Student Name: Галактионов Кирилл
Student ID: st067889

Современные технологии программирования в научных исследованиях
Лабораторная работа 1

1 Постановка задачи

Задача заключалась в оптимизации алгоритма нахождения максимального по модулю собственного числа заданной, вещественной, симметричной матрицы $\mathbf{A}$, при помощи метода прямых итераций. При этом, требовалось исследовать зависимость масштабируемости параллельной версии программы от ее вычислительной трудоемкости и проверить закон Амдала, изучив зависимости ускорения от числа потоков.

2 Описание тестового стенда

Вычисления проводились на выданном вычислительном узле, на котором доступно максимум 12 вычислительных потоков. Кроме того, проводилась проверка данных на персональном компьютере, с 16 потоками (8 ядер, по 2 логических ядра). Вычислительный узел:

- Операционная система: Ubuntu 20.04.4 LTS
- Процессор: Intel(R) Xeon(R) E-2136 CPU @ 3.30GHz, 6 ядер, 2 потока на ядро
- Доступная оперативная память: 62-63 ГБ
- Компилятор: gcc 10.2.0 (флаг -fopenmp)

Персональный компьютер:

- Операционная система: Windows 10 10.0.19044 N/A Build 19044
- Процессор: AMD64 Family 25 Model 80 Stepping 0 AuthenticAMD 3201 Mhz, 8 ядер, 2 потока на ядро
- Доступная оперативная память: 14 188 МБ
- Компилятор: Intel C++ Compiler 19.2 (флаг /Qopenmp)

Время работы алгоритмов вычислялось при помощи команды omp_get_wtime(). Проводилось по три серии экспериментов, затем полученные значения времен усреднялись.

3 Описание алгоритма решения задачи

Алгоритм прямых итераций, поиска максимального по модую собственного числа матрицы **A** заключается в итерационном применении следующей формулы: $v_{k+1} = \frac{Av_k}{||Av_k||}$, при которой последовательность векторов v_k сходится к собственному вектору, отвечающему максимальному по модулю собственному числу, а собственным числом является норма $||Av_k||$.

Так как итерации необходимо выполнять последовательно, то их нельзя запускать параллельно на разных потоках. Поэтому было принято решение реализовывать параллельное выполнение наиболее ресурсоёмкой операции - умножения матрицы на вектор. Реализовывалось параллельное матрично-векторное умножение при помощи технологии OpenMP и языка C++. Были исследован способ реализации параллельного умножения матрицы на вектор, который представляет собой распараллеливание по строкам. То есть элементы вектора результата вычисляются параллельно (внешний for). Код функции представлен ниже.

```
// Matrix-vector multiplication with OMP
vector<double> matrixVectorMultOMPSimplier(const vector<vector<double>>& mat, const
    vector<double>& vec, const int num_of_threads){
  int n = vec.size();
  vector<double> res(n);

// Divide outer for-loop to given number of threads
  omp_set_dynamic(0);
#pragma omp parallel for num_threads(num_of_threads)
  for (int i = 0; i < n; i++) {
    res[i] = 0.0;
    for (int j = 0; j < n; j++) {
      res[i] += mat[i][j] * vec[j];
    }
  }
  return res;
}
```

4 Программный код

Полный код программы, используемый для решения задачи в разных условиях представлен в Приложении 1 в разделе 7.

5 Результаты измерений

На графиках приведена зависимость ускорения от параметров задачи. Ускорение считалось как $K(N) = \frac{t_1}{t_N}$, где t_1 - время затрачиваемое на работу алгоритма в последовательном режиме, t_N - время работы алгоритма, при использовании N потоков.

5.1 Изучение масштабируемости

Для изучения масштабируемости строились графики зависимости ускорения от размерности задачи (размерности матрицы (n)) для разных значений количества потоков, и отвечающих вычислению на предоставленном вычислительном узле и на ПК.

5.1.1 Вычисления на выделенном узле

Рисунки 1 - 4.

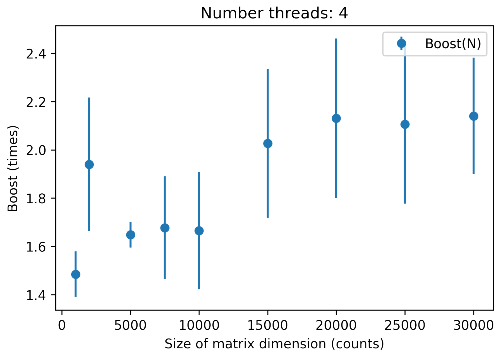

Рис. 1: 4 параллельных потока. Вычисления на выделенном узле

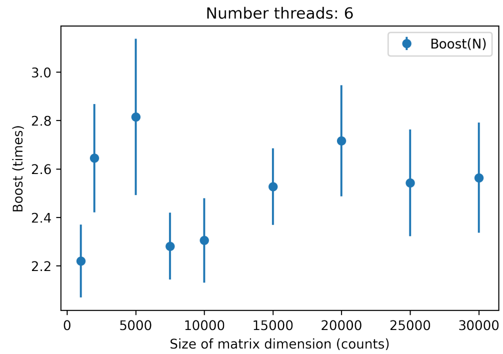

Рис. 2: 6 параллельных потоков. Вычисления на выделенном узле

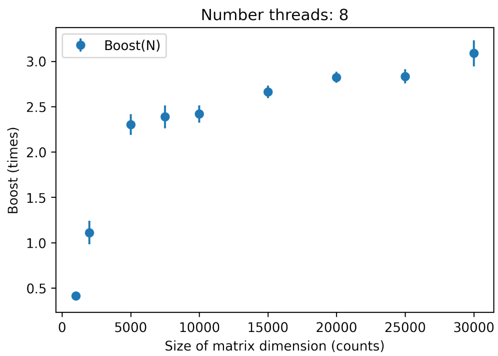

Рис. 3: 8 параллельных потоков. Вычисления на выделенном узле

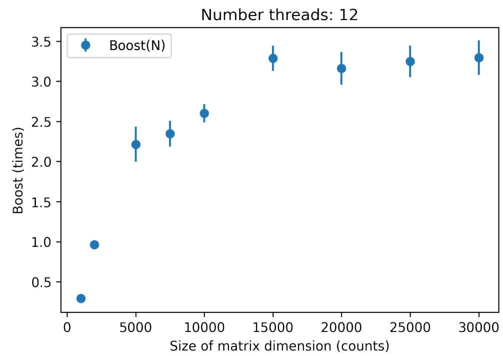

Рис. 4: 12 параллельных потоков. Вычисления на выделенном узле

5.1.2 Вычисления на ПК

Рисунки 5 - 8.

5.2 Проверка закона Амдала

Для проверки закона Амдала:

$$K(N) = \frac{N}{S * N + (1 - S)}$$

где, K - достигаемое ускорение, N - число потоков, S - доля последовательной части алгорима, который описывает как должно расти максимально возможное ускорение с ростом числа потоков, исследовалась зависимость ускорения от числа потоков, при фиксированной размерности задачи. Полученные зависимости представлены на рисунках 9 - 16.

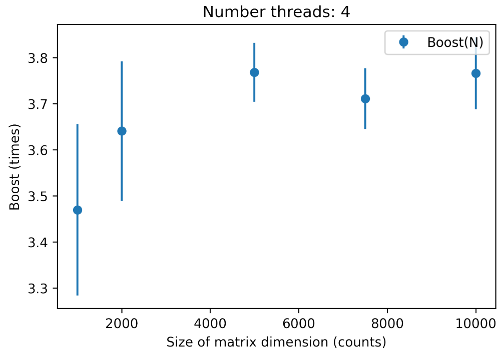

Рис. 5: 4 параллельных потока. Вычисления на ПК

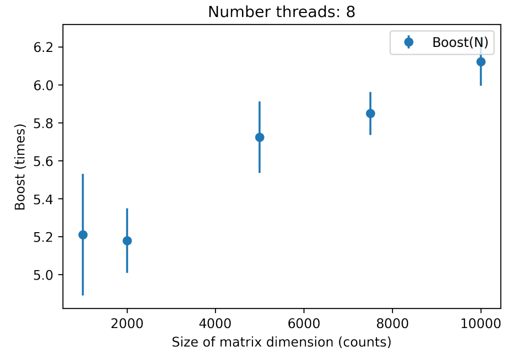

Рис. 6: 8 параллельных потоков. Вычисления на ПК

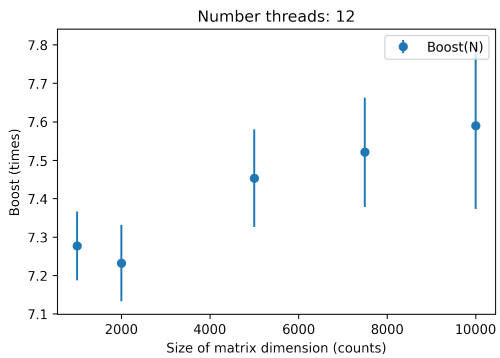

Рис. 7: 12 параллельных потоков. Вычисления на ПК

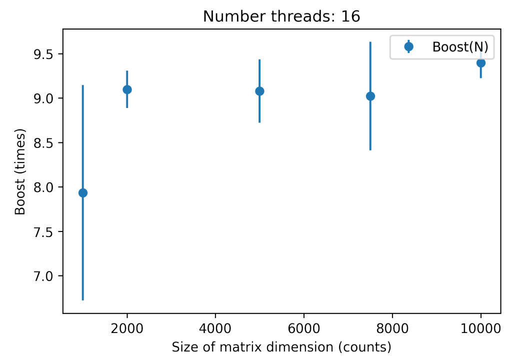

Рис. 8: 16 параллельных потоков. Вычисления на ПК

5.2.1 Вычисления на выделенном узле

Рисунки 9 - 12.

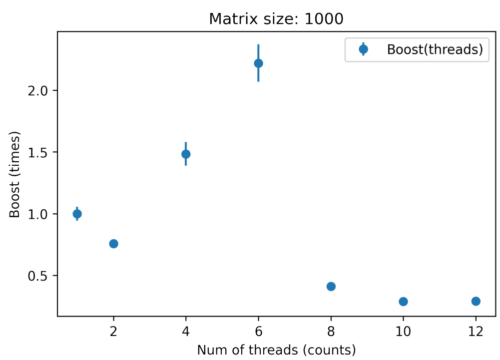

Рис. 9: Матрица размером в 1000. Вычисления на выделенном узле

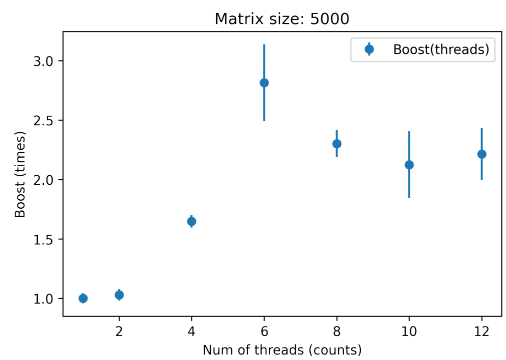

Рис. 10: Матрица размером в 5000. Вычисления на выделенном узле

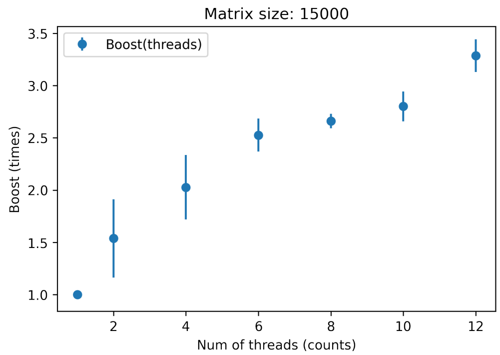

Рис. 11: Матрица размером в 15000. Вычисления на выделенном узле

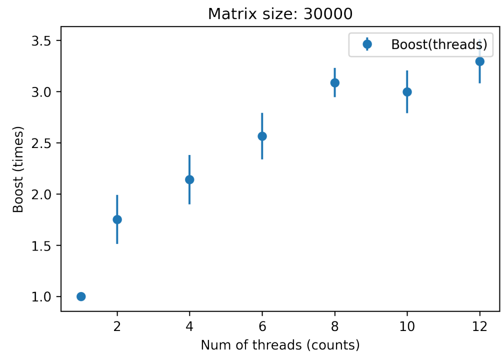

Рис. 12: Матрица размером в 30000. Вычисления на выделенном узле

5.2.2 Вычисления на ПК

Рисунки 13 - 16.

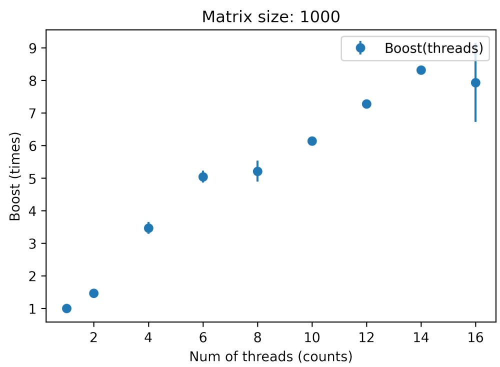

Рис. 13: Матрица размером в 1000. Вычисления на ПК

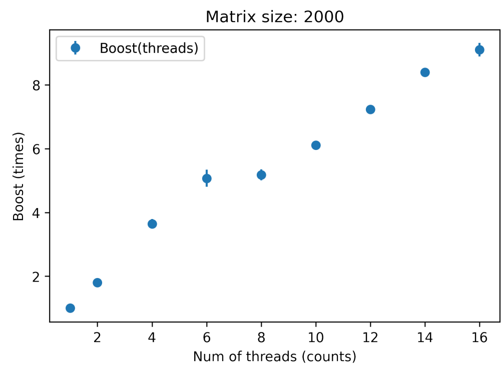

Рис. 14: Матрица размером в 2000. Вычисления на ПК

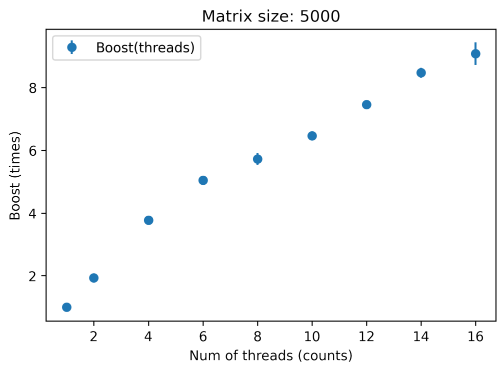

Рис. 15: Матрица размером в 5000. Вычисления на ПК

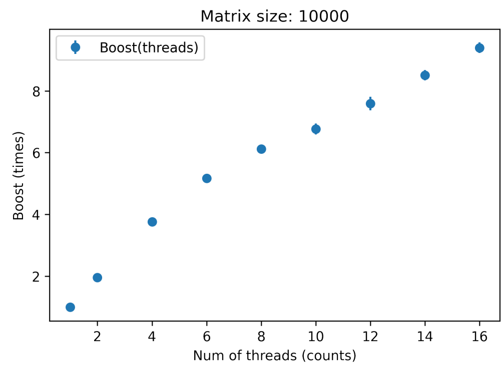

Рис. 16: Матрица размером в 10000. Вычисления на ПК

6 Обсуждение результатов и вывод

Были успешно проведены несколько серий экспериментов по оценке масштабируемости алгорима прямых итераций. По рисункам 1 - 4 можно заметить, что в области матриц малой размерности (до линейного размера в 10000) заметно увеличение ускорения при увеличении размера матрицы. Однако, при больших размерностях ускорение становится практически постоянным. При этом схожий характер имеют зависимости при 4, 6, 8, и 12 потоках. При оценке на ПК, для матриц больших размерностей данных получить не удалось, однако в интервале от 1000 до 10000 наблюдается слабый рост ускорения.

На рисунках 9 - 12 зависимости ускорения от числа потоков, наблюдается рост ускорения при увеличении числа потоков, но рост нелинейный, что не противоречит закону Амдала. Причем, при матрице малой размерности (Рис 9) в первом потоке наблюдается падение ускорения при увеличении числа потоков, что объясняется тем, что решающий вклад играет пересылка данных а не параллельные вычислительные операции. Кроме того, на первых двух графиках наблюдается скачок падения ускорения при переходе от 6 потоков к 8, можно предположить, что он вызван структурой процессора вычислительного узла. При проверке на ПК, получены схожие результаты.

Подводя итог, были изучены характеристики параллельного алгоритма вычисления максимального по модулю собственного числа матрицы, методом прямых итераций. Успешно проверен закон Амдала, и исследована масштабируемость параллельной программы.

7 Приложение 1. Код программы

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>
#include <cmath>
#include <fstream>
#include <iomanip>
#include "omp.h"

using namespace std;

vector<double> matrixVectorMult(const vector<vector<double>>& mat, const vector<double>&
    vec);
vector<double> matrixVectorMultOMP(const vector<vector<double>>& mat, const vector<double
    >& vec, const int num_of_threads);
vector<double> matrixVectorMultOMPSimplier(const vector<vector<double>>& mat, const
    vector<double>& vec, const int num_of_threads);
double vectorNorm(const vector<double>& vec);
void vectorNormalize(vector<double>& vec, const double vecNorm);



int main() {
  std::ofstream out;          // output stream
  out.open("res_my_3.txt");     //open file for output
  //const int n = 100;
  double tolerance = 1e-8;      // set precision for iteration method
  srand(static_cast <unsigned> (time(0)));
```

```
  int max_threads;

  // Print to console and file info about max available threads
  max_threads = omp_get_max_threads();
  out << "Max number of threads is " << max_threads << endl;
  out << endl;
  cout << "Max number of threads is " << max_threads << endl;
  cout << endl;

  // Define threads amounts for which the calculations will be held (with the step of 2)
  vector<int> threads(0);
  int num = 2, step = 2;
  while (num <= max_threads) {
    threads.push_back(num);
    num += step;
  }
  int numberOfExamples = threads.size();

  // Define matrix sizes
  const int ns[5] = { 1000, 2000, 5000, 7500, 10000 };
  // For every matrix size perform full calculations
  for (int n : ns) {

    vector<double> results1(0), results2(0);
    double resNoOmp;

    // Create and fill symmetric real Matrix M(n,n) with random numbers
    vector<vector<double>> A(n);
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        double value = 0.0;
        A[i].push_back(value);
      }
    }
    for (int i = 0; i < n; i++) {
      for (int j = i; j < n; j++) {
        double value = static_cast <double> (rand()) / static_cast <double> (RAND_MAX);
        A[i][j] = value;
        A[j][i] = value;
      }
    }



    // Preparation for iterations
    vector<double> eigVector(n), residueVector(n), nextVector(n);
    double residueValue, eigValue, start_time, total_time, referenceValue;

    // Calculations without OMP
    // Set initial vector and eigenvalue
    start_time = omp_get_wtime();
    residueValue = 100.0;
    for (int i = 0; i < n; i++) {
```

```
      eigVector[i] = 1.0;
    }
    eigValue = vectorNorm(eigVector);
    vectorNormalize(eigVector, eigValue);
    nextVector = matrixVectorMult(A, eigVector);

    // While reidue: ||Av - lv||, is greater than given precision, continue iterations
    while (residueValue > tolerance) {
      eigVector = nextVector;
      eigValue = vectorNorm(eigVector);       // Don't forget to normalize vector
      vectorNormalize(eigVector, eigValue);

      nextVector = matrixVectorMult(A, eigVector);  // Next vector is given by matrix-vector multiplication

      // Calculate residue
      for (int i = 0; i < n; i++) {
        residueVector[i] = nextVector[i] - eigValue * eigVector[i];
      }
      residueValue = vectorNorm(residueVector);
    }
    total_time = omp_get_wtime() - start_time;
    //cout << "No OMP; Value: " << eigValue << "; Time taken: " << total_time << endl;
    //out << "No OMP; Value: " << eigValue << "; Time taken: " << total_time << endl;
    resNoOmp = total_time;

    // Set calculated value as reference, for future evaluations
    referenceValue = eigValue;

    cout << endl;
    out << endl;
    // ////////////////////////////////////////////////////////////////////////////////////////////////
    // Calculations with OMP
    // Same calculations, matrix-vector multiplication with OMP
    for (int thread_num : threads) {
      start_time = omp_get_wtime();
      residueValue = 100.0;
      for (int i = 0; i < n; i++) {
        eigVector[i] = 1.0;
      }
      eigValue = vectorNorm(eigVector);
      vectorNormalize(eigVector, eigValue);
      nextVector = matrixVectorMultOMPSimplier(A, eigVector, thread_num);

      // Main computational loop
      while (residueValue > tolerance) {
        eigVector = nextVector;
        eigValue = vectorNorm(eigVector);
        vectorNormalize(eigVector, eigValue);

        nextVector = matrixVectorMultOMPSimplier(A, eigVector, thread_num);

```

```
        for (int i = 0; i < n; i++) {
          residueVector[i] = nextVector[i] - eigValue * eigVector[i];
        }
        residueValue = vectorNorm(residueVector);
      }
      total_time = omp_get_wtime() - start_time;
      // Check that result is the same that was received without OMP
      if (abs(eigValue - referenceValue) > 1e-4) {
        cout << "NOT CORRECT CALCULATION" << endl;
      }
      // cout << "OMP2  ; Value: " << eigValue << "; Time taken: " << total_time << "; Threads: " << thread_num << endl;
      // out << "OMP2  ; Value: " << eigValue << "; Time taken: " << total_time << "; Threads: " << thread_num << endl;
      results2.push_back(total_time);
    }


    // Print results to console and output file
    cout << endl;
    cout << "n = " << n << endl;
    cout << "Threads   Time" << endl;
    cout << " 1       " << setprecision(4) << setw(7) << fixed << resNoOmp << endl;
    cout << "Threads   TimeOMP1  TimeOMP2" << endl;
    out << endl;
    out << "n = " << n << endl;
    out << "Threads   Time" << endl;
    out << " 1       " << setprecision(4) << setw(7) << fixed << resNoOmp << endl;
    out << "Threads   TimeOMP1  TimeOMP2" << endl;
    for (int i = 0; i < numberOfExamples; i++) {
      cout << setw(2) << threads[i] << "       " << setprecision(4) << setw(7) << fixed << results2[i] << endl;
      out << setw(2) << threads[i] << "       " << setprecision(4) << setw(7) << fixed << results2[i] << endl;
    }

  }


  return 0;
}


// Matrix-vector multiplication without OMP
vector<double> matrixVectorMult(const vector<vector<double>>& mat, const vector<double>& vec){
  int n = vec.size();
  vector<double> res(n);
  for (int i = 0; i < n; i++) {
    res[i] = 0.0;
    for (int j = 0; j < n; j++) {
      res[i] += mat[i][j] * vec[j];
    }
  }
```

```
  return res;
}

// Matrix-vector multiplication with OMP
vector<double> matrixVectorMultOMPSimplier(const vector<vector<double>>& mat, const
    vector<double>& vec, const int num_of_threads){
  int n = vec.size();
  vector<double> res(n);

// Divide outer for-loop to given number of threads
  omp_set_dynamic(0);
#pragma omp parallel for num_threads(num_of_threads)
  for (int i = 0; i < n; i++) {
    res[i] = 0.0;
    for (int j = 0; j < n; j++) {
      res[i] += mat[i][j] * vec[j];
    }
  }
  return res;
}

// L2 vector norm
double vectorNorm(const vector<double>& vec){
  double res = 0.0;
  for (int i = 0; i < vec.size(); i++) {
    res += vec[i] * vec[i];
  }

  return sqrt(res);
}

// Vector normalization to 1
void vectorNormalize(vector<double>& vec, const double vecNorm){
  for (int i = 0; i < vec.size(); i++) {
    vec[i] = vec[i] / vecNorm;
  }
}
```

8 Приложение 2. Таблицы результатов для вычислений на кластере

n - размерность матрицы, Threads - число потоков, Time - время работы алгоритма (мс).

```
Max number of threads is 12

n = 1000
Threads   Time_1    Time_2    Time_3
 1        0.0526    0.0538    0.0576
 2        0.0723    0.0734    0.0704
 4        0.0382    0.0381    0.0342
 6        0.0255    0.0257    0.0227
 8        0.1217    0.1295    0.1470
```

```
10        0.1979    0.1797    0.1855
12        0.1840    0.1828    0.1912

n = 2000
 1        0.2106    0.2069    0.2314
 2        0.2860    0.1078    0.1905
 4        0.0998    0.1022    0.1326
 6        0.0821    0.0748    0.0885
 8        0.2117    0.2066    0.1654
10        0.2154    0.2191    0.2257
12        0.2178    0.2306    0.2245

n = 5000
 1        1.1023    1.1178    1.1814
 2        1.1455    1.1110    1.0492
 4        0.6909    0.6778    0.6951
 6        0.3719    0.3707    0.4658
 8        0.4985    0.4660    0.5123
10        0.6201    0.4514    0.5286
12        0.5751    0.4580    0.5023

n = 7500
 1        2.4906    2.4578    2.6490
 2        2.3043    2.2614    2.2782
 4        1.2624    1.7099    1.5586
 6        1.1029    1.1827    1.0456
 8        0.9990    1.1015    1.0806
10        1.2081    1.1148    1.0403
12        1.1303    1.1198    0.9873

n = 10000
 1        4.4407    4.3272    4.6926
 2        4.0143    3.6284    2.6855
 4        2.9703    2.9595    2.1537
 6        2.0665    2.0118    1.7632
 8        1.8985    1.8565    1.8085
10        1.9322    1.8401    1.7328
12        1.7339    1.7783    1.6623

n = 15000
 1       10.0128   10.2170   10.6535
 2        5.8960    5.2320    8.9620
 4        4.1437    5.0834    6.0088
 6        4.1425    3.7608    4.3183
 8        3.8771    3.8630    3.8645
10        3.4643    3.7000    3.8615
12        3.0557    3.0336    3.3096

n = 20000
 1       18.0420   18.5473   18.9409
 2       15.0620   12.8428   13.3728
 4        9.7462    9.5108    6.8052
 6        7.3587    6.0424    7.0403
 8        6.6270    6.4897    6.5536
```

```
10        6.2858    6.6623    6.3271
12        5.9459    5.3765    6.2354

n = 25000
 1       29.2715   27.9905   27.7045
 2       17.1810   14.5764   16.1248
 4       15.9824   10.9063   13.4554
 6       11.6402    9.8413   11.9413
 8        9.9832    9.8322   10.1685
10       10.1955   10.3921   10.5043
12        8.6035    9.3530    8.1943

n = 30000
 1       43.2645   42.5748   41.8035
 2       22.9841   21.0890   28.8155
 4       21.7409   16.7522   21.1472
 6       15.2117   18.6051   15.9730
 8       13.3111   14.6373   13.3967
10       12.8920   15.1859   14.5017
12       13.1735   13.7575   11.8024
```

9 Приложение 3. Таблицы результатов для вычислений на персональном компьютере

n - размерность матрицы, Threads - число потоков, Time - время работы алгоритма (мс).

```
Max number of threads is 16

n = 1000
Threads   Time_1    Time_2    Time_3
 1        0.3015    0.3045    0.2985
 2        0.2002    0.2104    0.2054
 4        0.0905    0.0898    0.0804
 6        0.0574    0.0626    0.0594
 8        0.0600    0.0529    0.0607
10        0.0491    0.0493    0.0490
12        0.0409    0.0418    0.0416
14        0.0362    0.0363    0.0362
16        0.0462    0.0341    0.0337

n = 2000
 1        1.1851    1.1987    1.1629
 2        0.6525    0.6546    0.6570
 4        0.3340    0.3065    0.3337
 6        0.2503    0.2245    0.2245
 8        0.2208    0.2375    0.2265
10        0.1914    0.1956    0.1932
12        0.1623    0.1646    0.1635
14        0.1399    0.1421    0.1406
16        0.1328    0.1304    0.1266

n = 5000
```

```
 1        6.3410    6.4369    6.1912
 2        3.2573    3.2783    3.2837
 4        1.6806    1.6653    1.6879
 6        1.2547    1.2564    1.2512
 8        1.0959    1.0707    1.1474
10        0.9692    0.9841    0.9818
12        0.8427    0.8476    0.8548
14        0.7362    0.7545    0.7487
16        0.7296    0.6904    0.6694

n = 7500
 1       14.2994   14.4509   13.8922
 2        7.2590    7.3007    7.2515
 4        3.8317    3.7991    3.8594
 6        2.7494    2.7472    2.7703
 8        2.3976    2.4382    2.4551
10        2.1236    2.1317    2.1656
12        1.8766    1.8795    1.9139
14        1.6584    1.7096    1.6784
16        1.4960    1.7213    1.5088

n = 10000
 1       25.4733   25.6224   24.6396
 2       12.8811   12.9290   12.8789
 4        6.6831    6.8082    6.6191
 6        4.8541    4.9594    4.8428
 8        4.0811    4.0970    4.1922
10        3.6275    3.7967    3.7693
12        3.2184    3.3863    3.3735
14        2.9335    3.0013    2.9623
16        2.6766    2.7106    2.6725
```