###### 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 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 3](#__RefHeading___Toc3482_1127455112)

[​ ЗАДАНИЕ 3](#__RefHeading___Toc3484_1127455112)

[​ ОПИСАНИЕ РАБОТЫ 4](#__RefHeading___Toc3486_1127455112)

[​ ЗАКЛЮЧЕНИЕ 7](#__RefHeading___Toc3488_1127455112)

[​ СПИСОК ЛИТЕРАТУРЫ 8](#__RefHeading___Toc3490_1127455112)

[Приложение 1. Листинг последовательной программы 9](#__RefHeading___Toc3492_1127455112)

[Приложение 2. Листинг параллельной программы с использованием OpenMP 15](#__RefHeading___Toc3494_1127455112)

[​ Приложение 3. Проверка корректности работы параллельной программы 22](#__RefHeading___Toc3496_1127455112)

[​ Приложение 4. Результаты работы параллельной программы на 1, 2, 4, 8, 16, 24 процессах 23](#__RefHeading___Toc3498_1127455112)

[​ Приложение 5. Графики времени, ускорения и эффективности 24](#__RefHeading___Toc3500_1127455112)

[Приложение 6. Листинг параллельной программы с использованием директивы 26](#__RefHeading___Toc3502_1127455112)

[​ Приложение 7. Результаты работы параллельной программы на 4х процессах, матрица размера 1000 \* 1000 32](#__RefHeading___Toc3504_1127455112)

[​ Приложение 8. График зависимости времени от размера чанка 34](#__RefHeading___Toc3506_1127455112)

# 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.

1. Замерить время работы программы на кластере или кафедральном сервере 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.

1. Провести исследование на определение оптимальных параметров

#pragma omp for schedule(...) при некотором фиксированном размере

задачи и количестве потоков.

1. 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: . Элементы вектора b получаются путем умножения матрицы A на вектор u. В этом случае решением системы будет вектор, равный вектору u. Начальные значения элементов вектора x равны 0.

1. Результаты проверки корректности см. Приложение 3.
2. 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.
3. 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 [matrixSize] [amount of threads]*, где ‘amount of threads’ – число потоков, ‘matrixSize’ – размер матрицы.

1. Результаты измерений программы на 1, 2, 4, 8, 12, 16 процессах представлены в Приложении 4.
2. Полученные результаты измерения времени представлены в Таблице 1.

Таблица1. Результаты измерений времени

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

1. На основании полученных данных рассчитаем эффективность и ускорение. Графики результатов представлены в Приложении 5.
2. 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? Она должна разделить цикл по итерациям между потоками. 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\_size. 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 итераций. 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.

1. 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 | Время работы, сек | Размер chunk | Время работы, сек | Размер chunk | Время работы, сек |
| Здесь по дефолту |
| 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 определение следующей порции итераций.

# ЗАКЛЮЧЕНИЕ

В ходе работы мы смогли написать параллельную программу для решения СЛАУ посредством использования директив библиотеки 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.

# **СПИСОК ЛИТЕРАТУРЫ**

1. Лекции с сайта кафедры Параллельных вычислений НГУ [Электронный ресурс]. URL: <https://ssd.sscc.ru/ru/chair/nsu/parallel-programming>
2. Онлайн учебник по OpenMP [Электронный ресурс]. URL: <https://pro-prof.com/archives/4335>
3. Спецификация OpenMP [Электронный ресурс]. URL: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf>
4. Сайт с пояснением sheduling-а [Электронный ресурс]. URL: https://610yilingliu.github.io/2020/07/15/ScheduleinOpenMP/

1. Статья с Хабра про atomic и reduction [Электронный ресурс]. URL: <https://habr.com/ru/company/intel/blog/88574/>

# 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 multiplyMatrixOnVector(const double \*matrix,

const double \*vector, double \*res,

const size\_t sizeInput) {

#pragma omp for

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];

}

}

}

/\*============ 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);

multiplyMatrixOnVector(fullMatrixA, vectorU, b, sizeInput);

delete[] vectorU;

}

/\*============ only for test, ended ============\*/

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

const size\_t sizeInput) {

double res = 0;

#pragma omp parallel for reduction(+ : res)

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) {

#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;

double numeratorTau, denominatorTau;

#pragma omp parallel num\_threads(amountOfThreads) \

shared(iterationCounts, critCurrentEnd, prevPrevCritEnd, \

isEndOfAlgo, Atmp, y, tauY, xCurr, xNext, matrixA, b, \

bNorm, yNorm, numeratorTau, denominatorTau)

{

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

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

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

numeratorTau =

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

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;

abort();

}

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

critCurrentEnd < prevPrevCritEnd) {

isEndOfAlgo = true;

break;

}

copyVector(xNext, xCurr, sizeInput);

#pragma omp barrier

#pragma omp single

{

prevPrevCritEnd = prevCritEnd;

prevCritEnd = critCurrentEnd;

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

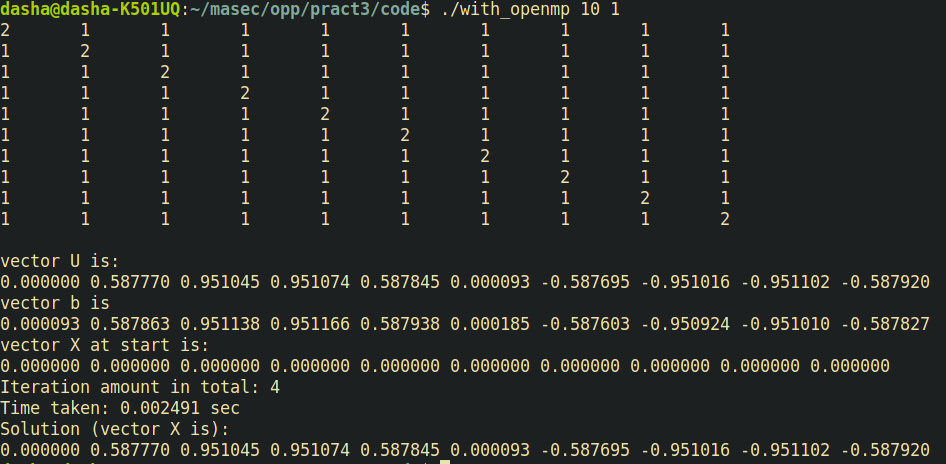


Рис. 1. Результат на 1м процессе

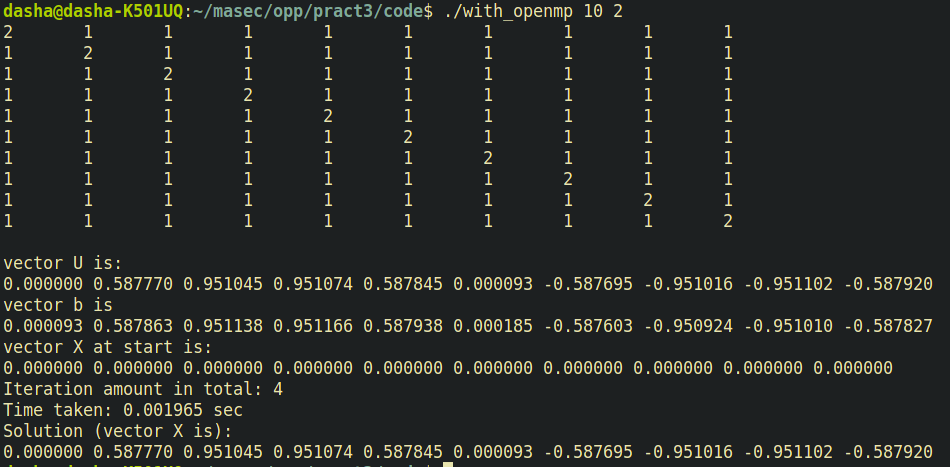


Рис. 2. Результат на 2х процессах

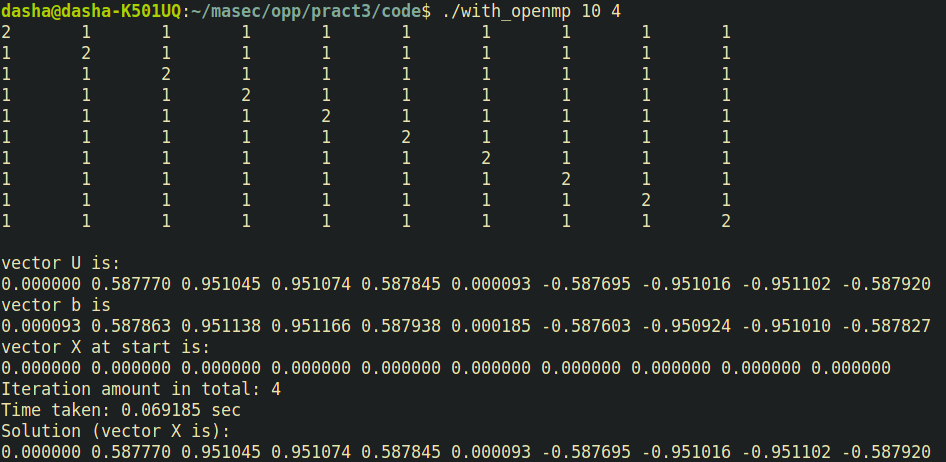


Рис. 2. Результат на 4х процессах

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

https://lh4.googleusercontent.com/Z_0J-wy_gxri1eMwwYgAAOHPV8qF_CZOlL0SIG7oHBmodNwFy-EYqnfyr1XVQgi-Xj_F-6XF2cB7f-ldIyMfUPC2LvZjOPWYjm2jxSYg1SWmB23CiM0YyD_qYv9p5ffMWDkZElcYziU8TmiMGkJKSEQ

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

https://lh5.googleusercontent.com/wjD4gJMDhQUW8RCzQAMbQ6qOkeoQl_Q5c-lzak8ROI6-ZCjr0hsDh-Oxs06B-lja4fMdhJqgYIUFyRY5zMxh5lAyrSm1frEBJkq-b7xSVf4nATXoJjz4XEKPvQZStj8eI9IqQzrGkZIe5piucDdWqrc­

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

https://lh6.googleusercontent.com/ll2fMs2nfAE_xF1VqGm4ZJ2oMkWYe5qyTdjHiUV2VCZemzziWlHcgC7GUW_K9iwzi3s3vnCwdM6PbcOm1tcNAK-WjgZCUBhvCosd_rSOaJe7-wvX8kb7AAF1ZSH1xb0vPwP9VtIZAr6url7nm9yvlTg

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

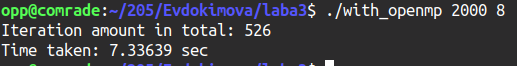


Рис. 4. Время работы параллельной программы на 8и процессах

https://lh5.googleusercontent.com/6G9YOi8KNHS04a2KyY3KaRXff4VYm6hF5tlmdaJc_mImxDYVrv2SyZ--a_y3HoHCPz5iBnrPTq10rpJ6XJ-YJmXPR5W_RIbMF0HMddKenSUE7PFX-kTIljQC_hp-xgI1_khfmo_B8BOkE0GHSftvPxk

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

https://lh3.googleusercontent.com/7H07IU_rcliiJvMEMCWfL2YBVYnrKHsfs_0btEKOhroRQIVLyPa9godmyOHGpzkba9mr-Ycox4MM3-5v18-HyIo4v9FGsDU_vCgNcJAll1Fcakw07IeSrb1Ubc0WflzKw2qJ5CgrmkQqplatvsJnzVg

Рис. 6. Время работы параллельной программы на 16 процессах

# Приложение 5. Графики времени, ускорения и эффективности

1. Рис. 1. График зависимости от времени
2. 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 \*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) {

#pragma omp for schedule (TYPE, CHUNK\_SIZE)

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];

}

}

}

/\*============ 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);

multiplyMatrixOnVector(fullMatrixA, vectorU, b, sizeInput);

delete[] vectorU;

}

/\*============ only for test, ended ============\*/

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

const size\_t sizeInput) {

double res = 0;

#pragma omp parallel for reduction(+ : res) schedule (TYPE, CHUNK\_SIZE)

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) {

#pragma omp for schedule (TYPE, CHUNK\_SIZE)

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 ========= \*/

///\* 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;

double numeratorTau, denominatorTau;

#pragma omp parallel num\_threads(amountOfThreads) \

shared(iterationCounts, critCurrentEnd, prevPrevCritEnd, \

isEndOfAlgo, Atmp, y, tauY, xCurr, xNext, matrixA, b, \

bNorm, yNorm, numeratorTau, denominatorTau)

{

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

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

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

numeratorTau =

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

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;

abort();

}

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

critCurrentEnd < prevPrevCritEnd) {

isEndOfAlgo = true;

break;

}

copyVector(xNext, xCurr, sizeInput);

#pragma omp barrier

#pragma omp single

{

prevPrevCritEnd = prevCritEnd;

prevCritEnd = critCurrentEnd;

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

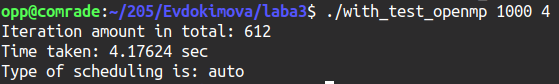


Рис. 1. Время работы на типе auto, размер чанка по дефолту

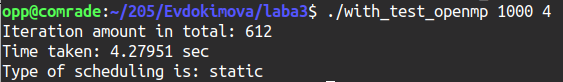


Рис. 2. Время работы на типе static, размер чанка по дефолту

https://lh6.googleusercontent.com/ll2fMs2nfAE_xF1VqGm4ZJ2oMkWYe5qyTdjHiUV2VCZemzziWlHcgC7GUW_K9iwzi3s3vnCwdM6PbcOm1tcNAK-WjgZCUBhvCosd_rSOaJe7-wvX8kb7AAF1ZSH1xb0vPwP9VtIZAr6url7nm9yvlTg

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

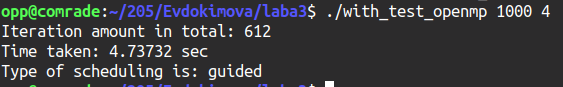


Рис. 4. Время работы на типе guided, размер чанка по дефолту

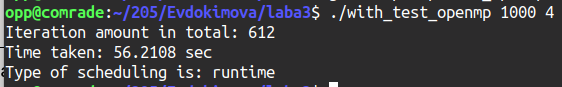


Рис. 5. Время работы на типе runtime, размер чанка по дефолту

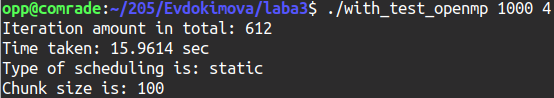


Рис. 6. Время работы на типе static, размер чанка равен 100

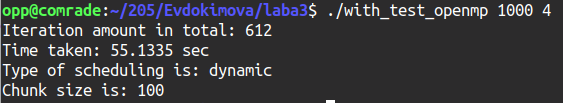


Рис. 7. Время работы на типе dynamic, размер чанка равен 100

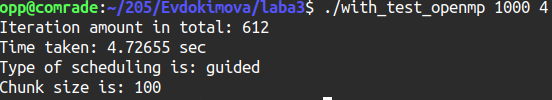


Рис. 8. Время работы на типе guided, размер чанка равен 100

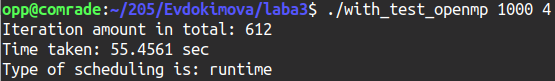


Рис. 9. Время работы на типе runtime, размер чанка равен 100

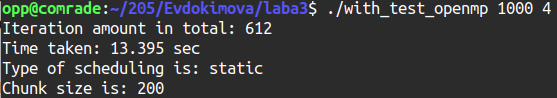


Рис. 10. Время работы на типе static, размер чанка равен 200

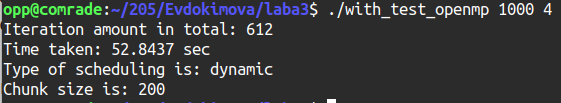


Рис. 11. Время работы на типе dynamic, размер чанка равен 200

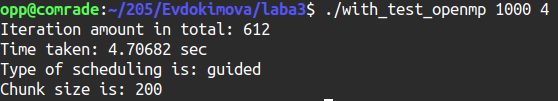


Рис. 12. Время работы на типе guided, размер чанка равен 200

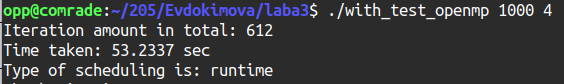


Рис. 13. Время работы на типе runtime, размер чанка равен 200

# Приложение 8. График зависимости времени от размера чанка

Рис. 1. График зависимости времени от размера чанка