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

Основы параллельного программирования

Отчет

О выполнении работы № 2

Работу выполнил:
Е. И. Биточкин
Группа: 22209
Преподаватель:
А. А. Артюхов

Новосибирск
2024

Содержание

1. Цель

- Реализовать решение СЛАУ методом простой итерации
- Разделить вычисление на несколько потоков
- Разделить по потокам и матрицу, и вектор значений
- Оценить и сравнить эффективность обоих методов
- Объяснить результаты

2. Задание

- Реализация однопроцессорного результата.
- Разбиение матрицы.
- Разбиение вектора.

3. Описание работы

3.1. Реализация с одним потоком

Для начала был реализован однопроцессорный вариант (Листинг 1). В нем нет OpenMP, и все считается в одном процессе.

3.2. Реализация с несколькими потоками

В первоначальном варианте (Листинг 2) не было редукции по сумме, а значения складывались в отдельный вектор. Это, очевидно, не эффективное по памяти решение, но предполагалось, что это даст небольшой выйгрыш по времени.
Было решение сделать вариант с редукцией (Листинг 3), чтобы сравнить два решения
Также был реализован описаный в спецификации вариант с вынесением в параллельный блок всего блока с вычислениями (Листинг 4).

3.3. Результаты измерений

Измерения проводились на моей машине и на кластере, с разными размерами и разными оптимизациями компилятора, чтобы исключить их влияние.

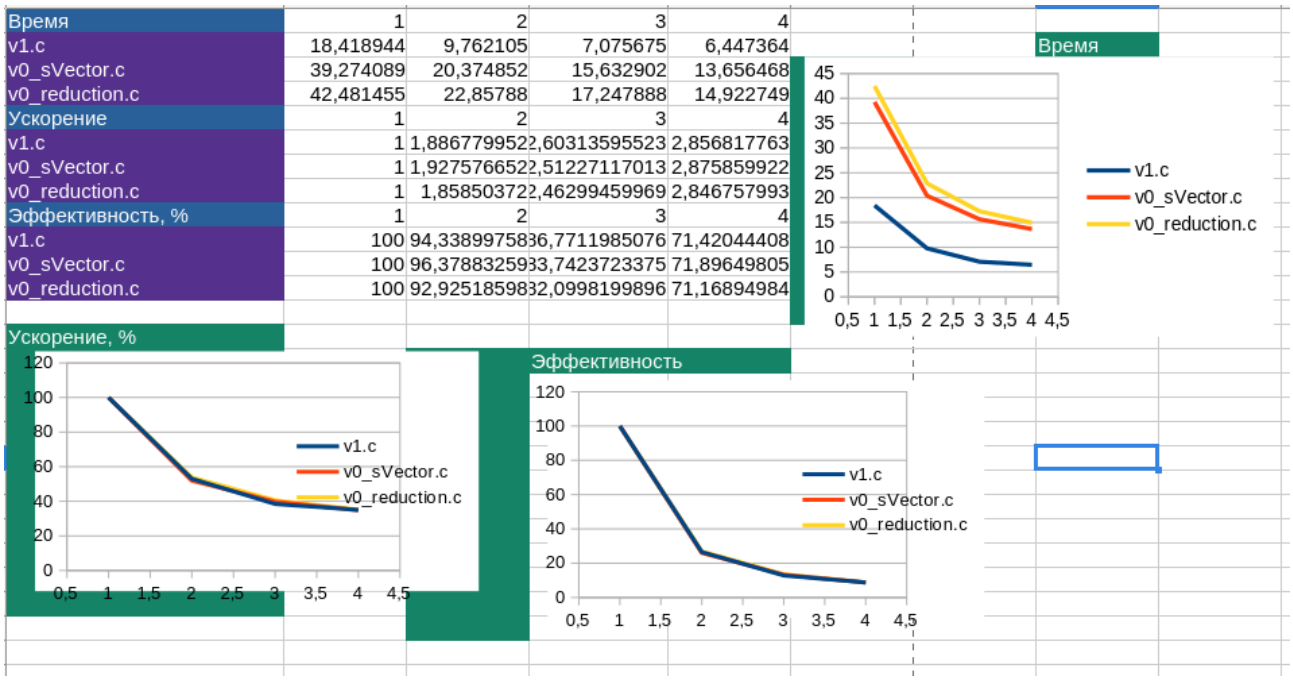

Рисунок 3.1. N=2k; -O3; Запуск на ноутбуке

Был также проведен замер на сервере, но с большим числом потоков. Однако, достоверность данных вызывает сомнения.

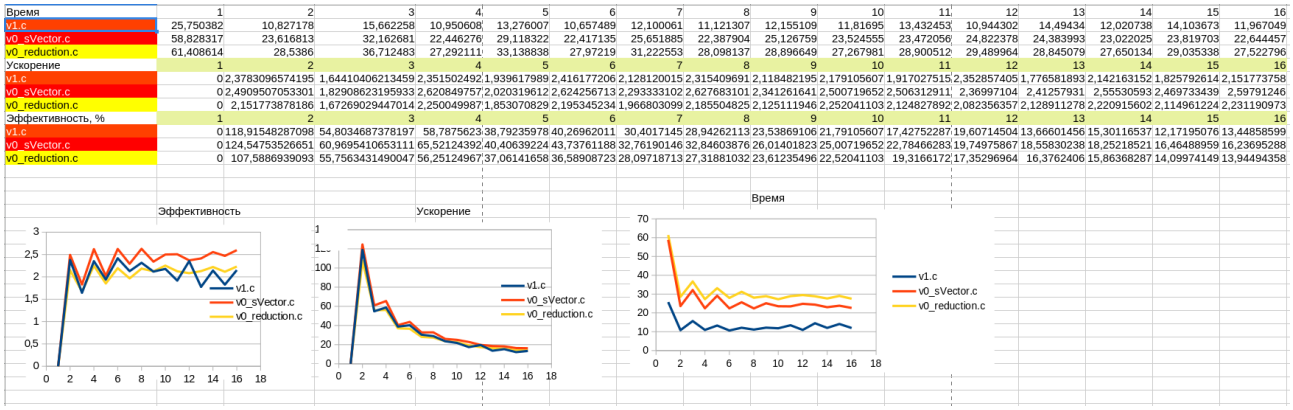

Рисунок 3.2. Запуск на кластере

Чтобы исключить влияние оптимизации, было принято решение изменить уровень оптимизации от компилятора: вместо O3 был взят O2:

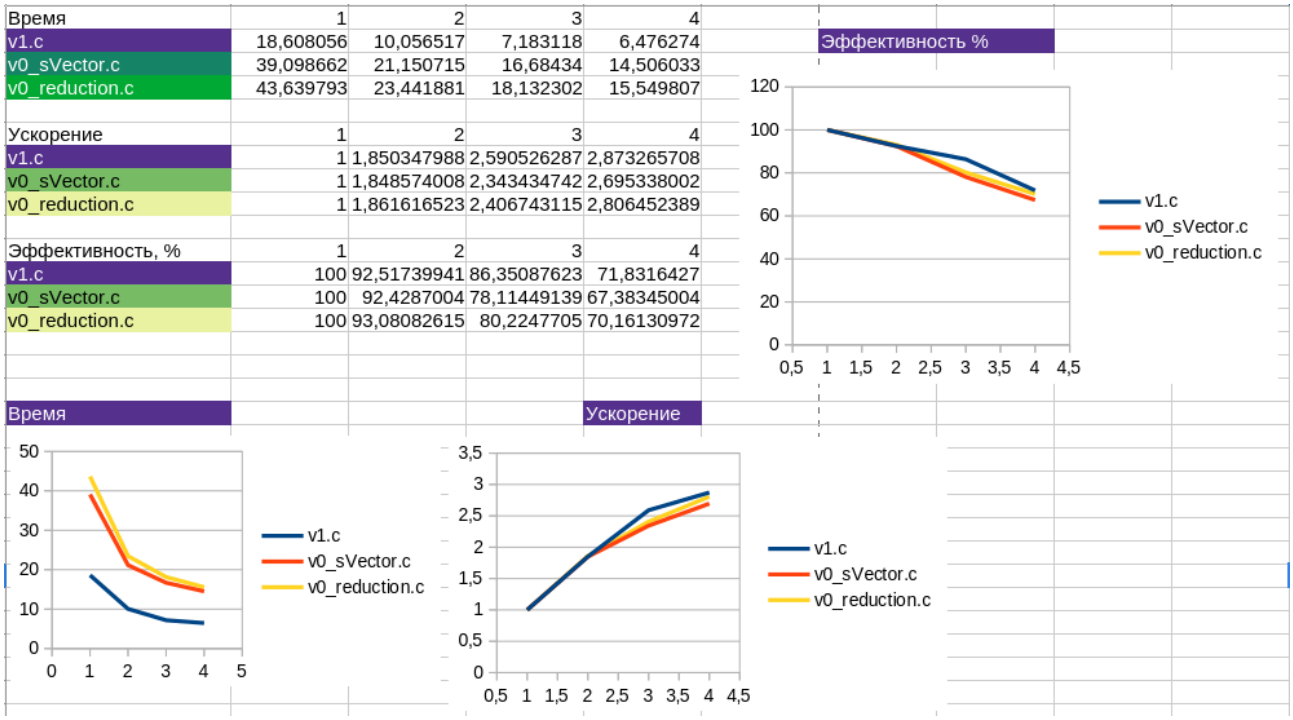

Рисунок 3.3. N=2k; -O2; ноутбук

Для большей убедительнотси полученых результатов было принято решение перепроверить предыдущий замер с большим размером матрицы:

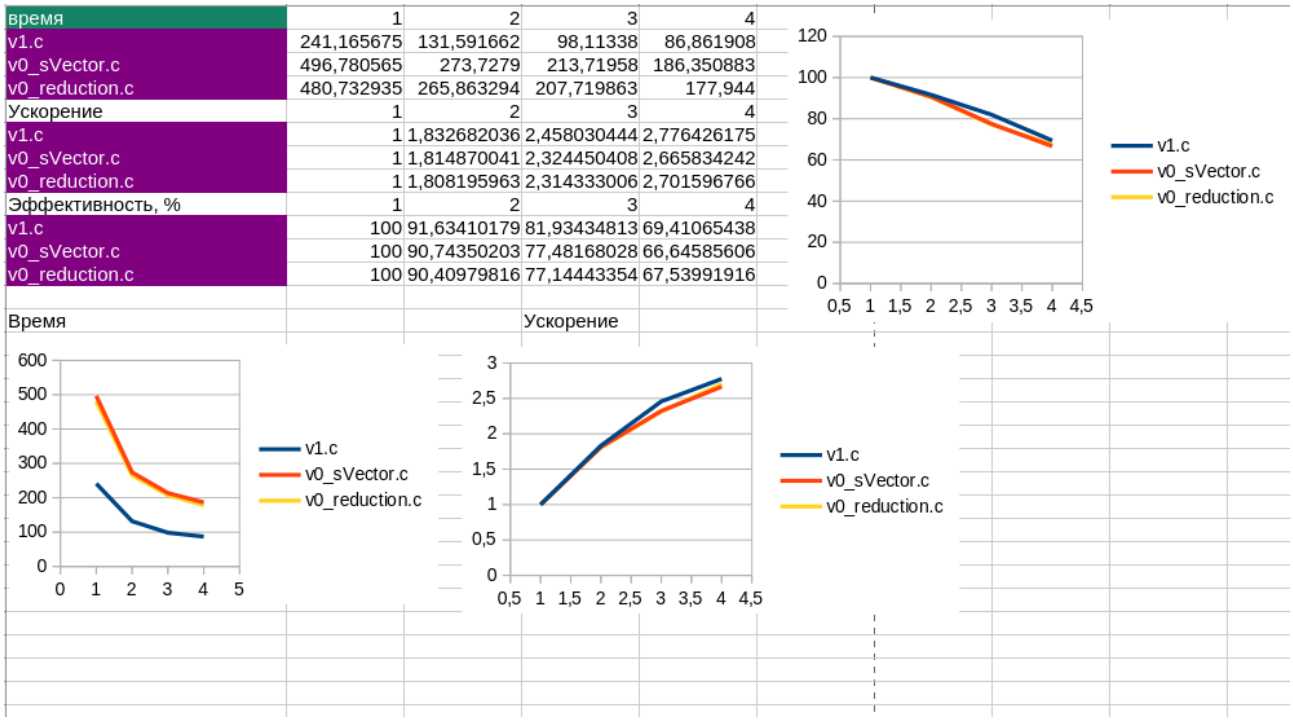

Рисунок 3.4. N=25k; -O2, ноутбук

3.4. Профилирование

3.4.1. static

Для профилирования был взят вариант с одной парралельной секцией.

	0	1	2	4	8
static, 1	22,433242	12,801077	7,377984	5,513413
static, 2	22,442853	12,769219	7,127199	5,33022
static, 4	22,745062	12,791263	7,056182	5,200483
static, 8	22,43578	12,783015	7,124807	5,164773
static, 16	22,522248	12,66321	7,127492	5,154173
average	22,515837	12,7615568	7,1627328	5,2726124

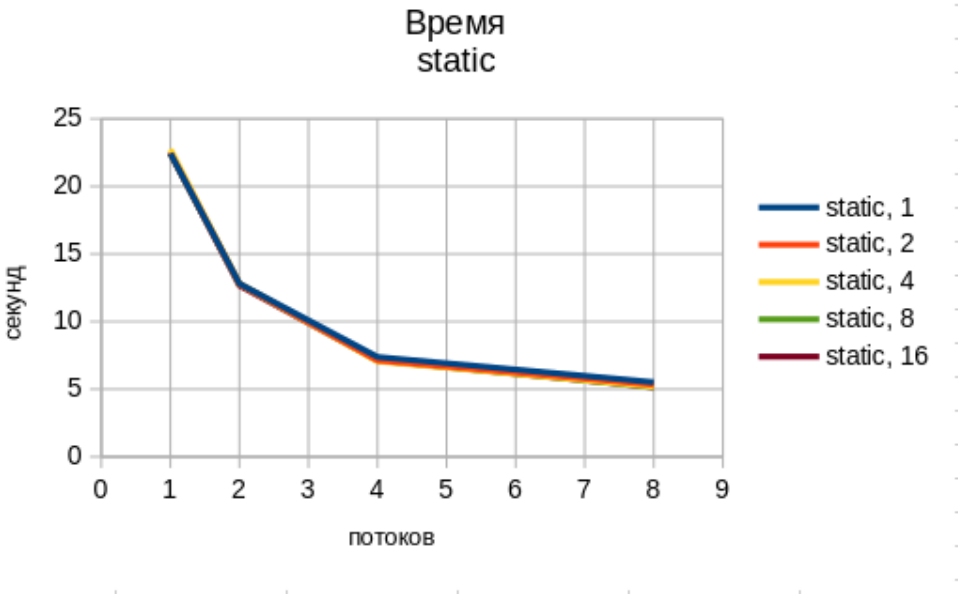

Рисунок 3.5. Время static

	0	1	2	4	8
static, 1	1	1,752449579047	3,040565282	4,068848461	
static, 2	1	1,757574445234	3,148902255	4,210492813	
static, 4	1	1,778171709862	3,223423375	4,37364414	
static, 8	1	1,755124280148	3,148966702	4,344001179	
static, 16	1	1,778557569526	3,15991207	4,3697113	
average	1	1,764375516764	3,144353937	4,273339579	

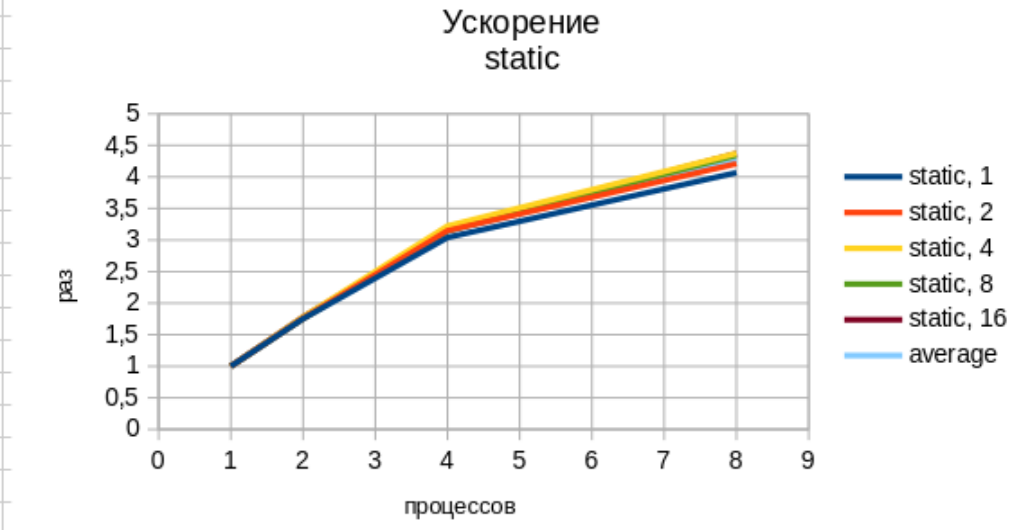

Рисунок 3.6. Ускорение static

	0	1	2	4	8
static, 1	100	87,62247895	76,01413204	50,86060576
static, 2	100	87,87872226	78,72255636	52,63116016
static, 4	100	88,90858549	80,58558439	54,67055175
static, 8	100	87,75621401	78,72416755	54,30001473
static, 16	100	88,92787848	78,99780175	54,62139125
average	100	88,21877584	78,60884842	53,41674473

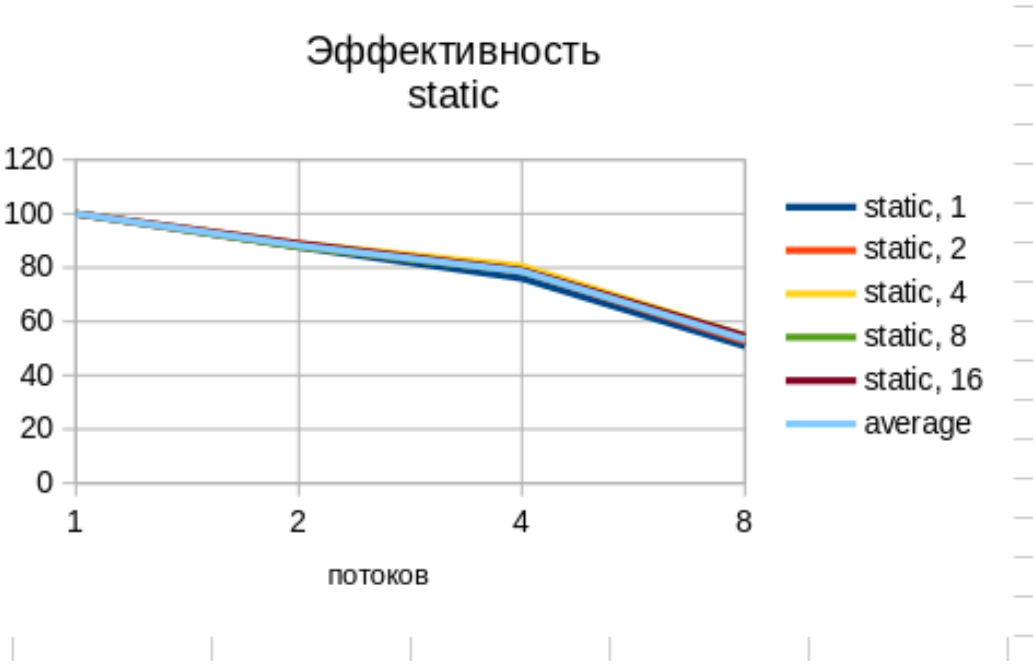

Рисунок 3.7. Эффективность static

3.4.2. dynamic

0	1	2	4	8
dynamic, 1	22,638619	13,128213	8,591348	6,420515
dynamic, 2	22,803704	12,884897	8,416478	6,381926
dynamic, 4	22,573944	12,774937	8,328699	6,238918
dynamic, 8	23,294493	12,715008	8,262546	6,170693
dynamic, 16	22,657126	12,64832	8,24058	6,147852
average	22,7935772	12,830275	8,3679302	6,2719808

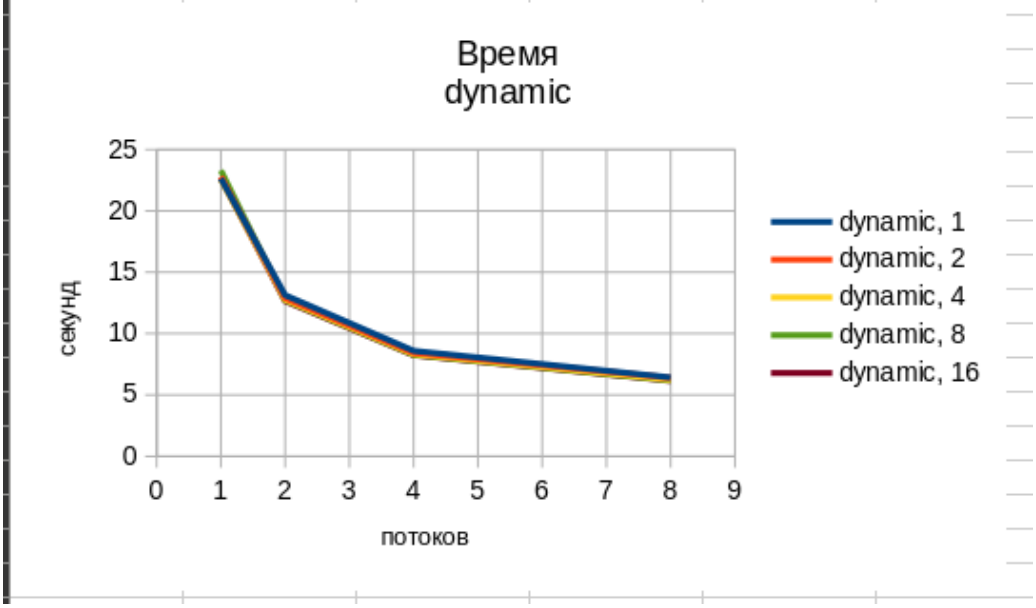

Рисунок 3.8. Время dynamic

	0	1	2	4	8
dynamic, 1	22,638619	13,128213	8,591348	6,420515	
dynamic, 2	22,803704	12,884897	8,416478	6,381926	
dynamic, 4	22,573944	12,774937	8,328699	6,238918	
dynamic, 8	23,294493	12,715008	8,262546	6,170693	
dynamic, 16	22,657126	12,64832	8,24058	6,147852	
average	22,7935772	12,830275	8,3679302	6,2719808	

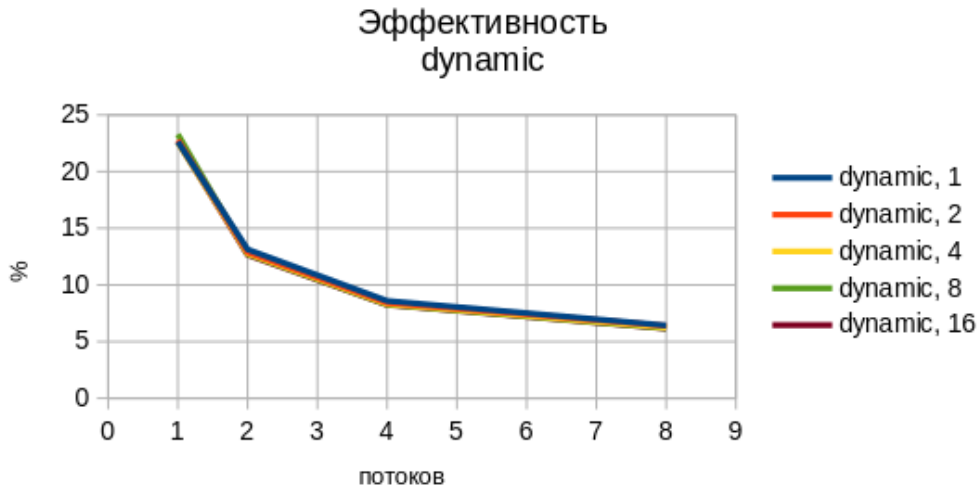

Рисунок 3.10. Эффективность dynamic

	0	1	2	4	8
dynamic, 1	1	1,724425023	2,63504854	3,525981794	
dynamic, 2	1	1,769801031	2,709411704	3,573169604	
dynamic, 4	1	1,76704934	2,710380577	3,618246625	
dynamic, 8	1	1,832047058	2,819287542	3,775020569	
dynamic, 16	1	1,791315052	2,749457684	3,685372712	
average	1	1,77654627	2,72392057	3,634191163	

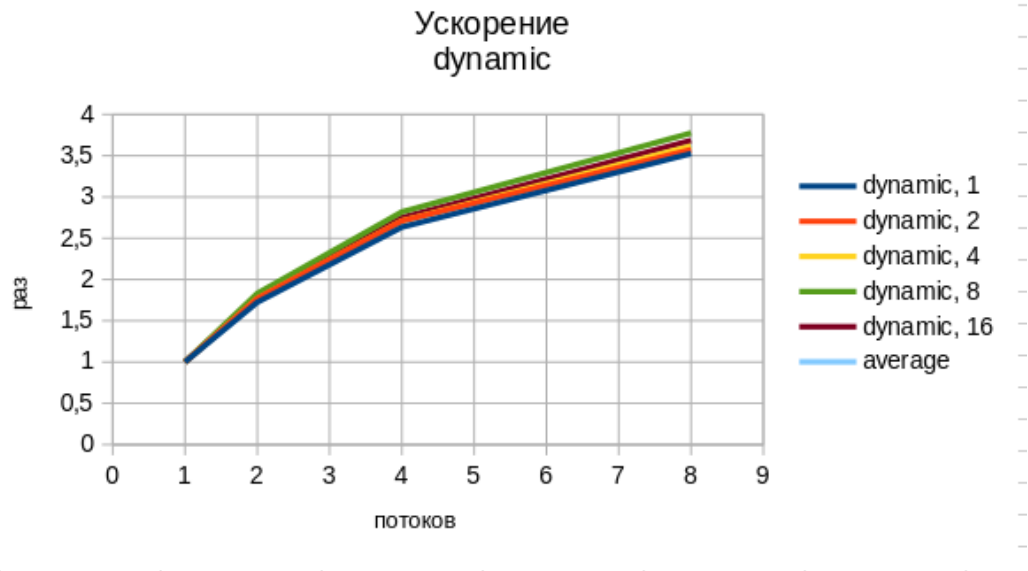

Рисунок 3.9. Ускорение dynamic

3.4.3. guided

	0	1	2	4	8
guided, 1	23,096098	12,630273	8,793945	6,311978	
guided, 2	22,608923	12,524201	8,655811	6,447757	
guided, 4	22,492655	12,679588	8,52887	6,171767	
guided, 8	22,625254	12,61428	8,544366	6,372996	
guided, 16	22,633798	12,638009	8,536113	6,35062	
average	22,6913456	12,6172702	8,611821	6,3310236	

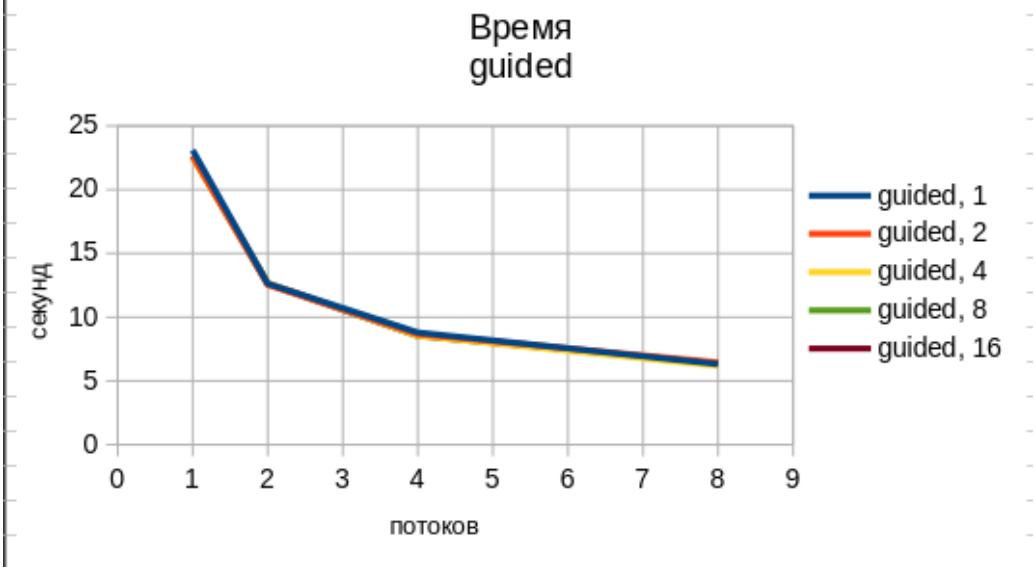

Рисунок 3.11. Время guided

	0	1	2	4	8
guided, 1	1	1,828630149	2,62636370821059	3,65909038339487	
guided, 2	1	1,805218792	2,6119936075314	3,50647876463086	
guided, 4	1	1,773926329	2,63723740659665	3,64444331744863	
guided, 8	1	1,793622307	2,64797341312392	3,55017545907765	
guided, 16	1	1,790930676	2,65153448648114	3,56402965379758	
average	1	1,798465651	2,63502052438874	3,58484351566992	

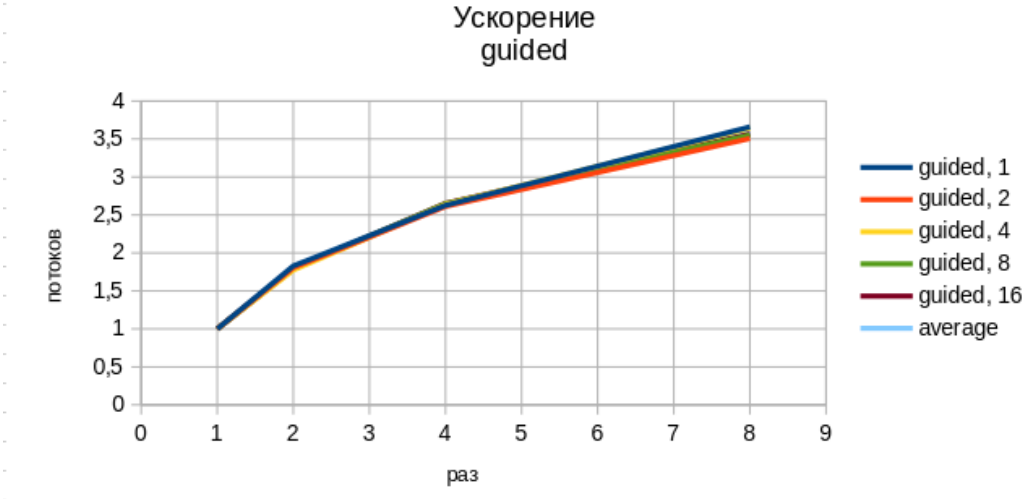

Рисунок 3.12. Ускорение guided

	0	1	2	4	8
guided, 1	100	91,43150746	65,65909271	45,7386297924359	
guided, 2	100	90,2609396	65,29984019	43,8309845578858	
guided, 4	100	88,69631647	65,93093516	45,5555414681079	
guided, 8	100	89,68111537	66,19933533	44,3771932384706	
guided, 16	100	89,54653379	66,28836216	44,5503706724698	
average	100	89,92328254	65,87551311	44,810543945874	

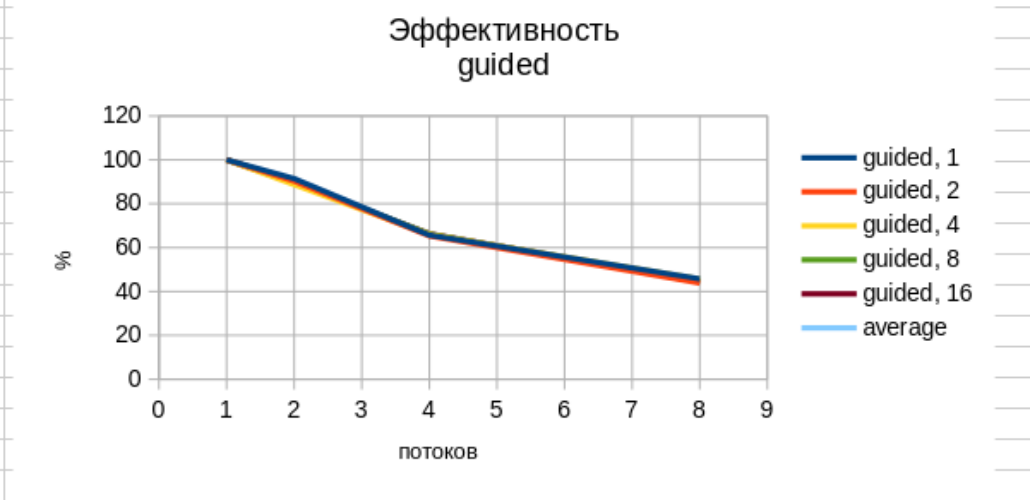

Рисунок 3.13. Эффективность guided

3.4.4. Сравнение

Возьмем среднее значение для каждого типа и сравним:

Время	1	2	4	8
static	22,515837	12,7615568	7,1627328	5,2726124
dynamic	22,7935772	12,830275	8,3679302	6,2719808
guided	22,6913456	12,6172702	8,611821	6,3310236

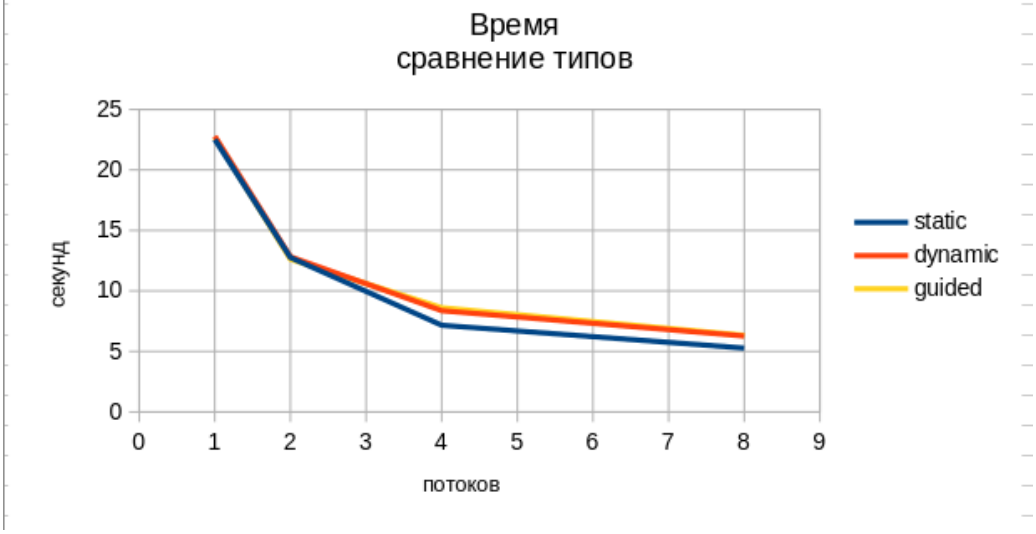

Рисунок 3.14. Сравнение времени

Ускорение	1	2	4	8
static	1	1,764375517	3,144353937	4,273339579
dynamic	1	1,776927501	2,724717209	3,635558261
guided	1	1,798465651	2,635020524	3,584843516

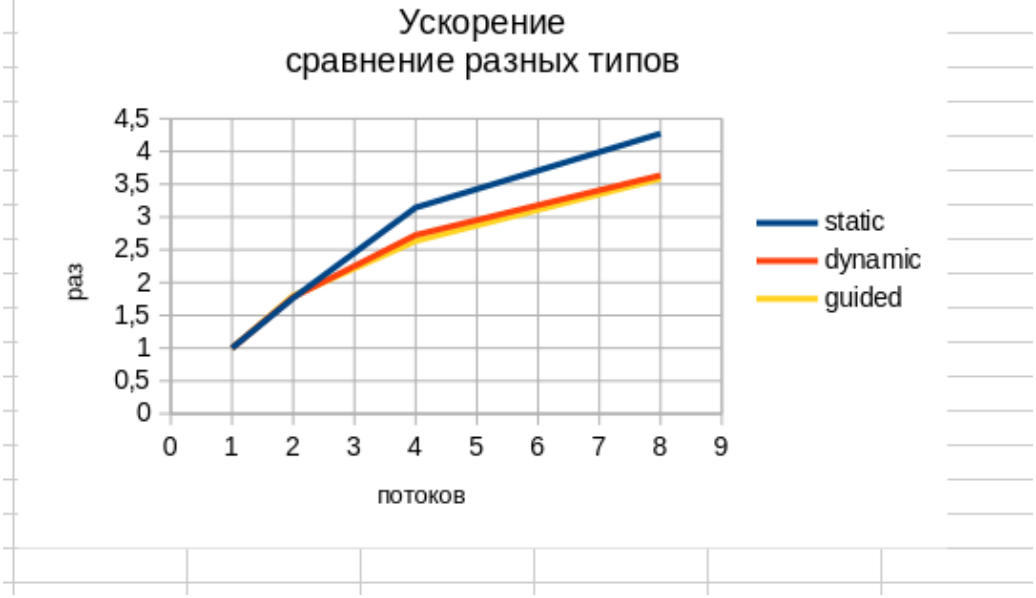

Рисунок 3.15. Сравнение ускорения

Эффективность	1	2	4	8
static	100	88,2187758381758	78,6088484184128	53,4167447319073
dynamic	100	88,8463750470013	68,1179302292908	45,444478263326
guided	100	89,9232825391988	65,8755131097186	44,810543945874

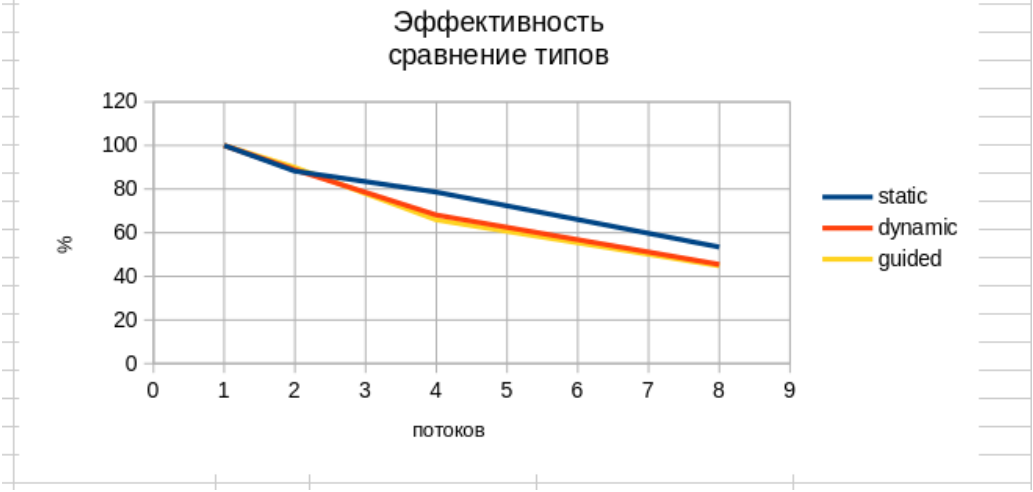

Рисунок 3.16. Сравнение эффективности

4. Заключение

Был реализован алгоритм простой итерации, вычисляющий значения параллельно на нескольких потоках в общей памяти.

5. Приложение

5.1. Исходный код

https://github.com/BigCubeCat/bpp_labs.git

5.2. Листинг 1

```
#include <float.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define EPSILON 0.0000001
#define TAU 0.000001
#define MAX_ITERATIONS 100000
#define N 1000

void InitA(double *A, const int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            A[i * n + j] = 1 + (i == j);
        }
    }
}

void InitB(double *B, const int n) {
    for (int i = 0; i < n; ++i) {
        B[i] = n + 1;
    }
}

int main(int argc, char **argv) {
    srand(time(NULL));

    double *A = (double *)malloc(N * N * sizeof(double));
    double *b = (double *)malloc(N * sizeof(double));
    double *x_n = (double *)malloc(N * sizeof(double));
    double *x = (double *)calloc(N, sizeof(double));

    InitA(A, N);
    InitB(b, N);

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);

    int countIters = 0;
    double endParam = DBL_MAX;
    double prevParam = DBL_MAX;

    // length of B vector
    double EPSILON_SQUARE = 0;
    for (int i = 0; i < N; ++i) EPSILON_SQUARE += b[i] * b[i];
    EPSILON_SQUARE *= EPSILON * EPSILON;

    // TAU setup
```

```
    int useTau = 0;
    double tau = TAU;

    for (; countIters < MAX_ITERATIONS; ++countIters) {
        endParam = 0;
        for (int i = 0; i < N; ++i) {
            double sum = -b[i];    // сразу отнимаем b
            for (int j = 0; j < N; ++j) {
                sum += A[i * N + j] * x_n[j];
            }
            x[i] = x_n[i] - sum * tau;
            endParam += sum * sum;
        }
        if (prevParam ⩽ endParam) {    // условие смены знака скаляра.
            if (useTau)
                break;    // Очевидно, что на прямой к числу можно
                 ↪  приближаться
                          // либо слева, либо справа
            tau *= -1;
            useTau = 1;
        }
        memcpy(x_n, x, N * sizeof(double));    // swap
        if (endParam < EPSILON_SQUARE ||
            endParam == DBL_MAX) {    // Условия выхода из спецификации
            break;
        }
        prevParam = endParam;
    }
    printf("%f\n", x[0]);
    printf("count iterations = %d\n", countIters);
    clock_gettime(CLOCK_MONOTONIC_RAW, &end);
    printf("Time taken: %lf sec.\n",
           end.tv_sec - start.tv_sec +
               0.000000001 * (end.tv_nsec - start.tv_nsec));

    free(A);
    free(b);
    free(x_n);
    free(x);

    return 0;
}
```

Листинг 1: Без OpenMP

5.3. Листинг 2

```
#include <float.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const double EPSILON = 0.0000001;
const double TAU = 0.000001;
const int MAX_ITERATIONS = 100000;
```

```
10
11  void Init(double *A, double *B, const int n) {
12      for (int i = 0; i < n; ++i) {
13          B[i] = n + 1;
14          for (int j = 0; j < n; ++j) {
15              A[i * n + j] = 1 + (i == j);
16          }
17      }
18  }
19
20  double calcEndValue(const double *bVector, int n, double eps) {
21      double res = 0;
22  #pragma omp parallel for reduction(+ : res)
23      for (int i = 0; i < n; ++i) {
24          res += bVector[i] * bVector[i];
25      }
26      return res * eps * eps;
27  }
28
29  int solve(const double *A, const double *b, double *sumVector, double
    ↪  *x,
30            double *x_n, int n, double bLen, double tau) {
31      int countIters = 0;
32      double nextParam;
33      double prevParam = DBL_MAX;
34      int flag = 1;
35      for (; flag && (countIters < MAX_ITERATIONS); ++countIters) {
36          nextParam = 0;
37          for (int i = 0; i < n; ++i) {
38              sumVector[i] = -b[i];
39          }
40  #pragma omp parallel for reduction(+ : nextParam)
41          for (int i = 0; i < n; ++i) {
42              for (int j = 0; j < n; ++j) {
43                  sumVector[i] += A[i * n + j] * x_n[j];
44              }
45              nextParam += sumVector[i] * sumVector[i];    // reduce
46              x[i] = x_n[i] - sumVector[i] * tau;
47          }
48          if (prevParam ⩽ nextParam) {
49              flag = 0;
50          }
51          double *tmp = x;
52          x = x_n;
53          x_n = tmp;
54
55          if (nextParam < bLen || nextParam == DBL_MAX) {
56              break;
57          }
58          prevParam = nextParam;
59      }
60      return countIters;
61  }
62
63  int main(int argc, char **argv) {
64      int n = atoi(argv[1]);
65      int countThreads = atoi(argv[2]);
66
67      double *A = (double *)malloc(n * n * sizeof(double));
68      double *b = (double *)malloc(n * sizeof(double));
```

```
    double *x_n = (double *)malloc(n * sizeof(double));
    double *sVector = (double *)malloc(n * sizeof(double));
    double *x = (double *)calloc(n, sizeof(double));
    Init(A, b, n);

    omp_set_num_threads(countThreads);

    double itime, ftime, exec_time;
    itime = omp_get_wtime();

    double bLen = calcEndValue(b, n, EPSILON);

    int countIters = solve(A, b, sVector, x, x_n, n, bLen, TAU);

    ftime = omp_get_wtime();
    exec_time = ftime - itime;
    printf("%f\n", exec_time);
    printf("%d\n", countIters);
    printf("%f\n", x[0]);

    free(A);
    free(b);
    free(x_n);
    free(x);

    return 0;
}
```

Листинг 2: $v0_s Vector.c; parallel for,$

5.4. Листинг 3

```
#include <float.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const double EPSILON = 0.0000001;
const double TAU = 0.000001;
const int MAX_ITERATIONS = 100000;

void Init(double *A, double *B, const int n) {
    for (int i = 0; i < n; ++i) {
        B[i] = n + 1;
        for (int j = 0; j < n; ++j) {
            A[i * n + j] = 1 + (i == j);
        }
    }
}

double calcEndValue(const double *bVector, int n, double eps) {
    double res = 0;
#pragma omp parallel for reduction(+ : res)
    for (int i = 0; i < n; ++i) {
        res += bVector[i] * bVector[i];
```

```
25          }
26          return res * eps * eps;
27      }
28
29      int solve(const double *A, const double *b, double *x, double *x_n, int
        ↪  n,
30                double bLen, double tau) {
31          int countIters = 0;
32          double nextParam;
33          double prevParam = DBL_MAX;
34          int flag = 1;
35          for (; flag && (countIters < MAX_ITERATIONS); ++countIters) {
36              nextParam = 0;
37      #pragma omp parallel for reduction(+ : nextParam)
38              for (int i = 0; i < n; ++i) {
39                  double sum = -b[i];
40      #pragma omp parallel for reduction(+ : sum)
41                  for (int j = 0; j < n; ++j) {
42                      sum += A[i * n + j] * x_n[j];
43                  }
44                  nextParam += sum * sum;    // reduce
45                  x[i] = x_n[i] - sum * tau;
46              }
47              if (prevParam ⩽ nextParam) {
48                  flag = 0;
49              }
50              memcpy(x_n, x, n * sizeof(double));    // swap
51              if (nextParam < bLen || nextParam == DBL_MAX) {
52                  break;
53              }
54              prevParam = nextParam;
55          }
56          return countIters;
57      }
58
59      int main(int argc, char **argv) {
60          int n = atoi(argv[1]);
61          int countThreads = atoi(argv[2]);
62
63          double *A = (double *)malloc(n * n * sizeof(double));
64          double *b = (double *)malloc(n * sizeof(double));
65          double *x_n = (double *)malloc(n * sizeof(double));
66          double *x = (double *)calloc(n, sizeof(double));
67          Init(A, b, n);
68
69          omp_set_num_threads(countThreads);
70
71          double itime, ftime, exec_time;
72          itime = omp_get_wtime();
73
74          double bLen = calcEndValue(b, n, EPSILON);
75
76          int countIters = solve(A, b, x, x_n, n, bLen, TAU);
77
78          ftime = omp_get_wtime();
79          exec_time = ftime - itime;
80          printf("%f\n", exec_time);
81          printf("%d\n", countIters);
82          printf("%f\n", x[0]);
83
```

```
84      free(A);
85      free(b);
86      free(x_n);
87      free(x);
88
89      return 0;
90  }
```

Листинг 3: v0_reduction.c; parallel for,

5.5. Листинг 4

```
1   #include <float.h>
2   #include <omp.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <string.h>
6
7   const double EPSILON = 0.0000001;
8   const double TAU = 0.000001;
9
10  void Init(double *A, double *B, const int n) {
11      for (int i = 0; i < n; ++i) {
12          B[i] = n + 1;
13          for (int j = 0; j < n; ++j) {
14              A[i * n + j] = 1 + (i == j);
15          }
16      }
17  }
18
19  int main(int argc, char **argv) {
20      if (argc < 3) {
21          perror("Usage: ./exe <N> <num threads>");
22          return 1;
23      }
24      int n = atoi(argv[1]);
25      int countThreads = atoi(argv[2]);
26
27      double *A = (double *)malloc(n * n * sizeof(double));
28      double *b = (double *)malloc(n * sizeof(double));
29      double *x_n = (double *)calloc(n, sizeof(double));
30      double *x = (double *)calloc(n, sizeof(double));
31      double bLen = 0;
32      Init(A, b, n);
33
34      double itime, ftime, exec_time;
35      itime = omp_get_wtime();
36
37      int countIters = 0;
38      double nextParam = DBL_MAX;
39      int flag = 1;
40      double tau = TAU;
41      int useTau = 0;
42
43      omp_set_num_threads(countThreads);
44
```

```
#pragma omp parallel
    {
#pragma omp for schedule(static) reduction(+ : bLen)
        for (int i = 0; i < n; ++i) {
            bLen += b[i] * b[i];
        }

        while (flag) {
#pragma omp single
            nextParam = 0;
#pragma omp for schedule(static) reduction(+ : nextParam)
            for (int i = 0; i < n; ++i) {
                double sum = -b[i];
                for (int j = 0; j < n; ++j) {
                    sum += A[i * n + j] * x[j];
                }
                x_n[i] = x[i] - sum * tau;
                nextParam += sum * sum;
            }
#pragma omp single
            {
                double *tmp = x;
                x = x_n;
                x_n = tmp;

                ++countIters;
                if (nextParam < EPSILON * bLen) {
                    flag = 0;
                }
            }
        }
    }

    ftime = omp_get_wtime();
    exec_time = ftime - itime;
    printf("%f\n", exec_time);
    printf("%d\n", countIters);
    printf("%f\n", x[0]);

    free(A);
    free(b);
    free(x_n);
    free(x);

    return 0;
}
```

Листинг 4: v1.c; parallel