Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное образовательное учреждение высшего образования

“НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО”

Факультет Программной инженерии и компьютерной техники

Направление подготовки (специальность) Системное и прикладное ПО

ОТЧЕТ

Лабораторная работа №4

по предмету «Параллельные вычисления»

Тема проекта: «Метод доверительных интервалов при измерении времени выполнения параллельной OpenMP-программы».

Обучающийся Худяков А. А. P4115

(Фамилия И.О.) **(номер группы)**

Преподаватель Жданов А. Д.

(Фамилия И.О.)

Санкт-Петербург

2023 г.

Оглавление

Описание решаемой задачи

В программе, полученной в результате выполнение ЛР3 изменить этап Generate так, чтобы он не зависел от количества потоков. Произвести замену вызовов функции gettimeofday на omp_get_wtime. Распараллелить этап сортировки, разбив его на 2 этапа: 1) Отсортировать первую и вторую половины массива в двух независимых потоках; 2) Объединить отсортированные половины в единый массив. Также необходимо разработать функцию, которая один раз в секунду выводит в консоль сообщение о текущем проценте завершения работы программы. Кроме того, необходимо обеспечить прямую совместимость написанной параллельной программы, т.е. написать функции – заглушки вида omp_*.

Вариант решаемой задачи:

Этап Map

Массив M1

Номер варианта	Операция
6	Кубический корень после деления на число e

Массив M2

Номер варианта	Операция
2	Модуль косинуса

Этап Merge

Номер варианта	Операция
4	Выбор большего (т.е. M2[i] = max(M1[i],M2[i]))

Этап Sort

Номер варианта	Операция
6	Сортировка вставками (Insertion sort)

Характеристика используемого оборудования:

Процессор AMD Ryzen 5 5600H with Radeon Graphics 3.3 GHz (AMD64 Family 25 Model 80 Stepping 0 AuthenticAMD ~3301 МГц)

Операционная система Ubuntu 22.04.2 LTS

Количество физических ядер: 6

Количество логических ядер: 12

Версия GCC: 11.3.0

Оперативная память 16 Гб

L2 Cache 3MB

L3 Cache 16MB

Вся работа проводилась на виртуальной машине VMware Workstation Player 16, ресурсы, выделенные виртуальной машине:

Количество процессоров: 6

Оперативная память 6 Гб

Текст программы

Текст программы будет представлен в конце отчета.

Эксперименты

Задание 1

Проведем измерения времени выполнения и параллельного ускорения для различных значений N. Результаты представим в виде таблиц и графиков

Таблица 1 - Время выполнения при различных N

N\T	**seq**	**2**	**3**	**4**	**6**	**10**
1000	11	12	14	17	65	112
4100	113	82	86	87	143	173
7200	327	237	244	250	300	333

10300	653	457	470	465	515	565
13400	1086	787	754	781	824	858
16500	1633	1158	1146	1183	1205	1241
19600	2287	1703	1648	1676	1678	1743
22700	3059	2260	2250	2259	2233	2278
25800	3933	2766	2881	2876	2913	2950
28900	4925	3612	3592	3548	3611	3655
32000	6034	4378	4356	4360	4399	4436

Таблица 2 - Параллельное ускорение при различных N

N\T	**seq**	**2**	**3**	**4**	**6**	**10**
1000	1	0,916667	0,785714	0,647059	0,169231	0,098214
4100	1	1,378049	1,313953	1,298851	0,79021	0,653179
7200	1	1,379747	1,340164	1,308	1,09	0,981982
10300	1	1,428884	1,389362	1,404301	1,267961	1,155752
13400	1	1,379924	1,440318	1,390525	1,317961	1,265734
16500	1	1,41019	1,424956	1,380389	1,355187	1,315874
19600	1	1,342924	1,387743	1,364558	1,362932	1,312106
22700	1	1,35354	1,359556	1,354139	1,369906	1,342845
25800	1	1,421909	1,365151	1,367524	1,350154	1,33322
28900	1	1,363511	1,371102	1,388106	1,363888	1,347469
32000	1	1,378255	1,385216	1,383945	1,371675	1,360234

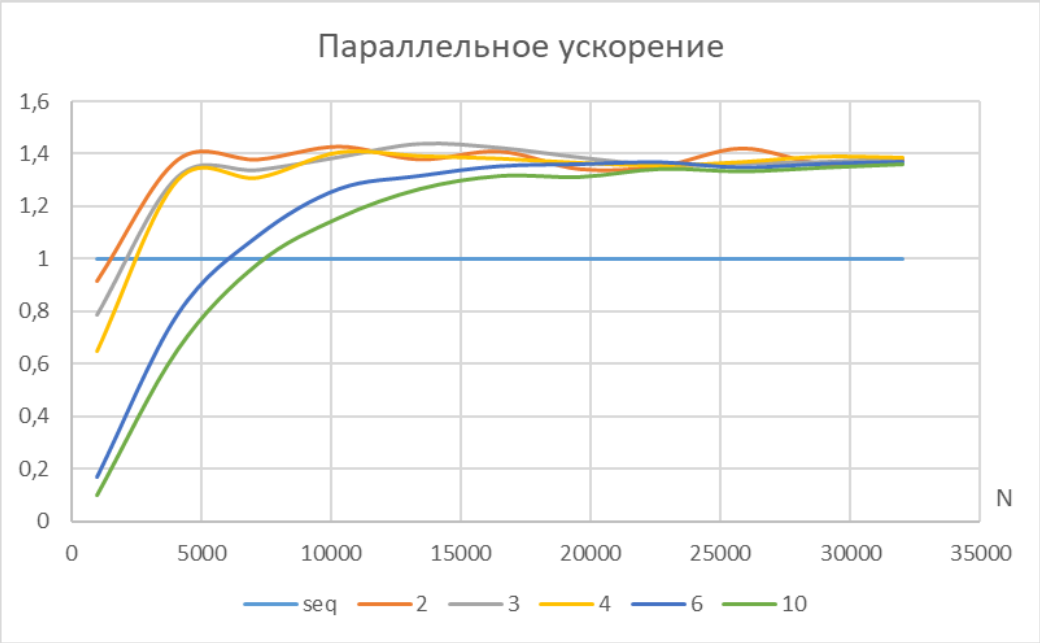


Рисунок 1 - График параллельного ускорения

В результате распараллеливания этапа сортировки, путем разбиения массива на 2 части и сортировки этих частей в независимых потоках удалось достичь прироста параллельного ускорения. Но этот прирост нельзя назвать

значительным, в самом лучшем случае наблюдает ускорение примерно в 1,5 раза.

Необязательное задание 1

Уменьшим число итераций с 100 до 10 и замеряем время двумя способами:

1) Использование минимального из 10 полученных замеров

Таблица 3 - Время выполнения при различных N

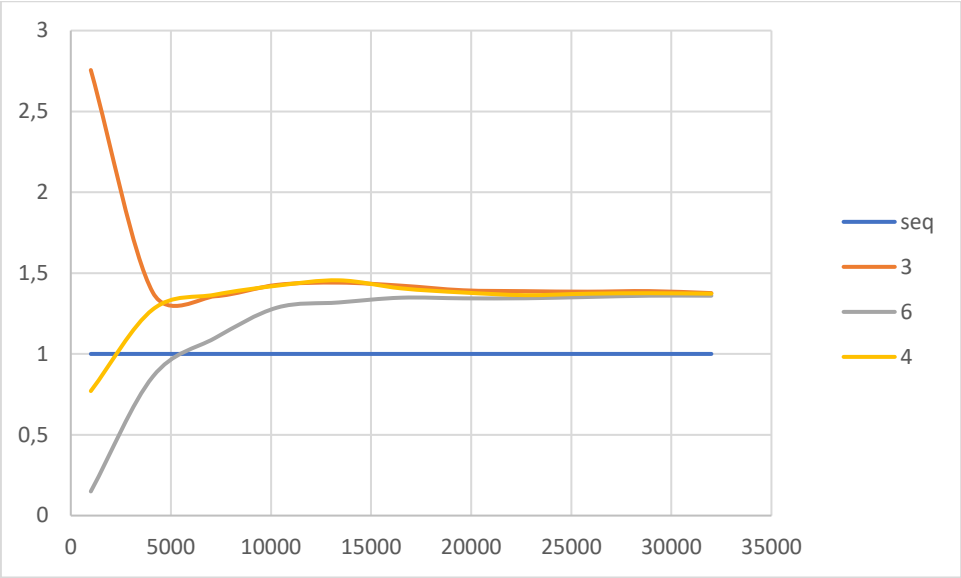
N\T	**seq**	**2**	**3**	**4**	**6**	**10**
1000	1	1	1	1	6	9
4100	11	7	8	8	12	16
7200	32	23	23	23	29	33
10300	64	45	45	45	50	55
13400	108	74	74	76	81	85
16500	159	110	111	113	118	121
19600	226	161	159	160	168	173
22700	301	213	212	219	223	224
25800	390	281	276	280	287	293
28900	483	341	344	353	354	357
32000	598	419	428	432	435	438

Таблица 4 - Параллельное ускорение при различных N

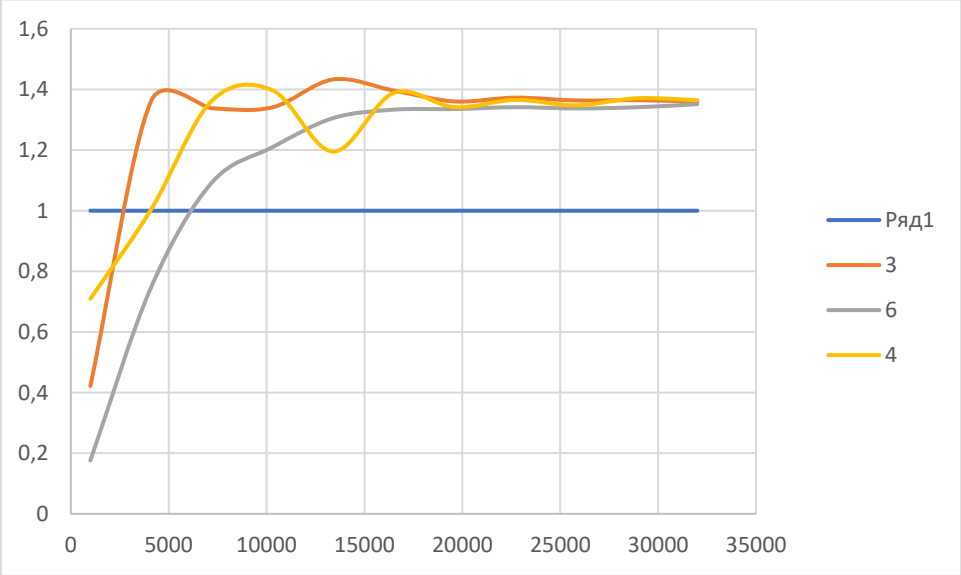
N\T	**seq**	**2**	**3**	**4**	**6**	**10**
1000	1	1	1	1	0,166667	0,111111
4100	1	1,571429	1,375	1,375	0,916667	0,6875
7200	1	1,391304	1,391304	1,391304	1,103448	0,969697
10300	1	1,422222	1,422222	1,422222	1,28	1,163636
13400	1	1,459459	1,459459	1,421053	1,333333	1,270588
16500	1	1,445455	1,432432	1,40708	1,347458	1,31405
19600	1	1,403727	1,421384	1,4125	1,345238	1,306358
22700	1	1,413146	1,419811	1,374429	1,349776	1,34375
25800	1	1,3879	1,413043	1,392857	1,358885	1,331058
28900	1	1,416422	1,40407	1,368272	1,364407	1,352941
32000	1	1,427208	1,397196	1,384259	1,374713	1,365297

2) По доверительным интервалам

Нижняя граница



Верхняя граница



Необязательное задание 2

Выполним параллельную сортировку не двух частей массива, а k частей в k потоках, где k – количество процессоров (ядер в системе), которое становится известным только на этапе выполнения программы с помощью команды «k = omp_get_num_procs()».

Представим время выполнения программы и параллельное ускорение для различных значений N

Таблица 5 - Время выполнения для различных N

N\T	seq	2	3	4	6	10
1000	11	9	10	16	63	111
4100	113	24	25	29	71	123
7200	327	50	50	51	109	169
10300	653	85	86	85	146	217
13400	1086	126	122	126	190	287
16500	1633	175	177	181	253	385
19600	2287	235	237	243	317	485
22700	3059	305	302	310	394	632
25800	3933	373	397	383	479	777
28900	4925	469	453	501	592	950
32000	6034	570	546	550	698	1106

Таблица 6 - Параллельное ускорение для различных N

N\T	seq	2	3	4	6	10
1000	1	1,222222	1,1	0,6875	0,174603	0,099099
4100	1	4,708333	4,52	3,896552	1,591549	0,918699
7200	1	6,54	6,54	6,411765	3	1,934911
10300	1	7,682353	7,593023	7,682353	4,472603	3,009217
13400	1	8,619048	8,901639	8,619048	5,715789	3,783972
16500	1	9,331429	9,225989	9,022099	6,454545	4,241558
19600	1	9,731915	9,649789	9,411523	7,214511	4,715464
22700	1	10,02951	10,12914	9,867742	7,763959	4,84019
25800	1	10,54424	9,906801	10,26893	8,210856	5,061776
28900	1	10,50107	10,87196	9,830339	8,319257	5,184211
32000	1	10,58596	11,05128	10,97091	8,644699	5,455696

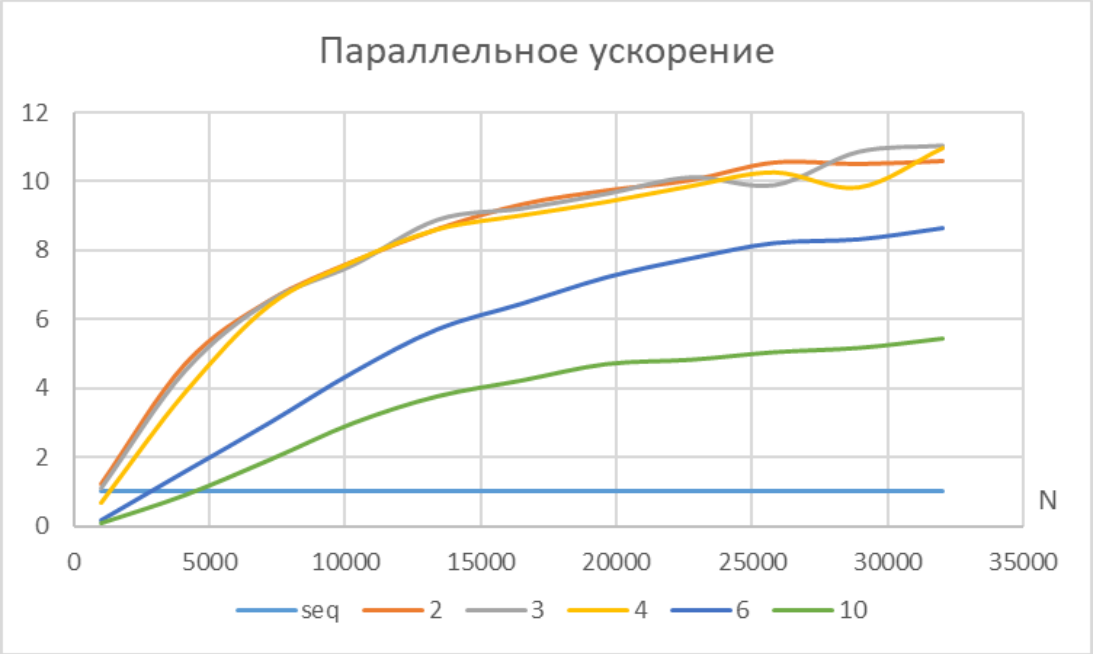


Рисунок 2 - График параллельного ускорения

Заключение

В результате выполнения лабораторной работы были выполнены следующие задачи: 1) Написана программа с генерацией данных, не зависящих от числа потоков; 2) Распараллелена сортировка путем разделения массива на 2 части и сортировки каждой из этих частей отдельно. Данное действие привело к увеличению параллельного ускорения, но большого прироста добиться не удалось, ускорение не превышает 1.5. Далее были проведены измерения времени двумя способами: выбор наименьшего значения из 10 измерений и расчет доверительного интервала с уровнем доверия 95%. Кроме того, было улучшено распараллеливание сортировки, теперь массив разделяется на k частей, где k - количество процессоров (ядер) в системе. Данная операция позволила повысить быстродействие программы и максимальное значение параллельного ускорения составило примерно 11.

Приложение – код программы

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. #include <sys/time.h>
 4. #include <math.h>
 5. #include <unistd.h>
 6.
 7. #ifdef _OPENMP
 8. #include <omp.h>
 9. #else
10. int omp_get_max_threads()
11. {
12.     return 1;
13. }
14.
15. int omp_get_num_procs()
16. {
17.     return 1;
18. }
19.
20. int omp_get_thread_num()
21. {
22.     return 0;
23. }
24.
25. void omp_set_num_threads(int thrds)
26. {
27.     return;
28. }
29.
30. double omp_get_wtime()
31. {
32.     struct timeval T;
33.     double time_ms;
34.
35.     gettimeofday(&T, NULL);
36.     time_ms = (1000.0 * ((double)T.tv_sec) + ((double)T.tv_usec)
/ 1000.0);
37.     return (double)(time_ms / 1000.0);
38. }
39.
40. void omp_set_nested(int b)
41. {
42.     return;
43. }
44. #endif
45.
46.
47. void generate_array(double *restrict m, int size, unsigned int
min, unsigned int max, int seed)
```

```
48. {
49.     //int tmp_seed;
50.     // #pragma omp for private(tmp_seed)
51.     for (int i = 0; i < size; ++i)
52.     {
53.         unsigned int tmp_seed = sqrt(i + seed);
54.         m[i] = ((double)rand_r(&tmp_seed) / (RAND_MAX)) * (max -
min) + min;
55.     }
56. }
57.
58. double max_el(double *restrict a, double *restrict b)
59. {
60.     return (*a) > (*b) ? (*a) : (*b);
61. }
62.
63. int min_el(int *restrict a, int * restrict b)
64. {
65.     return (*a) < (*b) ? (*a) : (*b);
66. }
67.
68. void copy_array(double *dst, double *src, int n) {
69.     for (int i = 0; i < n; ++i)
70.     {
71.         dst[i] = src[i];
72.     }
73. }
74.
75. void insert_sort(double *restrict M, int from, int to, int size)
76. {
77.     int key = 0;
78.     double temp = 0.0;
79.     for (int k = from; k < to-1; k++)
80.     {
81.         key = k + 1;
82.         temp = M[key];
83.         for (int j = k + 1; j > from; j--)
84.         {
85.             if (temp < M[j - 1])
86.             {
87.                 M[j] = M[j - 1];
88.                 key = j - 1;
89.             }
90.         }
91.         M[key] = temp;
92.     }
93. }
94.
95. void merge_sorted(double *src1, int n1, double *src2, int n2,
double *dst) {
96.     int i = 0, i1 = 0, i2 = 0;
```

```
 97.     while (i < n1 + n2) {
 98.         dst[i++] = src1[i1] > src2[i2] && i2 < n2 ? src2[i2++] :
src1[i1++];
 99.     }
100. }
101.
102. void merge_sort(double * restrict MM, int size, double *restrict
dst){
103.     int n_threads = omp_get_num_procs();
104.     int chunk_size = size / n_threads;
105.             int tid = omp_get_thread_num();
106.             int start = tid * chunk_size;
107.             int end = (tid == n_threads -1) ? size + 1 : (tid+1)
* chunk_size;
108.             insert_sort(MM, start, end, size);
109.     #pragma omp single
110.     {
111.         double * restrict cpy = malloc(size * sizeof(double));
112.
113.         copy_array(cpy, MM, size);
114.         copy_array(dst, MM, size);
115.         for (int k = 1; k < n_threads; ++k)
116.         {
117.             int n_done = chunk_size * k;
118.             int vsp = size - n_done;
119.             int n_cur_chunk = min_el(&(vsp), &(chunk_size));
120.             if(k==n_threads-1)
121.             {
122.                 n_cur_chunk = size - n_done;
123.             }
124.             int n_will_done = n_done + n_cur_chunk;
125.             merge_sorted(cpy, n_done, MM + n_done, n_cur_chunk,
dst);
126.             copy_array(cpy, dst, n_will_done);
127.         }
128.     }
129. }
130.
131. void merge_sort_halves(double * restrict MM, int size, double
*restrict dst, int num_threads)
132. {
133.     int n1 = size / 2;
134.     omp_set_num_threads(2);
135.     #pragma omp sections
136.     {
137.         #pragma omp section
138.         insert_sort(MM, 0, n1, size);
139.         #pragma omp section
140.         insert_sort(MM,n1, size + 1, size);
141.     }
142.     #pragma omp single
```

```
143.     merge_sorted(MM, n1, MM + n1, size - n1, dst);
144.     omp_set_num_threads(num_threads);
145. }
146.
147. int mainpart(int argc, char *argv[],int* progress, int *i)
148. {
149.     int N;
150.     double T1,T2;
151.     long long delta_ms;
152.     double Abeg = 1.0;
153.     double A = 315.0;
154.     double Aend = A * 10.0;
155.     unsigned int seed;
156.     N = atoi(argv[1]);       /* N равен первому параметру
командной строки */
157.     T1 = omp_get_wtime(); /* запомнить текущее время T1 */
158.     int N_2 = N / 2;
159.     double *restrict M1 = malloc(N * sizeof(double));
160.     double *restrict M2 = malloc(N_2 * sizeof(double));
161.     double *restrict M2_old = malloc(N_2 * sizeof(double));
162.     double *restrict M2_sorted = malloc(N_2 * sizeof(double));
163.     const int num_threads = atoi(argv[2]); /* amount of threads
*/
164.     #if defined(_OPENMP)
165.         omp_set_dynamic(0);
166.         omp_set_num_threads(num_threads);
167.     #endif
168.
169.     double expon = exp(1);
170.
171.     //for (int j = 0; j < 100; j++) /* 100 экспериментов */
172.     for (int j = 0; j < 10; j++) /* 10 экспериментов */
173.     {
174.         /* инициализировать начальное значение ГСЧ */
175.         double X = 0.0;
176.         seed = j;
177.         *i = j;
178.         // GENERATE
179.         /* Заполнить массив исходных данных размером N */
180.         generate_array(M1, N, Abeg, A, seed);
181.         /* Заполнить массив исходных данных размером N/2 */
182.         generate_array(M2, N_2, A, Aend, seed+2);
183.         #pragma omp parallel default(none) shared(N, N_2, M1, M2,
M2_old, M2_sorted, X, expon,  A, Abeg, Aend, num_threads)
184.         {
185.         // MAP
186.         #pragma omp for
187.             for (int j = 0; j < N; j++)
188.             {
189.                 M1[j] = cbrt(M1[j] / expon);
190.             }
```

```
191.
192.            #pragma omp for nowait
193.                for (int k = 0; k < N_2; k++)
194.                {
195.                    M2_old[k] = M2[k];
196.                }
197.            #pragma omp single
198.                M2[0] = fabs(cos(M2[0]));
199.
200.            #pragma omp for
201.                for (int k = 1; k < N_2; k++)
202.                {
203.                    M2[k] = M2[k] + M2_old[k - 1];
204.                }
205.
206.            #pragma omp for
207.                for (int k = 1; k < N_2; ++k)
208.                {
209.                    M2[k] = fabs(cos(M2[k]));
210.                }
211.
212.            //  MERGE
213.            #pragma omp for
214.                for (int k = 0; k < N_2; k++)
215.                {
216.                    M2[k] = max_el(&M1[k], &M2[k]);
217.                }
218.                /* Отсортировать массив с результатами указанным
    методом */
219.                // SORT
220.                if(num_threads == 2)
221.                 {
222.                     merge_sort_halves(M2,N_2, M2_sorted);
223.                 }
224.                 else
225.                 {
226.                     merge_sort(M2, N_2, M2_sorted);
227.                 }
228.
229.                // REDUCE
230.
231.                int k = 0;
232.                while (M2_sorted[k] == 0 && k < N_2 - 1)
233.                    k++;
234.                double minelem = M2_sorted[k];
235.                //  sum of matching array elements
236.                #pragma omp for
237.                for (int k = 0; k < N_2; k++)
238.                {
239.                    M2_old[k] = 0.0;
240.                    if ((int)(M2_sorted[k] / minelem) % 2 == 0)
```

```
241.                 {
242.                     M2_old[k] = sin(M2_sorted[k]);
243.                 }
244.             }
245.
246.             #pragma omp for reduction(+: X)
247.             for (int j = 0; j < N_2; ++j)
248.             {
249.                 X += M2_old[j];
250.             }
251.         }
252.         // printf("X = %f ", X);
253.         //printf("%f", X);
254.         //printf("\n\n");
255.     }
256.     *progress = 1;
257.     //gettimeofday(&T2, NULL); /* запомнить текущее время T2 */
258.     T2 = omp_get_wtime();
259.     //delta_ms = 1000 * (T2.tv_sec - T1.tv_sec) + (T2.tv_usec -
T1.tv_usec) / 1000;
260.     delta_ms = 1000* (T2 - T1);
261.     // printf("\nN=%d. Milliseconds passed: %ld\n", N, delta_ms);
/* T2 - T1 */
262.     printf("%lld\n", delta_ms);
263.     // printf("%lld\n", delta_ms);
264.     return 0;
265. }
266.
267. void progressnotifier(int *progress, int *i)
268. {
269.     double time = 0;
270.     while (*progress < 1)
271.     {
272.         double time_temp = omp_get_wtime();
273.         if (time_temp - time < 1)
274.         {
275.             usleep(100);
276.             continue;
277.         };
278.         //printf("\nPROGRESS: %d\n", *i);
279.         time = time_temp;
280.     }
281. }
282.
283. int main(int argc, char *argv[])
284. {
285.     double T1, T2;
286.
287.     int *progress = malloc(sizeof(int));
288.     *progress = 0;
289.     int *i = malloc(sizeof(int));
```

```
290.     *i = 0;
291.     omp_set_nested(1);
292.     #pragma omp parallel sections num_threads(2) shared(i,
progress)
293.     {
294.         #pragma omp section
295.         progressnotifier(progress, i);
296.         #pragma omp section
297.         mainpart(argc, argv, progress, i);
298.     }
299.     return 0;
300. }
301.
```