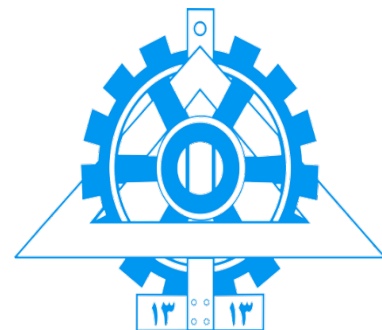


به نام خدا



دانشگاه تهران  
دانشکده مهندسی برق و کامپیوتر



# تمرین کامپیوتری ۲

برنامه نویسی موازی

دکتر صفری

اعضا گروه:

محمد تقی زاده گیوری ۸۱۰۱۹۸۳۷۳

نیلوفر محمدی ۸۱۰۱۹۶۶۸۷

نیمسال اول ۱۴۰۱-۰۲

## سوال ۱: پیاده سازی سریال

ابتدا بزرگ ترین عنصر را برابر با اولین عنصر آرایه قرار می دهیم.

برای پیدا کردن بزرگترین عنصر آرایه و اندیس آن، در یک حلقه کل اعضا آرایه را پیمایش می کنیم:

به هر عنصر که می‌رسیم، بررسی می‌کنیم که آیا این عنصر از بزرگ ترین عنصری که تا حالا پیدا شده

(که در ابتدا اولین عنصر آرایه است)، بزرگ تر هست یا نه. اگر بزرگ تر بود پس بزرگ ترین عنصر (max\_serial)

و اندیس آن (max\_index\_serial)، را آپدیت می‌کنیم.

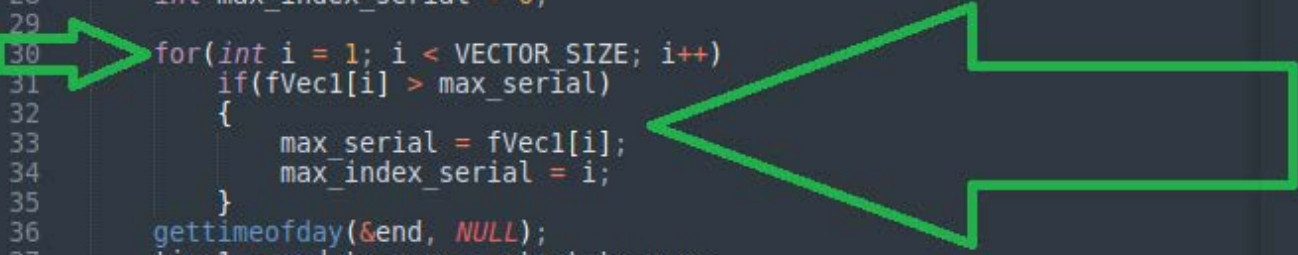
در نتیجه پس از پیمایش کل آرایه، بزرگ ترین عنصر و اندیس آن به ترتیب در متغیر های (max\_serial)

و (max\_index\_serial)، ذخیره می‌شوند.

```

13
14     float *fVec1;
15     fVec1 = new float [VECTOR_SIZE];
16
17     if (!fVec1) {
18         printf ("Memory allocation error!!\n");
19         return 1;
20     }
21     // Initialize vectors with random numbers
22     for (long i = 0; i < VECTOR_SIZE; i++)
23         fVec1[i] = static_cast<float> (rand()) / (static_cast<float> (RA
24
25     // Serial implementation
26     gettimeofday(&start, NULL);
27     float max_serial = fVec1[0];
28     int max_index_serial = 0;
29
30     for(int i = 1; i < VECTOR_SIZE; i++)
31         if(fVec1[i] > max_serial)
32         {
33             max_serial = fVec1[i];
34             max_index_serial = i;
35         }
36     gettimeofday(&end, NULL);
37     time1 = end.tv_usec - start.tv_usec;
38

```



## سوال ۱: پیاده سازی موازی

ابتدا بزرگ ترین عنصر را برابر با اولین عنصر آرایه قرار می دهیم.

پیاده سازی موازی مشابه پیاده سازی سریال است، با این تفاوت که حلقه ای که کل آرایه را پیمایش می کند، بین

چند ریسمان (thread)، توزیع شده و هر ریسمان (thread)، بخشی از تکرار های حلقه را انجام می دهد.

به این ترتیب با استفاده از `#pragma omp parallel`، تعدادی ریسمان (thread)، می سازیم. سپس

با استفاده از `#pragma omp for`، تکرار های حلقه را بین ریسمان ها توزیع می کنیم.

لازم به ذکر است که آرایه و بزرگ ترین عنصر و اندیس آن، باید به صورت `shared` تعریف شوند. چون هر

ریسمان ممکن است بخواهد مقدار بزرگ ترین عنصر و اندیس آن را حین پیمایشی که انجام میدهد، تغییر

دهد و آپدیت کند. آرایه هم باید به صورت `shared` تعریف شود، چون تمامی ریسمان ها نیاز به دسترسی

به عناصر آرایه دارند. تنها متغیر `i`، باید به صورت `private`، تعریف شود، چون هر ریسمان بخشی از حلقه

را انجام می دهد (مثلا از  $i = 12$  تا  $i = 15$ ) و این بخش ها مستقل از هم هستند و ربطی به هم ندارند.

```

38 // Paralle implementation
39
40 gettimeofday(&start, NULL);
41
42 float max_parallel = fVec1[0];
43 int max_index_parallel = 0;
44 int i;
45
46 #pragma omp parallel shared(max_parallel, max_index_parallel, fVec1) p
47 {
48     #pragma omp for
49     for (i = 1; i < VECTOR_SIZE; i++)
50         if (fVec1[i] > max_parallel)
51         {
52             max_parallel = fVec1[i];
53             max_index_parallel = i;
54         }
55     }
56 }
57
58 /*-- End of parallel region --*/
59
60 gettimeofday(&end, NULL);
61 time2 = end.tv_usec - start.tv_usec;
62
63

```

## سوال ۱: خروجی پیاده سازی سریال و موازی

در تصویر زیر، ابتدا خروجی های پیاده سازی سریال و موازی چاپ می شوند. سپس زمان اجرا پیاده سازی سریال و موازی چاپ شده و میزان تسریع، با تقسیم زمان سریال بر زمان موازی، محاسبه شده و چاپ می شود.

```

63
64     printf ("\nSerial Max = %f , index = %d\n", max_serial, max_index_serial);
65     printf ("Parallel Max = %f", max_parallel);
66     printf (" , index = %d\n", max_index_parallel);
67     printf ("Serial Run time = %ld u seconds\n", time1);
68     printf ("Parallel Run time = %ld u seconds\n", time2);
69     printf ("Speedup = %f\n\n", (float) (time1)/(float) time2);
70
71     return 0;

```

```

ariyataghizadeh@ariyataghizadeh-VirtualBox:~/Parallel_Programming/CA_2&3/CA_2/Q
1$ ./main

```

```

Serial Max = 99.999832 , index = 245298
Parallel Max = 99.999832, index = 245298
Serial Run time = 7972 u seconds
Parallel Run time = 3412 u seconds
Speedup = 2.336460

```

```

ariyataghizadeh@ariyataghizadeh-VirtualBox:~/Parallel_Programming/CA_2&3/CA_2/Q
1$ ./main

```

```

Serial Max = 99.999832 , index = 245298
Parallel Max = 99.999832, index = 245298
Serial Run time = 11538 u seconds
Parallel Run time = 3514 u seconds
Speedup = 3.283438

```

**سوال ۲: پیاده سازی سریال**

ابتدا یک عنصر از آرایه را به عنوان محور (pivot)، در نظر می گیریم که این عنصر را آخرین عنصر آرایه در نظر گرفتیم. سپس در طی یک حلقه، عناصری که کوچک تر از pivot هستند را به سمت چپ عنصر pivot، منتقل می کنیم و عناصری که بزرگ تر از pivot هستند، در سمت راست عنصر pivot قرار می گیرند. در نتیجه با فراخوانی بازگشتی quick\_sort\_serial، ابتدا زیر آرایه سمت چپ عنصر pivot، را مرتب می کنیم، سپس با فراخوانی بازگشتی quick\_sort\_serial، زیر آرایه سمت راست عنصر pivot، را مرتب می کنیم. در نتیجه آرایه به صورت صعودی مرتب می شود.

```
32 void quick_sort_serial(float *array, int start_index, int end_index)
33 {
34     if(start_index >= end_index)
35         return;
36
37     int pivot_index = partitioning_serial(array, start_index, end_index);
38
39     quick_sort_serial(array, start_index, pivot_index - 1);
40
41     quick_sort_serial(array, pivot_index + 1, end_index);
42 }
43
```

در تابع `partitioning_serial`، آرایه را طوری تغییر می دهیم که عناصر کوچک تر از `pivot`، سمت چپ `pivot` قرار گیرند و عناصر بزرگ تر از `pivot`، سمت راست `pivot` قرار گیرند. اندیس `pivot`، بیانگر موقعیت `pivot`، پس از تغییر آرایه است.

برای اینکه عناصر کوچک تر قبل از `pivot` قرار گیرند:

ابتدا اندیس `pivot` را برابر با اولین عنصر آرایه قرار می دهیم سپس طی یک حلقه کل آرایه را پیمایش می کنیم، در هر پیمایش، اگر عنصری که بررسی می کنیم از `pivot`، کوچک تر باشد، پس آن را با `array[pivot_index]` جابجا می کنیم. تا عنصر بررسی شده (که کوچک تر از `pivot` است)، به سمت چپ `pivot` منتقل شود. در نتیجه `pivot_index` را یک واحد اضافه می کنیم.

```

1  #include <stdio.h>
2  #include "sys/time.h"
3  #include "unistd.h"
4  #include "stdlib.h"
5  #include "omp.h"
6
7  #define VECTOR_SIZE 1048576 // 2^20
8
9  int partitioning_serial(float *array, int start_index, int end_index)
10 {
11     float pivot = array[end_index];
12     int pivot_index = start_index;
13     for(int i = start_index; i < end_index; i++)
14     {
15         if(array[i] <= pivot)
16         {
17             float temp = array[pivot_index];
18             array[pivot_index] = array[i];
19             array[i] = temp;
20             pivot_index++;
21         }
22     }
23     float temp = array[pivot_index];
24     array[pivot_index] = array[end_index];
25     array[end_index] = temp;
26     return pivot_index;
27 }

```



## سوال ۲: پیاده سازی موازی

پیاده سازی موازی مشابه پیاده سازی سریال است، با این تفاوت که:

بعد از اینکه با فراخوانی تابع `partitioning` کاری کردیم که عناصر کوچک تر از `pivot`، سمت چپ `pivot`، قرار گیرند،

و آرایه به صورت دو زیر آرایه درآمد، مرتب سازی زیر آرایه سمت راست `pivot` را به یک ریسمان واگذار می کنیم و

مرتب سازی زیر آرایه سمت چپ `pivot` را به یک ریسمان دیگر واگذار میکنیم تا مرتب سازی دو زیر آرایه به صورت

موازی اجرا شود. این کار را با استفاده از `#pragma omp sections` و `#pragma omp section`

انجام می دهیم.

```

66
67 void quick_sort_parallel(float *array, int start_index, int end_index)
68 {
69     if(start_index >= end_index)
70         return;
71
72     int pivot_index = partitioning_parallel(array, start_index, end_index);
73
74     #pragma omp parallel sections
75     {
76
77         #pragma omp section
78         quick_sort_parallel(array, start_index, pivot_index - 1);
79
80         #pragma omp section
81         quick_sort_parallel(array, pivot_index + 1, end_index);
82
83     }
84 }
85

```

**سوال ۲: خروجی پیاده سازی سریال و موازی**

در تصویر زیر، ابتدا زمان اجرا پیاده سازی سریال و موازی چاپ می شوند. سپس میزان تسریع، با تقسیم زمان سریال بر زمان موازی، محاسبه شده و چاپ می شود.

```
127  
128     printf ("Serial Run time = %ld u seconds\n", time1);  
129     printf ("Parallel Run time = %ld u seconds\n", time2);  
130     printf ("Speedup = %f\n\n", (float) (time1)/(float) time2);  
131  
132     return 0;
```

```
ariyataghizadeh@ariyataghizadeh-VirtualBox:~/Parallel_Programming/CA_2&3/CA_2/Q  
2$ ./main  
Serial Run time = 253195 u seconds  
Parallel Run time = 48367 u seconds  
Speedup = 5
```

```
ariyataghizadeh@ariyataghizadeh-VirtualBox:~/Parallel_Programming/CA_2&3/CA_2/Q  
2$ ./main  
Serial Run time = 241863 u seconds  
Parallel Run time = 28100 u seconds  
Speedup = 8
```



**سوال ۳: محاسبه زمان اجرا پیاده سازی سریال**

ابتدا با فراخوانی تابع `timeGetTime`، زمان شروع اجرا پیاده سازی سریال را محاسبه می کنیم.

```

29
30 // repeat experiment several times
31 for( i=0; i<1; i++ ){
32     // get starting time
33     starttime = timeGetTime();

```

سپس در انتها کد سریال، با فراخوانی تابع `timeGetTime`، زمان کنونی محاسبه شده، و با کم کردن زمان شروع، از آن،

مدت زمان اجرا پیاده سازی سریال محاسبه می شود.

```

54
55 // get ending time and use it to determine elapsed time
56 elapsedtime_serial = timeGetTime() - starttime;
57
58 // report elapsed time
59 printf("Time Elapsed Serial %10d mSecs Total=%lf Check Sum = %ld\n",
60        (int)(elapsedtime_serial * 1000), total, sum );

```

**سوال ۳: محاسبه متوسط زمان اجرا پیاده سازی موازی**

برای محاسبه متوسط زمان اجرا پیاده سازی موازی، یک آرایه به طول ۶ (تعداد دفعات اجرا حلقه بیرونی)، در نظر می گیریم و زمان اجرا هر تکرار حلقه بیرونی را در آن ذخیره می کنیم تا بعداً مجموع زمان های اجرا را محاسبه و متوسط زمان اجرا پیاده سازی موازی را محاسبه کنیم.

```

21 long int j, k, sum;
22 double sumx, sumy, total, z;
23 double starttime, starttime_thread, elapsedtime_serial;
24 double elapsedtime_parallel[6];
25 double elapsedtime_thread[4] = {0};

```

ابتدا در هر بار اجرا حلقه بیرونی، با فراخوانی تابع `timeGetTime`، زمان شروع اجرا پیاده سازی موازی را محاسبه می کنیم.

```

69 for( i=0; i<6; i++ )
70 {
71     // get starting time
72     starttime = timeGetTime();

```

سپس در انتها حلقه بیرونی، با فراخوانی تابع `timeGetTime`، زمان کنونی محاسبه شده، و با کم کردن زمان شروع، از آن، مدت زمان اجرا پیاده سازی موازی به ازای یک بار اجرا حلقه محاسبه می شود.

```

106
107 // get ending time and use it to determine elapsed time
108 elapsedtime_parallel[i] = timeGetTime() - starttime;
109
110 }
111

```

در آخر با یک حلقه به طول ۶، مجموع زمان های اجرا پیاده سازی موازی را بدست آورده، و با تقسیم آن، بر ۶، متوسط زمان اجرا پیاده سازی موازی محاسبه می شود.

```
111
112     double sum_of_parallel_elapsedtime = 0;
113
114     for(int n = 0; n < 6; n++)
115         sum_of_parallel_elapsedtime += elapsedtime_parallel[n];
116
117     double mean_parallel_elapsed_time = sum_of_parallel_elapsedtime/6;
118
```

## سوال ۳: محاسبه زمان اجرا هر ریسمان

برای محاسبه زمان اجرا هر ریسمان، یک آرایه به طول ۴ (تعداد ریسمان‌ها)، در نظر می‌گیریم و زمان اجرا هر ریسمان را در آن ذخیره می‌کنیم. از جایی که در ابتدا هیچ ریسمانی اجرا نشده است، در نتیجه مدت زمان اجرا هر ریسمان برابر با 0 بوده و مقدار اولیه این آرایه را برابر با 0 در نظر می‌گیریم.

```

21 long int j, k, sum;
22 double sumx, sumy, total, z;
23 double starttime, starttime_thread, elapsedtime_serial;
24 double elapsedtime_parallel[6];
25 double elapsedtime_thread[4] = {0};

```

ابتدا در هر بار اجرا حلقه for (که به صورت موازی اجرا می‌شود)، با فراخوانی تابع `timeGetTime`، شروع زمان را محاسبه می‌کنیم.

```

84 for( int j=0; j<VERYBIG; j++ )
85 {
86     starttime_thread = timeGetTime();
87

```

سپس در انتها حلقه for (که به صورت موازی اجرا می‌شود)، با فراخوانی تابع `timeGetTime`، زمان کنونی محاسبه شده، و با کم کردن زمان شروع، از آن، مدت زمان اجرا یک تکرار حلقه محاسبه می‌شود. زمان محاسبه شده مربوط به اجرا یک تکرار حلقه بوده که توسط `thread` شماره `omp_get_thread_num()` اجرا شده است. پس مدت زمان محاسبه شده مربوط به `thread` شماره `omp_get_thread_num()` بوده و به مدت زمان اجرا ریسمان شماره `omp_get_thread_num()` اضافه می‌شود.

```

103
104     elapsedtime_thread[omp_get_thread_num()] += (timeGetTime() - star
105 }
106

```

در نتیجه پس از اجرا حلقه for، زمان اجرا هر ریسمان در آرایه `elapsedtime_thread` موجود می‌باشد.

**سوال ۳: خروجی پیاده سازی سریال و موازی**

در ابتدا خروجی سریال و زمان اجرا سریال چاپ می شود.

```

54
55 // get ending time and use it to determine elapsed time
56 elapsedtime_serial = timeGetTime() - starttime;
57
58 // report elapsed time
59 printf("Time Elapsed Serial %10d mSecs Total=%lf Check Sum = %ld\n",
60        (int)(elapsedtime_serial * 1000), total, sum );

```

سپس زمان اجرا هر ریسمان در پیاده سازی موازی، چاپ می شود، متوسط زمان اجرا موازی و خروجی موازی، چاپ شده در آخر میزان تسريع، با تقسیم زمان سریال بر متوسط زمان موازی، محاسبه شده و چاپ می شود.

```

119 // report each thread elapsed time
120 printf("Time Elapsed thread 0 %10d mSecs\n",
121        (int)(elapsedtime_thread[0] * 1000));
122
123 printf("Time Elapsed thread 1 %10d mSecs\n",
124        (int)(elapsedtime_thread[1] * 1000));
125
126 printf("Time Elapsed thread 2 %10d mSecs\n",
127        (int)(elapsedtime_thread[2] * 1000));
128
129 printf("Time Elapsed thread 3 %10d mSecs\n",
130        (int)(elapsedtime_thread[3] * 1000));
131
132 // report mean parallel elapsed time
133 printf("Time Elapsed Parallel %10d mSecs Total=%lf Check Sum = %ld\n\n",
134        (int)(mean_parallel_elapsed_time * 1000), total, sum )
135
136 printf("Speedup = %f\n\n", (float) (elapsedtime_serial)/(float) mean_pa
137

```

```

ariyataghizadeh@ariyataghizadeh-VirtualBox:~/Parallel_Programming/CA_2&3/CA_2/Q
3/static$ ./main
Serial timing for 100000 iterations

Time Elapsed Serial      31061 mSecs Total=32.617277 Check Sum = 100000

OpenMP Parallel Timings for 100000 iterations

Time Elapsed thread 0      11969 mSecs
Time Elapsed thread 1      35762 mSecs
Time Elapsed thread 2      58749 mSecs
Time Elapsed thread 3      80435 mSecs
Time Elapsed Parallel      31158 mSecs Total=32.617277 Check Sum = 100000

Speedup = 0.996890

```

```

ariyataghizadeh@ariyataghizadeh-VirtualBox:~/Parallel_Programming/CA_2&3/CA_2/Q
3/static$ ./main
Serial timing for 100000 iterations

Time Elapsed Serial      31232 mSecs Total=32.617277 Check Sum = 100000

OpenMP Parallel Timings for 100000 iterations

Time Elapsed thread 0      11976 mSecs
Time Elapsed thread 1      35590 mSecs
Time Elapsed thread 2      59125 mSecs
Time Elapsed thread 3      81105 mSecs
Time Elapsed Parallel      31307 mSecs Total=32.617277 Check Sum = 100000

Speedup = 0.997582

```

در حالت `schedule (static)`، چون به `thread` های اولی، مقادیر کمی از `i` می رسد، حجم کار `thread` های اولی (مانند `thread` های 0 و 1)، کم است. در حالی که مقادیر بزرگتری از `i`، به `thread` های آخری می رسد و در نتیجه حجم کار `thread` های آخری (مانند `thread` های 2 و 3) زیاد است. به همین دلیل مشاهده می کنید که زمان اجرای ریسمان های اولی کم و زمان اجرا ریسمان های آخری بیش تر است.



```

ariyataghizadeh@ariyataghizadeh-VirtualBox:~/Parallel_Programming/CA_2&3/CA_2/Q
3/dynamic_1000$ ./main
Serial timing for 100000 iterations

Time Elapsed Serial      31168 mSecs Total=32.617277 Check Sum = 100000

OpenMP Parallel Timings for 100000 iterations

Time Elapsed thread 0    46369 mSecs
Time Elapsed thread 1    46146 mSecs
Time Elapsed thread 2    47072 mSecs
Time Elapsed thread 3    46925 mSecs
Time Elapsed Parallel    31092 mSecs Total=32.617277 Check Sum = 100000

Speedup = 1.002458

```

```

ariyataghizadeh@ariyataghizadeh-VirtualBox:~/Parallel_Programming/CA_2&3/CA_2/Q
3/dynamic_1000$ ./main
Serial timing for 100000 iterations

Time Elapsed Serial      33746 mSecs Total=32.617277 Check Sum = 100000

OpenMP Parallel Timings for 100000 iterations

Time Elapsed thread 0    47009 mSecs
Time Elapsed thread 1    46653 mSecs
Time Elapsed thread 2    46690 mSecs
Time Elapsed thread 3    46035 mSecs
Time Elapsed Parallel    31071 mSecs Total=32.617277 Check Sum = 100000

Speedup = 1.086089

```

در حالت (dynamic 1000) schedule، هنگامی که هر thread، اعلام آمادگی کند، اجرای 1000 تکرار از حلقه به آن thread، واگذار می شود. پس اگر آن 1000 تکرار از مقادیر کم i باشد (حجم کار آن کم باشد)، کار thread سریع تمام شده و مجدد اعلام آمادگی می کند. اما اگر آن 1000 تکرار از مقادیر بزرگ i باشد (حجم کار آن زیاد باشد)، کار thread، دیر تمام می شود و در نتیجه تعداد دفعات کم تری می تواند اعلام آمادگی کند.

در نتیجه هر thread یا تعداد زیادی کار سبک انجام می دهد یا تعداد کمی کار سنگین یا ترکیبی از این دو. پس حجم کاری که هر thread انجام می دهد، تقریباً در یک حد و اندازه می باشد. به همین دلیل مشاهده می کنید که زمان اجرا هر thread تقریباً در یک حدود است.

```

ariyataghizadeh@ariyataghizadeh-VirtualBox:~/Parallel_Programming/CA_2&3/CA_2/Q
3/dynamic_2000$ ./main
Serial timing for 100000 iterations

Time Elapsed Serial      36682 mSecs Total=32.617277 Check Sum = 100000

OpenMP Parallel Timings for 100000 iterations

Time Elapsed thread 0      45563 mSecs
Time Elapsed thread 1      47557 mSecs
Time Elapsed thread 2      45747 mSecs
Time Elapsed thread 3      47496 mSecs
Time Elapsed Parallel      31063 mSecs Total=32.617277 Check Sum = 100000

Speedup = 1.180883

```

```

ariyataghizadeh@ariyataghizadeh-VirtualBox:~/Parallel_Programming/CA_2&3/CA_2/Q
3/dynamic_2000$ ./main
Serial timing for 100000 iterations

Time Elapsed Serial      38526 mSecs Total=32.617277 Check Sum = 100000

OpenMP Parallel Timings for 100000 iterations

Time Elapsed thread 0      45677 mSecs
Time Elapsed thread 1      47545 mSecs
Time Elapsed thread 2      45681 mSecs
Time Elapsed thread 3      47635 mSecs
Time Elapsed Parallel      31100 mSecs Total=32.617277 Check Sum = 100000

Speedup = 1.238806

```

در حالت (dynamic 2000) schedule، هنگامی که هر thread، اعلام آمادگی کند، اجرای 2000 تکرار از حلقه به آن thread، واگذار می شود. پس اگر آن 2000 تکرار از مقادیر کم i باشد (حجم کار آن کم باشد)، کار thread سریع تمام شده و مجدد اعلام آمادگی می کند. اما اگر آن 2000 تکرار از مقادیر بزرگ i باشد (حجم کار آن زیاد باشد)، کار thread، دیر تمام می شود و در نتیجه تعداد دفعات کم تری می تواند اعلام آمادگی کند.

در نتیجه هر thread یا تعداد زیادی کار سبک انجام می دهد یا تعداد کمی کار سنگین یا ترکیبی از این دو. پس حجم کاری که هر thread انجام می دهد، تقریباً در یک حد و اندازه می باشد. به همین دلیل مشاهده می کنید که زمان اجرا هر thread تقریباً در یک حدود است.