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

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).

3. Замерить время работы последовательной программы и параллельных на 2, 4, 8, 16, 24 процессах.

4. 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.

5. 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.

6. 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, $u = \sin\frac{2\pi i}{N}$.

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

   g++ -o nompi nompi.cpp

   ./nompi 10
4. Далее началась работа на кластере.

   Размер матрицы, при котором время программы составляет не менее 30сек: 2000.
5. Скрипт (run_nompi.sh) для запуска программы nompi.cpp смотреть в Приложении 3.
6. Итак, для последовательной программы время работы составляет 51,00640 секунд, размер матрицы равен 2000.
7. 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.
8. Результаты проверки (на нескольких процессах) корректности см. Приложение 5. Команда для компиляции (не на кластере!):

   mpicxx -oversubscribe -o withmpi withmpi.cpp

   mpirun -n [number of processes]./withmpi
9. Затем началась работа на кластере. Был написан скрипт run_withmpi.sh (см. Приложение 6) для параллельной программы.
10. Было измерено время работы программы на 2, 4, 8, 16, 24 процессах. Скрипт представлен в Приложении 5.

11. Результаты измерения последовательной и параллельной программ на 2, 4, 8, 16, 24 процессах представлен в Приложении 6.
12. **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).
13. Полученные результаты измерения времени представлены в Таблице 1.

Таблица1. Результаты измерений времени

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

14. На основании полученных данных рассчитаем эффективность и ускорение. Графики результатов представлены в Приложении 7.
15. Было проведено профилирование программы на 16 процессах. Скрипт для запуска профилирования смотреть в Приложении 8.
1. 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 .
16. 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

```
dasha@dasha-K501UQ:[redacted]$ g++ -o nompi nompi.cpp
dasha@dasha-K501UQ:[redacted]$ ./nompi 10
2       1       1       1       1       1       1       1       1       1
1       2       1       1       1       1       1       1       1       1
1       1       2       1       1       1       1       1       1       1
1       1       1       2       1       1       1       1       1       1
1       1       1       1       2       1       1       1       1       1
1       1       1       1       1       2       1       1       1       1
1       1       1       1       1       1       2       1       1       1
1       1       1       1       1       1       1       2       1       1
1       1       1       1       1       1       1       1       2       1
1       1       1       1       1       1       1       1       1       2

vector U is:
0.000000 0.587770 0.951045 0.951074 0.587845 0.000093 -0.587695 -0.951016 -0.951102 -0.587920
vector b is
0.000093 0.587863 0.951138 0.951166 0.587938 0.000185 -0.587603 -0.950924 -0.951010 -0.587827
vector X at start is:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
iteration 1 ended ===
iteration 2 ended ===
iteration 3 ended ===
iteration 4 ended ===
Iteration amount in total: 4
Time taken: 0.000201 sec
Solution (vector X is):
0.000000 0.587770 0.951045 0.951074 0.587845 0.000093 -0.587695 -0.951016 -0.951102 -0.587920
```

Рис. 1. Проверка корректности работы последовательной программы

Приложение 3. Скрипт для запуска последовательной программы

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-ое 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 process-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

```
dasha@dasha-K501UQ:[redacted]$ mpicxx -o withmpi withmpi.cpp
dasha@dasha-K501UQ:[redacted]$ mpirun  --oversubscribe -n 4 ./withmpi 10
Full matrix A is:
2       1       1       1       1       1       1       1       1       1
1       2       1       1       1       1       1       1       1       1
1       1       2       1       1       1       1       1       1       1
1       1       1       2       1       1       1       1       1       1
1       1       1       1       2       1       1       1       1       1
1       1       1       1       1       2       1       1       1       1
1       1       1       1       1       1       2       1       1       1
1       1       1       1       1       1       1       2       1       1
1       1       1       1       1       1       1       1       2       1
1       1       1       1       1       1       1       1       1       2

vector u is:
0.000000 0.587785 0.951057 0.951057 0.587785 0.000000 -0.587785 -0.951057 -0.951057 -0.587785
vector X at start is:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
iteration 1 ended ====
Iteration amount in total: 1
Time taken: 0.046181 sec
Solution (vector X is):
0.000000 0.587785 0.951057 0.951057 0.587785 0.000000 -0.587785 -0.951057 -0.951057 -0.587785
```

Рис. 1. Результат на 4 процессах

```
dasha@dasha-K501UQ:[redacted]$ mpirun  --oversubscribe -n 8 ./withmpi 10
Full matrix A is:
2       1       1       1       1       1       1       1       1       1
1       2       1       1       1       1       1       1       1       1
1       1       2       1       1       1       1       1       1       1
1       1       1       2       1       1       1       1       1       1
1       1       1       1       2       1       1       1       1       1
1       1       1       1       1       2       1       1       1       1
1       1       1       1       1       1       2       1       1       1
1       1       1       1       1       1       1       2       1       1
1       1       1       1       1       1       1       1       2       1
1       1       1       1       1       1       1       1       1       2

vector u is:
0.000000 0.587785 0.951057 0.951057 0.587785 0.000000 -0.587785 -0.951057 -0.951057 -0.587785
vector X at start is:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
iteration 1 ended ====
Iteration amount in total: 1
Time taken: 0.033371 sec
Solution (vector X is):
0.000000 0.587785 0.951057 0.951057 0.587785 0.000000 -0.587785 -0.951057 -0.951057 -0.587785
```

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

```
hpcuser68@clu:~/pract1/code> cat task_nompi.o5382578
Iteration amount in total: 1116
Time taken: 53.0077 sec
```

Рис. 1. Время работы последовательной программы

```
hpcuser68@clu:~/pract1/code> cat task_withmpi.o5382583
Number of MPI process: 2
Iteration amount in total: 1199
Time taken: 29.2698 sec
```

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

```
hpcuser68@clu:~/pract1/code> cat task_withmpi.o5382584
Number of MPI process: 4
Iteration amount in total: 1199
Time taken: 21.1732 sec
```

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

```
hpcuser68@clu:~/pract1/code> cat task_withmpi.o5382585
Number of MPI process: 8
Iteration amount in total: 1199
Time taken: 23.2108 sec
```

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

```
hpcuser68@clu:~/pract1/code> cat task_withmpi.o5383510
Number of MPI process: 16
Iteration amount in total: 1199
Time taken: 4.08491 sec
```

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

```
hpcuser68@clu:~/pract1/code> cat task_withmpi.o5383516
Number of MPI process: 24
Iteration amount in total: 1199
Time taken: 2.96406 sec
```

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

Приложение 7. Графики времени, ускорения и эффективности

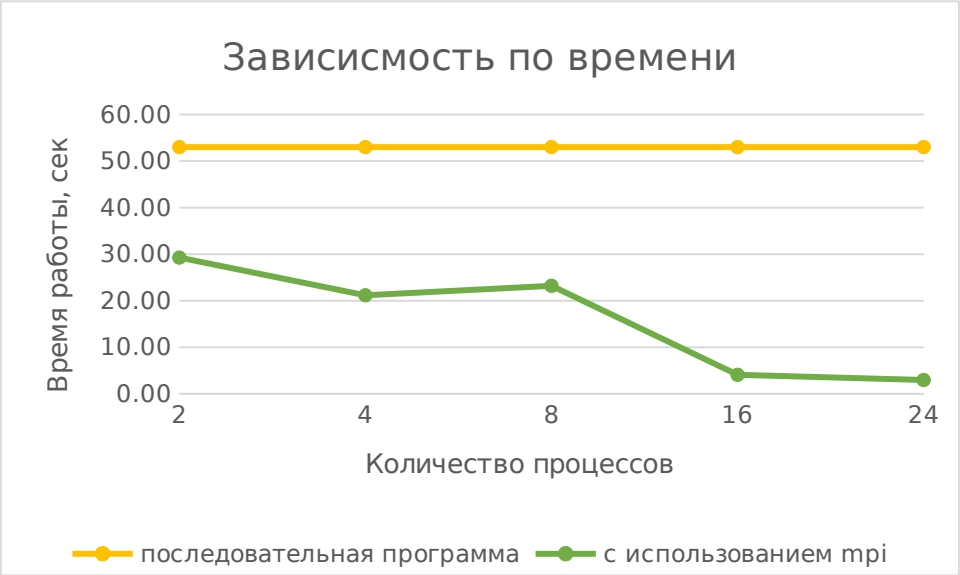

Рис. 1. График зависимости от времени

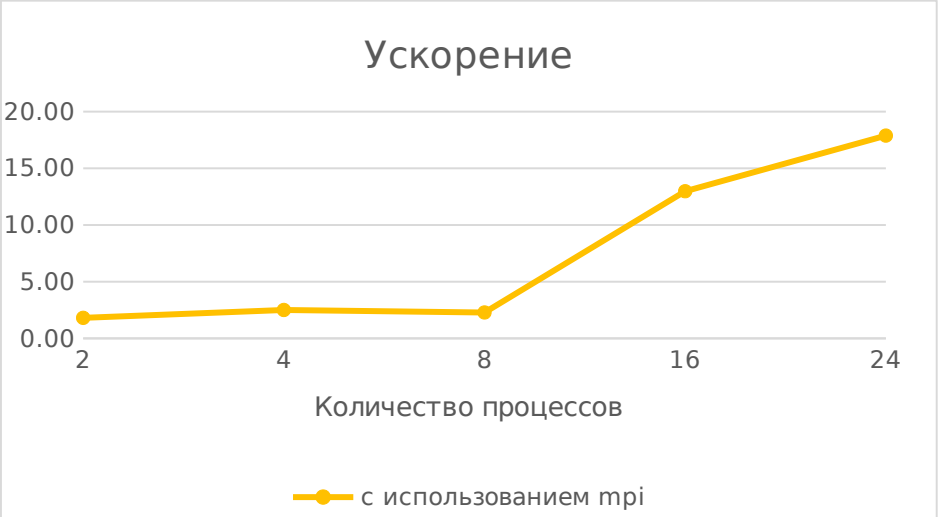

Рис. 2. График ускорения

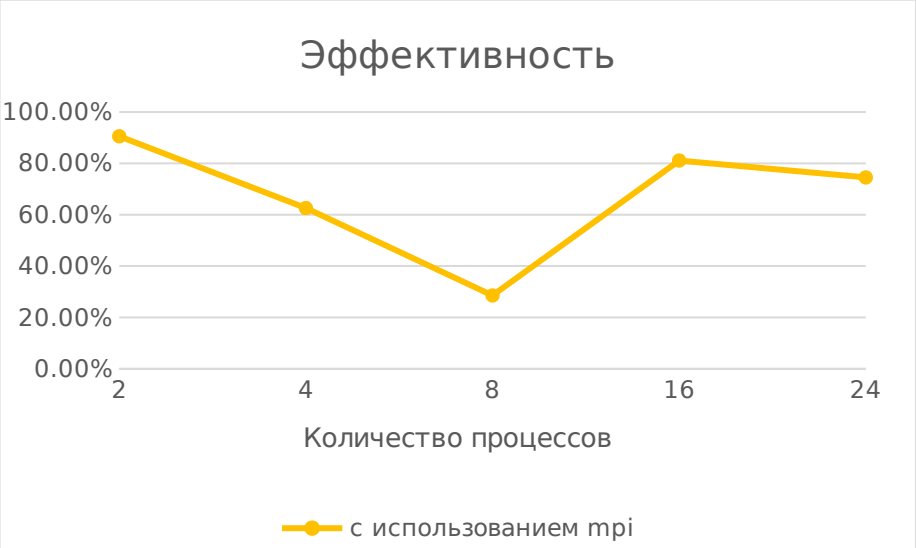

Рис. 3. График эффективности

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

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	25.1679 s		28.8727 s	16	1.57299 s
MPI_Bcast	1.89067 s		1.89067 s	15824	119.481e-6 s
MPI_Allgatherv	1.12859 s		1.12859 s	15808	71.3938e-6 s
MPI_Allreduce	521.625e-3 s		521.625e-3 s	15808	32.9975e-6 s
MPI_Scatterv	156.719e-3 s		156.719e-3 s	16	9.79493e-3 s
MPI_Finalize	7.15299e-3 s		7.15299e-3 s	16	447.062e-6 s
MPI_Wtime	46e-6 s		46e-6 s	32	1.4375e-6 s
MPI_Comm_rank	9e-6 s		9e-6 s	16	562.5e-9 s
MPI_Comm_size	8e-6 s		8e-6 s	16	500e-9 s

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

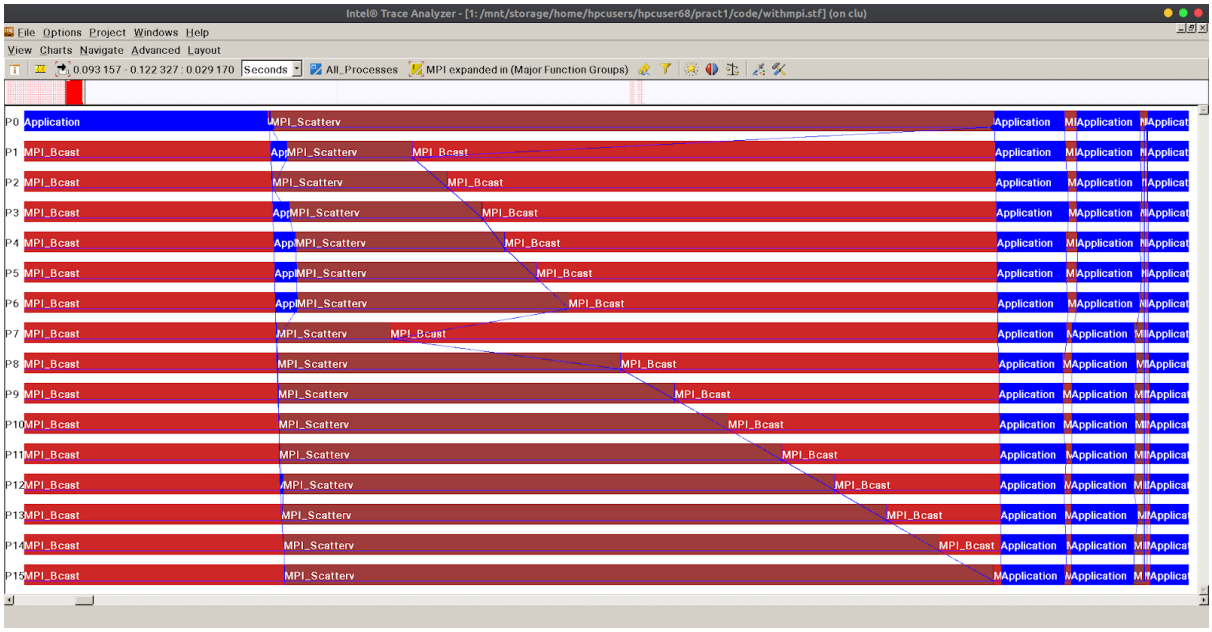

Рис. 2. Начало программы и распределение данных по всем процессам

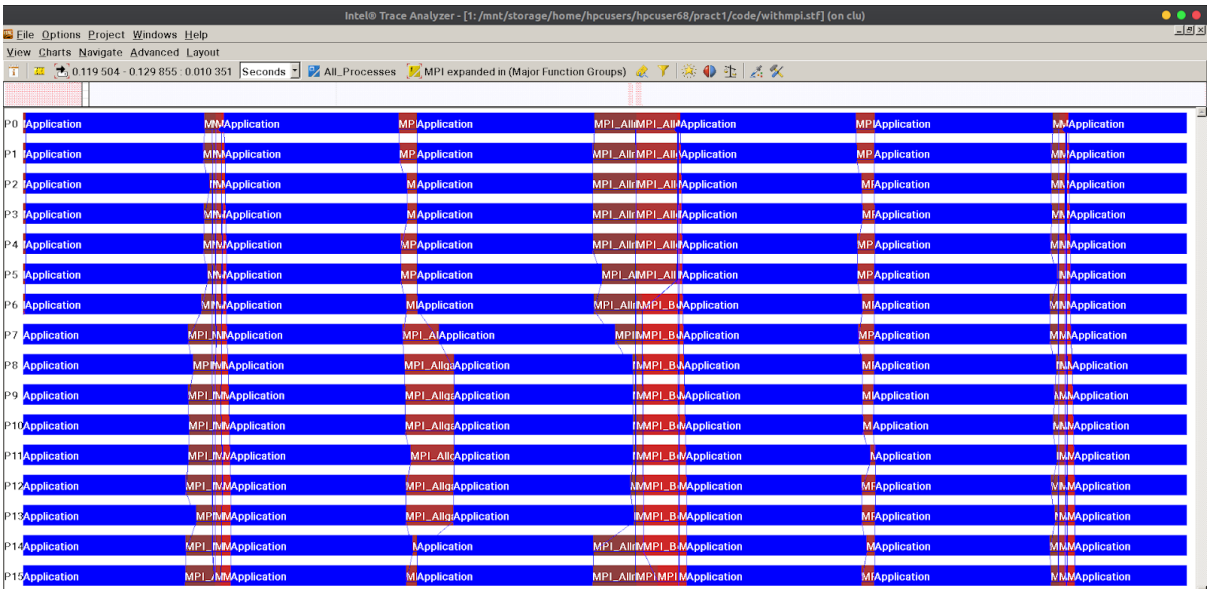

Рис. 3. Подсчет Y_n

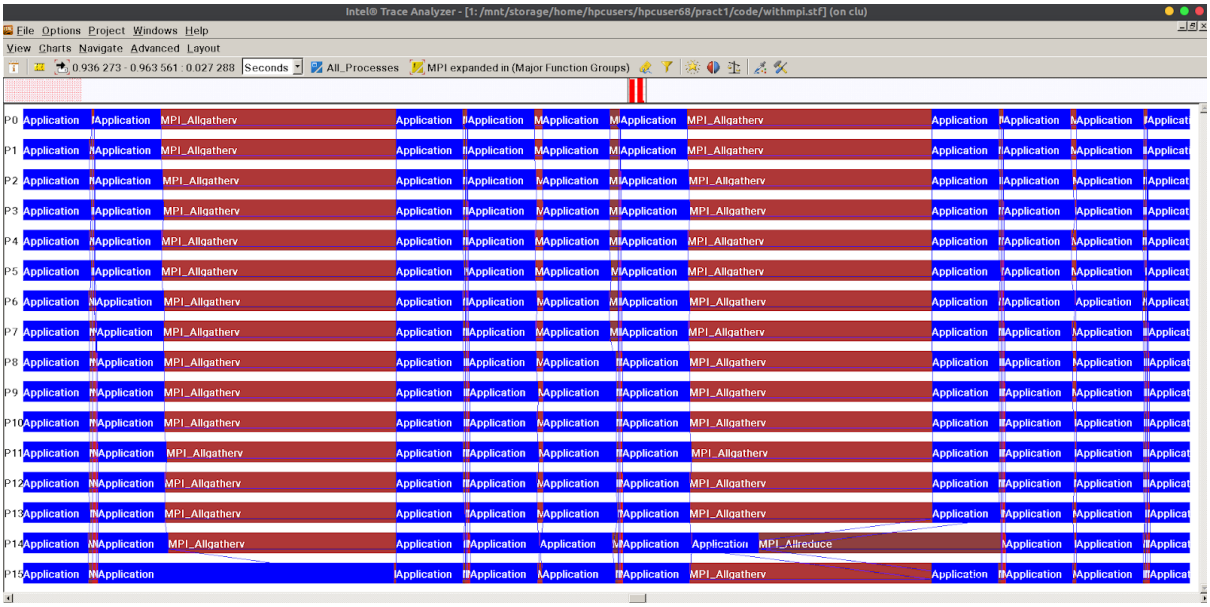

Рис. 4. Подсчет X_n+1

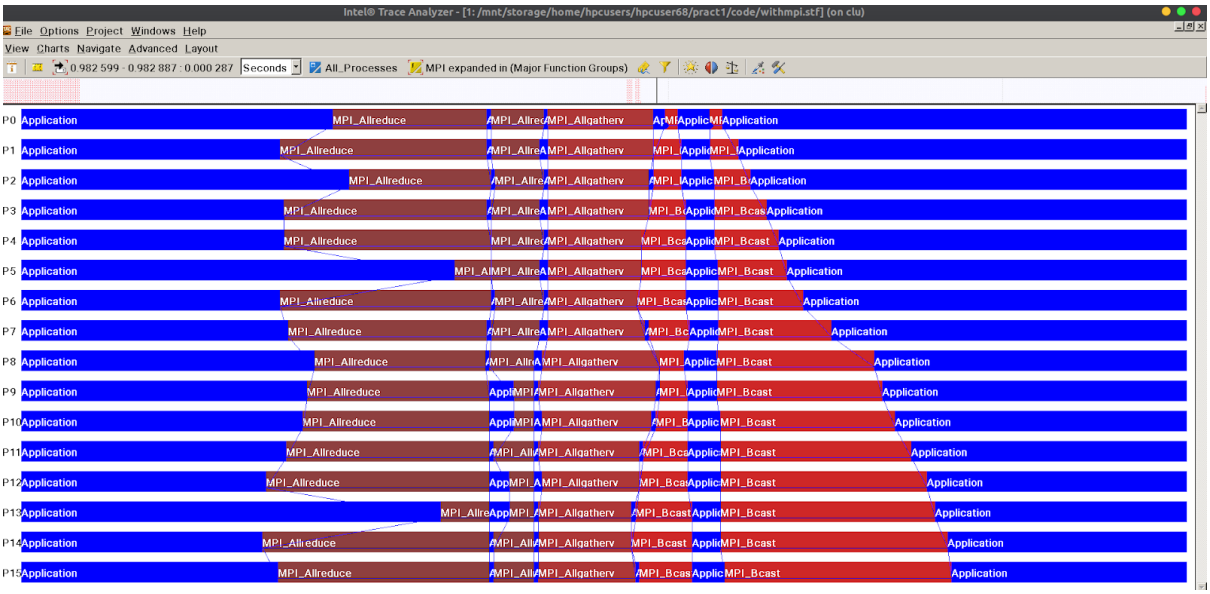

Рис. 5. Завершение программы