بسم الله الرحمن الرحیم

گزارش پروژه چهارم درس برنامه سازی موازی

انجام دهندگان:

محمد معین شفی – 810196492

علیرضا زارع نژاد اشکذری – 810196474

پاییز 1399

سوال 1

در این سوال هدف پیدا کردن مقدار ماکسیمم و اندیس آن در آرایه ای به طول 2^{20} می باشد.

```
#define VECTOR_SIZE 1048576

__attribute__((aligned(16))) float random_array[VECTOR_SIZE];
```

متغیر VECTOR SIZE را برابر طول آرایه و random array را به صورت بالا به صورت global تعریف می کنیم.

```
float get_random()
{
    static std::default_random_engine e;
    static std::uniform_real_distribution<> dis(0, 10); // rage 0 - 1
    return dis(e);
}

void make_random_float_arr() {

    if(!random_array){
        printf("Memory allocation error!!\n");
    }

    for (long i = 0; i < VECTOR_SIZE; i++){
        random_array[i] = get_random();
    }
}
```

برای تولید اعداد رندوم در آرایه از تابع get_random به صورت بالا استفاده می کنیم.

```
int serial_max(float* array) {
    Ipp64u start, end;
    int serial_duration;
    start = ippGetCpuClocks();
    float max_value = array[0];
    long max_index = 0;
    for (long i = 0; i < VECTOR_SIZE; i++){
        if(array[i] >= max_value){
            max_value = array[i];
            max_index = i;
        }
    }
    end = ippGetCpuClocks();
    serial_duration = (Ipp32s)(end - start);

    printf("Find Max in Serial: Max value = %f,  Index = %ld, Run time = %d\n", max_value, max_index, serial_duration);
    return serial_duration;
}
```

برای محاسبه مقدار ماکسیمم در حالت سریال کافی است که ابتدا با استفاده از تابع ippGetCpuClocks() ابتدا زمان شروع حلقه رو مقداردهی کنیم. مقدار max_value را برابر مقدار عنصراول آرایه و اندیس آن را نیر برابر 0 میگذاریم. سپس یک حلقه به اندازه طول آرایه خواهیم داشت و در آن بررسی می کنیم که آیا عنصر بزرگتری دیده شده است یا خیر. در صورت یافتن عضو بزرگتر مقدار max_value و max_index را به روز می کنیم. در نهایت نیز زمان پایان را به دست آورده و duration را نیز از تفاضل دو مقدار start و end به دست می آوریم.

برای محاسبه مقدار پارالل از چند روش استفاده می کنیم.

روش اول پارالل: lock

```
61
62
63  int omp_lock_max(float* array)
64  {
65      Ipp64u start, end;
66      int omp_lock_duration;
67      omp_lock_t lock;
68      omp_init_lock(&lock);
69
70      start = ippGetCpuClocks();
71
72      float max_value = array[0];
73      long max_index = 0;
74
75      #pragma omp parallel for num_threads(4)
76      for (long i = 0; i < VECTOR_SIZE; i++){
77          if(array[i] > max_value) {
78              omp_set_lock(&lock);
79                  max_value = array[i];
80                  max_index = i;
81              omp_unset_lock(&lock);
82          }
83      }
84
85      omp_destroy_lock(&lock);
86      end = ippGetCpuClocks();
87      omp_lock_duration = (Ipp32s)(end - start);
88
89      printf("Find Max with omp lock: Max value = %f,  Index = %ld, Run time = %d\n", max_value, max_index, omp_lock_duration);
90      return omp_lock_duration;
```

برای محاسبه ی مقدار ماکسیم در این روش بدین صورت عمل می کنیم که ابتدا مقدار start رو که شروع عمل محاسبه می باشد رو مقدار دهی می کنیم. سپس با استفاده از omp در بدنه ی construct آن از تعداد 4 ترد استفاده می کنیم. حلقه فوق با توجه به مکانیزم omp بین 4 ترد تقسیم می شود. متغیر lock را از نوع omp_lock_t تعریف کردیم و ابتدا آن را با تابع omp_init_lock() مقدار دهی می کنیم. حال هنگام بروز کردن مقدار ماکسیمم و اندیس آن در صورت یافتن عضو بزرگتر در بدنه ی حلقه ابتدا با استفاده از تابع omp_set_lock() مطمئن می شویم که تردهای دیگر در ناحیه بحرانی نیستند و در آن به آپدیت کردن مقدار ماکسیمم و اندیس آن می پردازیم.

در انتهای بدنه پارالل نیز که همه ترد ها کارشان تمام می شود مقدار end را به دست می آوریم. و با تفاصل این دو مقدار duration برنامه رو به دست می آوریم.

روش دوم پارالل: critical

```
long opm_critical_max(float* array)
{
    Ipp64u start, end;
    int omp_critical_duration;
    start = ippGetCpuClocks();
    float max_value = array[0];
    long max_index = 0;
    float max_value_local;
    long max_index_local;
    #pragma omp parallel num_threads(2) shared(array, max_value, max_index) private(max_value_local, max_index_local)
    {
        max_value_local = FLT_MIN;
        max_index_local = -1;
        #pragma omp for nowait
        for (long i = 0; i < VECTOR_SIZE; i++){
            if (array[i] > max_value_local){
                max_value_local = array[i];
                max_index_local = i;
            }
        }
        #pragma omp critical
        {
            if(max_value_local > max_value){
                max_value = max_value_local;
                max_index = max_index_local;
            }
        }
    }
    end = ippGetCpuClocks();
    omp_critical_duration = (Ipp32s)(end - start);
    printf("Find Max with omp crttical: Max value = %f,  Index = %ld, Run time = %d\n", max_value, max_index, omp_critical_duration);
    return omp_critical_duration;
}
```

در اینجا نیز دوباره از دو متغیر از نوع Ipp64u از کتابخانه ipp.h و تابع ippGetCpuClocks() استفاده کردیم که شروع و پایان کار زمانش را داشته باشیم. متغیر max_value را برابر عضو اول آرایه و max_index را برابر اندیس 0 قرار می دهیم.

در اینجا هر کدام از ترد ها متغیر ماکسیمم لوکال خود را به دست می آورند. بدین منظور از omp parallel for استفاده می کنیم. همچنین از no wait استفاده کردیم که هر کدام از ترد ها که کارشان تمام شد سراغ بدنه ی critical بروند و صبر نکنند. در حلقه هر کدام از ترد ها با توجه به اینکه هر کدام بخشی از حلقه را پیمایش می کنند مقدار ماکسیمم لوکال را به دست می آورند. پس از آن در بدنه ی critical مقدار متغیر max_value که بین ترد ها share شده و همچنین max_index را آپدیت می کنیم. توجه کنیم که متغیر max_value_local و max_index_local که private تعریف شده است برای هر ترد متفاوت است و لوکال آن می باشد.

در انتها مقدار duration را محاسبه کردیم.

روش سوم پارالل: reduction

```
struct find_max {
    float val;
    long index;
};

#pragma omp declare reduction(maximum : struct find_max : omp_out = omp_in.val > omp_out.val ? omp_in : omp_out)

long omp_max_with_reduction_declaration(float* array)
{

    Ipp64u start, end;
    int omp_reduction_duration;

    start = ippGetCpuClocks();

    struct find_max max;
    max.val = FLT_MIN;
    max.index = -1;

    #pragma omp parallel for num_threads(4) reduction(maximum:max)
    for (long i = 0; i < VECTOR_SIZE; i++){
        if (array[i] > max.val){
            max.val = array[i];
            max.index = i;
        }
    }

    end = ippGetCpuClocks();
    omp_reduction_duration = (Ipp32s)(end - start);
    printf("Find Max in omp reduction: Max value = %f, Max Index: %ld, Run time = %d\n", max.val, max.index, omp_reduction_duration);
    return omp_reduction_duration;
}
```

در این بخش ابتدا یک rediction تعریف می کنیم که دو مقدار struct که داخل آن مقدار و اندیس عضو قرار دارد رو گرفته و ماکسیمم آن را به دست می آورد.

حال کافی است هنگام ساخته بدنه ی construct از دستور opm paralle for با تعداد ترد 4 و استفاده از ریداکشن عمل کنیم. همان طور که دیده می شود حلقه روی آرایه را خواهیم داشت و داخل آن مقدار struct یعنی عضو ماکسیمم و اندیس را به روز می کنیم. در انتهای بدنه ی پارالل max.value مقدار ماکسیمم و max.index اندیس عوض ماکسیمم خواهد بود.

در زیر نیز اسکرین شاتی از اجرای برنامه قرار داده شده است:

```
Find Max in Serial:     Max value = 9.999991,  Index = 671343, Run time = 6539601
Find Max with omp lock: Max value = 9.999991,  Index = 671343, Run time = 3621822
Find Max with omp crttical:     Max value = 9.999991,  Index = 671343, Run time = 3649282
Find Max in omp reduction without index: Max value = 9.999991, Run time = 4591782
Find Max in omp reduction: Max value = 9.999991, Max Index: 671343, Run time = 2449475
Speed up with omp lock: 1.805611
Speed up with omp critical: 1.792024
Speed up with omp reduction without index: 1.424197
Speed up with omp reduction declaration: 2.669797
root@shafi:/home/moein/moein/pp/ca4/zare/Parallel-programming/ca4# ./part1
**************************************************
group members:
               Alireza Zarenejad:      810196474
               Mohammad Moein Shafi:   810196492
**************************************************

Find Max in Serial:     Max value = 9.999991,  Index = 671343, Run time = 6458563
Find Max with omp lock: Max value = 9.999991,  Index = 671343, Run time = 3014707
Find Max with omp crttical:     Max value = 9.999991,  Index = 671343, Run time = 3650790
Find Max in omp reduction without index: Max value = 9.999991, Run time = 4190515
Find Max in omp reduction: Max value = 9.999991, Max Index: 671343, Run time = 2438679
Speed up with omp lock: 2.142352
Speed up with omp critical: 1.769086
Speed up with omp reduction without index: 1.541234
Speed up with omp reduction declaration: 2.648386
root@shafi:/home/moein/moein/pp/ca4/zare/Parallel-programming/ca4# ./part1
**************************************************
group members:
               Alireza Zarenejad:      810196474
               Mohammad Moein Shafi:   810196492
**************************************************

Find Max in Serial:     Max value = 9.999991,  Index = 671343, Run time = 6554000
Find Max with omp lock: Max value = 9.999991,  Index = 671343, Run time = 2986683
Find Max with omp crttical:     Max value = 9.999991,  Index = 671343, Run time = 3668239
Find Max in omp reduction without index: Max value = 9.999991, Run time = 3155782
Find Max in omp reduction: Max value = 9.999991, Max Index: 671343, Run time = 2507801
Speed up with omp lock: 2.194408
Speed up with omp critical: 1.786688
Speed up with omp reduction without index: 2.076823
Speed up with omp reduction declaration: 2.613445
root@shafi:/home/moein/moein/pp/ca4/zare/Parallel-programming/ca4#
```

سوال 2

در ابتدای برنامه سه vector به نام های numbers_s و numbers_pm و numbers_pq با اندازه ی 20^2 (دو به توان بیست) تعریف می شود، سپس خانه های این vector ها در تابع fill_with_random_numbers مقداردهی می شوند. از آرایه ی numbers_s برای استفاده در قسمت سریال و از آرایه ی number_p برای استفاده در قسمت موازی استفاده می شود. دقت شود که مقادیر اولیه هر دو آرایه یکسان هستند. همچنین یک vector به اسم temp نیز برای استفاده در بخش موازی و در قسمت merge کردن تعریف شده است که در قسمت موازی توضیحات مربوط به آن آورده شده است.

```
void fill_with_random_numbers(
                                vector<float>& numbers_s,
                                vector<float>& numbers_pm,
                                vector<float>& numbers_pq)
{
    for (int i = 0; i < VECTOR_SIZE; i++)
    {
        numbers_s[i] = get_random();
        numbers_pm[i] = numbers_s[i];
        numbers_pq[i] = numbers_s[i];
    }
}
```

در این تابع از تابع get_random استفاده می شود که یک عدد اعشاری تصادفی بین صفر تا ۱۰ را بر میگرداند.

```
float get_random()
{
    static std::default_random_engine e;
    static std::uniform_real_distribution<> dis(0, 10);
    return dis(e);
}
```

سپس در تابع serial_part مرتب سازی به صورت سریال بر روی numbers_s صورت می گیرد و در تابع parallel_part_mergesort نیز مرتب سازی به روش موازی و با استفاده از الگوریتم mergesort بر روی numbers_pm صورت میگیرد و و در تابع parallel_part_quicksort نیز مرتب سازی به روش موازی و با استفاده از الگوریتم quicksort بر روی numbers_pq صورت میگرد و زمان اجرای هر یک محاسبه می شود.

```
int main()
{
    vector<float> numbers_s(VECTOR_SIZE);
    vector<float> numbers_pm(VECTOR_SIZE);
    vector<float> numbers_pq(VECTOR_SIZE);
    vector<float> temp(VECTOR_SIZE);
    print_student_info();
    fill_with_random_numbers(numbers_s, numbers_pm, numbers_pq);

    int serial_time = serial_part(numbers_s);
    if(serial_time == -1)
        return 1;

    int parallel_time_mergesort = parallel_part_mergesort(numbers_pm, temp);
    if(parallel_time_mergesort == -1)
        return 1;

    int parallel_time_quicksort = parallel_part_quicksort(numbers_pq);
    if(parallel_time_quicksort == -1)
        return 1;
```

در انتها نیز در تابع check_equality به بررسی یکسان بودن پاسخ های گرفته شده در بخش های سریال و موازی پرداخته می شود.

```
check_equality(numbers_s, numbers_pm, numbers_pq);

cout << "\nSerial takes " << serial_time << " clock cycles"<< endl;
cout << "Parallel with Mergesort takes " << parallel_time_mergesort << " clock cycles"<< endl;
cout << "Parallel with Quciksort takes " << parallel_time_quicksort << " clock cycles"<< endl;
cout << "\nSpeed up with Parallel Mergesort: " << float(serial_time) / parallel_time_mergesort << endl;
cout << "Speed up with Parallel Quicksort: " << float(serial_time) / parallel_time_quicksort << endl;

exit(EXIT_SUCCESS);
```

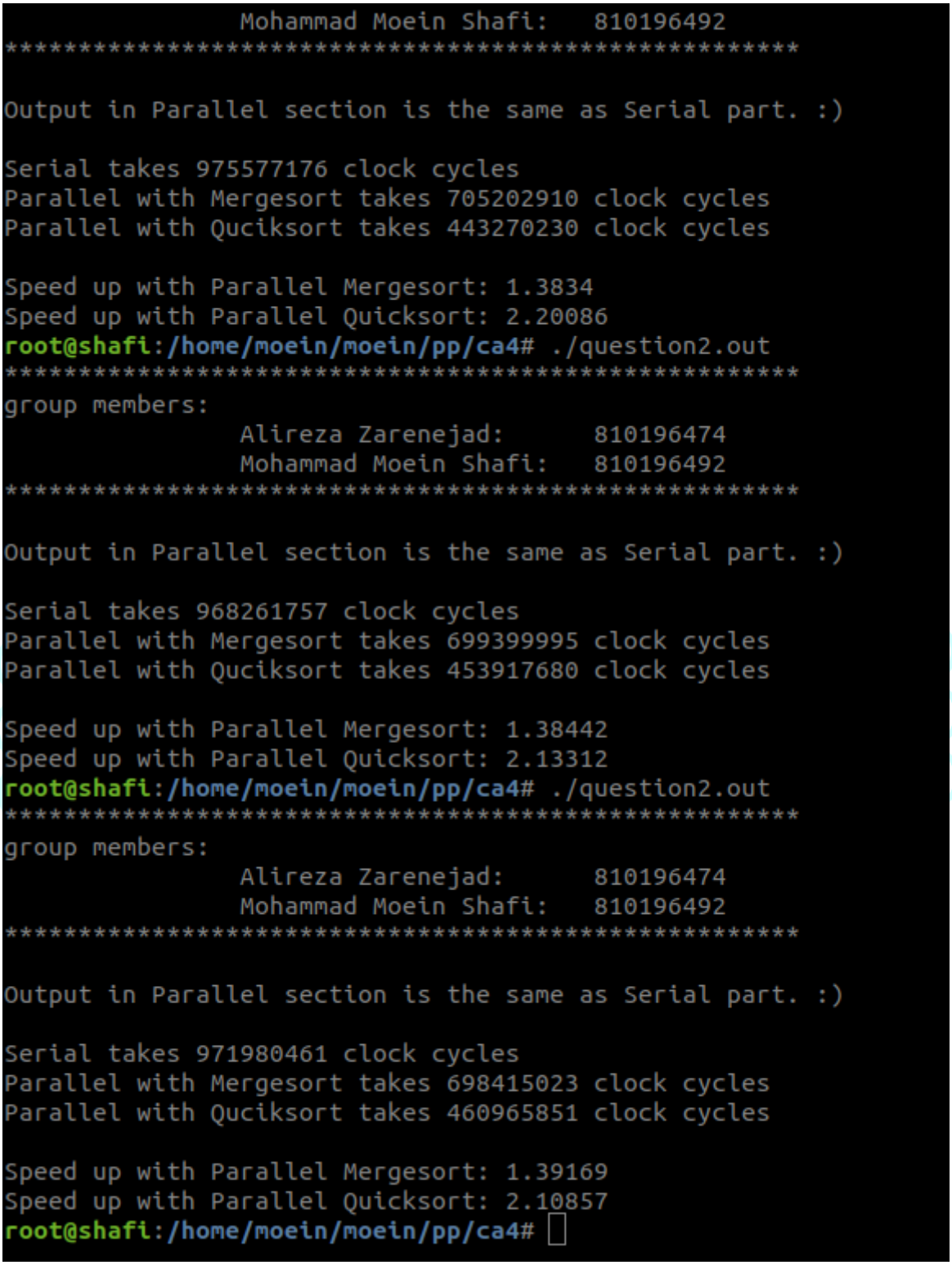
پیاده سازی تابع نام برده نیز به صورت زیر است:

```
void check_equality(vector<float>& numbers_s, vector<float>& numbers_pm, vector<float>& numbers_pq)
{
    for (int i = 0; i < VECTOR_SIZE; i++)
        if (numbers_s[i] != numbers_pm[i] || numbers_s[i] != numbers_pq[i])
        {
            cout << "s: " << numbers_s[i] << " , pm: " << numbers_pm[i] << " , pq: " << numbers_pq[i] << endl;
            cout << "Error! Output in Parallel section is not the same with the Serial part!\n";
            return;
        }
    cout << "Output in Parallel section is the same as Serial part. :)\n";
}
```

زمان اجرای هر بخش و speedup دریافت شده و همچنین میانگین آن طی اجراهای مختلف در جدول زیر مشخص شده است:

شماره اجرا	Speed up with parallel mergesort	Speed up with parallel quicksort
۱	1.3834	2.20086
۲	1.38442	2.13312
۳	1.39169	2.10857
میانگین	1.3865	2.1475

در زیر اسکرین شاتی از اجرای برنامه مربوط به سوال 2 آورده شده است:



در ادامه به بررسی هر یک از روش های استفاده شده می پردازیم:

- اجرای سریال

در این بخش به منظور داشتن بهترین عملکرد ، از الگوریتم quick sort استفاده شده است.

```
int serial_part(vector<float>& numbers_s)
{
    Ipp64u start, end;

    start = ippGetCpuClocks();
    quickSort(numbers_s, 0, VECTOR_SIZE - 1);
    end = ippGetCpuClocks();

    int serial_time = (Ipp32u)(end - start);
    return serial_time;
}
```

به طور خلاصه، این الگوریتم یک pivot انتخاب می کند و بر مبنای آن آرایه را به دو قسمت می شکند. یک قسمت برای اعداد کوچکتر از آن pivot و یک قسمت نیز برای اعداد بزرگتر از آن. سپس این کار را به صورت بازگشتی تکرار می کند تا سایز آرایه ی شکسته شده کمتر مساوی ۲ شود. (لازم به ذکر است در اینجا به منظور بهبود عملکرد استفاده از حافظه و صرفه جویی در زمان، در عمل این شکستن با استفاده از index ها صورت می گیرد و از آرایه های جداگانه استفاده نمی شود. در هربار که تابع quicksort به صورت بازگشتی صدا زده می شود، یک رنج خاصی از اعداد برای بررسی به آن داده می شود)

```
void quickSort(vector<float>& numbers_s, int low, int high)
{
    if (low < high)
    {
        int pi = partition(numbers_s, low, high);

        quickSort(numbers_s, low, pi - 1);
        quickSort(numbers_s, pi + 1, high);
    }
}
```

همانطور که مشخص است، مهمترین قسمت در این الگوریتم، چگونگی انتخاب pivot و شکستن آرایه به دو نیم است. به این منظور به صورت زیر عمل کردیم: (به علت واضح بودن آن از توضیح اضافی پرهیز شد)

```
int partition (vector<float>& numbers_s, int low, int high)
{
    float pivot = numbers_s[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (numbers_s[j] <= pivot)
        {
            i++;
            swap(numbers_s, i, j);
        }
    }
    swap(numbers_s, i + 1, high);
    return (i + 1);
}
```

تابع swap نیز به صورت زیر پیاده سازی شده است.

```
void swap(vector<float>& numbers_s, int first, int second)
{
    float temp = numbers_s[first];
    numbers_s[first] = numbers_s[second];
    numbers_s[second] = temp;
}
```

- اجرای موازی

در این بخش به منظور بررسی کاملتر و همچنین به علت اینکه در برخی منابع عنوان شده بود که الگوریتم mergesort در حالت موازی performance بهتری نسبت به الگوریتم quicksort در حالت موازی دارد، بنابراین تصمیم بر آن شد تا هر دو الگوریتم را به صورت موازی پیاده سازی کرده و عملکرد هر یک را مورد سنجش قرار دهیم.

- الگوریتم mergesort

ابتدا به بررسی الگوریتم mergesort می پردازیم. شروع این بخش از تابع parallel_part_mergesort می باشد. در ابتدای آن متغیر num_threads تعریف شده است که بیانگر تعداد مراحل مورد نظر برای ساخت task ها به صورت تودرتو است. طبق تست های مختلفی که صورت گرفت، عدد 40 بهترین performance را برای این

کار میداد. در ادامه با تعریف کردن متغیرهای start و end و استفاده ازتابع ippGetCpuClock ، زمان اجرای این بخش را اندازه می گیگیریم :) . بخش موازی با صدا زدن تابع parallel_mergesort آغاز می شود.

```
int parallel_part_mergesort(vector<float>& numbers_pm, vector<float>& temp)
{
    int num_threads = 40;
    Ipp64u start, end;

    start = ippGetCpuClocks();
    #pragma omp parallel default(shared)
        #pragma omp single
            parallel_mergesort(numbers_pm, temp, 0, VECTOR_SIZE - 1, num_threads);
    end = ippGetCpuClocks();

    int time = (Ipp32u)(end - start);
    return time;
}
```

همانطور که مستحضر هستید، الگوریتم mergesort را آرایه به دو نیم تقسیم کرده و عملیات مرتب سازی را بر روی هر قسمت جداگانه انجام داده و در نهایت دو قسمت موجود را با هم merge می کند و این کار به صورت بازگشتی تکرار می شود. شرط پایان این عملیات بازگشتی این است که اندازه آرایه ی شکسته شده کمتر مساوی با 2 شود. سپس بین آن خانه موجود مقایسه صورت گرفته و در صورت نیاز از تابع swap استفاده می شود (این تابع در بخش قبلی نشان داده شد). برای پیاده سازی موازی این الگوریتم نیز ، همانطور که در شکل نشان داده شده است، از task ها استفاده شده است. در هر مرحله تعداد threads برای مرحله بعد برابر با نصف مقدار threads موجود است. چون ممکن است کار یک بخش زودتر تمام شده و عملیات merge قبل از کامل شدن بخش دیگر صورت گیرد، از taskwait قبل از عملیات merge استفاده شده است. در صورتیکه threads برابر با 1 بود، ادامه آن task به صورت سریال انجام خواهد گرفت.

```
void parallel_mergesort(vector<float>& numbers_pm, vector<float>& temp,
                        int start, int end, int threads)
{
    if (start >= end)
        return;

    int m = 0;
    if (threads == 1 || end - start < 2)
        mergesort_serial(numbers_pm, temp, start, end);

    else if (threads > 1)
    {
        m = (end + start - 1) / 2;
        #pragma omp task default(shared)
            parallel_mergesort(numbers_pm, temp, start, m, threads / 2);

        #pragma omp task default(shared)
            parallel_mergesort(numbers_pm, temp, m + 1, end, threads / 2);

        #pragma omp taskwait
            merge(numbers_pm, temp, start, end);
    }
}
```

قسمت سریال برای این الگوریتم نیز، همانطور که در توضیحات آورده شده، به صورت زیر پیاده سازی شده است:

```
void mergesort_serial(vector<float>& numbers_pm,
                      vector<float>& temp, int start,
                      int end)
{
    if (end - start < 2)
    {
        if (numbers_pm[end] >= numbers_pm[start])
            return;
        else
        {
            swap(numbers_pm[start], numbers_pm[end]);
            return;
        }
    }

    int m = (end + start - 1) / 2;
    mergesort_serial(numbers_pm, temp, start, m);
    mergesort_serial(numbers_pm, temp, m + 1, end);
    merge(numbers_pm, temp, start, end);
}
```

- الگوریتم quicksort

اکنون به بررسی الگوریتم quicksort می پردازیم. این بخش نیز بسیار مشابه با الگوریتم mergesort پیاده سازی شده است و تنها در بخش های اندکی تفاوت وجود دارد. در این بخش نیز همانند بخش قبل، تا مرحله ای task ها را به صورت تودرتو ایجاد کرده و از جایی به بعد هر task به صورت سریال به کار خودش ادامه می دهد. شروع این بخش از تابع parallel_part_quicksort می باشد. در ابتدای آن متغیر num_threads تعریف شده است که بیانگر تعداد مراحل مورد نظر برای ساخت task ها به صورت تودرتو است. طبق تست های مختلفی که صورت گرفت، عدد 50 بهترین performance را برای این کار میداد. بخش موازی با صدا زدن تابع parallel_quicksort آغاز می شود.

```
int parallel_part_quicksort(vector<float>& numbers_pq)
{
    int num_threads = 50;
    Ipp64u start, end;

    start = ippGetCpuClocks();
    #pragma omp parallel default(shared)
        #pragma omp single
            parallel_quicksort(numbers_pq, 0, VECTOR_SIZE - 1, num_threads);
    end = ippGetCpuClocks();

    int time = (Ipp32u)(end - start);
    return time;
}
```

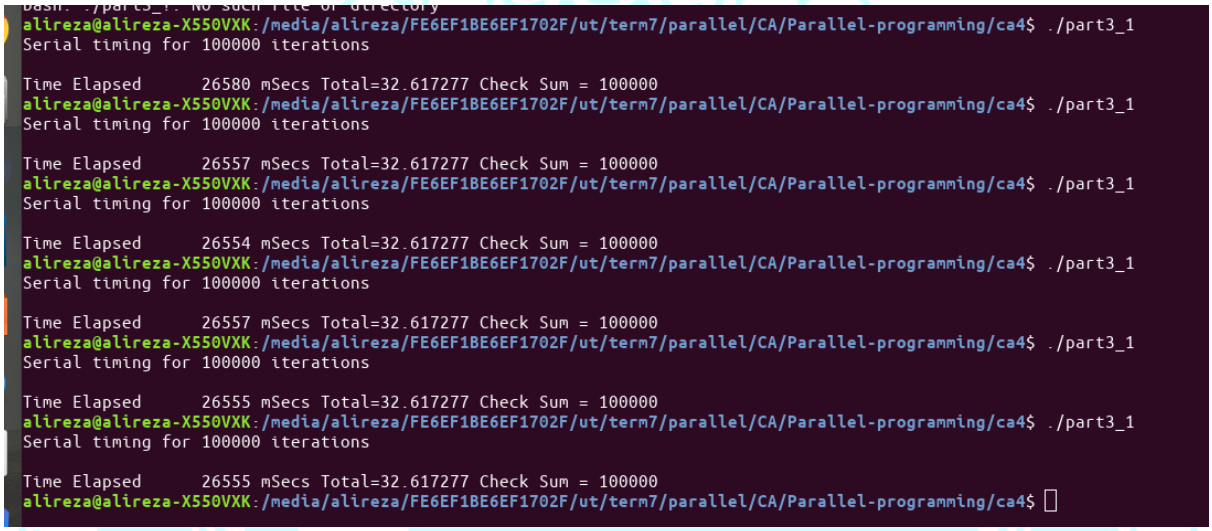
تابع parallel_quicksort نیز مشابه با آنچه در الگوریتم mergesort اتفاق می افتاد است. در اینجا نیز زمانی که مقدار threads برابر با 1 باشد ، ادامه ی task را به صورت سریال پیش میبریم. برای بخش سریال از همان کد موجود در بخش سریال استفاده شده است. در هر مرحله مقدار متغیر threads نصف شده و به مرحله بعد ارسال می شود. بخش مهم این قسمت همانند بخش سریال ، روش انتخاب pivot است که در اینجا نیز همانند بخش سریال عمل شد و از همان تابع استفاده شده است. به علت استفاده از task عملکرد الگوریتم quicksort به نسبت خوب بوده است.

```
int parallel_quicksort(vector<float>& numbers_pq, int low, int high, int threads)
{
    if (threads == 1)
        quickSort(numbers_pq, low, high);

    else if (low < high)
    {
        int pi = partition(numbers_pq, low, high);
        #pragma omp task default(shared)
            parallel_quicksort(numbers_pq, low, pi - 1, threads / 2);

        #pragma omp task default(shared)
            parallel_quicksort(numbers_pq, pi + 1, high, threads / 2);
    }
}
```

سوال 3

اجرای سریال:

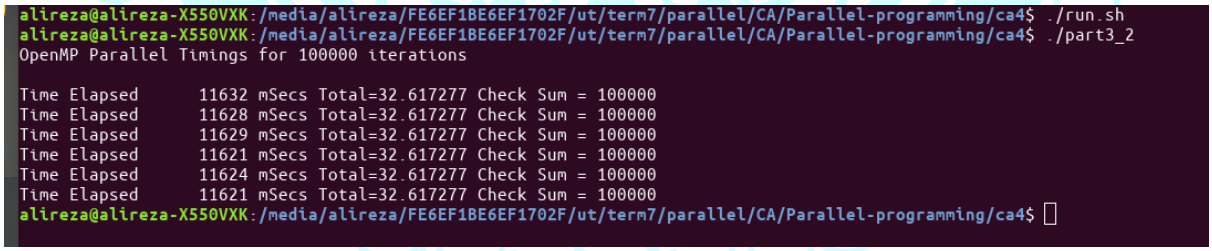
در ابتدا زمان اجرای برنامه سریال را به دست می آوریم. در زیر 6 اجرای برنامه آورده شده است.



همان طور که دیده می شود میانگین زمان اجرا طی این 6 اجرا برابر مقدار زیر است.

$$\frac{159358}{6} = 26559.6667\ msec = \ 26.5596\ sec$$

اجرای پارالل در مود static



میانگین اجرا در این حالت برابر است با:

$$\frac{69755}{6} = \ 11625.8333\ msec = 11.6258$$

و میزان تسریع در این حالت برابر است با:

$$speedup = \frac{26559.6667}{11625.8333} = \ 2.2845$$

اجرای پارالل در مود dynamic 1000

```
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$ ./run.sh
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$ ./part3_2
OpenMP Parallel Timings for 100000 iterations

Time Elapsed       6976 mSecs Total=32.617277 Check Sum = 100000
Time Elapsed       7281 mSecs Total=32.617277 Check Sum = 100000
Time Elapsed       6879 mSecs Total=32.617277 Check Sum = 100000
Time Elapsed       6887 mSecs Total=32.617277 Check Sum = 100000
Time Elapsed       6947 mSecs Total=32.617277 Check Sum = 100000
Time Elapsed       6935 mSecs Total=32.617277 Check Sum = 100000
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$
```

میانگین اجرا در این حالت برابر است با:

$$\frac{41905}{6} = \ 6984.16667\ msec$$

و میزان تسریع در این حالت برابر است با:

$$speedup = \frac{26559.6667}{6984.16667} = \ 3.8028$$

اجرای پارالل در مود dynamic 2000

```
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$ ./run.sh
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$ ./part3_2
OpenMP Parallel Timings for 100000 iterations

Time Elapsed       7127 mSecs Total=32.617277 Check Sum = 100000
Time Elapsed       7328 mSecs Total=32.617277 Check Sum = 100000
Time Elapsed       7051 mSecs Total=32.617277 Check Sum = 100000
Time Elapsed       7262 mSecs Total=32.617277 Check Sum = 100000
Time Elapsed       7054 mSecs Total=32.617277 Check Sum = 100000
Time Elapsed       7298 mSecs Total=32.617277 Check Sum = 100000
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$
```

میانگین اجرا در این حالت برابر است با:

$$\frac{43120}{6} = \ 7186.66667\ msec$$

و میزان تسریع در این حالت برابر است با:

$$speedup = \frac{26559.6667}{7186.66667} = \ 3.69568646$$

سپس کد را به گونه ای تغییر دادیم که میزان زمان هر ترد را بتوانیم به دست بیاوریم. عکس آن در زیر آمده است. همچنین کد تغییر یافته نیز در فایل question3_1_edit.cpp آمده است.

```
#pragma omp parallel num_threads (4) private( sumx, sumy, k )
{

  thread_starttime = timeGetTime();

#pragma omp for reduction( +: sum, total ) schedule(dynamic,2000 ) nowait

  for( int j=0; j<VERYBIG; j++ )
  {
    // increment check sum
    sum += 1;

    // Calculate first arithmetic series
    sumx = 0.0;
    for( k=0; k<j; k++ )
     sumx = sumx + (double)k;

    // Calculate second arithmetic series
    sumy = 0.0;
    for( k=j; k>0; k-- )
     sumy = sumy + (double)k;

    if( sumx > 0.0 )total = total + 1.0 / sqrt( sumx );
    if( sumy > 0.0 )total = total + 1.0 / sqrt( sumy );
  }

  thread_time = timeGetTime() - thread_starttime;
  printf("Thread %d Time Elapsed: %10d mSecs\n",omp_get_thread_num(), (int)(thread_time * 1000));

  }

  // get ending time and use it to determine elapsed time
  elapsedtime = timeGetTime() - starttime;

  // report elapsed time
  printf("Time Elapsed %10d mSecs Total=%lf Check Sum = %ld\n********************\n",
                 (int)(elapsedtime * 1000), total, sum );
```

اجرای کد پارالل در حالت static کد تغییر یافته:

```
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$ ./run.sh
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$ ./part3_2_edit
OpenMP Parallel Timings for 100000 iterations

Thread 0 Time Elapsed:       1660 mSecs
Thread 1 Time Elapsed:       4987 mSecs
Thread 2 Time Elapsed:       8304 mSecs
Thread 3 Time Elapsed:      11621 mSecs
Time Elapsed      11621 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 0 Time Elapsed:       1661 mSecs
Thread 1 Time Elapsed:       4983 mSecs
Thread 2 Time Elapsed:       8298 mSecs
Thread 3 Time Elapsed:      11617 mSecs
Time Elapsed      11617 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 0 Time Elapsed:       1670 mSecs
Thread 1 Time Elapsed:       5008 mSecs
Thread 2 Time Elapsed:       8298 mSecs
Thread 3 Time Elapsed:      11629 mSecs
Time Elapsed      11630 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 0 Time Elapsed:       1677 mSecs
Thread 1 Time Elapsed:       4994 mSecs
Thread 2 Time Elapsed:       8297 mSecs
Thread 3 Time Elapsed:      11618 mSecs
Time Elapsed      11618 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 0 Time Elapsed:       1664 mSecs
Thread 1 Time Elapsed:       4983 mSecs
Thread 2 Time Elapsed:       8298 mSecs
Thread 3 Time Elapsed:      11629 mSecs
Time Elapsed      11629 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 0 Time Elapsed:       1666 mSecs
Thread 1 Time Elapsed:       5101 mSecs
Thread 2 Time Elapsed:       8342 mSecs
Thread 3 Time Elapsed:      11652 mSecs
Time Elapsed      11652 mSecs Total=32.617277 Check Sum = 100000
********************
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$
```

اجرای کد پارالل در حالت dynamic 1000 کد تغییر یافته:

```
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$ ./part3_2_edit
OpenMP Parallel Timings for 100000 iterations

Thread 0 Time Elapsed:       6450 mSecs
Thread 1 Time Elapsed:       6601 mSecs
Thread 2 Time Elapsed:       6759 mSecs
Thread 3 Time Elapsed:       6866 mSecs
Time Elapsed       6866 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 3 Time Elapsed:       6453 mSecs
Thread 1 Time Elapsed:       6601 mSecs
Thread 2 Time Elapsed:       6742 mSecs
Thread 0 Time Elapsed:       6848 mSecs
Time Elapsed       6848 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 0 Time Elapsed:       6445 mSecs
Thread 1 Time Elapsed:       6591 mSecs
Thread 3 Time Elapsed:       6750 mSecs
Thread 2 Time Elapsed:       6871 mSecs
Time Elapsed       6871 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 2 Time Elapsed:       6494 mSecs
Thread 1 Time Elapsed:       6605 mSecs
Thread 3 Time Elapsed:       6718 mSecs
Thread 0 Time Elapsed:       6889 mSecs
Time Elapsed       6889 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 0 Time Elapsed:       6450 mSecs
Thread 2 Time Elapsed:       6612 mSecs
Thread 3 Time Elapsed:       6711 mSecs
Thread 1 Time Elapsed:       6860 mSecs
Time Elapsed       6860 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 0 Time Elapsed:       6449 mSecs
Thread 2 Time Elapsed:       6597 mSecs
Thread 3 Time Elapsed:       6725 mSecs
Thread 1 Time Elapsed:       6876 mSecs
Time Elapsed       6876 mSecs Total=32.617277 Check Sum = 100000
********************
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$
```

اجرای کد پارالل در حالت dynamic 2000 کد تغییر یافته:

```
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$ ./run.sh
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$ ./part3_2_edit
OpenMP Parallel Timings for 100000 iterations

Thread 1 Time Elapsed:       6720 mSecs
Thread 3 Time Elapsed:       7065 mSecs
Thread 0 Time Elapsed:       7066 mSecs
Thread 2 Time Elapsed:       7731 mSecs
Time Elapsed       7731 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 1 Time Elapsed:       6338 mSecs
Thread 3 Time Elapsed:       6864 mSecs
Thread 0 Time Elapsed:       7009 mSecs
Thread 2 Time Elapsed:       7245 mSecs
Time Elapsed       7249 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 0 Time Elapsed:       6421 mSecs
Thread 3 Time Elapsed:       6668 mSecs
Thread 2 Time Elapsed:       6955 mSecs
Thread 1 Time Elapsed:       7354 mSecs
Time Elapsed       7354 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 2 Time Elapsed:       6475 mSecs
Thread 3 Time Elapsed:       6678 mSecs
Thread 0 Time Elapsed:       6918 mSecs
Thread 1 Time Elapsed:       7292 mSecs
Time Elapsed       7292 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 2 Time Elapsed:       6333 mSecs
Thread 3 Time Elapsed:       6566 mSecs
Thread 0 Time Elapsed:       6874 mSecs
Thread 1 Time Elapsed:       7071 mSecs
Time Elapsed       7071 mSecs Total=32.617277 Check Sum = 100000
********************
Thread 0 Time Elapsed:       6668 mSecs
Thread 1 Time Elapsed:       6837 mSecs
Thread 3 Time Elapsed:       7095 mSecs
Thread 2 Time Elapsed:       7434 mSecs
Time Elapsed       7434 mSecs Total=32.617277 Check Sum = 100000
********************
alireza@alireza-X550VXK:/media/alireza/FE6EF1BE6EF1702F/ut/term7/parallel/CA/Parallel-programming/ca4$
```

توضیح و نتیجه گیری سوال:

به وضوح دیده می شود که در حالت static به دلیل اینکه حلقه ی داخلی تا j می رود، ترد شماره 1 کمترین مقدار duration و ترد 4 بیشترین مقدار duration را دارا می باشد. اما در حالت های dynamic چون به صورت پویا بین ترد ها تقسیم صورت می گیرد شاهد load balance بیشتری هستیم و در نتیجه مدت زمان duration ترد ها تقریبا با یکدیگر برابر است.