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 OpenMP»

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 OpenMP.

ЗАДАНИЕ

1. Последовательную программу из лабораторной работы 1, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью OpenMP. Реализовать два варианта программы:

   - Вариант 1: для каждого распараллеливаемого цикла создается отдельная параллельная секция **#pragma omp parallel for**,
   - Вариант 2: создается одна параллельная секция **#pragma omp parallel**, охватывающая весь итерационный алгоритм

   Уделить внимание тому, чтобы при запуске программы на различном числе OpenMP-потоков решалась одна и та же задача (исходные данные заполнялись одинаковым образом).

2. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: от 1 до числа доступных в узле. 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. На основании полученных результатов сделать вывод о целесообразности использования первого или второго варианта программ

ОПИСАНИЕ РАБОТЫ

1. Были написаны варианты программы (Приложение 1, Приложение 2, Приложение 3), использующую векторные расширения и OpenMP для решения СЛАУ методом сопряженных градиентов.

2. Были выбраны размер матрицы и точность, при которых время работы программы занимало примерно 30 секунд.

3. Было исследованы оптимальные параметры **#pragma omp schedule()** (Приложение 4). По полученным данным был сделан вывод что оба варианта работают наилучшим образом при выборе *guided* и размера куска равному размеру матрицы делённое на кол-во потоков. П размере куска меньше время работы увеличивается из-за того, что появляются дополнительные затраты на организацию работы параллельного кода. А при большем размере куска время увеличивается из-за того, что некоторые потоки бездействуют, ведь им не досталось куска.

4. Были проведены замеры времени работы программы на разном количестве потоков. По полученным данным был сделан график (Приложение 5). На графике мы можем заметить, что второй вариант программы работает примерно в 2 раза быстрее. Это происходит из-за того, что не тратится время на разделение на потоки и соединение обратно, так как есть только один параллельный участок, затрагивающий весь итерационный алгоритм, в отличии с первым вариантом, где происходит постоянное разделение на потоки перед каждым циклом.

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы были получены знания о разделении программы на потоки с использованием векторных расширений для более эффективного использования доступных ресурсов.
Проанализировав обе лабораторные работы можно сделать вывод, что на системах с общей памятью OpenMP эффективнее, чем MPI.

Приложение 1. *Код последовательной программы*

```
#include <cmath>
#include <chrono>
#include <iostream>
#include <fstream>
#include <immintrin.h>

using namespace std::chrono;

void matVecMul(const double *mat, const double *vec, int N, double *newVec) {
  for (int i = 0; i < N; i++) {
    __m128d vA = _mm_setzero_pd();
    for (int j = 0; j < N; j += 2) {
      vA += _mm_loadu_pd(&mat[i * N + j]) * _mm_loadu_pd(&vec[j]);
    }
    newVec[i] = _mm_hadd_pd(vA, vA)[0];
  }
}

void mulByConst(const double *vec, double c, int size, double *newVec) {
  for (int i = 0; i < size; i += 2) {
    _mm_storeu_pd(&newVec[i], _mm_loadu_pd(&vec[i]) * _mm_set1_pd(c));
  }
}

void subVec(const double *vec1, const double *vec2, int size, double *newVec) {
  for (int i = 0; i < size; i += 2) {
    _mm_storeu_pd(&newVec[i], _mm_loadu_pd(&vec1[i]) - _mm_loadu_pd(&vec2[i]));
  }
}

void sumVec(const double *vec1, const double *vec2, int size, double *newVec) {
  for (int i = 0; i < size; i += 2) {
    _mm_storeu_pd(&newVec[i], _mm_loadu_pd(&vec1[i]) + _mm_loadu_pd(&vec2[i]));
  }
}

double dotProduct(const double *vec1, const double *vec2, int size) {
  __m128d vA = _mm_setzero_pd();
  for (int i = 0; i < size; i += 2) {
    vA += _mm_loadu_pd(&vec1[i]) * _mm_loadu_pd(&vec2[i]);
  }
  return _mm_hadd_pd(vA, vA)[0];
}

void printMat(double *mat, int rows, int columns, std::ostream &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-007;

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

double res = 1;
double prevRes = 1;
bool diverge = false;
int divergeCount = 0;
int rightAnswerRepeat = 0;
int iterCount = 1;
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 += 2) {
        _mm_storeu_pd(&z[i], _mm_loadu_pd(&r[i]));
    }
    /// alpha
    matVecMul(A, z, N, Atmp);
    dotRR = dotProduct(r, r, N);
    alpha = dotRR / 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) / dotRR;
    /// zn+1 = rn+1 + beta * zn
    mulByConst(z, beta, N, betaZ);
    sumVec(rNext, betaZ, 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 += 2) {
```

```
            _mm_storeu_pd(&prevSolution[i], _mm_loadu_pd(&solution[i]));
            _mm_storeu_pd(&r[i], _mm_loadu_pd(&rNext[i]));
            _mm_storeu_pd(&z[i], _mm_loadu_pd(&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]);

    const std::string &fileName = argv[2];
    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;
    std::cout << "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 <immintrin.h>

#pragma omp declare reduction(sseSum: __m128d: omp_out += omp_in) initializer (omp_priv = _mm_setzero_pd())

#define TYPE guided
#define CHUNK chunkSize
#define BASE_CLAUSE default(none) num_threads(threadCount) schedule(TYPE, CHUNK)
int matrixSize = 1;
int threadCount = 1;
int chunkSize = 1;

using namespace std::chrono;

void matVecMul(const double *mat, const double *vec, int N, double *newVec) {
#pragma omp parallel for shared(mat, vec, N, newVec, chunkSize) BASE_CLAUSE
  for (int i = 0; i < N; i++) {
    __m128d vA = _mm_setzero_pd();
    for (int j = 0; j < N; j += 2) {
      vA += _mm_loadu_pd(&mat[i * N + j]) * _mm_loadu_pd(&vec[j]);
    }
    newVec[i] = _mm_hadd_pd(vA, vA)[0];
  }
}

void mulByConst(const double *vec, double c, int size, double *newVec) {
#pragma omp parallel for shared(c, vec, size, newVec, chunkSize) BASE_CLAUSE
  for (int i = 0; i < size; i += 2) {
    _mm_storeu_pd(&newVec[i], _mm_loadu_pd(&vec[i]) * _mm_set1_pd(c));
  }
}

void subVec(const double *vec1, const double *vec2, int size, double *newVec) {
#pragma omp parallel for shared(vec1, vec2, size, newVec, chunkSize) BASE_CLAUSE
  for (int i = 0; i < size; i += 2) {
    _mm_storeu_pd(&newVec[i], _mm_loadu_pd(&vec1[i]) - _mm_loadu_pd(&vec2[i]));
  }
}

void sumVec(const double *vec1, const double *vec2, int size, double *newVec) {
#pragma omp parallel for shared(vec1, vec2, size, newVec, chunkSize) BASE_CLAUSE
  for (int i = 0; i < size; i += 2) {
    _mm_storeu_pd(&newVec[i], _mm_loadu_pd(&vec1[i]) + _mm_loadu_pd(&vec2[i]));
  }
}

double dotProduct(const double *vec1, const double *vec2, int size) {
  __m128d vA = _mm_setzero_pd();
#pragma parallel omp for shared(vec1, vec2, size, sum, chunkSize) reduction(sseSum:sum) BASE_CLAUSE
  for (int i = 0; i < size; i++) {
    vA += _mm_loadu_pd(&vec1[i]) * _mm_loadu_pd(&vec2[i]);
  }
  return _mm_hadd_pd(vA, vA)[0];
}

void printMat(double *mat, int rows, int columns, std::ostream &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 N, std::ostream &stream) {
  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-007;

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

  double res = 1;
  double prevRes = 1;
  bool diverge = false;
  int divergeCount = 0;
  int rightAnswerRepeat = 0;
  int iterCount = 1;
  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
#pragma omp parallel for shared(z, r, N, chunkSize) BASE_CLAUSE
    for (int i = 0; i < N; i += 2) {
      _mm_storeu_pd(&z[i], _mm_loadu_pd(&r[i]));
    }
    /// alpha
    matVecMul(A, z, N, Atmp);
    dotRR = dotProduct(r, r, N);
    alpha = dotRR / 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) / dotRR;
    /// zn+1 = rn+1 + beta * zn
    mulByConst(z, beta, N, betaZ);
    sumVec(rNext, betaZ, 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;
#pragma omp parallel for shared(solution, prevSolution, r, rNext, z, zNext, N, chunkSize) BASE_CLAUSE
    for (long i = 0; i < N; i += 2) {
      _mm_storeu_pd(&prevSolution[i], _mm_loadu_pd(&solution[i]));
      _mm_storeu_pd(&r[i], _mm_loadu_pd(&rNext[i]));
      _mm_storeu_pd(&z[i], _mm_loadu_pd(&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 != 5) {
    std::cout << "Program needs 2 arguments: size, threadCount, filename, chunkPercent" << std::endl;
    return 0;
  }
  int N = atoi(argv[1]);
  matrixSize = N;
  threadCount = atoi(argv[2]);
  chunkSize = matrixSize / threadCount * atof(argv[4]);
  std::cout << "chunkSize: " << chunkSize << std::endl;

  const std::string &fileName = argv[3];
  std::ofstream fileStream(fileName);
  if (!fileStream) {
    std::cout << "error with output file" << std::endl;
    return 0;
  }

  fileStream << "Matrix size: " << N << " thread num: " << threadCount << std::endl;

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

```
#pragma omp parallel for shared(N, A, u, chunkSize) BASE_CLAUSE
  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, fileStream);
  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;
    std::cout << "Time: " << duration.count() / double(1000000000) << "sec" << std::endl;
  } else {
    fileStream << "Does not converge" << std::endl;
  }

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

Приложение 3. *Код программы вар.2*

```
#include <cmath>
#include <chrono>
#include <iostream>
#include <fstream>
#include <immintrin.h>

#pragma omp declare reduction(sseSum: __m128d: omp_out += omp_in) initializer (omp_priv = _mm_setzero_pd())

#define TYPE guided
#define CHUNK chunkSize
int matrixSize = 1;
int threadCount = 1;
int chunkSize = 1;

using namespace std::chrono;

double dotProduct(const double *vec1, const double *vec2, int size) {
  __m128d vA = _mm_setzero_pd();
  for (int i = 0; i < size; i += 2) {
    vA += _mm_loadu_pd(&vec1[i]) * _mm_loadu_pd(&vec2[i]);
  }
  return _mm_hadd_pd(vA, vA)[0];
}

void printMat(double *mat, int rows, int columns, std::ostream &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-007;

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

  __m128d vA = _mm_setzero_pd();
  __m128d vB = _mm_setzero_pd();

  double res = 1;
  double prevRes = 1;
  bool diverge = false;
  int divergeCount = 0;
```

```
  int rightAnswerRepeat = 0;
  int iterCount = 1;
#pragma omp parallel num_threads(threadCount) firstprivate(rightAnswerRepeat, divergeCount, diverge)
  while (res > EPSILON || rightAnswerRepeat < 5) {
#pragma omp single
    {
      if (res < EPSILON) {
        ++rightAnswerRepeat;
      } else {
        rightAnswerRepeat = 0;
      }
    }

    /// rn = b - A * xn
#pragma omp for schedule(TYPE, CHUNK)
    for (int i = 0; i < N; i++) {
      __m128d sum = _mm_setzero_pd();
      for (int j = 0; j < N; j += 2) {
        sum += _mm_loadu_pd(&A[i * N + j]) * _mm_loadu_pd(&prevSolution[j]);
      }
      Atmp[i] = _mm_hadd_pd(sum, sum)[0];
    }
#pragma omp for schedule(TYPE, CHUNK)
    for (int i = 0; i < N; i += 2) {
      _mm_storeu_pd(&r[i], _mm_loadu_pd(&b[i]) - _mm_loadu_pd(&Atmp[i]));
      /// zn = rn
      _mm_storeu_pd(&z[i], _mm_loadu_pd(&r[i]));
    }
    /// alpha
#pragma omp for schedule(TYPE, CHUNK)
    for (int i = 0; i < N; i++) {
      __m128d sum = _mm_setzero_pd();
      for (int j = 0; j < N; j += 2) {
        sum += _mm_loadu_pd(&A[i * N + j]) * _mm_loadu_pd(&z[j]);
      }
      Atmp[i] = _mm_hadd_pd(sum, sum)[0];
    }
#pragma omp single
    {
      vA = _mm_setzero_pd();
    }
#pragma omp for schedule(TYPE, CHUNK) reduction(sseSum: vA, vB)
    for (int i = 0; i < N; i += 2) {
      vA += _mm_loadu_pd(&r[i]) * _mm_loadu_pd(&r[i]);
      vB += _mm_loadu_pd(&Atmp[i]) * _mm_loadu_pd(&z[i]);
    }
#pragma omp single
    {
      dotRR = _mm_hadd_pd(vA, vA)[0];
      alpha = dotRR / _mm_hadd_pd(vB, vB)[0];
    }
    /// xn+1 = xn + alpha * zn
#pragma omp for schedule(TYPE, CHUNK)
    for (int i = 0; i < N; i += 2) {
      _mm_storeu_pd(&alphaZ[i], _mm_loadu_pd(&z[i]) * _mm_set1_pd(alpha));
      _mm_storeu_pd(&solution[i], _mm_loadu_pd(&prevSolution[i]) + _mm_loadu_pd(&alphaZ[i]));
    }
    /// rn+1 = rn - alpha * A * zn
#pragma omp for schedule(TYPE, CHUNK)
    for (int i = 0; i < N; i++) {
      __m128d sum = _mm_setzero_pd();
      for (int j = 0; j < N; j += 2) {
        sum += _mm_loadu_pd(&A[i * N + j]) * _mm_loadu_pd(&alphaZ[j]);
```

```
        }
        Atmp[i] = _mm_hadd_pd(sum, sum)[0];
    }
#pragma omp single
    {
        vA = _mm_setzero_pd();
    }
#pragma omp for schedule(TYPE, CHUNK) reduction(sseSum: vA)
    for (int i = 0; i < N; i += 2) {
        _mm_storeu_pd(&rNext[i], _mm_loadu_pd(&r[i]) - _mm_loadu_pd(&Atmp[i]));
        vA += _mm_loadu_pd(&rNext[i]) * _mm_loadu_pd(&rNext[i]);
    }
    /// beta
#pragma omp single
    {
        beta = _mm_hadd_pd(vA, vA)[0] / dotRR;
    }
    /// zn+1 = rn+1 + beta * zn
#pragma omp for schedule(TYPE, CHUNK)
    for (int i = 0; i < N; i += 2) {
        _mm_storeu_pd(&betaZ[i], _mm_loadu_pd(&z[i]) * _mm_set1_pd(beta));
        _mm_storeu_pd(&zNext[i], _mm_loadu_pd(&rNext[i]) + _mm_loadu_pd(&betaZ[i]));
    }
#pragma omp single
    {
        res = sqrt(_mm_hadd_pd(vA, vA)[0]) / normb;
    }
    if (prevRes < res || res == INFINITY || res == NAN) {
#pragma omp single
        {
            ++divergeCount;
        }
        if (divergeCount > 10 || res == INFINITY || res == NAN) {
            diverge = true;
            break;
        }
    } else {
#pragma omp single
        {
            divergeCount = 0;
        }
    }
#pragma omp single
    {
        prevRes = res;
        ++iterCount;
    }
#pragma omp for schedule(TYPE, CHUNK)
    for (long i = 0; i < N; i += 2) {
        _mm_storeu_pd(&prevSolution[i], _mm_loadu_pd(&solution[i]));
        _mm_storeu_pd(&r[i], _mm_loadu_pd(&rNext[i]));
        _mm_storeu_pd(&z[i], _mm_loadu_pd(&zNext[i]));
    }
  }
  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 != 5) {
        std::cout << "Program needs 2 arguments: size, threadCount, filename, chunkPercent" << std::endl;
        return 0;
    }
    int N = atoi(argv[1]);
    matrixSize = N;
    threadCount = atoi(argv[2]);
    chunkSize = matrixSize / threadCount * atof(argv[4]);
    std::cout << "chunkSize: " << chunkSize << std::endl;

    const std::string &fileName = argv[3];
    std::ofstream fileStream(fileName);
    if (!fileStream) {
        std::cout << "error with output file" << std::endl;
        return 0;
    }

    fileStream << "Matrix size: " << N << " thread num: " << threadCount << 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));
    }
    for (int i = 0; i < N; i++) {
        __m128d vA = _mm_setzero_pd();
        for (int j = 0; j < N; j += 2) {
            vA += _mm_loadu_pd(&A[i * N + j]) * _mm_loadu_pd(&u[j]);
        }
        b[i] = _mm_hadd_pd(vA, vA)[0];
    }

    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;
```

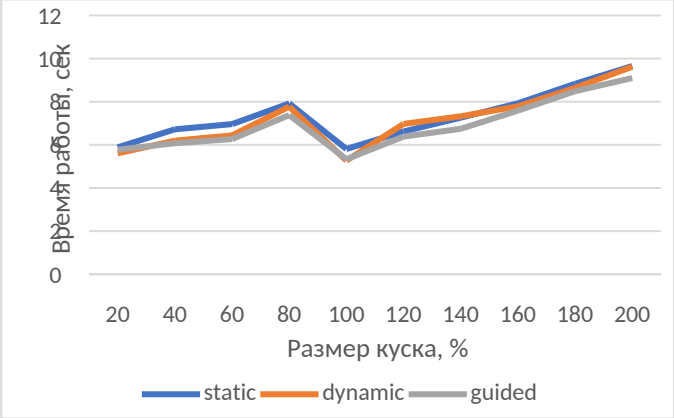
```
    std::cout << "Time: " << duration.count() / double(1000000000) << "sec" << std::endl;
  } else {
    fileStream << "Does not converge" << std::endl;
  }

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

Приложение 4. *Исследования* **#pragma omp schedule()**

За 100% размер куска берется N/кол-во потоков, где N - размер матрицы.

Вар.1:

	static	*dynamic*	*guided*
20	5,88306	5,60548	5,77482
40	6,7197	6,19513	6,0746
60	6,96117	6,43523	6,26761
80	7,91898	7,76932	7,37753
100	5,80743	5,2682	5,33303
120	6,61919	6,97014	6,38288
140	7,25941	7,32339	6,74216
160	7,92514	7,78371	7,58574
180	8,82994	8,64281	8,48732
200	9,65195	9,61938	9,09597



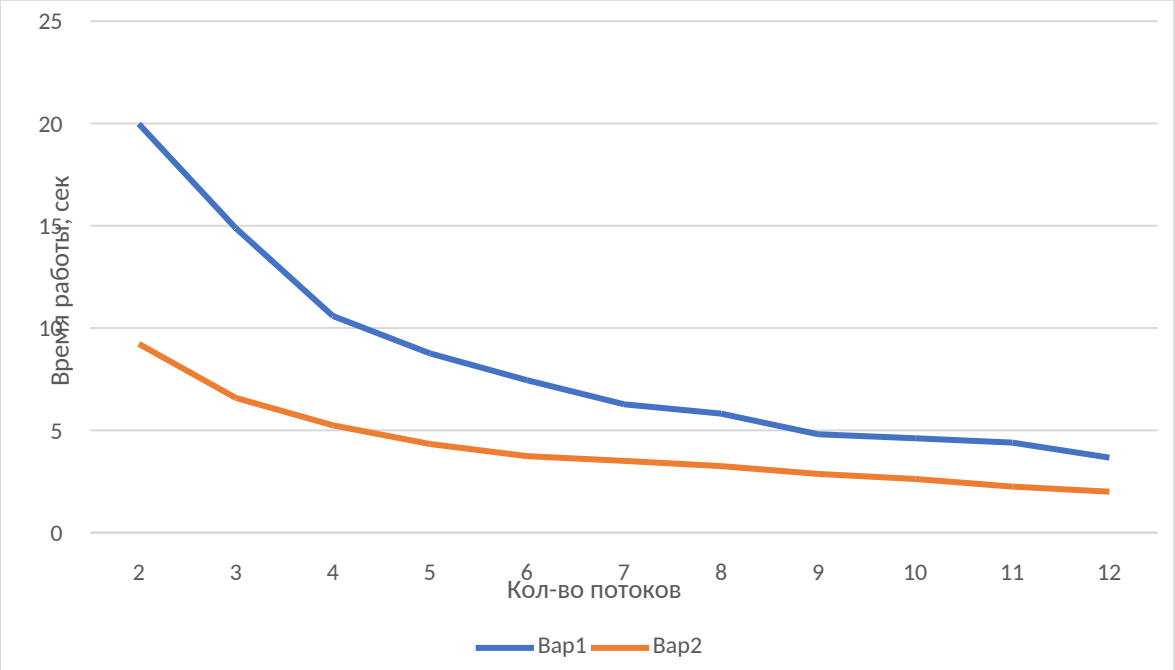
Вар.2:

	static	*dynamic*	*guided*
20	2,79103	2,79734	2,69942
40	3,06534	3,15653	2,94892
60	3,31917	3,39407	3,25673
80	3,91417	3,9814	3,49331
100	2,66425	2,69003	2,59734
120	3,18386	3,32867	3,21088
140	3,67761	3,75452	3,69579
160	4,37151	4,43063	4,30558
180	4,67385	4,68074	4,51426
200	4,96864	4,70993	4,87286

Приложение 5. *Замеры времени*

Время работы, сек

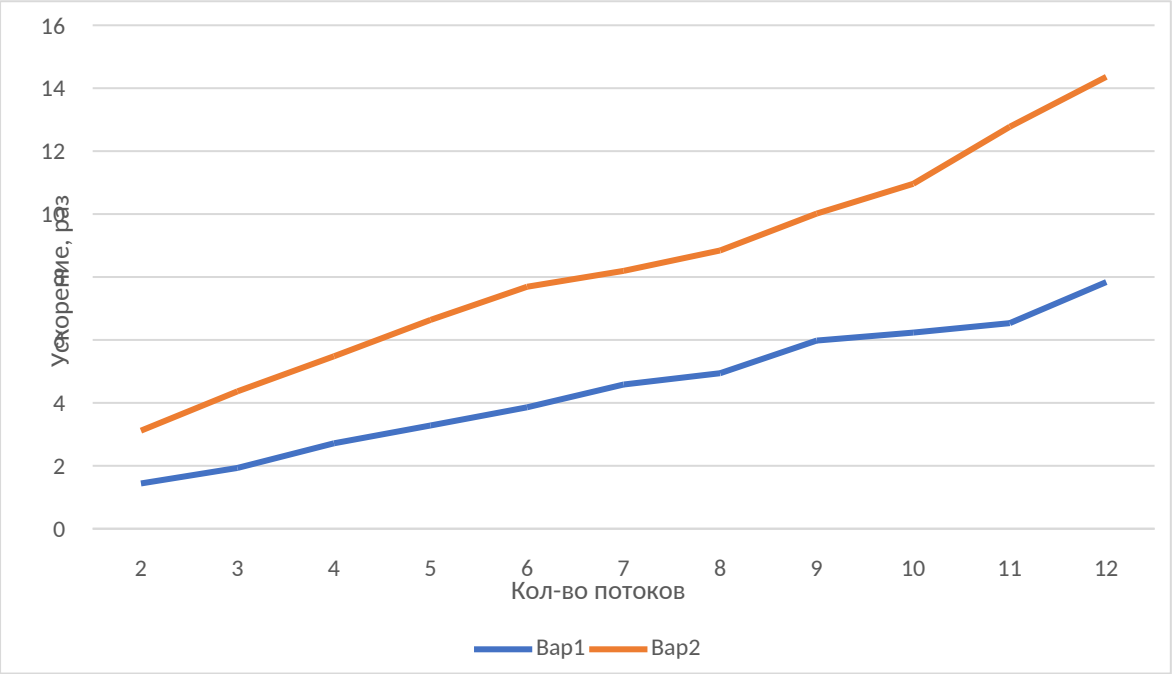
	2	3	4	5	6	7	8	9	10	11	12
Вар 1	19,98	14,88	10,59	8,76	7,45	6,27	5,82	4,81	4,61	4,40	3,67
Вар 2	9,23	6,59	5,25	4,33	3,74	3,51	3,25	2,87	2,62	2,25	2,00



Время работы последовательной программы: 28,7534с

Ускорение, раз

	2	3	4	5	6	7	8	9	10	11	12
Вар 1	1,44	1,93	2,72	3,28	3,86	4,58	4,94	5,98	6,23	6,53	7,84
Вар 2	3,12	4,37	5,48	6,64	7,69	8,19	8,84	10,02	10,96	12,77	14,36



Эффективность на поток, %

	2	3	4	5	6	7	8	9	10	11	12
Вар 1	0,72	0,64	0,68	0,66	0,64	0,65	0,62	0,66	0,62	0,59	0,65
Вар 2	1,56	1,46	1,37	1,33	1,28	1,17	1,11	1,11	1,10	1,16	1,20