МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В.Ломоносова

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ

Практикум по курсу
" Суперкомпьютеры и параллельная обработка данных"

Разработка параллельной версии программы для задачи Adi2D

ОТЧЕТ

о выполненном задании
студента 320 учебной группы факультета ВМК МГУ

Мещанинова Вячеслава Павловича

Лектор: доцент, к.ф.-м.н. Бахтин Владимир Александрович

Москва, 2020 г

Оглавление

Постановка задачи

1. Реализовать параллельную версию программы для задачи Adi2D с помощью технологий параллельного программирования *OpenMP*.
2. Протестировать полученную программу на вычислительном комплексе *Polus*.
3. Исследовать эффективность полученной программы.
4. Исследовать масштабируемость полученной программы.
5. Построить графики зависимости времени исполнения от числа ядер/процессоров для различного объёма входных данных.
6. Для каждого набора входных данных найти количество ядер/процессоров, при котором время выполнения задачи перестаёт уменьшаться.
7. Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.

Описание алгоритма

1. Опишем полученную для распараллеливания последовательную программу:
   Исходный код содержал функции:

```
void init() – функция для инициализации массива
void relax()– основная функция релаксации(усреднения) массива
```

`void verify()` – функция для проверки правильности работы алгоритма

`int main()` – функция запуска алгоритма

Самой ресурсозатратной в плане выполняемых вычислений в предоставленной для распараллеливания программе является функции `void relax(), int main()`. Код этих функций предоставлен ниже:

```
void relax()
{
   for(j=1; j<=N-2; j++)
   for(i=1; i<=N-2; i++)
   {
      A[i][j] = (A[i-1][j]+A[i+1][j])/2.;
   }

   for(j=1; j<=N-2; j++)
   for(i=1; i<=N-2; i++)
   {
      double e;
      e=A[i][j];
      A[i][j] =(A[i][j-1]+A[i][j+1])/2.;
      eps=Max(eps,fabs(e-A[i][j]));
   }
}

int main(int an, char **as)
{
   int it;
   init();

   for(it=1; it<=itmax; it++)
   {
      eps = 0.;
      relax();
      printf( "it=%4i   eps=%f\n", it,eps);
      if (eps < maxeps) break;
   }

   verify();

   return 0;
}
```

В закрашенной части программы в худшем случае выполняются itmax раз два двойных цикла, при этом обход матрицы происходит по столбцам, что крайне не эффективно, так как вызываемые ячейки памяти расположены не последовательно.

Также “неприятным моментом” является то, что при вычислениях в первом цикле обращение к элементам массивов происходит к соседним элементам по нулевому измерению – при больших объемах данных это может плохо

сказаться, так как вся матрица в КЭШ не поместится.

Сложность данного алгоритма составляет:

$$itmax * n^2$$

При n = 2000 и t = 1000 – самые большие значения, на которых будет производиться тестирование, в данной будет произведено:

$$itmax * n^2 = 1000 * 2000^2 = 4 * 10^9 = 4 \text{ млрд операций.}$$

Изложив основные аспекты предоставленной для распараллеливания программы, перейдем к описанию предпринятых к ней модификациям и эвристикам.

Предпринятые модификации к последовательной программе

Предоставленная последовательная программа имела ряд дефектов, которые желательно устранить перед разработкой параллельной версии программы:

1) Нехватка код-стайла, мне кажется, что намного проще поддерживать программу, когда на нее приятно смотреть.
2) Программа была переписана на C++, чтобы использовать все возможности языка.
3) Также я написал обертку, которая подсчитывает время выполнения любого метода – плюс модульности программы:

```
template<class Function>
double MeasureTime(Function func) {
    double start = omp_get_wtime();
    IterateRelax(func);
    double end = omp_get_wtime();
    return end - start;
}
```

4) Выделение памяти под массивы происходит на стеке, что в данной задаче не критично, так как ограничения по времени выполнения не предоставляют возможности выделения больших матриц, но в реальной жизни часто требуется

работать с огромными матрицами, которые все-таки лучше выделять в динамической памяти:

```
void Init() {
    data = new double[SIZE * SIZE];
    for (int i = 0; i <= SIZE - 1; i++) {
        for (int j = 0; j <= SIZE - 1; j++) {
            if (i == 0 || i == SIZE - 1 || j == 0 || j == SIZE - 1) {
                data[i * SIZE + j] = 0;
            } else {
                data[i * SIZE + j] = 1 + i + j;
            }
        }
    }
}

void Clear() {
    delete[] data;
}
```

5) Также для удобства проверки правильности работы параллельной программы была реализована функция печати массива:

```
void PrintMatrix() {
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
            std::cout << data[i * SIZE + j] << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

Реализация параллельной версии программы

Код проекта, вспомогательных скриптов и отчета доступны на https://gitlab.com/-/ide/project/msu-cmc/super-computer-practice

Было реализовано/переписано 3 версии алгоритма:

- Relax – базовый метод без параллелизма (см. impl_openmp.cpp)
- RelaxParallelOpenMP - оптимизация с использованием thread-параллелизма (см. impl_openmp.cpp)
    - Для OpenMP реализации использовались директивы parallel, for и reduction, private, shared
- RelaxMPI - оптимизация с использованием процессорного параллелизма (см. impl_mpi.cpp)
    - Для MPI версии использовались команды MPI_Bcast, MPI_Reduce, MPI_Barrier, MPI_Comm_size, MPI_Comm_rank, MPI_Scatterv, MPI_Gatherv, MPI_Wtime, MPI_Init, MPI_Finalize

Тестирование

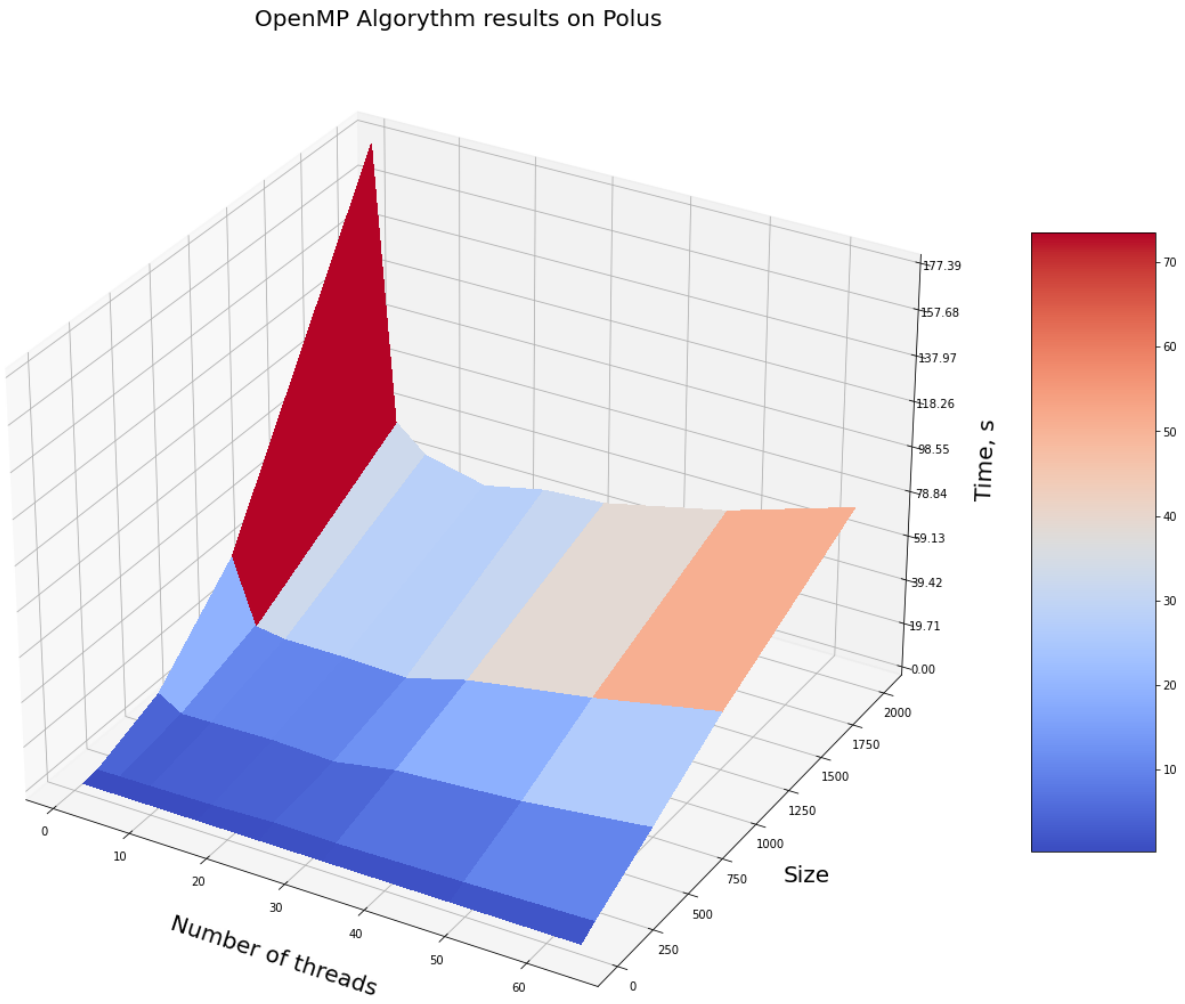
Тестирование проводилось на суперкомпьютере – *Polus*.

Polus – параллельная вычислительная система, состоящая из 5 вычислительных узлов. Один вычислительный узел содержит 2 десятиядерных процессора IBM POWER8 (каждое ядро имеет 8 потоков) всего 160 потоков.

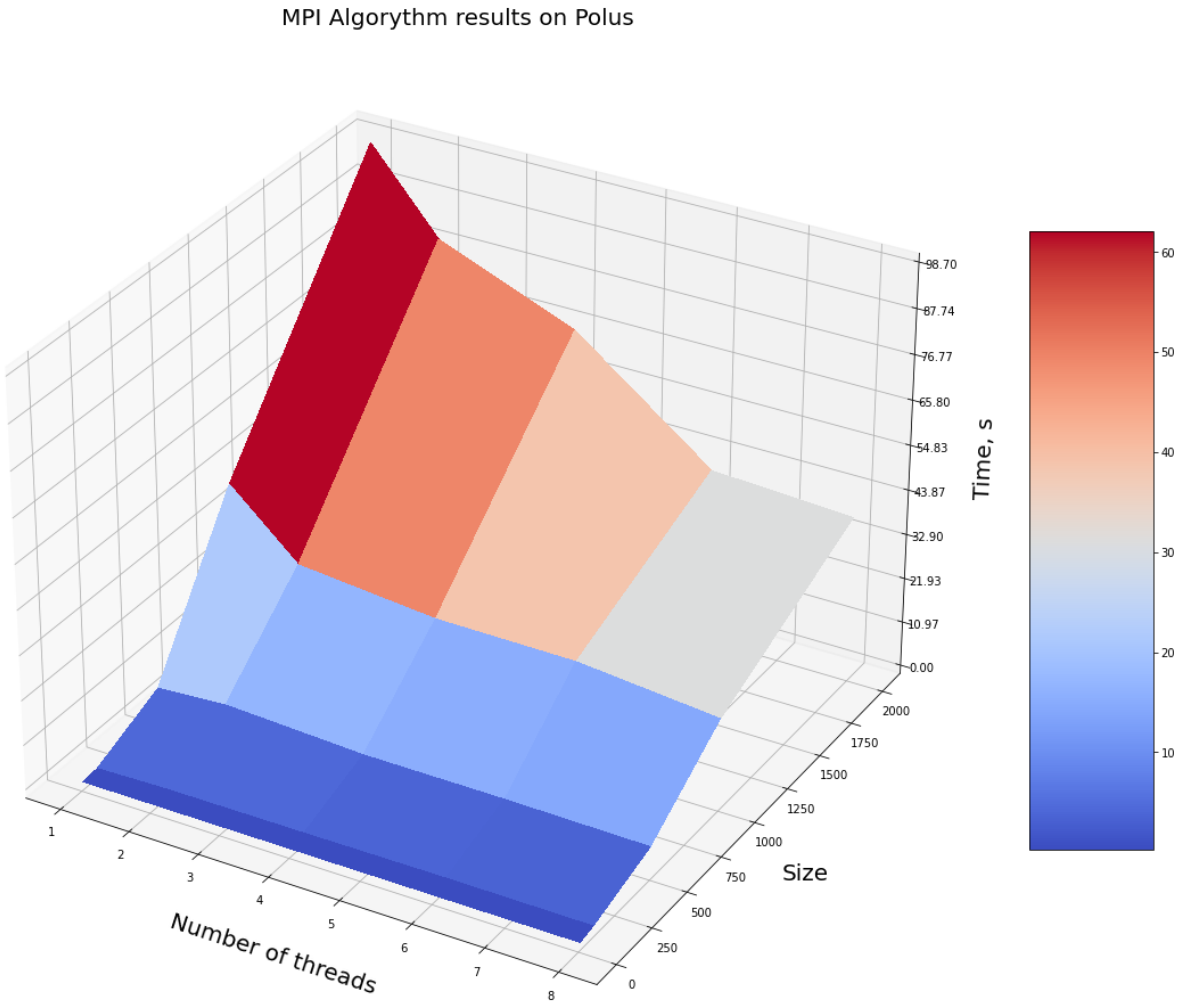
Итак, программа тестировалась на вычислительной машине *Polus* со следующим числом потоков – 1, 4, 8, 16, 32, 48, 64.

Для замера времени использовались функции omp_get_wtime() и MPI_Wtime() и за время выполнения принималось время работы релаксации матрицы. При этом для каждого размера и для каждого числа потоков проводилось от 5 до 10 прогонов. Так как для детерминированной процедуры невозможно случайное ускорение программы, а только лишь случайное замедление, то бралось минимальное время по всем прогонам.

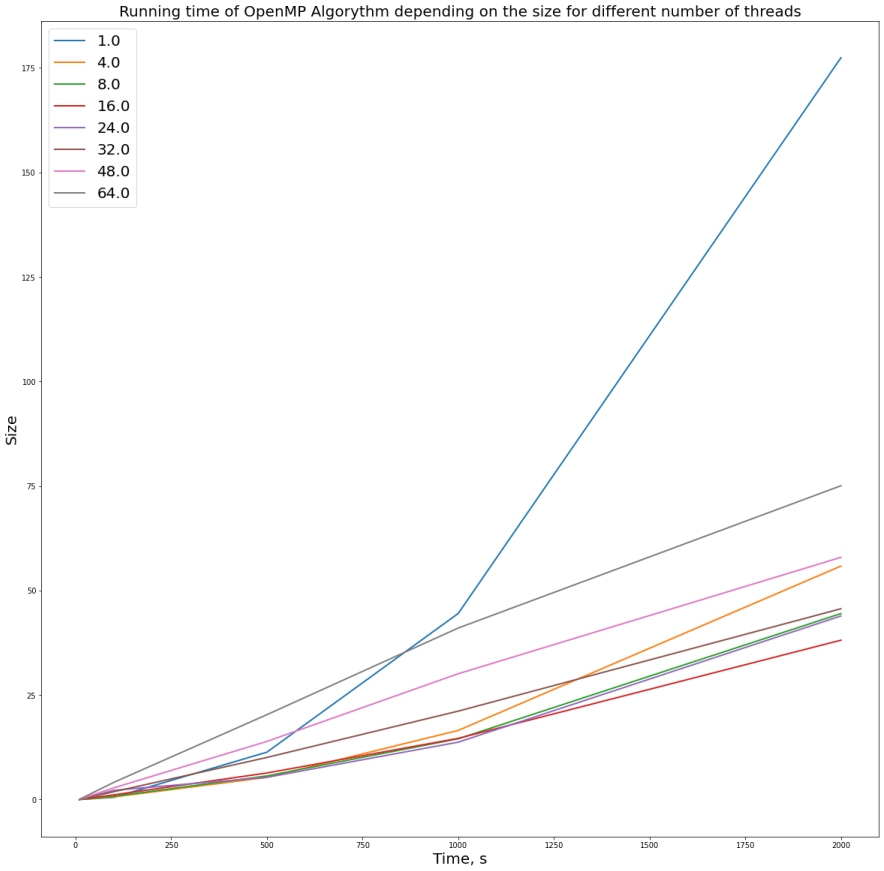
На графике можно увидеть, что не всегда с увеличением числа потоков увеличивается скорость программы. Например, для OpenMP версии программы 32 – это то число потоков, при котором достигается минимум времени работы алгоритма. Более того время выполнения может немного увеличиться, я объяснял это накладными расходами на создание потоков. Также я делаю вывод, что для маленьких размеров матрицы лучше использовать небольшое число процессов – 1 или 2, для больших же 32 – оптимальное количество.

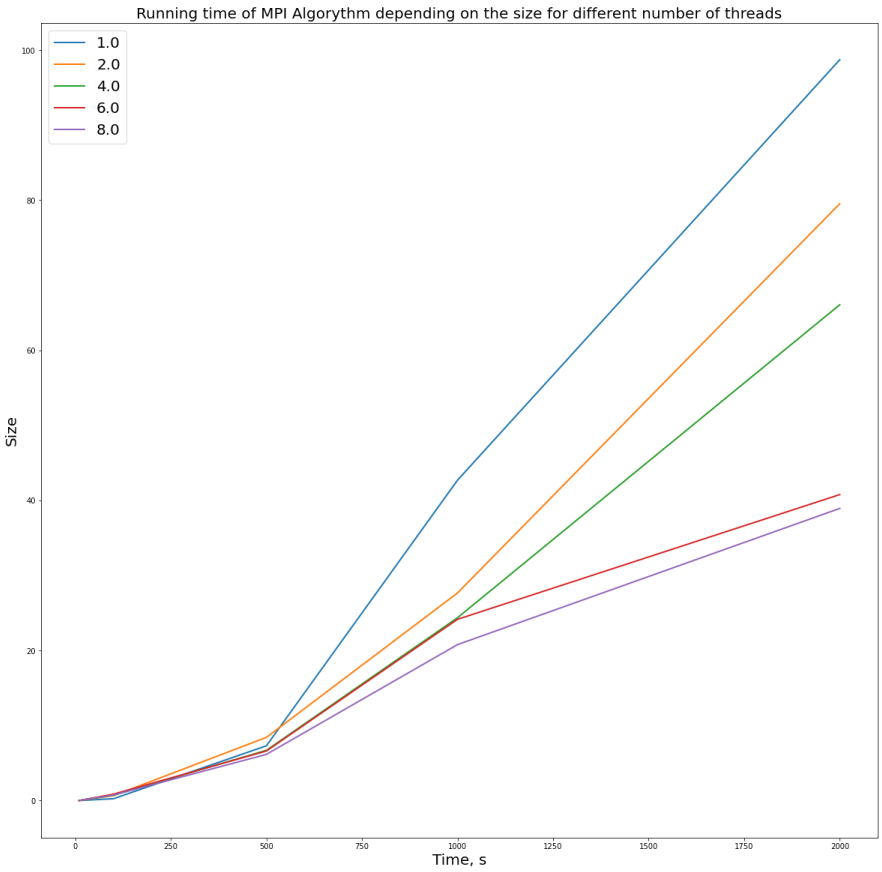


MPI- версия программы была протестирована на вычислительной машине Polus со следующим числом процессов – 1, 2, 4, 6, 8. В данном случае максимальное ускорение программы было при числе процессов, равном 8. При этом можно сделать тот же вывод, что для небольших размеров матрицы стоит использовать 1 процесс.



Также для определения оптимального числа процессов и нитей были отрисованы следующие графики для обеих версий программы:





Листинг кода программы

- Impl_openmp.cpp

```
#include <iostream>
#include <omp.h>
#include <cmath>

#define ROOT 0
int SIZE = 100;

int _comm_rank = 0;
int  comm size = 1;

double global_eps = 0;
double maxeps = 0.1e-7;
int itmax = 1000;

double *data;

inline int Ind(int i, int j) {
    return i * SIZE + j;
}

void Init() {
    data = new double[SIZE * SIZE];
    for (int i = 0; i <= SIZE - 1; i++) {
        for (int j = 0; j <= SIZE - 1; j++) {
            if (i == 0 || i == SIZE - 1 || j == 0 || j == SIZE - 1) {
                data[i * SIZE + j] = 0;
            } else {
                data[i * SIZE + j] = 1 + i + j;
            }
        }
    }
}

void PrintMatrix() {
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
            std::cout << data[i * SIZE + j] << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

template<class Function>
void IterateRelax(Function func) {
    for (int it = 1; it <= itmax; it++) {
        func();
        //printf("it=%4i   eps=%f\n", it, global_eps);
        //PrintMatrix();
        if (global_eps < maxeps) {
            break;
        }
        global_eps = 0;
    }
}

void Relax() {
    int i, j;
```

```
        for (i = 1; i <= SIZE - 2; i++) {
            for (j = 1; j <= SIZE - 2; j++) {
                data[i * SIZE + j] = (data[(i - 1) * SIZE + j] + data[(i + 1)
* SIZE + j]) / 2.;
            }
        }

        for (i = 1; i <= SIZE - 2; i++) {
            for (j = 1; j <= SIZE - 2; j++) {
                double e;
                e = data[i * SIZE + j];
                data[i * SIZE + j] = (data[i * SIZE + j - 1] + data[i * SIZE +
j + 1]) / 2.;
                global_eps = std::max(global_eps, fabs(e - data[Ind(i, j)]));
            }
        }

    }

    void RelaxParallelOpenMP() {
        int i, j;
        for (i = 1; i <= SIZE - 2; i++) {
    #pragma omp parallel for private(j) shared(i)
            for (j = 1; j <= SIZE - 2; j++) {
                data[i * SIZE + j] = (data[(i - 1) * SIZE + j] + data[(i + 1)
* SIZE + j]) / 2.;
            }
        }

        for (j = 1; j <= SIZE - 2; j++) {
    #pragma omp parallel for private(i) shared(j) reduction(max: global_eps)
            for (i = 1; i <= SIZE - 2; i++) {
                double e;
                e = data[i * SIZE + j];
                data[i * SIZE + j] = (data[i * SIZE + j - 1] + data[i * SIZE +
j + 1]) / 2.;
                global_eps = std::max(global_eps, fabs(e - data[Ind(i, j)]));
            }
        }
    }

    template<class Function>
    double MeasureTime(Function func) {
        double start = omp_get_wtime();
        IterateRelax(func);
        double end = omp_get_wtime();
        return end - start;
    }

    void Clear() {
        delete[] data;
    }

    template<class Function>
    double RunOneTest(std::ostream &fout, Function func, int size) {
        double min time = 3600;
        SIZE = size;
        int min_cycle = 3;
        for (int i = 1; i <= min_cycle; ++i) {
            Init();
            min_time = std::min(min_time, MeasureTime(func));
            Clear();
        }
        return min_time;
```

```
}


int main(int argc, char *argv[]) {
    for (int num_threads: {1, 4, 8, 16, 24, 32, 48, 64}) {
        omp_set_num_threads(num_threads);
        std::cout << num_threads << std::endl;
        auto Sizes = {10, 100, 500, 1000, 2000, 5000};
        for (int size: Sizes) {
            std::cout << size << " " << RunOneTest(std::cout,
RelaxParallelOpenMP, size) << std::endl;
        }
        std::cout << std::endl;
    }
}
```

- impl_mpi.cpp

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <vector>
#include <omp.h>
#include <fstream>
#include <cmath>

#define ROOT 0
int SIZE = 500;

int _comm_rank = 0;
int _comm_size = 1;

double global_eps = 0;
double maxeps = 0.1e-7;
int itmax = 1000;

double *data;

int Ind(int i, int j) {
    return i * SIZE + j;
}

void Init() {
    data = new double[SIZE * SIZE];
    for (int i = 0; i <= SIZE - 1; i++) {
        for (int j = 0; j <= SIZE - 1; j++) {
            if (i == 0 || i == SIZE - 1 || j == 0 || j == SIZE - 1) {
                data[Ind(i, j)] = 0;
            } else {
                data[Ind(i, j)] = 1 + i + j;
            }
        }
    }
}

void Clear() {
    delete[] data;
}

void PrintMatrix() {
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
```

```
            std::cout << data[Ind(i, j)] << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

template<class Function>
void IterateRelax(Function func) {
    for (int it = 1; it <= itmax; it++) {
        func();
        MPI_Bcast(&global_eps, 1, MPI_DOUBLE, ROOT, MPI_COMM_WORLD);
        if (global_eps < maxeps) {
            break;
        }
        global_eps = 0;
    }
}

void RelaxMPI() {
    MPI_Comm_size(MPI_COMM_WORLD, &_comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &_comm_rank);

    /*----------------- Вертикальное распараллеливание ------------------
*/
    {
        /*----- Инициализация массивов для MPI_Scatterv и MPI_Gatherv ----
--*/
        auto *num_col_per_proc = new int[_comm_size];
        auto *displc = new int[_comm_size];

        num_col_per_proc[0] = (SIZE / _comm_size + (0 < SIZE %
_comm_size));
        displc[0] = 0;

        for (int i = 1; i < _comm_size; ++i) {
            num_col_per_proc[i] = num_col_per_proc[i - 1];
            if (i == SIZE % _comm_size) {
                --num_col_per_proc[i];
            }
            displc[i] = displc[i - 1] + num_col_per_proc[i - 1];
        }
        /*-----------------------------------------------------------------
---*/

        int tmp_sz = num_col_per_proc[_comm_rank];
        auto *thread_data = new double[SIZE * tmp_sz];

        for (int i = 0; i < SIZE; ++i) {
            MPI_Scatterv(data + i * SIZE, num_col_per_proc, displc,
MPI_DOUBLE,
                         thread_data + i * tmp_sz, tmp_sz, MPI_DOUBLE,
ROOT,
                         MPI_COMM_WORLD);
        }

        for (int i = 1; i < SIZE - 1; ++i) {
            for (int j = 0; j < tmp_sz; ++j) {
                thread_data[i * tmp_sz + j] =
                        (thread_data[(i - 1) * tmp_sz + j] +
thread_data[(i + 1) * tmp_sz + j]) / 2;
            }
        }
```

```
        for (int i = 0; i < SIZE; ++i) {
            MPI_Gatherv(thread_data + i * tmp_sz, tmp_sz, MPI_DOUBLE, data
+ i * SIZE, num_col_per_proc, displc,
                        MPI_DOUBLE,
                        ROOT,
                        MPI_COMM_WORLD);
        }

        delete[] thread_data;
        delete[] displc;
        delete[] num_col_per_proc;
    }
    /*-------------------------------------------------------------------
*/


    /*---------------- Горизонтальное распараллеливание -----------------
*/
    {
        /*----- Инициализация массивов для MPI_Scatterv и MPI_Gatherv ----
--*/
        auto *num_str_per_proc = new int[_comm_size];
        auto *displc = new int[_comm_size];

        num_str_per_proc[0] = (SIZE / _comm_size + (0 < SIZE %
_comm_size)) * SIZE;
        displc[0] = 0;

        for (int i = 1; i < _comm_size; ++i) {
            num_str_per_proc[i] = num_str_per_proc[i - 1];
            if (i == SIZE % _comm_size) {
                num_str_per_proc[i] -= SIZE;
            }
            displc[i] = displc[i - 1] + num_str_per_proc[i - 1];
        }
        /*-------------------------------------------------------------------
---*/

        auto *thread_data = new double[num_str_per_proc[_comm_rank]];
        double eps = 0;
        MPI_Scatterv(data, num_str_per_proc, displc, MPI_DOUBLE,
thread_data, num_str_per_proc[_comm_rank], MPI_DOUBLE,
                     ROOT, MPI_COMM_WORLD);
        for (int i = 0; i < num_str_per_proc[_comm_rank] / SIZE; ++i) {
            for (int j = 1; j < SIZE - 1; ++j) {
                double tmp = thread_data[i * SIZE + j];
                thread_data[i * SIZE + j] = (thread_data[i * SIZE + j - 1]
+ thread_data[i * SIZE + j + 1]) / 2;
                eps = std::max(eps, fabs(tmp - thread_data[i * SIZE +
j]));
            }
        }
        MPI_Reduce(&eps, &global_eps, 1, MPI_DOUBLE, MPI_MAX, ROOT,
MPI_COMM_WORLD);
        MPI_Gatherv(thread_data, num_str_per_proc[_comm_rank], MPI_DOUBLE,
data, num str per proc, displc, MPI DOUBLE,
                    ROOT,
                    MPI_COMM_WORLD);
        delete[] thread_data;
        delete[] displc;
        delete[] num_str_per_proc;
    }
    /*-------------------------------------------------------------------
*/
```

```
}


template<class Function>
double MeasureTime(Function func) {
    double start = MPI_Wtime();
    IterateRelax(func);
    MPI_Barrier(MPI_COMM_WORLD);
    double end = MPI_Wtime();
    return end - start;
}


template<class Function>
double RunOneTest(std::ostream &fout, Function func, int size) {
    SIZE = size;
    if (ROOT == _comm_rank) {
        Init();
    }
    double time = MeasureTime(func);
    if (ROOT == _comm_rank) {
        Clear();
    }
    return time;
}

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int size = 100;
    if (argc >= 2) {
        size = std::stoi(argv[1]);
    }
    double time = RunOneTest(std::cout, RelaxMPI, size);
    if (ROOT == _comm_rank) {
        std::cout << size << " " << time << std::endl;
    }
    MPI_Finalize();
    return 0;
}
```

Приложение

Вместе с отчетом прикладываю файлы:

- OpenMP-версия программы: *impl_openmp.cpp*
- MPI-версия программы: *impl_mpi.cpp*
- Ноутбук с построением графиков: *Visualisation.ipynb*