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 РЕАЛИЗЦАИЯ РЕШЕНИЯ СИСТЕМЫ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИЙ УРАВНЕНИЙ С ПОМОЩЬЮ OPENMP

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 OpenMP.
2. Сравнение времени работы, ускорения и эффективности параллельной программы на разном количестве процессов.

ЗАДАНИЕ

1. Последовательную программу из предыдущей практической работы, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью OpenMP.

   Обязательное условие: создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.

2. Замерить время работы программы на кластере НГУ на 1, 2, 4, 8, 12, 16 потоках. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные и параметры задачи подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.

3. Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков.

4. Составить отчет, содержащий исходные коды разработанных программ и построенные графики.

ОПИСАНИЕ РАБОТЫ

Выбранный метод – метод минимальных невязок.

1. Был создан файл *noopenmp.cpp*, в котором была реализована последовательная программа СЛАУ методом минимальных невязок. Полный компилируемый листинг программы см. Приложение 1.
2. Был создан файл openmp.cpp, в котором реализован итерационный алгоритм решения СЛАУ методом минимальных невязок при помощи OpemMP. Полный компилируемый листинг программы см. в Приложении 2.
3. Корректность работы программы была проверена со следующими входными параметрами: размер матрицы равен 10.

   Элементы главной диагонали A равны 2.0, остальные равны 1.0. Вектор u, элементы которого заполняются произвольными значениями: $u = sin\ \frac{2\pi i}{N}$. 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.
4. Результаты проверки корректности см. Приложение 3.
5. Был создан файл openmp.cpp, в котором представлена «распараллеленная» программа. Размер матрицы, при котором время параллельной программы на одном процессе составляет не менее 30 сек: 2000, время работы: 54.027 сек.
6. Было измерено время работы программы на 1, 2, 4, 8, 12, 16 процессах.

   Команда для компиляции:

   g++ -fopenmp openmp.cpp -o with_openmp

   Команда для запуска:

   ./with_openmp 2000 [amount of threads], где ‘amount of threads’ – число потоков
7. Результаты измерений программы на 1, 2, 4, 8, 12, 16 процессах представлены в Приложении 4.
8. Полученные результаты измерения времени представлены в Таблице 1.

Таблица1. Результаты измерений времени

Количество процессов	Время работы параллельной программы, с
1	54,02700
2	26,97130
4	14,40470
8	7,33639
12	5,38880
16	4,37053

9. На основании полученных данных рассчитаем эффективность и ускорение. Графики результатов представлены в Приложении 5.
10. При помощи директивы #pragma omp for schedule(...) проведем исследование на нахождение оптимальных параметров. Листинг программы (test_openmp.cpp) для исследования с директивой #pragma omp for schedule представлен в Приложении 6. Для чего нужна эта директива? Предполагается, что корректная программа не должна зависеть от того, какой именно тред какую именно итерацию цикла выполнит. Существует 5 видов опций для этой директивы:
    - **static** - блочно-циклическое распределение цикла. размер блока - chunk. Первый блок из chunk_size итераций выполняет нулевая нить, второй блок - вторая нить и т.д. По дефолту размер chunk_size = кол-во итераций / число нитей.
    - **dynamic** - динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает chunk_size итераций; та нить, которая заканчивает выполнение своей порции итераций получает первую свободную порцию из chunk_size итерация. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Распределение по потокам происходит без определенного порядка.
    - **guided** - динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины chunk_size (по дефолту его размер равен 1) пропорционально количеству еще не распределенных итераций, деленных на количество нитей, выполняющих цикл. (Каждый последующий блок меньше предыдущего).
    - **auto** - способ распределения итераций выбирается компилятором. Параметр chunk_size не задается.
    - **runtime** - способ распределения итераций выбирается во время работы программы по значению переменной среды OMP_SCHEDULE. Параметр chunk_size не задается.
11. Перейдем к исследованию. Размер матрицы возьмем 1000 * 1000. Количество потоков равно 4.

    Проведем 3 «испытания»:
    - В первом размер chunk_size будет дефолтным.
    - Во втором размер chunk_size будет равен 100.
    - В третьем размер chunk_size будет равен 200.

Измерим время работы с такими параметрами для каждой из опций. Представим его в таблице №2.

Таблица №2. Результаты исследования

Тип schedule	Кол-во потоков	Размер chunk Здесь по дефолту	Время работы, сек	Размер chunk	Время работы, сек	Размер chunk	Время работы, сек
auto	4	?	4,17624	?	4,17624	?	4,17624
static	4	612/4 = 153	4,27951	100	15,9614	200	13,395
dynamic	4	1	135,495	100	55,1335	200	52,8437
guided	4	1	4,73732	100	4,72655	200	4,70682
runtime	4	1	56,2108	100	55,4561	200	53,2337

Параметры программ представлены в Приложении 7.

График зависимости времени от размера чанка представлен в Приложении 8.

На основании графика и данных таблицы №2 можно сделать вывод о том, что оптимальными параметрами являются auto и guided. Параметр static ведет себя оптимально благодаря однородности данных. Самый худший результат показывает dynamic из-за накладных расчётов треда на следующую порцию итераций.

ЗАКЛЮЧЕНИЕ

В ходе работы мы смогли «распараллелить» программу для решения СЛАУ посредством использования OpenMP.

Разработка параллельной программы с помощью директив этой библиотеки оказалось значительно быстрее благодаря своей простоте по сравнению с разработкой программы с MPI.

Из недостатков этой технологии можно отметить, что при параллельном выполнении программы необходима синхронизация вычислений, выполняемых в разных потоках, за это приходится платить увеличением времени работы программы.

ИСТОЧНИКИ ИНФОРМАЦИИ

1. Лекции с сайта кафедры Параллельных вычислений НГУ [URL]: https://ssd.sscc.ru/ru/chair/nsu/parallel-programming

2. Онлайн учебник по OpenMP [URL]: https://pro-prof.com/archives/4335

3. Спецификация OpenMP [URL]: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

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 multimplyMatrixOnVector(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);
  multimplyMatrixOnVector(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) {
  multimplyMatrixOnVector(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);
  multimplyMatrixOnVector(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. Листинг параллельной программы с использованием OpenMP

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <omp.h>

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 multimplyMatrixOnVector(const double *matrix,
                             const double *vector, double *res,
                             const size_t sizeInput) {
#pragma omp for
  for (int i = 0; i < sizeInput; i++) {
    res[i] = 0;
  }
#pragma omp for collapse(2)
  for (size_t i = 0; i < sizeInput; ++i) {
    for (size_t j = 0; j < sizeInput; ++j) {
#pragma omp atomic

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

/*============ only for test, starts ============*/
/* for testing when vector b uses sin*/
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);
  multimplyMatrixOnVector(fullMatrixA, vectorU, b, sizeInput);
```

```
  delete[] vectorU;
}
/*============ only for test, ended ============*/

void countScalarMult(const double *vector1, const double *vector2,
                     const size_t sizeInput, double *res) {
#pragma omp single
  { *res = 0; }
#pragma omp barrier

#pragma omp for reduction(+ : res[0])
  for (size_t i = 0; i < sizeInput; ++i) {
    *res += vector1[i] * vector2[i];
  }
}

void countVectorLength(const size_t sizeInput, const double *vector,
                       double *res) {
#pragma omp single
  { *res = 0; }
#pragma omp barrier

#pragma omp for reduction(+ : res[0])
  for (size_t i = 0; i < sizeInput; ++i) {
    *res += vector[i] * vector[i];
  }
#pragma omp single
  { *res = sqrt(*res); }
#pragma omp barrier
}

void countVectorMultNumber(const double *vector, double scalar,
                           double *res, const size_t sizeInput) {
#pragma omp for
  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) {
#pragma omp for
  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) {
#pragma omp for
```

```
  for (size_t i = 0; i < sizeInput; i++) {
    dst[i] = src[i];
  }
}

int main(int argc, char *argv[]) {
  // In this code the 1st arg is sizeMatrix
  // the 2nd arg is amount of threads

  if (argc != 3) {
    std::cout << "Bad input! Enter matrix size" << std::endl;
  }
  srand(0);

  const size_t sizeInput = atoi(argv[1]);
  const int amountOfThreads = atoi(argv[2]);

  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;

  double startt = omp_get_wtime();

  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, starts ========= */
  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, ended ========= */

  ///* for testing when vector b uses sin starts ========= */
  // 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 ended ========= */

  double bNorm, yNorm;
  countVectorLength(sizeInput, b, &bNorm);

  double numeratorTau, denominatorTau;

#pragma omp parallel num_threads(amountOfThreads)
  {
    while (1) {
      multimplyMatrixOnVector(matrixA, xCurr, Atmp,
                              sizeInput); // A * x_n

      substructVectors(Atmp, b, y, sizeInput); // y_n = A * x_n - b

      countVectorLength(sizeInput, y,
                        &yNorm); // || A * x_n - b ||

      multimplyMatrixOnVector(matrixA, y, Atmp, sizeInput); // A * y_n

      countScalarMult(y, Atmp, sizeInput,
                      &numeratorTau); // (y_n, A * y_n)
      countScalarMult(Atmp, Atmp, sizeInput,
                      &denominatorTau); // (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;
        abort();
      }

      if (critCurrentEnd < epsilon && critCurrentEnd < prevCritEnd &&
          critCurrentEnd < prevPrevCritEnd) {

        isEndOfAlgo = true;
        break;
      }

      copyVector(xNext, xCurr, sizeInput);

#pragma omp barrier

      prevPrevCritEnd = prevCritEnd;
      prevCritEnd = critCurrentEnd;

#pragma omp single
      iterationCounts++;

#pragma omp barrier

    }
  }

  double endt = omp_get_wtime();

  if (isEndOfAlgo) {
    std::cout << "Iteration amount in total: " << iterationCounts
              << std::endl;
    std::cout << "Time taken: " << endt - startt << " 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 3. Проверка корректности работы параллельной программы

```
dasha@dasha-K501UQ:~/masec/opp/pract3/code$ ./with_openmp 10 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       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 amount in total: 4
Time taken: 0.002491 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. Результат на 1м процессе

```
dasha@dasha-K501UQ:~/masec/opp/pract3/code$ ./with_openmp 10 2
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 amount in total: 4
Time taken: 0.001965 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. 2. Результат на 2х процессах

```
dasha@dasha-K501UQ:~/masec/opp/pract3/code$ ./with_openmp 10 4
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 amount in total: 4
Time taken: 0.069185 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. 2. Результат на 4х процессах

Приложение 4. Результаты работы параллельной программы на 1, 2, 4, 8, 16, 24 процессах

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_openmp 2000 1
Iteration amount in total: 526
Time taken: 54.027 sec
```

Рис. 1. Время работы параллельной программы на 1м процессе

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_openmp 2000 2
Iteration amount in total: 526
Time taken: 26.9713 sec
```

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

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_openmp 2000 4
Iteration amount in total: 526
Time taken: 14.4047 sec
```

Рис. 3. Время работы параллельной программы на 4х процессах

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_openmp 2000 8
Iteration amount in total: 526
Time taken: 7.33639 sec
```

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

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_openmp 2000 12
Iteration amount in total: 526
Time taken: 5.3888 sec
```

Рис. 5. Время работы параллельной программы на 12и процессах

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_openmp 2000 16
Iteration amount in total: 526
Time taken: 4.37053 sec
```

Рис. 6. Время работы параллельной программы на 16 процессах

Приложение 5. Графики времени, ускорения и эффективности

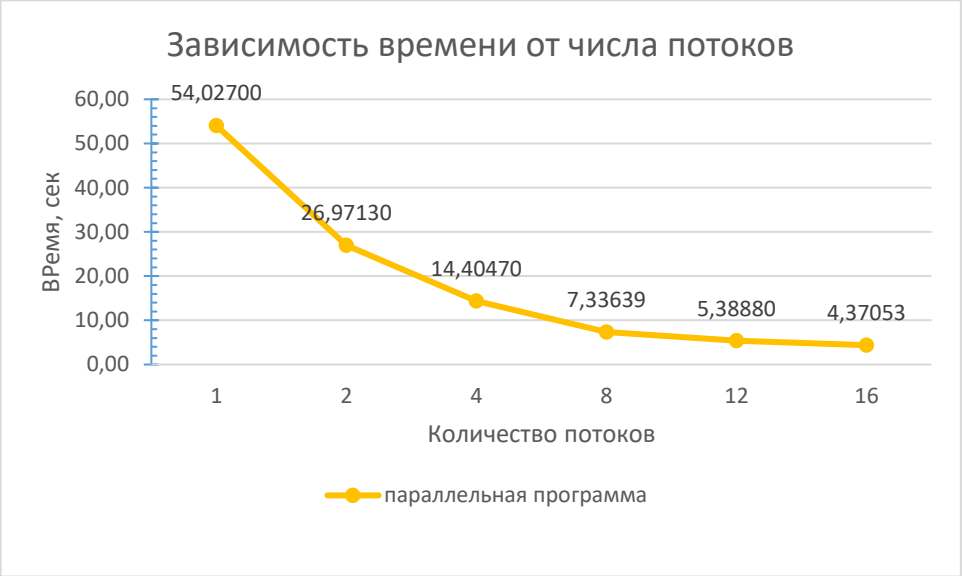

Рис. 1. График зависимости от времени

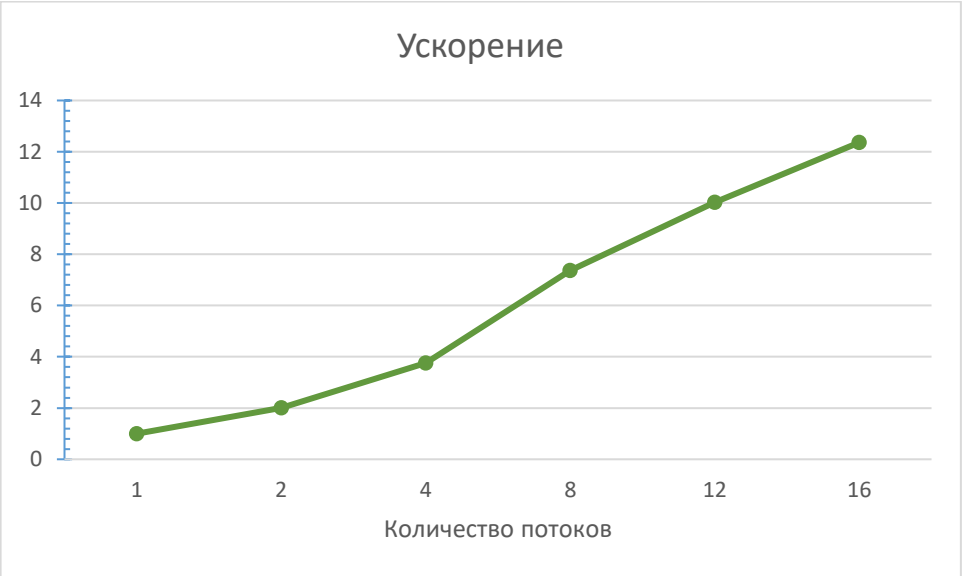

Рис. 2. График ускорения

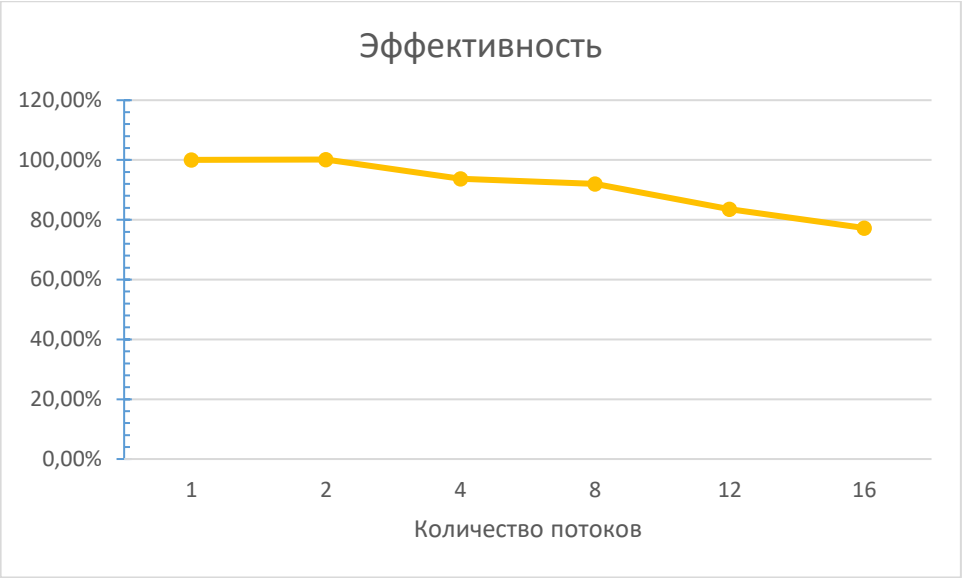

Рис. 3. График эффективности

Приложение 6. Листинг параллельной программы с использованием директивы

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <omp.h>

/* for printinf type of scheduling*/
#define xstr(x) str(x)
#define str(x) #x

// #define TYPE auto
#define TYPE static
// #define TYPE dynamic
// #define TYPE guided
// #define TYPE runtime

#define CHUNK_SIZE 200

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 *fillRandomVector(const size_t sizeInput) {
  double *vector = new double[sizeInput];
  for (size_t i = 0; i < sizeInput; ++i) {
    vector[i] = rand() % 500;
  }
  return vector;
}

void multimplyMatrixOnVector(const double *matrix,
                             const double *vector, double *res,
                             const size_t sizeInput) {
#pragma omp for schedule(TYPE, CHUNK_SIZE)
  for (int i = 0; i < sizeInput; i++) {
    res[i] = 0;
  }
#pragma omp for collapse(2) schedule(TYPE, CHUNK_SIZE)
  for (size_t i = 0; i < sizeInput; ++i) {
    for (size_t j = 0; j < sizeInput; ++j) {
#pragma omp atomic

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

void countScalarMult(const double *vector1, const double *vector2,
                     const size_t sizeInput, double *res) {
#pragma omp single
  { *res = 0; }
#pragma omp barrier

#pragma omp for reduction(+ : res[0]) schedule(TYPE, CHUNK_SIZE)
  for (size_t i = 0; i < sizeInput; ++i) {
    *res += vector1[i] * vector2[i];
  }
}

void countVectorLength(const size_t sizeInput, const double *vector,
                       double *res) {
#pragma omp single
  { *res = 0; }
#pragma omp barrier

#pragma omp for reduction(+ : res[0]) schedule(TYPE, CHUNK_SIZE)
  for (size_t i = 0; i < sizeInput; ++i) {
    *res += vector[i] * vector[i];
  }
#pragma omp single
  { *res = sqrt(*res); }
#pragma omp barrier
}
```

```
void countVectorMultNumber(const double *vector, double scalar,
                           double *res, const size_t sizeInput) {
#pragma omp for
  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) {
#pragma omp for schedule(TYPE, CHUNK_SIZE)
  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) {
#pragma omp for schedule(TYPE, CHUNK_SIZE)
  for (size_t i = 0; i < sizeInput; i++) {
    dst[i] = src[i];
  }
}

int main(int argc, char *argv[]) {
  // In this code the 1st arg is sizeMatrix
  // the 2nd arg is amount of threads

  if (argc != 3) {
    std::cout << "Bad input! Enter matrix size" << std::endl;
  }
  srand(0);

  const size_t sizeInput = atoi(argv[1]);
  const int amountOfThreads = atoi(argv[2]);

  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;

  double startt = omp_get_wtime();
```

```
  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, starts ========= */
  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, ended ========= */

  double bNorm, yNorm;
  countVectorLength(sizeInput, b, &bNorm);

  double numeratorTau, denominatorTau;

#pragma omp parallel num_threads(amountOfThreads)
  {
    while (1) {
      multimplyMatrixOnVector(matrixA, xCurr, Atmp,
                              sizeInput); // A * x_n

      substructVectors(Atmp, b, y, sizeInput); // y_n = A * x_n - b

      countVectorLength(sizeInput, y,
                        &yNorm); // || A * x_n - b ||

      multimplyMatrixOnVector(matrixA, y, Atmp, sizeInput); // A * y_n

      countScalarMult(y, Atmp, sizeInput,
                      &numeratorTau); // (y_n, A * y_n)
      countScalarMult(Atmp, Atmp, sizeInput,
                      &denominatorTau); // (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;

        abort();
      }

      if (critCurrentEnd < epsilon && critCurrentEnd < prevCritEnd &&
          critCurrentEnd < prevPrevCritEnd) {

        isEndOfAlgo = true;
        break;
      }

      copyVector(xNext, xCurr, sizeInput);

#pragma omp barrier

      prevPrevCritEnd = prevCritEnd;
      prevCritEnd = critCurrentEnd;

#pragma omp single
      iterationCounts++;

#pragma omp barrier
    }
  }

  double endt = omp_get_wtime();

  if (isEndOfAlgo) {
    std::cout << "Iteration amount in total: " << iterationCounts
              << std::endl;
    std::cout << "Time taken: " << endt - startt << " sec"
              << std::endl;

    std::cout << "Type of scheduling is: " << xstr(TYPE) << std::endl;
    std::cout << "Chunk size is: " << xstr(CHUNK_SIZE) << std::endl;

  } 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 7. Результаты работы параллельной программы на 4х процессах, матрица размера 1000 * 1000

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 4.17624 sec
Type of scheduling is: auto
```

Рис. 1. Время работы на типе auto, размер чанка по дефолту

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 4.27951 sec
Type of scheduling is: static
```

Рис. 2. Время работы на типе static, размер чанка по дефолту

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_openmp 2000 4
Iteration amount in total: 526
Time taken: 14.4047 sec
```

Рис. 3. Время работы на типе dynamic, размер чанка по дефолту

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 4.73732 sec
Type of scheduling is: guided
```

Рис. 4. Время работы на типе guided, размер чанка по дефолту

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 56.2108 sec
Type of scheduling is: runtime
```

Рис. 5. Время работы на типе runtime, размер чанка по дефолту

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 15.9614 sec
Type of scheduling is: static
Chunk size is: 100
```

Рис. 6. Время работы на типе static, размер чанка равен 100

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 55.1335 sec
Type of scheduling is: dynamic
Chunk size is: 100
```

Рис. 7. Время работы на типе dynamic, размер чанка равен 100

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 4.72655 sec
Type of scheduling is: guided
Chunk size is: 100
```
Рис. 8. Время работы на типе guided, размер чанка равен 100

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 55.4561 sec
Type of scheduling is: runtime
```
Рис. 9. Время работы на типе runtime, размер чанка равен 100

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 13.395 sec
Type of scheduling is: static
Chunk size is: 200
```
Рис. 10. Время работы на типе static, размер чанка равен 200

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 52.8437 sec
Type of scheduling is: dynamic
Chunk size is: 200
```
Рис. 11. Время работы на типе dynamic, размер чанка равен 200

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 4.70682 sec
Type of scheduling is: guided
Chunk size is: 200
```
Рис. 12. Время работы на типе guided, размер чанка равен 200

```
opp@comrade:~/205/Evdokimova/laba3$ ./with_test_openmp 1000 4
Iteration amount in total: 612
Time taken: 53.2337 sec
Type of scheduling is: runtime
```
Рис. 13. Время работы на типе runtime, размер чанка равен 200

Приложение 8. График зависимости времени от размера чанка

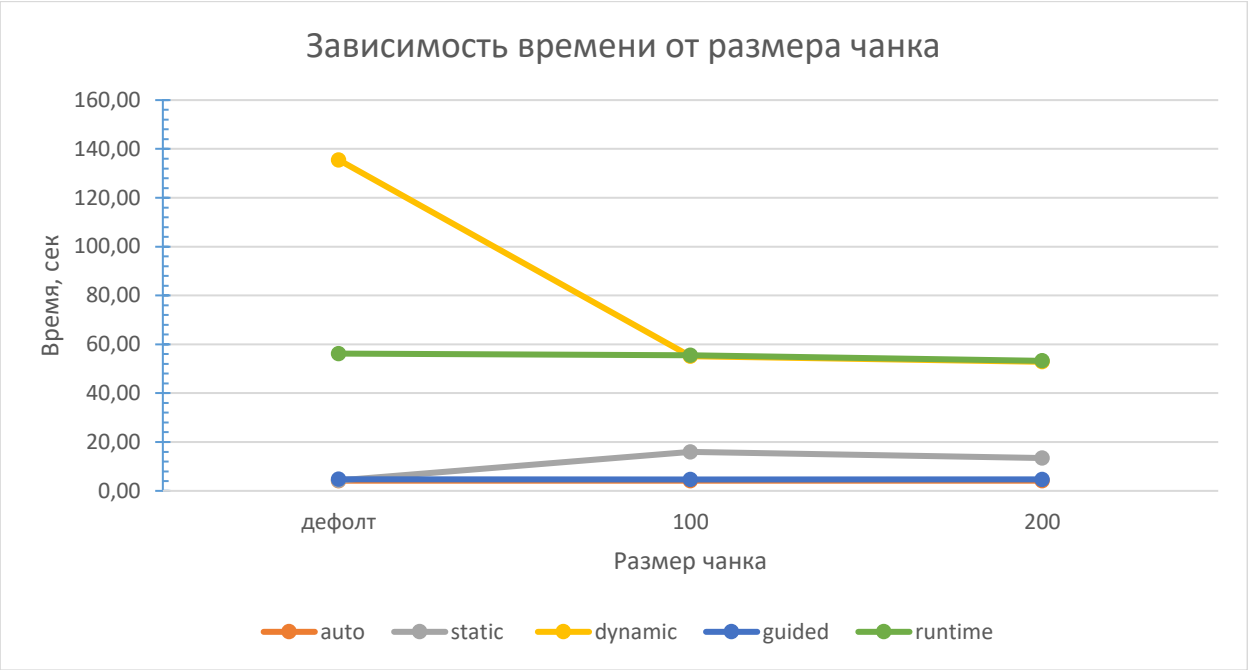

Рис. 1. График зависимости времени от размера чанка