Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

По контрольной работе № 1

По курсу: «Анализ алгоритмов»

Тема: «Параллельные вычисления»

Студент: Ле Ни Куанг

Группа: ИУ7и-56Б

Преподаватель: Волкова Л. Л.

Строганов Ю. В.

Москва

2020

Оглавление

Введение

Параллельные вычисления - способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно).

Целью работы: изучение параллельных вычисления, работая с разреженными матричными структурами.

Задача: Реализовать в параллельном режиме поиск строки с максимальной суммой элементов в симметричной разреженной матрице в схеме Дженнингса (см. ТСД). Матрицу не распаковывать.

1 Аналитический раздел

1.1 Симметричная разреженная матрица в схеме Дженнингса

Дженнингс [Jennings, 1966] предложил эффективную схему хранения симметричных матриц, и она вследствие своей простоты стала весьма популярной. Называется она профильной схемой, или схемой переменной ленты. Для каждой строки i симметричной матрицы A положим

$\beta_i = i - j_{min}(i)$

где $j_{min}(i)$ - минимальный столбцовый индекс строки i, для которого $a_{ij} \neq 0$.

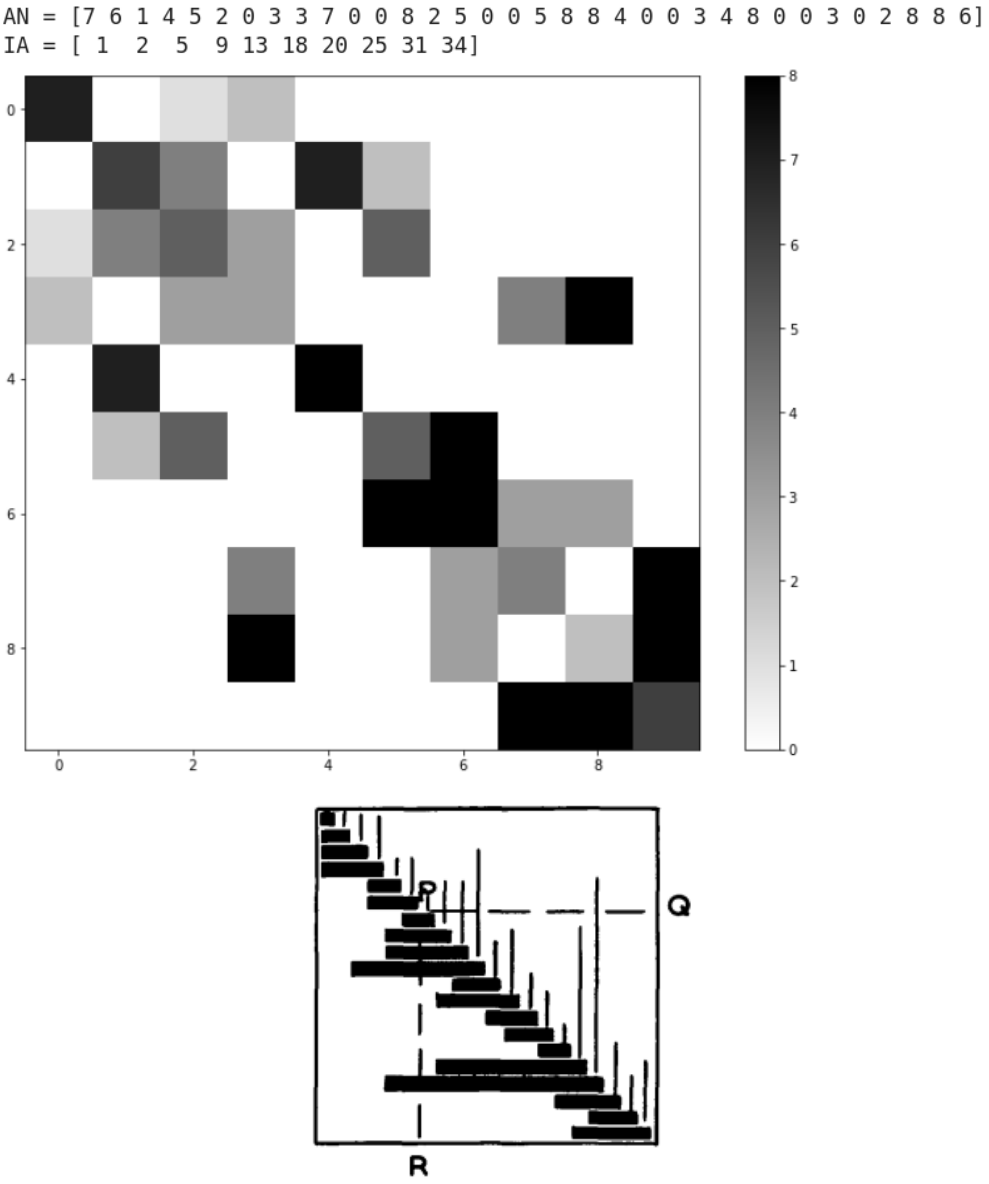

Рис. 1.1: Схема Дженнингса

2 Конструкторский раздел

В данном разделе будет приведено описание алгоритмы, который суммирует элементы в строке без распаковки матрицы.

2.1 Вычисление суммы строк матрицы в схеме Дженнингса

На рисунках 2.1 показаны пути, по которым вычисляется сумма строки матрицы.

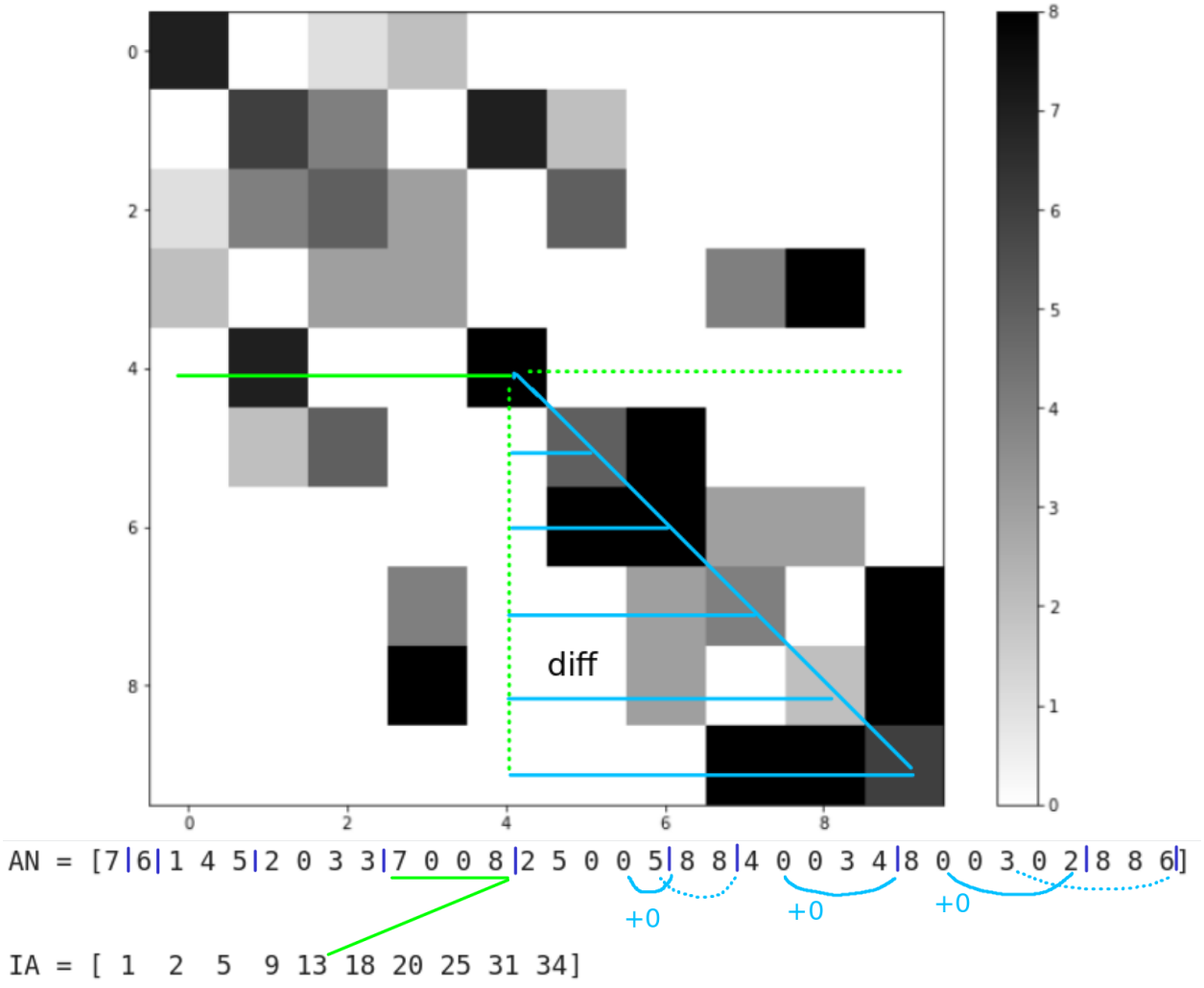

Рис. 2.1: Вычисление суммы строк матрицы в схеме Дженнингса

Код

```
int sum_row(int n)
{
    int start = (n == 0) ? 0 : IA[n-1];
    int stop = IA[n];
    int sum = 0;

    for (int i = start; i < stop; i++)
        sum += AN[i];

    for (int i = n+1, diff = 2; i < N; i++, diff++)
        if (IA[i] - IA[i-1] >= diff)
            sum += AN[IA[i]-diff];

    return sum;
}
```

2.2 Многопоточная реализация обхода строк матрицы

Реализация многопоточности просто разбивается на n меньших областей и находит индекс и максимальную сумму в этой области. Затем найти индекс строки с максимальной суммой из n областей.

3 Технологический раздел

3.1 Средства реализации

Язык программирования: C++, Python

Библиотеки: matplotlib, scipy.sparse (python, для генерации случайных матриц и визуализации)

Редактор: VS Code

Я использую эти инструменты потому, что они мощные, широко используемые.

3.2 Листинг кода

Листинг 3.1: Шаблон разреженной матрицы

```
template <size_t N, typename T=int>
class SymSparseMatrix
{
private:
    vector<T> AN;
    T IA[N];
    int _max_sum_row_index; // for testing

public:

    SymSparseMatrix(const char* path)
    {
        ifstream file(path);
        string line;
        int a;
        char b;

        getline(file, line);
        stringstream ss(line);
        while (ss >> a)
        {
            AN.push_back(a);
            ss >> b;
        }

        getline(file, line);
        int i = 0;
        stringstream ss(line);
        while (ss >> a)
        {
```

```
            IA[i++] = a;
            ss >> b;
        }

        file >> _max_sum_row_index;
    }

    ~SymSparseMatrix() {}


    ostream& display(ostream& os) const
    {
        os << "\n[AN]\n";
        for (auto i : AN) os << i << ',';
        os << "\n[IA]\n";
        for (int i = 0; i < N; i++) os << IA[i] << ',';
        os << '\n';
        return os;
    }

    bool test(int mr)
    {
        return mr == _max_sum_row_index;
    }

    int sum_row(int n)
    {
        int start = (n == 0) ? 0 : IA[n-1];
        int stop = IA[n];
        int sum = 0;

        for (int i = start; i < stop; i++)
            sum += AN[i];

        for (int i = n+1, diff = 2; i < N; i++, diff++)
            if (IA[i] - IA[i-1] >= diff)
                sum += AN[IA[i]-diff];

        return sum;
    }

    int max_row()
    {
        int mr = 0;
        int max = 0;    // only int
        for (int i = 0; i < N; i++)
        {
            int s = sum_row(i);
            if (s > max)
```

```
80              {
81                  mr = i;
82                  max = s;
83              }
84          }
85          return mr;
86      }
87
88      int max_row(size_t n_thread)
89      {
90          if (n_thread > N) n_thread = N;
91          int max_sum[n_thread];
92          int max_index[n_thread];
93
94          auto f = [&](size_t begin, int inc) {
95              int mr = 0;
96              int max = 0;
97              for (int i = begin; i < N; i += inc)
98              {
99                  int s = sum_row(i);
100                 if (s > max)
101                 {
102                     mr = i;
103                     max = s;
104                 }
105             }
106             max_sum[begin] = max;
107             max_index[begin] = mr;
108         };
109
110         parallelize(f, N, n_thread);
111
112         int mr = max_index[0];
113         int max = max_sum[0];
114         for (int i = 1; i < n_thread; i++)
115         {
116             if (max_sum[i] > max)
117             {
118                 mr = max_index[i];
119                 max = max_sum[i];
120             }
121         }
122         return mr;
123     }
124 };
125
126
127 template<size_t N, typename T=int>
128 ostream& operator<<(ostream& os, const SymSparseMatrix<N,T>& m)
```

```
{
    m.display(os);
    os << '\n';
    return os;
}


// function can run in parallel
using f_parallel_t = std::function<void(size_t begin, size_t end)>;

void parallelize(f_parallel_t f, size_t loop_size, size_t n_thread)
{
    if (n_thread > loop_size)
        n_thread = loop_size;

    size_t block_size = loop_size / n_thread;
    size_t i = 0;

    // + one main thread
    std::vector<std::thread> threads(n_thread-1);

    for (i = 0; i < n_thread-1; i++)
        threads[i] = std::thread(f, i, n_thread);

    // main thread
    f(i, n_thread);

    for (auto& thread : threads)
        thread.join();
}
```

Листинг 3.2: Функции поддерживают создание данных, построение графиков, экспорт в тестовые файлы

```
RANGE = 8

def randSymMatrix(rank, density=0.2, format='coo', dtype=np.int8):
    m = scipy.sparse.rand(rank, rank, density=density, format=format,
       dtype=dtype)
    m = m.todense() % RANGE
    return m


def randLinearPattern(rank, density=0.01, center=1.05, dtype=np.int8):
    m = randSymMatrix(rank, density, dtype=dtype)
    for i in range(rank):
        for j in range(i+1):
```

```
            if i == j:
                if not m[i,j]:
                    m[i,j] = randint(1,RANGE)
            else:
                if not randint(0, int((i-j)**center)):
                    m[i,j] = randint(1,RANGE)
    return m


def toJennings(m):
    AN = []
    IA = []
    try: m = m.tolist()
    except: pass
    l = len(m[0])
    for i in range(l):
        for j in range(i+1):
            if m[i][j]:
                AN += m[i][j:i+1]
                IA.append(len(AN))
                break
    return np.array(AN), np.array(IA)


def jenningsToMatrix(AN, IA):
    m = []
    l = len(IA)
    start = 0
    for i in range(l):
        end = IA[i]
        a = [0]*(i+1+start-end) + AN[start:end] + [0]*(l-i-1)
        m.append(a)
        start = end
    return np.matrix(m)


# change source matrix
def toHalfMatrix(m):
    l = m.shape[0]
    for i in range(l-1):
        for j in range(i+1, l):
            m[i,j] = 0
    return m


# change source matrix
def toFullMatrix(m):
    l = m.shape[0]
    for i in range(l-1):
```

```
62          for j in range(i+1, l):
63              m[i,j] = m[j,i]
64      return m
65
66
67  def plotMatrix(m):
68      try: toFullMatrix(m)
69      except: pass
70      plt.imshow(m, interpolation='none', cmap='binary')
71      plt.colorbar()
72
73
74  def toJenningsFile(path, m):
75      AN, IA = toJennings(m)
76      with open(path, 'w') as f:
77          f.write(','.join(map(str, AN)))
78          f.write('\n')
79          f.write(','.join(map(str, IA)))
80          f.write('\n')
81          f.write(str(np.sum(m, axis=0).argmax()))
```

4 Экспериментальный раздел

В данном разделе будет приведено пример работы программы, результаты тестирования и сравнение времени работы последовательного и параллельного алгоритма Винограда.

4.1 Пример работы и результаты тестирования

На рисунке 4.1 и 4.2 приведен пример работы программы и результат теста.

```
=== Benchmark ===
100,33,34,59,67,126,
200,143,134,98,75,170,
300,323,325,196,163,201,
400,597,565,356,219,284,
500,907,911,555,339,544,
600,1364,1319,788,480,548,
700,1811,1807,1088,683,693,
800,2430,2427,1458,857,1407,
900,3169,3144,1900,1065,1160,
1000,3860,3890,2318,1393,1795,

=== Program ===

[AN]
5,8,7,2,7,0,0,4,1,5,0,0,5,4,3,0,0,0,0,0,3,1,4,
5,7,0,0,0,5,8,8,8,
[IA]
1,2,4,8,9,14,21,23,24,32,

9

=== Testing ===
Not parallelize true
1 threads true
2 threads true
3 threads true
4 threads true
5 threads true
6 threads true
7 threads true
8 threads true
9 threads true
```

Рис. 4.1: Пример работы и результаты тестирования

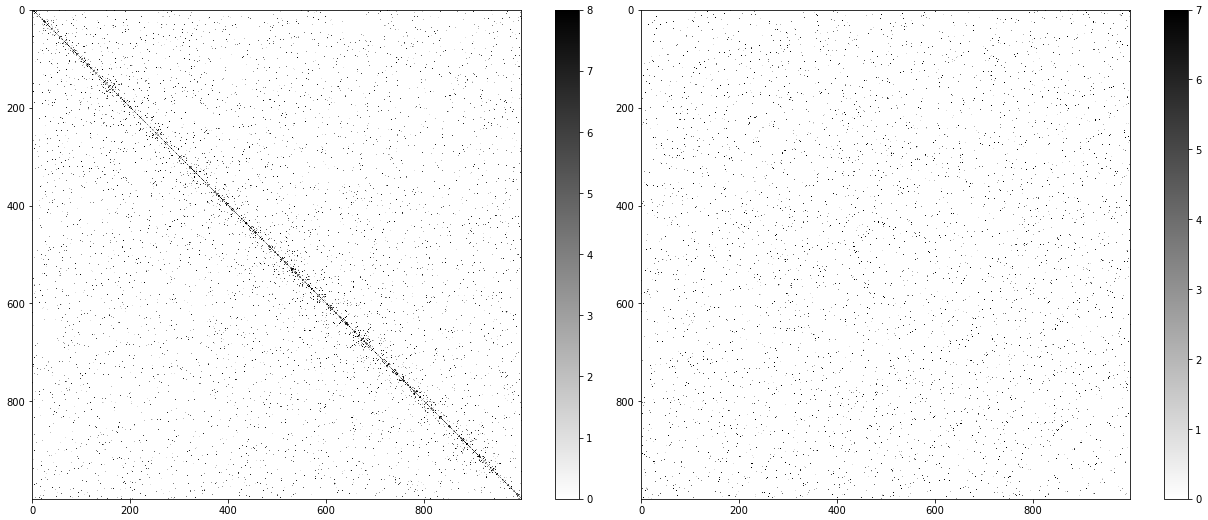

Рис. 4.2: Пример сгенерированной матрицы (1000x1000)

4.2 Сравнение времени работы

Операционная система - Ubuntu 20.04.1 LTS

Процессор - Intel® CoreTM i5-7300HQ CPU @ 2.50GHz × 4 (ЦП 4 ядра 4 потока)

В таблице 4.1 приведены замеры времени работы.

Размер	**Последо.**	**1 поток**	**2 поток**	**4 поток**	**8 поток**
100	33	34	59	67	126
200	143	134	98	75	170
300	323	325	196	163	201
400	597	565	356	219	284
500	907	911	555	339	544
600	1364	1319	788	480	548
700	1811	1807	1088	683	693
800	2430	2427	1458	857	1407
900	3169	3144	1900	1065	1160
1000	3860	3890	2318	1393	1795

Таблица 4.1: Времени работы (10^{-6}с)

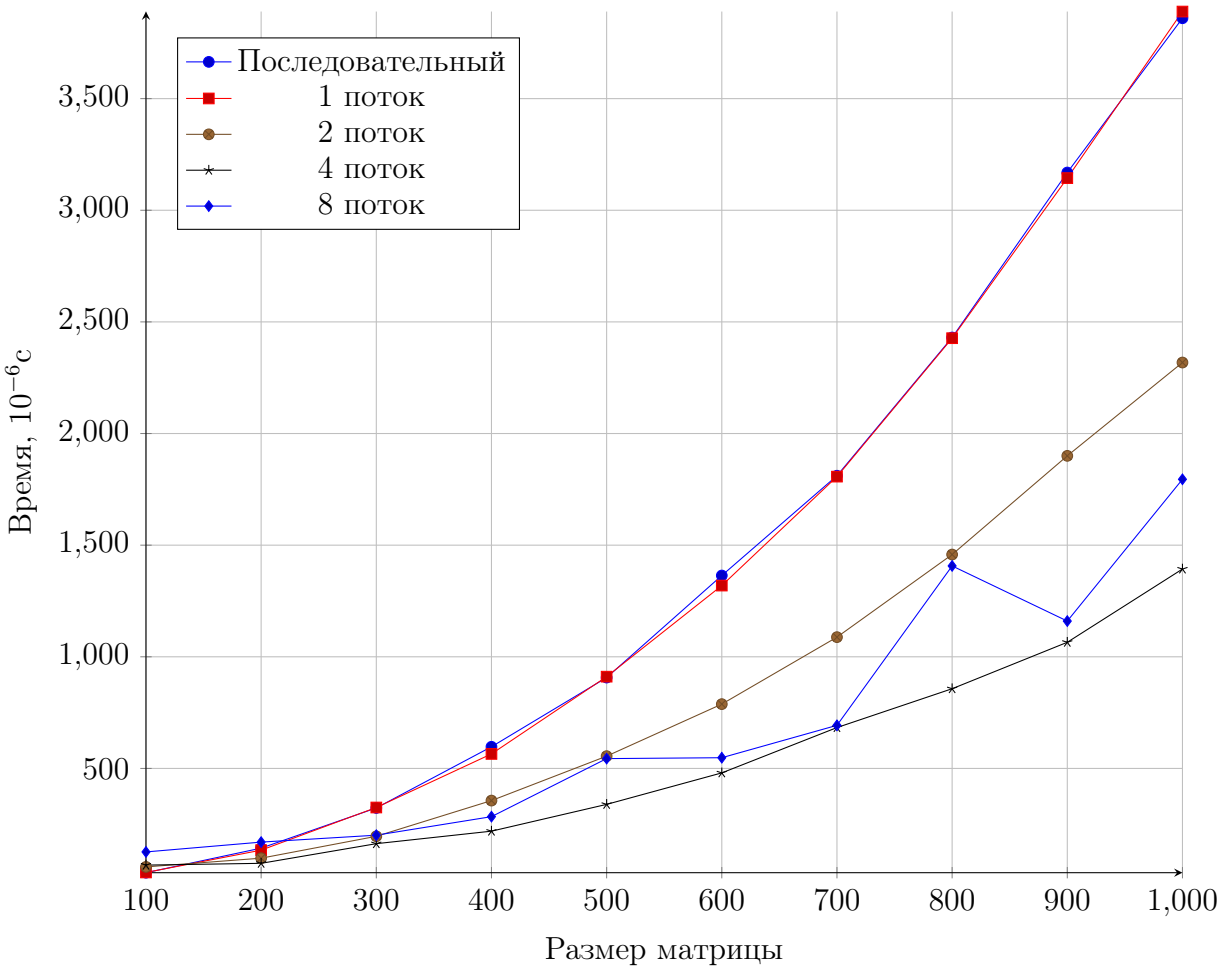

Рис. 4.3: Зависимость времени работы алгоритмов умножения матриц от размеры матрицы и количество потоков

4.3 Вывод

График показывает, что многопоточная версия более эффективна, когда количество потоков увеличивается, производительность увеличивается с увеличением количества потоков до тех пор, пока она не станет равной количеству ядер процессора, и наиболее эффективна, когда количество потоков равно количеству ядер процессора. Затем, если количество потоков увеличивается, происходит небольшое уменьшение из-за необходимости управлять большим количеством потоков.

Заключение

В ходе работы было изучено параллельных вычисления, работая с разреженными матричными структурами. Было сравнить временные характеристики последовательного и параллельного реализации и сделаны следующие выводы:

- производительность увеличивается с увеличением количества потоков до тех пор, пока она не станет равной количеству ядер процессора;
- многопоточная версия наиболее эффективна когда количество потоков равно количеству ядер процессора;
- время выполнения с использованием 4 потоков всего 46.5% по сравнению с последовательным выполнением (на матрице 1000x1000).