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 № 1

**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 процессов
- Разделить по процессам 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 процессом

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

## 3.2. Реализация с несколькими процессами

### 3.2.1. Без кольца

Реализация - (Листинг 2)

Матрица была разделена на несколько процессов. Части матрицы, также как и вектор b полностью передавались каждому процессу. Передача начальных данных была оптимизирована через MPI_Pack. Вектор ответа синхронизировался методом MPI_Allgatherv, собирая вектор и отправляя его всем процессам. На профилировании наглядно видно, как происходит синхронизация (фиолетовый цвет)

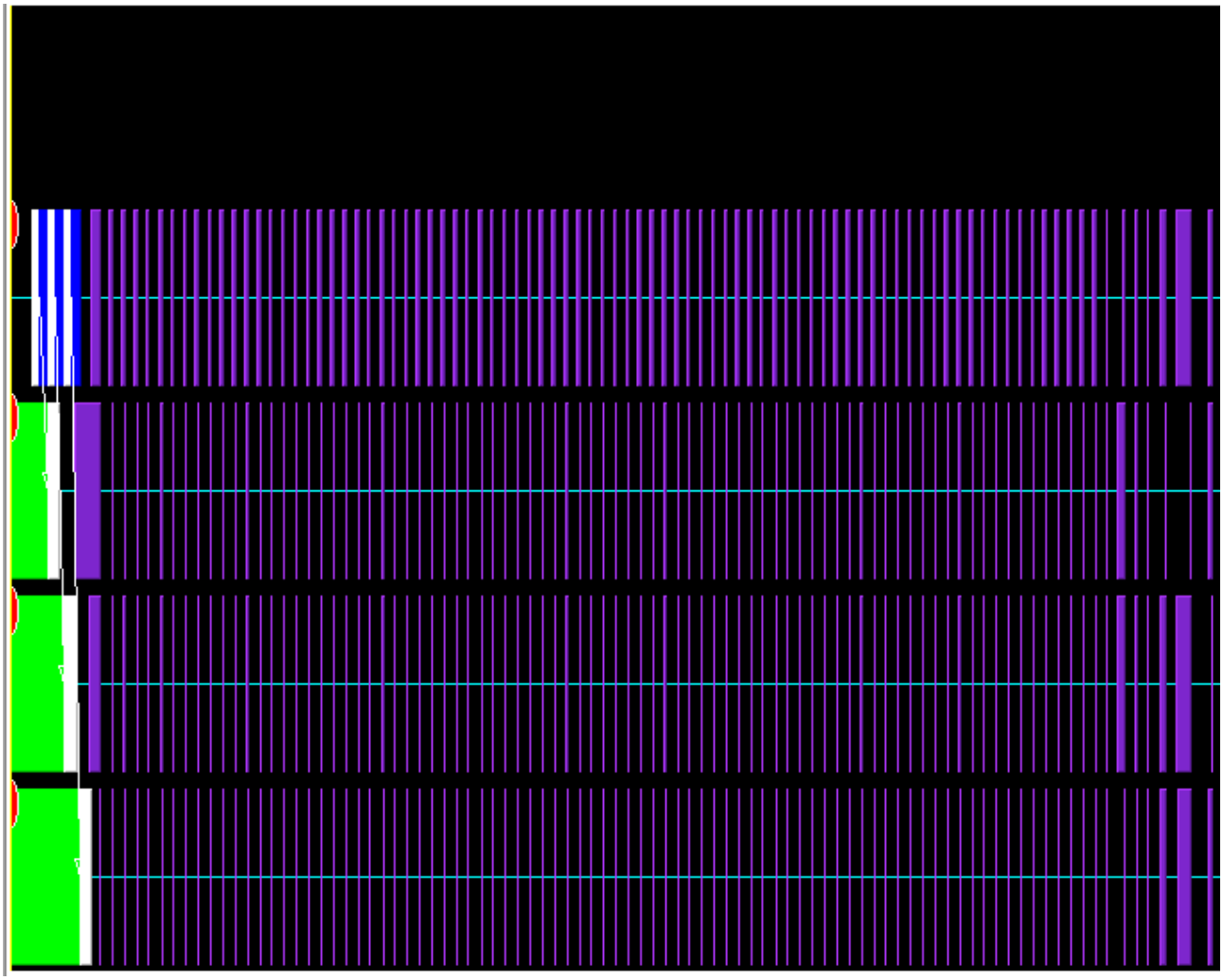


Рисунок 3.1. jumpshot

### 3.2.2. С кольцом

Реализация - (Листинг 3)

При больших размерах матрицы (и вектора соответственно), будет неоптимально хранить и копировать вектор ответов в каждом процессе. Во избежании этого, вектор ответа был разделен на части, которые передаются между процессами по кольцу. В отличе от предыдущей реализации, комуникации между процессами происходят чаще, что отчетливо видно на профилировании:

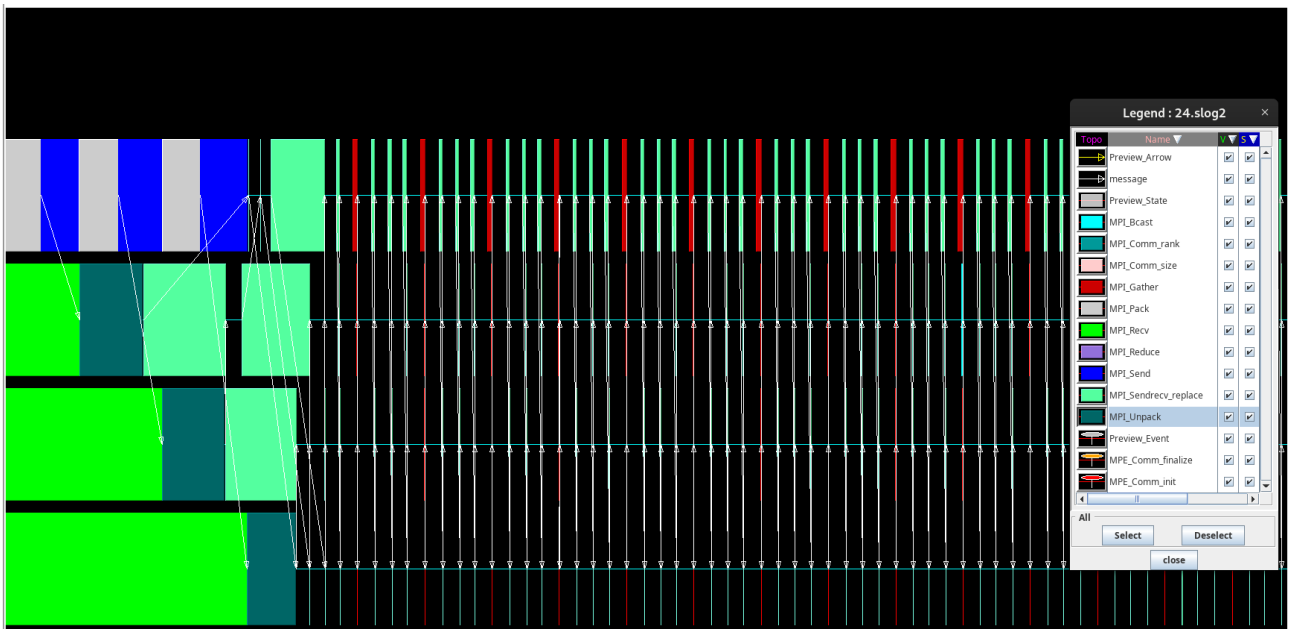



Рисунок 3.2. jumpshot

## 3.3. Сравнение

### 3.3.1. Сравнение двух реализаций

| Количество процессов | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| v1 (Allgatherv) | 21,630861 | 11,256043 | 9,179523 | 6,589661 | 4,616497 |
| v2 (передача x по кольцу) | 20,248594 | 12,457956 | 10,276039 | 6,961421 | 4,324897 |

| Количество процессов | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| V1 ускорение | 1 | 1,92171094229118 | 2,35642538288754 | 3,282545338827 | 4,685557253 |
| V2 ускорение | 1 | 1,62535443213959 | 1,97046683065333 | 2,9086868902197 | 4,681867337 |

| Количество процессов | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| V1 эффективнось | 100,00% | 96,09% | 58,91% | 41,03% | 29,28% |
| V2 Эффективность | 100,00% | 81,27% | 49,26% | 36,36% | 29,26% |

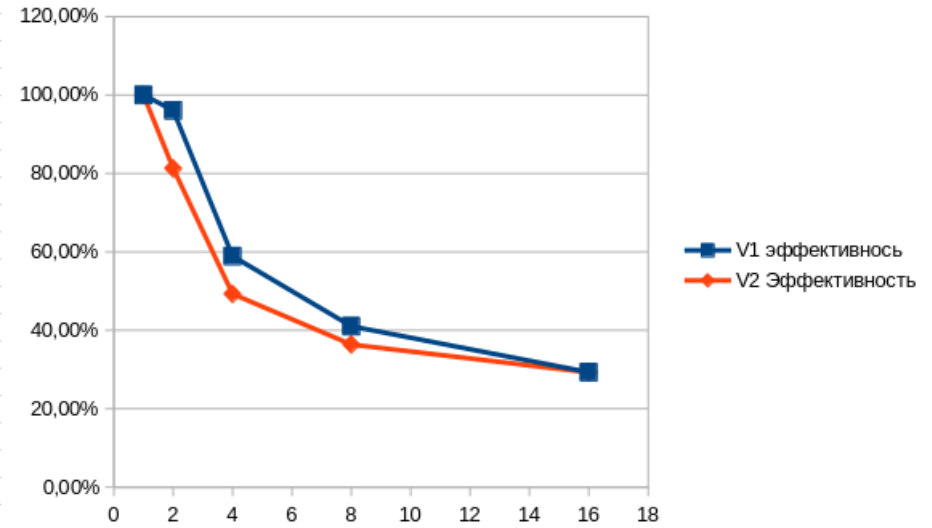



Рисунок 3.3. N = 10k, на одном узле

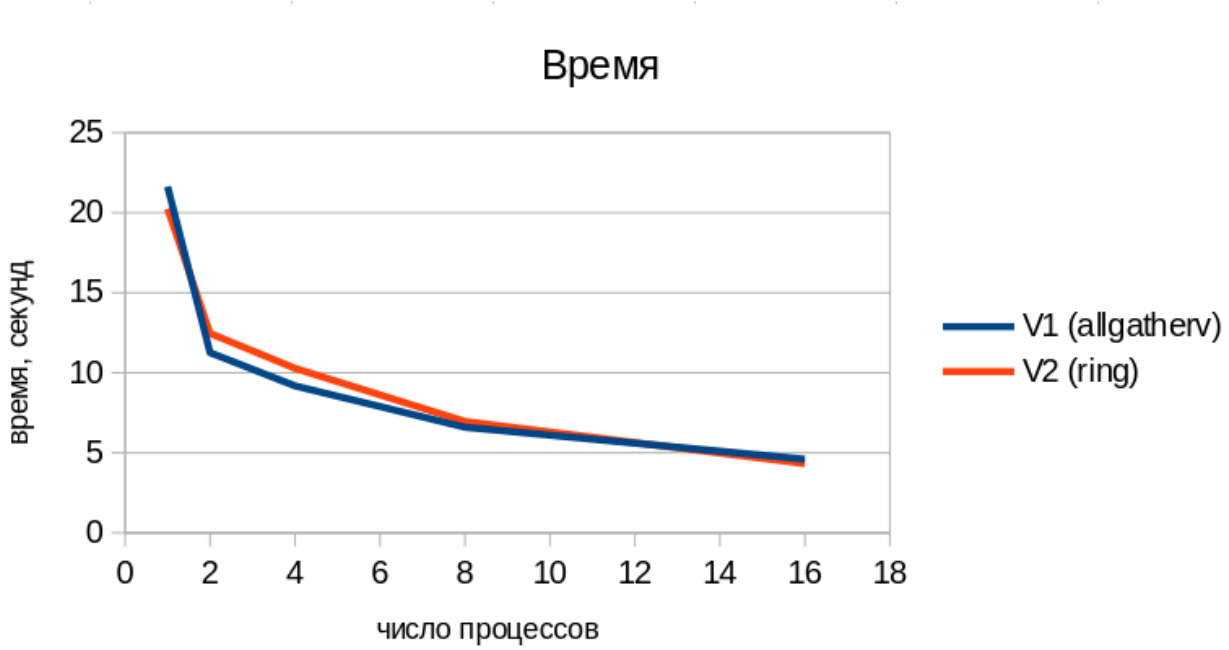



Рисунок 3.4. время: N = 10k, на одном узле

Как видно из графиков, вариант без передачи вектора по кольцу оказался эффективнее. Далее рассмотрена оптимизация именно его.

### 3.3.2. На разных узлах

Была выдвинута гипотеза, что на если запустить каждый MPI процесс на отдельном (по возможности) узле, то получится добиться лучшей эффективности.

| Количество процессов | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| время | 14,099302 | 7,949955 | 4,399675 | 3,914788 | 2,484261 |
| Количество процессов | 1 | 2 | 4 | 8 | 16 |
| ускорение | 1 | 1,77350714563793 | 3,20462352332843 | 3,601549304841 | 5,67545117 |
| Количество процессов | 1 | 2 | 4 | 8 | 16 |
| эффективнось | 100,00% | 88,68% | 80,12% | 45,02% | 35,47% |

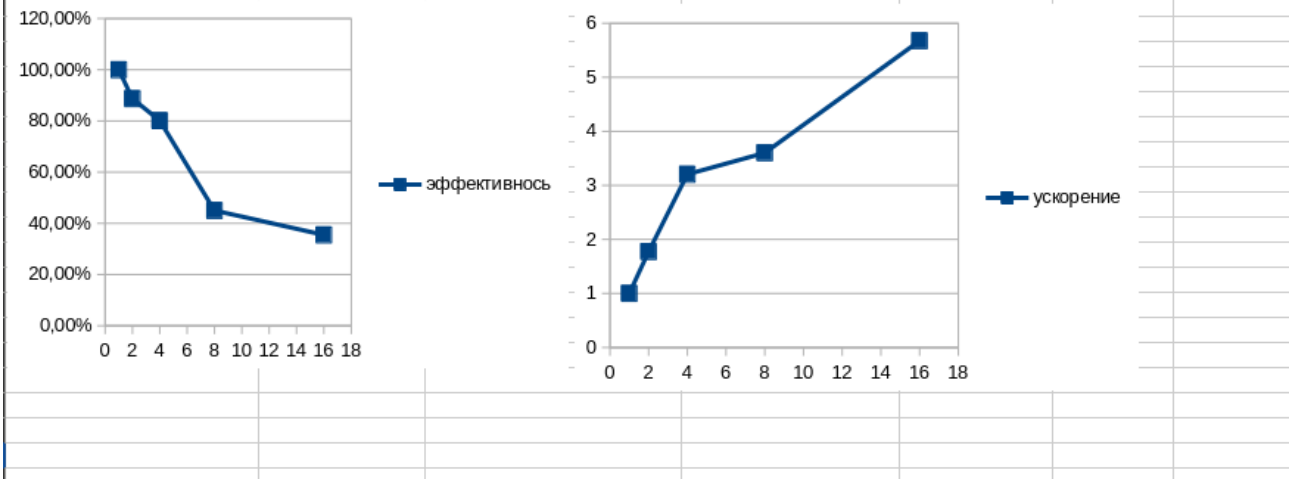



Рисунок 3.5. N = 10k, на разных узлах

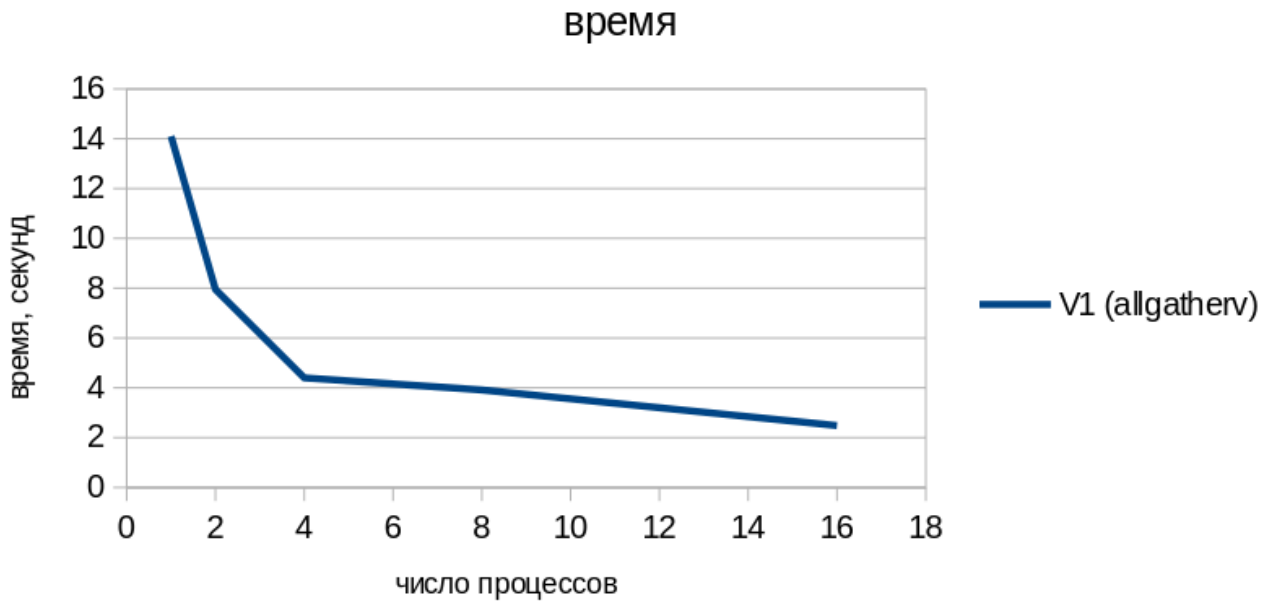



Рисунок 3.6. Время: N = 10k, на разных узлах

Графи показывает, что запуск процесса на отдельном узле дает незначительное (3 %) ускорение.

### 3.3.3. На больших данных

Была выдвинута гипотеза, что на если эффективность низка за счет затраты времени на упаковку данных при инициализации. Для того, чтобы абсорбировать накладные расходы от MPI_Pack.

| Количество процессов | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| v1 (Allgatherv) | 38,467763 | 19,735128 | 13,312279 | 8,385539 | 7,436 |
| Количество процессов | 1 | 2 | 4 | 8 | 16 |
| V1 ускорение | 1 | 1,94920260968158 | 2,889645191480744 | ,5873930107534 | 5,173179532 |
| Количество процессов | 1 | 2 | 4 | 8 | 16 |
| V1 эффективнось | 100,00% | 97,46% | 72,24% | 57,34% | 32,33% |

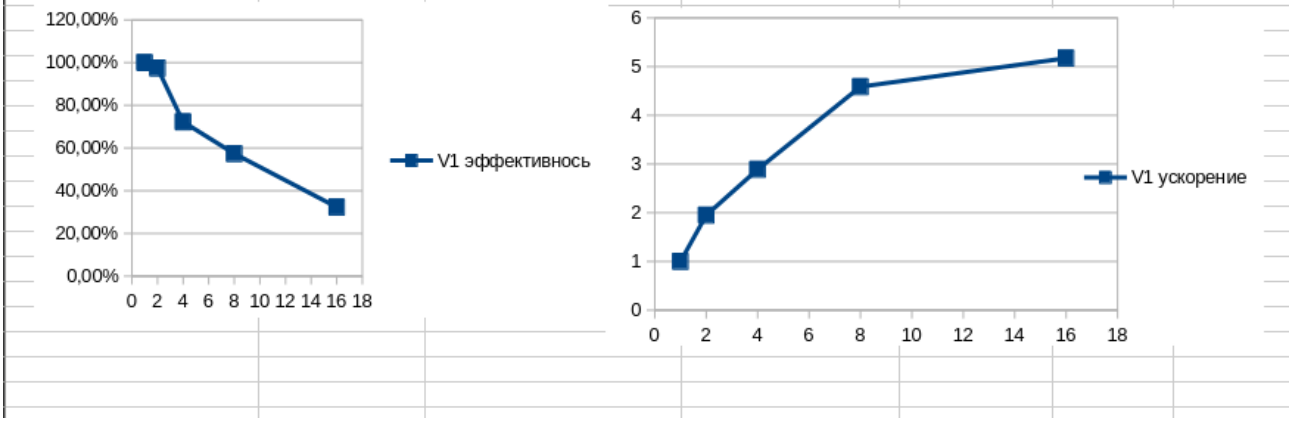



Рисунок 3.7. N = 20k, на одном узле

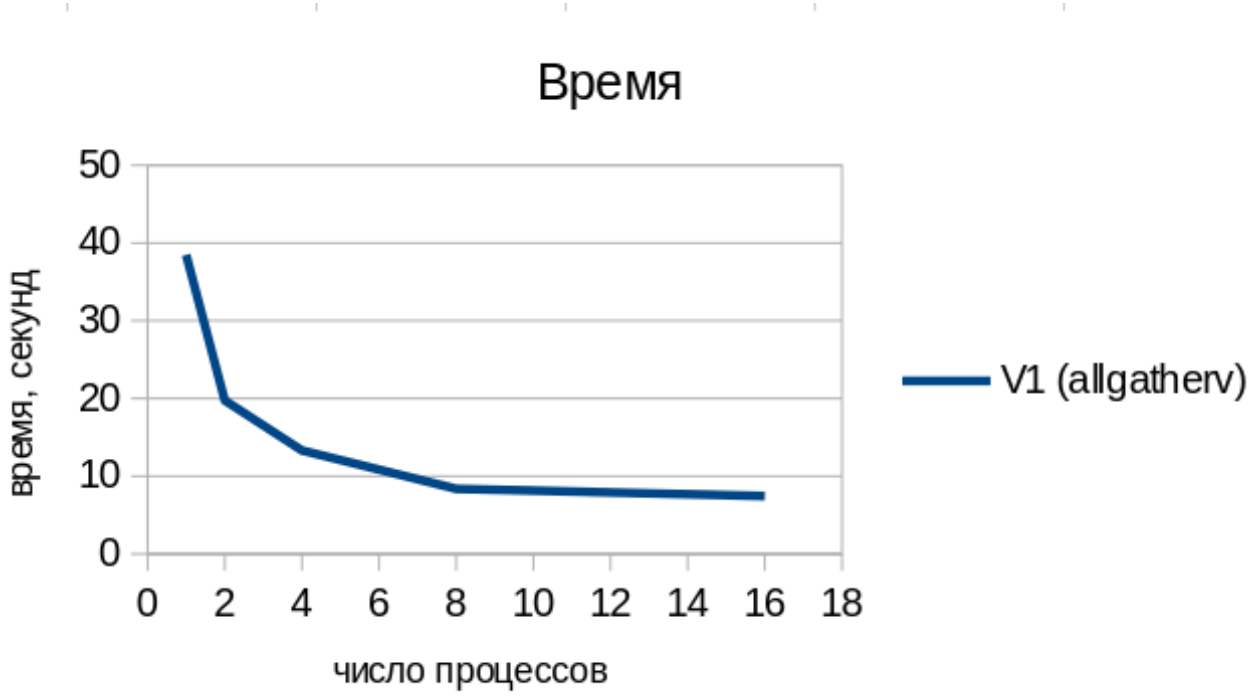
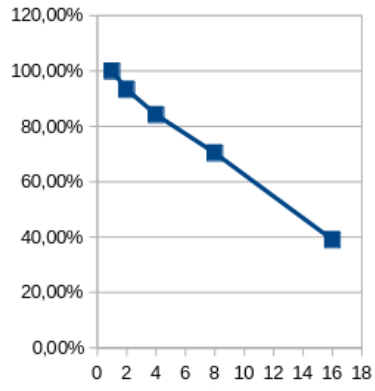



Рисунок 3.8. Время: N = 20k, на одном узле

График показывает увеличение эффективности при увеличении размера вхоных данных.

### 3.3.4. Больше данных, несколько узлов

| Количество процессов | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| v1 (Allgatherv) | 15,515144 | 8,310766 | 4,605961 | 2,755253 | 2,484261 |
| | | | | | |
| Количество процессов | 1 | 2 | 4 | 8 | 16 |
| V1 ускорение | 1 | 1,8668729212205 | 3,368492264697865 | 6,311140936967 | 6,245375989 |
| | | | | | |
| Количество процессов | 1 | 2 | 4 | 8 | 16 |
| V1 эффективнось | 100,00% | 93,34% | 84,21% | 70,39% | 39,03% |





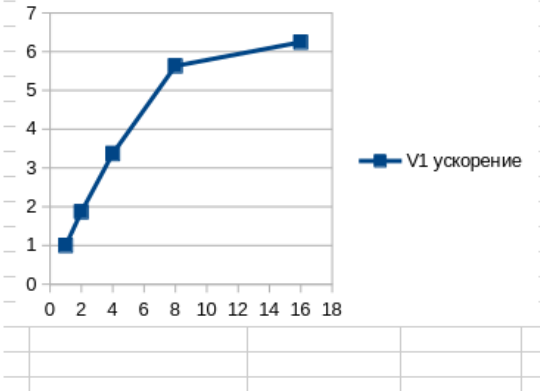



Рисунок 3.9. N = 25k, на отдельных узлах

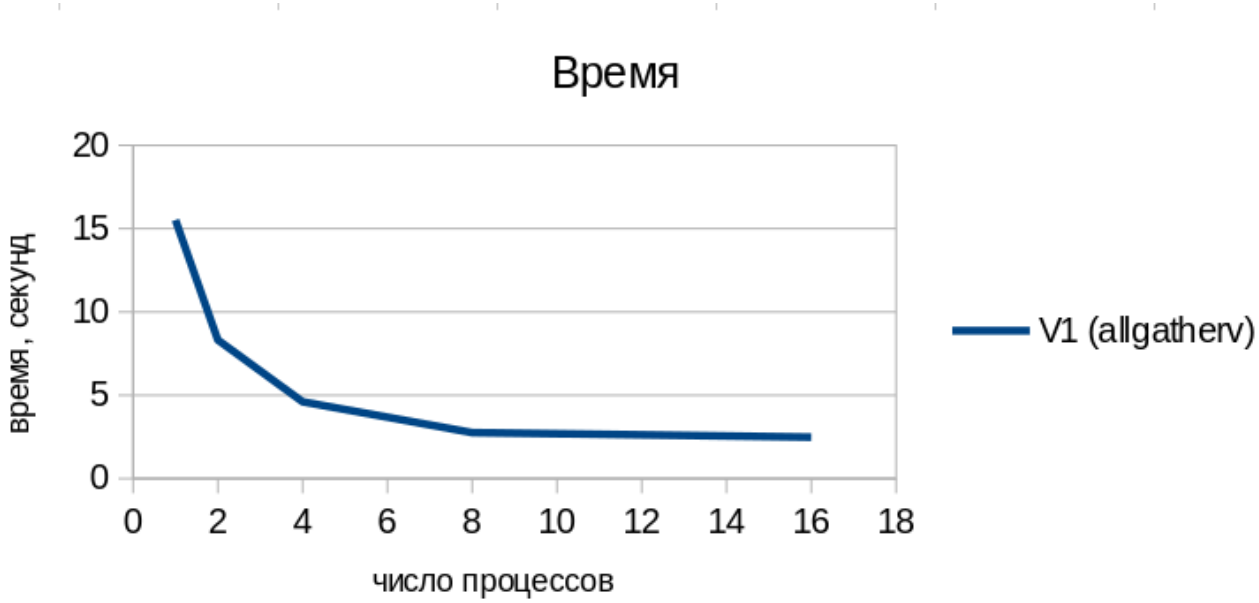



Рисунок 3.10. N = 25k, на отдельных узлах

Комибинируя подходы из предыдущих испытаний удалось добиться эффективности почти в 40% на 16 процессах.

# 4. Заключение

Был реализован алгоритм простой итерации, вычисляющий значения параллельно на нескольких процессах.

# 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 MPI

## 5.3. Листинг 2

```
#include <float.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#ifdef USE_MPE
#include <mpe.h>
#endif

```

```cpp
const int MAX_ITERATIONS = 10000;
const double EPSILON = 0.000001;
const double TAU = 0.00001;
const size_t N = 10000;

void getElapsedTime(double *xVectorNew, int startTime, int size, int rank,
                    int countIter) {
    double endTime = MPI_Wtime();
    double elapsedTime = endTime - startTime;

    double maxTime;
    MPI_Reduce(&elapsedTime, &maxTime, 1, MPI_DOUBLE, MPI_MAX, 0,
               MPI_COMM_WORLD);

    if (rank == 0) {
        printf("%f\n", maxTime);
        printf("\n--------------\n");
        printf("\n%f\n", xVectorNew[0]);
        printf("count iterations = %d\n", countIter);
        printf("Count of MPI process: %d\n", size);
        printf("\n--------------\n");
    }
}

double calc(double *matrix, double *xVector, double *bVector,
            double *xVectorNew, size_t n, double tao, size_t cnt, int first) {
    double res = 0;
#pragma omp parallel for
    for (size_t i = 0; i < cnt; ++i) {
        double s = -bVector[i];
        for (size_t j = 0; j < n; ++j) {
            s += matrix[i * n + j] * xVector[j];
        }
        res += s * s;
        xVectorNew[i] = xVector[first + i] - tao * s;
    }
    return res;
}

int countOfLines(int n, int rank, int size) {
    return n / size + (rank < n % size);
}

void init(double *matrix, double *bVector, double *xVector, int n) {
    for (int i = 0; i < n; ++i) {
        xVector[i] = 0;
        bVector[i] = n + 1;
        for (int j = 0; j < n; ++j) {
            matrix[i * n + j] = 1 + (i == j);
        }
    }
}

/*
 * Конфигурация данных по процессам
 */
void initLinesSettings(int *linesCount, int *firstLines, int size, int n) {
```

```
67      firstLines[0] = 0;
68      linesCount[0] = countOfLines(n, 0, size);
69
70      for (int i = 1; i < size; ++i) {
71          linesCount[i] = countOfLines(n, i, size);
72          firstLines[i] = firstLines[i - 1] + linesCount[i - 1];
73      }
74  }
75
76  double calcEndValue(double *bVector, int n, double eps) {
77      double res = 0;
78      for (int i = 0; i < n; ++i) {
79          res += bVector[i] * bVector[i];
80      }
81      return res * eps * eps;
82  }
83
84  double sumVector(double *vector, int size) {
85      int result = 0;
86      for (int i = 0; i < size; ++i) {
87          result += vector[i];
88      }
89      return result;
90  }
91
92  int solve(double *matrix, double *xVector, double *bVector, double
    ↪  *xVectorNew,
93            int *linesCount, int *firstLines, int rank, int bLen,
94            double prevParam, double tau) {
95      int countIter = 0;
96      int flag = 1;
97
98      for (; flag && (countIter < MAX_ITERATIONS); ++countIter) {
99          double dd = calc(matrix, xVector, bVector, xVectorNew, N, tau,
100                           (size_t)linesCount[rank],
                             ↪  (size_t)firstLines[rank]);
101         // Собираем вектор по кусочкам
102         MPI_Allgatherv(xVectorNew, linesCount[rank], MPI_DOUBLE,
            ↪  xVector,
103                        linesCount, firstLines, MPI_DOUBLE,
                           ↪  MPI_COMM_WORLD);
104         double endParam;
105         MPI_Reduce(&dd, &endParam, 1, MPI_DOUBLE, MPI_SUM, 0,
            ↪  MPI_COMM_WORLD);
106
107         if (rank == 0) {
108             if (prevParam ⩽ endParam || endParam ⩽ bLen) {
109                 flag = 0;
110             }
111             prevParam = endParam;
112             flag = flag && (endParam > bLen);
113         }
114         MPI_Bcast(&flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
115     }
116     return countIter;
117 }
118
119 int main(int argc, char **argv) {
120     int size, rank;
121     MPI_Init(&argc, &argv);
```

```
122 #ifdef USE_MATH_LIB
123     MPE_Init_log();
124 #endif
125     MPI_Comm_size(MPI_COMM_WORLD, &size);
126     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
127 
128     double startTime = MPI_Wtime();
129 
130     /* Считаем конфигурацию себя и других */
131     int *linesCount = (int *)malloc(size * sizeof(int));
132     int *firstLines = (int *)malloc(size * sizeof(int));
133     initLinesSettings(linesCount, firstLines, size, N);
134     size_t currentSize = (rank == 0 ? N : (size_t)linesCount[rank]);
135     /* выделение памяти */
136     double *matrix = (double *)malloc(sizeof(double) * N *
        ↪ currentSize);
137     double *xVector = (double *)malloc(sizeof(double) * N);
138     double *bVector = (double *)malloc(sizeof(double) * currentSize);
139     double *xVectorNew = (double *)calloc(linesCount[rank],
        ↪ sizeof(double));
140     int buffSize = sizeof(double) * (linesCount[0] * N + N +
        ↪ linesCount[0]);
141     void *buff = (void *)malloc(buffSize);
142 
143     if (rank == 0) {
144         init(matrix, bVector, xVector, N);
145         for (int i = 1; i < size; ++i) {
146             int pos = 0;
147             MPI_Pack(matrix + N * firstLines[i], N * linesCount[i],
                ↪ MPI_DOUBLE,
148                      buff, buffSize, &pos, MPI_COMM_WORLD);
149             MPI_Pack(xVector, (int)N, MPI_DOUBLE, buff, buffSize, &pos,
150                      MPI_COMM_WORLD);
151             MPI_Pack(bVector + firstLines[i], linesCount[i],
                ↪ MPI_DOUBLE, buff,
152                      buffSize, &pos, MPI_COMM_WORLD);
153             MPI_Send(buff, buffSize, MPI_BYTE, i, 0, MPI_COMM_WORLD);
154         }
155     } else {
156         int pos = 0;
157         MPI_Recv(buff, buffSize, MPI_BYTE, 0, 0, MPI_COMM_WORLD,
158                  MPI_STATUS_IGNORE);
159         MPI_Unpack(buff, buffSize, &pos, matrix, N * linesCount[rank],
160                    MPI_DOUBLE, MPI_COMM_WORLD);
161         MPI_Unpack(buff, buffSize, &pos, xVector, (int)N, MPI_DOUBLE,
162                    MPI_COMM_WORLD);
163         MPI_Unpack(buff, buffSize, &pos, bVector, linesCount[rank],
            ↪ MPI_DOUBLE,
164                    MPI_COMM_WORLD);
165     }
166     free(buff);
167 
168     double bLen = (rank == 0) ? calcEndValue(bVector, (int)N, EPSILON)
        ↪ : 0;
169     double prevParam = DBL_MAX;
170 
171     int countIter = solve(matrix, xVector, bVector, xVectorNew,
        ↪ linesCount,
172                           firstLines, rank, bLen, prevParam, TAU);
```

```
173         countIter += solve(matrix, xVector, bVector, xVectorNew,
            ↪  linesCount,
174                            firstLines, rank, bLen, prevParam, -TAU);
175 
176         getElapsedTime(xVectorNew, startTime, size, rank, countIter);
177 
178         free(matrix);
179         free(xVector);
180         free(xVectorNew);
181         free(bVector);
182         free(linesCount);
183         free(firstLines);
184 
185         MPI_Finalize();
186         return 0;
187     }
```

Листинг 2: MPI, нет кольца

## 5.4. Листинг 3

```
1   #include <float.h>
2   #include <mpi.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5   
6   #ifdef USE_MPE
7   #include <mpe.h>
8   #endif
9   
10  const int MAX_ITERATIONS = 10000;
11  const double EPSILON = 0.000001;
12  const double TAU = 0.00001;
13  const int N = 45000;
14  
15  /*
16   * countOfLine
17   * Возращает количество строчек в процессе с номером rank
18   */
19  int countOfLines(int n, int rank, int size) {
20      return n / size + (rank < n % size);
21  }
22  
23  /*
24   * Генерация данных
25   */
26  void init(double *matrix, double *bVector, double *xVector, int n) {
27      for (int i = 0; i < n; ++i) {
28          xVector[i] = 0;
29          bVector[i] = n + 1;
30          for (int j = 0; j < n; ++j) {
31              matrix[i * n + j] = 1 + (i == j);
32          }
33      }
34  }
35  
```

```
36  /*
37   * Конфигурация данных по процессам
38   */
39  void initLinesSettings(int *linesCount, int *firstLines, int size, int
    ↪  n) {
40      firstLines[0] = 0;
41      linesCount[0] = countOfLines(n, 0, size);
42
43      for (int i = 1; i < size; ++i) {
44          linesCount[i] = countOfLines(n, i, size);
45          firstLines[i] = firstLines[i - 1] + linesCount[i - 1];
46      }
47  }
48
49  double calcEndValue(double *bVector, int n, double eps) {
50      double res = 0;
51      for (int i = 0; i < n; ++i) {
52          res += bVector[i] * bVector[i];
53      }
54      return res * eps * eps;
55  }
56
57  double sumVector(double *vector, int size) {
58      int result = 0;
59      for (int i = 0; i < size; ++i) {
60          result += vector[i];
61      }
62      return result;
63  }
64
65  int main(int argc, char **argv) {
66      int size, rank;
67      MPI_Init(&argc, &argv);
68  #ifdef USE_MATH_LIB
69      MPE_Init_log();
70  #endif
71      MPI_Comm_size(MPI_COMM_WORLD, &size);
72      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
73
74      double startTime = MPI_Wtime();
75
76      /* Считаем конфигурацию себя и других */
77      int *linesCount = (int *)malloc(size * sizeof(int));
78      int *firstLines = (int *)malloc(size * sizeof(int));
79      initLinesSettings(linesCount, firstLines, size, N);
80      int currentSize = (rank == 0 ? N : linesCount[rank]);
81      /* выделение памяти */
82      double *matrix = (double *)malloc(sizeof(double) * N *
        ↪  currentSize);
83      double *xVector = (double *)malloc(sizeof(double) * N);
84      double *bVector = (double *)malloc(sizeof(double) * currentSize);
85      double *xVectorNew = (double *)calloc(linesCount[rank],
        ↪  sizeof(double));
86      double *sVector = malloc(linesCount[0] * sizeof(double));
87      double *xVectorPart = (double *)malloc(linesCount[0] *
        ↪  sizeof(double));
88      /*
89      int buffSize = sizeof(double) * (linesCount[0] * N + N +
        ↪  linesCount[0]);
```

```
90          void *buff = (void *)malloc(buffSize);
91      */
92          if (rank == 0) {
93              init(matrix, bVector, xVector, N);
94              for (int i = 1; i < size; ++i) {
95                  // int pos = 0;
96                  MPI_Send(matrix + N * firstLines[i], N * firstLines[i],
                     ↪  MPI_DOUBLE,
97                           i, 0, MPI_COMM_WORLD);
98                  MPI_Send(xVector, N, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
99                  MPI_Send(bVector + firstLines[i], linesCount[i],
                     ↪  MPI_DOUBLE, i, 0,
100                          MPI_COMM_WORLD);
101             }
102         } else {
103             // int pos = 0;
104             MPI_Recv(matrix, N * linesCount[rank], MPI_DOUBLE, 0, 0,
                     ↪  MPI_COMM_WORLD,
105                      MPI_STATUS_IGNORE);
106             MPI_Recv(xVector, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
107                      MPI_STATUS_IGNORE);
108             MPI_Recv(bVector, linesCount[rank], MPI_DOUBLE, 0, 0,
                     ↪  MPI_COMM_WORLD,
109                      MPI_STATUS_IGNORE);
110         }
111         // free(buff);
112
113         double bLen = (rank == 0) ? calcEndValue(bVector, N, EPSILON) : 0;
114
115         double *d = (double *)malloc(sizeof(double) * ((rank == 0) ? size :
             ↪  1));
116
117         int flag = 1;
118         int useTau = 0;
119         double tau = TAU;
120         double prevParam = DBL_MAX;
121         int countIter = 0;
122
123         int right_rank = (rank + 1) % size;
124         int left_rank = (rank - 1 + size) % size;
125
126         for (; flag && (countIter < MAX_ITERATIONS); ++countIter) {
127             double dd = 0;
128             for (int i = 0; i < linesCount[rank]; ++i) {
129                 sVector[i] = -bVector[i];
130                 xVectorPart[i] = xVectorNew[i];
131             }
132             for (int partId = 0; partId < size; ++partId) {
133                 // copy answer vector to copys;
134                 MPI_Sendrecv_replace(xVectorPart, linesCount[0],
                     ↪  MPI_DOUBLE,
135                                      left_rank, 0, right_rank, 0,
                                         ↪  MPI_COMM_WORLD,
136                                      MPI_STATUS_IGNORE);
137                 for (int i = 0; i < linesCount[rank]; ++i) {
138                     for (int j = 0; j < linesCount[partId]; ++j) {
139                         sVector[i] +=
140                             matrix[i * N + j + firstLines[partId]] *
                                 ↪  xVectorPart[j];
141                     }
```

```
142                 }
143             }
144             for (int i = 0; i < linesCount[rank]; ++i) {
145                 dd += sVector[i] * sVector[i];
146                 xVectorNew[i] = xVectorPart[i] - tau * sVector[i];
147             }
148 
149             MPI_Gather(&dd, 1, MPI_DOUBLE, d, 1, MPI_DOUBLE, 0,
                ↪ MPI_COMM_WORLD);
150 
151             if (rank == 0) {
152                 double endParam = 0;
153                 for (int i = 0; i < size; ++i) {
154                     endParam += d[i];
155                 }
156                 if (prevParam ⩽ endParam) {    // условие смены знака
                    ↪ скаляра.
157                     if (useTau) {
158                         flag = 0;    // Очевидно, что на прямой к числу
                            ↪ можно
159                     } else {
160                         tau *= -1;
161                         useTau = 1;
162                     }
163                 }
164                 prevParam = endParam;
165                 flag = flag && (endParam > bLen);
166             }
167 
168             MPI_Bcast(&tau, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
169             MPI_Bcast(&flag, 1, MPI_INT, 0,
170                       MPI_COMM_WORLD);    // Выходим из цикла всем
                        ↪ процессам
171         }
172 
173         double endTime = MPI_Wtime();
174         double elapsedTime = endTime - startTime;
175 
176         double maxTime;
177         MPI_Reduce(&elapsedTime, &maxTime, 1, MPI_DOUBLE, MPI_MAX, 0,
178                    MPI_COMM_WORLD);
179 
180         if (rank == 0) {
181             printf("%f\n", maxTime);
182             printf("\n--------------\n");
183             printf("\n%f\n", xVectorNew[0]);    // достаточно вывести один
184             printf("count iterations = %d\n", countIter);
185             printf("Count of MPI process: %d\n", size);
186             printf("\n--------------\n");
187         }
188 
189         free(matrix);
190         free(xVector);
```

Листинг 3: Передача вектора-решения по кольцу