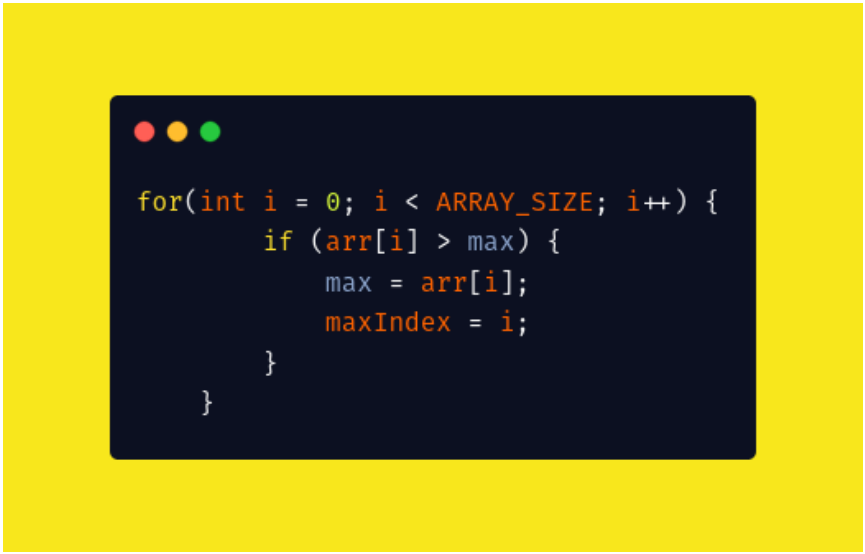
آرین حدادی ۸۱۰۱۹۶۴۴۸
ایمان مرادی ۸۱۰۱۹۶۵۶۰
پروژه چهارم درس برنامه نویسی موازی

محتویات فایل آپلود شده)

فایل آپلود شده شامل یک فایل main.cpp است که کد سوال اول و دوم بوده و makefile نیز برای make کردن همین فایل است.همچنین دو فایل مرتبط با سوال سوم نیز در فایل آپلود شده به همان نام های اولیه خود وجود دارند.فایل آخر هم report است که گزارش کار این تمرین است.

سوال اول)

برای زدن این سوال ابتدا حالت سریال آن را پیاده سازی میکنیم که منطق ساده ای دارد.از ابتدا خوانده و ماکسیمم گیری میکنیم.هرجا که عدد فعلی آرایه از مقدار ماکسیمم تا به اینجا بیشتر باشد ماکسیمم و اندیس ماکسیمم را با عدد مناسب آپدیت میکنیم.



```
for(int i = 0; i < ARRAY_SIZE; i++) {
        if (arr[i] > max) {
            max = arr[i];
            maxIndex = i;
        }
    }
```

برای بخش parallel هم به این صورت عمل میکنیم که از یک for استفاده میکنیم تا ماکسیمم گیری توسط هریک از thread ها روی بخشی از آرایه جداگانه انجام شود و سپس بین ماکسیمم های گرفته شده نیز ماکسیمم گرفته می شود تا مقدار ماکسیمم کل آرایه حاصل شود.
توجه شود که تعداد thread ها در این سوال ۸ تا انتخاب شد.درباره دلیل انتخاب این عدد هم می توان گفت از آنجایی که CPU استفاده شده ۱۲ هسته logical و ۶ هسته physical دارد اگر تعداد thread ها مقدار دیفالت

باشد overhead ارتباط نهایی این دو و ایجاد آن ها زیاد شده و کارایی کاهش می یابد و هرچه تعداد thread ها بیشتر باشد کار آن ها کمتر شده و سریعتر به نتیجه می رسند و لذا برای رسیدن به یک tradeoff بین این دو مسئله عدد ۸ برای تعداد thread ها انتخاب شد که در عمل هم speedup بدست آمده بهتر از مقدار دیفالت یعنی با ۱۲ thread بود.

از آنجا که ماکسیمم نهایی یک مقدار مشترک بین thread ها محسوب می شود لازم است که دسترسی به آن به نحوی محدود شود که بخش آپدیت کردن ماکسیمم کلی تنها در هر لحظه توسط یک thread انجام شود زیرا در غیر این صورت برای مثال اگر مقدار ماکسیمم ۲۰ باشد و دو thread که هرکدام مقدار را به ترتیب ۲۵ و ۳۰ محاسبه کرده اند بخواهند مقدار ماکسیمم کلی را آپدیت کنند ممکن است هر دو thread یک مقدار را بخوانند که ۲۰ است و ابتدا thread با ماکسیمم ۳۰ آپدیت را انجام دهد و چون این آپدیت کردن بعد از خواندن مقدار ماکسیمم کلی توسط thread با ماکسیمم ۲۵ انجام شده است thread با ماکسیمم ۲۵ سپس مقدار خود را در متغیر ماکسیمم کلی می نویسند به این شکل به جای ۳۰ مقدار ۲۵ را به عنوان ماکسیمم کلی بعد از عملیات این دو thread خواهیم داشت که اشتباه است.لذا کد بخش parallel به این شکل می شود:

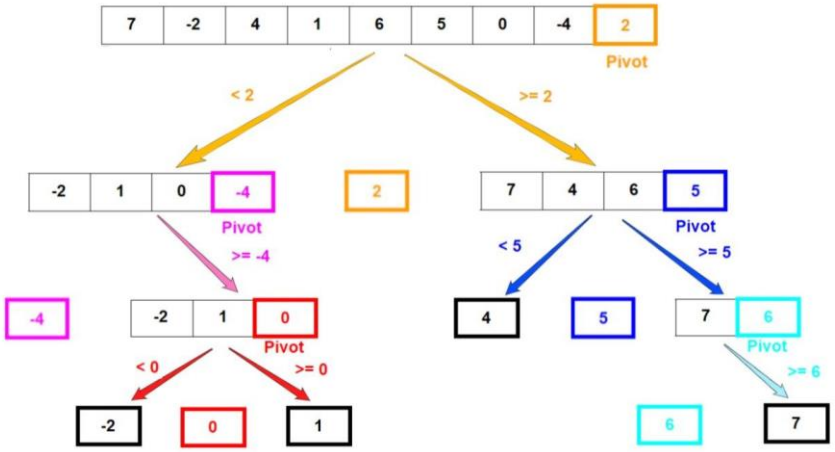
```
omp_lock_t lock;
omp_init_lock(&lock);

int i;
float maxThread;
int maxIndexThread;
#pragma omp parallel shared(arr, start, end, maxTotal, maxIndexTotal, lock) private(i, maxThread,
                            maxIndexThread) num_threads(NUM_OF_THREADS)
{

    maxThread = -1;
    #pragma omp for nowait
        for(i = 0; i < ARRAY_SIZE; i++) {
            if (arr[i] > maxThread) {
                maxThread = arr[i];
                maxIndexThread = i;
            }
        }

    omp_set_lock(&lock);
    if (maxThread > maxTotal) {
        maxTotal = maxThread;
        maxIndexTotal = maxIndexThread;
    }
    omp_unset_lock(&lock);

}
omp_destroy_lock(&lock);
```

توجه شود همانطور که مشاهده می شود برای بخش for از nowait clause استفاده شده است.دلیل آن این است که عملیات ماکسیمم گیری هر thread جداگانه جداگانه روی داده های متفاوت انجام شده و همچنین آپدیت کردن ماکسیمم نیز نیازی به اتمام کار سایر thread ها ندارد و لذا میتوان barrier ضمنی انتهای این for را برداشت تا بیهوده overhead آن را در اجرای برنامه نداشته باشیم.
خروجی اجرای این سوال با سوال دوم در انتهای سوال دوم آمده است.

سوال دوم)

ابتدا کمی در مورد الگوریتم quick sort توضیح می دهیم، سپس درباره ی موازی کردن آن بحث خواهیم کرد و در انتها پیاده سازی را به همراه جزئیاتش خواهیم دید.

الگوریتم quick sort یکی از الگوریتم های مبتنی بر مقایسه است که در بین الگوریتم های مرتب سازی از محبوبیت بالایی برخوردار است. این الگوریتم یک الگوریتم divide & conquer است که قسمت divide آن ارزش منطقی دارد و قسمت conquer آن دو بار اجرای همان الگوریتم است. اگر بخواهیم وارد جزئیات بیشتری شویم می توان گفت که در این الگوریتم ابتدا یک محور(pivot) برای آرایه انتخاب میکنیم که به این عمل partition میگوییم. در این مرحله اعداد کوچکتر از pivot در سمت چپ محور و اعداد بزرگتر از آن در سمت راست آن قرار میگیرند. سپس به صورت recursive با کال کردن تابع، یک بار برای قسمت سمت چپ و بار دیگر سمت راست میتوان همه ی آرایه را مرتب کرد. (در شکل زیر نمونه ای از quick sort را آورده ایم)



```
int partition(float* numbers, int startIndex, int endIndex) {
    int pivotIndex = startIndex;
    float pivotNumber = numbers[pivotIndex];
    for (int i = startIndex + 1; i ⩽ endIndex; i++) {
        if (numbers[i] ⩽ pivotNumber) {
            pivotIndex++;
            swap(numbers, i, pivotIndex);
        }
    }
    swap(numbers, pivotIndex, startIndex);
    return pivotIndex;
}

void serialQuickSort(float* numbers, int startIndex, int endIndex) {
    if (startIndex ⩾ endIndex)
        return;
    int pivot = partition(numbers, startIndex, endIndex);
    serialQuickSort(numbers, startIndex, pivot - 1);
    serialQuickSort(numbers, pivot + 1, endIndex);
}
```

با توجه به توضیحات داده شده حال میخواهیم قسمت هایی که میتوان آنها را به صورت موازی انجام دهیم را بیابیم. بنابراین از partition شروع میکنیم و اینکه آیا قابل موازی سازی است یا خیر. میدانیم که برای partition باید اعداد کوچکتر از pivot را به سمت چپ آن در آرایه و اعداد بزرگتر را به سمت راست آن عدد در آرایه انتقال داد. اگر بخواهیم بیش از یک ترد این کار را برای ما انجام دهد حتما در انتها به یک merge نیاز خواهیم داشت. در واقع این تابع یک تابع خوش تعریف برای موازی سازی نیست. اما در نقطه ی مقابل هر یک از دو قسمت دیگر را میتوان به آسانی و مستقل از هم اجرا کرد. چرا که از این پس این دو قسمت هیچ داده ی مشترکی نخواهند داشت. و البته یکسان بودن عملیات بر روی هر دو قسمت به خوش تعریف بودن آن کمک بیشتری میکند.

حال به کمک توضیحات ارایه شده quick sort را به صورت موازی و سریال پیاده سازی می کنیم. همان طور که گفته شد پیاده سازی موازی تنها در قسمت دوم یعنی موازی اجرا کردن قسمت های سمت راست و چپ به کار گرفته میشود. بنابراین تابع partition برای هر دو اجرا یکسان تعریف شده. اما برای اجرای موازی از task های OpenMP استفاده شده است. اولین استپ از parallelQuickSort را که برای کل آرایه call میشود را با هشت thread و به صورت موازی و سینگل ترد کال میکنیم. سینگل ترد میگوید که این تابع می بایست با یک thread اجرا شود و در صورت فراموشی این تکه از کد چند thread به صورت همزمان کل آرایه را مرتب میکنند که کاری نه تنها بیهوده بلکه از لحاظ performance ای بسیار بد خواهد بود. در واقع به صورت کلی هدف از single اجرا کردن اولین استپ این است که این تابع تنها یک بار اجرا شود.

```
#pragma omp parallel num_threads(NUM_OF_THREADS)
    {
        #pragma omp single
        {
            parallelQuickSort(arr, 0, ARRAY_SIZE - 1, ARRAY_SIZE);
        }
    }
```

حال در تابع parallelQuickSort پس از بررسی شرط خاتمه ی تابع و عمل partition مجددا هر کدام از قسمت ها را به همین صورت مرتب میکنیم. اما این بار هر کدام را به یک task میدهیم. Task ها به این صورت عمل میکنند که هر thread که خالی شد میتواند یکی از آنها را در اختیار بگیرد. بنابراین می توان قسمت ریکرسیو تابع را به صورت موازی انجام داد.

```
void parallelQuickSort(float* numbers, int startIndex, int endIndex, int size)
{   if (startIndex ≥ endIndex)
        return;
    int pivot = partition(numbers, startIndex, endIndex);

    #pragma omp task
    {
        parallelQuickSort(numbers, startIndex, pivot - 1, size);
    }
    #pragma omp task
    {
        parallelQuickSort(numbers, pivot + 1, endIndex, size);
    }
}
```

پس از اجرای الگوریتم برای هر دو حالت serial و parallel، نتیجه نهایی را verify میکنیم که اگر دو آرایه خروجی برابر باشند و مرتب هم باشند این verification موفقیت آمیز می شود.

در مورد speed up می بایست گفته شود که در الگوریتم quick sort چگونگی partition می تواند تاثیر مهمی بر روی سرعت الگوریتم بگذارد. پس بسیار مهم است که در هر دو حالت موازی و سریال به یک صورتی partition را تعریف کنیم که ذر هر دو آرایه به دو قسمت یکسان شکسته شود، بنابراین مثلا از انتخاب pivot به صورت random می بایست اجتناب کرد تا speed up مورد نظر دقیقا یکسان باشد.
برای این سوال نیز تعداد thread ها را ۸ در نظر گرفتیم.

نتیجه اجرا)

```
aryan@aryan-ROG-Strix-G531GW-G531GW:~/Desktop/parallel/Project 4$ ./main
Group Members:
1 - Aryan Haddadi 810196448
2 - Iman Moradi 810196560
-------------
Question 1

Maximum Element in Serial Calculation is 1.04857e+06
Maximum Element Index in Serial Calculation is 94399
Serial Calculation Time is 10799908 Clock Cycles.
Maximum Element in Parallel Calculation is 1.04857e+06
Maximum Element Index in Parallel Calculation is 94399
Parallel Calculation Time is 2282572 Clock Cycles.
Speedup is 4.73146
-------------
Question 2

Sorting Results Verification was Successful.
Serial Calculation Time is 431033266 Clock Cycles.
Parallel Calculation Time is 129808416 Clock Cycles.
Speedup is 3.32053
```

همانطور که مشاهده می شود خروجی های سوال اول در حالت موازی و سریال یکسان بوده و در سوال دوم هم نتیجه مرتب سازی چک شده است و نتیجه موفقیت آمیز بوده و جواب ها صحیح بوده اند.

توجه شود میزان speedup در سوال اول غالبا بین ۳ تا ۴ است و نتیجه بالا نتیجه یکی از اجرا هاست که اولین اجرا برای گرفتن عکس از خروجی برای گزارش کار به این شکل شد.

سوال سوم)

برای این سوال همانطور که در صورت تمرین بیان شده است ابتدا برای حالت سریال که فایل اول است زمان را محاسبه کردیم.

```
aryan@aryan-ROG-Strix-G531GW-G531GW:~/Desktop/parallel/Project 4/Q3$ ./a.out
Serial timing for 100000 iterations

Time Elapsed       21862 mSecs Total=32.617277 Check Sum = 100000
```

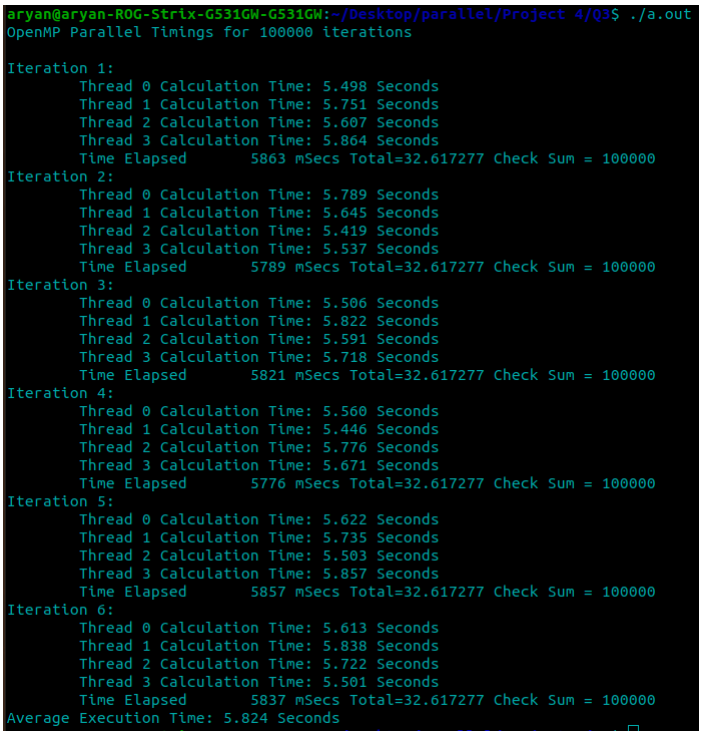
سپس به سراغ فایل دوم می رویم.چون حلقه بیرونی 6 بار اجرا می شود و از ما خواسته شده است که بین این 6 بار اجرا میانگین بگیریم.

همچنین از یک آرایه برای ذخیره سازی زمان اجرای هر thread در هربار اجرا استفاده میکنیم تا بتوانیم جداگانه زمان را برای هریک از آن ها نمایش دهیم.

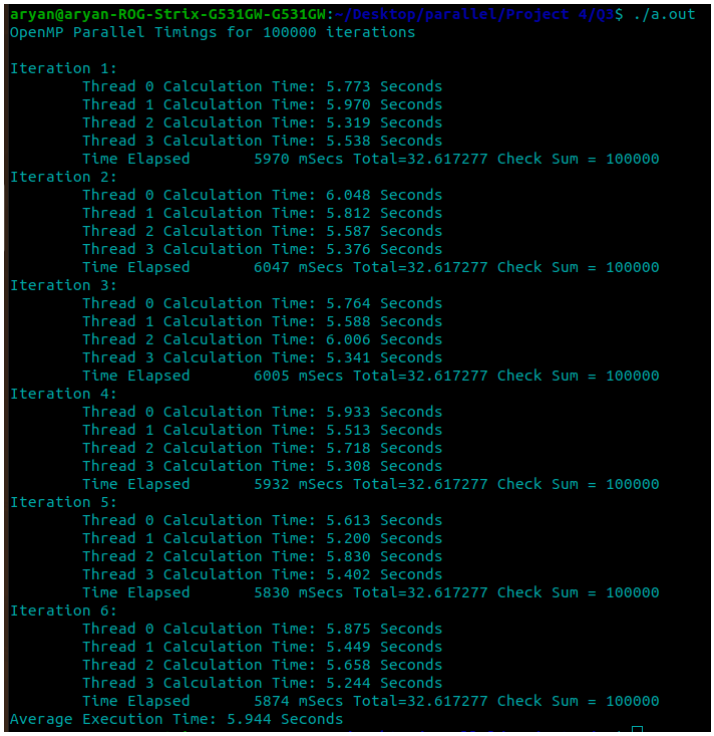
ابتدا حالت static را ران میکنیم.

```
aryan@aryan-ROG-Strix-G531GW-G531GW:~/Desktop/parallel/Project 4/Q3$ ./a.out
OpenMP Parallel Timings for 100000 iterations

Iteration 1:
        Thread 0 Calculation Time: 1.392 Seconds
        Thread 1 Calculation Time: 4.181 Seconds
        Thread 2 Calculation Time: 7.003 Seconds
        Thread 3 Calculation Time: 9.732 Seconds
        Time Elapsed       9731 mSecs Total=32.617277 Check Sum = 100000
Iteration 2:
        Thread 0 Calculation Time: 1.422 Seconds
        Thread 1 Calculation Time: 4.298 Seconds
        Thread 2 Calculation Time: 7.174 Seconds
        Thread 3 Calculation Time: 9.867 Seconds
        Time Elapsed       9866 mSecs Total=32.617277 Check Sum = 100000
Iteration 3:
        Thread 0 Calculation Time: 1.385 Seconds
        Thread 1 Calculation Time: 4.122 Seconds
        Thread 2 Calculation Time: 6.807 Seconds
        Thread 3 Calculation Time: 9.451 Seconds
        Time Elapsed       9451 mSecs Total=32.617277 Check Sum = 100000
Iteration 4:
        Thread 0 Calculation Time: 1.389 Seconds
        Thread 1 Calculation Time: 4.142 Seconds
        Thread 2 Calculation Time: 6.841 Seconds
        Thread 3 Calculation Time: 9.482 Seconds
        Time Elapsed       9481 mSecs Total=32.617277 Check Sum = 100000
Iteration 5:
        Thread 0 Calculation Time: 1.387 Seconds
        Thread 1 Calculation Time: 4.130 Seconds
        Thread 2 Calculation Time: 6.800 Seconds
        Thread 3 Calculation Time: 9.438 Seconds
        Time Elapsed       9438 mSecs Total=32.617277 Check Sum = 100000
Iteration 6:
        Thread 0 Calculation Time: 1.380 Seconds
        Thread 1 Calculation Time: 4.115 Seconds
        Thread 2 Calculation Time: 6.785 Seconds
        Thread 3 Calculation Time: 9.415 Seconds
        Time Elapsed       9415 mSecs Total=32.617277 Check Sum = 100000
Average Execution Time: 9.564 Seconds
```

سپس حالت dynamic, 1000 را اجرا میکنیم.



در نهایت نیز حالت dynamic, 2000 را اجرا می کنیم.



در مقام تحلیل این نتایج می توان گفت که اگر به کدی که اجرا می شود نگاه کنیم میبینیم که آن thread ای که j های بزرگتری دارد تعداد عملیات هایی که باید انجام دهد بیشتر است.زیرا در داخل for با متغیر j دو تا for داریم که تعداد دفعات اجرا شدنشان به j بستگی داشته و هرچقدر j بیشتر باشد تعداد دفعات اجرای آن ها نیز بیشتر خواهد شد.لذا زمانی که زمان بندی را static میکنیم thread های با شماره بیشتر که j بزرگتری نصیبشان می شود مدت زمان بیشتری را مشغول اجرا شدن خواهند بود و در عوض thread های با شماره کوچکتر زمان قابل توجهی را بیکار(idle) خواهند ماند که از نظر بهینگی امر مطلوبی نیست.اما زمانی که زمان بندی را dynamic میکنیم هر thread زمانی که اجرای وظایفش به اتمام می رسد درخواست عملیات های بیشتر کرده و به این شکل توازنی بین میزان کاری که به هر thread تخصیص می یابد ایجاد می شود و به همین علت اگر به اعداد و زمان اجرای هر thread دقت شود میبینیم اختلاف ها چندان نیست درحالی که در حالت static اختلاف زمان اجرای thread ها قابل توجه است و همچنین در حالتdynamic به علت توازن مذکور و بهینگی بیشتر برنامه زمان متوسط اجرای این ۶ بار کمتر می شود.اما در دو حالت dynamic که chunk size یکی برابر 1000 و دیگری برابر با 2000 است تفاوت خاصی دیده نمیشود که علت آن این است که تعداد دفعاتی که حلقه اجرا می شود زیاد بوده و chunk size ها نسبت به آن کوچک هستند و به همین تفاوت خاصی بین این دو حالت ایجاد نمی شود.