Санкт-Петербургский политехнический университет Петра Великого
Институт прикладной математики и механики
Высшая школа прикладной математики и вычислительной физики

Введение в технологии суперкомпьютерных вычислений

Отчёт по лабораторной работе №5
Распараллеливание классических алгоритмов решения систем линейных алгебраических уравнений

Работу выполнил:
Цопанов М. Т.
Группа:
5030301/00102
Преподаватель:
Гатаулин Я. А.

Санкт-Петербург
2023

Содержание

1. Цель работы

С помощью технологии OpenMP реализовать программы, выполняющие распараллеливание прямых и классических итерационных методов решения систем линейных алгебраических уравнений.

2. Задание к работе

Разработать программу, параллелизующую средствами OpenMP заданный метод решения системы линейных алгебраических уравнений (метод Гаусса-Жордана). Исследовать эффективность параллелизации.

3. Характеристики компьютера

Количество ядер: 48, объём оперативной памяти: 128 Гб.

4. Исследование последовательной версии программы

Разработана последовательная версия программы для сравнения с тестовым решением системы 3-х линейных алгебраических уравнений методом Гаусса-Жордана. Вывод расширенной матрицы и вектора решений представлены на рисунке 4.1.

```
mts19@fedora Лабораторная работа 5  ✗ gcc -o program.out -fopenmp main.c module.c
mts19@fedora Лабораторная работа 5  ✗ export OMP_NUM_THREADS=1
mts19@fedora Лабораторная работа 5  ✗ ./program.out
0.840188        0.394383        0.783099        0.553970
0.798440        0.911647        0.197551        0.477397
0.335223        0.768230        0.277775        0.628871

-0.410866
0.712443
0.789427
Time = 0.000035 seconds.
```

Рисунок 4.1. Результат работы последовательной версии программы.

Для того чтобы удостовериться в том, что программа работает верно, решим данную систему в математическом пакете GNU Octave. В левом окне код, а в правом вектор решений (рисунке 4.2).

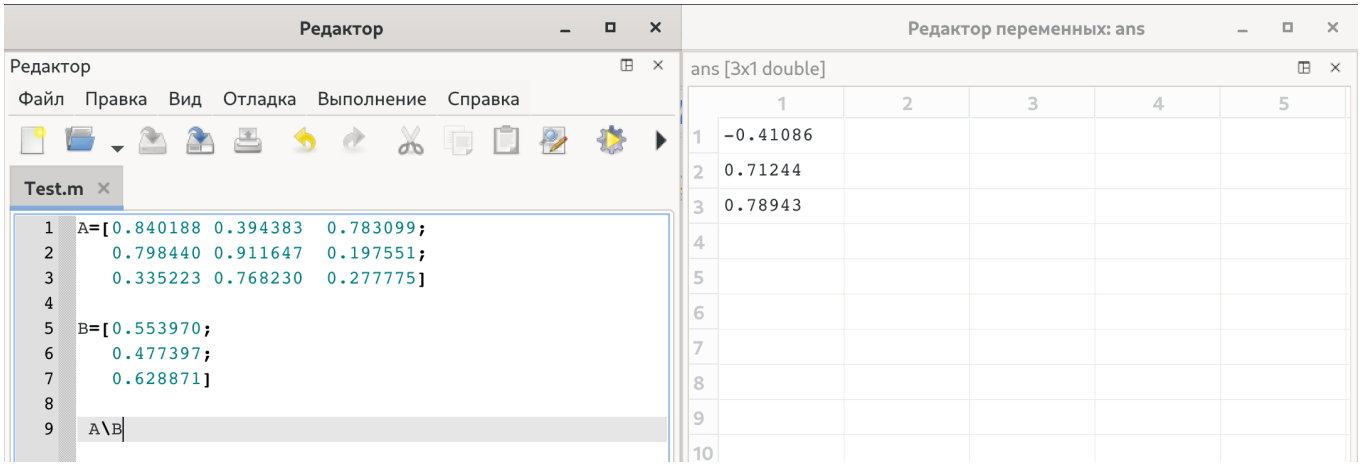

Рисунок 4.2. Результат решения СЛАУ в GNU Octave.

Как видно, решения совпадают, поэтому можем приступить к исследованию параллельной версии программы.

5. Исследование параллельной версии программы

Также сравним с тестовым решением системы 3-х линейных алгебраических уравнений методом Гаусса-Жордана параллельный алгоритм при $TN = 8$ (рисунок 5.1). Как видно, программа работает верно.

```
mts19@fedora Лабораторная работа 5  ✗ gcc -o program.out -fopenmp main.c module.c
mts19@fedora Лабораторная работа 5  ✗ export OMP_NUM_THREADS=8
mts19@fedora Лабораторная работа 5  ✗ ./program.out
0.840188        0.394383        0.783099        0.553970
0.798440        0.911647        0.197551        0.477397
0.335223        0.768230        0.277775        0.628871

-0.410866
0.712443
0.789427
Time = 0.000903 seconds.
```

Рисунок 5.1. Результат работы параллельной версии программы для $TN = 8$.

Приступим к изучению влияния различных условий на время работы параллельной программы. Первый эксперимент направлен на изучение влияния размерности матрицы на время выполнения при постоянном числе нитей $TN = 8$ (рисунок 5.2). Как видно, при увеличении размерности матрицы увеличивается и время работы.

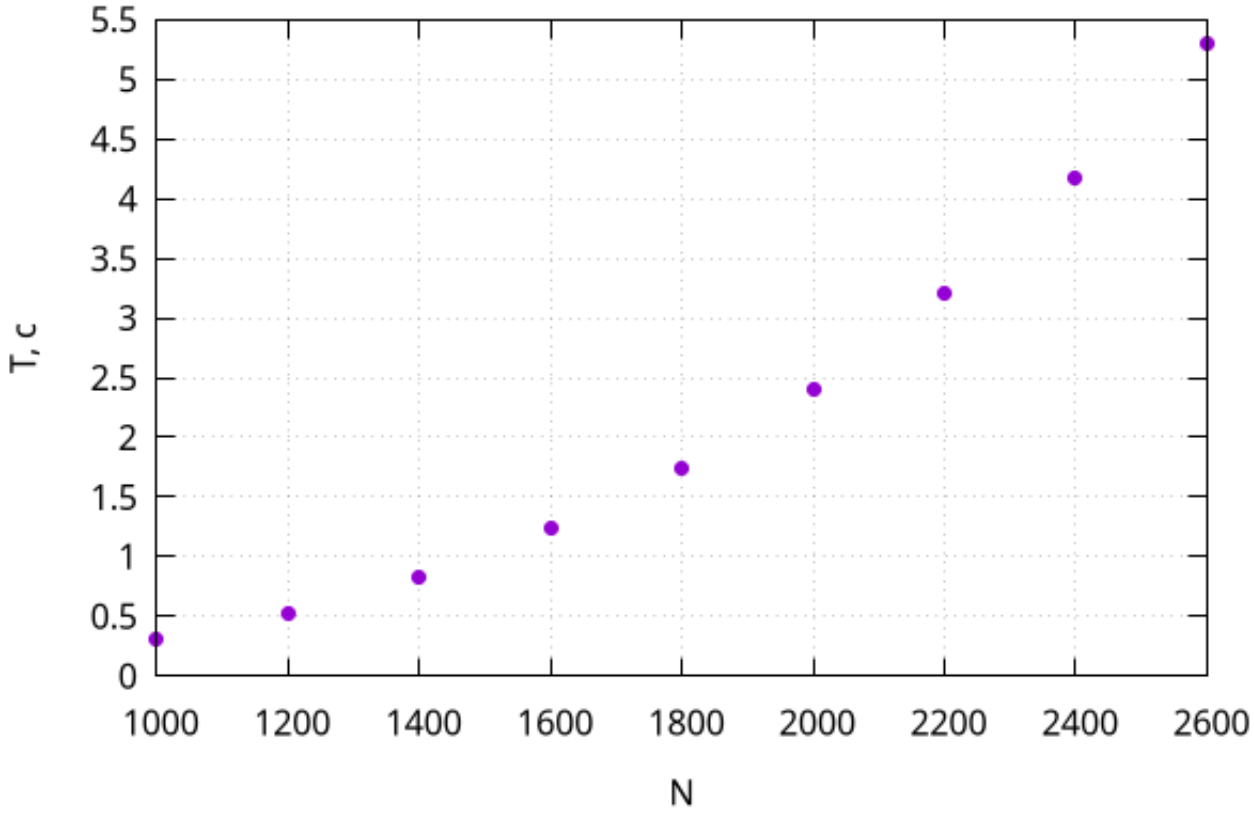

Рисунок 5.2. Зависимость времени выполнения программы от размерности матрицы для числа нитей $TN = 8$.

Теперь возьмём за основу размерность матрицы 1000x1000 и для неё исследуем зависимость времени выполнения, ускорения и эффективности от количества нитей (рисунок 5.3, рисунок 5.4, рисунок 5.5). Исследуя зависимости видно, что сначала увеличение количества нитей существенно влияет на скорость выполнения программы, но потом наступает некоторое насыщение и большого выигрыша не происходит.

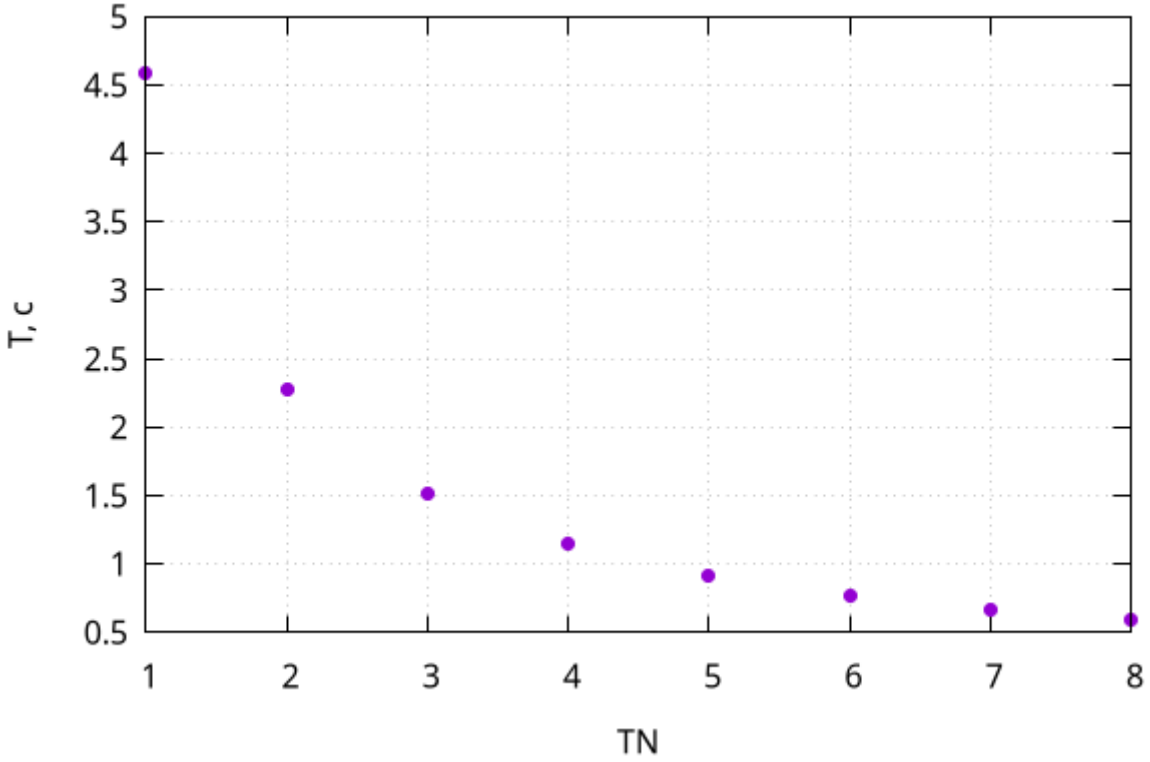

Рисунок 5.3. Зависимость времени выполнения программы от числа нитей для размерности матрицы 1000x1000.

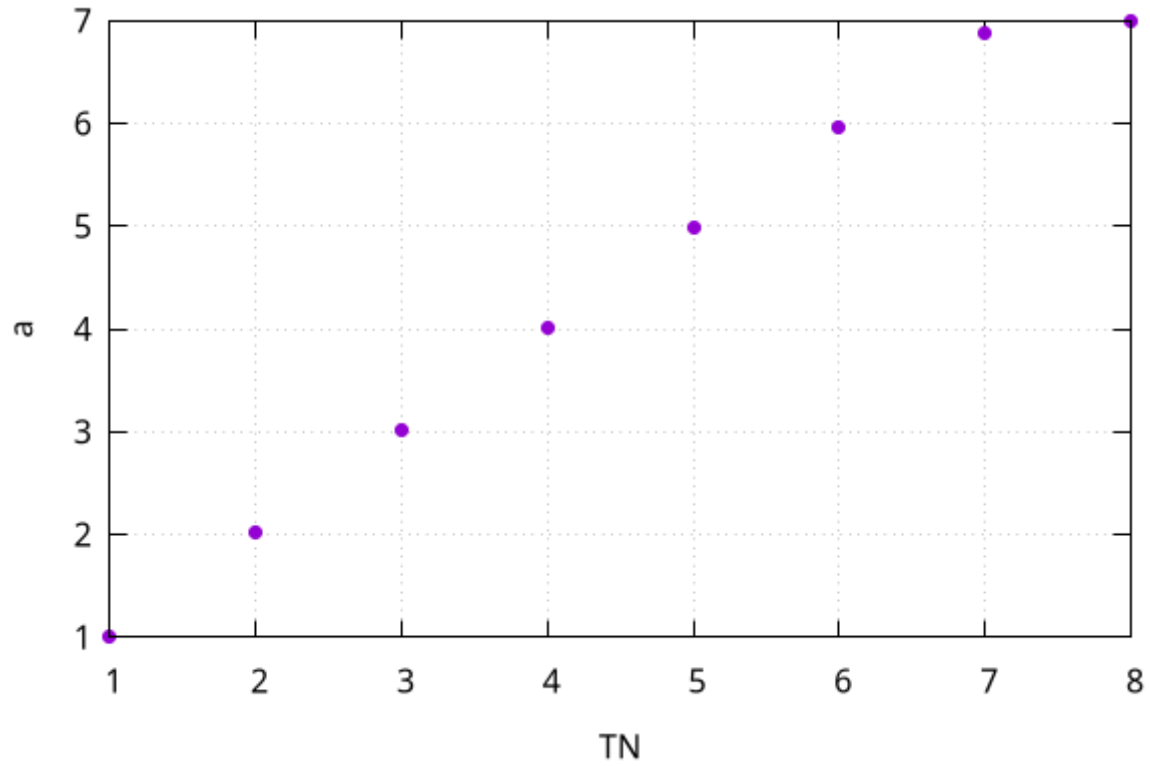

Рисунок 5.4. Зависимость ускорения ($a = \frac{T_1}{T_{tn}}$) выполнения программы от числа нитей для размерности матрицы 1000x1000.

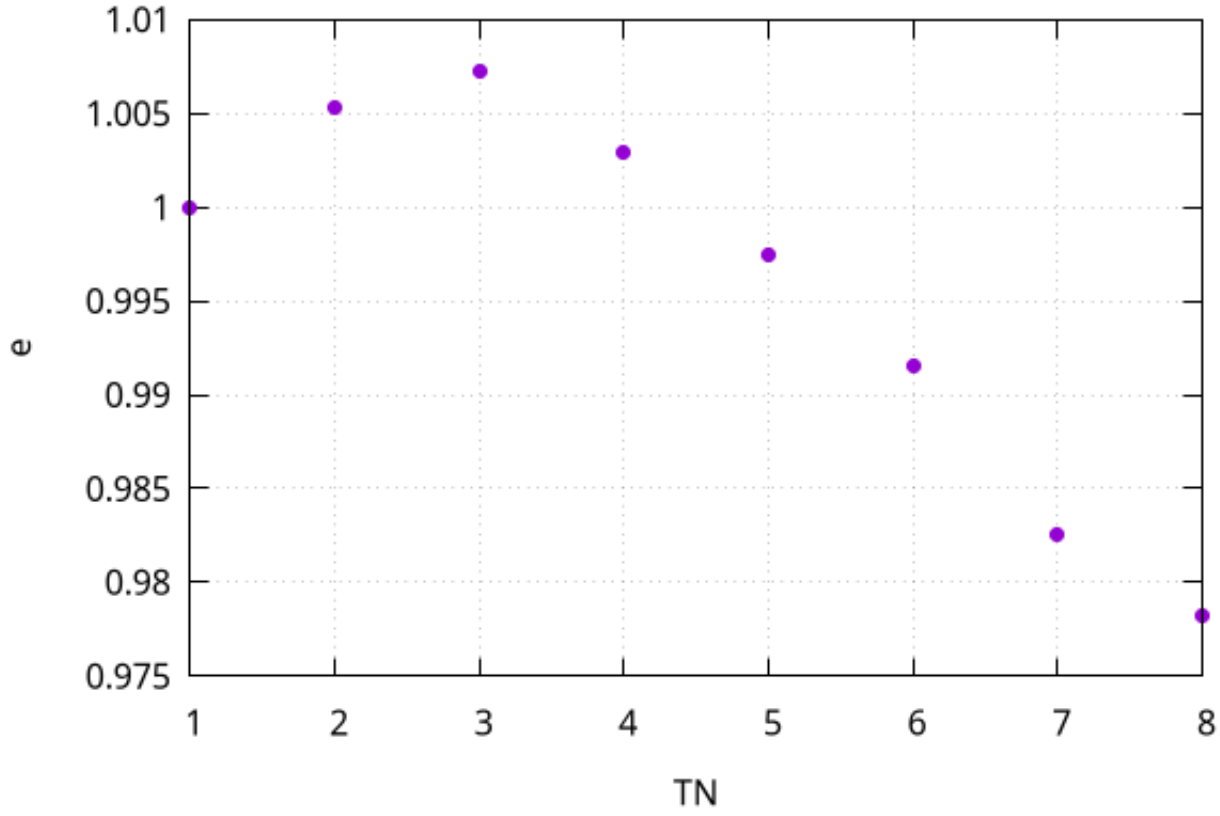

Рисунок 5.5. Зависимость эффективности ($e = \frac{a}{TN}$) выполнения программы от числа нитей для размерности матрицы 1000x1000.

Теперь, используя $TN = 8$ и размерность 1000x1000, исследуем влияние оптимизатора и способа распределения итераций между ниятми (таблица 5.1, таблица 5.2). Использование опций оптимизации компилятора уменьшает время работы, однако при переходе с первого уровня на третий разница почти незаметна, а способ распределения итераций между нитями не оказывает сильного влияния на время выполнения программы.

	O0	O1	O3
t, сек	0.5863	0.3804	0.3779

Таблица 5.1

Зависимость времени выполнения от опций оптимизации компилятора для $TN = 8$ и размерности матрицы 1000x1000.

	STATIC	DYNAMIC, 500	GUIDED
t, сек	0.5956	0.6232	0.6176

Таблица 5.2

Зависимость времени выполнения от способа распределения итераций между нитями для $TN = 8$ и размерности матрицы 1000x1000.

6. Вывод

В данной работе было реализовано расспаралеливание программы для решения СЛАУ методом Жордана-Гаусса и исследование зависимости эффективности работы программы от различных опций оптимизации и ускорений, предоставленных директивой OpenMP. Основными выводами являются следующие факты:

1. При увеличении размерности матрицы время выполнения программы увеличивается экспоненциально, это связано с тем, что данный метод является точным.

2. С увеличением количества нитей ускорение растёт, а эффективность увеличивается до 3 нити, после падает.

3. Способ распределения итераций между циклами не оказывает сильного влияния на время выполнения программы

4. Опции оптимизатора ускоряют время выполнения программы примерно в 1,5 раза.

7. Коды программ

main.c

```
f main
#include "module.h"
#include <math.h>
#include <omp.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#define N 2000

int main(void) {
  omp_set_dynamic(0);
  double_t **A = random_matrix_alloc(N, N);
  double_t *B = random_vector_alloc(N);
   for (size_t i = 0; i < N; i++) {
   for (size_t j = 0; j < N; j++) {
     printf("%lf\t", A[i][j]);
   }
   printf("%lf\n", B[i]);
  }
   printf("\n");
  double_t start_time = omp_get_wtime();
  for (size_t i = 0; i < N - 1; i++) {
#pragma omp parallel default(none) shared(A, B, i)
  {
#pragma omp for nowait
      for (size_t j = i + 1; j < N; j++) {
        A[i][j] /= A[i][i];
      }
#pragma omp single
      B[i] /= A[i][i];
#pragma omp single
      A[i][i] = 1.0;
#pragma omp for collapse(2) nowait
      for (size_t k = i + 1; k < N; k++) {
        for (size_t j = i + 1; j < N; j++) {
          A[k][j] -= A[k][i] * A[i][j];
        }
      }
#pragma omp for
      for (size_t k = i + 1; k < N; k++) {
        B[k] -= B[i] * A[k][i];
      }
#pragma omp for
      for (size_t k = i + 1; k < N; k++) {
        A[k][i] = 0.0;
      }
    }
  }
  B[N - 1] /= A[N - 1][N - 1];
  A[N - 1][N - 1] = 1.0;
  for (size_t i = N - 1; i > 0; i--) {
#pragma omp parallel for shared(A, B, i)
    for (size_t k = i - 1; k < SIZE_MAX; k--) {
      B[k] -= B[i] * A[k][i];
    }
  }
  double_t end_time = omp_get_wtime();
   for (size_t i = 0; i < N; i++) {
   printf("%lf\n", B[i]);
  }
  printf("Time - %lf seconds.\n", end_time - start_time);
  free(A);
  free(B);

  return EXIT_SUCCESS;
}
```

Рисунок 7.1. Тестовая программа.

```
f main
#include "module.h"
#include <math.h>
#include <omp.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#define N 2000

int main(void) {
  omp_set_dynamic(0);
  double_t **A = random_matrix_alloc(N, N);
  double_t *B = random_vector_alloc(N);
  double_t start_time = omp_get_wtime();
  for (size_t i = 0; i < N - 1; i++) {
#pragma omp parallel default(none) shared(A, B, i)
    {
#pragma omp for nowait
      for (size_t j = i + 1; j < N; j++) {
        A[i][j] /= A[i][i];
      }
#pragma omp single
      B[i] /= A[i][i];
#pragma omp single
      A[i][i] = 1.0;
#pragma omp for collapse(2) nowait
      for (size_t k = i + 1; k < N; k++) {
        for (size_t j = i + 1; j < N; j++) {
          A[k][j] -= A[k][i] * A[i][j];
        }
      }
#pragma omp for
      for (size_t k = i + 1; k < N; k++) {
        B[k] -= B[i] * A[k][i];
      }
#pragma omp for
      for (size_t k = i + 1; k < N; k++) {
        A[k][i] = 0.0;
      }
    }
  }
  B[N - 1] /= A[N - 1][N - 1];
  A[N - 1][N - 1] = 1.0;
  for (size_t i = N - 1; i > 0; i--) {
#pragma omp parallel for shared(A, B, i)
    for (size_t k = i - 1; k < SIZE_MAX; k--) {
      B[k] -= B[i] * A[k][i];
    }
  }
  double_t end_time = omp_get_wtime();
  printf("Time = %lf seconds.\n", end_time - start_time);
  free(A);
  free(B);

  return EXIT_SUCCESS;
}
```

Рисунок 7.2. Параллельный алгоритм.

```c
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define drand() ((double_t)rand() / (RAND_MAX + 1.0))

void *Malloc(size_t size) {
  void *ptr = malloc(size);
  if (ptr == NULL) {
    perror("malloc failed");
    exit(EXIT_FAILURE);
  }
  return ptr;
}

double_t **random_matrix_alloc(size_t nrows, size_t ncolumns) {
  double_t **matrix = (double_t **)Malloc(
      nrows * (sizeof(double_t *) + ncolumns * sizeof(double_t)));
  double_t *start = (double_t *)((char *)matrix + nrows * sizeof(double_t *));
  for (size_t i = 0; i < nrows; i++) {
    matrix[i] = start + i * ncolumns;
  }
  for (size_t i = 0; i < nrows; i++) {
    for (size_t j = 0; j < ncolumns; j++) {
      matrix[i][j] = drand();
    }
  }
  return matrix;
}

double_t *random_vector_alloc(size_t nrows) {
  double_t *vector = (double_t *)Malloc(nrows * sizeof(double_t));
  for (size_t i = 0; i < nrows; i++) {
    vector[i] = drand();
  }
  return vector;
}
```

Рисунок 7.3. Модуль.