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:
Тептев М. А.
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 решение системы линейных алгебраических уравнений методом симметричных последовательных верхних релаксаций (SSOR). 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 симметричных последовательных верхних релаксаций (SSOR). Вывод расширенной матрицы и вектора решений представлены на рисунке 4.1.

```
mts19@fedora Лабораторная работа 5  ✗ gcc -o program.out -fopenmp main.c array.c
mts19@fedora Лабораторная работа 5  ✗ export OMP_NUM_THREADS=1
mts19@fedora Лабораторная работа 5  ✗ ./program.out
-81.000000      42.000000       7.000000        -93.000000
42.000000       55.000000       13.000000       -47.000000
7.000000        13.000000       5.000000        72.000000
Time: 0.001721 seconds
-0.588183       -10.386850      42.229265       ⏎
```

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).

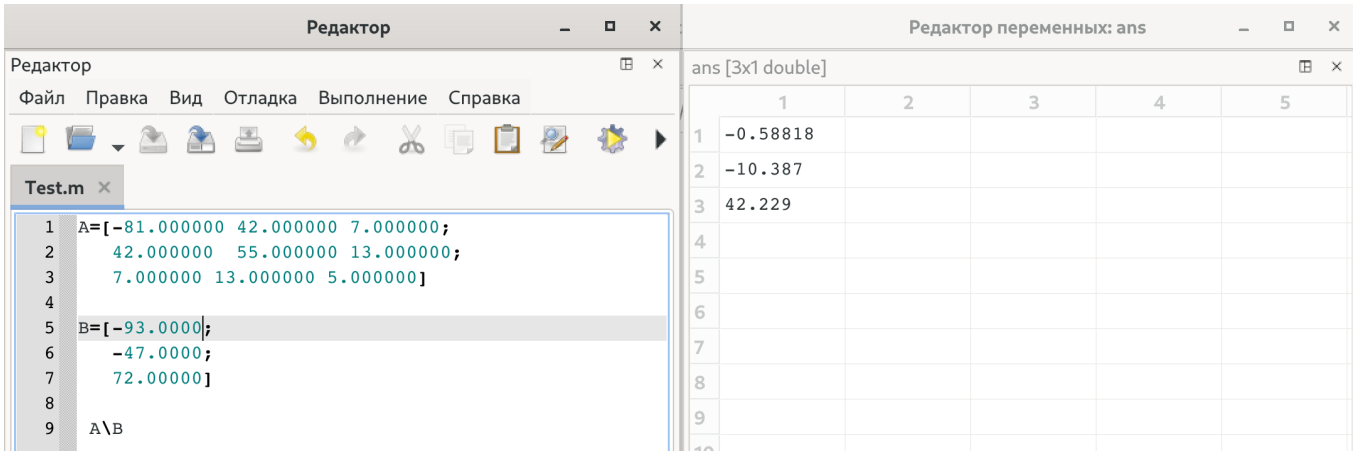

Рисунок 4.2. Результат решения СЛАУ в GNU Octave.

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 симметричных последовательных верхних релаксаций (SSOR) 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 array.c
mts19@fedora Лабораторная работа 5  ✗ export OMP_NUM_THREADS=8
mts19@fedora Лабораторная работа 5  ✗ ./program.out
-81.000000      42.000000       7.000000        -93.000000
42.000000       55.000000       13.000000       -47.000000
7.000000        13.000000       5.000000        72.000000
Time: 0.006394 seconds
-0.588183       -10.386850      42.229265       ⏎
```

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.

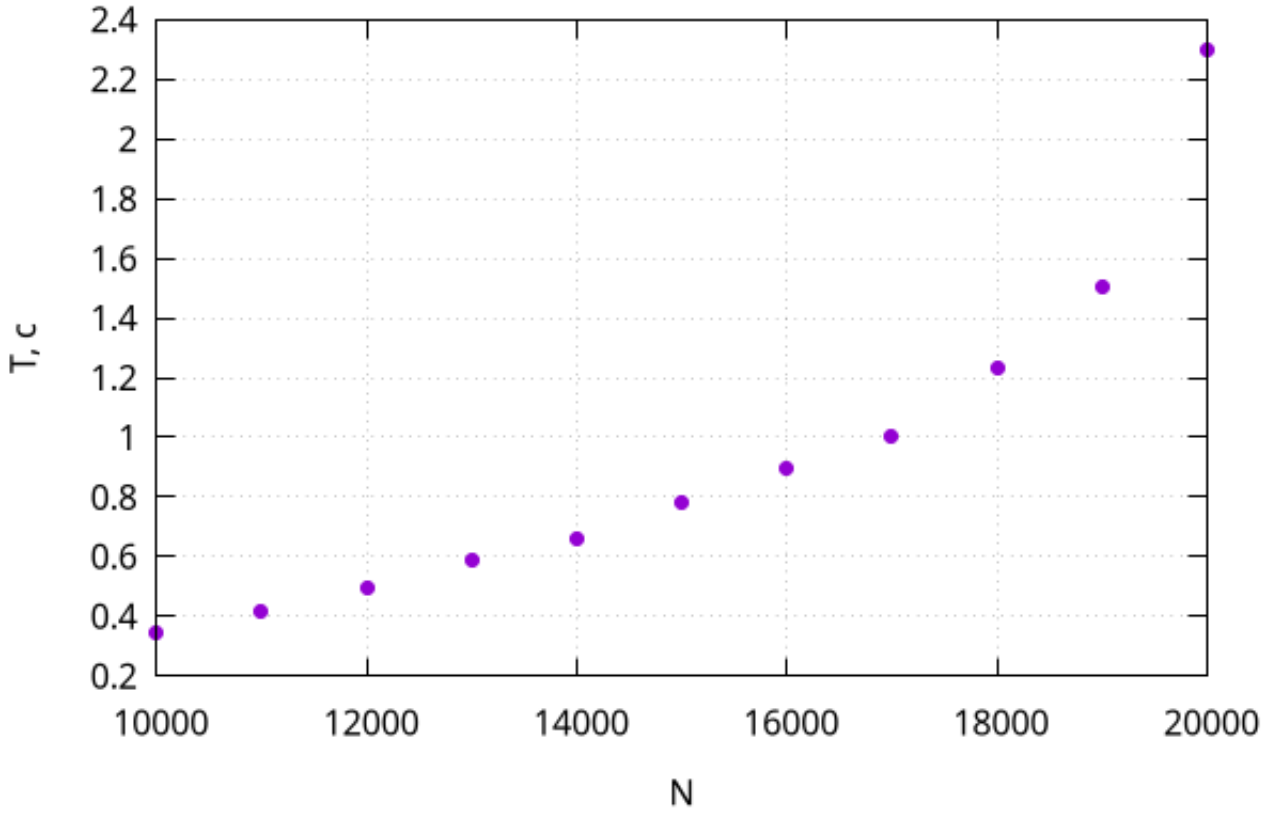

Рисунок 5.2. Зависимость времени выполнения программы от размерности матрицы для числа нитей $TN = 8$.

Теперь возьмём за основу размерность матрицы 10000x10000 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.

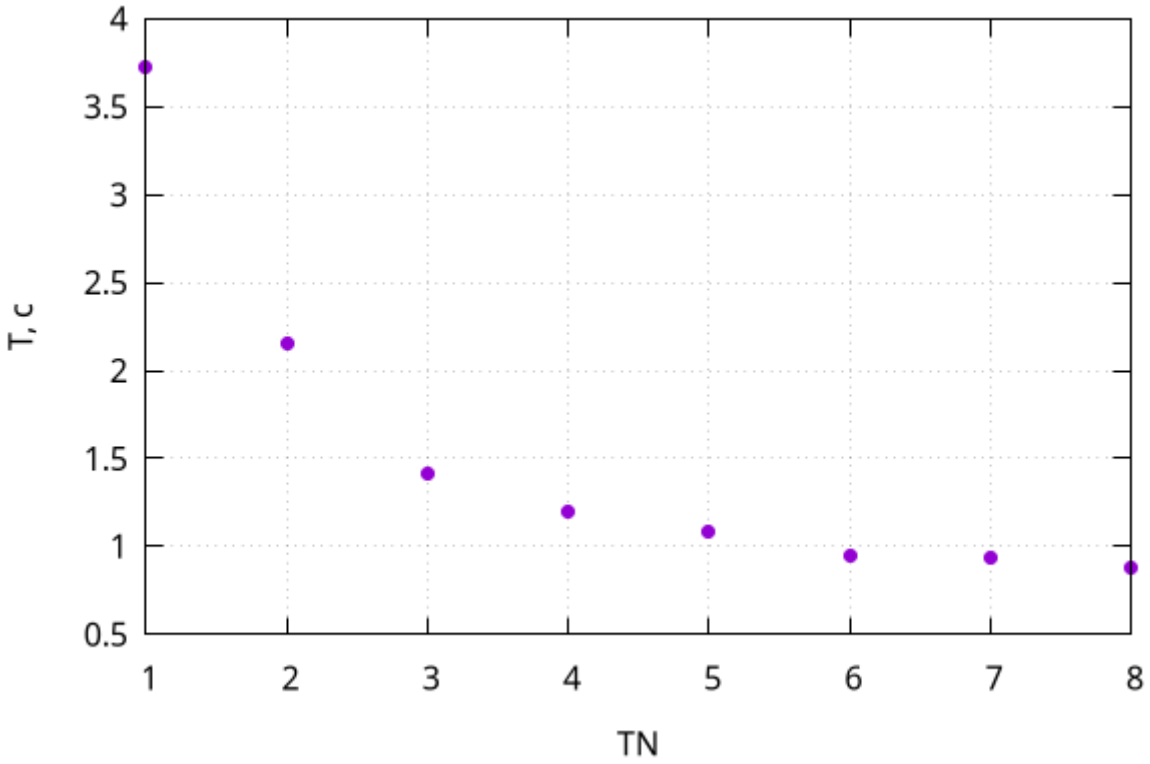

Рисунок 5.3. Зависимость времени выполнения программы от числа нитей для размерности матрицы 10000x10000.

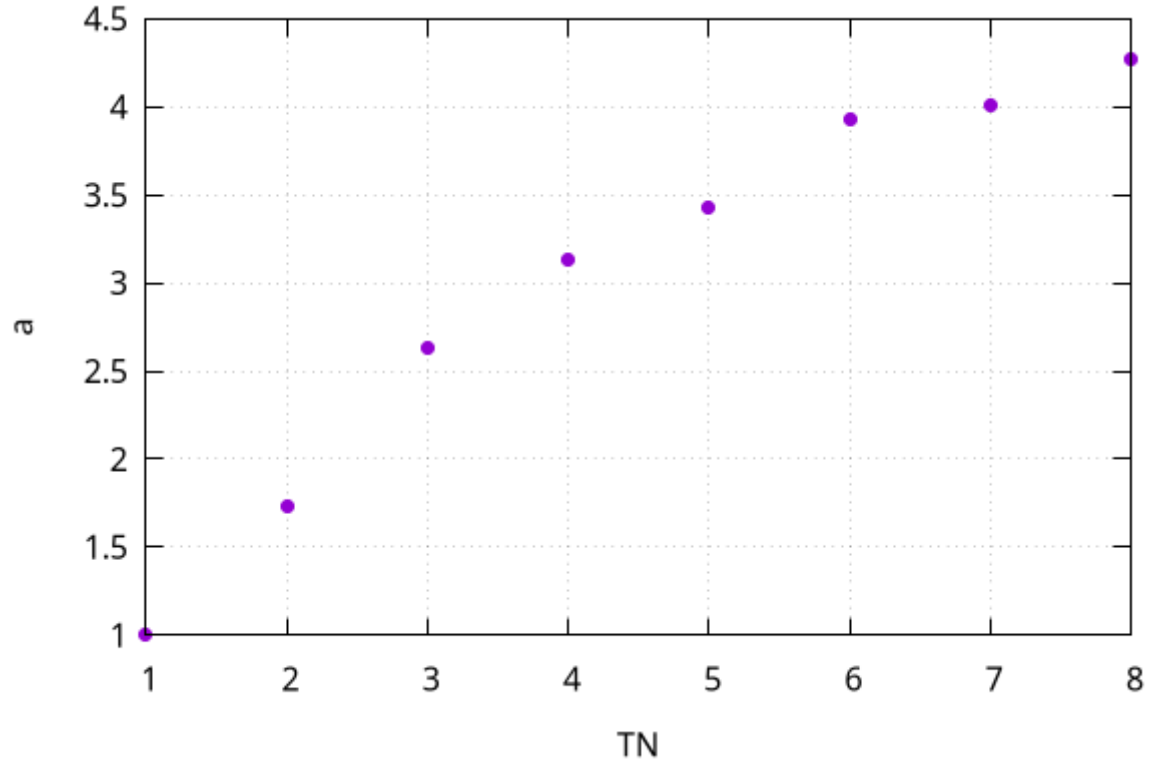

Рисунок 5.4. Зависимость ускорения ($a = \frac{T_1}{T_{tn}}$) выполнения программы от числа нитей для размерности матрицы 10000x10000.

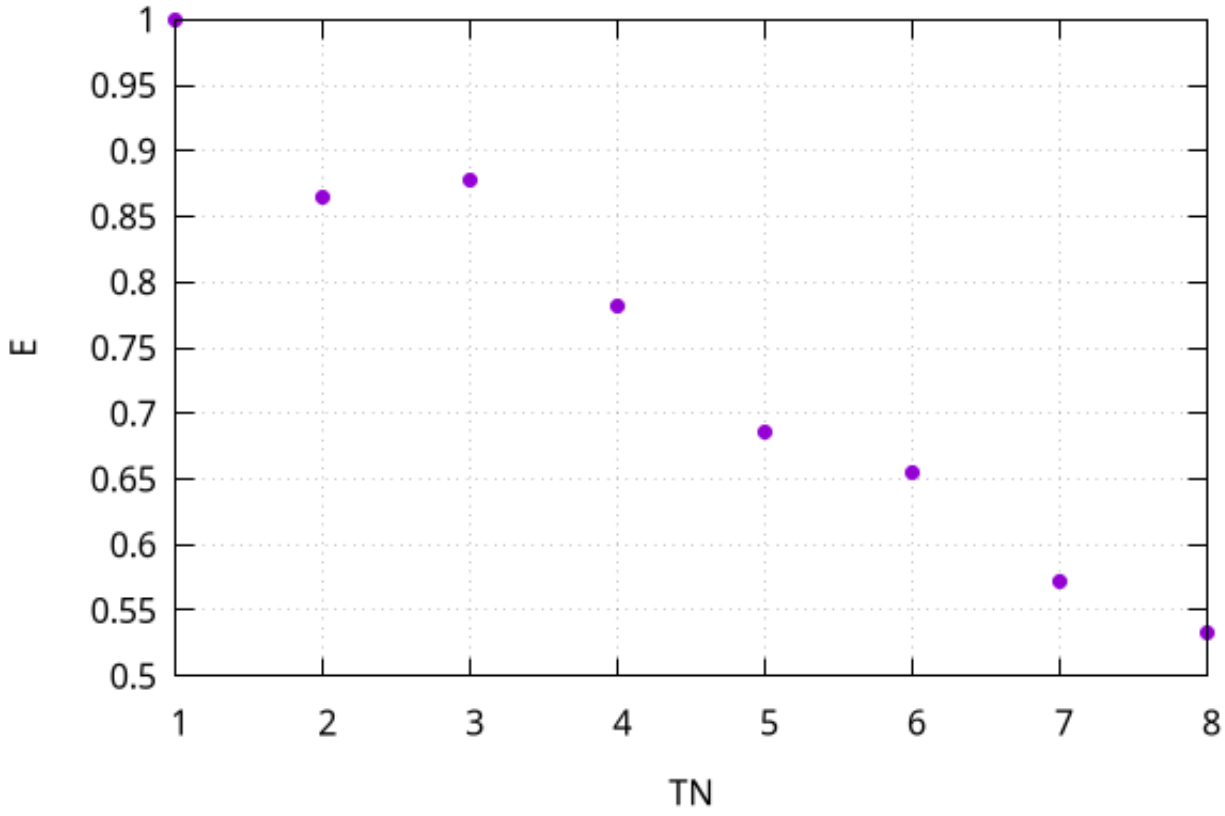

Рисунок 5.5. Зависимость эффективности ($e = \frac{a}{TN}$) выполнения программы от числа нитей для размерности матрицы 10000x10000.

Теперь, используя $TN = 8$ и размерность 10000x10000, 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	0.872842	0.774930	0.736636

Таблица 5.1

Зависимость времени выполнения от опций оптимизации компилятора для $TN = 8$ и размерности матрицы 10000x10000.

	STATIC	DYNAMIC, 500	GUIDED
t, сек	0.889253	0.901267	0.858979

Таблица 5.2

Зависимость времени выполнения от способа распределения итераций между нитями для $TN = 8$ и размерности матрицы 10000x10000.

6. Вывод

В данной работе было реализовано расспараллеливание программы для решения СЛАУ методом симметричных последовательных верхних релаксаций (SSOR) 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.
2. С увеличением количества нитей TN ускорение растет, а эффективность падает.
3. Способ распределения итераций между циклами не оказывает сильного влияния на время выполнения программы.
4. Опции оптимизатора ускоряют время выполнения программы примерно в 1,2 раза.

7. Коды программ

```
#include "array.h"
#include <math.h>
#include <omp.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#define N 15000
#define OMEGA 1.7
#define EPS 0.000001

void init_residual(double *R, double *C, double *X, size_t n);

int main() {
  omp_set_dynamic(0);
#pragma omp declare reduction(                                                  \
    maxfabs:double_t                                                            \
    : omp_out = fabs(omp_in) > omp_out ? fabs(omp_in) : omp_out)                \
    initializer(omp_priv = 0.0)

  double **A = random_symmetric_positive_definite_matrix_alloc(N);
  double *B = random_vector_alloc(N);
  // for (size_t i = 0; i < N; i++) {
  //   for (size_t j = 0; j < N; j++) {
  //     printf("%f\t", A[i][j]);
  //   }
  //   printf("%f\n", B[i]);
  // }

  double start_time = omp_get_wtime();
```

```
  double **P = matrix_alloc(N, N);
  double *C = vector_alloc(N);
#pragma omp parallel shared(A, B, P, C)
  {
#pragma omp for collapse(2) nowait
    for (size_t i = 0; i < N; i++) {
      for (size_t j = 0; j < N; j++) {
        if (i == j) {
          P[i][j] = 0.0;
        } else {
          P[i][j] = -A[i][j] / A[i][i];
        }
      }
    }
#pragma omp for
    for (size_t i = 0; i < N; i++) {
      C[i] = B[i] / A[i][i];
    }
  }
  free(A);
  free(B);

  double *R = vector_alloc(N);
  double *X = vector_alloc(N);
#pragma omp parallel for shared(X)
  for (size_t i = 0; i < N; i++) {
    X[i] = 0.0;
  }
  double abs_max_residual;
  do {
#pragma omp parallel shared(R, C, X)
    { init_residual(R, C, X, N); }
    for (size_t i = 0; i < N; i++) {
#pragma omp parallel for reduction(+ : R[i]) shared(i, P, X)
      for (size_t j = 0; j < N; j++) {
        R[i] += P[i][j] * X[j];
      }
      X[i] += OMEGA * R[i];
    }
#pragma omp parallel shared(R, C, X)
    { init_residual(R, C, X, N); }
    for (size_t i = N - 1; i < SIZE_MAX; i--) {
#pragma omp parallel for reduction(+ : R[i]) shared(i, P, X)
      for (size_t j = 0; j < N; j++) {
        R[i] += P[i][j] * X[j];
      }
      X[i] += OMEGA * R[i];
    }
    abs_max_residual = 0.0;
#pragma omp parallel for reduction(maxfabs : abs_max_residual) shared(R)
```

```
    for (size_t i = 0; i < N; i++) {
      if (fabs(R[i]) > abs_max_residual) {
        abs_max_residual = fabs(R[i]);
      }
    }
  } while (abs_max_residual ⩾ EPS);

  double end_time = omp_get_wtime();
  printf("Time: %f seconds\n", end_time - start_time);

  free(P);
  free(C);
  free(R);
  // for (size_t i = 0; i < N; i++) {
  //   printf("%f\t", X[i]);
  // }
  free(X);

  return 0;
}

void init_residual(double *R, double *C, double *X, size_t n) {
#pragma omp for
  for (size_t i = 0; i < n; i++) {
    R[i] = C[i] - X[i];
  }
}
```

Модуль:

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
#define MIN -100

double **matrix_alloc(size_t nrows, size_t ncolumns) {
  double **matrix =
      (double **)malloc(nrows * (sizeof(double *) + ncolumns * sizeof(double)));
  if (matrix == NULL) {
    perror("Matrix malloc failed");
    exit(EXIT_FAILURE);
  }
  double *start = (double *)((char *)matrix + nrows * sizeof(double *));
  for (size_t i = 0; i < nrows; i++) {
    matrix[i] = start + i * ncolumns;
  }
  return matrix;
}

double **random_symmetric_positive_definite_matrix_alloc(size_t n) {
```

```
  double **matrix = matrix_alloc(n, n);
  double **temp_matrix = matrix_alloc(n, n);
  for (size_t i = 0; i < n; i++) {
    for (size_t j = 0; j < n; j++) {
      temp_matrix[i][j] = (double)(MIN + rand() % (MAX - MIN + 1));
    }
  }
  for (size_t i = 0; i < n; i++) {
    for (size_t j = 0; j < n; j++) {
      matrix[i][j] = (temp_matrix[i][j] + temp_matrix[j][i]) / 2.0;
    }
  }
  free(temp_matrix);
  return matrix;
}

double *vector_alloc(size_t n) {
  double *vector = (double *)malloc(n * sizeof(double));
  if (vector == NULL) {
    perror("Vector malloc failed");
    exit(EXIT_FAILURE);
  }
  return vector;
}

double *random_vector_alloc(size_t n) {
  double *vector = vector_alloc(n);
  for (size_t i = 0; i < n; i++) {
    vector[i] = (double)(MIN + rand() % (MAX - MIN + 1));
  }
  return vector;
}
```