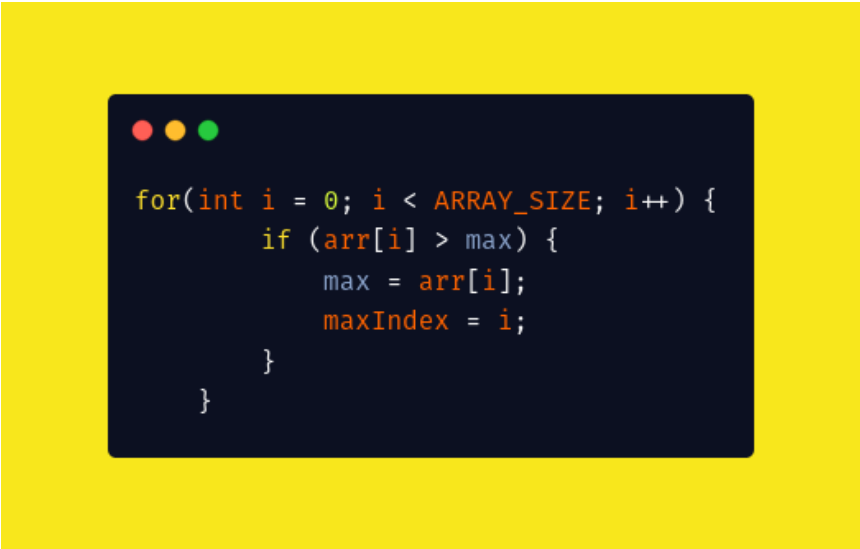
آرین حدادی ۸۱۰۱۹۶۴۴۸
ایمان مرادی ۸۱۰۱۹۶۵۶۰
پروژه چهارم درس برنامه نویسی موازی

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 فایل main.cpp است.فایل سوم هم report است که گزارش کار این تمرین است.

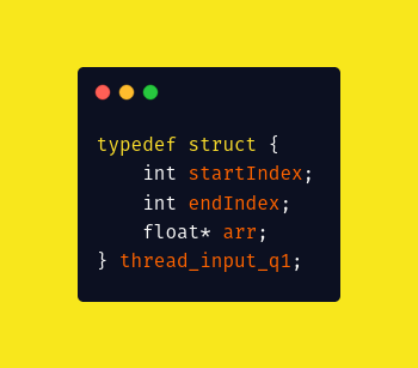
سوال اول)

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 با استفاده از کتابخانه pthreads از چندین thread برای محاسبه ماکسیمم و اندیس آن استفاده میکنیم.به این صورت که به هر thread وظیفه محاسبه ماکسیمم و اندیس بخشی از آرایه را می دهیم و سپس ماکسیمم نهایی را بین این خروجی هر یک از threadها محاسبه می کنیم.
میدانیم که ورودی تابعی که thread شروع به اجرای آن میکند درتابع pthread_create تنها یکی میتواند باشد.به همین علت اگر بخواهیم اطلاعات بیشتری را به ارایه تابع پاس بدهیم نیاز داریم که از یک struct استفاده کنیم.struct زیر را به این منظور تعریف می کنیم.



همانطور که مشاهده می شود برای هر thread مشخص میکنیم که از کدام اندیس تا کدام اندیس را در نظر گرفته و آرایه ای که میخواهد روی آن کار کند را هم به آن پاس می دهیم.

توجه شود می توانستیم که آرایه را به صورت global تعریف کنیم اما چون global برای کد از نظر معیار های clean code مناسب نیست ترجیح داده شد که به این شکل به هر thread یک پوینتر به این آرایه داده شود تا متغیر های global کمتری داشته باشیم.برای آنکه overhead زیادی هم نداشته باشیم صرفا پوینتر به ابتدای این ارایه را به هر thread می دهیم.

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 صورت mutual exclusive باشد 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 از mutex lock ها استفاده میکنیم.

کد زیر بخش ایجاد thread ها است:

```
pthread_t th[NUM_OF_THREADS];
thread_input_q1 thread_inputs[NUM_OF_THREADS];
int chunkSize = ARRAY_SIZE / NUM_OF_THREADS;

pthread_mutex_init(&lock, NULL);

for(int i = 0; i < NUM_OF_THREADS; i++) {
    thread_inputs[i] = {i * chunkSize, (i + 1) * chunkSize, arr};
    pthread_create(&th[i], NULL, findLocalMaximum, (void *) &thread_inputs[i]);
}

for (int i = 0; i < NUM_OF_THREADS; i++) {
    pthread_join(th[i], NULL);
}

pthread_mutex_destroy(&lock);
```

کد زیر هم کد تابعی است که هر thread آن را اجرا می کند:

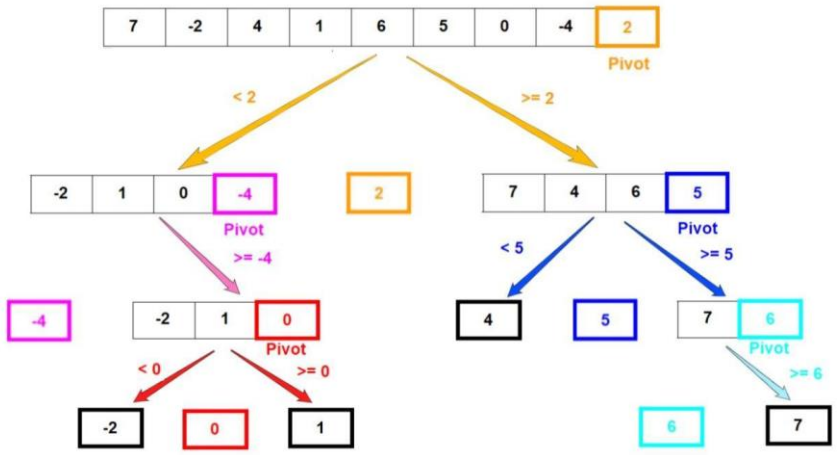
```
void* findLocalMaximum(void* arg) {
    thread_input_q1* input = (thread_input_q1*) arg;
    float* arr = input->arr;
    int startIndex = input->startIndex;
    int endIndex = input->endIndex;

    float localMax = 0;
    int localMaxIndex;

    for(int i = startIndex; i < endIndex; i++) {
        if (arr[i] >= localMax) {
            localMax = arr[i];
            localMaxIndex = i;
        }
    }

    pthread_mutex_lock(&lock);

    if (localMax > maxParallel) {
        maxParallel = localMax;
        maxIndexParallel = localMaxIndex;
    }

    pthread_mutex_unlock(&lock);

    pthread_exit(NULL);
}
```

همانطور که مشاهده می شود هر thread در اندیس خود از ارایه مشترک کار ماکسیمم گیری را برای خودش انجام داده و در متغیری به نام localMax و اندیس آن را در localMaxIndex ذخیره می کند و در نهایت با استفاده از یک mutex lock مقدار ماکسیمم نهایی و اندیس آن ذخیره می شود.اجرای کد این سوال با کد سوال بعد در انتهای گزارش کار 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)



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 است که تابع زیر است.کد بخش محاسبه سریال کهserialQuickSort است نیز در این تصویر دیده می شود.

```
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);
}
```

اما برای اجرای موازی ابتدا باید یک struct مناسب برای ورودی تابعی که ثرد ها در آن شروع به کار میکنند در نظر بگیریم.برای این کار struct زیر را تعریف می کنیم:

```
typedef struct {
    int begin;
    int end;
    float* numbers;
} thread_input_q2;
```

در مرحله بعد ابتدا یک ثرد اصلی و main ایجاد می شود که میخواهد کل این آرایه را مرتب کند.این ثرد تابع parallelSort را اجرا می کند و هربار که ثردی این تابع را اجرا می کند مرحله اول quick sort که partition کردن و تعیین pivot است انجام می دهد و سپس سمت راست و چپ quick sort را به یک ثرد دیگر می دهد که آرایه مذکور را مرتب کنند.به این شکل هر ثرد دو تا از خود می سازد.

تابع parallelSort به شکل زیر است:

```
void* parallelSort(void* arg) {
    thread_input_q2* param = (thread_input_q2*) arg;
    if (param->end - param->begin < MINIMUM_SIZE_OF_THREAD_CHUNK) {
        serialQuickSort(param->numbers, param->begin, param->end);
    }
    else {
        int pivot = partition(param->numbers, param->begin, param->end);

        thread_input_q2 firstThreadParam = {param->begin, pivot-1, param->numbers};
        thread_input_q2 secondThreadParam = {pivot+1, param->end, param->numbers};

        pthread_t firstThread, secondThread;

        pthread_create(&firstThread, NULL, parallelSort, &firstThreadParam);
        pthread_create(&secondThread, NULL, parallelSort, &secondThreadParam);

        pthread_join(firstThread, NULL);
        pthread_join(secondThread, NULL);
    }
    pthread_exit(NULL);
}
```

نکته ای که وجود دارد آن است که از آنجا که تعداد زیاد thread ها باعث افت عملکرد برنامه می شود بهتر است که عملیات شکستن آرایه را در حالت موازی تا یک حدف انجام داده و از آن فراتر نرویم.به این شکل تا زمانی که طول آرایه از یک مقداری که در برنامه ما تحت عنوان MINIMUM_SIZE_OF_THREAD_CHUNK با دستور define مشخص شده است.در کدی که تحویل داده ده این مقدار برابر ۱۰ هزار است.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 مهم است که در هر دو حالت موازی و سریال به صورتی partition را تعریف کنیم که در هر دو آرایه به دو قسمت یکسان شکسته شود، بنابراین مثلا از انتخاب pivot به صورت random می بایست اجتناب کرد تا speedتا فرآیند مرتب سازی در هر دو روش به صورت یکسان انجام شده و اسپید آپ معتبر باشد.

نتیجه اجرا)

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.

میزان speedup در سوال اول معمولا بین ۲ تا ۳ است و گاها از ۳ نیز بیشتر می شود.خروجی سوال دوم نیز بین ۴ تا ۵ است.

```
aryan@aryan-ROG-Strix-G531GW-G531GW:~/Desktop/parallel/Project 5$ ./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 263411
Serial Calculation Time is 4886850 Clock Cycles.
Maximum Element in Parallel Calculation is 1.04857e+06
Maximum Element Index in Parallel Calculation is 263411
Parallel Calculation Time is 1522878 Clock Cycles.
Speedup is 3.20896
-------------
Question 2

Sorting Results Verification was Successful.
Serial Calculation Time is 434484842 Clock Cycles.
Parallel Calculation Time is 96221512 Clock Cycles.
Speedup is 4.51546
```