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

ОТЧЕТ

О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью OpenMP»

студентки 2 курса, группы 21207

Черновской Яны Тихоновны

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Власенко

Новосибирск 2023

СОДЕРЖАНИЕ

ЦЕЛЬ

Изучить стандарт для распараллеливания программ на языке Си openMP

ЗАДАНИЕ

1. Последовательную программу из предыдущей практической работы, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью OpenMP.

   ОБЯЗАТЕЛЬНОЕ УСЛОВИЕ: создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.

2. Замерить время работы программы на кластере НГУ на 1, 2, 4, 8, 12, 16 потоках. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные и параметры задачи подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.

3. Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков.

Вариант задания:

Метод простой итерации

В методе простой итерации преобразование решения на каждом шаге задается формулой:

$$x^{n+1} = x^n - \tau(Ax^n - b).$$

Здесь τ – константа, параметр метода. В зависимости от значения параметра τ последовательность $\{x^n\}$ может сходиться к решению быстрее или медленнее, или вообще расходиться. В качестве подходящего значения τ

[1]Общая формула для итерационных методов выглядит следующим образом: $x^{n+1} = f(x^{n+1}, x^n, x^{n-1}, \ldots, x^0)$, но для целей лабораторных работ достаточно будет формулы, представленной в тексте.

можно взять 0.01 или -0.01. Знак параметра τ зависит от задачи. Если с некоторым знаком решение начинает расходиться, то следует сменить его на противоположный. Критерий завершения счета:

$$\frac{\|Ax^n - b\|_2}{\|b\|_2} < \varepsilon,$$

где $\|u\|_2 = \sqrt{\sum_{i=0}^{N-1} u_i^2}$. Для тестирования метода значение ε можно взять равным 10^{-5}.

ОПИСАНИЕ РАБОТЫ

1. Была написана параллельная реализация решения системы линейных алгебраических уравнений с помощью openMP

a. На 1 потоке

```
hpcuser221@clu:~/lab2> cat omp_slae.sh.o5399683
OMP_NUM_THREADS = 1


1
Total time is 57.553917 seconds
```

b. На 2 потоках

```
hpcuser221@clu:~/lab2> cat omp_slae.sh.o5399685
OMP_NUM_THREADS = 2


2
Total time is 31.065260 seconds
```

c. На 4 потоках

```
OMP_NUM_THREADS = 4


4
Total time is 14.692519 seconds
```

d. На 8 потоках

```
OMP_NUM_THREADS = 8


8
Total time is 7.302411 seconds
```

e. На 16 потоках

```
OMP_NUM_THREADS = 16

16
Total time is 8.060798 seconds
```

f. На 24 потоках

```
OMP_NUM_THREADS = 24

24
Total time is 6.924275 seconds
```

3. Были построены графики времени, ускорения и эффективности, где

а) Ускорение: Sp = T1 / Tp, где T1 - время работы на 1 потоке, Tp - время работы параллельной программы на p процессах/потоках

б) Эффективность Ep = Sp / p * 100%.

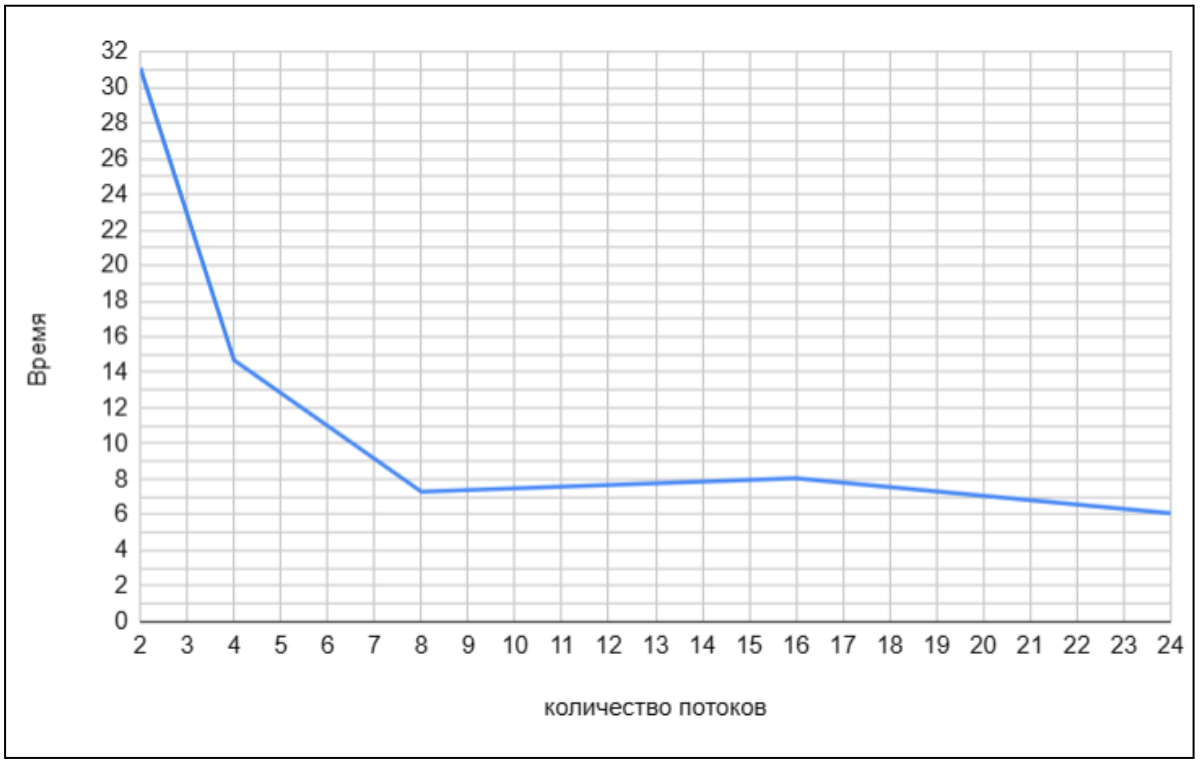

Рис 1. время работы программы

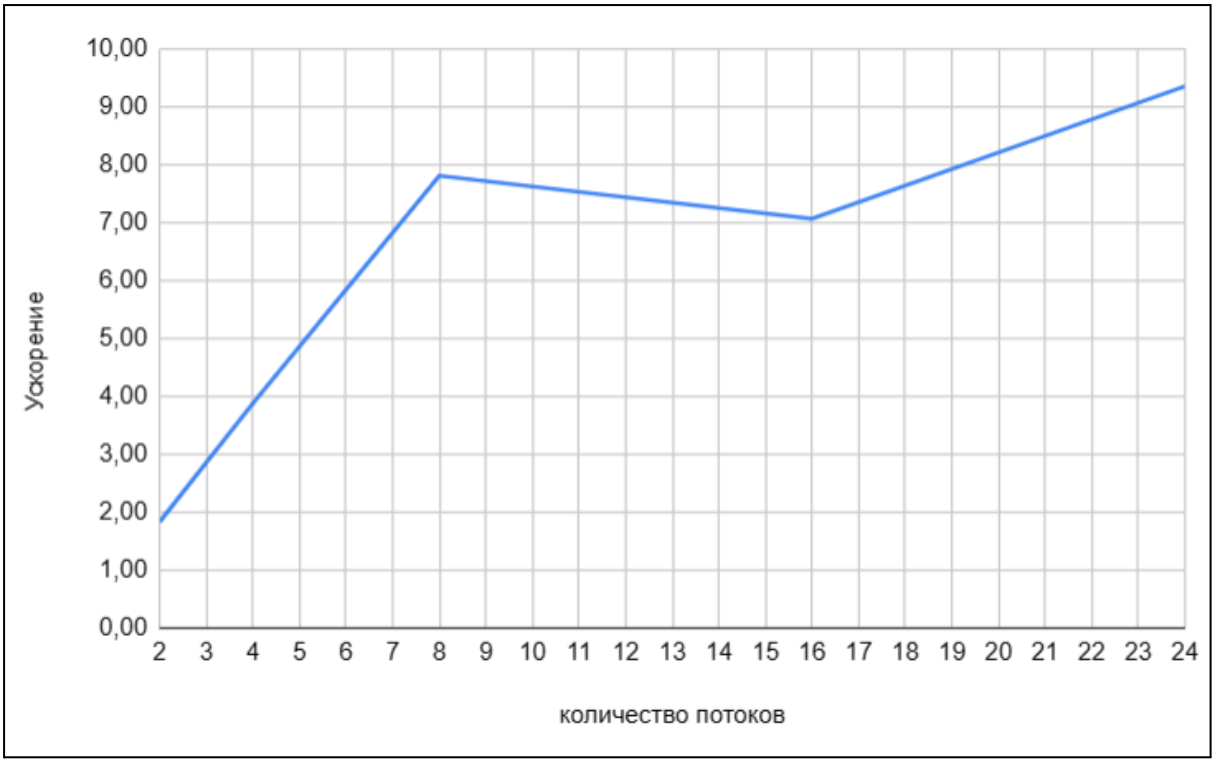

Рис2. ускорение

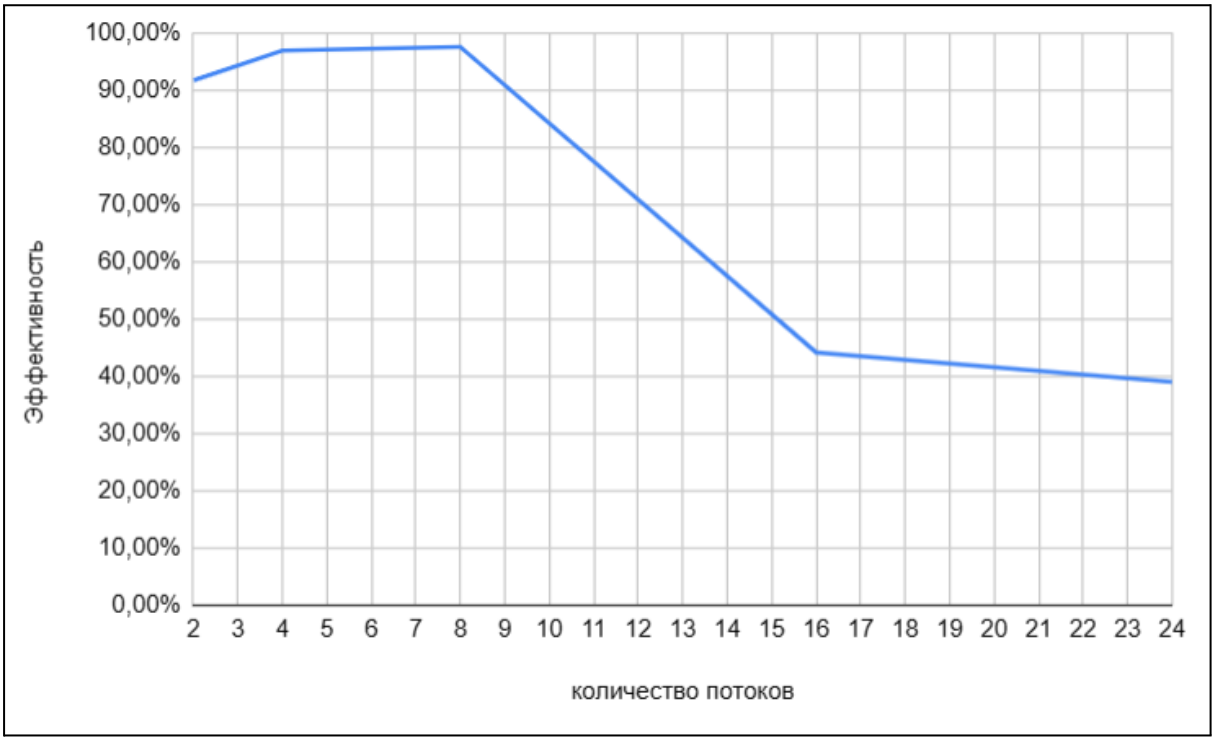

Рис3. эффективность

4. Было проведено исследование на определение оптимальных параметров #pragma omp for schedule(...) при фиксированном размере задачи (N = 5000) и количестве потоков(n = 4)

Chunk	Static	Dynamic	Guided
1	6,78	9,41	4,65
2	6,73	9,03	4,76
3	6,83	8,69	4,64
4	5,79	8,57	4,57
5	5,53	7,61	4,51
6	5,45	6,81	4,59
7	6,2	6,37	4,61
8	4,86	5,96	4,58
9	5	5,21	4,68
10	5,04	4,82	4,5
50	5,09	5,32	4,79
100	5,02	5,29	5,05
500	7,57	7,87	7,86
1000	14,58	15,72	15,71
1500	17,91	20,13	19,75

ЗАКЛЮЧЕНИЕ

По полученным данным в таблице можно сделать вывод, что static и quided работает примерно одинаково на большинстве размеров chunk, dynamic требует же ручного подбора размера. Наиболее быстро сработала программа с параметром quided с размером chunk = 10.

ПРИЛОЖЕНИЕ

Приложение 1. Листинг файла makefile

```
1.  omp_slae.out: omp_slae.c
2.          gcc -fopenmp -o $@ omp_slae.c -lm -std=c99hpcuser221@clu:~/lab2>
```

Приложение 2. Скрипт для запуска параллельной программы

```
1. #!/bin/sh
2. #PBS -l walltime=00:00:50
3. #PBS -l select=1:ncpus=12:ompthreads=24
4.
5. cd $PBS_O_WORKDIR
6. echo "OMP_NUM_THREADS = $OMP_NUM_THREADS"
7. echo
8. ./omp_slae.out
9.
```

Приложение 3. Полный листинг параллельной программы на C

```
1.  #include <math.h>
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.  #include <omp.h>
5.
6.  #define N 5000
7.  #define EPSILON 1e-6
8.  #define MAX_ITERATION_COUNT 50000
9.  #define TAU  1e-5
10.
11. void generate_matrix(double* matrix);
12. void generate_vector(double* vector);
13.
14. void print_matrix(const double* matrix);
15. void print_vector(const double* vector);
16.
17. double count_square_norm(const double *vector, int size);
18. void set_matrix_part(int* line_counts, int* offsets, int size, int thread_num);
19.
20. void mul(const double* matrix, const double* vector, double* result, int lines);
21. void sub_vectors(const double* vector1, const double* vector2, double* result, int
    size);
22. void count_new_x(double* x, const double* vector, int size);
23. void check(double* A, double*x , double* b);
24.
25. int main(int argc, char **argv)
26. {
27.     double* A = malloc(sizeof(double) * N * N);
28.     double* x = malloc(sizeof(double) * N);
29.     double* b = malloc(sizeof(double) * N);
30.
31.     int  num_threads = omp_get_max_threads();
32.     printf("%d\n", num_threads);
33.     int* line_counts = malloc(sizeof(int) * num_threads);
34.     int* offsets = malloc(sizeof(int) * num_threads);
35.     double *buffer = malloc(sizeof(double) * N);
36.
37.     set_matrix_part(line_counts, offsets, N, num_threads);
38.
39.     generate_matrix(A);
40.     generate_vector(x);
41.     generate_vector(b);
42.
43.     double b_norm = sqrt(count_square_norm(b, N));
```

```
44.     int count_iterations = 0;
45.     double res = 1;
46.     double sum_norm = 0;
47.
48.     double begin = omp_get_wtime();
49.     //print_matrix(A);
50.     //print_vector(b);
51.
52.     int iter_count = 0;
53.     #pragma omp parallel
54.     {
55.         int thread_id = omp_get_thread_num();
56.         for (iter_count = 0; res > EPSILON && iter_count < MAX_ITERATION_COUNT;
    ++iter_count)
57.         {
58.             mul(A + offsets[thread_id] * N, x, buffer + offsets[thread_id],
    line_counts[thread_id]);
59.             sub_vectors(buffer + offsets[thread_id], b + offsets[thread_id], buffer +
    offsets[thread_id], line_counts[thread_id]);
60.
61.     #pragma omp barrier
62.             count_new_x(x + offsets[thread_id], buffer + offsets[thread_id],
    line_counts[thread_id]);
63.
64.     #pragma omp single
65.             sum_norm = 0;
66.
67.     #pragma omp atomic
68.             sum_norm += count_square_norm(buffer + offsets[thread_id],
    line_counts[thread_id]);
69.
70.     #pragma omp barrier
71.     #pragma omp single
72.             res = sqrt(sum_norm) / b_norm;
73.         }
74.     }
75.
76.     double end = omp_get_wtime();
77.
78.     if (count_iterations == MAX_ITERATION_COUNT){
79.         printf("Wrong tau\n");
80.     }
81.
82.     else{
83.         printf("Total time is %f seconds\n", (end - begin));
84.         //check(A, x, b);
85.         //print_vector(x);
86.     }
```

```
87.
88.     free(A);
89.     free(x);
90.     free(b);
91.     free(buffer);
92.
93.     return EXIT_SUCCESS;
94. }
95.
96. void check(double* A, double*x , double* b){
97.     double * result = malloc(sizeof(double) * N);
98.     mul(A,x, result, N);
99.     for (int i = 0; i < N; i++){
100.            if (result[i] - b[i] < EPSILON || b[i] - result[i] < EPSILON){}
101.            else{
102.                printf("not okay\n");
103.                return;
104.            }
105.        }
106.        printf("correct\n");
107.    }
108.
109.    void generate_vector(double* vector)
110.    {
111.        for (int i = 0; i < N; i++)
112.        {
113.            vector[i] = (double)rand() / RAND_MAX * 10.0 - 5.0;
114.        }
115.    }
116.
117.    void generate_matrix(double* matrix)
118.    {
119.        for(int i = 0; i < N; i++)
120.        {
121.            for(int j = 0; j < i; j++)
122.            {
123.                matrix[i * N + j] = matrix[j * N + i];
124.            }
125.
126.            for(int j = i; j < N; j++)
127.            {
128.                matrix[i * N + j] = (double)rand() / RAND_MAX * 2.0 - 1.0; //float in range
    -1 to 1
129.                if(i == j) matrix[i * N + j] = matrix[i * N + j] + N;
130.
131.            }
132.        }
133.    }
```

```
134.
135.
136.  void print_matrix(const double* matrix)
137.  {
138.      for (int i = 0; i < N; i++)
139.      {
140.          for (int j = 0; j < N; j++)
141.          {
142.              printf("%f ", matrix[i * N + j]);
143.          }
144.
145.          printf("\n");
146.      }
147.
148.      printf("\n");
149.  }
150.
151.  void print_vector(const double* vector)
152.  {
153.      for(int i = 0; i < N; i++)
154.      {
155.          printf("%f ", vector[i]);
156.      }
157.
158.      printf("\n");
159.  }
160.
161.
162.  double count_square_norm(const double* vector, int size)
163.  {
164.      double norm_value = 0;
165.      for (int i = 0; i < size; i++)
166.      {
167.          norm_value += vector[i] * vector[i];
168.      }
169.
170.      return norm_value;
171.  }
172.
173.  void set_matrix_part(int* line_counts, int* offsets, int size, int thread_num)
174.  {
175.      int offset = 0;
176.      for (int i = 0; i < thread_num; ++i)
177.      {
178.          line_counts[i] = size / thread_num;
179.
180.          if (i < size % thread_num)
181.          {
```

```
182.                 ++line_counts[i];
183.             }
184.
185.             offsets[i] = offset;
186.             offset += line_counts[i];
187.
188.         }
189.     }
190.
191.     void mul(const double* matrix, const double* vector, double* result, int lines)
192.     {
193.         for (int i = 0; i < lines; i++)
194.         {
195.             result[i] = 0;
196.
197.             for (int j = 0; j < N; j++)
198.             {
199.                 result[i] += matrix[i * N + j] * vector[j];
200.             }
201.         }
202.
203.     }
204.
205.     void sub_vectors(const double* vector1, const double* vector2, double*
    result, int size)
206.     {
207.         for (int i = 0; i < size; i++)
208.         {
209.             result[i] = vector1[i] - vector2[i];
210.         }
211.     }
212.
213.     void count_new_x(double* x, const double* vector, int size)
214.     {
215.         for (int i = 0; i < size; i++)
216.         {
217.             x[i] = x[i] - TAU * vector[i];
218.         }
219.     }
```

Приложение 4. Полный листинг параллельной программы на C для исследования оптимальных параметров #pragma omp for schedule(...)

```
1.  #include <math.h>
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.  #include <omp.h>
5.
6.  #define N 5000
7.  #define EPSILON 1e-6
8.  #define MAX_ITERATION_COUNT 50000
9.  #define TAU  1e-5
10.
11. void generate_matrix(double* matrix);
12. void generate_vector(double* vector);
13.
14. void print_matrix(const double* matrix);
15. void print_vector(const double* vector);
16.
17. double count_square_norm(const double *vector, int size);
18. void set_matrix_part(int* line_counts, int* offsets, int size, int thread_num);
19.
20. void mul(const double* matrix, const double* vector, double* result, int lines);
21. void  sub_vectors(const  double*  vector1,  const  double*  vector2,  double*  result,
    int size);
22. void count_new_x(double* x, const double* vector, int size);
23. void check(double* A, double*x , double* b);
24.
25. int main(int argc, char **argv)
26. {
27.     double* A = malloc(sizeof(double) * N * N);
28.     double* x = malloc(sizeof(double) * N);
29.     double* b = malloc(sizeof(double) * N);
30.
31.     int  num_threads = omp_get_max_threads();
32.     int* line_counts = malloc(sizeof(int) * num_threads);
33.     int* offsets = malloc(sizeof(int) * num_threads);
34.     double *buffer = malloc(sizeof(double) * N);
35.
36.     set_matrix_part(line_counts, offsets, N, num_threads);
37.
38.     generate_matrix(A);
39.     generate_vector(x);
40.     generate_vector(b);
41.
42.     double b_norm = sqrt(count_square_norm(b, N));
43.     int count_iterations = 0;
44.     double res = 1;
45.     double sum_norm = 0;
46.
47.     double begin = omp_get_wtime();
48.     //print_matrix(A);
49.     //print_vector(b);
50.
51.     int iter_count = 0;
```

```
52.
53.     int thread_id = omp_get_thread_num();
54.                 for (iter_count = 0; res > EPSILON && iter_count <
    MAX_ITERATION_COUNT; ++iter_count)
55.     {
56.                     mul(A + offsets[thread_id] * N, x, buffer + offsets[thread_id],
    line_counts[thread_id]);
57.             sub_vectors(buffer + offsets[thread_id], b + offsets[thread_id], buffer +
    offsets[thread_id], line_counts[thread_id]);
58.
59.                     count_new_x(x + offsets[thread_id], buffer + offsets[thread_id],
    line_counts[thread_id]);
60.
61.
62.         sum_norm = 0;
63.
64.                     sum_norm += count_square_norm(buffer + offsets[thread_id],
    line_counts[thread_id]);
65.
66.         res = sqrt(sum_norm) / b_norm;
67.     }
68.
69.     double end = omp_get_wtime();
70.
71.     if (count_iterations == MAX_ITERATION_COUNT){
72.         printf("Wrong tau\n");
73.     }
74.
75.     else{
76.         printf("%f \n", (end - begin));
77.         //check(A, x, b);
78.         //print_vector(x);
79.     }
80.
81.     free(A);
82.     free(x);
83.     free(b);
84.     free(buffer);
85.
86.     return EXIT_SUCCESS;
87. }
88.
89. void check(double* A, double*x , double* b){
90.     double * result = malloc(sizeof(double) * N);
91.     mul(A,x, result, N);
92.     for (int i = 0; i < N; i++){
93.         if (result[i] - b[i] < EPSILON || b[i] - result[i] < EPSILON){}
94.         else{
95.             printf("not okay\n");
96.             return;
97.         }
98.     }
99.     printf("correct\n");
100.    }
101.
```

```
102.  void generate_vector(double* vector)
103.  {
104.      for (int i = 0; i < N; i++)
105.      {
106.          vector[i] = (double)rand() / RAND_MAX * 10.0 - 5.0;
107.      }
108.  }
109.
110.  void generate_matrix(double* matrix)
111.  {
112.      for(int i = 0; i < N; i++)
113.      {
114.          for(int j = 0; j < i; j++)
115.          {
116.              matrix[i * N + j] = matrix[j * N + i];
117.          }
118.
119.          for(int j = i; j < N; j++)
120.          {
121.                matrix[i * N + j] = (double)rand() / RAND_MAX * 2.0 - 1.0; //float in
    range -1 to 1
122.              if(i == j) matrix[i * N + j] = matrix[i * N + j] + N;
123.
124.          }
125.      }
126.  }
127.
128.
129.  void print_matrix(const double* matrix)
130.  {
131.      for (int i = 0; i < N; i++)
132.      {
133.          for (int j = 0; j < N; j++)
134.          {
135.              printf("%f ", matrix[i * N + j]);
136.          }
137.
138.          printf("\n");
139.      }
140.
141.      printf("\n");
142.  }
143.
144.  void print_vector(const double* vector)
145.  {
146.      for(int i = 0; i < N; i++)
147.      {
148.          printf("%f ", vector[i]);
149.      }
150.
151.      printf("\n");
152.  }
153.
154.
155.  double count_square_norm(const double* vector, int size)
```

```
156.    {
157.        double norm_value = 0;
158.
159.        #pragma omp parallel for schedule(runtime) reduction(+: norm_value)
160.        for (int i = 0; i < size; i++)
161.        {
162.            norm_value += vector[i] * vector[i];
163.        }
164.
165.        return norm_value;
166.    }
167.
168.    void set_matrix_part(int* line_counts, int* offsets, int size, int thread_num)
169.    {
170.        int offset = 0;
171.        for (int i = 0; i < thread_num; ++i)
172.        {
173.            line_counts[i] = size / thread_num;
174.
175.            if (i < size % thread_num)
176.            {
177.                ++line_counts[i];
178.            }
179.
180.            offsets[i] = offset;
181.            offset += line_counts[i];
182.
183.        }
184.    }
185.
186.    void mul(const double* matrix, const double* vector, double* result, int lines)
187.    {
188.        #pragma omp parallel for schedule(runtime)
189.        for (int i = 0; i < lines; i++)
190.        {
191.            result[i] = 0;
192.
193.            for (int j = 0; j < N; j++)
194.            {
195.                result[i] += matrix[i * N + j] * vector[j];
196.            }
197.        }
198.
199.    }
200.
201.    void sub_vectors(const double* vector1, const double* vector2, double*
        result, int size)
202.    {
203.        #pragma omp parallel for schedule(runtime)
204.        for (int i = 0; i < size; i++)
205.        {
206.            result[i] = vector1[i] - vector2[i];
207.        }
208.    }
209.
```

```
210.    void count_new_x(double* x, const double* vector, int size)
211.    {
212.        #pragma omp parallel for schedule(runtime)
213.        for (int i = 0; i < size; i++)
214.        {
215.            x[i] = x[i] - TAU * vector[i];
216.        }
217.    }
218.
```

Приложение 5. Скрипт для запуска параллельной программы на C для исследования оптимальных параметров #pragma omp for schedule(...)

```
1.  #!/bin/bash
2.  #PBS -l walltime=00:30:00
3.  #PBS -l select=1:ncpus=4:ompthreads=4
4.  cd $PBS_O_WORKDIR
5.
6.  echo "OMP_NUM_THREADS = $OMP_NUM_THREADS"
7.
8.  echo "Static"
9.  echo
10.
11. for (( i = 1; i <= 10; i++ ))
12. do
13.     OMP_SCHEDULE="static, $i" ./omp_slae_schedule.out
14. done
15.
16. for (( i = 50; i <= 100; i+=50))
17. do
18.     OMP_SCHEDULE="static, $i" ./omp_slae_schedule.out
19. done
20.
21. for (( i = 500; i <= 1500; i+=500 ))
22. do
23.     OMP_SCHEDULE="static,$i" ./omp_slae_schedule.out
24. done
25. echo
26.
27. echo "Dynamic"
28. echo
29.
30. for (( i = 1; i <= 10; i++ ))
31. do
32.     OMP_SCHEDULE="dynamic,$i" ./omp_slae_schedule.out
33. done
34.
35. for (( i = 50; i <= 100; i+=50 ))
36. do
37.     OMP_SCHEDULE="dynamic,$i" ./omp_slae_schedule.out
38. done
39.
40. for (( i = 500; i <= 1500; i+=500 ))
41. do
42.     OMP_SCHEDULE="dynamic,$i" ./omp_slae_schedule.out
43. done
44. echo
```

```
45.
46. echo "Guided"
47. echo
48.
49. for (( i = 1; i <= 10; i++ ))
50. do
51.     OMP_SCHEDULE="guided,$i" ./omp_slae_schedule.out
52. done
53.
54. for (( i = 50; i <= 100; i+=50 ))
55. do
56.     OMP_SCHEDULE="guided,$i" ./omp_slae_schedule.out
57. done
58.
59. for (( i = 500; i <= 1500; i+=500 ))
60. do
61.     OMP_SCHEDULE="guided,$i" ./omp_slae_schedule.out
62. done
63. echo
64.
```