###### 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

ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ РЕШЕНИЯ СИСТЕМЫ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИЙ УРАВНЕНИЙ С ПОМОЩЬЮ MPI

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](#_Toc129722850)

[ЗАДАНИЕ 3](#_Toc129722851)

[ОПИСАНИЕ РАБОТЫ 4](#_Toc129722852)

[ЗАКЛЮЧЕНИЕ 6](#_Toc129722853)

[Приложение 1. Листинг последовательной программы 7](#_Toc129722854)

[Приложение 2. Проверка корректности работы последовательной программы 13](#_Toc129722855)

[Приложение 3. Скрипт для запуска последовательной программы 14](#_Toc129722856)

[Приложение 4. Листинг параллельной программы с использованием функций MPI 15](#_Toc129722857)

[Приложение 4. Проверка корректности работы параллельной программы 24](#_Toc129722858)

[Приложение 5. Скрипт для работы параллельной программы 25](#_Toc129722859)

[Приложение 6. Результаты работы последовательной и параллельной на 2, 4, 8, 16, 24 процессах 26](#_Toc129722860)

[Приложение 7. Графики времени, ускорения и эффективности 27](#_Toc129722861)

[Приложение 8. Скрипт для профилирования параллельной программы 28](#_Toc129722862)

[Приложение 9. Скрины из traceanalyzer для 16 процессов 29](#_Toc129722863)

# ЦЕЛЬ

1. Написание решения СЛАУ итерационным методом с помощью последовательной и параллельных программ.
2. Ознакомление и работа с кластером НГУ.
3. Изучение профилирования.
4. Сравнение времени работы, ускорения и эффективности последовательной и параллельной программ.

# ЗАДАНИЕ

1. Написать 2 программы (последовательную и параллельную с  
   использованием MPI) на языке C/C++, которые реализуют итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b в соответствии с выбранным вариантом. Здесь A– матрица размером N×N, x и b – векторы длины N.   
   Тип элементов – double.
2. Параллельную программу реализовать с тем условием, что матрица A и вектор b инициализируются на каком-либо одном процессе, а затем матрица A «разрезается» по строкам на близкие по размеру, возможно не одинаковые, части, а вектор b раздается каждому процессу.  
   Уделить внимание тому, чтобы при запуске программы на различном

числе MPI-процессов решалась одна и та же задача (исходные данные

заполнялись одинаковым образом).

1. Замерить время работы последовательной программы и параллельных на 2, 4, 8, 16, 24 процессах.
2. Построить графики времени, ускорения и эффективности распараллеливания от числа используемых ядер.  
   Исходные данные, параметры N и ε подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.
3. Выполнить профилирование программы с помощью jumpshot или ITAC (Intel Trace Analyzer and Collecter) при использовании 16-и/или 24-х ядер. На основании полученных результатов сделать вывод.
4. Составить отчет, содержащий исходные коды разработанных программ и построенные графики.

# ОПИСАНИЕ РАБОТЫ

Выбранный метод – метод минимальных невязок.

1. Был создан файл *nompi.cpp*, в котором была реализована последовательная программа СЛАУ методом минимальных невязок и добавлено время измерения программы с помощью функции clock\_gettime().Полный компилируемый листинг программы см. Приложение 1.
2. Корректность работы программы была проверена со следующими входными параметрами: размер матрицы равен 10.

Элементы главной диагонали A равны 2.0, остальные равны 1.0. Формируется вектор u, элементы которого заполняются произвольными значениями, например, . Элементы вектора b получаются путем умножения матрицы A на вектор u. В этом случае решением системы будет вектор, равный вектору u. Начальные значения элементов вектора x равны 0.

1. Результаты проверки корректности см. Приложение 2. Команда для компиляции:

*g++ -o nompi nompi.cpp*

*./nompi 10*

1. Далее началась работа на кластере.   
   Размер матрицы, при котором время программы составляет не менее 30сек: 2000.
2. Скрипт (run\_nompi.sh) для запуска программы nompi.cpp смотреть в Приложении 3.
3. Итак, для последовательной программы время работы составляет 51,00640 секунд, размер матрицы равен 2000.
4. Был создан файл withmpi.cpp, в котором реализован итерационный алгоритм решения СЛАУ методом минимальных невязок при помощи MPI-функций. Полный компилируемый листинг программы см. в Приложении 4.
5. Результаты проверки (на нескольких процессах) корректности см. Приложение 5. Команда для компиляции (не на кластере!):

*mpicxx -oversubscribe -o withmpi withmpi.cpp*

*./withmpi 10*

1. Затем началась работа на кластере. Был написан скрипт run\_withmpi.sh (см. Приложение 6) для параллельной программы.
2. Было измерено время работы программы на 2, 4, 8, 16, 24 процессах. Скрипт представлен в Приложении 5.
3. Результаты измерения последовательной и параллельной программ на 2, 4, 8, 16, 24 процессах представлен в Приложении 6.
4. **Важно!** Компиляция на последовательной и параллельной программах должна производиться одним и тем же компилятором – от Intel (icpc, mpiicpc).

В случае, если последовательная программа скомпилирована с помощью компилятора GNU, а параллельная – mpiicpc, то, во-первых, это не корректное сравнение по времени, а во-вторых, параллельная будет показывать очень большое ускорение. Это связано с тем, что на компиляторе GNU (g++) по дефолту уровень оптимизации – O0, а на Intel – O2, который и дает большое ускорение (<https://www.cc.kyushu-u.ac.jp/scp/eng/system/ITO/04-1_intel_compiler.html>).

1. Полученные результаты измерения времени представлены в Таблице 1.

Таблица1. Результаты измерений времени

|  |  |  |
| --- | --- | --- |
| Количество процессов | Время работы последовательной  программы, с | Время работы параллельной программы, с |
| 2 | 53,00770 | 29,26980 |
| 4 | 21,17320 |
| 8 | 23,21080 |
| 16 | 4,08491 |
| 24 | 2,96406 |

1. На основании полученных данных рассчитаем эффективность и ускорение. Графики результатов представлены в Приложении 7.
2. Было проведено профилирование программы на 16 процессах. Скрипт для запуска профилирования смотреть в Приложении 8.
3. Скрины из traceanalyzer смотреть в Приложении 9. Для ознакомления с профилированием использовался следующий сайт: <https://www.cism.ucl.ac.be/Services/Formations/ICS/intel-2018.0.128/itac/2018.0.015/doc/get_started.htm> .
4. Выводы представлены в заключении.

# ЗАКЛЮЧЕНИЕ

В ходе работы мы смогли «распараллелить» программу для решения СЛАУ посредством использования функций MPI. Оказалось, что увеличение числа процессов может ускорить время работы программы.

Из результатов анализа профилирования видно, что мнго времени тратится на Bcast, т. к. на каждой итерации вектор X передавался всем процессам; тау, логическая переменная конца завершения цикла и количество итераций — это хоть и переменные (не такие большие, как массив (т. е. вектор X)), но на передачу их значений всем процессам тратится достаточно много времени.

Так же много времени забирает функция Allgatherv, поскольку в методе минимальных невязок 2 умножения матрицы на вектор (тяжелая операция), т. е. мы должны с каждого процесса собрать данные.

Меньше всего времени тратится на Allreduce, поскольку используется для редукции только дважды.

По результатам графиков времени, ускорения и эффективности можно сделать вывод о том, что при увеличении числа процессов увеличиваются так же и затраты на передачу между ними, что негативно сказывается на показателях работы программы.

# Приложение 1. Листинг последовательной программы

#include <cmath>

#include <cstdlib>

#include <iostream>

void printMatrix(double \*matrix, const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; ++i) {

    for (size\_t j = 0; j < sizeInput; ++j) {

      std::cout << matrix[j + i \* sizeInput] << "\t";

    }

    std::cout << std::endl;

  }

  std::cout << std::endl;

}

void printVector(const double \*vector, const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; ++i) {

    std::cout << std::fixed << vector[i] << " ";

  }

  std::cout << std::endl;

}

double \*fillRandomMatrix(const size\_t sizeInput) {

  double \*matrixA = new double[sizeInput \* sizeInput];

  for (size\_t i = 0; i < sizeInput; ++i) {

    for (size\_t j = 0; j < sizeInput; ++j) {

      if (i == j) {

        matrixA[i \* sizeInput + j] = 9999;

      } else {

        matrixA[i \* sizeInput + j] = rand() % 500 + 150.15;

      }

    }

  }

  return matrixA;

}

double \*fillConstantMatrix(const size\_t sizeInput) {

  double \*matrixA = new double[sizeInput \* sizeInput];

  for (size\_t i = 0; i < sizeInput; ++i) {

    for (size\_t j = 0; j < sizeInput; ++j) {

      if (i == j) {

        matrixA[i \* sizeInput + j] = 2.0;

      } else {

        matrixA[i \* sizeInput + j] = 1.0;

      }

    }

  }

  return matrixA;

}

double \*fillRandomVector(const size\_t sizeInput) {

  double \*vector = new double[sizeInput];

  for (size\_t i = 0; i < sizeInput; ++i) {

    vector[i] = rand() % 500;

  }

  return vector;

}

double \*fillConstantVector(const size\_t sizeInput) {

  double \*vector = new double[sizeInput];

  for (size\_t i = 0; i < sizeInput; ++i) {

    vector[i] = sizeInput + 1.0;

  }

  return vector;

}

void multiplyMatrixOnVector(const double \*matrix,

                             const double \*vector, double \*res,

                             const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; ++i) {

    res[i] = 0;

    for (size\_t j = 0; j < sizeInput; ++j) {

      res[i] += matrix[i \* sizeInput + j] \* vector[j];

    }

  }

}

double \*fillVectorU(const size\_t sizeInput) {

  double \*vector = new double[sizeInput];

  for (size\_t i = 0; i < sizeInput; ++i) {

    vector[i] = sin(2 \* 3.1415 \* i / sizeInput);

  }

  return vector;

}

void fillVectorB(double \*b, const double \*fullMatrixA,

                 size\_t sizeInput) {

  double \*vectorU = fillVectorU((int)sizeInput);

  // std::cout << "vector U is: " << std::endl;

  // printVector(vectorU, sizeInput);

  multiplyMatrixOnVector(fullMatrixA, vectorU, b, sizeInput);

  delete[] vectorU;

}

double countScalarMult(const double \*vector1, const double \*vector2,

                       const size\_t sizeInput) {

  double res = 0;

  for (size\_t i = 0; i < sizeInput; ++i) {

    res += vector1[i] \* vector2[i];

  }

  return res;

}

double countVectorLength(const size\_t sizeInput,

                         const double \*vector) {

  return sqrt(countScalarMult(vector, vector, sizeInput));

}

void countVectorMultNumber(const double \*vector, double scalar,

                           double \*res, const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; ++i) {

    res[i] = scalar \* vector[i];

  }

}

void substructVectors(const double \*vector1, const double \*vector2,

                      double \*res, const size\_t sizeInput) {

  for (size\_t j = 0; j < sizeInput; ++j) {

    res[j] = vector1[j] - vector2[j];

  }

}

void copyVector(const double \*src, double \*dst,

                const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; i++) {

    dst[i] = src[i];

  }

}

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

  if (argc != 2) {

    std::cout << "Bad input! Enter matrix size" << std::endl;

  }

  srand(0);

  const size\_t sizeInput = atoi(argv[1]);

  const double precision = 1e-10;

  const double epsilon = 1e-10;

  const size\_t maxIterationCounts = 50000;

  size\_t iterationCounts = 0;

  double critCurrentEnd = 1;

  double prevCritEnd = 1;

  double prevPrevCritEnd = 1;

  bool isEndOfAlgo = false;

  struct timespec endt, startt;

  clock\_gettime(CLOCK\_MONOTONIC\_RAW, &startt);

  double \*Atmp = new double[sizeInput];

  double \*y = new double[sizeInput];

  double \*tauY = new double[sizeInput];

  double \*xCurr = new double[sizeInput];

  double \*xNext = new double[sizeInput];

  ///\* for testing data with RANDOM values ========= \*/

  double \*matrixA = fillRandomMatrix(sizeInput);

  // printMatrix(matrixA, sizeInput);

  double \*b = fillRandomVector(sizeInput);

  // std::cout << "vector b is: " << std::endl;

  // printVector(b, sizeInput);

  std::fill(xCurr, xCurr + sizeInput, 0);

  // std::cout << "vector X at start is: " << std::endl;

  // printVector(xCurr, sizeInput);

  ///\* for testing data with RANDOM values ========= \*/

  ///\* for testing when vector b uses sin ========= \*/

  // double \*matrixA = fillConstantMatrix(sizeInput);

  // // printMatrix(matrixA, sizeInput);

  // double \*b = new double[sizeInput];

  // fillVectorB(b, matrixA, sizeInput);

  // // std::cout << "vector b is" << std::endl;

  // // printVector(b, sizeInput);

  // std::fill(xCurr, xCurr + sizeInput, 0);

  // // std::cout << "vector X at start is: " << std::endl;

  // // printVector(xCurr, sizeInput);

  ///\* for testing when vector b uses sin ========= \*/

  double bNorm = countVectorLength(sizeInput, b);

  while (1) {

    multiplyMatrixOnVector(matrixA, xCurr, Atmp,

                            sizeInput);      // A \* x\_n

    substructVectors(Atmp, b, y, sizeInput); // y\_n = A \* x\_n - b

    double yNorm =

        countVectorLength(sizeInput, y); // || A \* x\_n - b ||

    std::fill(Atmp, Atmp + sizeInput, 0);

    multiplyMatrixOnVector(matrixA, y, Atmp, sizeInput); // A \* y\_n

    double numeratorTau =

        countScalarMult(y, Atmp, sizeInput); // (y\_n, A \* y\_n)

    double denominatorTau =

        countScalarMult(Atmp, Atmp, sizeInput); // (A \* y\_n, A \* y\_n)

    double tau = numeratorTau / denominatorTau;

    countVectorMultNumber(y, tau, tauY, sizeInput); // tau \* y

    substructVectors(xCurr, tauY, xNext,

                     sizeInput); // x\_n+1 = x\_n - tau \* y

    double critCurrentEnd = yNorm / bNorm;

    if (iterationCounts > maxIterationCounts) {

      std::cout << "Too many iterations. Change init values"

                << std::endl;

      delete[] xCurr;

      delete[] xNext;

      delete[] Atmp;

      delete[] y;

      delete[] tauY;

      delete[] matrixA;

      delete[] b;

      return 0;

    }

    if (critCurrentEnd < epsilon && critCurrentEnd < prevCritEnd &&

        critCurrentEnd < prevPrevCritEnd) {

      isEndOfAlgo = true;

      break;

    }

    copyVector(xNext, xCurr, sizeInput);

    prevPrevCritEnd = prevCritEnd;

    prevCritEnd = critCurrentEnd;

    iterationCounts++;

    // std::cout << "iteration " << iterationCounts

    //           << " ended ===" << std::endl;

  }

  clock\_gettime(CLOCK\_MONOTONIC\_RAW, &endt);

  if (isEndOfAlgo) {

    std::cout << "Iteration amount in total: " << iterationCounts

              << std::endl;

    std::cout << "Time taken: "

              << endt.tv\_sec - startt.tv\_sec +

                     precision \* (endt.tv\_nsec - startt.tv\_nsec)

              << " sec" << std::endl;

    // std::cout << "Solution (vector X is): " << std::endl;

    // printVector(xNext, sizeInput);

  } else {

    std::cout << "There are no solutions =( Change input \n";

  }

  delete[] xCurr;

  delete[] xNext;

  delete[] Atmp;

  delete[] y;

  delete[] tauY;

  delete[] matrixA;

  delete[] b;

  return 0;

}

# Приложение 2. Проверка корректности работы последовательной программы

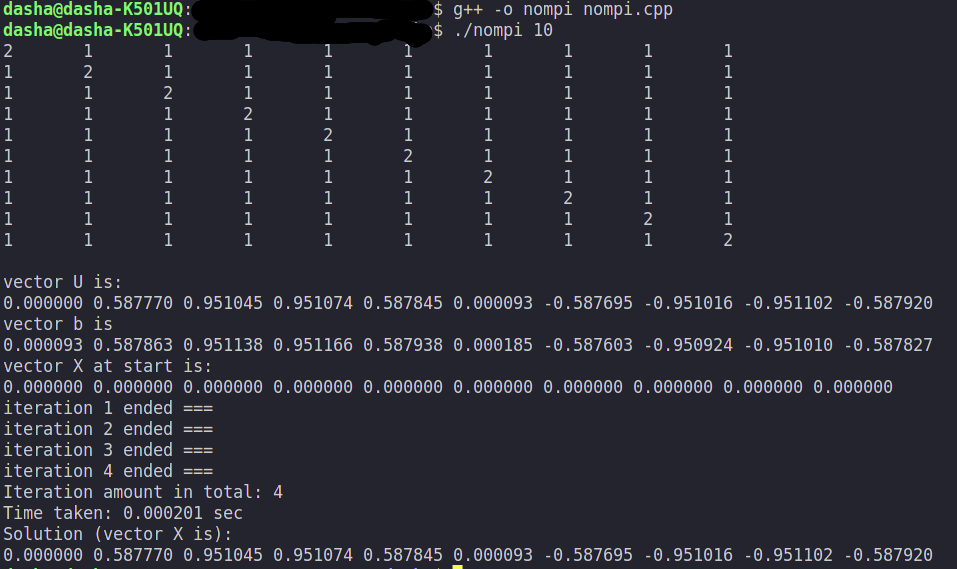


Рис. 1. Проверка корректности работы последовательной программы

# Приложение 3. Скрипт для запуска последовательной программы

1. Файл run\_nompi.sh:

#!/bin/sh

#PBS -l walltime=00:05:00

#PBS -l select=1:ncpus=1:mem=10000m

#PBS -N task\_nompi

cd $PBS\_O\_WORKDIR

icpc -O0 nompi.cpp -o nompi

./nompi 2000

# Приложение 4. Листинг параллельной программы с использованием функций MPI

#include <cmath>

#include <iostream>

#include <mpi.h>

void printMatrix(double \*matrix, const size\_t sizeInput) {

  std::cout << "Full matrix A is: " << std::endl;

  for (size\_t i = 0; i < sizeInput; ++i) {

    for (size\_t j = 0; j < sizeInput; ++j) {

      std::cout << matrix[j + i \* sizeInput] << "\t";

    }

    std::cout << std::endl;

  }

  std::cout << std::endl;

}

void printVector(const double \*vector, const size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; ++i) {

    std::cout << std::fixed << vector[i] << " ";

  }

  std::cout << std::endl;

}

void fillRandomMatrix(double \*fullMatrA, size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; i++) {

    for (size\_t j = 0; j < sizeInput; j++) {

      fullMatrA[i \* sizeInput + j] =

          (i == j) ? 9999 : rand() % 500 + 150.15;

    }

  }

}

void fillConstantMatrix(double \*fullMatrA, size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; i++) {

    for (size\_t j = 0; j < sizeInput; j++) {

      fullMatrA[i \* sizeInput + j] = (i == j) ? 2.0 : 1.0;

    }

  }

}

void fillRandomVector(double \*vector, size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; i++) {

    vector[i] = rand() % 500;

  }

}

void fillConstantVector(double \*vector, size\_t sizeInput) {

  for (size\_t i = 0; i < sizeInput; i++) {

    vector[i] = sizeInput + 1;

  }

}

void multiplyMatrixOnVector(const double \*matrix,

                             const double \*vector, double \*res,

                             size\_t matrixRowsNumber,

                             size\_t matrixColumnsNumber) {

  for (size\_t i = 0; i < matrixRowsNumber; i++) {

    res[i] = 0;

    for (size\_t j = 0; j < matrixColumnsNumber; j++) {

      res[i] += matrix[i \* matrixColumnsNumber + j] \* vector[j];

    }

  }

}

double \*fillVectorU(const int N) {

  double \*u = new double[N];

  for (int i = 0; i < N; i++) {

    u[i] = sin(2 \* M\_PI \* (i) / N);

  }

  return u;

}

void fillVectorB(double \*b, const double \*fullMatrixA,

                 size\_t sizeInput) {

  double \*u = fillVectorU((int)sizeInput);

  std::cout << "vector u is: " << std::endl;

  printVector(u, sizeInput);

  multiplyMatrixOnVector(fullMatrixA, u, b, sizeInput, sizeInput);

  delete[] u;

}

double countScalarMult(const double \*vector1, const double \*vector2,

                       const size\_t size) {

  double res = 0;

  for (size\_t i = 0; i < size; i++) {

    res += vector1[i] \* vector2[i];

  }

  return res;

}

double countVectorLength(const double \*vector, const size\_t size) {

  return sqrt(countScalarMult(vector, vector, size));

}

void countVectorMultNumber(const double \*vector, double scalar,

                           double \*res, const size\_t size) {

  for (size\_t i = 0; i < size; ++i) {

    res[i] = scalar \* vector[i];

  }

}

void subtractVectors(const double \*vector1, const double \*vector2,

                     double \*res, const size\_t firstVectorSize,

                     const int vector1ElementsOffset,

                     const int vector2ElementsOffset) {

  for (size\_t i = 0; i < firstVectorSize; i++) {

    res[i] = vector1[vector1ElementsOffset + i] -

             vector2[vector2ElementsOffset + i];

  }

}

void copyVectors(const double \*src, double \*dst, const size\_t size) {

  for (size\_t i = 0; i < size; i++) {

    dst[i] = src[i];

  }

}

int \*countElemsNumInEachProc(size\_t amountOfProcs, size\_t sizeInput) {

  int \*elemsNum = new int[amountOfProcs];

  int basicRowsCount = sizeInput / amountOfProcs;

  int restRowsCount = sizeInput % amountOfProcs;

  for (size\_t i = 0; i < amountOfProcs; i++) {

    elemsNum[i] = basicRowsCount \* sizeInput;

    if (restRowsCount > 0) {

      elemsNum[i] += sizeInput;

      --restRowsCount;

    }

  }

  return elemsNum;

}

int \*countRowsInEachProcess(const int \*elementsNumberArr,

                            int amountOfProcs, size\_t sizeInput) {

  int \*rowsNum = new int[amountOfProcs];

  for (size\_t i = 0; i < amountOfProcs; i++) {

    rowsNum[i] = elementsNumberArr[i] / sizeInput;

  }

  return rowsNum;

}

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

  int \*elementsOffsetArray = new int[amountOfProcs];

  int elementsOffset = 0;

  for (size\_t i = 0; i < amountOfProcs; i++) {

    elementsOffsetArray[i] = elementsOffset;

    elementsOffset += elemsNum[i];

  }

  return elementsOffsetArray;

}

int \*createRowsOffsetArr(const int \*elementsOffsetArray,

                         int amountOfProcs, size\_t sizeInput) {

  int \*rowsOffsetArray = new int[amountOfProcs];

  for (size\_t i = 0; i < amountOfProcs; i++) {

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

  }

  return rowsOffsetArray;

}

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

  if (argc != 2) {

    std::cout << "Bad input! Enter matrix size" << std::endl;

  }

  srand(0);

  const size\_t sizeInput = atoi(argv[1]);

  const double epsilon = 10e-10;

  double critCurrentEnd = 1;

  double prevCritEnd = 1;

  double prevPrevCritEnd = 1;

  bool isEndOfAlgo = false;

  size\_t iterationCounts = 0;

  const size\_t maxIterationCounts = 50000;

  MPI\_Init(&argc, &argv);

  double startTime = MPI\_Wtime();

  int amountOfProcs, rankOfCurrProc;

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

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

  double \*fullMatrA = new double[sizeInput \* sizeInput];

  double \*b = new double[sizeInput];

  double \*xCurr = new double[sizeInput];

  ///\* for testing when vector b uses sin ========= \*/

  // if (rankOfCurrProc == 0) {

  //   fillConstantMatrix(fullMatrA, sizeInput);

  //   printMatrix(fullMatrA, sizeInput);

  //   fillVectorB(b, fullMatrA, sizeInput);

  //   std::fill(xCurr, xCurr + sizeInput, 0);

  //   std::cout << "vector X at start is: " << std::endl;

  //   printVector(xCurr, sizeInput);

  // }

  ///\* for testing when vector b uses sin ========= \*/

  ///\* for testing data with RANDOM values ========= \*/

  if (rankOfCurrProc == 0) {

    fillRandomMatrix(fullMatrA, sizeInput);

    // printMatrix(fullMatrA, sizeInput);

    fillRandomVector(b, sizeInput);

    // std::cout << "vector b is: " << std::endl;

    // printVector(b, sizeInput);

    std::fill(xCurr, xCurr + sizeInput, 0);

    // std::cout << "vector X at start is: " << std::endl;

    // printVector(xCurr, sizeInput);

  }

  ///\* for testing data with RANDOM values ========= \*/

  double bNorm;

  if (rankOfCurrProc == 0) {

    bNorm = countVectorLength(b, sizeInput);

  }

  // sends a message from root to all group process, including itself

  MPI\_Bcast(b, sizeInput, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

  // amount of elems, handling each process

  int \*elemsNum = countElemsNumInEachProc(amountOfProcs, sizeInput);

  // amount of rows, which each process handles

  int \*rowsNum =

      countRowsInEachProcess(elemsNum, amountOfProcs, sizeInput);

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

  int \*elementsOffsetArray =

      createElemsOffsetArr(elemsNum, amountOfProcs);

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

  int \*rowsOffsetArray = createRowsOffsetArr(

      elementsOffsetArray, amountOfProcs, sizeInput);

  double \*partMatrA = new double[elemsNum[rankOfCurrProc]];

  /\*\*\* MPI\_Scatterv - рапределяет блоки данных по всем процессам

   \* @param sendbuf [in] - address of send buffer (significant only at

   \* root)

   \* @param sendcounts [in] - array (=group size)

   \* specifying the number of elements to send to each processor

   \* @param displs [in] - array (=group size).

   \* Entry i specifies the displacement (relative to sendbuf from

   \* which to take the outgoing data to process i)

   \* i-oe значение посылает данные в i-й блок

   \* @param sendtype - type of sending data

   \* @param recvbuf [out] - address of reciever buffer's starting

   \* @param recvcounts - amount of elems that we recieve

   \* @param recvtype - type of receiving data

   \* @param root - number of proсess-reciever

   \* @param Comm - communicator

   \*/

  MPI\_Scatterv(fullMatrA, elemsNum, elementsOffsetArray, MPI\_DOUBLE,

               partMatrA, elemsNum[rankOfCurrProc], MPI\_DOUBLE, 0,

               MPI\_COMM\_WORLD);

  double \*partMultResultVector\_Ax =

      new double[rowsNum[rankOfCurrProc]];

  double \*partMultResultVector\_Ay =

      new double[rowsNum[rankOfCurrProc]];

  double \*partVectorY = new double[rowsNum[rankOfCurrProc]];

  double \*partNextX = new double[rowsNum[rankOfCurrProc]];

  double \*partMultVectorByScalar\_TauY =

      new double[rowsNum[rankOfCurrProc]];

  double \*vectorY = new double[sizeInput];

  double \*xNext = new double[sizeInput];

  // int widthMatrixPart = sizeInput;

  // int heightMatrixPart = rowsNum[rankOfCurrProc];

  while (1) {

    MPI\_Bcast(xCurr, sizeInput, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    multiplyMatrixOnVector(partMatrA, xCurr, partMultResultVector\_Ax,

                            rowsNum[rankOfCurrProc], sizeInput);

    subtractVectors(partMultResultVector\_Ax, b, partVectorY,

                    rowsNum[rankOfCurrProc], 0,

                    rowsOffsetArray[rankOfCurrProc]);

    /\*\*\* MPI\_Allgatherv - собирает блоки с разным числом элементов от

   каждого процесса

   \* @param sendbuf [in] - starting address of send buffer

   \* @param sendcounts [in] - amount of elems in send buffer

   \* @param sendtype - data type of send buffer elements

   \* @param recvbuf [out] - address of receive buffer

   \* @param recvcounts - array (=group size) containing the number of

   elements that are to be received from each process

   \* @param displs - array (=group size). Entry i

   specifies the displacement (relative to recvbuf) at which to place

   the incoming data from process i

   \* @param recvtype - data type of recieve buffer elements

   \* @param Comm - communicator

   \*/

    // y\_n = A \* x\_n - b

    MPI\_Allgatherv(partVectorY, rowsNum[rankOfCurrProc], MPI\_DOUBLE,

                   vectorY, rowsNum, rowsOffsetArray, MPI\_DOUBLE,

                   MPI\_COMM\_WORLD);

    // A \* y\_n

    multiplyMatrixOnVector(partMatrA, vectorY,

                            partMultResultVector\_Ay,

                            rowsNum[rankOfCurrProc], sizeInput);

    // || A \* x\_n - b ||

    double yNorm = 0;

    if (rankOfCurrProc == 0) {

      yNorm = countVectorLength(vectorY, sizeInput);

    }

    // start counting tau

    double numeratorTau, denominatorTau;

    double partScalarProduct\_YAy =

        countScalarMult(partVectorY, partMultResultVector\_Ay,

                        rowsNum[rankOfCurrProc]);

    // (y\_n, A \* y\_n)

    MPI\_Allreduce(&partScalarProduct\_YAy, &numeratorTau, 1,

                  MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

    double partScalarProduct\_AyAy = countScalarMult(

        partMultResultVector\_Ay, partMultResultVector\_Ay,

        rowsNum[rankOfCurrProc]);

    /\*\* MPI\_Allreduce - combines values from all processes and

     \* distributes the result back to all processes

     \* @param sendbuf - starting address of send buffer

     \* @param recvbuf - starting address of receive buffer

     \* @param count - number of elements in send buffer

     \* @param datatype - data type of elements of send buffer

     \* @param op - operation

     \* @param comm - communicator

     \*/

    // (A \* y\_n, A \* y\_n)

    MPI\_Allreduce(&partScalarProduct\_AyAy, &denominatorTau, 1,

                  MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

    double tau = numeratorTau / denominatorTau;

    countVectorMultNumber(partVectorY, tau,

                          partMultVectorByScalar\_TauY,

                          rowsNum[rankOfCurrProc]);

    subtractVectors(xCurr, partMultVectorByScalar\_TauY, partNextX,

                    rowsNum[rankOfCurrProc],

                    rowsOffsetArray[rankOfCurrProc], 0);

    // x\_n+1 = x\_n - tau \* y

    MPI\_Allgatherv(partNextX, rowsNum[rankOfCurrProc], MPI\_DOUBLE,

                   xNext, rowsNum, rowsOffsetArray, MPI\_DOUBLE,

                   MPI\_COMM\_WORLD);

    critCurrentEnd = yNorm / bNorm;

    if (critCurrentEnd < epsilon && critCurrentEnd < prevCritEnd &&

        critCurrentEnd < prevPrevCritEnd) {

      isEndOfAlgo = true;

    }

    if (iterationCounts > maxIterationCounts) {

      isEndOfAlgo = true;

    }

    MPI\_Bcast(&isEndOfAlgo, 1, MPI\_C\_BOOL, 0, MPI\_COMM\_WORLD);

    if (isEndOfAlgo) {

      break;

    }

    copyVectors(xNext, xCurr, sizeInput);

    prevPrevCritEnd = prevCritEnd;

    prevCritEnd = critCurrentEnd;

    iterationCounts++;

    // if (rankOfCurrProc == 0) {

    //   std::cout << "iteration " << iterationCounts

    //             << " ended ====" << std::endl;

    // }

  }

  double endTime = MPI\_Wtime();

  if (iterationCounts < maxIterationCounts) {

    if (rankOfCurrProc == 0) {

      std::cout << "Iteration amount in total: " << iterationCounts

                << std::endl;

      std::cout << "Time taken: " << endTime - startTime << " sec"

                << std::endl;

      // std::cout << "Solution (vector X is): " << std::endl;

      // printVector(xCurr, sizeInput);

    }

  }

  else {

    if (rankOfCurrProc == 0) {

      std::cout << "There are no solutions =( Change input \n";

    }

  }

  delete[] fullMatrA;

  delete[] partMatrA;

  delete[] partMultVectorByScalar\_TauY;

  delete[] partVectorY;

  delete[] partNextX;

  delete[] partMultResultVector\_Ax;

  delete[] partMultResultVector\_Ay;

  delete[] b;

  delete[] xCurr;

  delete[] xNext;

  delete[] vectorY;

  delete[] rowsOffsetArray;

  delete[] elementsOffsetArray;

  delete[] rowsNum;

  delete[] elemsNum;

  MPI\_Finalize();

  return 0;

}

# Приложение 4. Проверка корректности работы параллельной программы

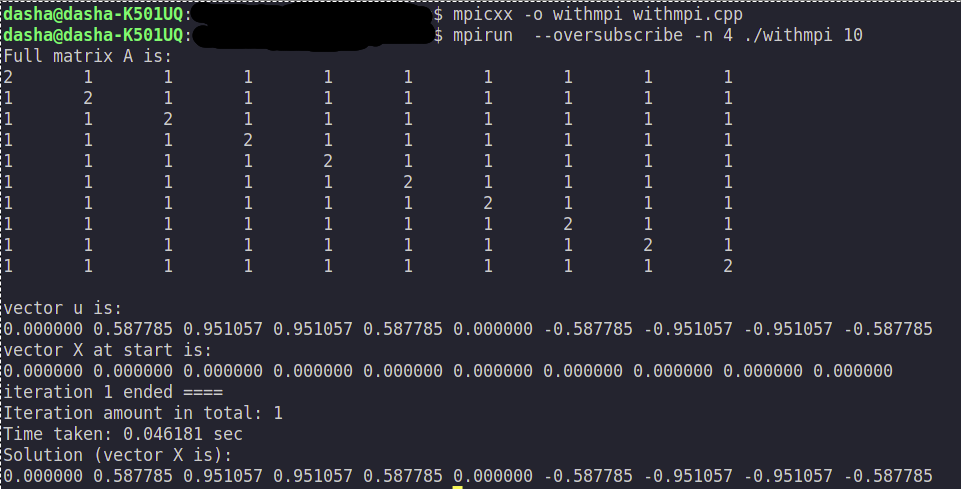


Рис. 1. Результат на 4 процессах

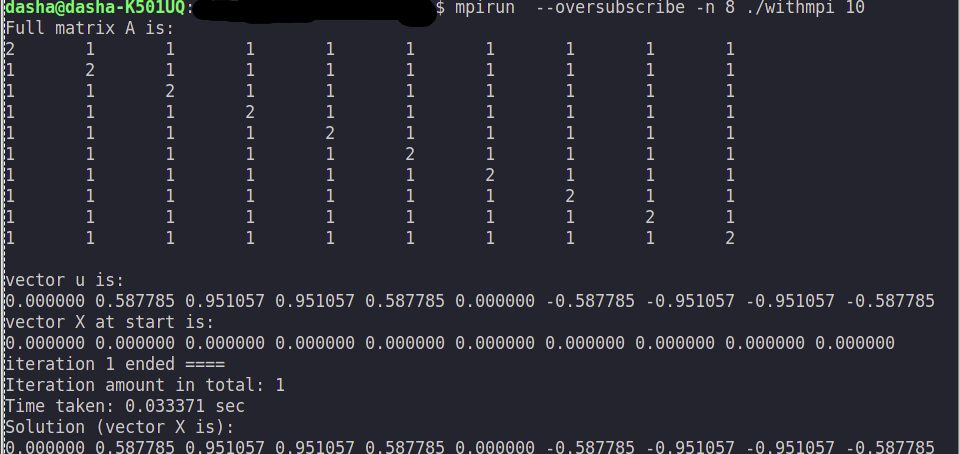


Рис. 2. Результат на 8 процессах

# Приложение 5. Скрипт для работы параллельной программы

Файл run\_withmpi.sh

#!/bin/bash

#PBS -l walltime=00:05:00

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

#PBS -m n

#PBS -N task\_withmpi

cd $PBS\_O\_WORKDIR

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

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

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

mpirun -hostfile $PBS\_NODEFILE -np $MPI\_NP ./withmpi 2000

\*Примечание.

Для измерения времени на разных процессах: 2, 4, 8, 16, 24 изменяется количество узлов и процессов. Т.е. строка с выбором узлов и процессов будет выглядеть так:

Для 2х процессов:

#PBS -l select=2:ncpus=1:mpiprocs=1:mem=10000m

Для 4х процессов:

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

Для 8и процессов:

#PBS -l select=2:ncpus=4:mpiprocs=4:mem=10000m

Для 16и процессов:

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

Для 24х процессов:

#PBS -l select=2:ncpus=12:mpiprocs=12:mem=10000m

# Приложение 6. Результаты работы последовательной и параллельной на 2, 4, 8, 16, 24 процессах

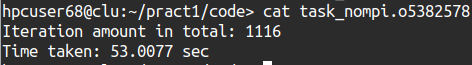


Рис. 1. Время работы последовательной программы

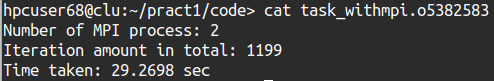


Рис. 2. Время работы параллельной программы на 2х процессах

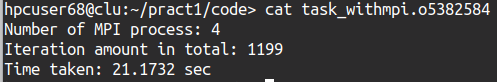


Рис. 2. Время работы параллельной программы на 4х процессах

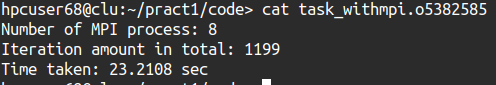


Рис. 4. Время работы параллельной программы на 8и процессах

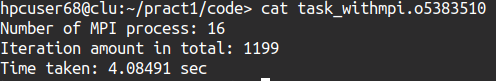


Рис. 5. Время работы параллельной программы на 16и процессах

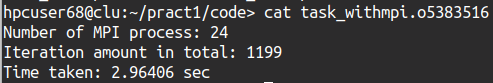


Рис. 6. Время работы параллельной программы на 24и процессах

# Приложение 7. Графики времени, ускорения и эффективности

1. Рис. 1. График зависимости от времени
2. Рис. 2. График ускорения

Рис. 3. График эффективности

# Приложение 8. Скрипт для профилирования параллельной программы

Файл run\_alanysis.sh

#!/bin/bash

#PBS -l walltime=00:02:00

#PBS -l select=2:ncpus=8:mpiprocs=8:mem=10000m,place=scatter:exclhost

#PBS -m n

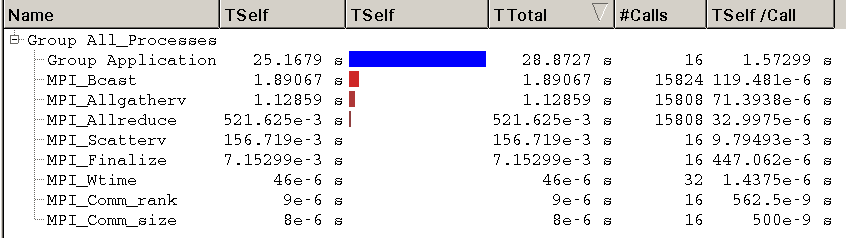
cd $PBS\_O\_WORKDIR

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

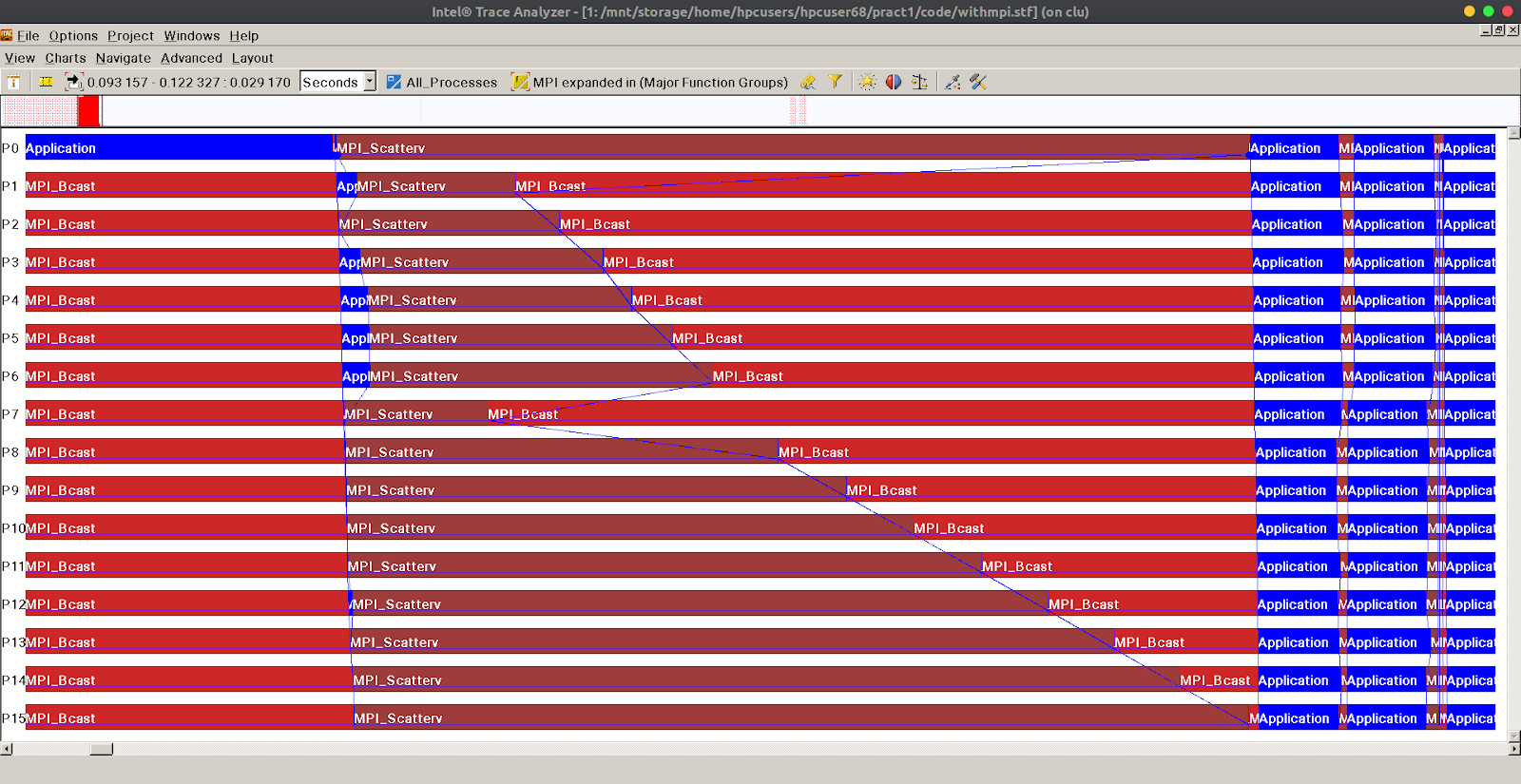
echo "Number of MPI process: $MPI\_NP"

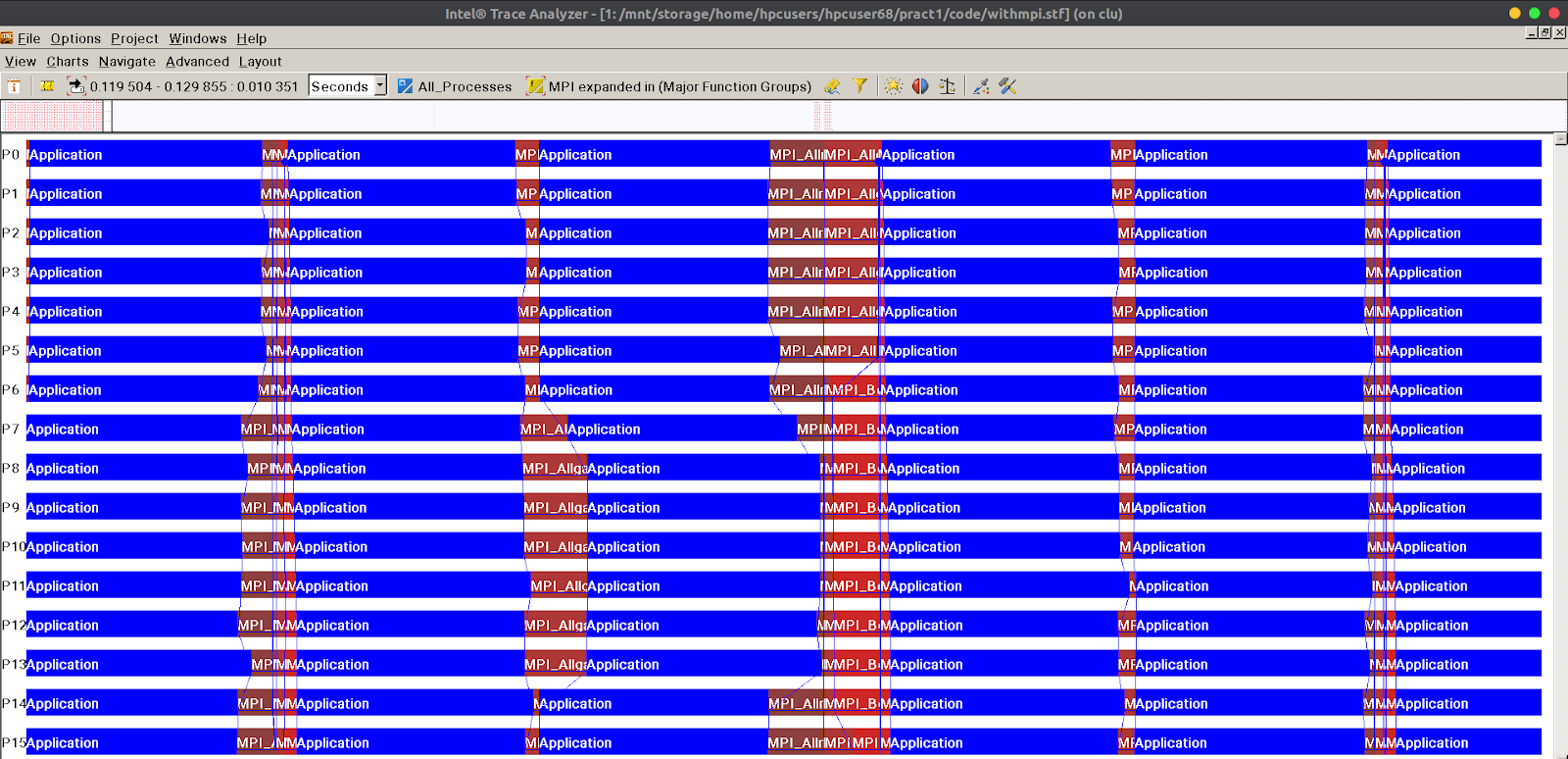
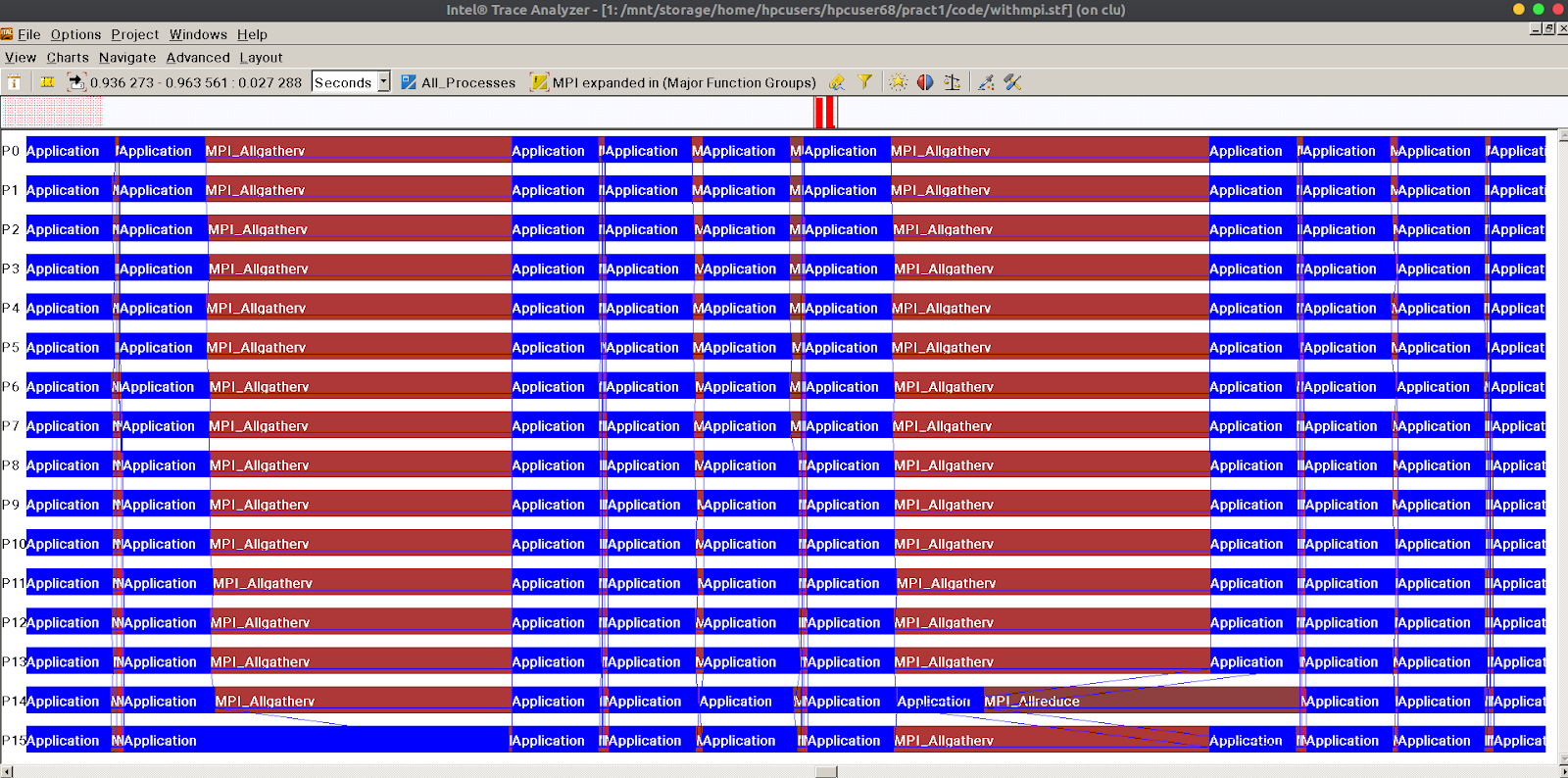
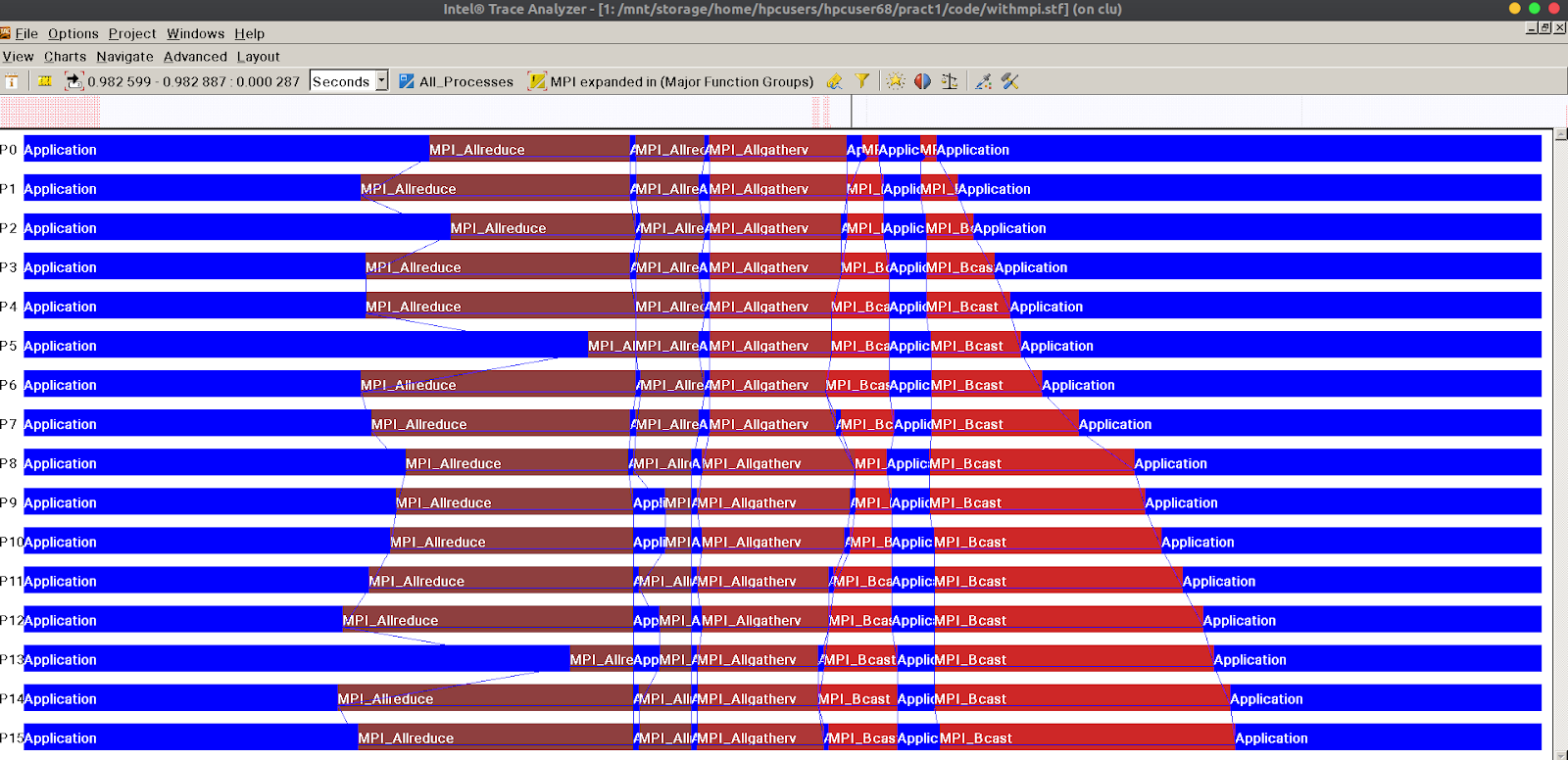
mpirun -trace -machinefile $PBS\_NODEFILE -np $MPI\_NP -perhost 2 ./withmpi 2000

# Приложение 9. Скрины из traceanalyzer для 16 процессов



1. Рис. 1. Расположение в порядке убывания по времени используемых функций



1. Рис. 2. Начало программы и распределение данных по всем процессам
2. 
3. Рис. 3. Подсчет Y\_n
4.   
   Рис. 4. Подсчет X\_n+1
5.   
   Рис. 5. Завершение программы