###### 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

«Игра "Жизнь" Дж. Конвея»

студента 2 курса, группы 21203

***Кондренко Кирилла Павловича***

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

*Кандидат технических наук*

А. Ю. Власенко

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#__RefHeading___Toc114782465)

[ЗАДАНИЕ 3](#__RefHeading___Toc114782466)

[ОПИСАНИЕ РАБОТЫ 4](#__RefHeading___Toc114782467)

[АНАЛИЗ ПОЛУЧЕННЫХ ДАННЫХ 6](#__RefHeading___Toc5525_2357992727)

[ЗАКЛЮЧЕНИЕ 7](#__RefHeading___Toc114782468)

[Приложение 1. Результаты измерений времени 8](#__RefHeading___Toc1147824692)

[Приложение 2. Результаты профилирования 16](#__RefHeading___Toc11478246921)

[Приложение 3. Исходный код программы 22](#__RefHeading___Toc114782469)

[Приложение 4. Исходные коды дополнительных файлов 31](#__RefHeading___Toc1147824691)

# ЦЕЛЬ

Практическое освоение методов реализации алгоритмов мелкозернистого параллелизма на крупноблочном параллельном вычислительном устройстве на примере реализации клеточного автомата «Игра "Жизнь" Дж. Конвея» с использованием неблокирующих коммуникаций библиотеки *MPI*.

# ЗАДАНИЕ

1. Написать параллельную программу на языке C/C++ с использованием *MPI*, реализующую клеточный автомат игры "Жизнь" с завершением программы по повтору состояния клеточного массива в случае одномерной декомпозиции массива по строкам и с циклическими границами массива. Проверить корректность исполнения алгоритма на различном числе процессорных ядер и различных размерах клеточного массива, сравнив с результатами, полученными для исходных данных вручную.
2. Измерить время работы программы при использовании различного числа процессорных ядер: 1, 2, 4, 8, 16, … . Размеры клеточного массива *X* и *Y* подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд. Построить графики зависимости времени работы, ускорения и эффективности распараллеливания от числа используемых ядер.
3. Произвести профилирование программы и выполнить ее оптимизацию. Попытаться достичь 50-процентной эффективности параллельной реализации на 16 ядрах для выбранных *X* и *Y.*

# ОПИСАНИЕ РАБОТЫ

Изначально была написана последовательная программа, так как её написание проще написания параллельной программы, и в дальнейшем планировалось написать параллельную программу, изменив исходный код лишь незначительно. В ходе этого процесса оказалось, что последовательную программу нужно изменить значительно, так как используемые в *MPI* механизмы «не вписывались» в написанный последовательный вариант. После написания параллельной программы было произведено тестирование на различном количество процессов при малых и больших изначальных полях. Дальше были сделаны модификации в коде, которые, как казалось, должны увеличить производительность:

1. Использование библиотечной функции memcmp() для сравнения двух поколений. Эта оптимизация в значительной степени увеличила производительность (в среднем на 40% на всех тестируемых конфигурациях).
2. Подсчитывание количества живых клеток для каждого нового поколения, чтобы использовать это как «хэш» (для ускорения сравнения поколений между собой). В ходе добавления этих модификаций тестирование происходило при начальной конфигурации «Glider». Поскольку данная конфигурация предполагает, что практически в любой момент времени в любом поколении количество живых клеток не меняется, то внедрение данной модификации не оказало положительного влияния на производительность (а кое-где даже ухудшило её). Однако далее были проведены замеры времени на конфигурации, которая заполнялась случайно, в следствие чего количество живых клеток уже постоянно изменялось и данная модификация внесла значительный вклад в увеличение производительности (30% в случайно-генерируемой конфигурации).
3. Замена использования функций языка C на макросы. Современные компиляторы сами могут делать данную процедуру («подстановку»), однако если сделать её самому — прирост производительности будет гарантирован. В среднем, оптимизация уменьшила время работы на 20% (для всех тестируемых конфигураций).
4. Чтение содержимого файла с использование MPI\_File\_open() вместо встроенных функций языка C. Эта оптимизация оказалось плохой. Она не просто не увеличивала производительность, но ещё и уменьшала её (всё зависит от размера изначальной конфигурации).
5. Использование функции MPI\_Test(). Данная функции позволяет вместо ожидания завершения асинхронной операции выполнить какую-нибудь другую операцию. Применение данной оптимизации не оказало существенного влияния на производительность, однако это не значит, что эта оптимизация не нужна, поскольку она и не ухудшила производительность, но в некоторый момент может её и улучшить.
6. Использование типа uint8\_t вместо типов bool и int. Изначально в программе поколения хранились в массиве типа int. Почти сразу же стало ясно, что здесь можно использовать тип bool, причём сразу по двум причинам: само поколение будет занимать меньше места в оперативной памяти, передача данных между *MPI*-процессами будет происходить быстрее. Была произведена замена на тип bool. Однако, после чтения документации по языкам программирования C/C++ было выяснено, что стандартами не гарантируется , что тип bool будет занимать в памяти ровно 1 байт. В следствие этого было принято решение заменить использование типа bool на тип uint8\_t, размер которого гарантировано равен 1 байт.

После применения вышеперечисленных модификаций было произведено «ручное» профилирование, то есть профилирование без использования сторонних программ-профилировщиков. Это было сделано для того чтобы можно было проанализировать затраты по времени не для отдельных типов функций, а для логически разделённых блоков. После этого было произведено профилирование при использовании 16 процессов помощью внешних профилировщиков (*ITAC* — intel trace analyzer and collector).

**Отпрофилированные конфигурации**

* «Glider 600x600» — «глайдер». Все клетки заполнены нулями, при этом единицами инициализированы пять клеток: (1,2), (2,3), (3,1), (3,2) и (3,3), а поле имеет размеры 600 на 600
* «Random 400x400» — поле заполнено случайным числами (использовалась функция языка C rand() с «зерном» 0), а поле имеет размеры 400 на 400
* «Concentrated 6000x6000» — поле имеет размеры 6000 на 6000, а любая клетка *(xi, yi)* имеет значение
* «Ones 15000x15000» — поле заполнено единицами и имеет размеры 15000 на 15000

# АНАЛИЗ ПОЛУЧЕННЫХ ДАННЫХ

Полученные данные во многом противоречивы между собой, однако сделать некоторые выводы всё же можно. Во-первых, видно, что в среднем полное время работы (как функция от количества процессов) убывает. К тому же можно заметить, что несмотря на то что в некоторых местах она всё же возрастает, время, затрачиваемое на «эволюцию» (расчёт нового поколения на основе текущего), — строго убывающая функция. Во-вторых, видно, что если количество итераций сравнительно мало (например, десятки), то время на ожидание завершения асинхронных операций и на вычисление вектора флагов остановки практически не затрачивается, что положительно сказывается на производительности. При этом можно обратить внимание на то что, хотя количество итераций в конфигурации «Ones 15000x15000» даже меньше, чем в конфигурации «Concentrated 6000x6000», при этом время работы меньше во втором случае. Это можно объяснить тем, что в первом случае размеры поколений значительно больше, чем во втором. Как следствие — запись итогового поколения в выходной файл и передача строк самого первого поколения занимают существенно больше времени в первом случае, чем во втором.

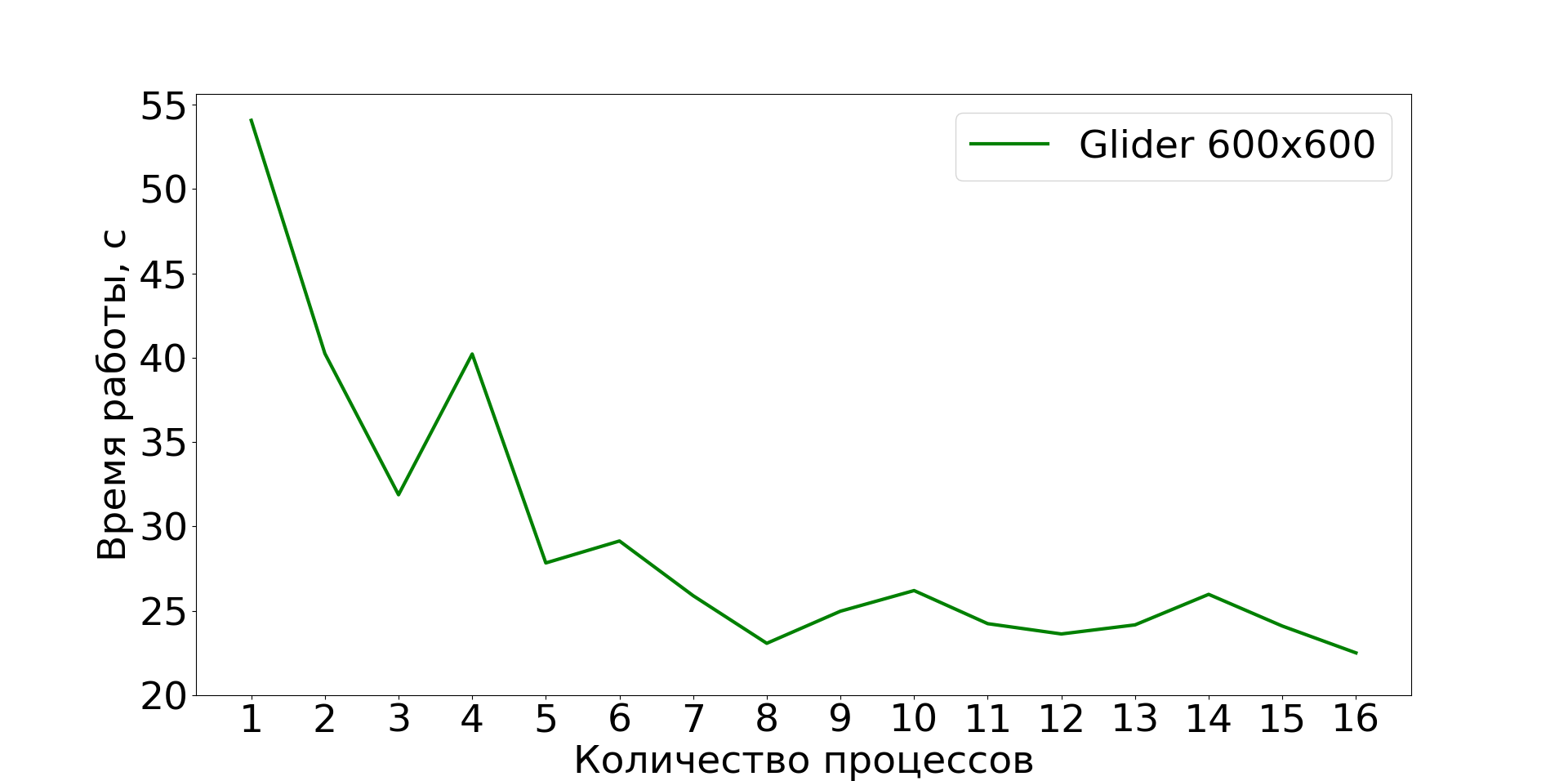
В таблицах присутствует строка под названием «Процент попаданий». При вычислении вектора флагов остановки необходимо сравнивать текущее поколение со всеми предыдущими, однако в программе используется оптимизация: считать и хранить для каждого поколения количество живых клеток в нём. Это позволяет в разы ускорить сравнения поколений (сначала сравниваем хэшы, если они разные — поколения разные, и только если они одинаковые — сравниваем клетки поколений). Видно, что чем выше это значение, тем (при прочих равных) производительность выше. Поэтому, например, конфигурация «Glider 600x600» работает существенно медленнее, чем конфигурация «Random 400x400». При этом ещё можно заметить, что в конфигурации «Glider 600x600» это значение убывает (от числа процессов) при числе процессов > 1. Это может служить причиной того, что в этой конфигурации при увеличении числа процессов время работы может увеличиваться (то есть производительность уменьшается).

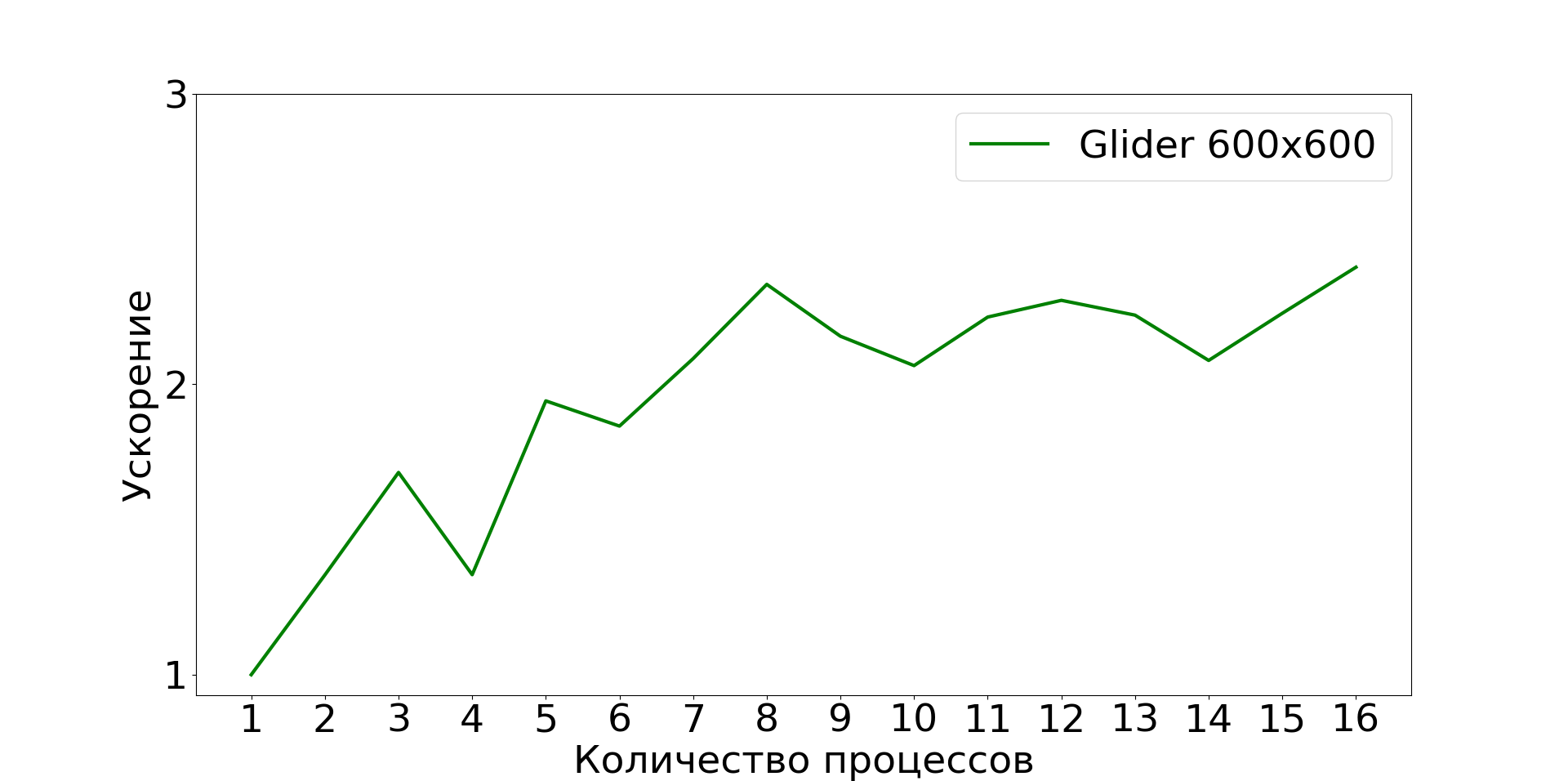
# ЗАКЛЮЧЕНИЕ

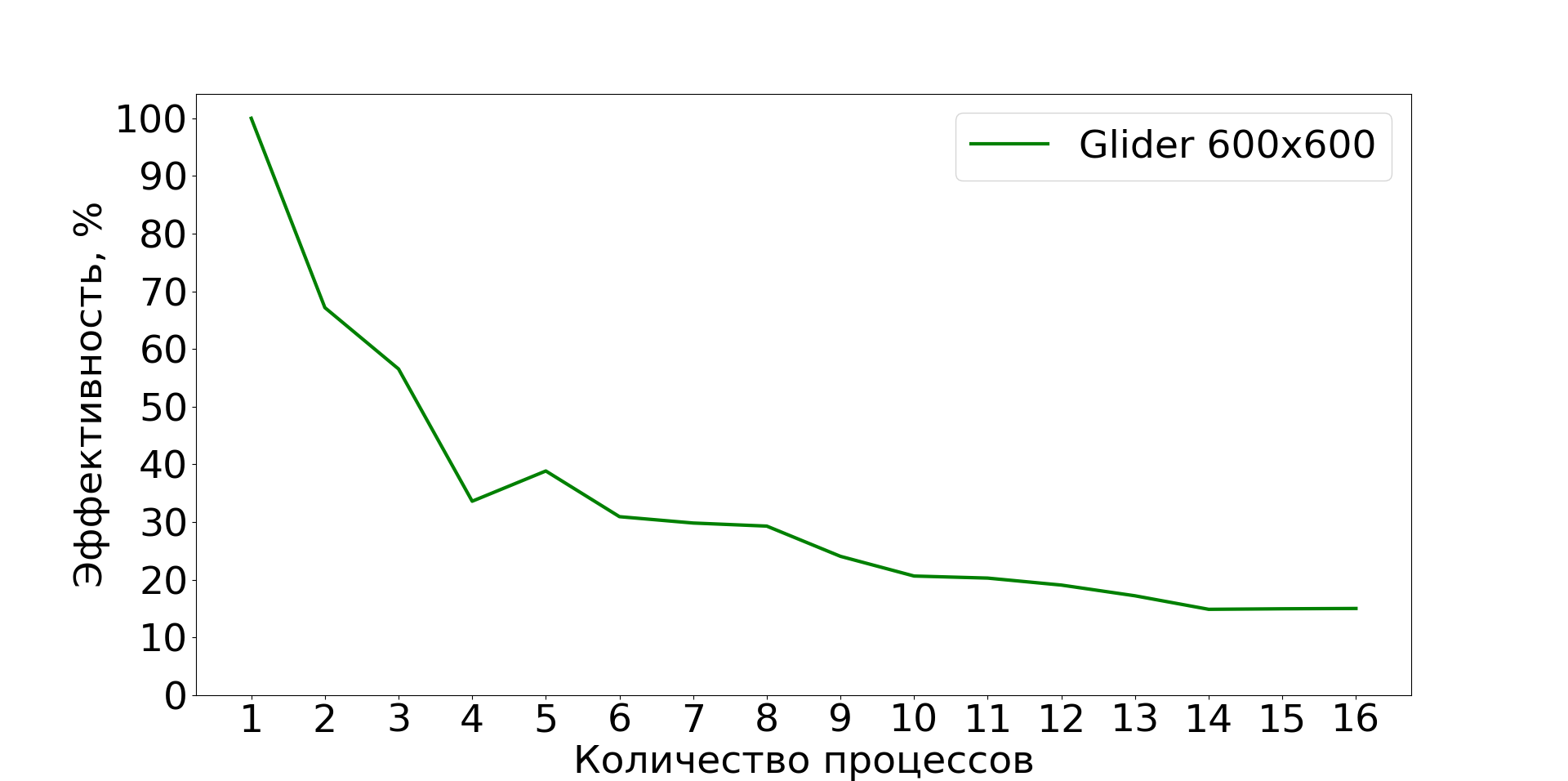
По результатам работы я пришёл к следующим выводам:

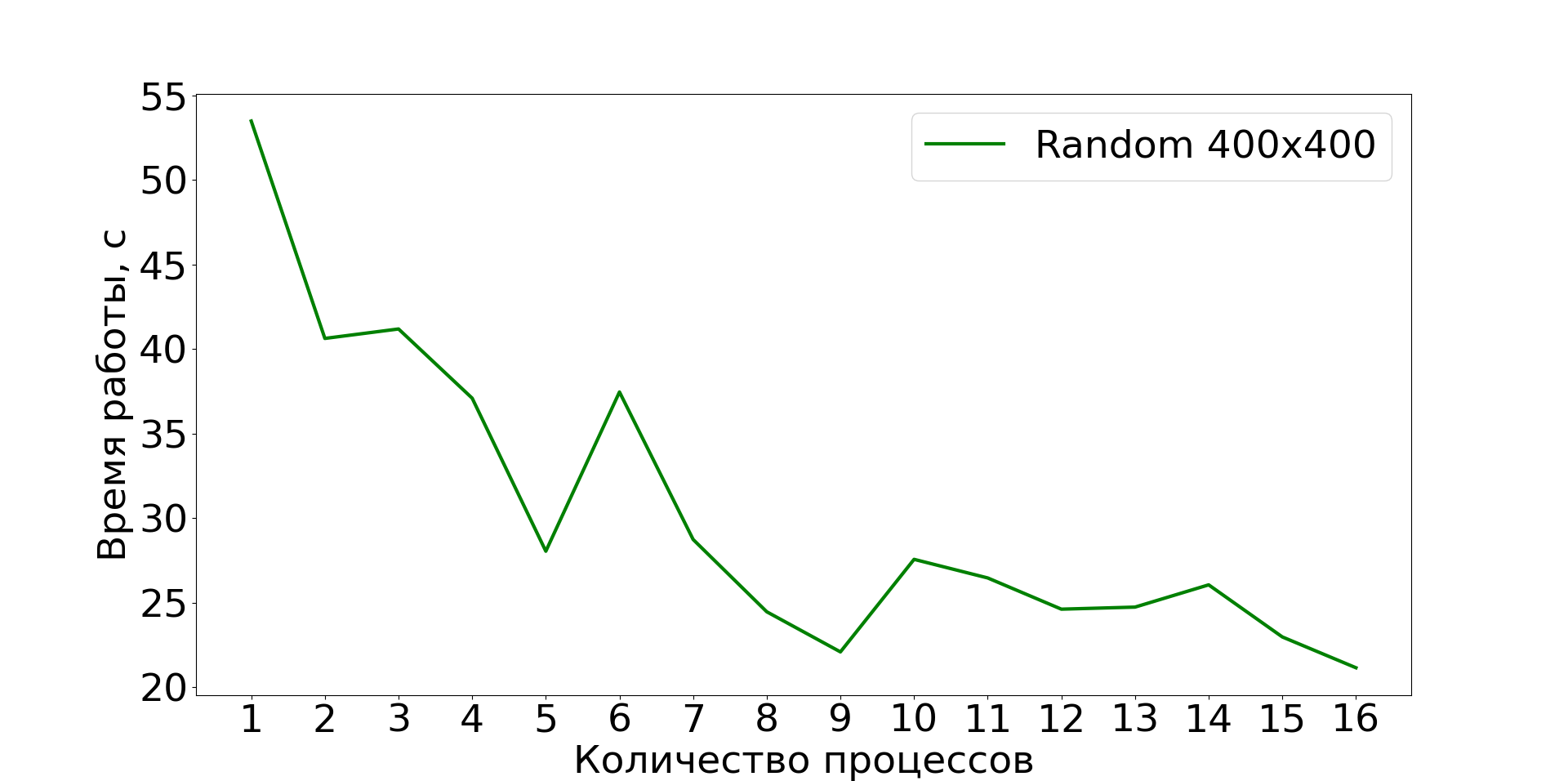
1. Многие кажущиеся очевидными оптимизации (подстановка реализаций функций, хеширование результатов, …) могут в значительной степени ускорить вычислительные программы.
2. Использование асинхронных операций во многих случаях увеличивает производительность программ, использующих *MPI*.
3. Игру "Жизнь" Дж. Конвея можно распараллелить при помощи *MPI*, при этом ускорение и эффективность будут зависеть лишь от самого первого поколения (а именно от размера мира и количества итераций, через которое произойдёт повторение).
4. Во многих параллельных программах, использующих распределённую систему памяти, получение и отправка данных могут являться «бутылочным горлышком» для производительности.

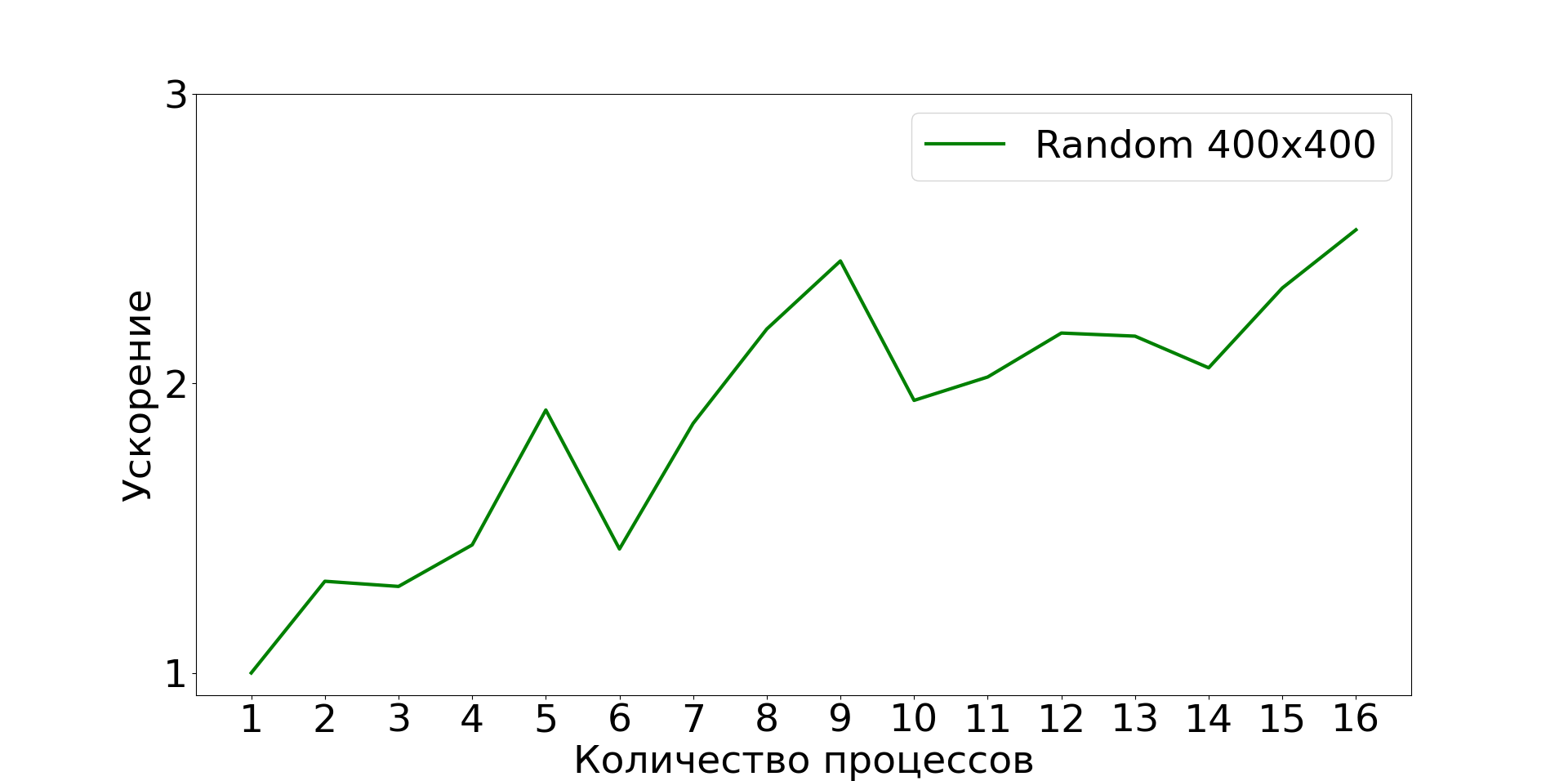
# Приложение 1. Результаты измерений времени

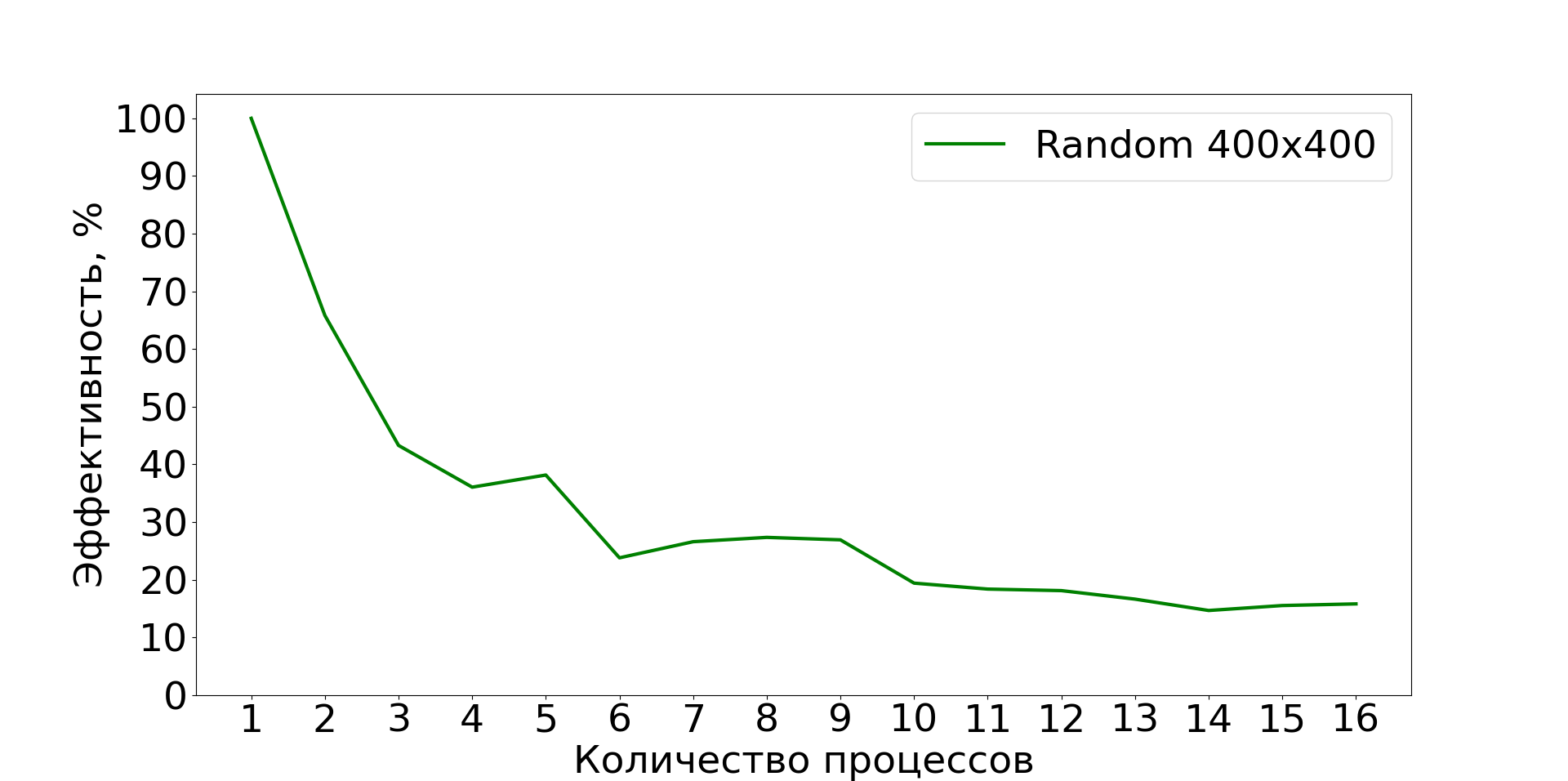


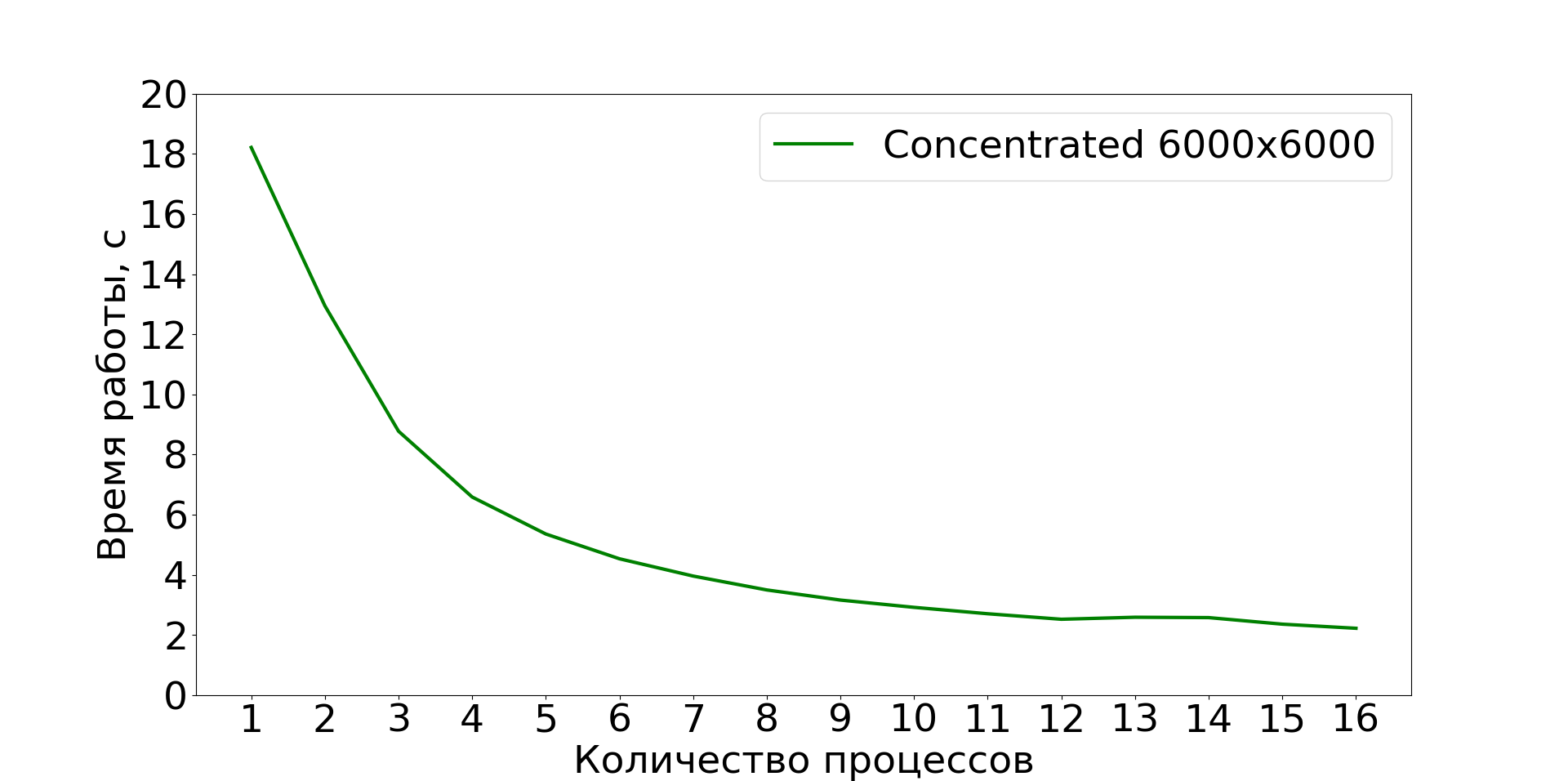


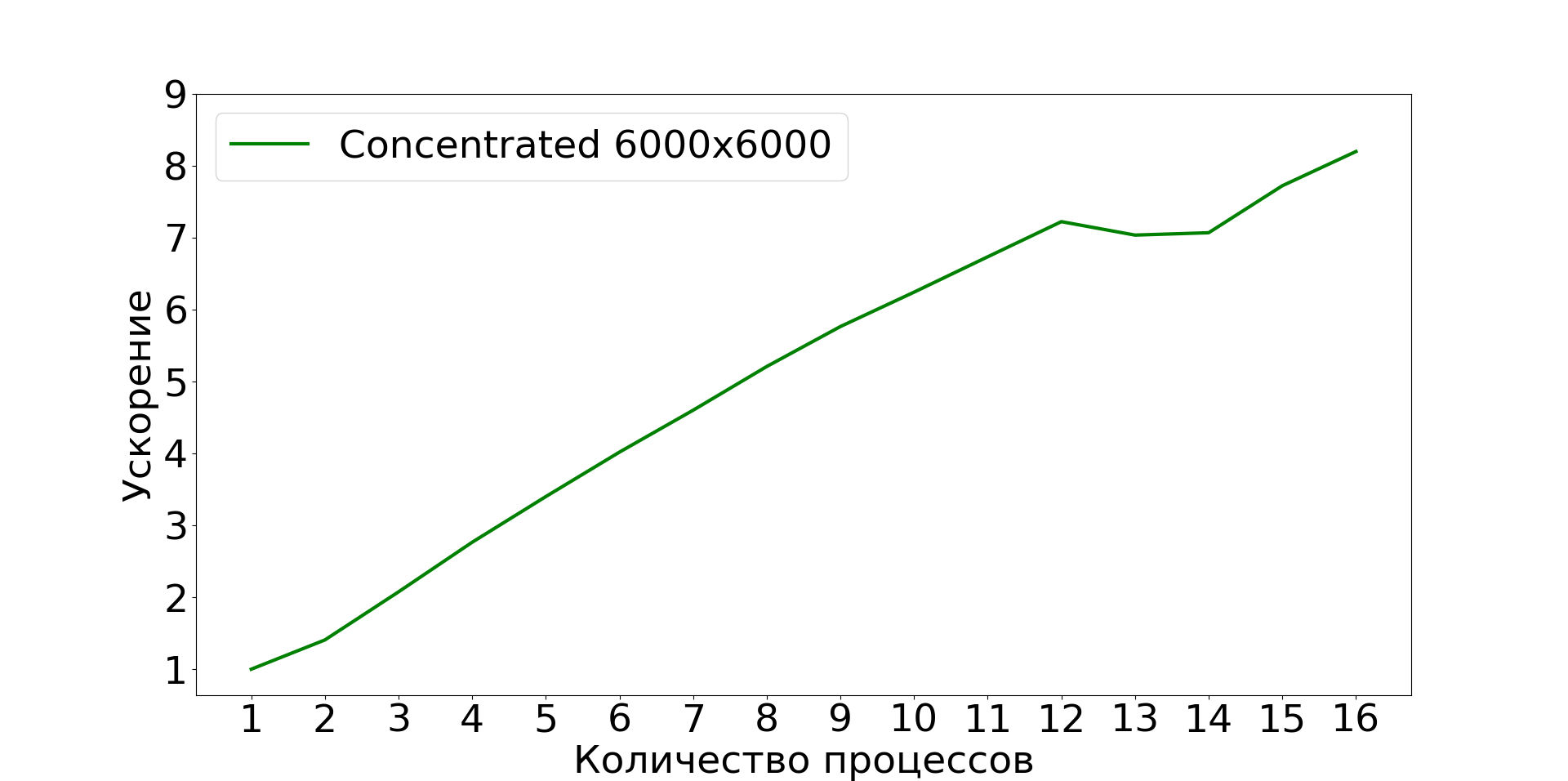
**

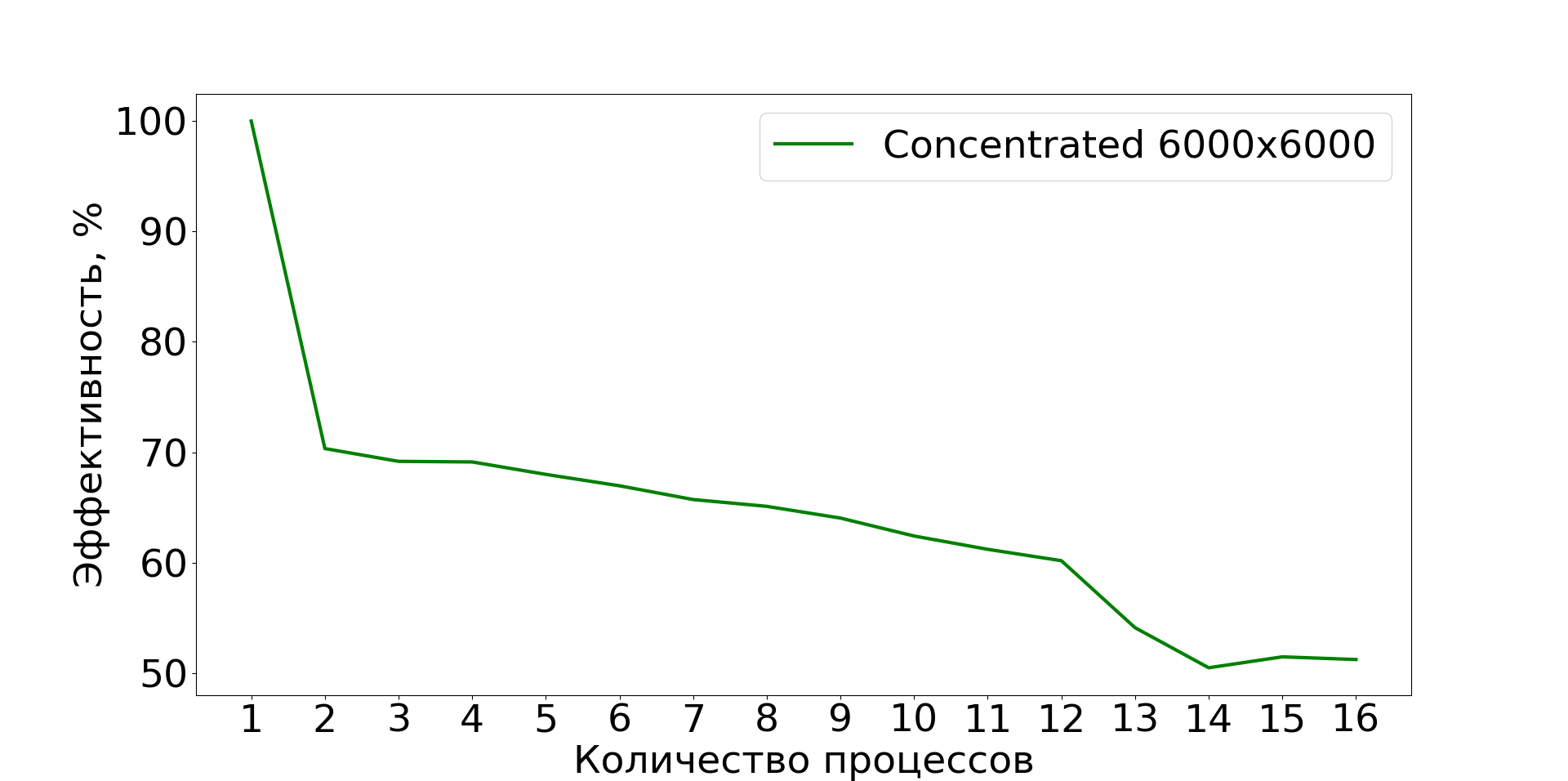
**

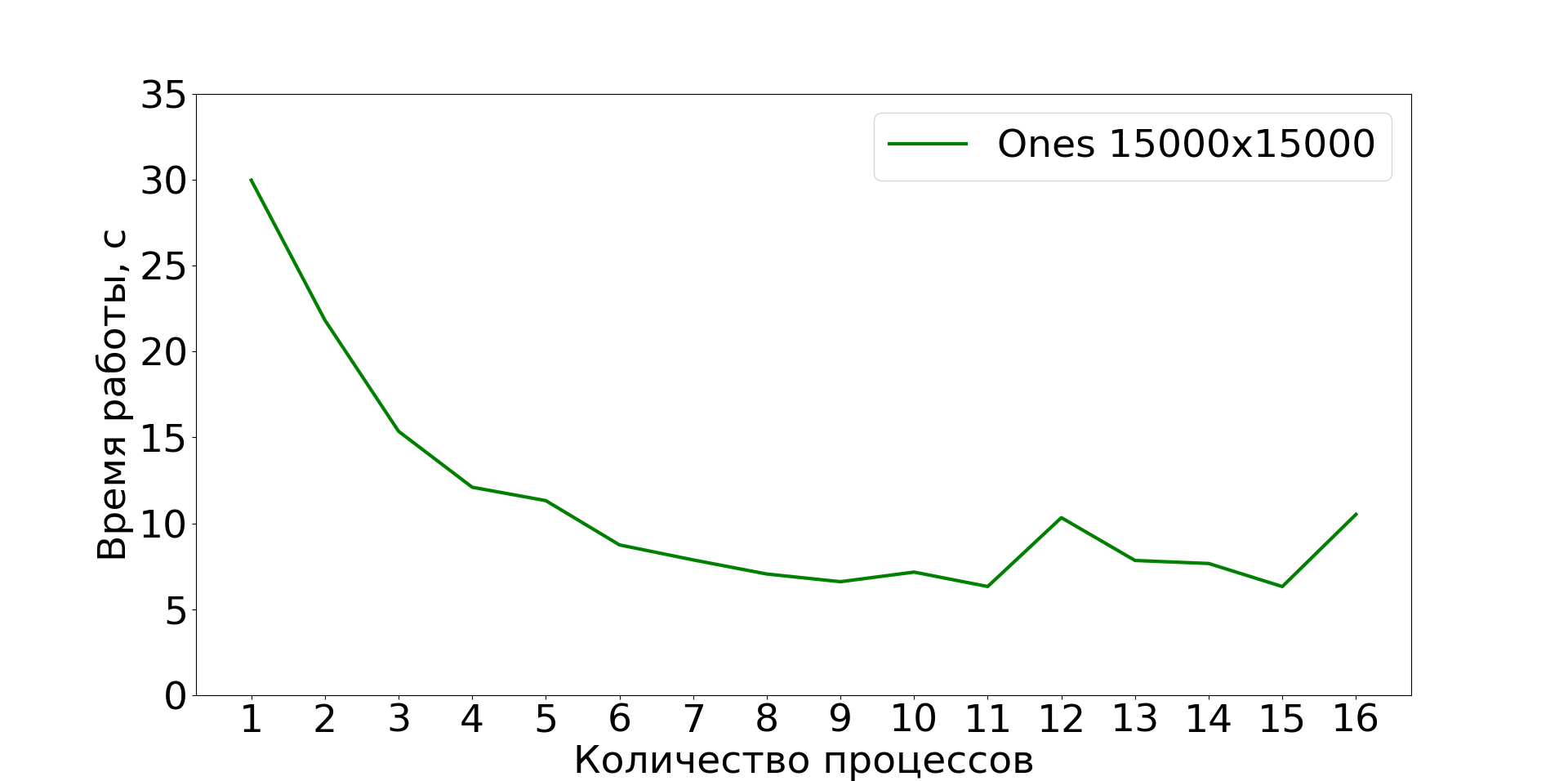
**

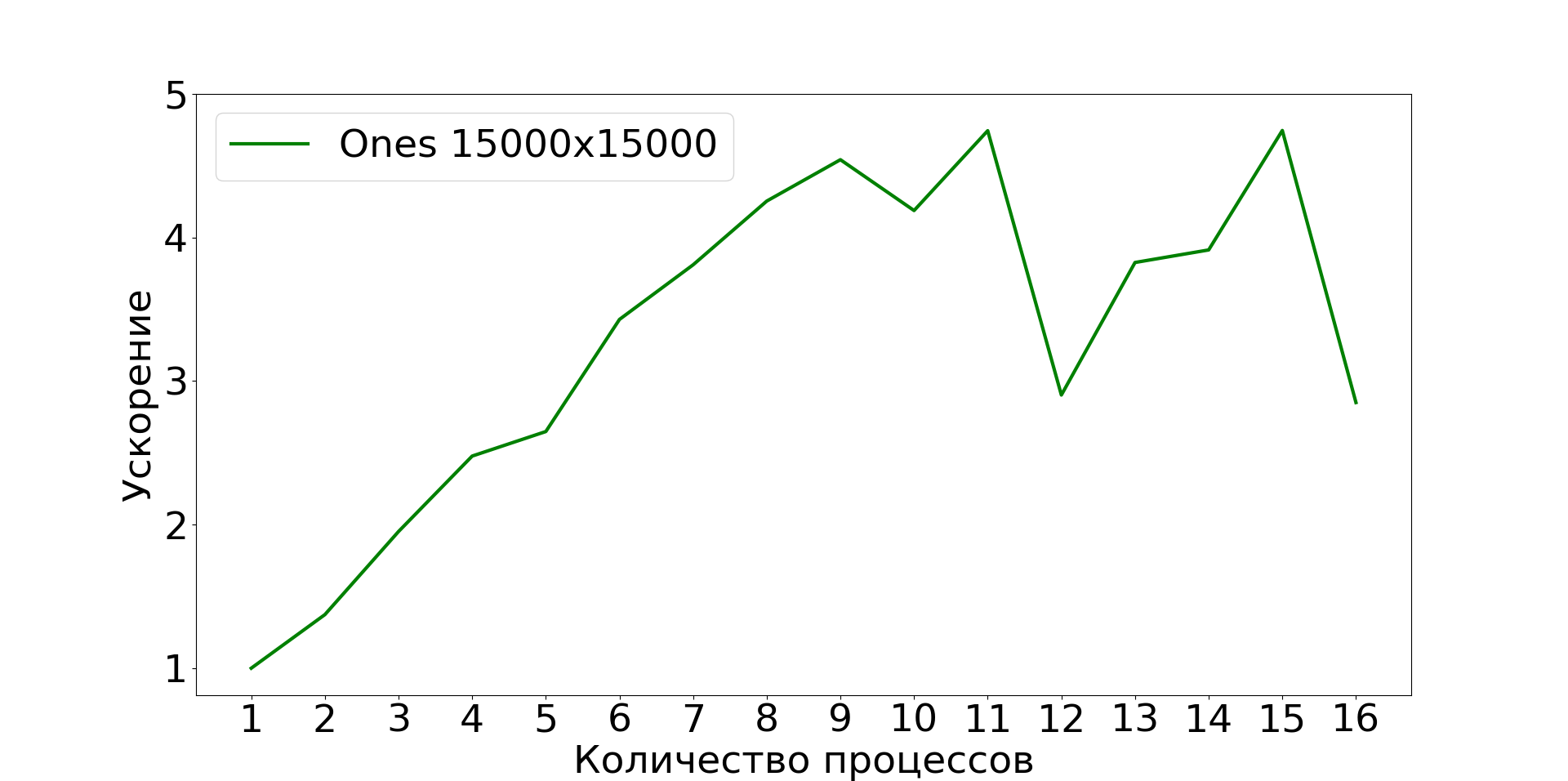
**

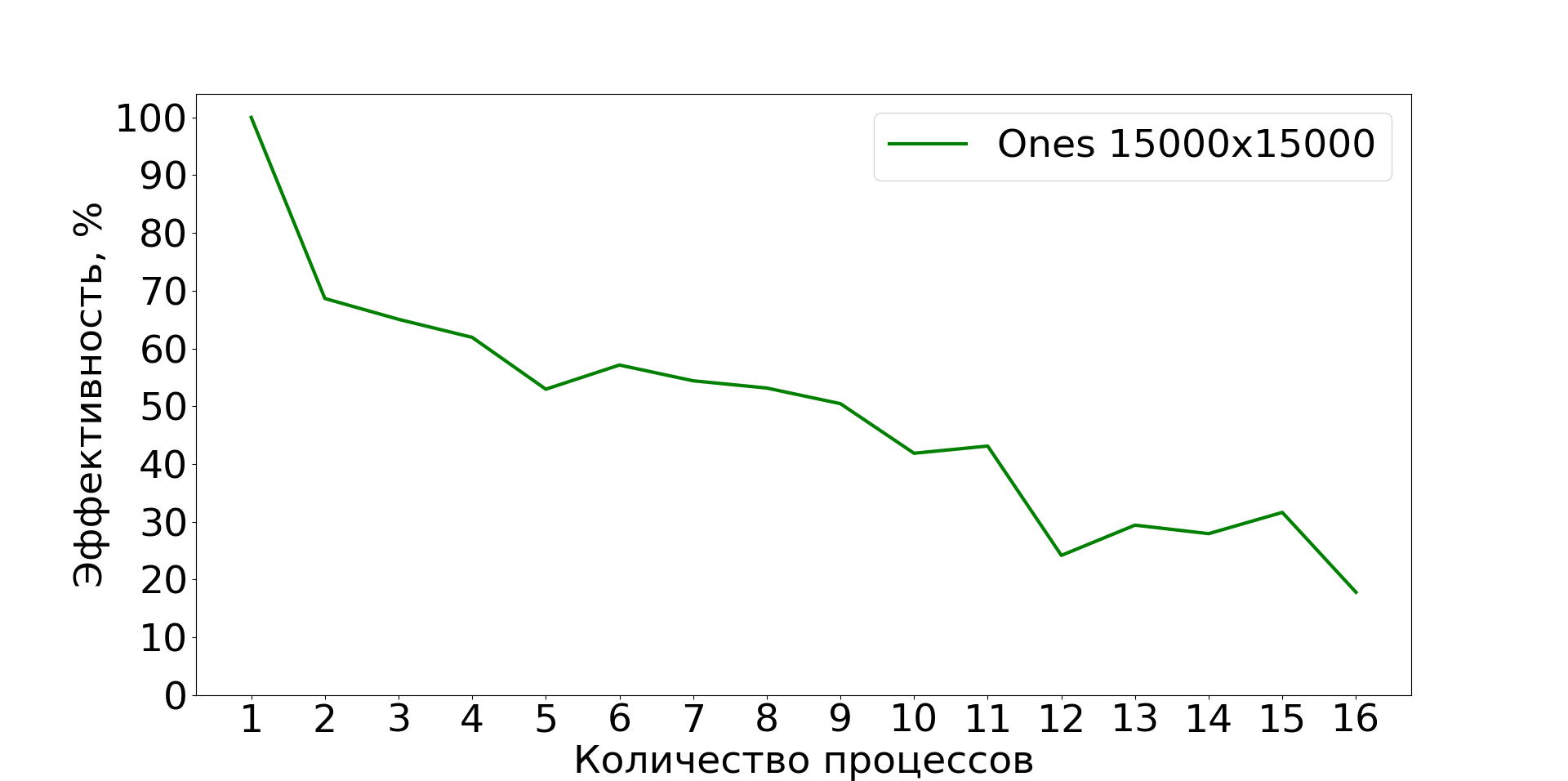
**

**

**

**

**

**

# Приложение 2. Результаты профилирования

**«Ручное» профилирование**

Glider 600x600

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Количество процессов | Количество итераций | Процент попаданий,  % | «Эволюция», мс | Ожидание,  мс | Вычисление вектора остановки | Общее время,  мс |
| 1 | 2400 | 0 | 32326 | 18 | 21438 | 54070 |
| 2 | 2400 | 51 | 22598 | 5386 | 12089 | 40248 |
| 3 | 2400 | 45 | 15781 | 3570 | 12218 | 31876 |
| 4 | 2400 | 40 | 12689 | 5414 | 12121 | 40219 |
| 5 | 2400 | 33 | 8967 | 1960 | 16758 | 27833 |
| 6 | 2400 | 29 | 7462 | 3615 | 17918 | 29135 |
| 7 | 2400 | 25 | 6345 | 2761 | 16603 | 25894 |
| 8 | 2400 | 23 | 5681 | 3528 | 13695 | 23070 |
| 9 | 2400 | 20 | 5010 | 1556 | 17916 | 24972 |
| 10 | 2400 | 19 | 4580 | 2037 | 19194 | 26196 |
| 11 | 2400 | 17 | 4112 | 2107 | 17747 | 24234 |
| 12 | 2400 | 16 | 3791 | 3103 | 16452 | 23624 |
| 13 | 2400 | 15 | 3491 | 1546 | 18940 | 24162 |
| 14 | 2400 | 14 | 3220 | 2096 | 19828 | 25972 |
| 15 | 2400 | 13 | 3067 | 1659 | 18938 | 24093 |
| 16 | 2400 | 12 | 3029 | 3179 | 16020 | 22503 |

Random 400x400

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Количество процессов | Количество итераций | Процент попаданий,  % | «Эволюция», мс | Ожидание,  мс | Вычисление вектора остановки | Общее время,  мс |
| 1 | 8742 | 99 | 52908 | 44 | 114 | 53493 |
| 2 | 8742 | 99 | 36890 | 3200 | 204 | 40631 |
| 3 | 8742 | 99 | 24233 | 16393 | 223 | 41190 |
| 4 | 8742 | 98 | 18192 | 18375 | 1169 | 37094 |
| 5 | 8742 | 95 | 14495 | 8497 | 4800 | 28039 |
| 6 | 8742 | 91 | 11859 | 13577 | 11779 | 37460 |
| 7 | 8742 | 84 | 10231 | 3547 | 14634 | 28734 |
| 8 | 8742 | 84 | 9031 | 3695 | 11446 | 24455 |
| 9 | 8742 | 84 | 8235 | 1340 | 12283 | 22081 |
| 10 | 8742 | 84 | 7324 | 5925 | 14027 | 27558 |
| 11 | 8742 | 68 | 6713 | 5355 | 14051 | 26457 |
| 12 | 8742 | 84 | 6116 | 5892 | 13309 | 24610 |
| 13 | 8742 | 84 | 5644 | 2048 | 16761 | 24732 |
| 14 | 8742 | 84 | 5316 | 2859 | 17582 | 26049 |
| 15 | 8742 | 84 | 4946 | 3221 | 14430 | 22967 |
| 16 | 8742 | 84 | 4805 | 3301 | 12693 | 21143 |

Concentrated 6000x6000

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Количество процессов | Количество итераций | Процент попаданий,  % | «Эволюция», мс | Ожидание,  мс | Вычисление вектора остановки | Общее время,  мс |
| 1 | 12 | 97 | 16143 | 0 | 0 | 18213 |
| 2 | 12 | 97 | 11174 | 0 | 0 | 12947 |
| 3 | 12 | 97 | 7446 | 1 | 0 | 8776 |
| 4 | 12 | 97 | 5587 | 6 | 0 | 6587 |
| 5 | 12 | 97 | 4475 | 4 | 0 | 5357 |
| 6 | 12 | 97 | 3735 | 3 | 0 | 4533 |
| 7 | 12 | 99 | 3196 | 8 | 0 | 3959 |
| 8 | 12 | 97 | 2797 | 7 | 0 | 3497 |
| 9 | 12 | 97 | 2495 | 8 | 0 | 3160 |
| 10 | 12 | 97 | 2245 | 6 | 0 | 2918 |
| 11 | 12 | 97 | 2045 | 8 | 0 | 2705 |
| 12 | 12 | 97 | 1876 | 8 | 0 | 2522 |
| 13 | 12 | 97 | 1756 | 147 | 0 | 2589 |
| 14 | 12 | 99 | 1630 | 137 | 0 | 2577 |
| 15 | 12 | 97 | 1525 | 125 | 0 | 2359 |
| 16 | 12 | 97 | 1432 | 118 | 0 | 2222 |

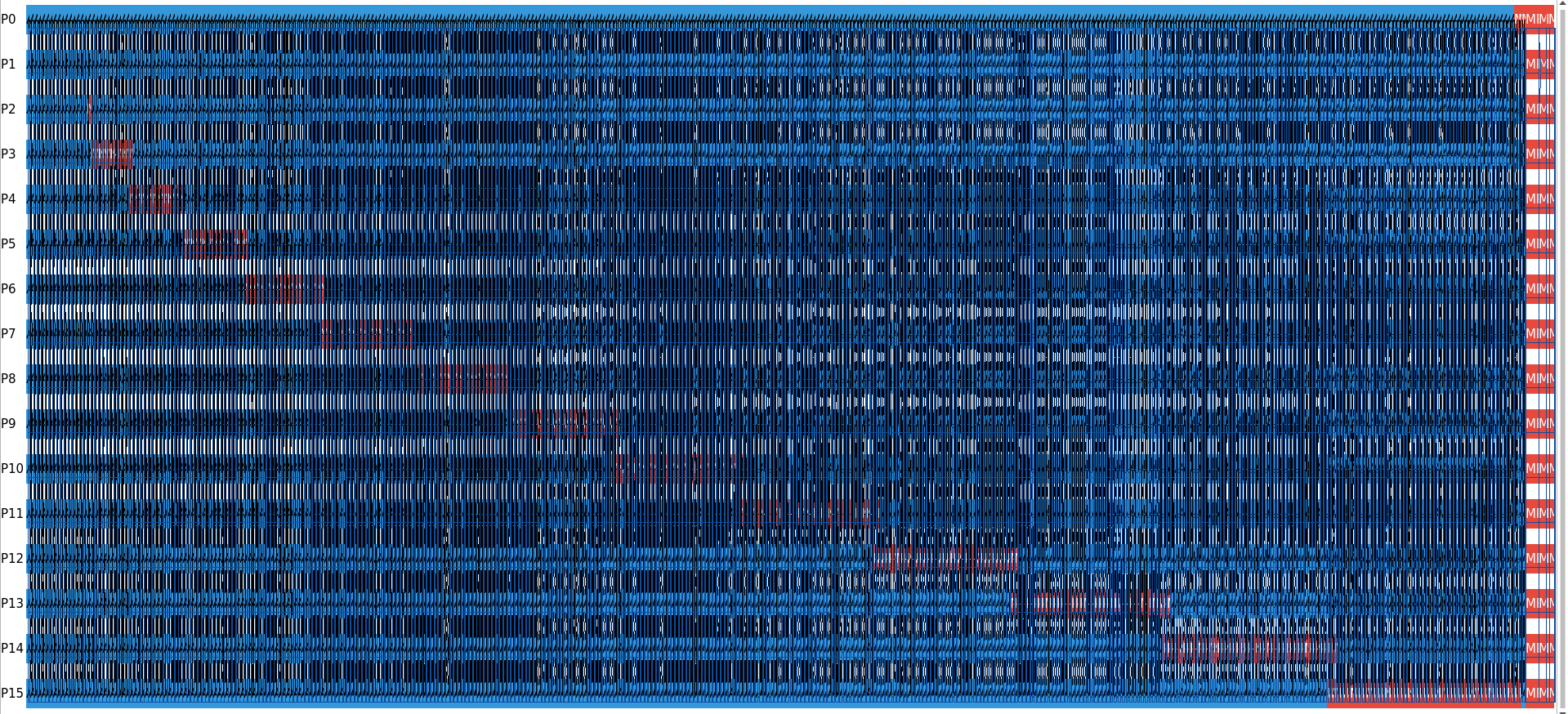
Ones 15000x15000

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Количество процессов | Количество итераций | Процент попаданий,  % | «Эволюция», мс | Ожидание,  мс | Вычисление вектора остановки | Общее время,  мс |
| 1 | 2 | 66 | 16807 | 0 | 0 | 29968 |
| 2 | 2 | 66 | 11643 | 0 | 0 | 21824 |
| 3 | 2 | 66 | 7766 | 2 | 0 | 15355 |
| 4 | 2 | 66 | 5824 | 120 | 0 | 12098 |
| 5 | 2 | 66 | 4650 | 943 | 0 | 11318 |
| 6 | 2 | 66 | 3891 | 61 | 0 | 8741 |
| 7 | 2 | 66 | 3330 | 122 | 0 | 7868 |
| 8 | 2 | 66 | 2919 | 19 | 0 | 7047 |
| 9 | 2 | 66 | 2597 | 13 | 0 | 6600 |
| 10 | 2 | 66 | 2335 | 297 | 0 | 7158 |
| 11 | 2 | 66 | 2126 | 7 | 0 | 6319 |
| 12 | 2 | 66 | 1947 | 428 | 0 | 10326 |
| 13 | 2 | 66 | 1832 | 5 | 0 | 7835 |
| 14 | 2 | 66 | 1689 | 5 | 0 | 7660 |
| 15 | 2 | 66 | 1587 | 37 | 0 | 6317 |
| 16 | 2 | 66 | 1490 | 4 | 0 | 10516 |

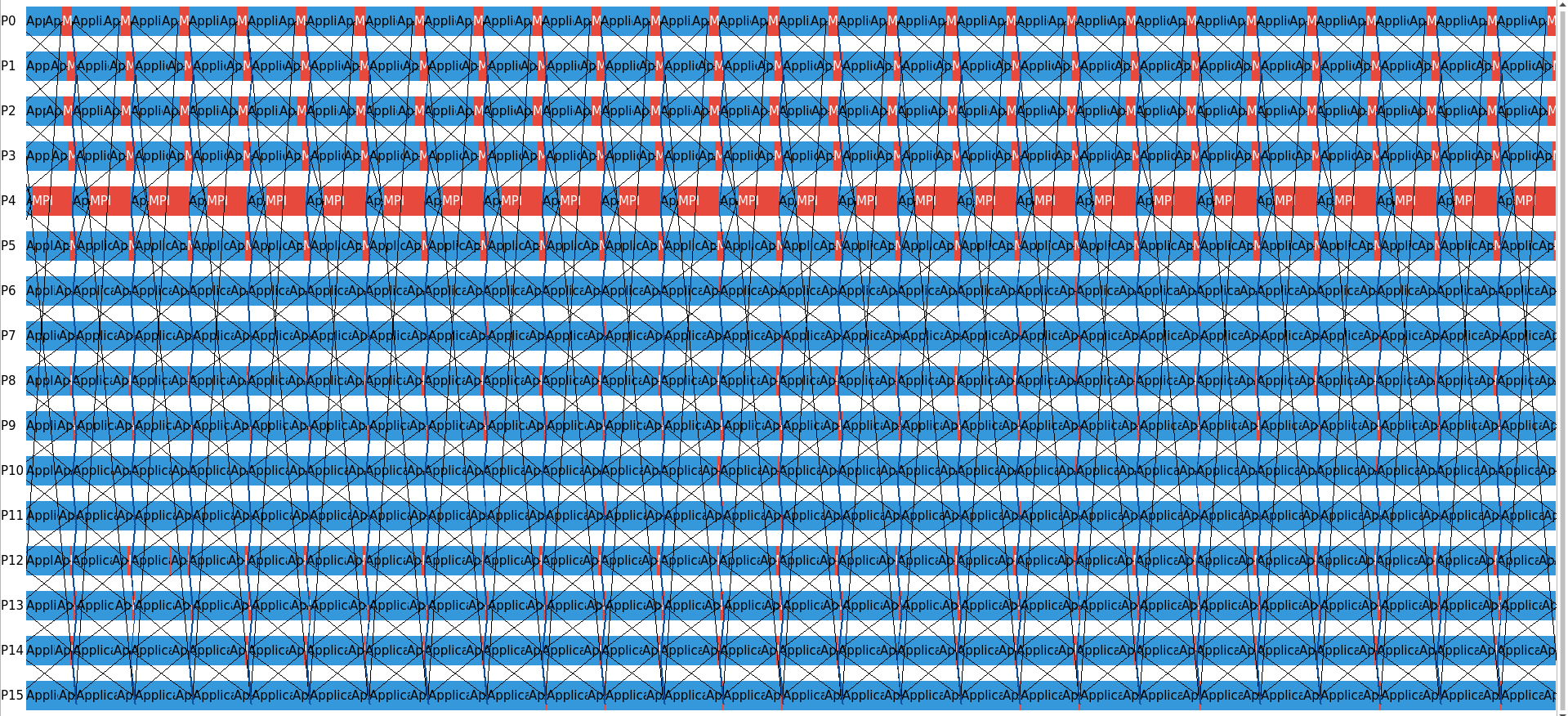
**Профилирование с использованием *ITAC***

Glider 600x600

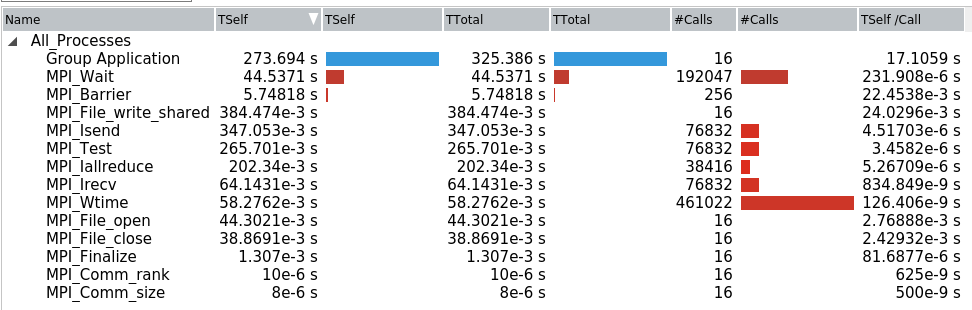
Полная временная шкала

**

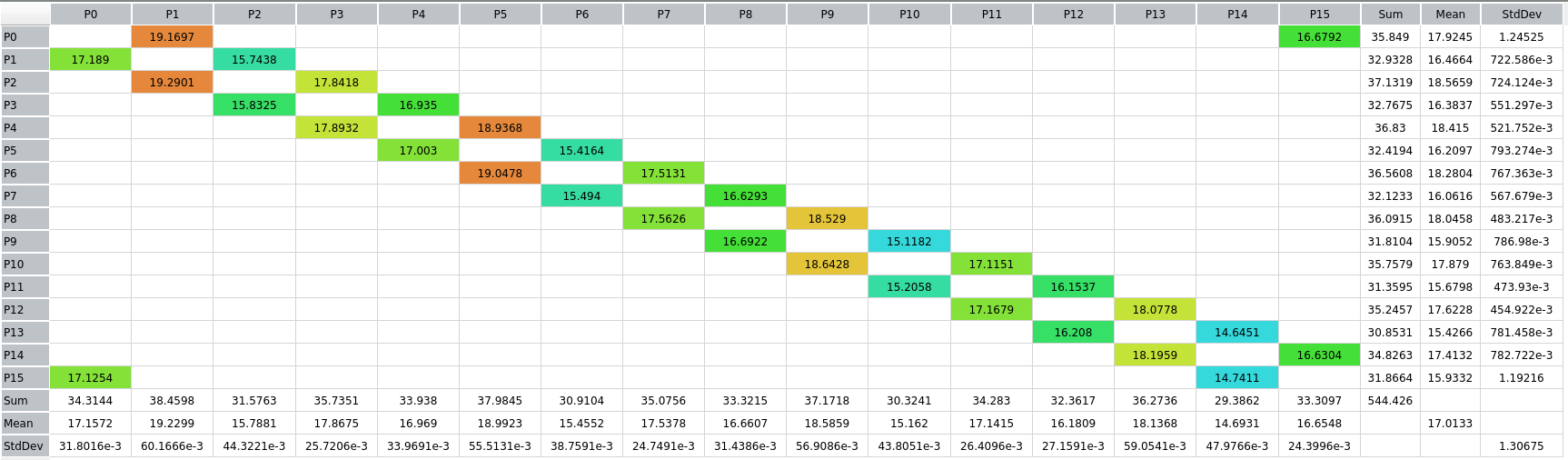
Временная шкала некоторого участка

**

Статистика вызовов функций

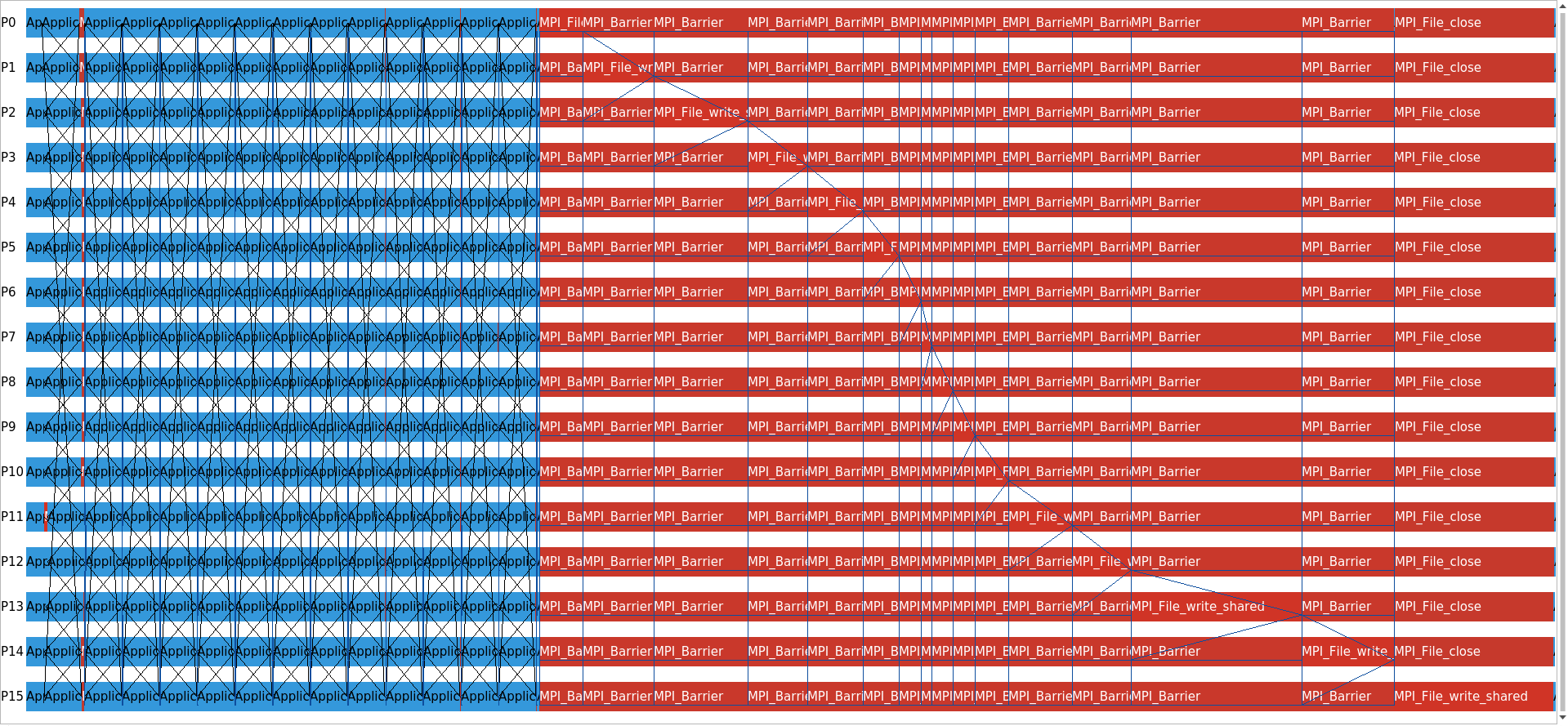


Статистика сообщений между *MPI*-процессами

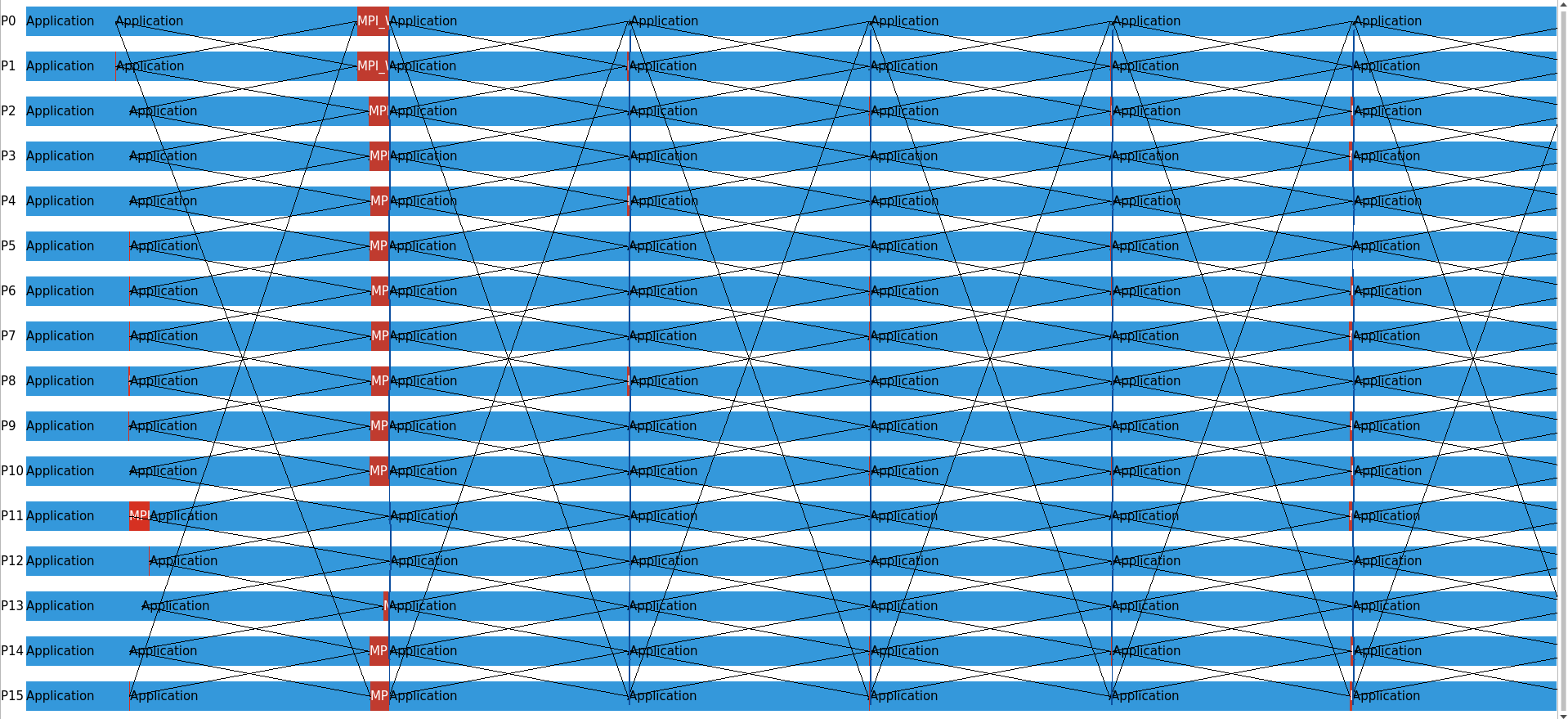


Concentrated 6000x6000

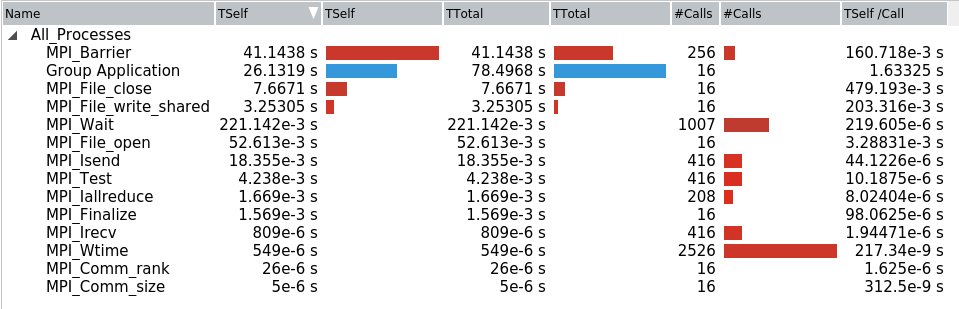
Полная временная шкала

**

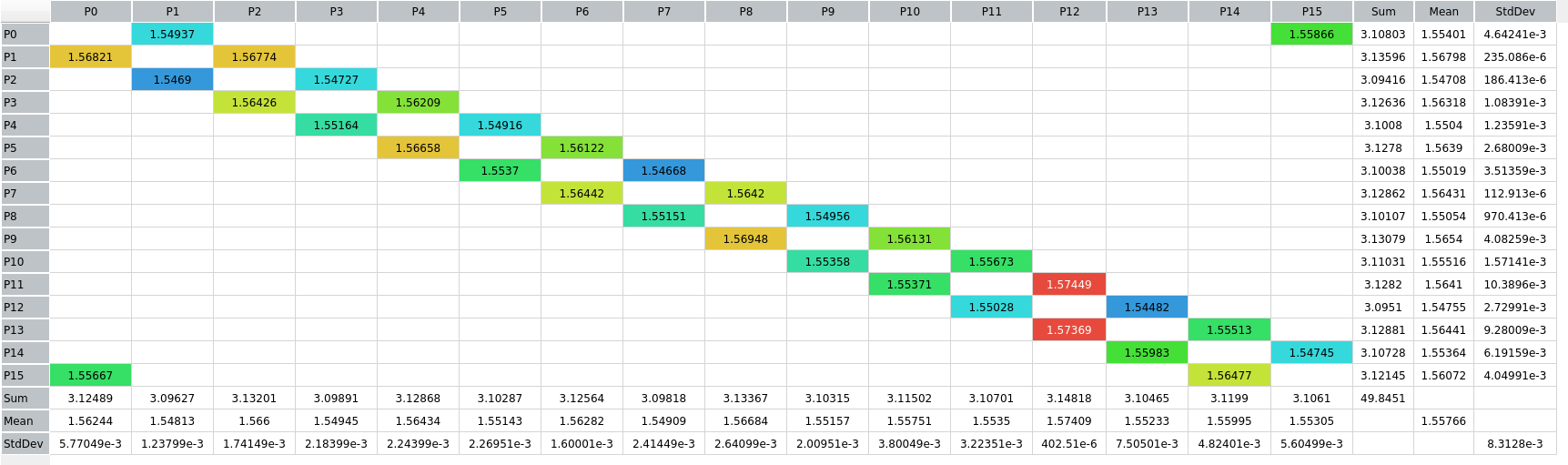
Временная шкала некоторого участка

**

Статистика вызовов функций

**

Статистика сообщений между *MPI*-процессами

**

# Приложение 3. Исходный код программы

#include <stdlib.h>  
#include <stdio.h>  
#include <mpi/mpi.h>  
#include <malloc.h>  
#include "generation.h"  
  
#define **ROOT\_RANK** 0  
#define **START\_CAPACITY** 1  
#define **MAX\_ITERATIONS\_COUNT** 10000  
  
#define **DEBUG**  
  
#ifdef **DEBUG**  
void print\_time(const char\* *name*, double *time\_s*) {  
 printf("%-40s %-8.0f ms\n", *name*, *time\_s* \* 1000);  
}  
#endif  
  
unsigned char get\_alive\_neighbours\_of\_internal\_cell(const *Cell*\* *generation*,  
 unsigned long *width*,  
 const unsigned long *y*,  
 const unsigned long *x*) {  
  
 return **GET\_CELL**(*generation*, *width*, *y* - 1, *x*) +  
 **GET\_CELL**(*generation*, *width*, *y* - 1, (*x* + 1) % *width*) +  
 **GET\_CELL**(*generation*, *width*, *y*, (*x* + 1) % *width*) +  
 **GET\_CELL**(*generation*, *width*, *y* + 1, (*x* + 1) % *width*) +  
 **GET\_CELL**(*generation*, *width*, *y* + 1, *x*) +  
 **GET\_CELL**(*generation*, *width*, *y* + 1, (*width* + *x* - 1) % *width*) +  
 **GET\_CELL**(*generation*, *width*, *y*, (*width* + *x* - 1) % *width*) +  
 **GET\_CELL**(*generation*, *width*, *y* - 1, (*width* + *x* - 1) % *width*);  
}  
  
unsigned char get\_alive\_neighbours\_of\_upper\_cell(const *Cell*\* *generation*,  
 const *Cell*\* *neighbour\_bottom\_row*,  
 unsigned long *width*,  
 const unsigned long *y*,  
 const unsigned long *x*) {  
  
 return *neighbour\_bottom\_row*[*x*] +  
 *neighbour\_bottom\_row*[(*x* + 1) % *width*] +  
 **GET\_CELL**(*generation*, *width*, *y*, (*x* + 1) % *width*) +  
 **GET\_CELL**(*generation*, *width*, *y* + 1, (*x* + 1) % *width*) +  
 **GET\_CELL**(*generation*, *width*, *y* + 1, *x*) +  
 **GET\_CELL**(*generation*, *width*, *y* + 1, (*width* + *x* - 1) % *width*) +  
 **GET\_CELL**(*generation*, *width*, *y*, (*width* + *x* - 1) % *width*) +  
 *neighbour\_bottom\_row*[(*width* + *x* - 1) % *width*];  
}  
  
unsigned char get\_neighbours\_count\_of\_bottom\_cell(const *Cell*\* *generation*,  
 const *Cell*\* *neighbour\_upper\_row*,  
 unsigned long *width*,  
 const unsigned long *y*,  
 const unsigned long *x*) {  
  
 return **GET\_CELL**(*generation*, *width*, *y* - 1, *x*) +  
 **GET\_CELL**(*generation*, *width*, *y* - 1, (*x* + 1) % *width*) +  
 **GET\_CELL**(*generation*, *width*, *y*, (*x* + 1) % *width*) +  
 *neighbour\_upper\_row*[(*x* + 1) % *width*] +  
 *neighbour\_upper\_row*[*x*] +  
 *neighbour\_upper\_row*[(*width* + *x* - 1) % *width*] +  
 **GET\_CELL**(*generation*, *width*, *y*, (*width* + *x* - 1) % *width*) +  
 **GET\_CELL**(*generation*, *width*, *y* - 1, (*width* + *x* - 1) % *width*);  
}  
  
void write\_generation\_to\_file(const char\* *filename*,  
 const *Cell*\* *generation*,  
 unsigned long *local\_height*,  
 unsigned long *width*,  
 int *rank*,  
 int *size*) {  
  
 *MPI\_File* out;  
 MPI\_File\_open(**MPI\_COMM\_WORLD**,  
 *filename*,  
 **MPI\_MODE\_WRONLY** | **MPI\_MODE\_CREATE**,  
 **MPI\_INFO\_NULL**,  
 &out);  
  
 char\* buffer = calloc(*local\_height* \* (*width* + 1), sizeof(char));  
 write\_generation(*generation*, *local\_height*, *width*, buffer);  
  
 for (int i = 0; i < *size*; i++) {  
 MPI\_Barrier(**MPI\_COMM\_WORLD**);  
  
 if (*rank* == i) {  
 MPI\_File\_write\_shared(out,  
 buffer,  
 (int)(*local\_height* \* (*width* + 1)),  
 **MPI\_CHAR**,  
 **MPI\_STATUS\_IGNORE**);  
 }  
 }  
  
 free(buffer);  
 MPI\_File\_close(&out);  
}  
  
void evolve\_cells\_in\_upper\_row(const *Cell*\* *current\_generation*,  
 const *Cell*\* *upper\_neighbour\_bottom\_row*,  
 unsigned long *width*,  
 unsigned long\* *alive\_cells\_count\_in\_next\_generation*,  
 *Cell*\* *next\_generation*) {  
  
 const unsigned long y1 = 0;  
 for (unsigned long x = 0; x < *width*; x++) {  
 const unsigned char alive\_neighbours\_count = get\_alive\_neighbours\_of\_upper\_cell(*current\_generation*,  
 *upper\_neighbour\_bottom\_row*,  
 *width*,  
 y1,  
 x);  
  
 \**alive\_cells\_count\_in\_next\_generation* += evolve(*current\_generation*,  
 *next\_generation*,  
 *width*,  
 alive\_neighbours\_count,  
 y1,  
 x);  
 }  
}  
  
void evolve\_cells\_in\_bottom\_row(const *Cell*\* *current\_generation*,  
 const *Cell*\* *bottom\_neighbour\_upper\_row*,  
 unsigned long *local\_height*,  
 unsigned long *width*,  
 unsigned long\* *alive\_cells\_count\_in\_next\_generation*,  
 *Cell*\* *next\_generation*) {  
  
 const unsigned long y2 = *local\_height* - 1;  
 for (unsigned long x = 0; x < *width*; x++) {  
 const unsigned char alive\_neighbours\_count = get\_neighbours\_count\_of\_bottom\_cell(*current\_generation*,  
 *bottom\_neighbour\_upper\_row*,  
 *width*,  
 y2,  
 x);  
  
 \**alive\_cells\_count\_in\_next\_generation* += evolve(*current\_generation*,  
 *next\_generation*,  
 *width*,  
 alive\_neighbours\_count,  
 y2,  
 x);  
 }  
}  
  
**bool** check\_repeat(const *BitValue*\* *reduced\_stop\_vector*, *size\_t current\_iteration*) {  
 for (*size\_t* j = 0; j < *current\_iteration*; j++) {  
 if (*reduced\_stop\_vector*[j]) {  
 return **true**;  
 }  
 }  
 return **false**;  
}  
  
int main(int *argc*, char\*\* *argv*) {  
 MPI\_Init(&*argc*, &*argv*);  
  
 #ifdef **DEBUG**  
const double start\_time = MPI\_Wtime();  
 #endif  
  
 const unsigned long height = strtoul(*argv*[1], **NULL**, 10);  
 const unsigned long width = strtoul(*argv*[2], **NULL**, 10);  
 const char\* in\_name = *argv*[3];  
 const char\* out\_name = *argv*[4];  
  
 int rank;  
 int size;  
 MPI\_Comm\_rank(**MPI\_COMM\_WORLD**, &rank);  
 MPI\_Comm\_size(**MPI\_COMM\_WORLD**, &size);  
  
 const unsigned long k1 = rank \* height / size;  
 const unsigned long k2 = (rank + 1) \* height / size;  
 const unsigned long local\_height = k2 - k1;  
  
 *Cell*\* current\_generation = **CREATE\_GENERATION**(local\_height, width);  
 *Cell*\* next\_generation = **NULL**;  
  
 *FILE*\* in = fopen(in\_name, "r");  
 fseek(in, (long) (k1 \* (width + 1)), **SEEK\_SET**);  
 unsigned long alive\_cells\_in\_current\_generation = read\_generation(current\_generation,  
 local\_height,  
 width,  
 in);  
  
 *size\_t* capacity = **START\_CAPACITY**;  
 *Cell*\*\* generations\_history = calloc(capacity, sizeof(*Cell*\*));  
 unsigned long\* alive\_cells\_history = calloc(capacity, sizeof(unsigned long));  
 *BitValue*\* local\_stop\_vector = calloc(capacity, sizeof(*BitValue*));  
 *BitValue*\* reduced\_stop\_vector = calloc(capacity, sizeof(*BitValue*));  
 *Cell*\* bottom\_neighbour\_upper\_row = calloc(width, sizeof(*Cell*));  
 *Cell*\* upper\_neighbour\_bottom\_row = calloc(width, sizeof(*Cell*));  
  
 *MPI\_Request* send\_upper\_row\_req;  
 *MPI\_Request* send\_bottom\_row\_req;  
 *MPI\_Request* recv\_bottom\_neighbour\_upper\_row\_req;  
 *MPI\_Request* recv\_upper\_neighbour\_bottom\_row\_req;  
 *MPI\_Request* reducing\_stop\_vectors\_req;  
  
 #ifdef **DEBUG**  
double calculating\_stop\_flags\_vector\_time = 0;  
 double evolving\_time = 0;  
 double waiting\_time\_1 = 0;  
 double waiting\_time\_2 = 0;  
 double waiting\_time\_3 = 0;  
 #endif  
  
 *size\_t* current\_iteration = 0;  
 while (current\_iteration < **MAX\_ITERATIONS\_COUNT**) {  
 *// Initiate receiving upper-neighbour's bottom row*  
MPI\_Irecv(upper\_neighbour\_bottom\_row,  
 (int)width,  
 **MPI\_UINT8\_T**,  
 (size + rank - 1) % size,  
 1,  
 **MPI\_COMM\_WORLD**,  
 &recv\_upper\_neighbour\_bottom\_row\_req);  
  
 *// Initiate receiving bottom-neighbour's upper row*  
MPI\_Irecv(bottom\_neighbour\_upper\_row,  
 (int)width,  
 **MPI\_UINT8\_T**,  
 (rank + 1) % size,  
 0,  
 **MPI\_COMM\_WORLD**,  
 &recv\_bottom\_neighbour\_upper\_row\_req);  
  
 *// Initiate sending bottom row*  
MPI\_Isend(current\_generation + width \* (local\_height - 1),  
 (int)width,  
 **MPI\_UINT8\_T**,  
 (rank + 1) % size,  
 1,  
 **MPI\_COMM\_WORLD**,  
 &send\_bottom\_row\_req);  
  
 *// Initiate sending upper row*  
MPI\_Isend(current\_generation,  
 (int)width,  
 **MPI\_UINT8\_T**,  
 (size + rank - 1) % size,  
 0,  
 **MPI\_COMM\_WORLD**,  
 &send\_upper\_row\_req);  
  
 #ifdef **DEBUG**  
const double t1 = MPI\_Wtime();  
 #endif  
  
 *// Calculate vector of stop-flags*  
for (*size\_t* j = 0; j < current\_iteration; j++) {  
 local\_stop\_vector[j] = alive\_cells\_in\_current\_generation != alive\_cells\_history[j]  
 ? **false**  
: **EQUAL**(current\_generation, generations\_history[j], local\_height, width);  
 }  
  
 #ifdef **DEBUG**  
const double t2 = MPI\_Wtime();  
 #endif  
  
 *// Initiate sending and "land"-ing stop-vector-flags from all to all*  
MPI\_Iallreduce(local\_stop\_vector,  
 reduced\_stop\_vector,  
 (int)current\_iteration,  
 **MPI\_UINT8\_T**,  
 **MPI\_LAND**,  
 **MPI\_COMM\_WORLD**,  
 &reducing\_stop\_vectors\_req);  
  
 next\_generation = **CREATE\_GENERATION**(local\_height, width);  
  
 unsigned long alive\_cells\_in\_next\_generation = 0;  
  
 #ifdef **DEBUG**  
const double t3 = MPI\_Wtime();  
 #endif  
  
 *// Evolve internal cells*  
const unsigned long stop = local\_height - 1;  
 for (unsigned long y = 1; y < stop; y++) {  
 for (unsigned long x = 0; x < width; x++) {  
 const unsigned char alive\_neighbours\_count = get\_alive\_neighbours\_of\_internal\_cell(current\_generation,  
 width,  
 y,  
 x);  
  
 alive\_cells\_in\_next\_generation += evolve(current\_generation,  
 next\_generation,  
 width,  
 alive\_neighbours\_count,  
 y,  
 x);  
 }  
 }  
  
 #ifdef **DEBUG**  
const double t4 = MPI\_Wtime();  
 #endif  
  
 **bool** checked\_repeat = **false**;  
 int flag;  
  
 MPI\_Test(&reducing\_stop\_vectors\_req, &flag, **MPI\_STATUS\_IGNORE**);  
  
 if (flag) {  
 checked\_repeat = **true**;  
  
 if (check\_repeat(reduced\_stop\_vector, current\_iteration)) {  
 break;  
 }  
 }  
  
 #ifdef **DEBUG**  
const double t5 = MPI\_Wtime();  
 #endif  
  
 MPI\_Wait(&recv\_upper\_neighbour\_bottom\_row\_req, **MPI\_STATUS\_IGNORE**);  
  
 #ifdef **DEBUG**  
const double t6 = MPI\_Wtime();  
 #endif  
  
 evolve\_cells\_in\_upper\_row(current\_generation,  
 upper\_neighbour\_bottom\_row,  
 width,  
 &alive\_cells\_in\_next\_generation,  
 next\_generation);  
  
 #ifdef **DEBUG**  
const double t7 = MPI\_Wtime();  
 #endif  
  
 if (!checked\_repeat) {  
 MPI\_Test(&reducing\_stop\_vectors\_req, &flag, **MPI\_STATUS\_IGNORE**);  
  
 if (flag) {  
 checked\_repeat = **true**;  
  
 if (check\_repeat(reduced\_stop\_vector, current\_iteration)) {  
 break;  
 }  
 }  
 }  
  
 #ifdef **DEBUG**  
const double t8 = MPI\_Wtime();  
 #endif  
  
 MPI\_Wait(&recv\_bottom\_neighbour\_upper\_row\_req, **MPI\_STATUS\_IGNORE**);  
  
 #ifdef **DEBUG**  
const double t9 = MPI\_Wtime();  
 #endif  
  
 evolve\_cells\_in\_bottom\_row(current\_generation,  
 bottom\_neighbour\_upper\_row,  
 local\_height,  
 width,  
 &alive\_cells\_in\_next\_generation,  
 next\_generation);  
  
 #ifdef **DEBUG**  
const double t10 = MPI\_Wtime();  
 double t11;  
 double t12;  
 #endif  
  
 if (!checked\_repeat) {  
 #ifdef **DEBUG**  
t11 = MPI\_Wtime();  
 #endif  
  
 MPI\_Wait(&reducing\_stop\_vectors\_req, **MPI\_STATUS\_IGNORE**);  
  
 #ifdef **DEBUG**  
t12 = MPI\_Wtime();  
 #endif  
  
 if (check\_repeat(reduced\_stop\_vector, current\_iteration)) {  
 break;  
 }  
 }  
  
 *// Add current generation to container of previous generations*  
if (capacity == current\_iteration) {  
 capacity \*= 2;  
 generations\_history = reallocarray(generations\_history, capacity, sizeof(*Cell*\*));  
 local\_stop\_vector = reallocarray(local\_stop\_vector, capacity, sizeof(*BitValue*));  
 reduced\_stop\_vector = reallocarray(reduced\_stop\_vector, capacity, sizeof(*BitValue*));  
 alive\_cells\_history = reallocarray(alive\_cells\_history, capacity, sizeof(unsigned long));  
 }  
 generations\_history[current\_iteration] = current\_generation;  
 alive\_cells\_history[current\_iteration] = alive\_cells\_in\_current\_generation;  
  
 alive\_cells\_in\_current\_generation = alive\_cells\_in\_next\_generation;  
  
 current\_iteration++;  
  
 MPI\_Wait(&send\_bottom\_row\_req, **MPI\_STATUS\_IGNORE**);  
 MPI\_Wait(&send\_upper\_row\_req, **MPI\_STATUS\_IGNORE**);  
  
 current\_generation = next\_generation;  
  
 #ifdef **DEBUG**  
evolving\_time += (t4 - t3) + (t7 - t6) + (t10 - t9);  
 calculating\_stop\_flags\_vector\_time += t2 - t1;  
 waiting\_time\_1 += t6 - t5;  
 waiting\_time\_2 += (t9 - t8);  
 waiting\_time\_3 += (t12 - t11);  
 #endif  
 }  
  
 write\_generation\_to\_file(out\_name,  
 current\_generation,  
 local\_height,  
 width,  
 rank,  
 size);  
  
 for (*size\_t* i = 0; i < current\_iteration; i++) {  
 free(generations\_history[i]);  
 }  
 free(generations\_history);  
 free(alive\_cells\_history);  
 free(current\_generation);  
 free(next\_generation);  
 free(upper\_neighbour\_bottom\_row);  
 free(bottom\_neighbour\_upper\_row);  
 free(reduced\_stop\_vector);  
 free(local\_stop\_vector);  
 fclose(in);  
  
 #ifdef **DEBUG**  
const double end\_time = MPI\_Wtime();  
 if (rank == **ROOT\_RANK**) {  
 printf("%-40s %d\n", "Reached max iterations count", current\_iteration == **MAX\_ITERATIONS\_COUNT**);  
 printf("%-40s %zu\n", "Iterations", current\_iteration);  
 print\_time("Evolving", evolving\_time);  
 print\_time("Waiting 1", waiting\_time\_1);  
 print\_time("Waiting 2", waiting\_time\_2);  
 print\_time("Waiting 3", waiting\_time\_3);  
 print\_time("Calculating stop-flags-vector", calculating\_stop\_flags\_vector\_time);  
 print\_time("Total time", end\_time - start\_time);  
 }  
 #endif  
  
 MPI\_Finalize();  
}

# Приложение 4. Исходные коды дополнительных файлов

**generation.h**

#pragma once  
  
#include <aio.h>  
#include <malloc.h>  
#include <stdio.h>  
#include <stdbool.h>  
#include <string.h>  
  
typedef *u\_int8\_t BitValue*;  
typedef *BitValue Cell*;  
  
#define **GET\_CELL**(G, W, Y, X) ((G)[(Y) \* (W) + (X)])  
#define **SET\_CELL**(G, W, Y, X, V) ((G)[(Y) \* (W) + (X)] = (V))  
#define **EQUAL**(G1, G2, H, W) (memcmp((G1), (G2), (H) \* (W) \* sizeof(Cell)) == 0)  
#define **CREATE\_GENERATION**(H, W) (calloc((H) \* (W), sizeof(Cell)))  
#define **EVOLVE\_CELL**(IS\_ALIVE, ALIVE\_NEIGHBOURS) ((ALIVE\_NEIGHBOURS) == 3 || ((IS\_ALIVE) && (ALIVE\_NEIGHBOURS) == 2))  
  
*/\**  
 *\* Reads generation from the file.*  
 *\**  
 *\* Return value — number of alive cells in the generation.*  
 *\*/*  
unsigned long read\_generation(*Cell*\* *generation*, unsigned long *height*, unsigned long *width*, *FILE*\* *file*);  
  
*/\**  
 *\* Writes generation to the buffer.*  
 *\*/*  
void write\_generation(const *Cell*\* *generation*, unsigned long *height*, unsigned long *width*, char\* *buffer*);  
  
*/\**  
 *\* Evolves cell in the old generation.*  
 *\**  
 *\* Return value — whether cell will be alive or not.*  
 *\*/*  
**bool** evolve(const *Cell*\* *old\_generation*,  
 *Cell*\* *new\_generation*,  
 unsigned long *width*,  
 unsigned long *alive\_neighbours\_count*,  
 unsigned long *y*,  
 unsigned long *x*);

**generation.c**

#include "generation.h"  
  
#define **CHAR\_TO\_DIGIT**(C) ((C) - '0')  
#define **DIGIT\_TO\_CHAR**(D) ((D) + '0')  
  
unsigned long read\_generation(*Cell*\* *generation*, unsigned long *height*, unsigned long *width*, *FILE*\* *file*) {  
 unsigned long ret = 0;  
  
 char c;  
 for (unsigned long y = 0; y < *height*; y++) {  
 for (unsigned long x = 0; x < *width*; x++) {  
 fread(&c, sizeof(char), 1, *file*);  
 char cell = **CHAR\_TO\_DIGIT**(c);  
 ret += cell;  
 **SET\_CELL**(*generation*, *width*, y, x, cell);  
 }  
 fread(&c, sizeof(char), 1, *file*);  
 }  
  
 return ret;  
}  
  
void write\_generation(const *Cell*\* *generation*, unsigned long *height*, unsigned long *width*, char\* *buffer*) {  
 for (unsigned long y = 0; y < *height*; y++) {  
 for (unsigned long x = 0; x < *width*; x++) {  
 *buffer*[y \* (*width* + 1) + x] = **DIGIT\_TO\_CHAR**(GET\_CELL(*generation*, *width*, y, x));  
 }  
 *buffer*[y \* (*width* + 1) + *width*] = '\n';  
 }  
}  
  
**bool** evolve(const *Cell*\* *old\_generation*,  
 *Cell*\* *new\_generation*,  
 unsigned long *width*,  
 unsigned long *alive\_neighbours\_count*,  
 unsigned long *y*,  
 unsigned long *x*) {  
  
 const **bool** is\_alive = **GET\_CELL**(*old\_generation*, *width*, *y*, *x*);  
 const **bool** will\_be\_alive = **EVOLVE\_CELL**(is\_alive, *alive\_neighbours\_count*);  
 **SET\_CELL**(*new\_generation*, *width*, *y*, *x*, will\_be\_alive);  
 return will\_be\_alive;  
}