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 №4

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

Санкт-Петербург
2023

Содержание

1. Задание

Реализовать и распараллелить средствами OpenMP операцию вычисления степени разряженности матрицы большой размерности. Сравнить время работы программ, определить ускорение и эффективность распараллеливания.

2. Характеристики компьютера

Количество ядер: 4, количество потоков: 8, объём оперативной памяти: 8Гб.

3. Исследование последовательной версии программы

Разработана последовательная версия программы, для сравнения с тестовым решением степени разряженности единичной матрицы размера 16x16 (рисунок 3.1). Как видно, программа работает верно, поэтому можем приступить к исследованию параллельной версии программы.

```
mts19@fedora Лабораторная работа 4  ✗ ./program.out
Enter number of threads: 1
Time: 0.000017 seconds
Number of zeros: 240
Sparsity: 0.937500
```

Рисунок 3.1. Результат работы последовательной версии программы для матрицы 16x16.

4. Исследование параллельной версии программы

Также сравним с тестовым решением степени разряженности единичной матрицы размера 16x16 параллельный алгоритм при $TN = 8$ (рисунок 4.1). Как видно, программа работает верно.

```
mts19@fedora Лабораторная работа 4  ✗ ./program.out
Enter number of threads: 8
Time: 0.000805 seconds
Number of zeros: 240
Sparsity: 0.937500
```

Рисунок 4.1. Результат работы параллельной версии программы для матрицы 16x16 и $TN = 8$.

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 4.2). Как видно, при увеличении размерности матрицы увеличивается и время работы.

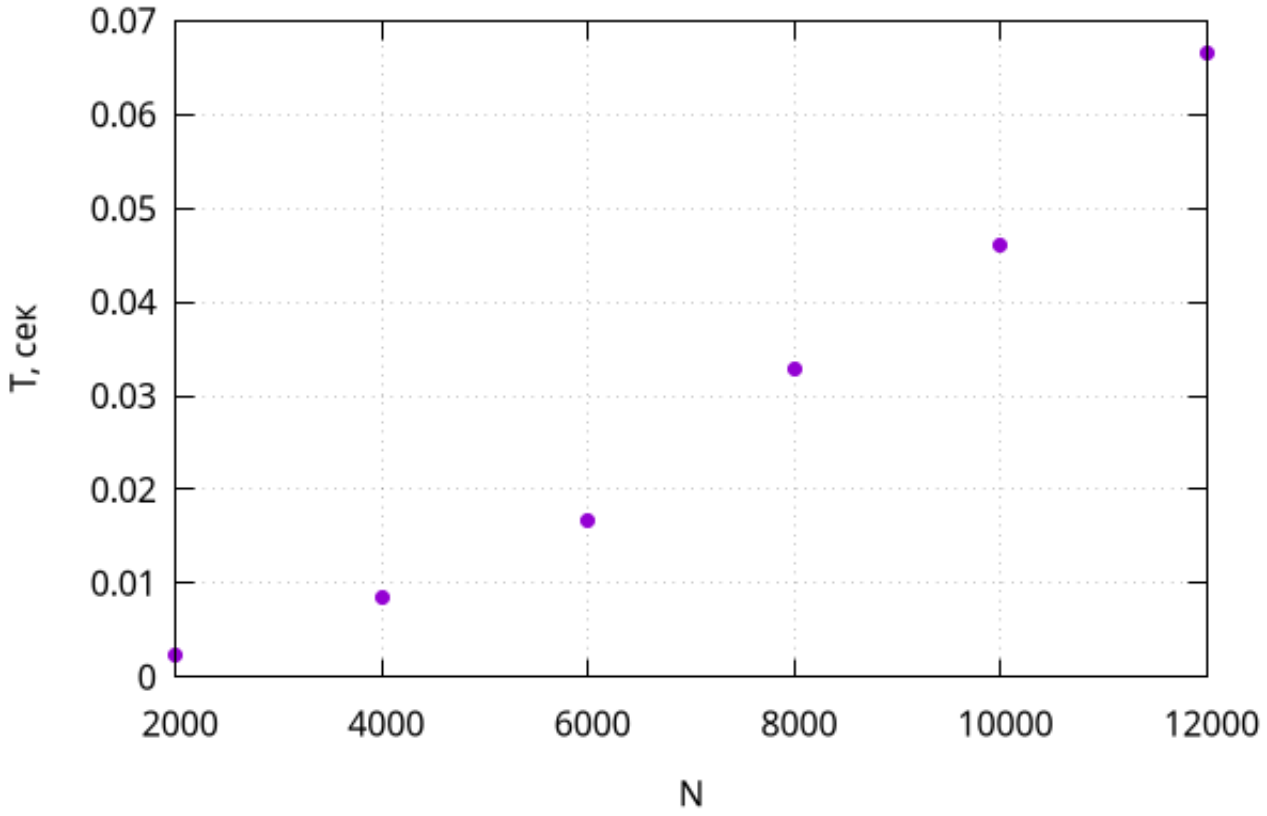

Рисунок 4.2. Зависимость времени выполнения программы от размерности матрицы для числа нитей $TN = 8$.

Теперь возьмём за основу размерность матрицы 20000x20000 и для неё исследуем зависимость времени выполнения, ускорения и ефективности от количества нитей (рисунок 4.3, рисунок 4.4, рисунок 4.5). Исследуя зависимости видно, что сначала увеличение количества нитей существенно влияет на скорость выполнения программы, но потом наступает некоторое насыщение и большого выигрыша не происходит.

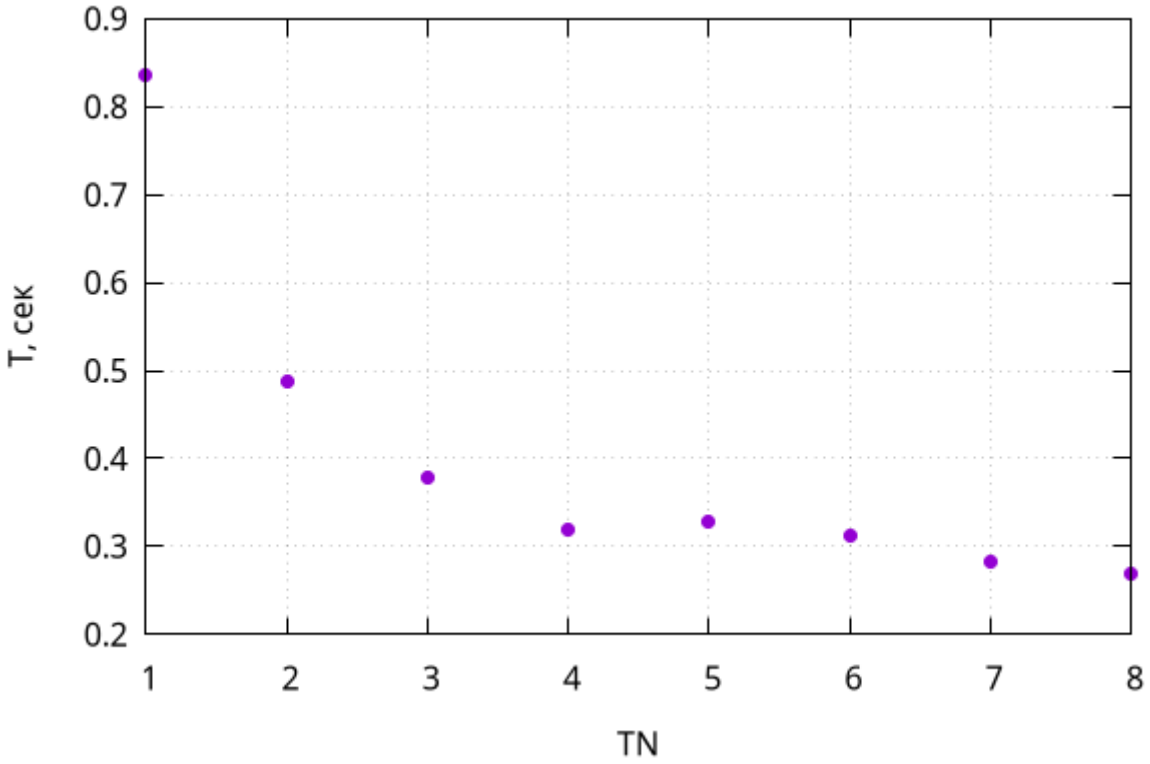

Рисунок 4.3. Зависимость времени выполнения программы от числа нитей для размерности матрицы 20000x20000.

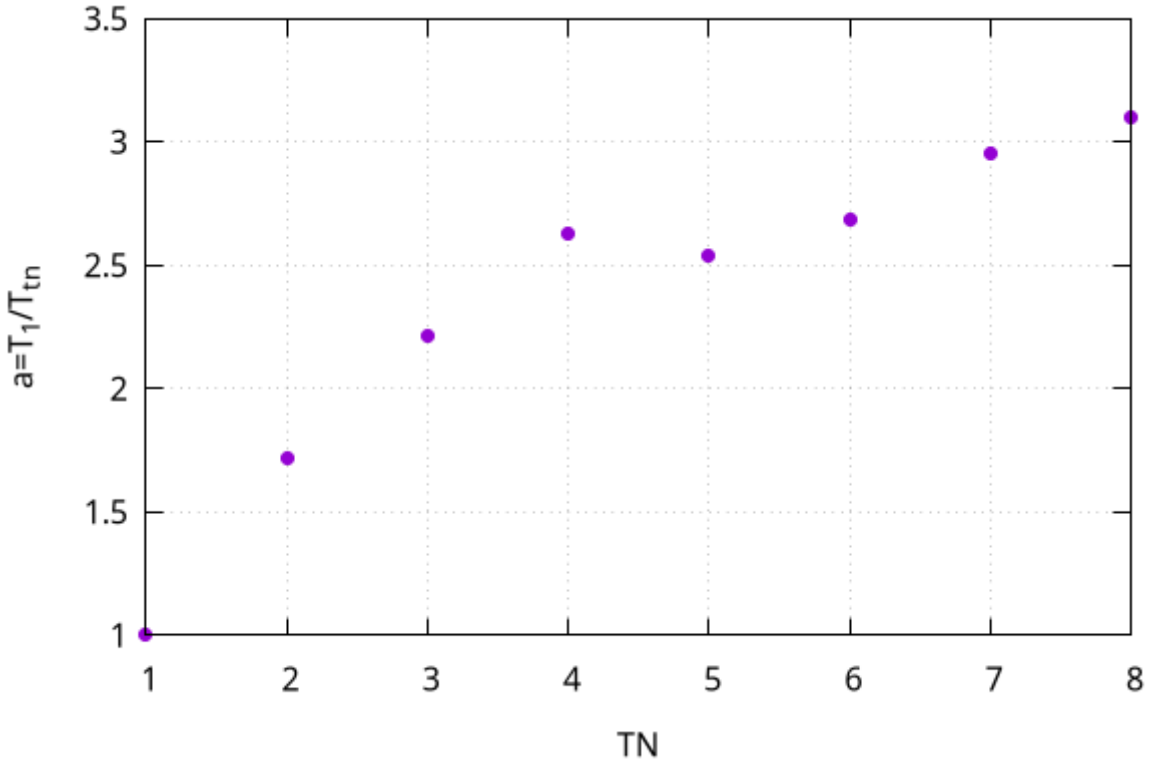

Рисунок 4.4. Зависимость ускорения ($a = \frac{T_1}{T_{tn}}$) выполнения программы от числа нитей для размерности матрицы 20000x20000.

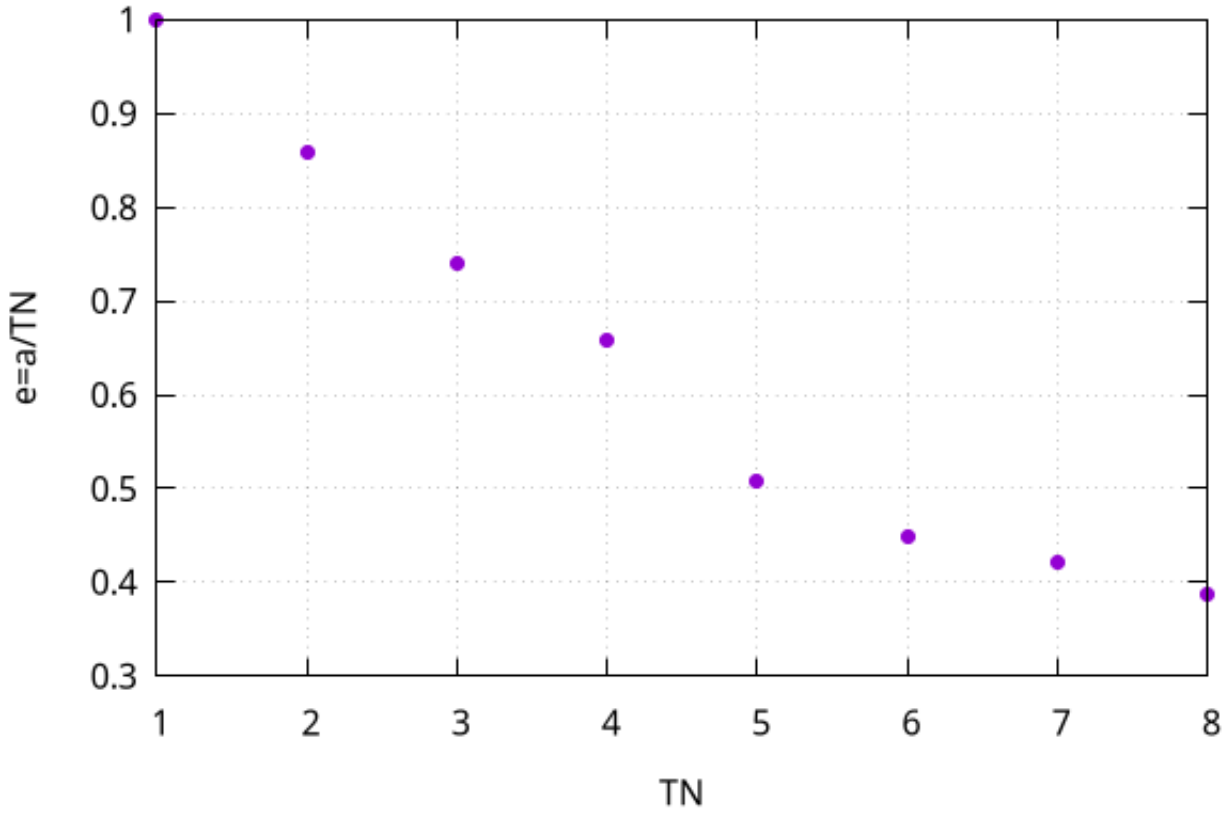

Рисунок 4.5. Зависимость еффективности ($e = \frac{a}{TN}$) выполнения программы от числа нитей для размерности матрицы 20000x20000.

Теперь, используя $TN = 8$ и размерность 20000x20000, исследуем влияние оптимизатора и способа распределения итераций между ниятми (таблица 4.1, таблица 4.2). 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	0.184550	0.122800	0.122768

Таблица 4.1

Зависимость времени выполнения от опций оптимизации компилятора для $TN = 8$ и размерности матрицы 20000x20000.

	STATIC	DYNAMIC, 500	GUIDED
t, сек	0.184299	0.189336	0.182553

Таблица 4.2

Зависимость времени выполнения от способа распределения итераций между нитями для $TN = 8$ и размерности матрицы 20000x20000.

5. Вывод

В данной работе было реализовано расспаралеливание программы для вычисления степени разряженности матрицы 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 увеличивается время выполнения программы.

2. С увеличением количества нитей возрастает ускорение, но падает эффективность параллелизации.

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.

6. Коды программ

```c
#include "arrays.h"
#include <math.h>
#include <omp.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define N 16
#define EPS 0.000001

int main() {
  size_t num_threads;
  printf("Enter number of threads: ");
  scanf("%zd", &num_threads);
  omp_set_num_threads(num_threads);

  double **A = two_dimensional_array_alloc(N, N);
  two_dimensional_array_random(A, N, N);
  for (size_t i = 0; i < N; i++) {
    for (size_t j = 0; j < N; j++) {
      if (i == j) {
        A[i][j] = 1.0;
      } else {
        A[i][j] = 0.0;
      }
    }
  }

  size_t num_zeros = 0;

  double start_time = omp_get_wtime();
#pragma omp parallel for collapse(2) shared(A) reduction(+ : num_zeros)
  for (size_t i = 0; i < N; i++) {
    for (size_t j = 0; j < N; j++) {
      if (fabs(A[i][j]) <= EPS) {
        num_zeros++;
      }
    }
  }
  double end_time = omp_get_wtime();

  free(A);

  double sparsity = num_zeros / pow(N, 2.0);

  printf("Time: %f seconds\n", end_time - start_time);
  printf("Number of zeros: %zd\n", num_zeros);
  printf("Sparsity: %f\n", sparsity);

  return 0;
}
```

Рисунок 6.1. Тестовая программа.

```c
#include "arrays.h"
#include <math.h>
#include <omp.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define N 20000
#define EPS 0.000001

int main() {
  size_t num_threads;
  printf("Enter number of threads: ");
  scanf("%zd", &num_threads);
  omp_set_num_threads(num_threads);

  double **A = two_dimensional_array_alloc(N, N);
  two_dimensional_array_random(A, N, N);

  size_t num_zeros = 0;

  double start_time = omp_get_wtime();
#pragma omp parallel for collapse(2) shared(A) reduction(+ : num_zeros)
  for (size_t i = 0; i < N; i++) {
    for (size_t j = 0; j < N; j++) {
      if (fabs(A[i][j]) <= EPS) {
        num_zeros++;
      }
    }
  }
  double end_time = omp_get_wtime();

  free(A);

  double sparsity = num_zeros / pow(N, 2.0);

  printf("Time: %f seconds\n", end_time - start_time);
  printf("Number of zeros: %zd\n", num_zeros);
  printf("Sparsity: %f\n", sparsity);

  return 0;
}
```

Рисунок 6.2. Параллельный алгоритм.

```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define frand() ((double)rand() / (RAND_MAX + 1.0))

double **two_dimensional_array_alloc(size_t rows, size_t columns) {
  double **arr = (double **)malloc(rows * sizeof(double *) +
                                   rows * columns * sizeof(double));
  if (arr == NULL) {
    perror("malloc failed");
    exit(EXIT_FAILURE);
  }
  double *start = (double *)((char *)arr + rows * sizeof(double *));
  for (size_t i = 0; i < rows; i++) {
    arr[i] = start + i * columns;
  }
  return arr;
}

void two_dimensional_array_random(double **arr, size_t rows, size_t columns) {
  for (size_t i = 0; i < columns; i++) {
    for (size_t j = 0; j < rows; j++) {
      arr[i][j] = frand();
    }
  }
}
```

Рисунок 6.3. Модуль.