###### 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» ДЖОНА КОНВЕЯ

Студентки 2 курса, группы 21205

**Евдокимовой Дари Евгеньевны**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук, доцент

А.Ю. Власенко

Новосибирск 2023

**СОДЕРЖАНИЕ**

[​ ЦЕЛЬ 3](#__RefHeading___Toc1471_2204003058)

[​ ЗАДАНИЕ 3](#__RefHeading___Toc1473_2204003058)

[​ ОПИСАНИЕ РАБОТЫ 4](#__RefHeading___Toc1475_2204003058)

[​ ЗАКЛЮЧЕНИЕ 6](#__RefHeading___Toc1477_2204003058)

[Приложение 1. Листинг последовательной программы 7](#__RefHeading___Toc1479_2204003058)

[Приложение 2. Листинг параллельной программы 11](#__RefHeading___Toc1481_2204003058)

[Приложение 3. Скрипт для запуска параллельной программы 18](#__RefHeading___Toc1483_2204003058)

[​ Приложение 4. Результаты замеров выполнения работы на кластере 19](#__RefHeading___Toc1487_2204003058)

[​ Приложение 6. Графики результатов вычислений 20](#__RefHeading___Toc1489_2204003058)

[​ Приложение 7. Скрины из traceanalyzer 25](#__RefHeading___Toc1493_2204003058)

# ЦЕЛЬ

1. 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

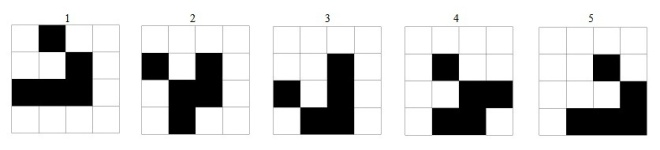
повтору состояния клеточного массива в случае одномерной декомпозиции

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.

1. 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.
2. Произвести профилирование программы и выполнить ее оптимизацию. Попытаться достичь 50-процентной эффективности параллельной реализации на 16 ядрах для выбранных X и Y.
3. Составить отчет, содержащий исходные коды разработанных программ и построенные графики.

# ОПИСАНИЕ РАБОТЫ

1. Был создан файл *sequential.cpp,* в котором написана последовательная программа реализации клеточного автомата. Полный компилируемый листинг программы см. в Приложении 1.
2. Был создан файл *parallel.cpp*, в котором была реализована параллельная программа реализации клеточного автомата с помощью MPI. Полный компилируемый листинг программы см. Приложение 2.
3. Как проверялась корректность работы исполнения программы?

Рис 1. Глайдер   
В качестве исходной фигуры на автомате был выбран глайдер (парусник). Это фигура, в которой единицами инициализированы пять клеток (1,2), (2,3), (3,1), (3,2) и (3,3), согласно рис. 1. Такая конфигурация с периодом в 4 итерации воспроизводит саму себя со смещением на одну клетку по диагонали вправо-вниз.

Через сколько итераций глайдер на поле размером X \* Y вернется в исходное состояние? Ответ: через 4 \* НОК(X, Y) итераций.

1. В программе для тестирования использовалась матрица размером 300 \* 300. Глайдер вернется (1й раз) в исходное состояние через 4 \* НОК(300, 300) = 1200 итераций.
2. Скрипты для запуска на кластере смотреть в Приложении 3.
3. Результаты работы программы представлены в таблице 1. Скрины с кластера смотреть в Приложении 4.

Таблица 1. Результаты замеров времени

|  |  |
| --- | --- |
| Количество процессов | Время работы, сек |
| 1 | 75.262 |
| 2 | 83.9337 |
| 4 | 52.1971 |
| 8 | 28.3995 |
| 12 | 19.3388 |
| 16 | 14.6773 |

1. График зависимости времени, ускорения и эффективности смотреть в Приложении 6.
2. Профилирование было сделано на кафедральном сервере.

Команда для компиляции (не на кластере!):

*mpicxx -o parallel parallel.cpp*

*mpirun -trace -n 16 ./parallel 300 300*

*traceanalyzer parallel.stf*

Здесь 16 — число процессов, 300 — размеры матрицы.

1. Скрины смотреть в приложении 7.

# ЗАКЛЮЧЕНИЕ

В ходе работы мы смогли «распараллелить» программу для реализации клеточного автомата «Жизнь» Дж. Конвея с использованием неблокирующих коммуникаций библиотеки MPI.

# Приложение 1. Листинг последовательной программы

#include <fstream>

#include <iostream>

void generateGlider(int \*data, int rows, int columns) {

data[0 \* columns + 1] = 1;

data[1 \* columns + 2] = 1;

data[2 \* columns + 0] = 1;

data[2 \* columns + 1] = 1;

data[2 \* columns + 2] = 1;

}

void printMatrixToFile(int \*data, int rows, int columns,

std::fstream &file) {

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < columns; ++j) {

file << data[i \* columns + j] << " ";

}

file << "\n";

}

}

void copyMatrix(int \*dest, int \*src, int rows, int columns) {

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < columns; ++j) {

dest[i \* columns + j] = src[i \* columns + j];

}

}

}

// x - rows, y - columns

int countNeighbors(int \*data, int rows, int columns, int xMatr,

int yMatr) {

int sum = 0;

for (int i = -1; i < 2; ++i) { // rows

for (int j = -1; j < 2; ++j) { // columns

int currRow = (xMatr + i + rows) % rows;

int currColumn = (yMatr + j + columns) % columns;

sum += data[currRow \* columns + currColumn];

}

}

sum -= data[xMatr \* columns + yMatr]; // cell itself

// std::cout << "SUMMA is: " << sum << std::endl << std::endl;

return sum;

}

void computeNextGeneration(int \*oldData, int \*nextData, int rows,

int columns) {

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < columns; ++j) {

int state = oldData[i \* columns + j];

// std::cout << i << " " << j << " ";

int neighborsAmount =

countNeighbors(oldData, rows, columns, i, j);

if (state == 0 && neighborsAmount == 3) {

nextData[i \* columns + j] = 1;

} else if (state == 1 &&

(neighborsAmount < 2 || neighborsAmount > 3)) {

nextData[i \* columns + j] = 0;

} else {

nextData[i \* columns + j] = oldData[i \* columns + j] = state;

}

}

}

}

bool equalsToPrevEvolution(int \*prevMatr, int \*currMatrix, int rows,

int columns) {

for (int i = 0; i < rows; i++) {

for (int j = 0; j < columns; j++) {

if (currMatrix[i \* columns + j] != prevMatr[i \* columns + j]) {

return false;

}

}

}

return true;

}

int main(int argc, char \*\*argv) {

if (argc != 3) {

std::cout << "Bad amount of arguments!\n"

<< "Enter rows amount, then enter columns amount.\n"

<< std::endl;

return 0;

}

const int rowsAmount = atoi(argv[1]);

const int columnsAmount = atoi(argv[2]);

std::fstream inputFile;

inputFile.open("start.txt",

std::ios::in | std::ios::out | std::ios::trunc);

if (!inputFile) {

std::cout << "Can't open input file\n";

return 0;

}

int \*currentGen = new int[rowsAmount \* columnsAmount]();

generateGlider(currentGen, rowsAmount, columnsAmount);

printMatrixToFile(currentGen, rowsAmount, columnsAmount, inputFile);

std::fstream outputFile;

outputFile.open("end.txt", std::ios::out | std::ios::trunc);

if (!outputFile) {

std::cout << "Can't open output file\n";

delete[] currentGen;

inputFile.close();

return 0;

}

outputFile << "start matrix -------------" << std::endl;

printMatrixToFile(currentGen, rowsAmount, columnsAmount,

outputFile);

int \*nextGen = new int[rowsAmount \* columnsAmount]();

const long maxIterations = 1000;

int \*\*historyOfEvolution = new int \*[maxIterations]();

int iterCurr = 0;

bool repeated = false;

while (iterCurr < maxIterations && !repeated) {

historyOfEvolution[iterCurr] = new int[rowsAmount \* columnsAmount]();

copyMatrix(historyOfEvolution[iterCurr], currentGen, rowsAmount,

columnsAmount);

computeNextGeneration(currentGen, nextGen, rowsAmount,

columnsAmount);

outputFile << "after iter: " << iterCurr <<

"-----------------"

<< std::endl;

printMatrixToFile(nextGen, rowsAmount, columnsAmount,

outputFile);

copyMatrix(currentGen, nextGen, rowsAmount, columnsAmount);

for (int i = iterCurr; i > -1; --i) {

if (equalsToPrevEvolution(historyOfEvolution[i], nextGen, rowsAmount,

columnsAmount)) {

// std::cout << "iter " << iterCurr << ": equals" <<

// std::endl; // in real: iter + 1

repeated = true;

break;

}

}

iterCurr++;

// std::cout << "curr iteration: " << iterCurr << std::endl;

}

if(repeated) {

std::cout << "First repeat of cell automat stage after: "

<< iterCurr << " iterCurr" << std::endl;

}

else {

std::cout << "Finished after iteration: " << iterCurr << std::endl;

}

inputFile.close();

outputFile.close();

delete[] currentGen;

delete[] nextGen;

for (int i = 0; i < maxIterations; i++) {

delete[] historyOfEvolution[i];

}

delete[] historyOfEvolution;

return 0;

}

# Приложение 2. Листинг параллельной программы

#include "mpi.h"

#include <iostream>

#include <iterator>

#include <vector>

void generateGlider(bool \*extendedPartMatr, int columnsAmount) {

std::fill(startMatrix, startMatrix + columnsAmount \* rowsAmount,

false);

extendedPartMatr[0 \* columnsAmount + 1] = true;

extendedPartMatr[1 \* columnsAmount + 2] = true;

extendedPartMatr[2 \* columnsAmount + 0] = true;

extendedPartMatr[2 \* columnsAmount + 1] = true;

extendedPartMatr[2 \* columnsAmount + 2] = true;

}

bool equalsMatrices(const bool \*first, const bool \*second,

int realStart, int realEnd) {

for (int i = realStart; i < realEnd; ++i) {

if (first[i] != second[i]) {

return false;

}

}

return true;

}

void calcStopVectors(std::vector<bool \*> historyOfEvolution,

bool \*stopVector, bool \*extendedPartMatr,

int rowsAmount, int columnsAmount) {

int vectorSize = historyOfEvolution.size() - 1;

auto it = historyOfEvolution.begin();

for (int i = 0; i < vectorSize; ++i) {

stopVector[i] =

equalsMatrices(\*it, extendedPartMatr, columnsAmount,

columnsAmount \* (rowsAmount + 1));

it++;

}

}

bool isStop(int rowsAmount, int columnsAmount,

const bool \*stopMatrix) {

for (int i = 0; i < columnsAmount; ++i) {

bool stop = true;

for (int j = 0; j < rowsAmount; ++j) {

stop &= stopMatrix[j \* columnsAmount + i];

}

if (stop)

return true;

}

return false;

}

int countNeighbors(bool \*oldData, int columnsAmount, int i, int j) {

int neighboirsAmount =

(oldData[i \* columnsAmount + (j + 1) % columnsAmount]) +

(oldData[i \* columnsAmount +

(j + columnsAmount - 1) % columnsAmount]) +

(oldData[(i + 1) \* columnsAmount + (j + 1) % columnsAmount]) +

(oldData[(i + 1) \* columnsAmount +

(j + columnsAmount - 1) % columnsAmount]) +

(oldData[(i - 1) \* columnsAmount + (j + 1) % columnsAmount]) +

(oldData[(i - 1) \* columnsAmount +

(j + columnsAmount - 1) % columnsAmount]) +

(oldData[(i + 1) \* columnsAmount + j]) +

(oldData[(i - 1) \* columnsAmount + j]);

return neighboirsAmount;

}

void computeNextGeneration(bool \*oldData, bool \*nextData,

int rowsAmount, int columnsAmount) {

for (int i = 1; i < rowsAmount - 1; ++i) {

for (int j = 0; j < columnsAmount; ++j) {

int state = oldData[i \* columnsAmount + j];

int neighborsAmount =

countNeighbors(oldData, columnsAmount, i, j);

if (state == 0 && neighborsAmount == 3) {

nextData[i \* columnsAmount + j] = 1;

} else if (state == 1 &&

(neighborsAmount < 2 || neighborsAmount > 3)) {

nextData[i \* columnsAmount + j] = 0;

} else {

nextData[i \* columnsAmount + j] =

oldData[i \* columnsAmount + j] = state;

}

}

}

}

int \*countElemsNumInEachProc(int amountOfProcs, int rows,

int columns) {

int \*elemsNum = new int[amountOfProcs];

int basicRowsCount = rows / amountOfProcs;

int restRowsCount = rows % amountOfProcs;

for (int i = 0; i < amountOfProcs; i++) {

elemsNum[i] = basicRowsCount \* columns;

if (restRowsCount > 0) {

elemsNum[i] += columns;

--restRowsCount;

}

}

return elemsNum;

}

int \*countRowsInEachProcess(const int \*elementsNumberArr,

int amountOfProcs, int columns) {

int \*rowsNumArr = new int[amountOfProcs];

for (int i = 0; i < amountOfProcs; i++) {

rowsNumArr[i] = elementsNumberArr[i] / columns;

}

return rowsNumArr;

}

int \*createElemsOffsetArr(const int \*elemsNum, int amountOfProcs) {

int \*elementsOffsetArray = new int[amountOfProcs];

int elementsOffset = 0;

for (int i = 0; i < amountOfProcs; i++) {

elementsOffsetArray[i] = elementsOffset;

elementsOffset += elemsNum[i];

}

return elementsOffsetArray;

}

int \*createRowsOffsetArr(const int \*elementsOffsetArray,

int amountOfProcs, int rows) {

int \*rowsOffsetArray = new int[amountOfProcs];

for (int i = 0; i < amountOfProcs; i++) {

rowsOffsetArray[i] = elementsOffsetArray[i] / rows;

}

return rowsOffsetArray;

}

void startLife(int amountOfProcs, int rankOfCurrProc,

bool \*startMatrix, int rowsAmount, int columnsAmount) {

// amount of elems, handling each process

int \*elemsInEachProc = countElemsNumInEachProc(amountOfProcs, rowsAmount,

columnsAmount);

// amount of rows, which each process handles

int \*rowsNumArr =

countRowsInEachProcess(elemsInEachProc, amountOfProcs, columnsAmount);

// count, from which element data sends to each process

int \*offset = createElemsOffsetArr(elemsInEachProc, amountOfProcs);

// count, from which row data sends to each process

int \*rowsOffsetArray =

createRowsOffsetArr(offset, amountOfProcs, rowsAmount);

// if (rankOfCurrProc == 0) {

// for (int i = 0; i < amountOfProcs; i++) {

// std::cout << i << ": " << rowsNumArr[i] << std::endl;

// }

// }

bool \*extendedPartMatr =

new bool[elemsInEachProc[rankOfCurrProc] + columnsAmount \* 2];

bool \*basePartMatr = extendedPartMatr + columnsAmount;

MPI\_Scatterv(startMatrix, elemsInEachProc, offset, MPI\_C\_BOOL, basePartMatr,

elemsInEachProc[rankOfCurrProc], MPI\_C\_BOOL, 0, MPI\_COMM\_WORLD);

int prevRank = (rankOfCurrProc + amountOfProcs - 1) % amountOfProcs;

int nextRank = (rankOfCurrProc + 1) % amountOfProcs;

std::vector<bool \*> historyOfEvolution;

int iteration = 0;

bool stop = false;

while (!stop) {

bool \*extendedNextPartMatr =

new bool[elemsInEachProc[rankOfCurrProc] + 2 \* columnsAmount];

bool \*baseNextPartMatr = extendedNextPartMatr + columnsAmount;

historyOfEvolution.push\_back(extendedPartMatr);

iteration++;

MPI\_Request requestSendFirstLine;

// 1 - initiation of sending first line to the prev core

MPI\_Isend(basePartMatr, columnsAmount, MPI\_C\_BOOL, prevRank, 1,

MPI\_COMM\_WORLD, &requestSendFirstLine);

MPI\_Request requestSendLastLine;

// 2 - initiation of sending last line the to next core

MPI\_Isend(basePartMatr + elemsInEachProc[rankOfCurrProc] - columnsAmount,

columnsAmount, MPI\_C\_BOOL, nextRank, 0, MPI\_COMM\_WORLD,

&requestSendLastLine);

MPI\_Request requestGetLastLine;

// 3 - initiation of receiving last line from the previous

// core

MPI\_Irecv(extendedPartMatr, columnsAmount, MPI\_C\_BOOL, prevRank,

0, MPI\_COMM\_WORLD, &requestGetLastLine);

MPI\_Request requestGetFirstLine;

// 4 - initiation of receiving first line from the next core

MPI\_Irecv(basePartMatr + elemsInEachProc[rankOfCurrProc], columnsAmount,

MPI\_C\_BOOL, nextRank, 1, MPI\_COMM\_WORLD,

&requestGetFirstLine);

// 5 - count vector of stop flags

MPI\_Request flagsReq;

bool \*stopVector;

bool \*stopMatrix;

int vectorSize = historyOfEvolution.size() - 1;

if (vectorSize > 1) {

stopVector = new bool[vectorSize];

calcStopVectors(historyOfEvolution, stopVector,

extendedPartMatr, rowsNumArr[rankOfCurrProc],

columnsAmount);

stopMatrix = new bool[vectorSize \* amountOfProcs];

// 6 - init changing of stop vectors with all cores

MPI\_Iallgather(stopVector, vectorSize, MPI\_C\_BOOL,

stopMatrix, vectorSize, MPI\_C\_BOOL,

MPI\_COMM\_WORLD, &flagsReq);

}

// 7 - count stages of rows, except first and last line

computeNextGeneration(basePartMatr, baseNextPartMatr,

rowsNumArr[rankOfCurrProc], columnsAmount);

MPI\_Status status;

// 8 - wait end sending 1st line to prev core

MPI\_Wait(&requestSendFirstLine, &status);

// 9 - wait end of receiving from the 3rd step

MPI\_Wait(&requestGetLastLine, &status);

// 10 - count stages of the first line

computeNextGeneration(extendedPartMatr, extendedNextPartMatr, 3,

columnsAmount);

// 11 - wait end sending last line to the next core

MPI\_Wait(&requestSendLastLine, &status);

// 12 - wait end receiving

MPI\_Wait(&requestGetFirstLine, &status);

// 13 - count stages of the last line

computeNextGeneration(

basePartMatr +

(rowsNumArr[rankOfCurrProc] - 2) \* columnsAmount,

baseNextPartMatr +

(rowsNumArr[rankOfCurrProc] - 2) \* columnsAmount,

3, columnsAmount);

if (vectorSize > 1) {

// 14 - wait end of exchanging stop vectors with each other

// process

MPI\_Wait(&flagsReq, &status);

// 15 - compare vectors of stop

stop = isStop(amountOfProcs, vectorSize, stopMatrix);

delete[] stopVector;

delete[] stopMatrix;

}

if (stop) {

break;

}

extendedPartMatr = extendedNextPartMatr;

basePartMatr = baseNextPartMatr;

}

if (rankOfCurrProc == 0) {

std::cout << "Repeat after: " << iteration - 1 << " iterations"

<< std::endl;

}

for (auto matrix : historyOfEvolution) {

delete[] matrix;

}

historyOfEvolution.clear();

delete[] offset;

delete[] elemsInEachProc;

}

int main(int argc, char \*argv[]) {

if (argc != 3) {

std::cout

<< "Bad amount of arguments!\n"

<< "Enter rowsAmount amount, then enter columns amount.\n"

<< std::endl;

return 0;

}

const int rowsAmount = std::atoi(argv[1]);

const int columnsAmount = std::atoi(argv[2]);

MPI\_Init(&argc, &argv);

int amountOfProcs, rankOfCurrProc;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &amountOfProcs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rankOfCurrProc);

bool \*startMatrix = nullptr;

if (rankOfCurrProc == 0) {

startMatrix = new bool[rowsAmount \* columnsAmount];

generateGlider(startMatrix, columnsAmount);

}

double startt = MPI\_Wtime();

startLife(amountOfProcs, rankOfCurrProc, startMatrix, rowsAmount,

columnsAmount);

double endt = MPI\_Wtime();

if (rankOfCurrProc == 0) {

std::cout << "Time taken: " << endt - startt << " sec."

<< std::endl;

}

if (rankOfCurrProc == 0)

delete[] startMatrix;

MPI\_Finalize();

return 0;

}

# Приложение 3. Скрипт для запуска параллельной программы

#!/bin/bash

#PBS -l walltime=00:05:00

#PBS -l select=2:ncpus=8:mpiprocs=8:mem=10000m

#PBS -m n

#PBS -N task\_life

cd $PBS\_O\_WORKDIR

MPI\_NP=$(wc -l $PBS\_NODEFILE | awk '{ print $1 }')

echo "Number of MPI process: $MPI\_NP"

mpiicpc -O0 parallel.cpp -o parallel -std=c++14

mpirun -hostfile $PBS\_NODEFILE -np $MPI\_NP ./parallel 300 300

# Приложение 4. Результаты замеров выполнения работы на кластере

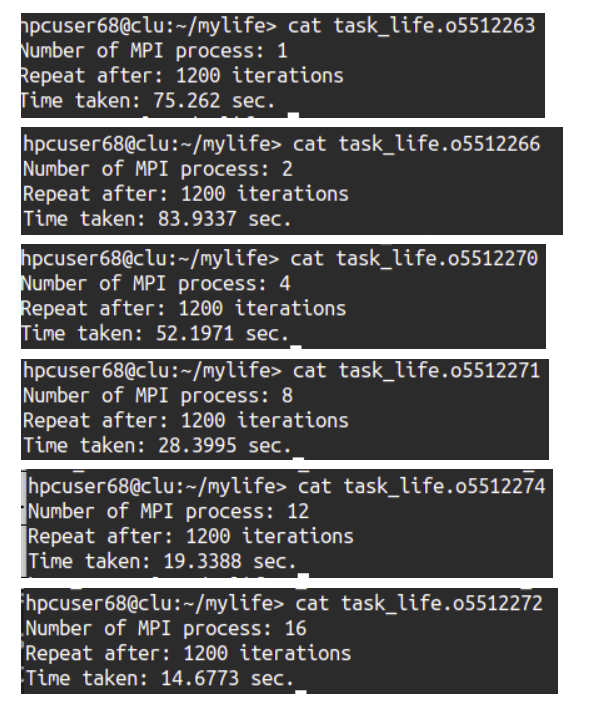
****

Рис.1. Результаты замеров выполнения работы на кластере

# Приложение 6. Графики результатов вычислений

Рис. 1. График зависимости времени от количества процессов

Рис. 2. График ускорения

Рис 3. График эффективности

# Приложение 7. Скрины из traceanalyzer

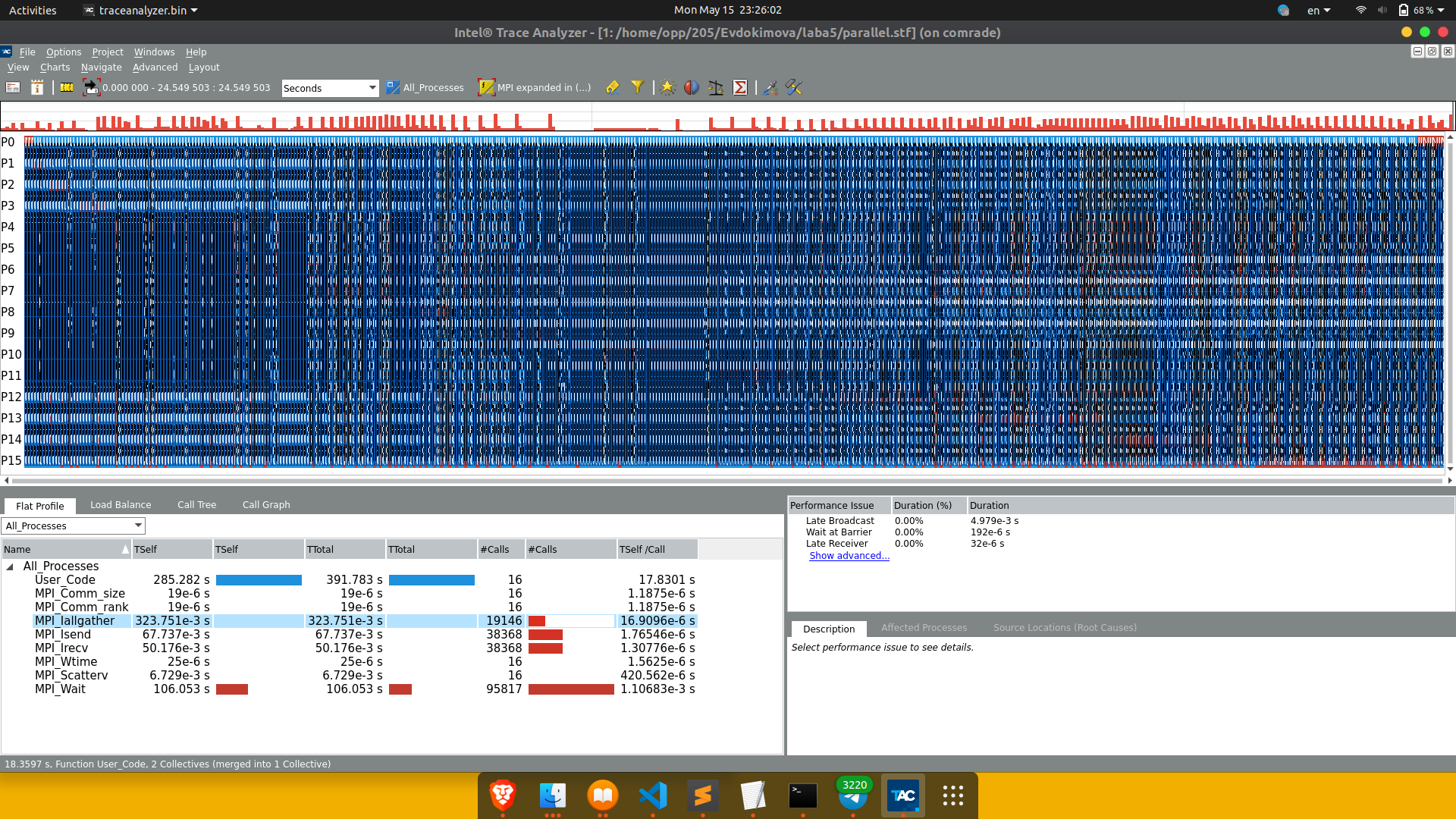


Рис. 1. Общий работы всей программы и общее время работы функций

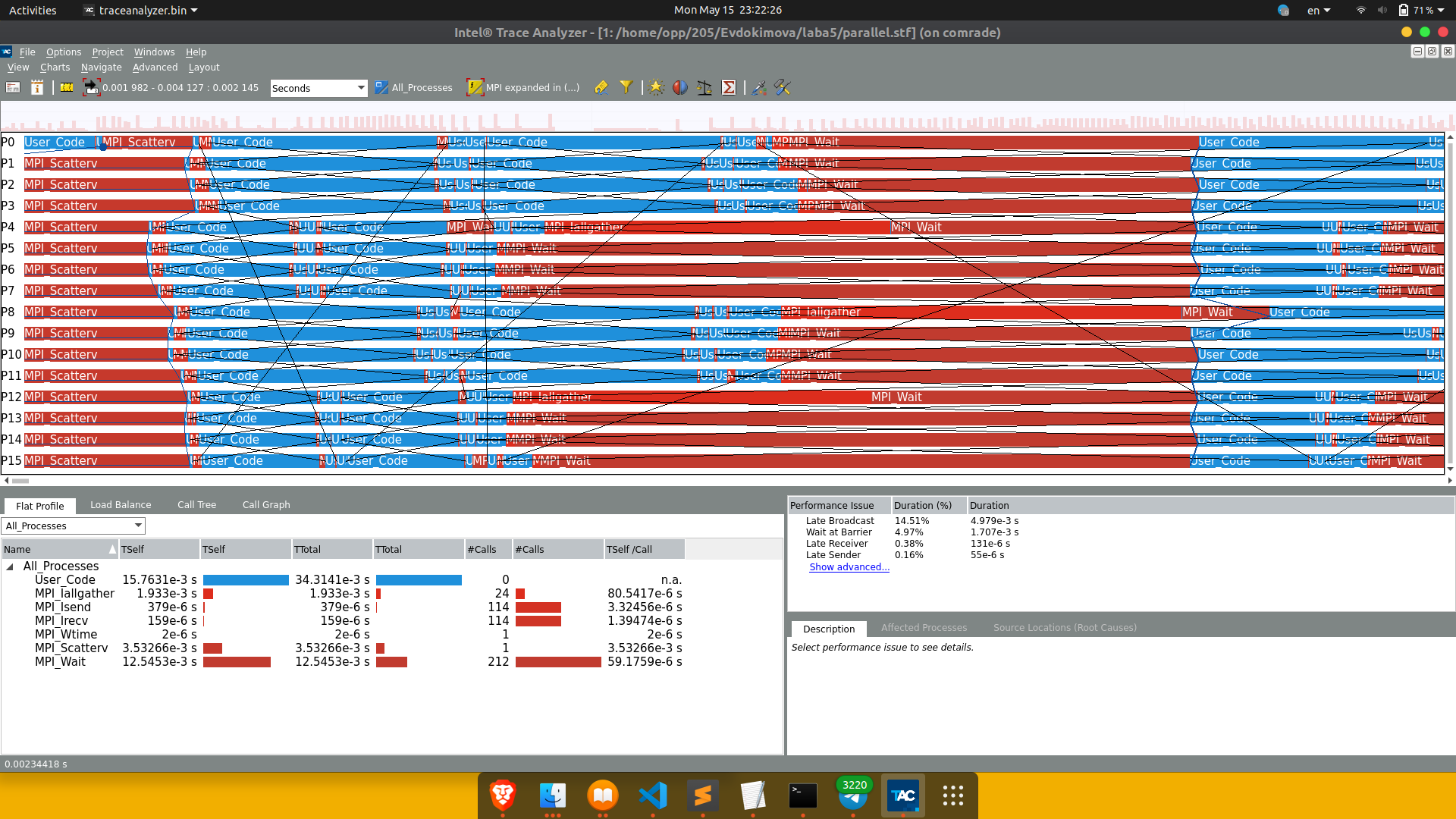


Рис. 2. Начало работы программы

Черные линии - это раздача первой и последней строки части матрицы

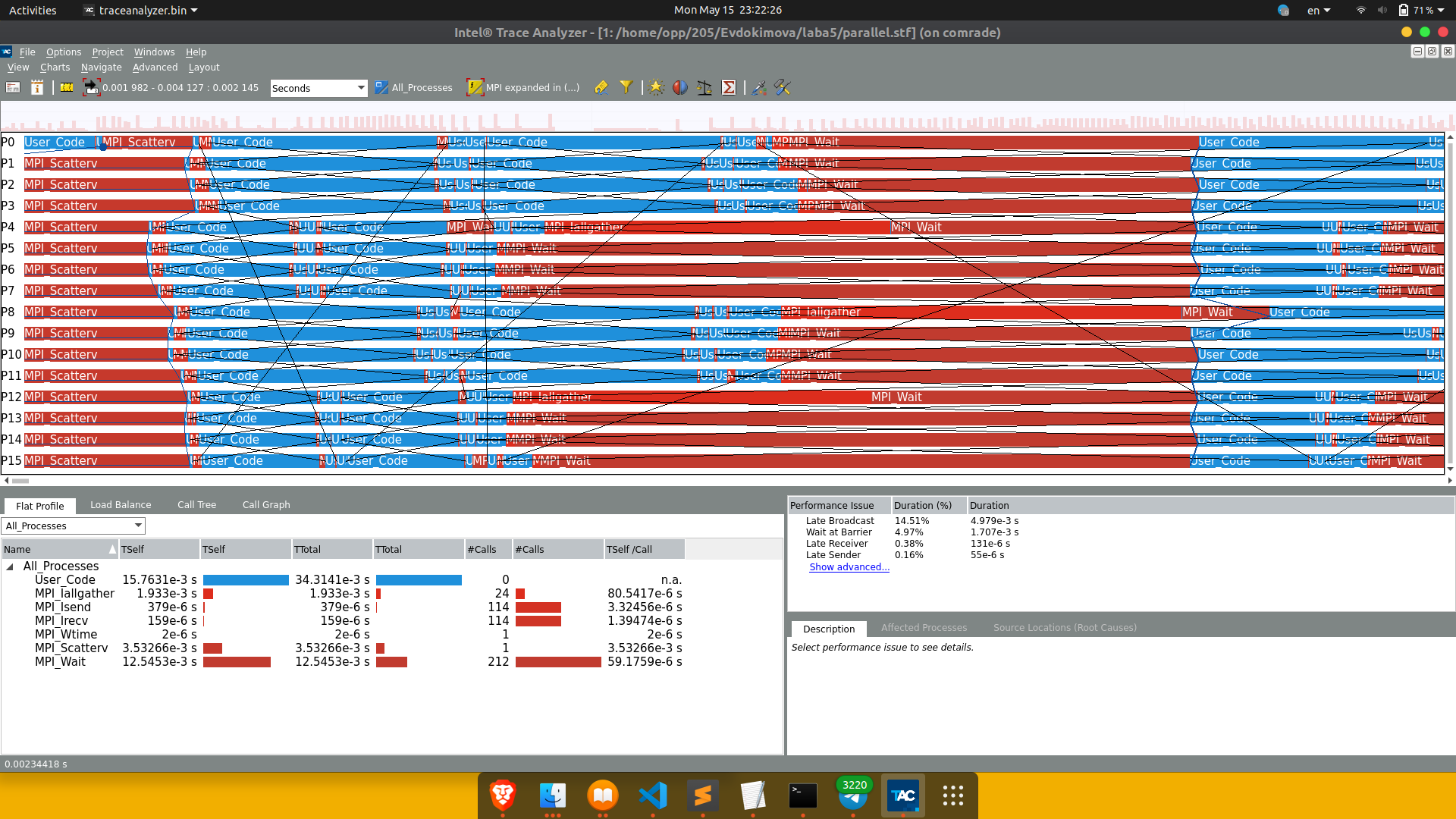


Рис. 3. Использование Allgather

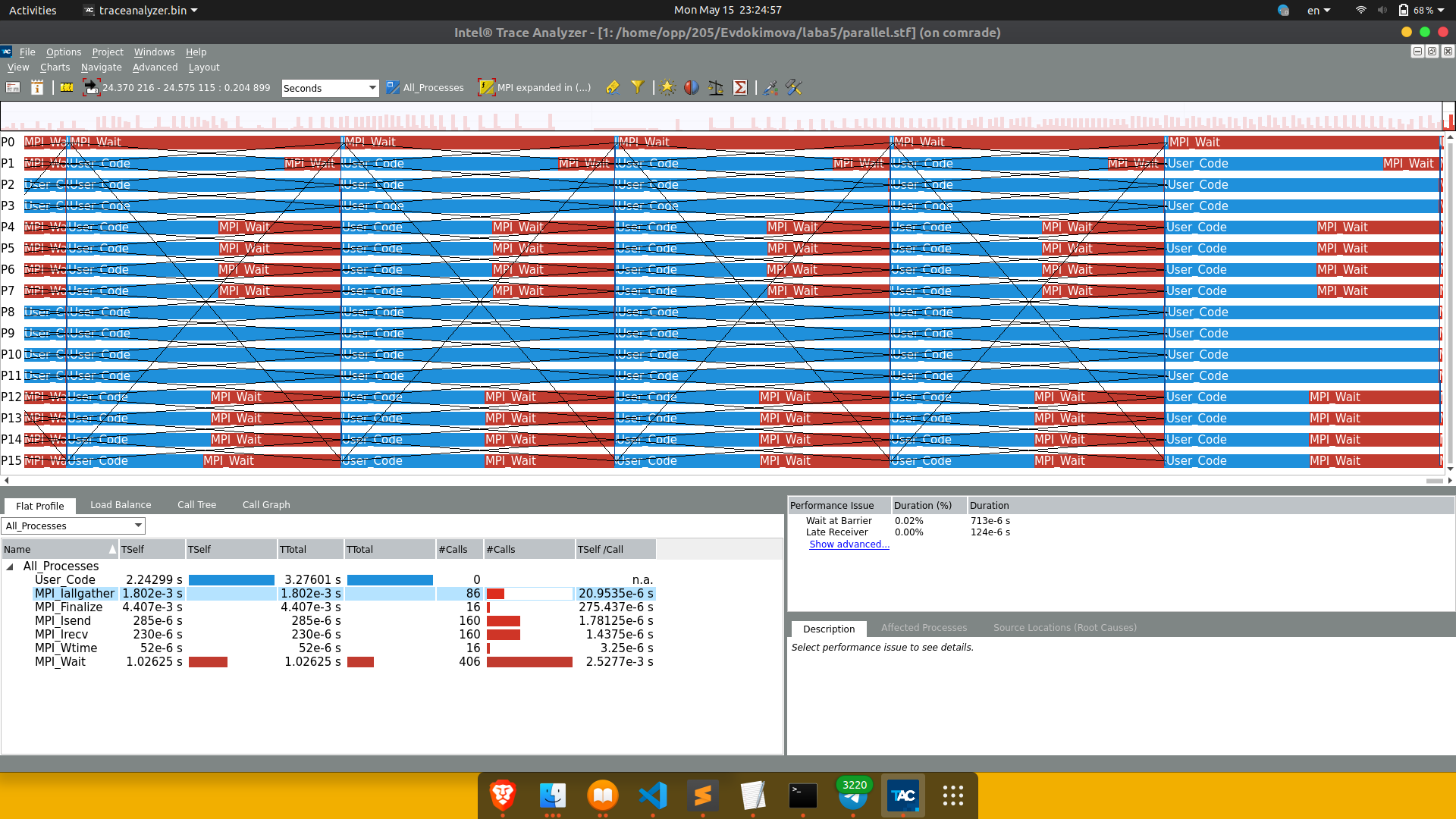


Рис. 4. Окончание работы программы

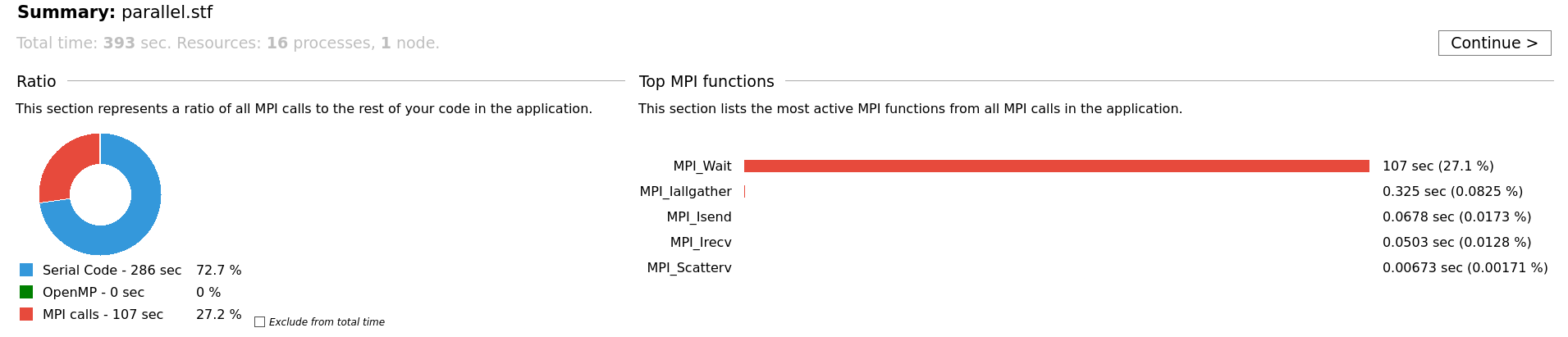


Рис. 5. Общее время работы функций на диаграмме