процесс выполнения шага по времени распараллеливается полностью, в то время как в openmp область параллельного выполнения весьма ограничена.**

8. Дополнительные материалы.

Верификатор — простая консольная программа, которая сверяет результаты работы либо последовательного и параллельного алгоритмов, либо сверяет работу последовательного алгоритма с файлом, содержащим финальное стабильное состояние области, к которому стремится точное решение.

```
// Basic inclusions.
#include <cstdlib>
#include <cstdio>
```

```cpp
#include <fstream>
#include <iostream>

// No MPI and project inclusions here.

// Using directives.
using namespace std;

// This function reads data from a file
void read_file(double* matrix, const int size, const char* file_name) {
	// Filestream...
	ifstream read_file;
	read_file.open(file_name, ios::in);

	// Read data.
	for (int i = 0; i < size; ++i) {
		read_file >> matrix[i];
	}

	read_file.close();
}

// This function calculates the average mistake. It should be close to zero for correct
solutions.
// Performance doesn't matter at all here, since we run test examples.
double calc_corr(double* left_mtrx, double* right_mtrx, int length, int width, int p_size,
const int mode) {
	// Result and operational elements.
	double result = 0;
	double left_element = 0;
	double right_element = 0;

	// Number of blocks in matrix and global addresses.
	int side_size = (int)round(sqrt(p_size));
	int para_adress = 0;
	int test_adress = 0;

	// Local addresses.
	int l_para_adress = 0;
	int l_test_adress = 0;

	if (mode == 1) { // Verify parallel.
		// Operate each block separately.
		std::cout << "side:" << side_size << "\n";
		for (int i = 0; i < side_size; ++i) { // X axis
			for (int j = 0; j < side_size; ++j) { // Y axis
				// Set rank and global address.
				para_adress = (i * side_size + j) * (length /
side_size) * (width / side_size);
				test_adress = i * side_size * (length / side_size) *
(width / side_size) + j * (width / side_size);

				// Inner cycle - work with elements.
				for (int p = 0; p < (length / side_size); ++p) { // X
```

```
axis in outer cycle.
                                                for (int q = 0; q < (width / side_size); ++q) {
// Y axis in inner cycle.
                                                        l_para_adress = para_adress + p *
(width / side_size) + q;
                                                        l_test_adress = test_adress + p *
width + q;
                                                        result +=
pow(abs(left_mtrx[l_test_adress] - right_mtrx[l_para_adress]), 2);
                                                }
                                        }
                                }
                        }
                }
                else { // Verify sequental.
                        for (int i = 0; i <= length * width; ++i) {
                                result += pow(abs(left_mtrx[i] - right_mtrx[i]), 2);
                        }
                }
                return sqrt(result / (length * width));
}

// Main function :)
// This program is very simple. It takes two files as arguments.
// Also number of processors used for parallel calculation and
// global field size should be passed as parameters. Then the program
// calculates difference in sequental and parallel solutions.
// It is NOT supposed to run on a supercomputer... it's just a check.
void main(int argc, char* argv[]) {
        // Field size etc parameters...
        int p_size = atoi(argv[1]);
        int global_length = atoi(argv[2]);
        int global_width = atoi(argv[3]);

        // Memory for matrices.
        double* seq_matrix = new double[global_length * global_width];
        double* par_matrix = new double[global_length * global_width];

        // Here we read data.
        read_file(seq_matrix, global_length * global_width, argv[4]);
        read_file(par_matrix, global_length * global_width, argv[5]);

        // Here we set mode.
        int mode = atoi(argv[6]);

        // Now calculate the average mistake.
        std::cout << calc_corr(seq_matrix, par_matrix, global_length, global_width,
p_size, mode);

        // Free memory.
        delete seq_matrix;
        delete par_matrix;
}
```

9. Приложение 1. Последовательная версия программы.

И последовательная, и параллельная версия программы состоят из трёх файлов — Trans_solver.h , Trans_solver.cpp , Main.cpp . Класс Trans_solver отвечает за вычислительный процесс. Следует обратить внимание, что последовательная версия программы также использует MPI, хотя процессы никак не взаимодействуют между собой. Это — необходимая мера для того, чтобы программу можно было поставить в общую очередь на выполнение на системе Blue Gene / P.

```
// ------------------------------------------------------------------------
// This file contains class to solve given transcalency equations.
// ------------------------------------------------------------------------
#define _USE_MATH_DEFINES

// MPI inclusions.
#include <mpi.h>

// Basic inclusions.
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <fstream>
#include <cmath>

// Definitions
// Material is considered to be stainless steel.
#define DEFAULT_LAMBDA 1.0 // Lambda.
#define DEFAULT_C 1.0 // C.
#define DEFAULT_RO 1.0 // Ro.
// Timestep is considered 0.01 second.
#define DEFAULT_T_STEP 0.01
// Grid steps are considered M_PI both for x and y axis.
#define DEFAULT_X_STEP M_PI
#define DEFAULT_Y_STEP M_PI

// MPI_Sorter class is used to implement a sorting network to sort array fragments. Only
declaration is here.
// ------------------------------------------------------------------------
class Trans_solver
{
private:
        // Step parameters.
        double x_step;
        double y_step;
        double exponential_coeff;

        // Data field parameters.
        int field_length; // Data field length.
        int field_width; // Data field width.
        double* array_data; // A field of dots calculated.
```

```
        double* dub_array; // A field of dots calculated duplicate.

        // Border parameters.
        double* upper_border; // Border data :)
        double* lower_border; // Border data :)
        double* left_border; // Border data :)
        double* right_border; // Border data :)
        double border_value; // Default border value.

        // Tridiagonal equation system coefficients for horizontal solution step.
        double* alpha_coeff_y_axis;
        double* charlie_coeff_y_axis;
        double* beta_coeff_y_axis;

        double* alpha_coeff_x_axis;
        double* charlie_coeff_x_axis;
        double* beta_coeff_x_axis;

        // Equation coefficients Ay(i-1) - Cy(i) + By(i+1) = - F
        double A_coeff_x_axis;
        double B_coeff_x_axis;
        double C_coeff_x_axis;
        double A_coeff_y_axis;
        double B_coeff_y_axis;
        double C_coeff_y_axis;

        double* F_coeff_x_axis;
        double* F_coeff_y_axis;

        // This internal function is used to simulate heat generators.
        double generate_heat(const int iteration_number, const int x_pos, const int
y_pos);
public:
        // Constructor.
        Trans_solver();

        void Init(int data_length, int data_width, int p_rank, int p_size); // Initialize
solver with processor cart.
        void Fill_data(const int iter_num); // Fill solver with data.

        void Iterate(const int iter_num); // Perform a local iteration.
        void Output(const int mode); // Output data.

        // Destructor.
        virtual ~Trans_solver();
};

// Project inclusions.
#include "Trans_solver.h"

// ---------------------------------------------------------------------------
// This file contains functions used in trans_solver class
// ---------------------------------------------------------------------------
```

```
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------

// This internal function is used to simulate heat generators.
double Trans_solver::generate_heat(const int iter_num, const int x_pos, const int y_pos) {
        // This function can give an additional heat source to any cell in the field.
        // As you can see, it depends on coordinates and iteration number.
        // So it's basically f(r,t).
        // For test purposes it's considered a simple given function, nondependant of
position
        // This doesn't change any of calculation principles, since these additional heat
sources always provide a known value.

        // Actual function.
        //double result = (double)iter_num * DEFAULT_T_STEP + DEFAULT_T_STEP / 2.0;
        //return 0.2 * cos(((double)result / DEFAULT_T_STEP) / 20.0);

        // Test variant for checking.
        return 0.0;
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------

// Default constructor.
Trans_solver::Trans_solver() {
}

// Initialize solver ( make a cart of processes ).
void Trans_solver::Init(int data_length, int data_width, int p_rank, int p_size) {
        // Calculate field parameters and initialize field data.
        field_length = data_length;
        field_width = data_width;
        array_data = new double[field_length * field_width];
        dub_array = new double[field_length * field_width];

        // Allocate memory for border fields.
        // Border values will be set later during data filling.
        upper_border = new double[field_length];
        lower_border = new double[field_length];
        left_border = new double[field_width];
        right_border = new double[field_width];

        // Allocate memory for Tridiagonal coefficients.
        alpha_coeff_y_axis = new double[field_width];
        charlie_coeff_y_axis = new double[field_width];
        beta_coeff_y_axis = new double[field_width];

        alpha_coeff_x_axis = new double[field_length];
        charlie_coeff_x_axis = new double[field_length];
        beta_coeff_x_axis = new double[field_length];

        // Calculate step sizes.
        x_step = (2.0 * DEFAULT_X_STEP) / (double)(data_length - 1);
```

```
        y_step = (2.0 * DEFAULT_Y_STEP) / (double)(data_width - 1);

        // Calculate static equation coefficients.
        A_coeff_x_axis = DEFAULT_LAMBDA / (x_step * x_step);
        B_coeff_x_axis = DEFAULT_LAMBDA / (x_step * x_step);
        C_coeff_x_axis = -2.0 * (DEFAULT_LAMBDA / (x_step * x_step)) - (DEFAULT_RO *
DEFAULT_C) / DEFAULT_T_STEP;

        A_coeff_y_axis = DEFAULT_LAMBDA / (y_step * y_step);
        B_coeff_y_axis = DEFAULT_LAMBDA / (y_step * y_step);
        C_coeff_y_axis = -2.0 * (DEFAULT_LAMBDA / (y_step * y_step)) - (DEFAULT_RO *
DEFAULT_C) / DEFAULT_T_STEP;

        // Allocate memory for equation coefficients.
        F_coeff_x_axis = new double[field_length];
        F_coeff_y_axis = new double[field_width];
}

// Fill solver with test source data.
void Trans_solver::Fill_data(const int iter_num) {
        // Formula used is sqrt((x - global_length/2)(y - global_width/2)) +
sqrt((global_length/2)(global_width/2)) + 1
        // X and Y are global coordinates. They're calculated as field_param * process
coord + local cell coord.
        exponential_coeff = pow(M_E, -(iter_num) * DEFAULT_T_STEP);
        for (int length = 0; length < field_length; ++length) {
                for (int width = 0; width < field_width; ++width) {
                        array_data[field_width * length + width] =
                                // Build test surface.
                                exponential_coeff * sin((double)(length) * x_step) +
                                +exponential_coeff * sin((double)(width) * y_step);
                }
        }
}

// Perform a single iteration.
void Trans_solver::Iterate(const int iter_num) {
        // Set new border values ( we know a precise solution ).
        exponential_coeff = pow(M_E, -(iter_num + 1) * DEFAULT_T_STEP);
        for (int i = 0; i < field_length; ++i) left_border[i] = exponential_coeff *
sin(double(i) * x_step); // Left edge
        for (int i = 0; i < field_length; ++i) right_border[i] = exponential_coeff *
sin(double(i) * x_step); // Right edge
        for (int i = 0; i < field_width; ++i) upper_border[i] = exponential_coeff *
sin(double(i) * y_step); // Upper edge
        for (int i = 0; i < field_width; ++i) lower_border[i] = exponential_coeff *
sin(double(i) * y_step); // Lower edge.

        // This coeddicient is used in string subtraction process.
        double subtract_coeff_y_axis = 0;

        // First step is "width solution".
        // ---------------------------------------------------------------
        // ---------------------------------------------------------------
```

```
        for (int i = 0; i < field_length; ++i) { // Solution is done separately for every
horizontal line.

                // Initial action is calculating all coefficients of the tridiagonal
equation. Note that A,B,C are same for all equations and only F differs.
                for (int j = 0; j < field_width; ++j) {

                        // Pass A, B, C coeffs to the tridiagonal equation.
                        alpha_coeff_y_axis[j] = A_coeff_y_axis;
                        charlie_coeff_y_axis[j] = C_coeff_y_axis;
                        beta_coeff_y_axis[j] = B_coeff_y_axis;

                        // Calculate F_coeff.
                        F_coeff_y_axis[j] =
                                - array_data[i * field_width + j] * (DEFAULT_RO *
DEFAULT_C / (DEFAULT_T_STEP)) -
                                generate_heat(iter_num, i, j);
                }

                // In case our processor is responsible for the edge rectangle,
                // we can adjust F_coefficient right now according to REAL border values.
                // This also means that left data buffer will not be actually used. We
still keep it, since other processors will be busy with them anyway.
                F_coeff_y_axis[0] -= A_coeff_y_axis * left_border[i];
                alpha_coeff_y_axis[0] = 0;
                F_coeff_y_axis[field_width - 1] -= B_coeff_y_axis * right_border[i];
                beta_coeff_y_axis[field_width - 1] = 0;

                // Now we have to solve the equation system, which is done in several
basic steps.
                //
--------------------------------------------------------------------------

                // At first step, we subtract equation lines top-down; this process makes
all alpha coeffs = 0 and adjusts f_coeffs.
                for (int j = 1; j < field_width; ++j) {
                        // Subtraction coefficient calculation.
                        subtract_coeff_y_axis = alpha_coeff_y_axis[j] /
charlie_coeff_y_axis[j - 1];

                        // Solution vector.
                        F_coeff_y_axis[j] -= subtract_coeff_y_axis * F_coeff_y_axis[j -
1];

                        // Alpha and charlie vectors. Beta vector doesn't change during
this step.
                        alpha_coeff_y_axis[j] = 0; // -= subtract_coeff_y_axis *
charlie_coeff_y_axis[j - 1];
                        charlie_coeff_y_axis[j] -= beta_coeff_y_axis[j - 1] *
subtract_coeff_y_axis;
                }

                // At second step same sequence of operations is done in reversal.
                for (int j = field_width - 2; j >= 0; --j) {
```

```
                        // Subtraction coefficient calculation.
                        subtract_coeff_y_axis = beta_coeff_y_axis[j] /
charlie_coeff_y_axis[j + 1];

                        // Solution vector.
                        F_coeff_y_axis[j] -= subtract_coeff_y_axis * F_coeff_y_axis[j +
1];

                        // Beta vector becomes zero, charlie vector is not affected
( alpha should bezero ).
                        // charlie_coeff_y_axis[j] -= subtract_coeff_y_axis *
alpha_coeff_y_axis[j + 1];
                        beta_coeff_y_axis[j] = 0;
                }

                // Calculate actual values now.
                for (int j = field_width - 1; j >= 0; --j) {
                        dub_array[i * field_width + j] = F_coeff_y_axis[j] /
charlie_coeff_y_axis[j];
                }
        }
        // ------------------------------------------------------------------
        // ------------------------------------------------------------------

        // This coeddicient is used in string subtraction process.
        double subtract_coeff_x_axis = 0;

        // Second step is "length solution".
        // ------------------------------------------------------------------
        // ------------------------------------------------------------------
        for (int i = 0; i < field_width; ++i) { // Solution is done separately for every
vertical line.

                // Initial action is calculating all A,B,C,F coefficients of the
tridiagonal equation. Note that A,B,C are same for all equations and only F differs.
                // So we only need to calculate F coefficient.
                for (int j = 0; j < field_length; ++j) {

                        // Pass A, B, C coeffs to the tridiagonal equation.
                        alpha_coeff_x_axis[j] = A_coeff_x_axis;
                        charlie_coeff_x_axis[j] = C_coeff_x_axis;
                        beta_coeff_x_axis[j] = B_coeff_x_axis;

                        // Calculate F_coeff.
                        F_coeff_x_axis[j] =
                                - dub_array[j * field_length + i] * (DEFAULT_RO *
DEFAULT_C / (DEFAULT_T_STEP))
                                - generate_heat(iter_num, j, i);
                }

                // In case our processor is responsible for the edge rectangle,
                // we can adjust F_coefficient right now according to REAL border values.
                F_coeff_x_axis[0] -= A_coeff_x_axis * upper_border[i];
                alpha_coeff_x_axis[0] = 0;
```

```
                F_coeff_x_axis[field_length - 1] -= B_coeff_x_axis * lower_border[i];
                beta_coeff_x_axis[field_length - 1] = 0;

                // Now we have to solve the equation system, which is done in several
basic steps.
                //
---------------------------------------------------------------------------

                // At first step, we subtract equation lines top-down; this process makes
all alpha coeffs = 0 and adjusts f_coeffs. However,
                for (int j = 1; j < field_length; ++j) {
                        // Subtraction coefficient calculation.
                        subtract_coeff_x_axis = alpha_coeff_x_axis[j] /
charlie_coeff_x_axis[j - 1];

                        // Solution vector.
                        F_coeff_x_axis[j] -= subtract_coeff_x_axis * F_coeff_x_axis[j -
1];

                        // Alpha and charlie vectors. Beta vector doesn't change during
this step.
                        alpha_coeff_x_axis[j] = 0; // -= subtract_coeff_x_axis *
charlie_coeff_x_axis[j - 1];
                        charlie_coeff_x_axis[j] -= beta_coeff_x_axis[j - 1] *
subtract_coeff_x_axis;
                }

                // At second step same sequence of operations is done in reversal.
                for (int j = field_length - 2; j >= 0; --j) {
                        // Subtraction coefficient calculation.
                        subtract_coeff_x_axis = beta_coeff_x_axis[j] /
charlie_coeff_x_axis[j + 1];

                        // Solution vector.
                        F_coeff_x_axis[j] -= subtract_coeff_x_axis * F_coeff_x_axis[j +
1];

                        // Beta vector becomes zero, charlie vector is not affected
( alpha is zero ).
                        // charlie_coeff_x_axis[j] -= subtract_coeff_x_axis *
alpha_coeff_x_axis[j + 1];
                        beta_coeff_x_axis[j] = 0;
                }

                // Finally, every process locally fills the remaining dub_array cells.
                for (int j = field_length - 1; j >= 0; --j) {
                        array_data[j * field_length + i] = F_coeff_x_axis[j] /
charlie_coeff_x_axis[j];
                }
        }
        // As horizontal solution is over, we should have fully updated data ready for
the next step. That means iteration is over.
        // ---------------------------------------------------------------
        // ---------------------------------------------------------------
```

```
}

// Output results to a file.
void Trans_solver::Output(const int mode) {
        std::fstream out_file;
        if (mode == 1) { // Use stdout.
                for (int i = 0; i < field_width; ++i) {
                        for (int j = 0; j < field_length; ++j) {
                                std::cout << array_data[i * field_length + j] << " ";
                        }
                }
        }
        else
        if (mode == 2) { // Use fstream.
                out_file.open("std_out.txt", std::ios::out);
                for (int i = 0; i < field_width; ++i) {
                        for (int j = 0; j < field_length; ++j) {
                                out_file << array_data[i * field_length + j] << " ";
                        }
                        out_file << "\n";
                }
                out_file.close();
        }
}

// Default destructor.
Trans_solver::~Trans_solver(){
        // Delete field data.
        if (array_data != NULL) { delete array_data; }
        if (dub_array != NULL) { delete dub_array; }

        // Delete border data.
        if (upper_border != NULL) { delete upper_border; }
        if (lower_border != NULL) { delete lower_border; }
        if (left_border != NULL) { delete left_border; }
        if (right_border != NULL) { delete right_border; }

        // Delete coefficients data.
        if (F_coeff_x_axis != NULL) { delete F_coeff_x_axis; }
        if (F_coeff_y_axis != NULL) { delete F_coeff_y_axis; }
}

// Basic inclusions.
#include <cstdlib>
#include <cstdio>
#include <iostream>

// Mpi inclusions.
#include "mpi.h"

// Using directives.
using namespace std;

// Project inclusions.
```

```cpp
#include "Trans_solver.cpp"

// Main function :)
// ------------------------------------------------------------------------------
int main(int argc, char* argv[]) {
	// MPI rank & size variables.
	int p_rank, p_size;

	// Command line arguments.
	int data_length = atoi(argv[1]);
	int data_width = atoi(argv[2]);
	int iter_num = atoi(argv[3]);
	int calc_mode = atoi(argv[4]);
	int out_mode = atoi(argv[5]);

	// Initialize MPI.
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &p_rank);
	MPI_Comm_size(MPI_COMM_WORLD, &p_size);

	// Time parameters.
	double start_time;
	double end_time;

	Trans_solver Eq_solver; // Create a solver.
	Eq_solver.Init(data_length, data_width, p_rank, p_size); // Initialize solver.
	Eq_solver.Fill_data(0); // Load test data to solver.

	start_time = MPI_Wtime();

	// Perform as many iterations as needed.
	if (calc_mode == 0) { // Calculation mode.
		for (int i = 0; i < iter_num; ++i) {
			Eq_solver.Iterate(i);
		}
	}
	else if (calc_mode == 1) { // Generate test solution
		Eq_solver.Fill_data(iter_num);
	}

	MPI_Barrier(MPI_COMM_WORLD);
	end_time = MPI_Wtime();

	double total_time = end_time - start_time;
	if (out_mode == 0) { // Print time.
		cout << "Calculation time is : " << total_time << "\n";
	}
	else { // Verify
		Eq_solver.Output(out_mode);
	}

	// Finalize MPI.
	MPI_Finalize();
```

```
        // Return :)
        return 0;
}
```

10. Приложение 2. Параллельная версия программы.

Параллельная версия программы отличается от последовательной реализацией функции вычисления шага по времени. Нужно обратить внимание на ограниченность использования openmp – использовать openmp в самом внешнем цикле «нехорошо», поскольку внутри цикла происходит взаимодействие узлов через MPI. Из трёх крупных внутренних циклов распараллеливается через openmp только первый, т. к. в остальных двух имеется строго последовательная зависимость по данным. Ещё openmp используется при обработке «глобальной» матрицы.

```
// ------------------------------------------------------------------------------
// This file contains class to solve given transcalency equations.
// ------------------------------------------------------------------------------
#define _USE_MATH_DEFINES

// MPI inclusions.
#include <mpi.h>
#include <omp.h>

// Basic inclusions.
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <fstream>
#include <cmath>

// Definitions
#define default_dims 2
        // Material is considered to be stainless steel ( coeffs are added for FAST
calc ).
#define DEFAULT_LAMBDA 1.0 // Lambda.
#define DEFAULT_C 1.0 // C.
#define DEFAULT_RO 1.0 // Ro.
        // Timestep is considered 0.01 second.
#define DEFAULT_T_STEP 0.01
        // Default grid steps are considered M_PI both for x and y axis.
#define DEFAULT_X_STEP M_PI
#define DEFAULT_Y_STEP M_PI

// MPI_Sorter class is used to implement a sorting network to sort array fragments. Only
declaration is here.
// ------------------------------------------------------------------------------
class Trans_solver
{
private:
```

```
// Global data parameters.
int global_length;
int global_width;

// Step parameters.
double x_step;
double y_step;
double exponential_coeff;

// MPI_Cart parameters.
int p_rank;
int n_dims;
int* coords;
int* dub_coords;
int* dim_sizes;
int* periods;
MPI_Comm cart_comm;

// MPI_Cart send_recv parameters.
int shift_source, shift_dest;

MPI_Status status; // Status of MPI messages received.
MPI_Request s_request; // Send request for MPI messages.
MPI_Request r_request; // Receive request for MPI messages.

// Data field parameters.
int field_length; // Data field length.
int field_width; // Data field width.
double* array_data; // A field of dots calculated.
double* dub_array; // A field of dots calculated duplicate.

// Border parameters.
double* upper_border; // Border data :)
double* lower_border; // Border data :)
double* left_border; // Border data :)
double* right_border; // Border data :)
double border_value; // Default border value.

// Tridiagonal equation system coefficients for horizontal solution step.
double* alpha_coeff_y_axis;
double* charlie_coeff_y_axis;
double* beta_coeff_y_axis;

double* alpha_coeff_x_axis;
double* charlie_coeff_x_axis;
double* beta_coeff_x_axis;

// Auxillary data fields for matrix operations.
double* left_data_buffer;
double* right_data_buffer;

double* up_data_buffer;
double* down_data_buffer;
```

```
        // Equation coefficients Ay(i-1) - Cy(i) + By(i+1) = - F
        double A_coeff_x_axis;
        double B_coeff_x_axis;
        double C_coeff_x_axis;
        double A_coeff_y_axis;
        double B_coeff_y_axis;
        double C_coeff_y_axis;

        double* F_coeff_x_axis;
        double* F_coeff_y_axis;

        // Data buffers for upper line in reversal step of the method.
        double appendix; // Additional value used in forming the global matrix.
        double auxillary; // Data element from previous process - used in the follower.
        double* send_buf;
        double* recv_buf;

        // Global matrix used in the final step of the method.
        double* global_x_array;
        double* global_y_array;
        double* global_x_mtrx;
        double* global_y_mtrx;

        //This internal function solves a simple version of original equaition.
        void simple_solve(double* global_mtrx, double* global_array, const int length);

        // This internal function is used to simulate heat generators.
        double generate_heat(const int iteration_number, const int x_pos, const int
y_pos);

public:
        // Constructor.
        Trans_solver();

        void Init(int data_length, int data_width, int p_rank, int p_size); // Initialize
solver with processor cart.
        void Fill_data(const int iter_num); // Fill solver with test source data.

        void Iterate(const int iter_num); // Perform a local iteration.
        void Output(const int mode); // Output data.

        // Destructor.
        virtual ~Trans_solver();
};

// Project inclusions.
#include "Trans_solver.h"

// ------------------------------------------------------------------------------
// This file contains functions used in trans_solver class
// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------
// ------------------------------------------------------------------------------
```

```
//This internal function solves a simple version of original equaition.
void Trans_solver::simple_solve(double* global_mtrx, double* global_array, const int
length) {
	double global_subtract_coeff_1;
	double global_subtract_coeff_2;

	int upper = 0;
	int lower = length - 1;

	int i = 0;
	int j = 0;

	// Straight walk.
	#pragma omp parallel private(upper, lower) num_threads(2)
	{
		if (omp_get_thread_num() == 0) {
			for (i = 1; i < length / 2; ++i) {
				global_subtract_coeff_1 = global_mtrx[4 * i] /
global_mtrx[4 * (i - 1) + 1];
				global_mtrx[4 * i] -= global_mtrx[4 * (i - 1) + 1] *
global_subtract_coeff_1;
				global_mtrx[4 * i + 1] -= global_mtrx[4 * (i - 1) + 2]
* global_subtract_coeff_1;
				global_mtrx[4 * i + 3] -= global_mtrx[4 * (i - 1) + 3]
* global_subtract_coeff_1;
				upper++;
			}
		}
		else {
			for (j = length - 2; j >= length / 2; --j) {
				global_subtract_coeff_2 = global_mtrx[4 * j + 2] /
global_mtrx[4 * (j + 1) + 1];
				global_mtrx[4 * j + 1] -= global_mtrx[4 * (j + 1)] *
global_subtract_coeff_2;
				global_mtrx[4 * j + 2] -= global_mtrx[4 * (j + 1) + 1]
* global_subtract_coeff_2;
				global_mtrx[4 * j + 3] -= global_mtrx[4 * (j + 1) + 3]
* global_subtract_coeff_2;
				lower--;
			}
		}
	}

	// Synchronise.
	global_subtract_coeff_1 = global_mtrx[4 * (upper + 1)] / global_mtrx[4 * (upper)
+ 1];
	global_mtrx[4 * (upper + 1)] -= global_mtrx[4 * (upper)+1] *
global_subtract_coeff_1;
	global_mtrx[4 * (upper + 1) + 1] -= global_mtrx[4 * (upper)+2] *
global_subtract_coeff_1;
	global_mtrx[4 * (upper + 1) + 3] -= global_mtrx[4 * (upper)+3] *
global_subtract_coeff_1;
```

```
        global_subtract_coeff_2 = global_mtrx[4 * (lower - 1) + 2] / global_mtrx[4 *
(lower) + 1];
        global_mtrx[4 * (lower - 1) + 1] -= global_mtrx[4 * (lower)] *
global_subtract_coeff_2;
        global_mtrx[4 * (lower - 1) + 2] -= global_mtrx[4 * (lower)+1] *
global_subtract_coeff_2;
        global_mtrx[4 * (lower - 1) + 3] -= global_mtrx[4 * (lower)+3] *
global_subtract_coeff_2;

        #pragma omp parallel num_threads(2)
        {
                if (omp_get_thread_num() == 0) {
                        // Reverse walk.
                        for (int p = i + 1; p < length; ++p) {
                                global_subtract_coeff_1 = global_mtrx[4 * p] /
global_mtrx[4 * (p - 1) + 1];
                                global_mtrx[4 * p] -= global_mtrx[4 * (p - 1) + 1] *
global_subtract_coeff_1;
                                global_mtrx[4 * p + 1] -= global_mtrx[4 * (p - 1) + 2]
* global_subtract_coeff_1;
                                global_mtrx[4 * p + 3] -= global_mtrx[4 * (p - 1) + 3]
* global_subtract_coeff_1;
                        }
                } else {
                        // Reverse walk.
                        for (int q = j - 1; q >= 0; --q) {
                                global_subtract_coeff_2 = global_mtrx[4 * q + 2] /
global_mtrx[4 * (q + 1) + 1];
                                global_mtrx[4 * q + 1] -= global_mtrx[4 * (q + 1)] *
global_subtract_coeff_2;
                                global_mtrx[4 * q + 2] -= global_mtrx[4 * (q + 1) + 1]
* global_subtract_coeff_2;
                                global_mtrx[4 * q + 3] -= global_mtrx[4 * (q + 1) + 3]
* global_subtract_coeff_2;
                        }
                }
        }

        // Solutions, finally !.
        for (int m = 0; m < length; ++m) {
                global_array[m] = global_mtrx[4 * m + 3] / global_mtrx[4 * m + 1];
        }
}

// This internal function is used to simulate heat generators.
double Trans_solver::generate_heat(const int iter_num, const int x_pos, const int y_pos) {
        // This function can give an additional heat source to any cell in the field.
        // As you can see, it depends on coordinates and iteration number.
        // So it's basically f(r,t).
        // For test purposes it's considered a simple given function, nondependant of
position
        // This doesn't change any of calculation principles, since these additional heat
sources always provide a known value.
```

```
        // Actual function.
        //double result = (double)iter_num * DEFAULT_T_STEP + DEFAULT_T_STEP / 2.0;
        //return 0.2 * cos(((double)result / DEFAULT_T_STEP) / 20.0);

        // Test variant for checking.
        return 0.0;
}

// ------------------------------------------------------------------------------
// ------------------------------------------------------------------------------

// Default constructor.
Trans_solver::Trans_solver() {
}

// Initialize solver ( make a cart of processes ).
void Trans_solver::Init(int data_length, int data_width, int p_rank, int p_size) {
        // Set up global field parameters.
        global_length = data_length;
        global_width = data_width;

        // Allocate memory for cart parameters.
        n_dims = default_dims; // Number of dimensions - 2 by default.
        dim_sizes = new int[n_dims];
        periods = new int[n_dims];
        coords = new int[n_dims];
        dub_coords = new int[n_dims];

        // Calculate cart parameters and initialize cart.
        this->p_rank = p_rank;
        int cart_size = static_cast<int>(trunc(sqrt(p_size))); // HERE SHOULD BE A CHECK
WITH AN EXCEPTION !!!
        for (int i = 0; i < n_dims; ++i) {
                dim_sizes[i] = cart_size;
                periods[i] = 1; // The cart is periodic.
        }
        MPI_Cart_create(MPI_COMM_WORLD, default_dims, dim_sizes, periods, false,
&cart_comm);

        // Calculate field parameters and initialize field data.
        field_length = ((data_length % cart_size == 0) * data_length + (data_length %
cart_size != 0) * (data_length + cart_size - data_length % cart_size)) / cart_size;
        field_width = ((data_width % cart_size == 0) * data_width + (data_width %
cart_size != 0) * (data_width + cart_size - data_width % cart_size)) / cart_size;
        array_data = new double[field_length * field_width];
        dub_array = new double[field_length * field_width];

        // Allocate memory for border fields.
        // Border values will be set later during data filling.
        upper_border = new double[field_length];
        lower_border = new double[field_length];
        left_border = new double[field_width];
        right_border = new double[field_width];
```

```
        // Allocate memory for tridiagonal coefficients.
        alpha_coeff_y_axis = new double[field_width];
        charlie_coeff_y_axis = new double[field_width];
        beta_coeff_y_axis = new double[field_width];

        alpha_coeff_x_axis = new double[field_length];
        charlie_coeff_x_axis = new double[field_length];
        beta_coeff_x_axis = new double[field_length];

        // Allocate memory for auxillary data fields used in matrix operations.
        left_data_buffer = new double[field_width];
        right_data_buffer = new double[field_width];

        up_data_buffer = new double[field_length];
        down_data_buffer = new double[field_length];

        // Calculate step sizes.
        x_step = (2.0 * DEFAULT_X_STEP) / (double)(data_length - 1);
        y_step = (2.0 * DEFAULT_Y_STEP) / (double)(data_width - 1);

        // Calculate static equation coefficients.
        A_coeff_x_axis = DEFAULT_LAMBDA / (x_step * x_step);
        B_coeff_x_axis = DEFAULT_LAMBDA / (x_step * x_step);
        C_coeff_x_axis = -2.0 * (DEFAULT_LAMBDA / (x_step * x_step)) - (DEFAULT_RO *
DEFAULT_C) / DEFAULT_T_STEP;

        A_coeff_y_axis = DEFAULT_LAMBDA / (y_step * y_step);
        B_coeff_y_axis = DEFAULT_LAMBDA / (y_step * y_step);
        C_coeff_y_axis = -2.0 * (DEFAULT_LAMBDA / (y_step * y_step)) - (DEFAULT_RO *
DEFAULT_C) / DEFAULT_T_STEP;

        // Allocate memory for equation coefficients.
        F_coeff_x_axis = new double[field_length];
        F_coeff_y_axis = new double[field_width];

        // Allocate memory for send and receive buffers.
        send_buf = new double[4];
        recv_buf = new double[4];

        // Allocate memory for global matrix.
        global_x_array = new double[dim_sizes[0]];
        global_y_array = new double[dim_sizes[1]];
        global_x_mtrx = new double[4 * dim_sizes[0]];
        global_y_mtrx = new double[4 * dim_sizes[1]];
}

// Fill solver with test source data.
void Trans_solver::Fill_data(const int iter_num) {
        // Cart coordinates are required to set data appropriately.
        MPI_Cart_coords(cart_comm, p_rank, n_dims, coords);

        // Global offsets.
        int x_offset = field_length * coords[0];
        int y_offset = field_width * coords[1];
```

```
	exponential_coeff = pow(M_E, -(iter_num) * DEFAULT_T_STEP);
	// Formulae used is sqrt((x - global_length/2)(y - global_width/2)) +
sqrt((global_length/2)(global_width/2)) + 1
	// X and Y are global coordinates. They're calculated as field_param * process
coord + local cell coord.
	#pragma omp parallel for
	for (int length = 0; length < field_length; ++length) {
		for (int width = 0; width < field_width; ++width) {
			array_data[field_width * length + width] =
				// Build test surface.
				exponential_coeff * sin((double)(x_offset + length) *
x_step) +
				+ exponential_coeff * sin((double)(y_offset + width) *
y_step);
		}
	}
}

// Perform a single iteration.
void Trans_solver::Iterate(const int iter_num) {
	// Global offsets.
	int x_offset = field_length * coords[0];
	int y_offset = field_width * coords[1];

	//Set borders if process is responsible for the edge rectangle.
	exponential_coeff = pow(M_E, -(iter_num + 1) * DEFAULT_T_STEP);
	if (coords[1] == 0) { for (int i = 0; i < field_length; ++i) left_border[i] =
exponential_coeff * sin(double(x_offset + i) * x_step); } // Set left edge
	if (coords[1] == dim_sizes[1] - 1) { for (int i = 0; i < field_length; ++i)
right_border[i] = exponential_coeff * sin(double(x_offset + i) * x_step); } // Set right
edge
	if (coords[0] == 0) { for (int i = 0; i < field_width; ++i) upper_border[i] =
exponential_coeff * sin(double(y_offset + i) * y_step); } // Set upper edge
	if (coords[0] == dim_sizes[0] - 1) { for (int i = 0; i < field_width; ++i)
lower_border[i] = exponential_coeff * sin(double(y_offset + i) * y_step); } // Set lower
edge

	// This coeddicient is used in string subtraction process.
	double subtract_coeff_y_axis = 0;

	// First step is "width" solution
	// ---------------------------------------------------------------
	// ---------------------------------------------------------------
	for (int i = 0; i < field_length; ++i) { // Solution is done separately for every
horizontal line.
		// Don't forget to reset appendix from previous step.
		appendix = 0;
		auxillary = 0;

		// Initial action is calculating all coefficients of the tridiagonal
equation. Note that A,B,C are same for all equations and only F differs.
		#pragma omp parallel num_threads(4)
		{
```

```
                        #pragma omp for
                        for (int j = 0; j < field_width; ++j) {
                                // Pass A, B, C coeffs to the tridiagonal equation.
                                alpha_coeff_y_axis[j] = A_coeff_y_axis;
                                charlie_coeff_y_axis[j] = C_coeff_y_axis;
                                beta_coeff_y_axis[j] = B_coeff_y_axis;

                                // Calculate F_coeff.
                                F_coeff_y_axis[j] =
                                        - array_data[i * field_width + j] * (DEFAULT_RO
* DEFAULT_C / (DEFAULT_T_STEP)) -
                                        generate_heat(iter_num, i, j);

                                // Left and right data buffers contain only zeros by
default.
                                left_data_buffer[j] = 0;
                                right_data_buffer[j] = 0;
                        }
                }

                // Prepare left data buffer for storing "defective" lines.
                left_data_buffer[0] = alpha_coeff_y_axis[0];

                // In case our processor is responsible for the edge rectangle,
                // we can adjust F_coefficient right now according to REAL border values.
                // This also means that left data buffer will not be actually used. We
still keep it, since other processors will be busy with them anyway.
                if (coords[1] == 0) {
                        F_coeff_y_axis[0] -= A_coeff_y_axis * left_border[i];
                        alpha_coeff_y_axis[0] = 0;
                        left_data_buffer[0] = 0;
                }
                if (coords[1] == dim_sizes[1] - 1) {
                        F_coeff_y_axis[field_width - 1] -= B_coeff_y_axis *
right_border[i];
                        beta_coeff_y_axis[field_width - 1] = 0;
                }

                // Now we have to solve the equation system, which is done in several
basic steps.
                //
--------------------------------------------------------------------------

                // At first step, we subtract equation lines top-down; this process makes
all alpha coeffs = 0 and adjusts f_coeffs. However,
                // it also causes left  data buffers to become non-zero and thus making
those "defective" lines.
                for (int j = 1; j < field_width; ++j) {
                        // Subtraction coefficient calculation.
                        subtract_coeff_y_axis = alpha_coeff_y_axis[j] /
charlie_coeff_y_axis[j - 1];

                        // Solution vector.
                        F_coeff_y_axis[j] -= subtract_coeff_y_axis * F_coeff_y_axis[j -
```

```
1];

                        // Alpha and charlie vectors. Beta vector doesn't change during
this step.
                        alpha_coeff_y_axis[j] = 0;
                        charlie_coeff_y_axis[j] -= beta_coeff_y_axis[j - 1] *
subtract_coeff_y_axis;

                        // Left data buffer gets filled with defective data :(
                        left_data_buffer[j] -= left_data_buffer[j - 1] *
subtract_coeff_y_axis;
                }

                // At second step same sequence of operations is done in reversal.

                // This nullifies beta coefficient and fills right data buffers with
unnecessary data.
                // Right data buffer values are initialized here;
                if (field_width >= 2) {
                        right_data_buffer[field_width - 1] =
charlie_coeff_y_axis[field_width - 1];
                        right_data_buffer[field_width - 2] =
beta_coeff_y_axis[field_width - 2];
                }

                // This also changes data in left data buffer !!!
                for (int j = field_width - 3; j >= 0; --j) {
                        // Subtraction coefficient calculation.
                        subtract_coeff_y_axis = beta_coeff_y_axis[j] /
charlie_coeff_y_axis[j + 1];

                        // Solution vector.
                        F_coeff_y_axis[j] -= subtract_coeff_y_axis * F_coeff_y_axis[j +
1];

                        // Beta vector becomes zero, charlie vector is not affected
( alpha is zero ).
                        beta_coeff_y_axis[j] -= subtract_coeff_y_axis *
charlie_coeff_y_axis[j + 1];
                        charlie_coeff_y_axis[j] -= subtract_coeff_y_axis *
alpha_coeff_y_axis[j + 1];

                        // Right data buffer is filled with defective data. Left data
buffer is adjusted.
                        left_data_buffer[j] -= left_data_buffer[j + 1] *
subtract_coeff_y_axis;
                        right_data_buffer[j] -= right_data_buffer[j + 1] *
subtract_coeff_y_axis;
                }

                // Now the tricky part: we take the upper line, take CHARLIE, LEFT, RIGHT
and SOLUTION
                // values from there and send them to the previous process, if there's
one.
```

```
                send_buf[0] = left_data_buffer[0];
                send_buf[1] = charlie_coeff_y_axis[0];
                send_buf[2] = right_data_buffer[0];
                send_buf[3] = F_coeff_y_axis[0];

                if (dim_sizes[1] > 1) { // Send only there is somebody to receive.
                        MPI_Cart_shift(cart_comm, 1, -1, &shift_source, &shift_dest);
                        MPI_Sendrecv(send_buf, 4, MPI_DOUBLE, shift_dest, 0, recv_buf,
4, MPI_DOUBLE, shift_source, 0, cart_comm, &status);
                }
                else {
                        for (int k = 0; k <= 3; ++k) { recv_buf[k] = send_buf[k]; }
                }

                // Then we adjust actual values according to what we received and fill
ONE line of a global matrix.
                if (coords[1] != dim_sizes[1] - 1) { // It's not the "rightest" process.
                        // Calculate subtraction coefficient.
                        subtract_coeff_y_axis = beta_coeff_y_axis[field_width - 1] /
recv_buf[1];

                        // Calculate actual new values.
                        charlie_coeff_y_axis[field_width - 1] -= subtract_coeff_y_axis *
recv_buf[0];
                        beta_coeff_y_axis[field_width - 1] = 0;
                        appendix = -subtract_coeff_y_axis * recv_buf[2];
                        F_coeff_y_axis[field_width - 1] -= subtract_coeff_y_axis *
recv_buf[3];
                }

                // Form a line for the global matrix.
                send_buf[0] = left_data_buffer[field_width - 1];
                send_buf[1] = charlie_coeff_y_axis[field_width - 1];
                send_buf[2] = appendix;
                send_buf[3] = F_coeff_y_axis[field_width - 1];

                if (dim_sizes[1] > 1) { // Send only there is somebody to receive.
                        if (coords[1] == 0) { // Now processes with coords[1] == 0
receive data to global matrix.
                                dub_coords[0] = coords[0];
                                for (int j = 0; j <= 3; ++j) { global_y_mtrx[j] =
send_buf[j]; }
                                for (int j = 1; j < dim_sizes[1]; ++j) {
                                        dub_coords[1] = j;
                                        MPI_Cart_rank(cart_comm, dub_coords,
&shift_source);
                                        MPI_Recv(&global_y_mtrx[4 * j], 4, MPI_DOUBLE,
shift_source, 0, cart_comm, &status);
                                }
                        }
                        else { // And other processes send data to them.
                                dub_coords[0] = coords[0];
                                dub_coords[1] = 0;
```

```
                                MPI_Cart_rank(cart_comm, dub_coords, &shift_dest);
                                MPI_Send(send_buf, 4, MPI_DOUBLE, shift_dest, 0,
cart_comm);
                        }
                }
                else {
                        for (int k = 0; k <= 3; ++k) { global_y_mtrx[k] = send_buf[k]; }
                }

                // Simple version of the method is used for solving the equation with
global matrix.
                simple_solve(global_y_mtrx, global_y_array, dim_sizes[1]);

                // Results are sent to processes, directly into appopriate locations of
their dub_arrays.
                if (dim_sizes[1] > 1) { // There is somebody to receive.
                        if (coords[1] == 0) {
                                dub_array[i * field_width + (field_width - 1)] =
global_y_array[0];
                                dub_coords[0] = coords[0];
                                for (int j = 1; j < dim_sizes[1]; ++j) {
                                        dub_coords[1] = j;
                                        MPI_Cart_rank(cart_comm, dub_coords,
&shift_source);
                                        MPI_Send(&global_y_array[j], 1, MPI_DOUBLE,
shift_source, 0, cart_comm);
                                        MPI_Send(&global_y_array[j - 1], 1, MPI_DOUBLE,
shift_source, 0, cart_comm);
                                }
                        }
                        else {
                                MPI_Recv(&dub_array[i * field_width + (field_width -
1)], 1, MPI_DOUBLE, shift_dest,
                                        0, cart_comm, &status);
                                MPI_Recv(&auxillary, 1, MPI_DOUBLE, shift_dest,
                                        0, cart_comm, &status);
                        }
                }
                else {
                        dub_array[i * field_width + (field_width - 1)] =
global_y_array[0];
                }

                // Finally, every process locally fills the remaining dub_array cells.
                for (int j = field_width - 2; j >= 0; --j) {
                        dub_array[i * field_width + j] =
                        (F_coeff_y_axis[j] - left_data_buffer[j] * auxillary -
dub_array[i * field_width + (field_width - 1)] * right_data_buffer[j])
                        / charlie_coeff_y_axis[j];
                }
        }
        // ------------------------------------------------------------
        // ------------------------------------------------------------
```

```
        // This coeddicient is used in string subtraction process.
        double subtract_coeff_x_axis = 0;

        // Second step is "length" solution
        // ---------------------------------------------------------------
        // ---------------------------------------------------------------
        for (int i = 0; i < field_width; ++i) { // Solution is done separately for every
vertical line.
                // Don't forget to reset appendix from previous step.
                appendix = 0;
                auxillary = 0;

                // Initial action is calculating all A,B,C,F coefficients of the
tridiagonal equation. Note that A,B,C are same for all equations and only F differs.
                // So we only need to calculate F coefficient.
                #pragma omp parallel
                {
                        #pragma omp for
                        for (int j = 0; j < field_length; ++j) {
                                // Pass A, B, C coeffs to the tridiagonal equation.
                                alpha_coeff_x_axis[j] = A_coeff_x_axis;
                                charlie_coeff_x_axis[j] = C_coeff_x_axis;
                                beta_coeff_x_axis[j] = B_coeff_x_axis;

                                // Calculate F_coeff.
                                F_coeff_x_axis[j] =
                                        - dub_array[j * field_length + i] * (DEFAULT_RO
* DEFAULT_C / (DEFAULT_T_STEP))
                                        - generate_heat(iter_num, j, i);

                                // Left and right data buffers contain only zeros by
default.
                                up_data_buffer[j] = 0;
                                down_data_buffer[j] = 0;
                        }
                }

                // Prepare up data buffer for storing "defective" lines.
                up_data_buffer[0] = alpha_coeff_x_axis[0];

                // In case our processor is responsible for the edge rectangle,
                // we can adjust F_coefficient right now according to REAL border values.
                // This also means that up data buffer will not be actually used. We
still keep it, since other processors will be busy with them anyway.
                alpha_coeff_x_axis[0] = 0;
                if (coords[0] == 0) {
                        F_coeff_x_axis[0] -= A_coeff_x_axis * upper_border[i];
                        alpha_coeff_x_axis[0] = 0;
                        up_data_buffer[0] = 0;
                }
                if (coords[0] == dim_sizes[0] - 1) {
                        F_coeff_x_axis[field_length - 1] -= B_coeff_x_axis *
```

```
lower_border[i];
                        beta_coeff_x_axis[field_length - 1] = 0;
                }

                // Now we have to solve the equation system, which is done in several
basic steps.
                //
---------------------------------------------------------------------------

                // At first step, we subtract equation lines top-down; this process makes
all alpha coeffs = 0 and adjusts f_coeffs. However,
                // It also causes up  data buffers to become non-zero and thus making
those "defective" lines.
                for (int j = 1; j < field_length; ++j) {
                        // Subtraction coefficient calculation.
                        subtract_coeff_x_axis = alpha_coeff_x_axis[j] /
charlie_coeff_x_axis[j - 1];

                        // Solution vector.
                        F_coeff_x_axis[j] -= subtract_coeff_x_axis * F_coeff_x_axis[j -
1];

                        // Alpha and charlie vectors. Beta vector doesn't change during
this step.
                        alpha_coeff_x_axis[j] = 0; // -= subtract_coeff_x_axis *
charlie_coeff_x_axis[j - 1];
                        charlie_coeff_x_axis[j] -= beta_coeff_x_axis[j - 1] *
subtract_coeff_x_axis;

                        // Up data buffer gets filled with defective data :(
                        up_data_buffer[j] -= up_data_buffer[j - 1] *
subtract_coeff_x_axis;
                }

                // At second step same sequence of operations is done in reversal.
                // This nullifies beta coefficient and fills down data buffers with
unnecessary data.
                // Down data buffer values are initialized here;
                if (field_length >= 2) {
                        down_data_buffer[field_length - 1] =
charlie_coeff_x_axis[field_length - 1];
                        down_data_buffer[field_length - 2] =
beta_coeff_x_axis[field_length - 2];
                }

                // This also changes data in up data buffer !!!
                for (int j = field_length - 3; j >= 0; --j) {
                        // Subtraction coefficient calculation.
                        subtract_coeff_x_axis = beta_coeff_x_axis[j] /
charlie_coeff_x_axis[j + 1];

                        // Solution vector.
                        F_coeff_x_axis[j] -= subtract_coeff_x_axis * F_coeff_x_axis[j +
1];
```

```
                        // Beta vector becomes zero, charlie vector is not affected
( alpha is zero ).
                        beta_coeff_x_axis[j] -= subtract_coeff_x_axis *
charlie_coeff_x_axis[j + 1];

                        // Down data buffer is filled with defective data. Up data
buffer is adjusted.
                        up_data_buffer[j] -= up_data_buffer[j + 1] *
subtract_coeff_x_axis;
                        down_data_buffer[j] -= down_data_buffer[j + 1] *
subtract_coeff_x_axis;
                }

                // Now the tricky part: we take the upper line, take CHARLIE, UP, DOWN
and SOLUTION
                // values from there and send them to the previous process, if there's
one.
                send_buf[0] = up_data_buffer[0];
                send_buf[1] = charlie_coeff_x_axis[0];
                send_buf[2] = down_data_buffer[0];
                send_buf[3] = F_coeff_x_axis[0];

                if (dim_sizes[0] > 1) { // There is somebody to receive.
                        MPI_Cart_shift(cart_comm, 0, -1, &shift_source, &shift_dest);
                        MPI_Sendrecv(send_buf, 4, MPI_DOUBLE, shift_dest, 0, recv_buf,
4, MPI_DOUBLE, shift_source, 0, cart_comm, &status);
                }
                else {
                        for (int k = 0; k <= 3; ++k) { recv_buf[k] = send_buf[k]; }
                }

                // Then we adjust actual values according to what we received and fill
ONE line of a global matrix.
                if (coords[0] != dim_sizes[0] - 1) { // It's not the "lowest" process.
                        // Calculate subtraction coefficient.
                        subtract_coeff_x_axis = beta_coeff_x_axis[field_length - 1] /
recv_buf[1];

                        // Calculate actual new values.
                        charlie_coeff_x_axis[field_length - 1] -= subtract_coeff_x_axis
* recv_buf[0];
                        beta_coeff_x_axis[field_length - 1] = 0;
                        appendix -= subtract_coeff_x_axis * recv_buf[2];
                        F_coeff_x_axis[field_length - 1] -= subtract_coeff_x_axis *
recv_buf[3];
                }

                // Form a line for the global matrix.
                send_buf[0] = up_data_buffer[field_length - 1];
                send_buf[1] = charlie_coeff_x_axis[field_length - 1];
                send_buf[2] = appendix;
                send_buf[3] = F_coeff_x_axis[field_length - 1];
```

```
                if (dim_sizes[0] > 1) { // Send only there is somebody to receive.
                        if (coords[0] == 0) { // Now processes with coords[0] == 0
receive data to global matrix.
                                dub_coords[1] = coords[1];
                                for (int j = 0; j <= 3; ++j) { global_x_mtrx[j] =
send_buf[j]; }
                                for (int j = 1; j < dim_sizes[0]; ++j) {
                                        dub_coords[0] = j;
                                        MPI_Cart_rank(cart_comm, dub_coords,
&shift_source);
                                        MPI_Recv(&global_x_mtrx[4 * j], 4, MPI_DOUBLE,
shift_source, 0, cart_comm, &status);
                                }
                        }
                        else { // And other processes send data to them.
                                dub_coords[0] = 0;
                                dub_coords[1] = coords[1];
                                MPI_Cart_rank(cart_comm, dub_coords, &shift_dest);
                                MPI_Send(send_buf, 4, MPI_DOUBLE, shift_dest, 0,
cart_comm);
                        }
                }
                else {
                        for (int k = 0; k <= 3; ++k) { global_x_mtrx[k] = send_buf[k]; }
                }

                // Simple version of the method is used for solving the equation with
global matrix.
                simple_solve(global_x_mtrx, global_x_array, dim_sizes[0]);

                // Results are sent to processes, directly into appopriate locations of
their dub_arrays.
                if (dim_sizes[0] > 1) { // There is somebody to receive.
                        if (coords[0] == 0) {
                                array_data[(field_width - 1) * field_length + i] =
global_x_array[0];
                                dub_coords[1] = coords[1];
                                for (int j = 1; j < dim_sizes[1]; ++j) {
                                        dub_coords[0] = j;
                                        MPI_Cart_rank(cart_comm, dub_coords,
&shift_source);
                                        MPI_Send(&global_x_array[j], 1, MPI_DOUBLE,
shift_source, 2, cart_comm);
                                        MPI_Send(&global_x_array[j - 1], 1, MPI_DOUBLE,
shift_source, 2, cart_comm);
                                }
                        }
                        else {
                                MPI_Recv(&array_data[(field_width - 1) * field_length +
i], 1, MPI_DOUBLE, shift_dest,
                                        2, cart_comm, &status);
                                MPI_Recv(&auxillary, 1, MPI_DOUBLE, shift_dest,
                                        2, cart_comm, &status);
```

```
                        }
                }
                else {
                        array_data[(field_width - 1) * field_length + i] =
global_x_array[0];
                }

                // Finally, every process locally fills the remaining dub_array cells.
                for (int j = field_length - 2; j >= 0; --j) {
                        array_data[j * field_length + i] =
                                (F_coeff_x_axis[j] - up_data_buffer[j] * auxillary -
array_data[(field_width - 1) * field_length + i] * down_data_buffer[j]) /
charlie_coeff_x_axis[j];
                }
        }
        // As horizontal solution is over, we should have fully updated data ready for
the next step. That means iteration is over.
        // ----------------------------------------------------------------
        // ----------------------------------------------------------------
}



// Output results to a file.
void Trans_solver::Output(const int mode) {
        double send_double = 0;
        double recv_double = 0;

        // Calculate source process.
        if (!((coords[0] == 0) && (coords[1] == 0))) { // Need to receive.
                dub_coords[0] = coords[0];
                dub_coords[1] = coords[1] - 1;
                if (dub_coords[1] < 0) {
                        dub_coords[1] = dim_sizes[1] - 1;
                        dub_coords[0]--;
                }
                MPI_Cart_rank(cart_comm, dub_coords, &shift_source);
                MPI_Recv(&recv_double, 1, MPI_DOUBLE, shift_source, 5, cart_comm,
&status);
        }


        std::fstream out_file;
        // In parallel version processes write into memory one after another.
        if (mode == 1) { // Use stdout.
                for (int i = 0; i < field_width; ++i) {
                        for (int j = 0; j < field_length; ++j) {
                                std::cout << array_data[i * field_length + j] << " ";
                        }
                }
        }
        // In parallel version processes write into memory one after another.
        else { // Use fstream.
```

```
                if ((coords[0] == 0) && (coords[1] == 0)) {
                        out_file.open("std_out.txt", std::fstream::out);
                }
                else {
                        out_file.open("std_out.txt", std::fstream::in |
std::fstream::out | std::fstream::ate);
                }
                for (int i = 0; i < field_length; ++i) {
                        for (int j = 0; j < field_width; ++j) {
                                out_file << array_data[i * field_width + j] << " ";
                        }
                }
                out_file.close();
        }

        // Calculate dest_process.
        if (!((coords[0] == dim_sizes[0] - 1) && (coords[1] == dim_sizes[1] - 1))) { //
Need to send.
                dub_coords[1] = coords[1] + 1;
                dub_coords[0] = coords[0];
                if (dub_coords[1] >= dim_sizes[1]) {
                        dub_coords[1] = 0;
                        dub_coords[0]++;
                }
                MPI_Cart_rank(cart_comm, dub_coords, &shift_dest);
                MPI_Send(&send_double, 1, MPI_DOUBLE, shift_dest, 5, cart_comm);
        }
}

// Default destructor.
Trans_solver::~Trans_solver(){
        // Delete cart data.
        if (dim_sizes != NULL) { delete dim_sizes; }
        if (periods != NULL) { delete periods; }
        if (coords != NULL) { delete coords; }
        if (dub_coords != NULL) { delete dub_coords; }

        // Delete field data.
        if (array_data != NULL) { delete array_data; }
        if (dub_array != NULL) { delete dub_array; }

        // Delete auxillary data fields.
        if (left_data_buffer != NULL) { delete left_data_buffer; }
        if (right_data_buffer != NULL) { delete right_data_buffer; }

        // Delete border data.
        if (upper_border != NULL) { delete upper_border; }
        if (lower_border != NULL) { delete lower_border; }
        if (left_border != NULL) { delete left_border; }
        if (right_border != NULL) { delete right_border; }

        // Delete tridiagonal coeffs data.
        if (alpha_coeff_x_axis != NULL) { delete alpha_coeff_x_axis; }
```

```
        if (beta_coeff_x_axis != NULL) { delete beta_coeff_x_axis; }
        if (charlie_coeff_x_axis != NULL) { delete charlie_coeff_x_axis; }

        if (alpha_coeff_y_axis != NULL) { delete alpha_coeff_y_axis; }
        if (beta_coeff_y_axis != NULL) { delete beta_coeff_y_axis; }
        if (charlie_coeff_y_axis != NULL) { delete charlie_coeff_y_axis; }

        // Delete coefficients data.
        if (F_coeff_x_axis != NULL) { delete F_coeff_x_axis; }
        if (F_coeff_y_axis != NULL) { delete F_coeff_y_axis; }

        // Delete sendrecv buffers data.
        if (send_buf != NULL) { delete send_buf; }
        if (recv_buf != NULL) { delete recv_buf; }

        // Delete global matrices data.
        if (global_x_mtrx != NULL) { delete global_x_mtrx; }
        if (global_y_mtrx != NULL) { delete global_y_mtrx; }

        // Delete global arrays data.
        if (global_x_array != NULL) { delete global_x_array; }
        if (global_y_array != NULL) { delete global_y_array; }
}

// Basic inclusions.
#include <cstdlib>
#include <cstdio>
#include <iostream>

// Mpi inclusions.
#include "mpi.h"

// Using directives.
using namespace std;

// Project inclusions.
#include "Trans_solver.cpp"

// Main function :)
// ------------------------------------------------------------------------------
int main(int argc, char* argv[]) {
        // MPI rank & size variables.
        int p_rank, p_size;

        // Command line arguments.
        int data_length = atoi(argv[1]);
        int data_width = atoi(argv[2]);
        int iter_num = atoi(argv[3]);
        int calc_mode = atoi(argv[4]);
        int out_mode = atoi(argv[5]);

        // Initialize MPI.
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &p_rank);
```

```
        MPI_Comm_size(MPI_COMM_WORLD, &p_size);

        // Time parameters.
        double start_time;
        double end_time;

        Trans_solver Eq_solver; // Create a solver.
        Eq_solver.Init(data_length, data_width, p_rank, p_size); // Initialize solver.
        Eq_solver.Fill_data(0); // Load test data to solver.

        start_time = MPI_Wtime();

        // Perform as many iterations as needed.
        // Perform as many iterations as needed.
        if (calc_mode == 0) { // Calculation mode.
                for (int i = 0; i < iter_num; ++i) {
                        Eq_solver.Iterate(i);
                }
        }
        else if (calc_mode == 1) { // Generate test solution
                Eq_solver.Fill_data(iter_num);
        }

        MPI_Barrier(MPI_COMM_WORLD);
        end_time = MPI_Wtime();

        double total_time = end_time - start_time;
        if (out_mode == 0) { // Calculation mode.
                if (p_rank == 0) {
                        cout << "Calculation time is : " << total_time << "\n";
                }
        }
        else { // Verify
                Eq_solver.Output(out_mode);
        }

        // Finalize MPI.
        MPI_Finalize();

        // Return :)
        return 0;
}
```