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 ЛАБОРАТОРНОЙ РАБОТЫ

««Параллельная реализация решения системы линейных алгебраических уравнений с помощью MPI»

студента Бородина Артёма Максимовича 2 курса, 19205 группы

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
к.т.н, доцент
А.Ю. Власенко

Новосибирск 2021

СОДЕРЖАНИЕ

ЦЕЛЬ

1. Решение СЛАУ, используя MPI.

ЗАДАНИЕ

1. Написать программу на языке C или C++, которая реализует 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. Программу распараллелить с помощью MPI с разрезанием матрицы A по строкам на близкие по размеру, возможно не одинаковые, части. Соседние строки матрицы должны располагаться в одном или в соседних MPI-процессах. Реализовать два варианта программы:
   - Вариант 1: векторы x и b дублируются в каждом MPI-процессе,
   - Вариант 2: векторы x и b разрезаются между MPI-процессами аналогично матрице A.

3. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: 1,2, 4, 8, 16. Построить графики зависимости времени работы программы, 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.

4. Выполнить профилирование двух вариантов программы с помощью MPE при использовании 16-и ядер

5. На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программы.

ОПИСАНИЕ РАБОТЫ

1. Была написана последовательная программа для решения СЛАУ (Приложение 1).

2. Были написаны варианты программы (Приложение 2, Приложение 3), использующие MPI для решения СЛАУ методом сопряженных градиентов. Были проведены проверки на корректность работы программ при различных размерах матрицы и количестве процессов.

3. Были выбраны размер матрицы и точность, при которых время работы программы на одном потоке занимало около 30 секунд.

4. Было проведено профилирование обоих вариантов программы (Приложение 4, Приложение 5). По результатам профилирования можно увидеть, что работа с MPI занимает до трети времени работы программы. Некоторые процессы ждут, пока другие считают. Так происходит из-за того, что функции, использованные в программе, блокирующие и заставляют процессы ждать окончания работы самого медленного процесса.

5. Были проведены замеры времени работы программы на разном количестве процессов и узлов. По полученным данным были сделаны графики (Приложение 6).

6. При анализе полученных данных можно заметить постепенное уменьшение эффективности (примерно на 1% на каждые 2 процесса) с ростом количества процессов. Это может быть связано с тем, что увеличивается время, затраченное на коммуникацию между процессами.

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы были получены знания о разделении программы на процессы для более эффективного использования доступных ресурсов.

Приложение 1. *Код последовательной программы*

```
#include <cmath>
#include <chrono>
#include <iostream>
#include <fstream>

using namespace std::chrono;

void matVecMul(const double *mat, const double *vec, int N, double *newVec) {
  for (int i = 0; i < N; i++) {
    newVec[i] = 0;
    for (int j = 0; j < N; j++) {
      newVec[i] += mat[i * N + j] * vec[j];
    }
  }
}

void mulByConst(const double *vec, double c, int size, double *newVec) {
  for (int i = 0; i < size; i++) {
    newVec[i] = vec[i] * c;
  }
}

void subVec(const double *vec1, const double *vec2, int size, double *newVec) {
  for (int i = 0; i < size; i++) {
    newVec[i] = vec1[i] - vec2[i];
  }
}

void sumVec(const double *vec1, const double *vec2, int size, double *newVec) {
  for (int i = 0; i < size; i++) {
    newVec[i] = vec1[i] + vec2[i];
  }
}

double dotProduct(const double *vec1, const double *vec2, int size) {
  double sum = 0;
  for (int i = 0; i < size; ++i) {
    sum += vec1[i] * vec2[i];
  }
  return sum;
}

void printMat(double *mat, int rows, int columns, std::ofstream &stream) {
  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < columns; ++j) {
      stream << mat[i * columns + j] << " ";
    }
    stream << std::endl;
  }
}

double *solveSLAE(const double *A, double *b, int N) {
  auto *solution = new double[N]; //xn+1
  std::fill(solution, solution + N, 0);
  auto *prevSolution = new double[N]; // xn
  std::fill(prevSolution, prevSolution + N, 0);

  auto *Atmp = new double[N];
  auto *r = new double[N];
  auto *z = new double[N];
  auto *rNext = new double[N];
  auto *zNext = new double[N];
```

```
auto *alphaZ = new double[N];
auto *betaZ = new double[N];

double alpha;
double beta;

const double EPSILON = 1e-009;

double normb = sqrt(dotProduct(b, b, N));
double res = 1;
double prevRes = 1;
bool diverge = false;
int divergeCount = 0;
int rightAnswerRepeat = 0;
int iterCount = 0;
while (res > EPSILON && rightAnswerRepeat < 5) {
  if (res < EPSILON) {
    ++rightAnswerRepeat;
  } else {
    rightAnswerRepeat = 0;
  }

  /// rn = b - A * xn
  matVecMul(A, prevSolution, N, Atmp);
  subVec(b, Atmp, N, r);
  /// zn = rn
  for (int i = 0; i < N; ++i) {
    z[i] = r[i];
  }
  /// alpha
  matVecMul(A, z, N, Atmp);
  alpha = dotProduct(r, r, N) / dotProduct(Atmp, z, N);
  /// xn+1 = xn + alpha * zn
  mulByConst(z, alpha, N, alphaZ);
  sumVec(prevSolution, alphaZ, N, solution);
  /// rn+1 = rn - alpha * A * zn
  matVecMul(A, alphaZ, N, Atmp);
  subVec(r, Atmp, N, rNext);
  /// beta
  beta = dotProduct(rNext, rNext, N) / dotProduct(r, r, N);
  /// zn+1 = rn+1 + beta * zn
  mulByConst(z, beta, N, betaZ);
  sumVec(rNext, betaZ, N, zNext);

  res = sqrt(dotProduct(r, r, N)) / normb;
  if (prevRes < res || res == INFINITY || res == NAN) {
    ++divergeCount;
    if (divergeCount > 10 || res == INFINITY || res == NAN) {
      diverge = true;
      break;
    }
  } else {
    divergeCount = 0;
  }
  prevRes = res;
  for (long i = 0; i < N; i++) {
    prevSolution[i] = solution[i];
    r[i] = rNext[i];
    z[i] = zNext[i];
  }
  ++iterCount;
}
delete[](prevSolution);
```

```
    delete[](Atmp);
    delete[](r);
    delete[](z);
    delete[](rNext);
    delete[](zNext);
    delete[](alphaZ);
    delete[](betaZ);

    std::cout << "iterCount: " << iterCount << std::endl;

    if (diverge) {
        delete[](solution);
        return nullptr;
    } else {
        return solution;
    }
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        std::cout << "Program needs 2 arguments: size, filename" << std::endl;
        return 0;
    }
    int N = atoi(argv[1]);
    std::string name(argv[2]);

    const std::string &fileName = name;
    std::ofstream fileStream(fileName);
    if (!fileStream) {
        std::cout << "error with output file" << std::endl;
        return 0;
    }

    fileStream << "Matrix size: " << N << std::endl;

    auto *b = new double[N];
    auto *u = new double[N];
    auto *A = new double[N * N];

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (i == j) {
                A[i * N + j] = 2;
            } else {
                A[i * N + j] = 1;
            }
        }
        u[i] = sin(2 * M_PI * i / double(N));
    }
    matVecMul(A, u, N, b);

    auto startTime = system_clock::now();
    double *solution = solveSLAE(A, b, N);
    auto endTime = system_clock::now();
    auto duration = duration_cast<nanoseconds>(endTime - startTime);

    if (solution != nullptr) {
        fileStream << "Answer:" << std::endl;
        printMat(u, 1, N, fileStream);
        fileStream << "SLAE solution:" << std::endl;
        printMat(solution, 1, N, fileStream);
        fileStream << "Time: " << duration.count() / double(1000000000) << "sec" << std::endl;
    } else {
```

```
    fileStream << "Does not converge" << std::endl;
  }

  delete[](solution);
  delete[](b);
  delete[](A);
  return 0;
}
```

Приложение 2. *Код программы вар.1*

```
#include <cmath>
#include <chrono>
#include <iostream>
#include <fstream>
//#include <unistd.h>
#include <winsock2.h>
#include "mpi.h"

using namespace std::chrono;

void
matVecMul(const double *mat, const double *vec, int *sizesPerThreads, int *dispositions, int N, double *newVec) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  auto *tmpVec = new double[sizesPerThreads[rank]];
  for (int i = 0; i < sizesPerThreads[rank]; i++) {
    tmpVec[i] = 0;
    for (int j = 0; j < N; j++) {
      tmpVec[i] += mat[i * N + j] * vec[j];
    }
  }

  MPI_Allgatherv(tmpVec, sizesPerThreads[rank], MPI_DOUBLE, newVec,
           sizesPerThreads, dispositions, MPI_DOUBLE, MPI_COMM_WORLD);
  delete[](tmpVec);
}

void
mulByConst(const double *vec, double c, const int *sizesPerThreads, const int *dispositions, int N, double *newVec) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  auto *tmpVec = new double[N];
  std::fill(tmpVec, tmpVec + N, 0);
  for (int i = dispositions[rank]; i < dispositions[rank] + sizesPerThreads[rank]; i++) {
    tmpVec[i] = vec[i] * c;
  }
  MPI_Allreduce(tmpVec, newVec, N, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  delete[](tmpVec);
}

void
subVec(const double *vec1, const double *vec2, const int *sizesPerThreads, const int *dispositions, int N,
    double *newVec) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  auto *tmpVec = new double[N];
  std::fill(tmpVec, tmpVec + N, 0);
  for (int i = dispositions[rank]; i < dispositions[rank] + sizesPerThreads[rank]; i++) {
    tmpVec[i] = vec1[i] - vec2[i];
  }
  MPI_Allreduce(tmpVec, newVec, N, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  delete[](tmpVec);
}

void
sumVec(const double *vec1, const double *vec2, const int *sizesPerThreads, const int *dispositions, int N,
    double *newVec) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  auto *tmpVec = new double[N];
  std::fill(tmpVec, tmpVec + N, 0);
```

```
  for (int i = dispositions[rank]; i < dispositions[rank] + sizesPerThreads[rank]; i++) {
    tmpVec[i] = vec1[i] + vec2[i];
  }
  MPI_Allreduce(tmpVec, newVec, N, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  delete[](tmpVec);
}

double dotProduct(const double *vec1, const double *vec2, const int *sizesPerThreads, const int *dispositions) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  double sum = 0;
  for (int i = dispositions[rank]; i < dispositions[rank] + sizesPerThreads[rank]; i++) {
    sum += vec1[i] * vec2[i];
  }
  double fullSum;
  MPI_Allreduce(&sum, &fullSum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  return fullSum;
}

void printMat(double *mat, int rows, int columns, std::ofstream &stream) {
  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < columns; ++j) {
      stream << mat[i * columns + j] << " ";
    }
    stream << std::endl;
  }
}

double *solveSLAE(const double *A, double *b, int *sizesPerThreads, int *dispositions, int N) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  auto *solution = new double[N]; // xn+1
  std::fill(solution, solution + N, 0);
  auto *prevSolution = new double[N]; // xn
  std::fill(prevSolution, prevSolution + N, 0);

  auto *Atmp = new double[N];
  auto *r = new double[N];
  auto *z = new double[N];
  auto *rNext = new double[N];
  auto *zNext = new double[N];
  auto *alphaZ = new double[N];
  auto *betaZ = new double[N];

  double alpha;
  double beta;

  const double EPSILON = 1e-009;

  double normb = sqrt(dotProduct(b, b, sizesPerThreads, dispositions));
  double dotRR;

  double res = 1;
  double prevRes = 1;
  bool diverge = false;
  int divergeCount = 0;
  int rightAnswerRepeat = 0;
  int iterCount = 0;
  while (res > EPSILON && rightAnswerRepeat < 5) {
    if (res < EPSILON) {
      ++rightAnswerRepeat;
    } else {
```

```
            rightAnswerRepeat = 0;
        }

        /// rn = b - A * xn
        matVecMul(A, prevSolution, sizesPerThreads, dispositions, N, Atmp);
        subVec(b, Atmp, sizesPerThreads, dispositions, N, r);
        /// zn = rn
        for (int i = 0; i < N; ++i) {
            z[i] = r[i];
        }
        /// alpha
        matVecMul(A, z, sizesPerThreads, dispositions, N, Atmp);
        dotRR = dotProduct(r, r, sizesPerThreads, dispositions);
        alpha = dotRR / dotProduct(Atmp, z, sizesPerThreads, dispositions);
        /// xn+1 = xn + alpha * zn
        mulByConst(z, alpha, sizesPerThreads, dispositions, N, alphaZ);
        sumVec(prevSolution, alphaZ, sizesPerThreads, dispositions, N, solution);
        /// rn+1 = rn - alpha * A * zn
        matVecMul(A, alphaZ, sizesPerThreads, dispositions, N, Atmp);
        subVec(r, Atmp, sizesPerThreads, dispositions, N, rNext);
        /// beta
        beta = dotProduct(rNext, rNext, sizesPerThreads, dispositions) / dotRR;
        /// zn+1 = rn+1 + beta * zn
        mulByConst(z, beta, sizesPerThreads, dispositions, N, betaZ);
        sumVec(rNext, betaZ, sizesPerThreads, dispositions, N, zNext);

        res = sqrt(dotRR) / normb;
        if (prevRes < res || res == INFINITY || res == NAN) {
            ++divergeCount;
            if (divergeCount > 10 || res == INFINITY || res == NAN) {
                diverge = true;
                break;
            }
        } else {
            divergeCount = 0;
        }
        prevRes = res;
        for (long i = 0; i < N; i++) {
            prevSolution[i] = solution[i];
            r[i] = rNext[i];
            z[i] = zNext[i];
        }
        ++iterCount;
    }
    delete[](prevSolution);
    delete[](Atmp);
    delete[](r);
    delete[](z);
    delete[](rNext);
    delete[](zNext);
    delete[](alphaZ);
    delete[](betaZ);

    std::cout << "iterCount: " << iterCount << std::endl;

    if (diverge) {
        delete[](solution);
        return nullptr;
    } else {
        return solution;
    }
}
```

```
int main(int argc, char *argv[]) {
  if (argc != 3) {
    std::cout << "Program needs 2 arguments: size, filename" << std::endl;
    return 0;
  }
  int N = atoi(argv[1]);
  std::string name(argv[2]);

  MPI_Init(&argc, &argv);
  int threadCount, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &threadCount);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  std::string fileName = std::to_string(rank) + "rank-" + name;
  std::ofstream fileStream(fileName);
  if (!fileStream) {
    std::cout << "error with output file" << std::endl;
    MPI_Finalize();
    return 0;
  }

  char host[32];
  gethostname(host, 32);
  std::cout << "Host name: " << host << " rank: " << rank << std::endl;
  if (rank == 0) {
    fileStream << "Matrix size: " << N << " threadCount: " << threadCount << std::endl;
  }

  int sizesPerThreads[threadCount];
  int dispositions[threadCount];
  std::fill(sizesPerThreads, sizesPerThreads + threadCount, N / threadCount);
  sizesPerThreads[threadCount - 1] += N % threadCount;
  dispositions[0] = 0;
  for (int i = 1; i < threadCount; ++i) {
    dispositions[i] = dispositions[i - 1] + sizesPerThreads[i - 1];
  }

  auto *b = new double[N];
  auto *u = new double[N];
  auto *A = new double[sizesPerThreads[rank] * N];

  for (int i = 0; i < sizesPerThreads[rank]; i++) {
    for (int j = 0; j < N; j++) {
      if (i + dispositions[rank] == j) {
        A[i * N + j] = 2;
      } else {
        A[i * N + j] = 1;
      }
    }
  }

  for (int i = 0; i < N; ++i) {
    u[i] = sin(2 * M_PI * i / double(N));
  }
  matVecMul(A, u, sizesPerThreads, dispositions, N, b);

  auto startTime = system_clock::now();
  double *solution = solveSLAE(A, b, sizesPerThreads, dispositions, N);
  auto endTime = system_clock::now();
  auto duration = duration_cast<nanoseconds>(endTime - startTime);

  if (rank == 0 && solution != nullptr) {
    fileStream << "Answer:" << std::endl;
```

```
    printMat(u, 1, N, fileStream);
    fileStream << "SLAE solution:" << std::endl;
    printMat(solution, 1, N, fileStream);
    fileStream << "Time: " << duration.count() / double(1000000000) << "sec" << std::endl;
  } else if (solution == nullptr) {
    fileStream << "Does not converge" << std::endl;
  }

  delete[](solution);
  delete[](b);
  delete[](A);
  MPI_Finalize();
  return 0;
}
```

Приложение 3. *Код программы вар.2*

```
#include <cmath>
#include <chrono>
#include <iostream>
#include <fstream>
//#include <unistd.h>
#include <winsock2.h>
#include "mpi.h"

using namespace std::chrono;

void matVecMul(const double *mat, double *vec, int *sizesPerThreads, int *dispositions, int N, double *newVec) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  auto *tmpVec = new double[N];
  MPI_Allgatherv(vec, sizesPerThreads[rank], MPI_DOUBLE, tmpVec,
           sizesPerThreads, dispositions, MPI_DOUBLE, MPI_COMM_WORLD);
  for (int i = 0; i < sizesPerThreads[rank]; i++) {
    newVec[i] = 0;
    for (int j = 0; j < N; j++) {
      newVec[i] += mat[i * N + j] * tmpVec[j];
    }
  }
  delete[](tmpVec);
}

void mulByConst(const double *vec, double c, const int *sizesPerThreads, double *newVec) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for (int i = 0; i < sizesPerThreads[rank]; i++) {
    newVec[i] = vec[i] * c;
  }
}

void subVec(const double *vec1, const double *vec2, const int *sizesPerThreads, double *newVec) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for (int i = 0; i < sizesPerThreads[rank]; i++) {
    newVec[i] = vec1[i] - vec2[i];
  }
}

void sumVec(const double *vec1, const double *vec2, const int *sizesPerThreads, double *newVec) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for (int i = 0; i < sizesPerThreads[rank]; i++) {
    newVec[i] = vec1[i] + vec2[i];
  }
}

double dotProduct(const double *vec1, const double *vec2, const int *sizesPerThreads) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  double sum = 0;
  for (int i = 0; i < sizesPerThreads[rank]; ++i) {
    sum += vec1[i] * vec2[i];
  }
  double fullSum;
  MPI_Allreduce(&sum, &fullSum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  return fullSum;
}
```

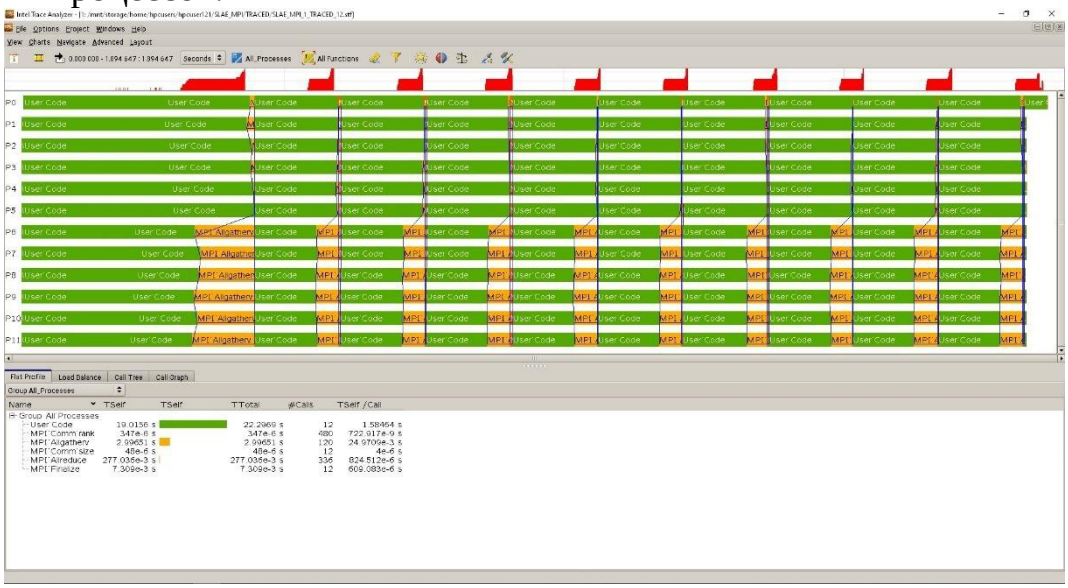
```
void printMat(double *mat, int rows, int columns, std::ofstream &stream) {
  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < columns; ++j) {
      stream << mat[i * columns + j] << " ";
    }
    stream << std::endl;
  }
}

double *solveSLAE(double *A, double *b, int *sizesPerThreads, int *dispositions, int N) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  auto *solution = new double[sizesPerThreads[rank]]; // xn+1
  std::fill(solution, solution + sizesPerThreads[rank], 0);
  auto *prevSolution = new double[sizesPerThreads[rank]]; // xn
  std::fill(prevSolution, prevSolution + sizesPerThreads[rank], 0);

  auto *Atmp = new double[sizesPerThreads[rank]];
  auto *r = new double[sizesPerThreads[rank]];
  auto *z = new double[sizesPerThreads[rank]];
  auto *rNext = new double[sizesPerThreads[rank]];
  auto *zNext = new double[sizesPerThreads[rank]];
  auto *alphaZ = new double[sizesPerThreads[rank]];
  auto *betaZ = new double[sizesPerThreads[rank]];

  double alpha;
  double beta;

  const double EPSILON = 1e-009;

  double normb = sqrt(dotProduct(b, b, sizesPerThreads));
  double dotRR;

  double res = 1;
  double prevRes = 1;
  bool diverge = false;
  int divergeCount = 0;
  int rightAnswerRepeat = 0;
  int iterCount = 0;
  while (res > EPSILON && rightAnswerRepeat < 5) {
    if (res < EPSILON) {
      ++rightAnswerRepeat;
    } else {
      rightAnswerRepeat = 0;
    }

    /// rn = b - A * xn
    matVecMul(A, prevSolution, sizesPerThreads, dispositions, N, Atmp);
    subVec(b, Atmp, sizesPerThreads, r);
    /// zn = rn
    for (int i = 0; i < sizesPerThreads[rank]; ++i) {
      z[i] = r[i];
    }
    /// alpha
    matVecMul(A, z, sizesPerThreads, dispositions, N, Atmp);
    dotRR = dotProduct(r, r, sizesPerThreads);
    alpha = dotRR / dotProduct(Atmp, z, sizesPerThreads);
    /// xn+1 = xn + alpha * zn
    mulByConst(z, alpha, sizesPerThreads, alphaZ);
    sumVec(prevSolution, alphaZ, sizesPerThreads, solution);
    /// rn+1 = rn - alpha * A * zn
    matVecMul(A, alphaZ, sizesPerThreads, dispositions, N, Atmp);
```

```
    subVec(r, Atmp, sizesPerThreads, rNext);
    /// beta
    beta = dotProduct(rNext, rNext, sizesPerThreads) / dotRR;
    /// zn+1 = rn+1 + beta * zn
    mulByConst(z, beta, sizesPerThreads, betaZ);
    sumVec(rNext, betaZ, sizesPerThreads, zNext);

    res = sqrt(dotRR) / normb;
    if (prevRes < res || res == INFINITY || res == NAN) {
      ++divergeCount;
      if (divergeCount > 10 || res == INFINITY || res == NAN) {
        diverge = true;
        break;
      }
    } else {
      divergeCount = 0;
    }
    prevRes = res;
    for (long i = 0; i < sizesPerThreads[rank]; i++) {
      prevSolution[i] = solution[i];
      r[i] = rNext[i];
      z[i] = zNext[i];
    }
    ++iterCount;
  }
  delete[](prevSolution);
  delete[](Atmp);
  delete[](r);
  delete[](z);
  delete[](rNext);
  delete[](zNext);
  delete[](alphaZ);
  delete[](betaZ);

  std::cout << "iterCount: " << iterCount << std::endl;

  if (diverge) {
    delete[](solution);
    return nullptr;
  } else {
    auto *fullSolution = new double[N];
    MPI_Allgatherv(solution, sizesPerThreads[rank], MPI_DOUBLE, fullSolution,
                   sizesPerThreads, dispositions, MPI_DOUBLE, MPI_COMM_WORLD);
    return fullSolution;
  }
}

int main(int argc, char *argv[]) {
  if (argc != 3) {
    std::cout << "Program needs 2 arguments: size, filename" << std::endl;
    return 0;
  }
  int N = atoi(argv[1]);
  std::string name(argv[2]);

  MPI_Init(&argc, &argv);
  int threadCount, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &threadCount);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  std::string fileName = std::to_string(rank) + "rank-" + name;
  std::ofstream fileStream(fileName);
  if (!fileStream) {
```

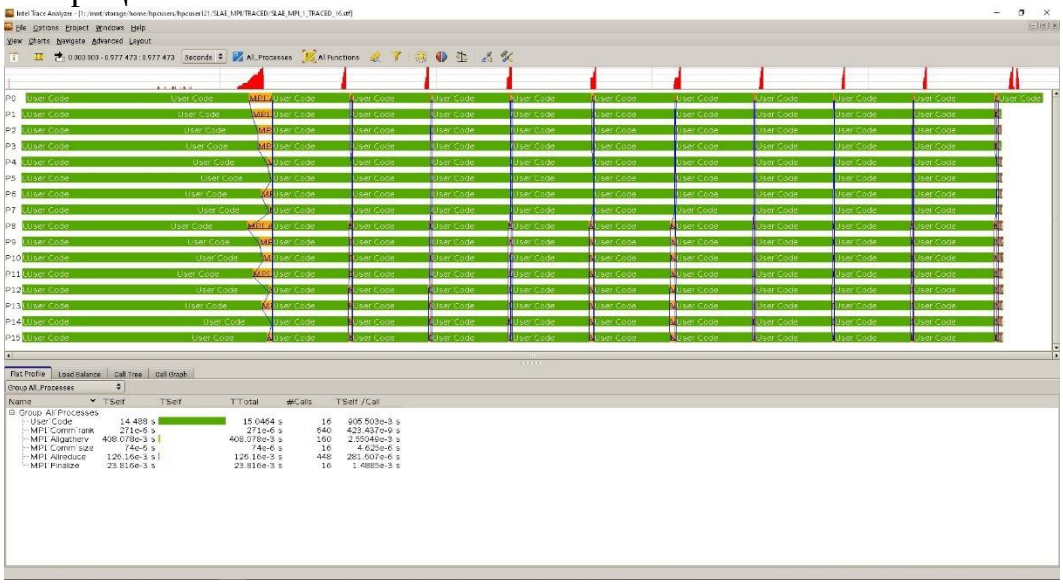
```
    std::cout << "error with output file" << std::endl;
    MPI_Finalize();
    return 0;
  }

  char host[32];
  gethostname(host, 32);
  std::cout << "Host name: " << host << " rank: " << rank << std::endl;
  if (rank == 0) {
    fileStream << "Matrix size: " << N << " threadCount: " << threadCount << std::endl;
  }

  int sizesPerThreads[threadCount];
  int dispositions[threadCount];
  std::fill(sizesPerThreads, sizesPerThreads + threadCount, N / threadCount);
  sizesPerThreads[threadCount - 1] += N % threadCount;
  dispositions[0] = 0;
  for (int i = 1; i < threadCount; ++i) {
    dispositions[i] = dispositions[i - 1] + sizesPerThreads[i - 1];
  }

  auto *b = new double[sizesPerThreads[rank]];
  auto *u = new double[sizesPerThreads[rank]];
  auto *A = new double[sizesPerThreads[rank] * N];

  for (int i = 0; i < sizesPerThreads[rank]; i++) {
    for (int j = 0; j < N; j++) {
      if (i + dispositions[rank] == j) {
        A[i * N + j] = 2;
      } else {
        A[i * N + j] = 1;
      }
    }
    u[i] = sin(2 * M_PI * (i + dispositions[rank]) / double(N));
  }
  matVecMul(A, u, sizesPerThreads, dispositions, N, b);


  auto startTime = system_clock::now();
  double *solution = solveSLAE(A, b, sizesPerThreads, dispositions, N);
  auto endTime = system_clock::now();
  auto duration = duration_cast<nanoseconds>(endTime - startTime);

  auto *fullU = new double[N];
  MPI_Allgatherv(u, sizesPerThreads[rank], MPI_DOUBLE, fullU,
                 sizesPerThreads, dispositions, MPI_DOUBLE, MPI_COMM_WORLD);

  if (rank == 0 && solution != nullptr) {
    fileStream << "Answer" << std::endl;
    printMat(fullU, 1, N, fileStream);
    fileStream << "SLAE solution" << std::endl;
    printMat(solution, 1, N, fileStream);
    fileStream << "Time: " << duration.count() / double(1000000000) << "sec" << std::endl;
  } else if (solution == nullptr) {
    fileStream << "Does not converge" << std::endl;
  }

  delete[](solution);
  delete[](b);
  delete[](A);
  MPI_Finalize();
  return 0;
}
```

Приложение 4. *Профилирование программы вар. 1*

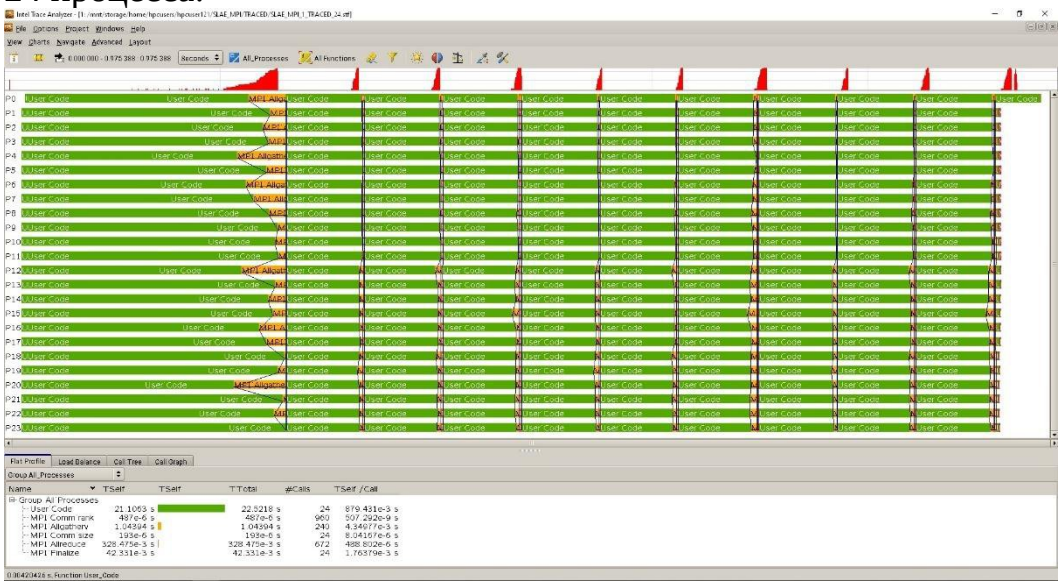
12 процессов:



16 процессов:

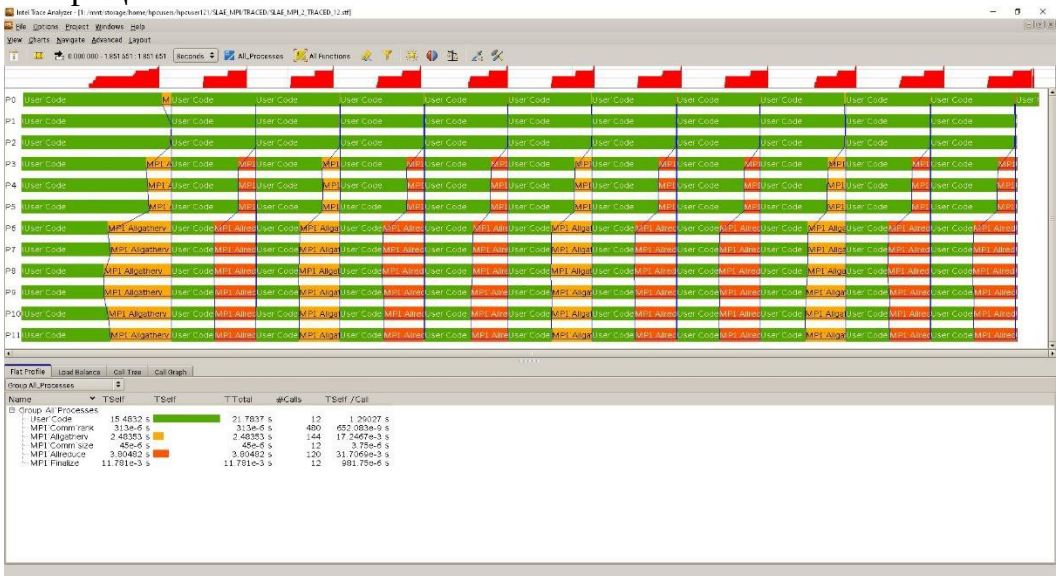


24 процесса:

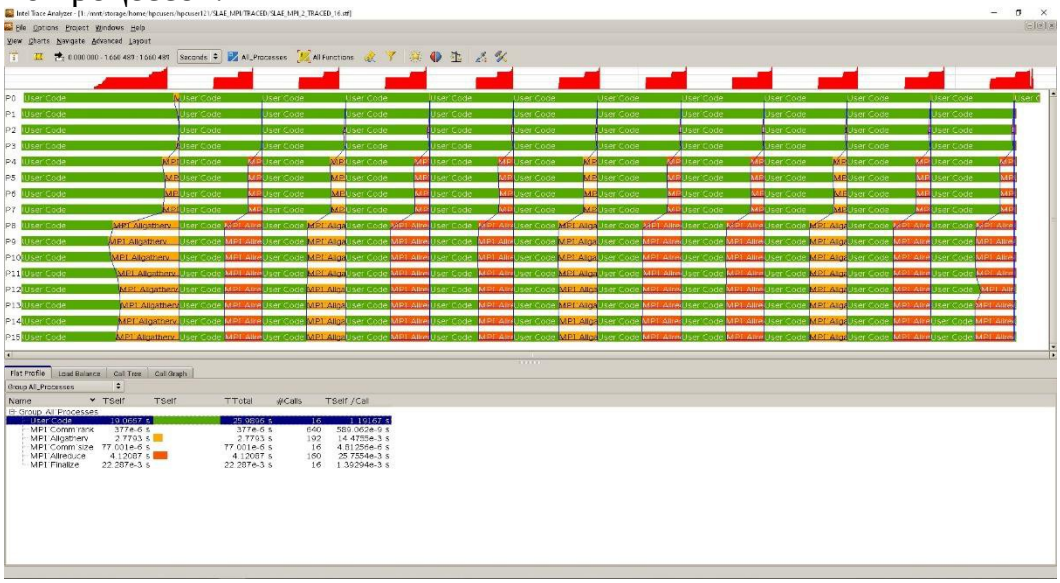


Приложение 5. *Профилирование программы вар. 2*

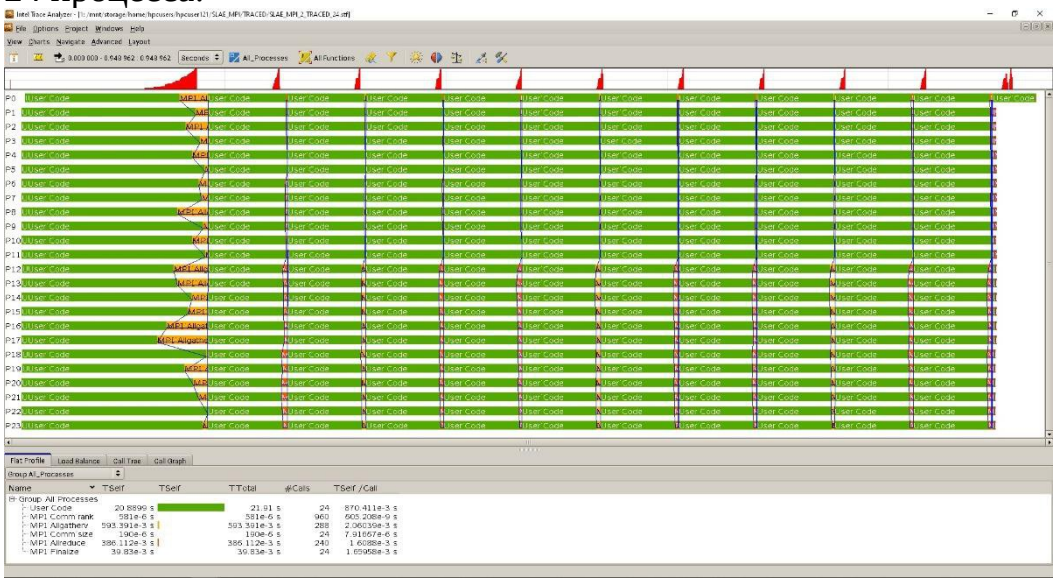
12 процессов:



16 процессов:



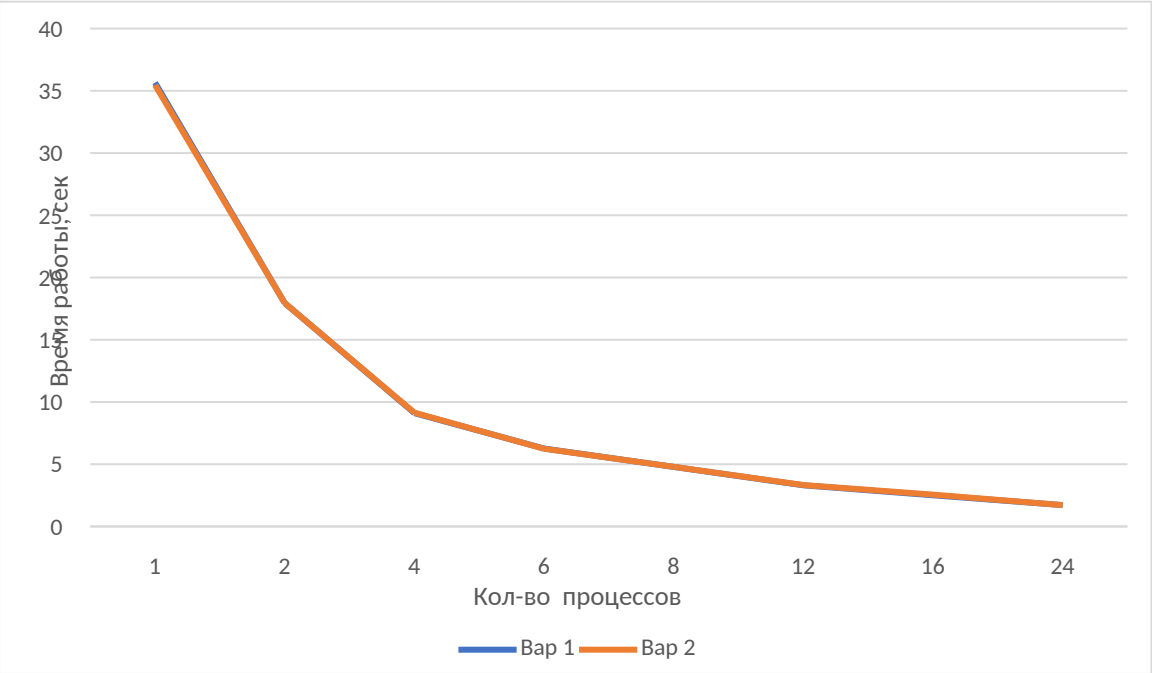
24 процесса:



Приложение 6. *Замеры времени, ускорение, эффективность*

Время работы, сек

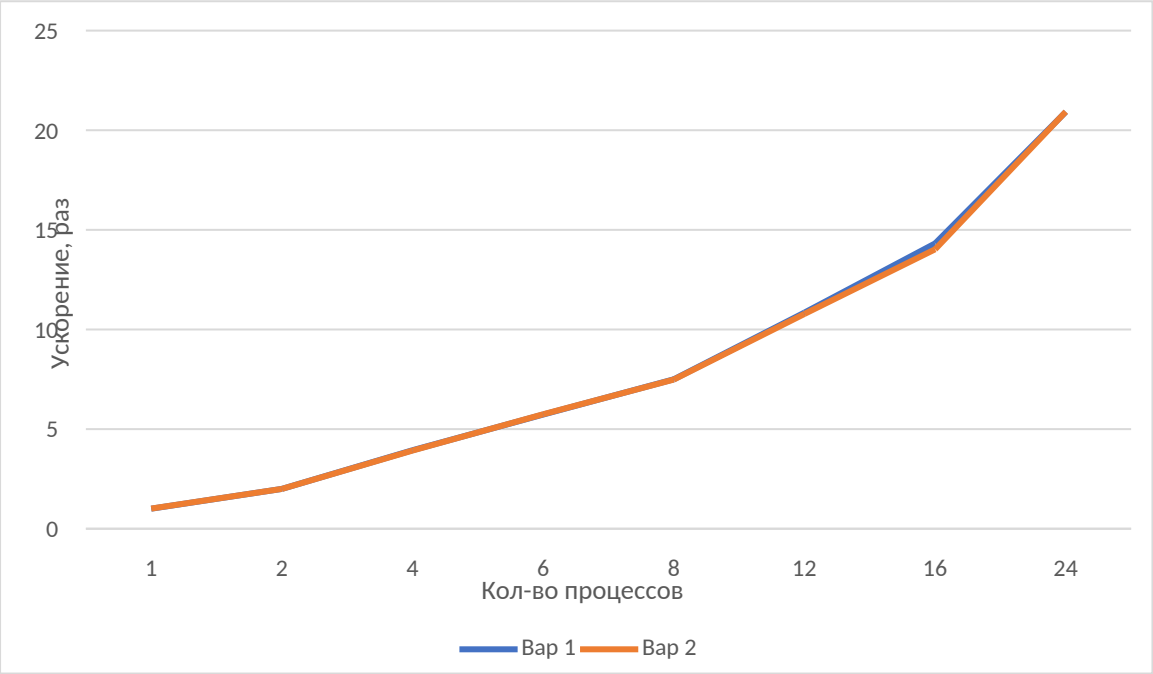
	1	2	4	6	8	12	16	24
Вар 1	35,6533	17,9478	9,10463	6,26056	4,78337	3,30719	2,50541	1,71491
Вар 2	35,4598	17,9495	9,14267	6,24468	4,7919	3,32791	2,55817	1,71412



Время последовательной программы: 35,8761с

Ускорение, раз

	1	2	4	6	8	12	16	24
Вар 1	1	1,998914	3,940424	5,730494	7,500172	10,84791	14,31945	20,92011
Вар 2	1	1,998724	3,924029	5,745066	7,486822	10,78037	14,02413	20,92975



Эффективность на процесс, %

	1	2	4	6	8	12	16	24
Вар 1	1	0,999457	0,985106	0,955082	0,937522	0,903993	0,894966	0,871671
Вар 2	1	0,999362	0,981007	0,957511	0,935853	0,898364	0,876508	0,872073