МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ

«Применение неблокирующих операций MPI»

студента 2 курса, группы 20205

Муратова Максима Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
доцент
Власенко А. Ю.

Новосибирск, 2021

СОДЕРЖАНИЕ

ЦЕЛИ

1. Найти зависимость времени выполнения параллельной программы от числа процессов, её выполняющих;
2. Обмен данными организовать неблокирующими операциями;
3. При помощи профилирования сделать анализ эффективности использовзания времени выполнения программы.

ЗАДАНИЕ

1. Вычислительное задание – реализовать клеточный автомат «Жизнь» на поле N x M со склеенными краями. Начальное положение – 1 «глайдер» в левом верхнем углу. Условие завершения программы – повторение какой-либо из предыдущих позиций;
2. N=480, M=470;
3. Число процессов – все целые числа в диапазоне от 1 до 24.
4. Профилировщиком Intel Trace Analyzer and Collector (ITAC) пропрофилировать программу при 4, 6, 8, 10, 12, 14, 16, 18 и 20 процессах.

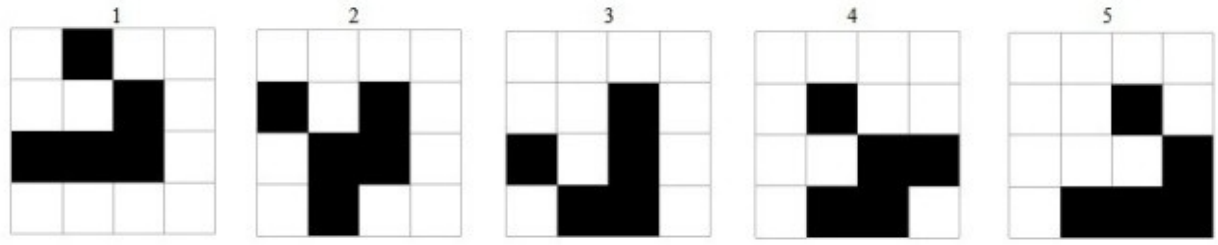
ОПИСАНИЕ РАБОТЫ

Исходный код программы (приложение 1) распределён между следующими файлами:

- `life.cpp` – исходный код основной программы;
- `field.h` – заголовочный файл, который содержит описание класса FieldHistory, который сохраняет в себе предыдущие состояния поля;
- `field.cpp` – реализация методов данного класса.

Программа собирается утилитой `make`, конфигурация `Makefile` описана в приложении 2.

Правила клеточного автомата «Жизнь» просты. Клетка либо живая, либо мёртвая. У неё есть 8 соседей. Если клетка живая, то она перейдёт в следующее поколение, если у неё 2 или 3 «живых» соседа, иначе – гибель от «перенаселения» или «одиночества». Если же клетка мёртвая, то она «оживёт» в следующем поколении, если вокруг неё ровно 3 «живых» соседа. Последующие поколения сменяются по тем же правилам.



За 4 поколения «глайдер» (позиция 1 на рис. выше) воспроизведёт себя в положении на 1 ряд ниже и на 1 столбец правее, чем был до этого. Чтобы позиция повторилась, должно пройти наименьшее количество сдвигов такое, что оно кратно и числу строк, и числу столбцов. Иначе говоря, это число равно НОК(N, M), тогда число поколений для повторения начальной позиции при заданных размерах поля составит 4 * НОК(480, 470) = 90 240.

Очень внимательно был рассмотрен вопрос, как сохранять предыдущие состояния поля и как сравнивать их между собой. Данный вопрос был делегирован классу FieldHistory. Были применены следующие оптимизации:

1. Отбрасывание повторившихся состояний. Если состояние фрагмента поля повторилось с каким-либо из предыдущих поколений, то в историю запишется не сам фрагмент поля, а номер поколения, с которым произошло совпадение. Это позволяет сэкономить память,

особенно в случае использования большого количества процессорных ядер, а также уменьшить количество сравнений фрагментов полей.

2. Предварительное сравнение по первой «живой» клетке. После первой оптимизации обозначилась проблема: нулевому процессу требовалось слишком много времени, чтобы фиксировать повторы пустого состояния поля, так как в его «исторической базе данных» пустое состояние поля не первое, как у других процессов. Поэтому было решено улучшить алгоритм сравнения фрагментов поля: прежде чем как начать сравнивать новое состояние поля со старыми состояниями, у него сначала ищется адрес первой «живой» клетки. У старых состояний их адреса первых «живых» клеток были посчитаны ещё тогда, когда их добавляли в базу состояний поля, так что повторный поиск адресов их первых «живых» клеток не проводится. Теперь, если при сравнении состояний поля у них не совпали адреса их первых «живых» клеток, то их сравнение автоматически пропускается, если же совпали, то состояния сравниваются после данной клетки в обычном режиме. Смысл оптимизации в том, что поиском первой «живой» клетки мы частично сравнили новое состояние сразу со всеми предыдущими. Развитие данной идеи в вектор, в котором хранились бы количества подряд идущих «живых» и «мёртвых» клеток, не обвенчалось успехом, так как проигрывало по времени, хотя и сильно выигрывала по памяти. Данная оптимизация ускорила программу в 150 раз в случае небольшого количества процессорных ядер.

Тем не менее, повторной профилировкой было установлено, что с ростом числа процессов начинают расти временные расходы на проверку флагов состояния. Это привело к тому, что после 8 процессов время выполнения начинало только увеличиваться. Данную ситуацию удалось исправить тем предположением, что на каждом процессе можно хранить не все флаги останова, а только те, которые сравниваются с текущим состоянием поля (сам процесс проверки флагов был распараллелен ещё до обнаружения данной проблемы). Оптимизация сказывается при числе процессорных ядер, не меньшем 6. На 24 ядрах за счёт распараллеливания было получено ускорение в 9.73 раза по сравнению с 1 ядром. Со всеми результатами замерами времени выполнения можно ознакомиться в приложении 4.

После всех оптимизаций была повторно проведено профилирование программы на 4, 8, 12, 16 и 20 ядрах. На 24 ядрах профилирование не удалось, потому что файлы с результатами профилирования при таком большом количестве ядер оказывались слишком большими и не позволяли себя открыть. В остальных случаях был проведён анализ: сколько времени тратилось на изменение состояния поля, и на сравнение флагов?

При любом количестве ядер тратилось примерно одинаковое количество времени на изменение состояния игрового поля, хотя ближе к завершению программы наблюдалось сокращение времени выполнения. Важно отметить, что это время обратно пропорционально количеству ядер.

Диаграммы профилирования доступны в приложении 5.

С флагами всё гораздо проще. Так как у каждого процесса свой массив флагов состояния, то количество флагов пропорционально количеству процессов. Можно временно получить ускорение благодаря тому, что увеличение числа процессов означает больше проверок флагов в пределах одного вектора, которое может быть досрочно завершено из-за несоответствия хотя бы одного флага, однако такое ускорение будет быстро нивелировано расходами на коммуникации Allgather и Allreduce по той же причине: рост число коммутирующих процессов.

ЗАКЛЮЧЕНИЕ

1. Максимальное ускорение программы за счёт её распараллеливания составило 9.73 раза на 24 ядрах;
2. Дальнейшее уменьшение времени выполнения программы проблематично из-за других оптимизаций, которые ускорили программу до предела;
3. Основной сдерживающий фактор ускорения программы – при росте числа процессов увеличиваются расходы на проверку флагов состояния из-за увеличения тяжеловесности коммутиурующих операций.

Приложение 1. *Исходный код программы*

life.cpp

```
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <vector>
#include "field.h"

#define correct_argc 2

void inline next_generation_in_row(const cell_t *old_generation, const
           cell_t *row_below, const cell_t *row_above, cell_t
           *new_generation, int row_size) {
        int neighbour_count = old_generation[1] +
                         old_generation[row_size - 1];
        neighbour_count += row_below[row_size - 1] +
                    row_below[0] + row_below[1];
        neighbour_count += row_above[row_size - 1] +
                    row_above[0] + row_above[1];
        new_generation[0] = neighbour_count == 3 ||
                    (neighbour_count == 2 && old_generation[0]);

        neighbour_count = old_generation[0] +
                    old_generation[row_size - 2];
        neighbour_count += row_below[row_size - 2] +
                    row_below[row_size - 1] + row_below[0];
        neighbour_count += row_above[row_size - 2] +
                    row_above[row_size - 1] + row_above[0];
        new_generation[row_size - 1] = neighbour_count == 3 ||
                    (neighbour_count == 2 &&
                    old_generation[row_size - 1]);

        for(int index = 1; index < row_size - 1; index++) {
                neighbour_count = old_generation[index - 1] +
                                old_generation[index + 1];
                neighbour_count += row_below[index - 1] +
                                row_below[index] + row_below[index + 1];
                neighbour_count += row_above[index - 1] +
                                row_above[index] + row_above[index + 1];
                new_generation[index] = neighbour_count == 3 ||
                                (neighbour_count == 2 &&
                                old_generation[index]);
        }
}
```

```
int main(int argc, char **argv) {
        if(argc != correct_argc + 1) {
                std::cerr << "got " << argc - 1 «
                    " arguments, expected ";
                std::cerr << correct_argc << '\n';
                return 1;
        }

        int height = atoi(argv[1]);
        int width = atoi(argv[2]);
        MPI_Init(&argc, &argv);
        double begin_time = MPI_Wtime();
        int rank, rank_below, rank_above, process_count, row_count;
        MPI_Comm_size(MPI_COMM_WORLD, &process_count);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        rank_below = (rank + 1) % process_count;
        rank_above = (rank + process_count - 1) % process_count;
        row_count = height * (rank + 1) / process_count - height * rank /
                    process_count;
        cell_t *sub_field = new cell_t[row_count * width];

        MPI_Datatype row_t;
        MPI_Type_contiguous(width, MPI_INT8_T, &row_t);
        MPI_Type_commit(&row_t);

        const int root = 0;
        if(rank == root) {
                cell_t *init_field = new cell_t[height * width];
                for(int index = 0; index < height * width; index++)
                        init_field[index] = 0;
                add_glider(0, 0, init_field, height, width);
                int *counts = new int[process_count];
                int *offsets = new int[process_count];
                offsets[0] = 0; counts[process_count - 1] = height -
                        height * (process_count - 1) / process_count;
                for(int index = 1; index < process_count; index++) {
                        offsets[index] = height * index / process_count;
                        counts[index - 1] = offsets[index] -
                                                offsets[index - 1];
                }

                MPI_Scatterv(init_field, counts, offsets, row_t,
                     sub_field, row_count, row_t, root, MPI_COMM_WORLD);

                delete[] init_field;
                delete[] counts;
                delete[] offsets;
        }
        else MPI_Scatterv(NULL, NULL, NULL, row_t, sub_field, row_count,
```

```
                    row_t, root, MPI_COMM_WORLD);

        cell_t *buff_new_gen = new cell_t[row_count * width];
        cell_t *row_below = new cell_t[width];
        cell_t *row_above = new cell_t[width];

        int iteration;

        FieldHistory checker{row_count * width};
        std::vector<int> *flags = new std::vector<int>[process_count];
        int *last_flags = new int[process_count];

        MPI_Request border_requests[4];
        MPI_Status border_statuses[4];
        bool is_added;
        int last_my_flag;

        for(iteration = 0;; iteration++) {
                MPI_Isend(sub_field, 1, row_t, rank_above, rank * 2,
                                MPI_COMM_WORLD, border_requests);
                MPI_Isend(sub_field + width * (row_count - 1), 1, row_t,
                                rank_below, rank * 2 + 1, MPI_COMM_WORLD,
                                border_requests + 2);
                MPI_Irecv(row_above, 1, row_t, rank_above,rank_above *
                          2 + 1, MPI_COMM_WORLD, border_requests + 1);
                MPI_Irecv(row_below, 1, row_t, rank_below, rank_below *
                                2, MPI_COMM_WORLD, border_requests + 3);

                is_added = checker.add(sub_field);
                last_my_flag = checker.last_flag();

                for(int index = 1; index < row_count - 1; index++)
                        next_generation_in_row(sub_field + width * index,
                                        sub_field + width * (index - 1),
                                        sub_field + width * (index + 1),
                                        buff_new_gen + width * index,
                                        width);

                MPI_Waitall(2, border_requests + 1, border_statuses + 1);
                next_generation_in_row(sub_field, sub_field + width,
                                row_above, buff_new_gen, width);

                MPI_Wait(border_requests, border_statuses);
                MPI_Wait(border_requests + 3, border_statuses + 3);
                next_generation_in_row(sub_field + width *
                                (row_count - 1), row_below,
                                sub_field + width * (row_count - 2),
                                buff_new_gen + width * (row_count – 1),
                                width);
```

```
                if(is_added) {
                        sub_field = buff_new_gen;
                        buff_new_gen = new cell_t[row_count * width];
                }
                else std::swap(buff_new_gen, sub_field);

                MPI_Allgather(&last_my_flag, 1, MPI_INT, last_flags, 1,
                                        MPI_INT, MPI_COMM_WORLD);
                bool is_repeated = false;

                for(size_t index = 0; !is_repeated && index <
                                flags[0].size(); index++) {
                        is_repeated = true;
                        for(int sub_rank = 0; sub_rank < process_count &&
                                        is_repeated; sub_rank++) {
                              is_repeated = flags[sub_rank][index] ==
                                                last_flags[sub_rank];
                        }
                }
                bool is_repeated_somewhere = false;
                MPI_Allreduce(&is_repeated, &is_repeated_somewhere, 1,
                         MPI_INT8_T, MPI_SUM, MPI_COMM_WORLD);
                if(is_repeated_somewhere)
                        break;
                if(iteration % process_count == rank) {
                        for(int index = 0; index < process_count;
                                                index++)
                            flags[index].push_back(last_flags[index]);
                }
        }

        double end_time = MPI_Wtime();

        if(rank == root) {
                std::cerr << end_time - begin_time << '\n';
                std::cout << iteration + 1 << '\n';
        }

        delete[] sub_field;
        delete[] row_below;
        delete[] row_above;
        delete[] buff_new_gen;
        delete[] flags;
        delete[] last_flags;
        MPI_Type_free(&row_t);
        MPI_Finalize();
        return 0;
}
```

field.h

```
#ifndef H_FIELD
#define H_FIELD

#include <vector>
typedef bool cell_t;

class FieldHistory {
        std::vector<cell_t*> snapshots;
        std::vector<int> factor_set;
        std::vector<int> first_alive_cells;
        int buffer_size;

        bool compare(const cell_t *a, int first_alive_a, const cell_t *b,
                        int first_alive_b) const;
        int first_alive(const cell_t *field) const;

public:
        FieldHistory(int);
        ~FieldHistory();

        bool add(cell_t *snapshot);
        int last_flag() const;
};


void add_glider(int row, int column, cell_t *field, int height,
             int width);

#endif //H_FIELD
```

field.cpp

```
#include "field.h"
#include <cstdlib>

void add_glider(int row, int column, cell_t *field, int height,
       int width) {
        field[row * width + column + 1] =
                field[(row + 1) * width + column + 2] = 1;
        for(int index = 0; index < 3; index++)
                field[(row + 2) * width + column + index] = 1;
}

int FieldHistory::first_alive(const cell_t *field) const {
        for(int index = 0; index < buffer_size; index++)
                if(field[index] == 1) return index;
```

```
        return buffer_size;
}

FieldHistory::FieldHistory(int data_size):
        buffer_size(data_size) {}

FieldHistory::~FieldHistory() {
        for(size_t index = 0; index < snapshots.size(); index++)
                delete[] snapshots[index];
}

bool FieldHistory::compare(const cell_t *a, int first_alive_a,
       const cell_t *b, int first_alive_b) const {
        if(a == nullptr || b == nullptr)
                return false;
        if(first_alive_a != first_alive_b)
                return false;
        for(int index = first_alive_a + 1; index < buffer_size; index++)
                if(a[index] != b[index])
                        return false;
        return true;
}



bool FieldHistory::add(cell_t *snapshot) {
        int first = first_alive(snapshot);
        for(size_t index = 0; index < snapshots.size(); index++) {
                if(compare(snapshots[index], first_alive_cells[index],
                                        snapshot, first)) {
                        factor_set.push_back(index);
                        return false;
                }
        }
        factor_set.push_back(snapshots.size());
        snapshots.push_back(snapshot);
        first_alive_cells.push_back(first);
        return true;
}

int FieldHistory::last_flag() const {
        return factor_set.back();
}
```

Приложение 2. `Makefile`

Файл сборки проекта

```
CXX=icpc
CXXP=mpiicpc
CFLAGS=-c -std=c++14 -O3 -Wall

FIELD_SOURCE=field
LIFE_SOURCE=life

all: $(LIFE_SOURCE)

$(LIFE_SOURCE): $(LIFE_SOURCE).o $(FIELD_SOURCE).o
	$(CXXP) $(LIFE_SOURCE).o $(FIELD_SOURCE).o -o $(LIFE_SOURCE)

$(LIFE_SOURCE).o: $(LIFE_SOURCE).cpp $(FIELD_SOURCE).h
	$(CXXP) $(CFLAGS) $(LIFE_SOURCE).cpp -o $(LIFE_SOURCE).o

$(FIELD_SOURCE).o: $(FIELD_SOURCE).cpp $(FIELD_SOURCE).h
	$(CXX) $(CFLAGS) $(FIELD_SOURCE).cpp -o $(FIELD_SOURCE).o

clean:
	rm -rf *.o
```

Приложение 3. *Скрипты для запуска на кластере*

- **run.sh**

```
#!/bin/bash

#PBS -l walltime=00:33:00
#PBS -l select=2:ncpus=12:mpiprocs=12:mem=24000m,place=scatter
#PBS -m n


cd $PBS_O_WORKDIR

MPI_NP_GOT=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')
echo "Number of MPI processes: $MPI_NP_GOT"


echo 'File $PBS_NODEFILE:'
cat  $PBS_NODEFILE
HEIGHT=480
WIDTH=470
REPORT_FILE=report.txt
CHECK_FILE=check.txt

for(( MPI_NP = 1; $MPI_NP <= $MPI_NP_GOT; MPI_NP++ )); do
        for(( ITER = 0; $ITER < 4; ITER++ )); do
                FORMAT="$HEIGHT\t$WIDTH\t$MPI_NP\t"
                echo -ne "$FORMAT" >>$REPORT_FILE
                echo -ne "$FORMAT" >>$CHECK_FILE
                mpirun -machinefile $PBS_NODEFILE -np $MPI_NP \
                    ./life $HEIGHT $WIDTH >>$CHECK_FILE 2>>$REPORT_FILE
        done
done
```

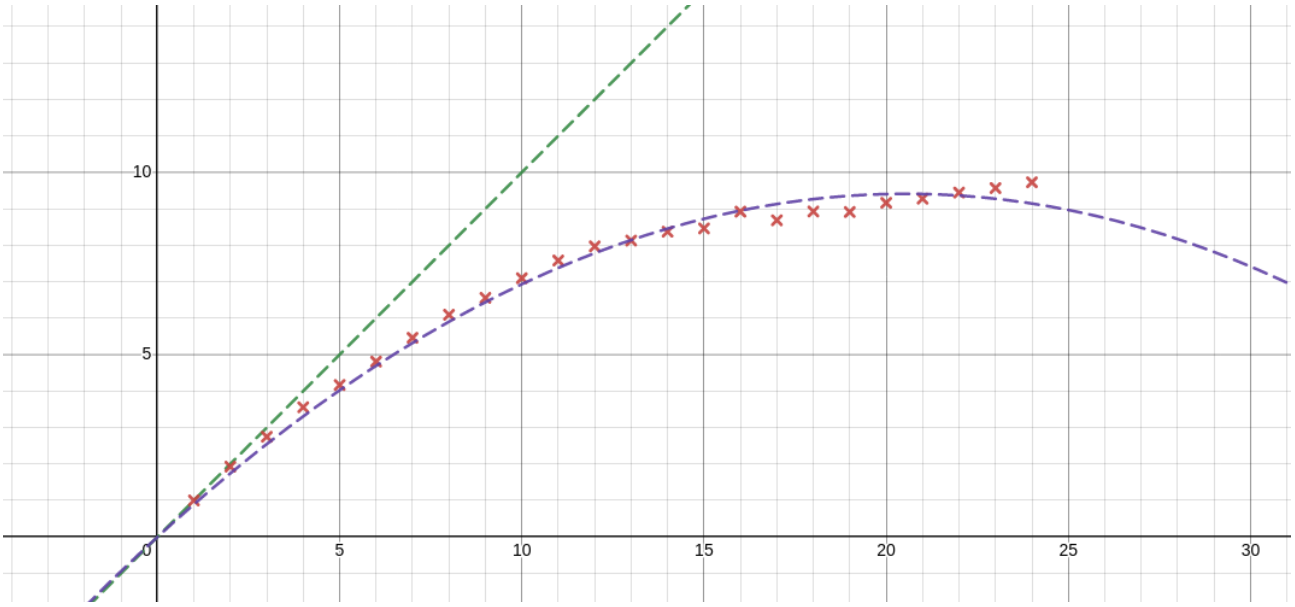
- **trace.sh**

```
#!/bin/bash

#PBS -l walltime=00:02:00
#PBS -l select=2:ncpus=8:mpiprocs=8:mem=24000m,place=scatter
#PBS -m n


cd $PBS_O_WORKDIR

MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')
echo "Number of MPI processes: $MPI_NP"


echo 'File $PBS_NODEFILE:'
cat  $PBS_NODEFILE
HEIGHT=480
WIDTH=470
REPORT_FILE=report-trace.txt
CHECK_FILE=check-trace.txt

mpirun -trace -machinefile $PBS_NODEFILE -np $MPI_NP ./life \
             $HEIGHT $WIDTH >>$CHECK_FILE 2>>$REPORT_FILE
```

Приложение 4. *Замеры времени выполнения при разном числе процессов*

Число процессов	**Лучшее время, с**	**Ускорение, раз**	**Эффективность, %**
1	79.9604	1.000	100
2	41.4897	1.927	96.4
3	29.1036	2.747	91.6
4	22.4942	3.555	88.9
5	19.1975	4.165	83.3
6	16.6303	4.808	80.1
7	14.6456	5.460	78.0
8	13.1242	6.093	76.2
9	12.1955	6.557	72.9
10	11.2700	7.095	70.9
11	10.5488	7.580	68.9
12	10.0335	7.969	66.4
13	9.83831	8.127	62.5
14	9.55190	8.371	59.8
15	9.45020	8.461	56.4
16	8.96309	8.921	55.8
17	9.20871	8.683	51.1
18	8.95954	8.925	49.6
19	8.97237	8.912	46.9
20	8.72617	9.163	45.8
21	8.61931	9.277	44.2
22	8.46628	9.445	42.9
23	8.35694	9.568	41.6
24	8.22030	9.727	40.5

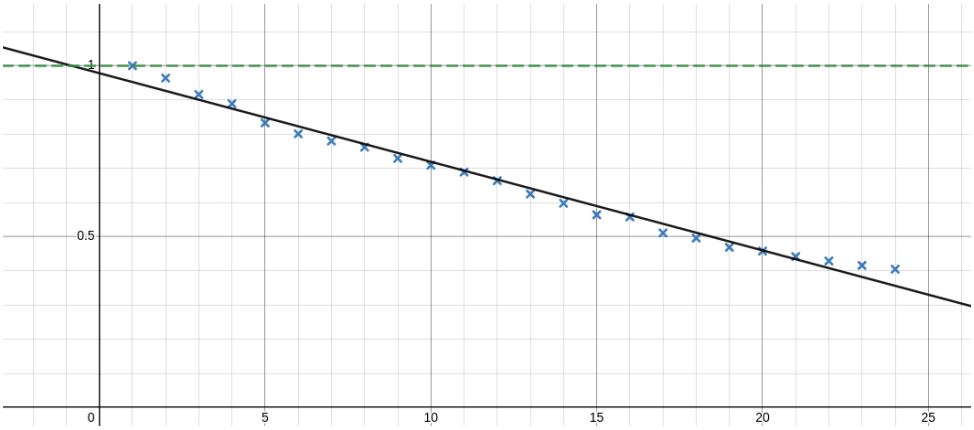
График зависимости ускорения программы от количества процессов



Легенда:

- Ox – число процессов p ;
- Oy – ускорение программы по сравнению с 1 процессом n ;
- Зелёная линия – прямая $n=p$;
- Красные крестики – полученные на основании таблицы выше ускорения;
- Фиолетовая кривая – аппроксимация фунцией $n=0.9162\,p-0.0223\,p^2$, $R^2=0.994$.

График зависимости эффективности процессоров от их количества



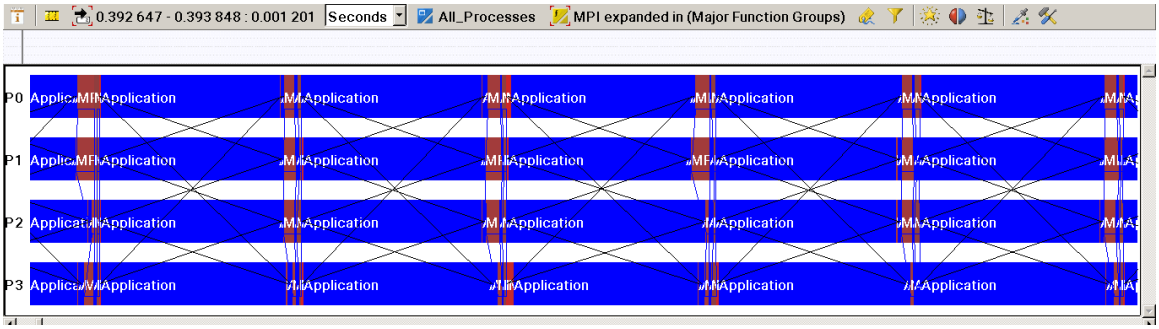
Легенда:

- Ox – число процессов p ;
- Oy – эффективность процессов по сравнению с 1 процессом k ;
- Зелёная линия – прямая $k=1$;
- Голубые крестики – полученные на основании таблицы выше эффективности процессов;
- Чёрная прямая – аппроксимация фунцией $k=0.9784-0.0259\,p$, $R^2=0.993$

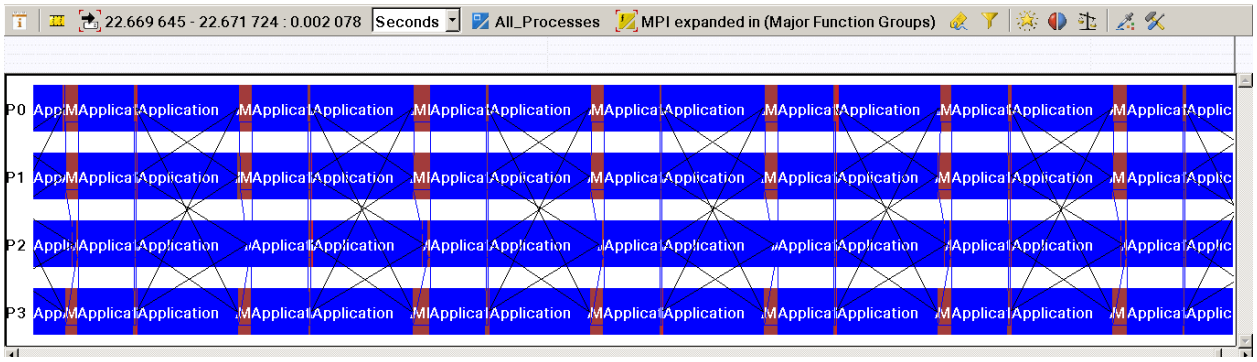
- 4 процесса
  - Общая картина

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	81.7683 s		92.2656 s	4	20.4421 s
MPI_Comm_size	3e-6 s		3e-6 s	4	750e-9 s
MPI_Comm_rank	4e-6 s		4e-6 s	4	1e-6 s
MPI_Type_contiguous	7e-6 s		7e-6 s	4	1.75e-6 s
MPI_Finalize	3.931e-3 s		3.931e-3 s	4	982.75e-6 s
MPI_Type_free	43e-6 s		43e-6 s	4	10.75e-6 s
MPI_Isend	552.569e-3 s		552.569e-3 s	721928	765.407e-9 s
MPI_Irecv	252.59e-3 s		252.59e-3 s	721928	349.883e-9 s
MPI_Type_commit	9e-6 s		9e-6 s	4	2.25e-6 s
MPI_Wtime	26e-6 s		26e-6 s	8	3.25e-6 s
MPI_Waitall	445.964e-3 s		445.964e-3 s	360964	1.23548e-6 s
MPI_Allgather	7.52821 s		7.52821 s	360964	20.8558e-6 s
MPI_Scatterv	869e-6 s		869e-6 s	4	217.25e-6 s
MPI_Allreduce	1.39188 s		1.39188 s	360964	3.856e-6 s
MPI_Wait	321.246e-3 s		321.246e-3 s	721928	444.983e-9 s

  - Первые итерации



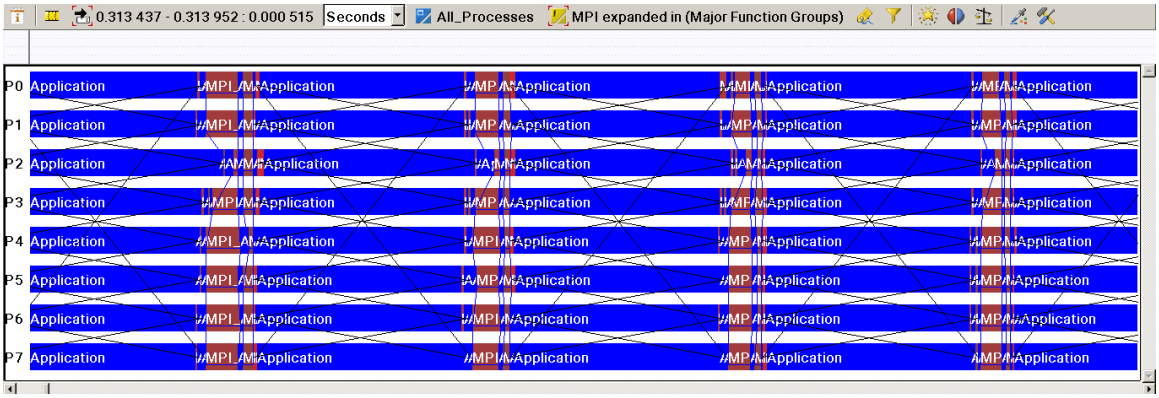
  - Последние итерации



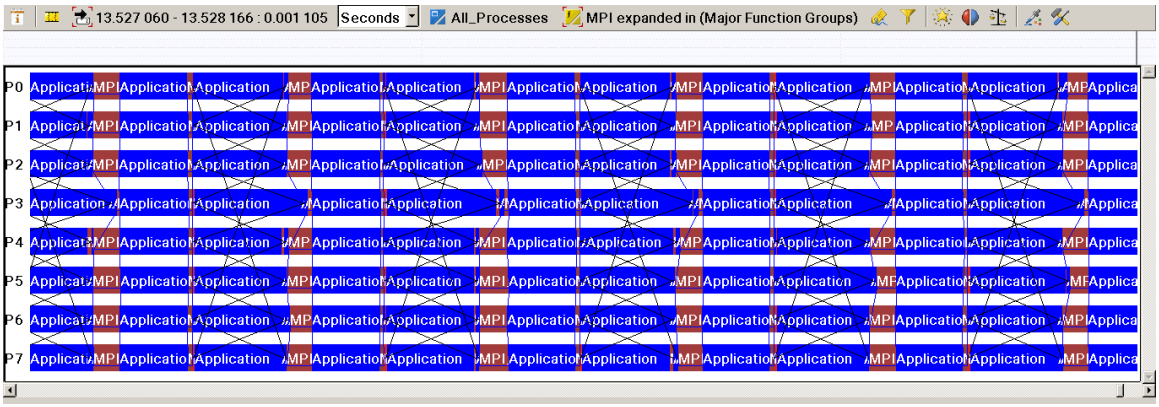
- 8 процессов
  - Общая картина

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	93.3746 s		109.98 s	8	11.6718 s
MPI_Comm_size	4e-6 s		4e-6 s	8	500e-9 s
MPI_Comm_rank	1e-6 s		1e-6 s	8	125e-9 s
MPI_Type_contiguous	13e-6 s		13e-6 s	8	1.625e-6 s
MPI_Finalize	11.514e-3 s		11.514e-3 s	8	1.43925e-3 s
MPI_Type_free	82e-6 s		82e-6 s	8	10.25e-6 s
MPI_Isend	960.036e-3 s		960.036e-3 s	1443856	664.911e-9 s
MPI_Irecv	496.728e-3 s		496.728e-3 s	1443856	344.029e-9 s
MPI_Type_commit	16e-6 s		16e-6 s	8	2e-6 s
MPI_Wtime	41e-6 s		41e-6 s	16	2.5625e-6 s
MPI_Waitall	726.361e-3 s		726.361e-3 s	721928	1.00614e-6 s
MPI_Allgather	10.751 s		10.751 s	721928	14.892e-6 s
MPI_Scatterv	1.75e-3 s		1.75e-3 s	8	218.75e-6 s
MPI_Allreduce	3.1004 s		3.1004 s	721928	4.29461e-6 s
MPI_Wait	557.875e-3 s		557.875e-3 s	1443856	386.379e-9 s

  - Первые итерации



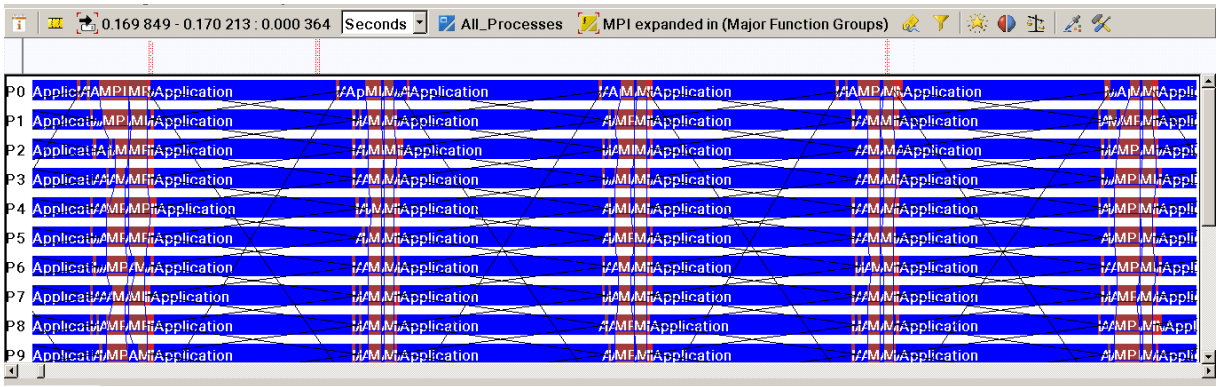
  - Последние итерации



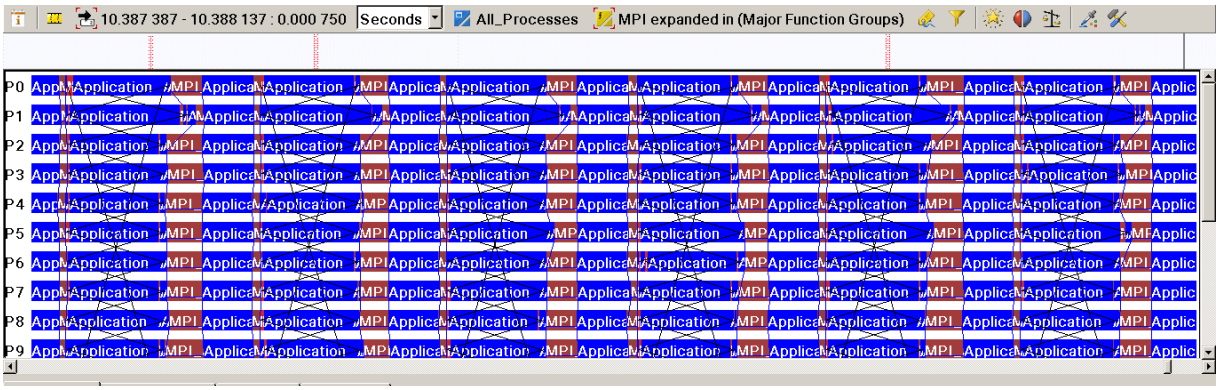
- 12 процессов
  - Общая картина

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	103.108 s		127.833 s	12	8.59232 s
MPI_Comm_size	6e-6 s		6e-6 s	12	500e-9 s
MPI_Comm_rank	7e-6 s		7e-6 s	12	583.333e-9 s
MPI_Type_contiguous	20e-6 s		20e-6 s	12	1.66667e-6 s
MPI_Finalize	27.207e-3 s		27.207e-3 s	12	2.26725e-3 s
MPI_Type_free	157e-6 s		157e-6 s	12	13.0833e-6 s
MPI_Isend	1.38246 s		1.38246 s	2165784	638.317e-9 s
MPI_Irecv	743.201e-3 s		743.201e-3 s	2165784	343.156e-9 s
MPI_Type_commit	26e-6 s		26e-6 s	12	2.16667e-6 s
MPI_Wtime	66e-6 s		66e-6 s	24	2.75e-6 s
MPI_Waitall	1.11148 s		1.11148 s	1082892	1.0264e-6 s
MPI_Allgather	14.2845 s		14.2845 s	1082892	13.1911e-6 s
MPI_Scatterv	2.607e-3 s		2.607e-3 s	12	217.25e-6 s
MPI_Allreduce	6.35008 s		6.35008 s	1082892	5.864e-6 s
MPI_Wait	823.032e-3 s		823.032e-3 s	2165784	380.016e-9 s

  - Первые итерации



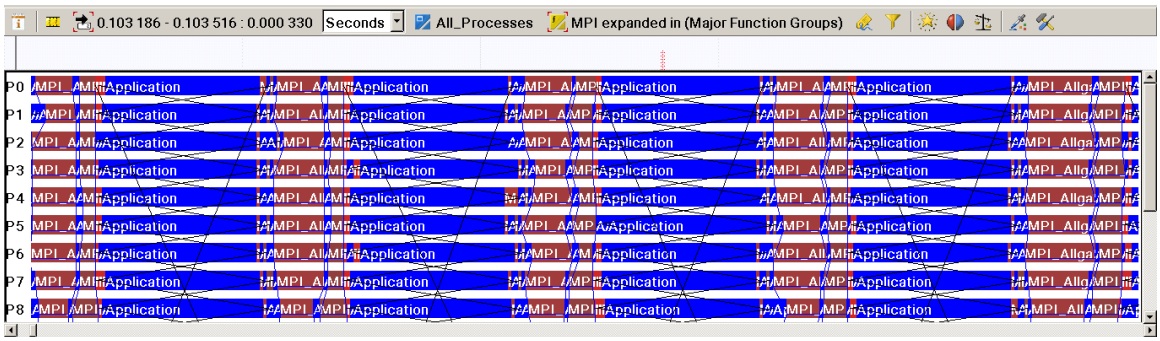
  - Последние итерации



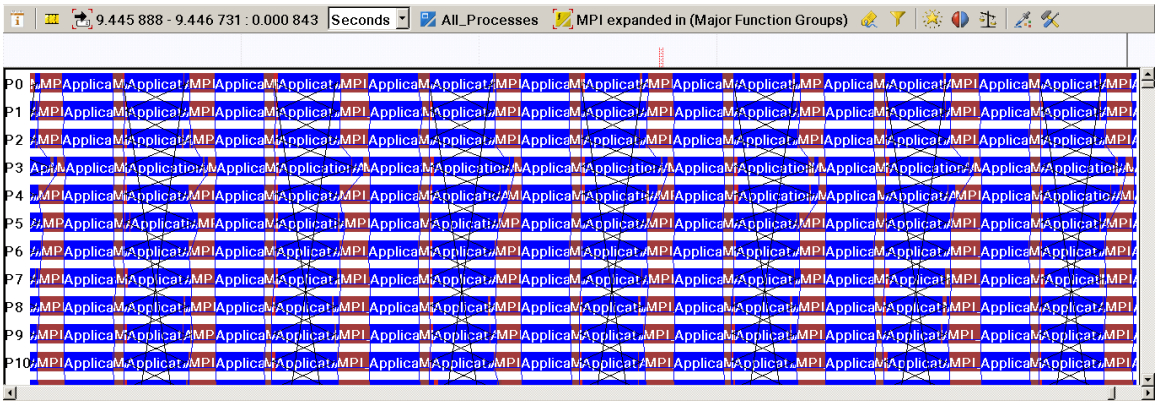
- 16 процессов
  - Общая картина

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	115.078 s		154.735 s	16	7.19238 s
MPI_Comm_size	10e-6 s		10e-6 s	16	625e-9 s
MPI_Comm_rank	8e-6 s		8e-6 s	16	500e-9 s
MPI_Type_contiguous	27e-6 s		27e-6 s	16	1.6875e-6 s
MPI_Finalize	22.701e-3 s		22.701e-3 s	16	1.41881e-3 s
MPI_Type_free	151e-6 s		151e-6 s	16	9.4375e-6 s
MPI_Isend	2.08664 s		2.08664 s	2887712	722.593e-9 s
MPI_Irecv	1.06466 s		1.06466 s	2887712	368.686e-9 s
MPI_Type_commit	29e-6 s		29e-6 s	16	1.8125e-6 s
MPI_Wtime	47e-6 s		47e-6 s	32	1.46875e-6 s
MPI_Waitall	1.64628 s		1.64628 s	1443856	1.1402e-6 s
MPI_Allgather	21.2277 s		21.2277 s	1443856	14.7021e-6 s
MPI_Scatterv	4.754e-3 s		4.754e-3 s	16	297.125e-6 s
MPI_Allreduce	12.417 s		12.417 s	1443856	8.59991e-6 s
MPI_Wait	1.18682 s		1.18682 s	2887712	410.988e-9 s

  - Первые итерации



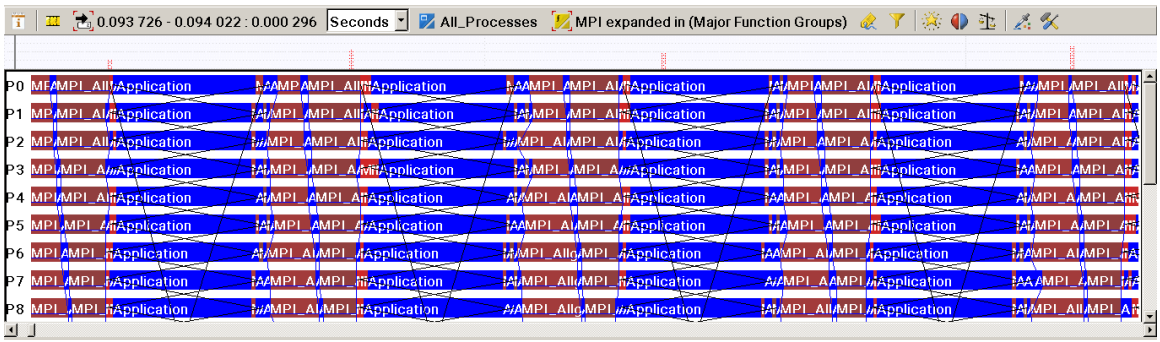
  - Последние итерации



- 20 процессов
  - Общая картина

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
MPI_Comm_size	6e-6 s		6e-6 s	20	300e-9 s
MPI_Comm_rank	7e-6 s		7e-6 s	20	350e-9 s
MPI_Type_contiguous	23e-6 s		23e-6 s	20	1.15e-6 s
MPI_Finalize	36.32e-3 s		36.32e-3 s	20	1.816e-3 s
MPI_Type_free	218e-6 s		218e-6 s	20	10.9e-6 s
MPI_Isend	2.67457 s		2.67457 s	3609640	740.952e-9 s
MPI_Irecv	1.33204 s		1.33204 s	3609640	369.023e-9 s
MPI_Type_commit	43e-6 s		43e-6 s	20	2.15e-6 s
MPI_Wtime	63e-6 s		63e-6 s	40	1.575e-6 s
MPI_Waitall	2.06639 s		2.06639 s	1804820	1.14493e-6 s
MPI_Allgather	34.6413 s		34.6413 s	1804820	19.1938e-6 s
MPI_Scatterv	4.116e-3 s		4.116e-3 s	20	205.8e-6 s
MPI_Allreduce	21.297 s		21.297 s	1804820	11.8001e-6 s
MPI_Wait	1.49937 s		1.49937 s	3609640	415.379e-9 s
Group Application	128.157 s		191.708 s	20	6.40785 s

  - Первые итерации



  - Последние итерации

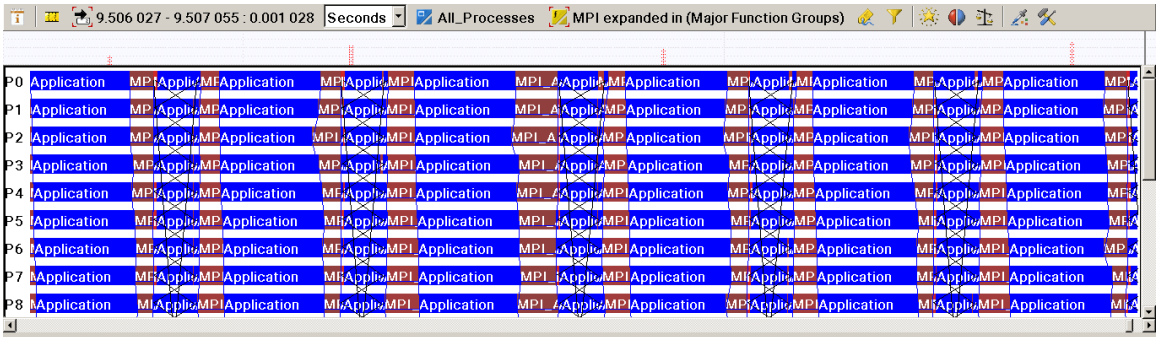
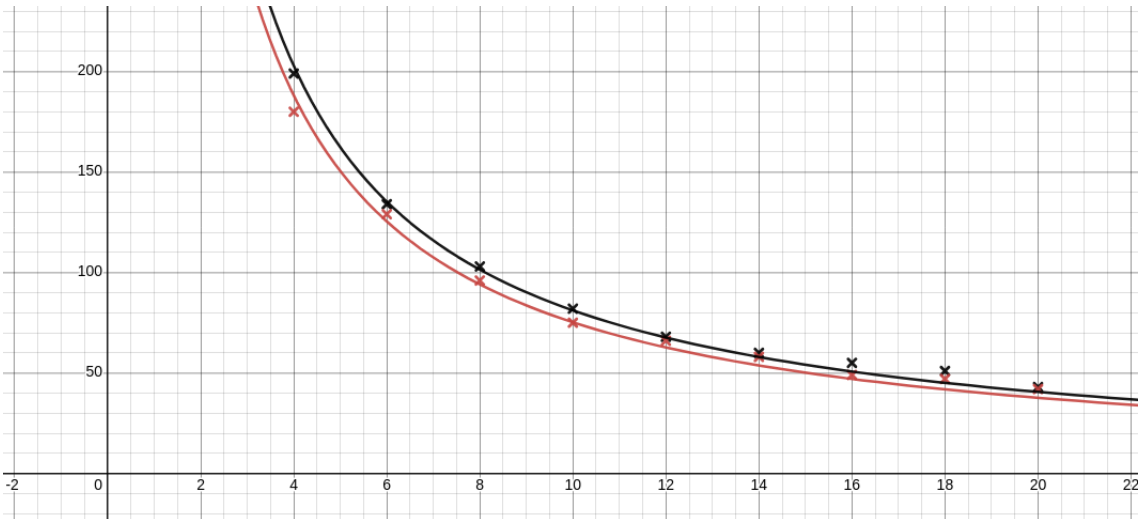


Таблица зависимости времени исполнения конкретного дейстия от числа процессов и номера итерации

Первое число в ячейче соответствует первым итерациям, второе – последним

Число процессов	**Изменение состояния поля, мкс**
4	199-180
6	134-129
8	103-96
10	82-75
12	68-66
14	60-58
16	55-49
18	51-47
20	43-42

Зависимость времени времени, необходимое на обновление состояния игрового поля, от числа процессов



Легенда:

- Ox – число процессов p ;
- Oy – время в микросекундах, необходимое на обновление состояния игрового поля t ;
- Чёрные крестики – время, которое реально ушло на обновление состояния поля в первых итерациях;
- Красные крестики – время, которое реально ушло на обновление состояния поля в последних итерациях;
- Чёрная кривая – аппроксимация замеров за первые итерации кривой $t = \frac{811.481}{p}; R^2 = 0.9959$;
- Чёрная кривая – аппроксимация замеров за последние итерации кривой $t = \frac{752.438}{p}; R^2 = 0.9904$.