به نام خدا

گزارش تمرین کامپیوتری ۵ درس برنامه نویسی موازی

دیار محمدی ۸۱۰۱۹۶۵۵۳

رستا تدین ۸۱۰۱۹۶۴۳۶

سوال اول

می‌خواهیم در این سوال بزرگترین عدد موجود در یک آرایه از اعداد ممیز شناور به طول 2^{20} را به همراه ایندکس آن گزارش کنیم.

برای این کار از کتابخانه Pthread استفاده می‌کنیم.

ابتدا آرایه را با اعداد رندم با استفاده از قطعه کد زیر مقداردهی اولیه می‌کنیم.

```
//initialization
   for (int i=0; i < N; i++)
   {
       array[i] = rand()/float(RAND_MAX)*MAX_NUM+0.08;
   }
```

برای قسمت سریال تنها کافی است روی آرایه پیمایش کرده و بزرگترین عدد و ایندکی آن را گزارش کنیم. این کار با قطعه کد زیر انجام می‌گیرد:

```
//serial
   start_c = std::chrono::system_clock::now();
   float max = 0.0;
   int max_idx;
   for (int i = 0; i < N; i++)
   {
       if(array[i] > max)
       {
           max = array[i];
           max_idx = i;
       }
   }
   end_c = std::chrono::system_clock::now();
```

برای گرفتن زمان این کار از کتابخانه chrono استفاده کرده‌ایم.

برای قسمت موازی متغیرهای Max و Idx که مقادیر بیشترین عدد موجود در آرایه و ایندکس آن را نگهداری می‌کنند global هستند و میان تردها به اشتراک گذاشته می‌شوند. همچنین array نیز که شامل آرایه از اعداد ماست global است تا همه thread ها به آن دسترسی داشته باشند.

```
#define N 1048576
#define N_THREADS 6

float array[N];

float Max = MIN_NUM;
int Idx;
pthread_mutex_t lock;
```

از ۶ ترد در این پروژه استفاده کرده‌ایم. بدنه اصلی برنامه موازی به صورت زیر است.

```
//parallel
   int thread_nums[N_THREADS];
   pthread_t threads[N_THREADS];

   start_c = std::chrono::system_clock::now();
   for (int i = 0; i < N_THREADS; i++)
   {
       thread_nums[i] = i;
       pthread_create(&threads[i], NULL, get_maximum, (void
*)&thread_nums[i]);
   }
   for (int i = 0; i < N_THREADS; i++)
       pthread_join(threads[i], NULL);

   end_c = std::chrono::system_clock::now();
```

ابتدا به تعداد N-THREADS رشته ساخته می‌شود و id لوکال شان در thread_nums ذخیره می‌شود (از این آرایه برای به دست آوردن اینکه هر ترد مسئول پردازش کدام قسمت از آرایه است استفاده می‌شود) و هر یک از تردها id واقعی خود را در آرایه threads می‌ریزد. هر رشته تابع get_maximum را اجرا کرده و همچنین local thread id خود را به عنوان آرگومان به تابع می‌دهد.

نهایتا در بدنه اصلی برای تمام رشته‌های pthread_join صدا زده می‌شود تا از تمام شدن کار همه آن‌ها اطمینان حاصل شود.

تابع get_maximum به صورت زیر است:

```
void *get_maximum(void* arg)
{
   int thread_num = *(int *) arg;
   int n = N / N_THREADS;  //number of elements to handle for each thread
   int start = thread_num * n;
   int end = (thread_num != N_THREADS - 1) ? start + n : N;

   float local_max = MIN_NUM;
   int local_idx;
   for (int i = start; i < end; i++)
   {
       if(array[i] > local_max)
       {
           local_max = array[i];
           local_idx = i;
       }
   }
   pthread_mutex_lock(&lock);
   if(local_max > Max)
   {
       Max = local_max;
       Idx = local_idx;
   }
   pthread_mutex_unlock(&lock);
   pthread_exit(NULL);
}
```

ابتدا از void *arg مقدار local thread id که پاس داده شده است بازیابی می‌شود و با توجه به طول آرایه و این id قسمتی از آرایه که مربوط به این رشته است محاسبه می‌شود. هر رشته یک local_max و local_idx دارد که نگهدارنده مقدار بیشینه در قسمتی از آرایه مربوط به این رشته است. هنگامی که این مقادیر با پیمایش از start تا end (قسمت آرایه مربوط به هر رشته) یافت شدند، مقدارهای Max و Idx که global هستند باید به روزرسانی شوند. با توجه به اینکه هر رشته می‌تواند در این متغیرها بنویسد نیاز به استفاده از mutex_lock داریم. هر رشته که این lock را acquire می‌تواند وارد قسمت بحرانی شده و مقدار این دو متغیر را عوض کنید و اگر مقدار موجود در Max از local_max کوچکتر باشد آن را به روز رسانی کند. نهایتا هم باید قفل unlock شود. کار رشته با pthread_exit پایان پیدا می‌کند.

نتیجه این قسمت را به همراه speedup در زیر مشاهده می‌کنید:

```
~/R/u/7/P/C/CAS ./out
Rasta Tadayon:          810196436
Diyar Mohammadi:        810196553

serial results:
max = 999998.438        index = 245298
serial time = 3.108 ms


parallel results:
max = 999998.438        index = 245298
parallel time = 0.772 ms
SPEEDUP = 4.027
```

سوال دوم

در این سوال قصد داریم یک quicksort را روی یک آرایه با دو روش سریال و موازی پیاده‌سازی و اجرا کنیم.

دو آرایه با سایز داده‌شده را با مقادیر رندوم و یکسان برای هر دو آرایه می‌سازیم.

همچنین چون می‌خواهیم همه‌ی تردها به arrayP دسترسی داشته باشند آن‌را به صورت global تعریف می‌کنیم.

```
struct timeval start1, end1, time1;
struct timeval start2, end2, time2;

float* arrayS = new float[ARRAY_SIZE];
arrayP = new float[ARRAY_SIZE];
for (int i=0; i<ARRAY_SIZE; i++)
    arrayS[i] = arrayP[i] = rand();
```

الگوریتم quicksort سریال را اجرا میکنیم. در این روش ابتدا pivot را تعیین کرده و عناصر بزرگ‌تر را سمت راست و عناصر کوچک‌تر را در سمت چپ آن قرار می‌دهیم. حال خود quicksort را روی زیر آرایه چپ و راست اجرا می‌کنیم.

```
void partition(float* array, int &i, int &j)
{
   float tmp;
   float pivot = array[(i + j) / 2];

   while (i <= j) {
       while (array[i] < pivot)
           i++;
       while (array[j] > pivot)
           j--;
       if (i <= j) {
           tmp = array[i];
           array[i] = array[j];
           array[j] = tmp;
           i++;
           j--;
       }
   }
```

```
}

void quickSortS(float* array, int left, int right)
{
   int i = left, j = right;

   partition(array, i, j);

   if (left < j)
       quickSortS(array, left, j);

   if (i< right)
       quickSortS(array, i, right);
}

   //Serial
   gettimeofday(&start1, NULL);
       quickSortS(arrayS, 0, ARRAY_SIZE-1);
   gettimeofday(&end1, NULL);
```

برای روش موازی از تابع partitionP استفاده می‌کنیم که در آن اطمینان حاصل می‌کنیم pivot انتخاب شده از یک چهارم عناصر کوچکتر یا از یک چهارم عناصر آرایه بزرگ‌تر نباشد تا تردی با لود بسیار کم تشکیل نشود.

```
void partitionP(float* array, int &i, int &j)
{
   float tmp;
   float pivot;
   int N = j - i + 1;
   int count;
   while ( (count < float(N) * 0.25) || (count > float(N) * 0.75) )
   {
       pivot = array[rand() % N + i];
       count = 0;
       for (int k=i; k<=j; k++)
           if (pivot < array[k])
               count++;
   }
   while (i <= j) {
       while (array[i] < pivot)
```

```
            i++;
        while (array[j] > pivot)
            j--;
        if (i <= j) {
            tmp = array[i];
            array[i] = array[j];
            array[j] = tmp;
            i++;
            j--;
        }
    }
}
```

پس از انجام partitioning برای اجرای quicksort روی هریک از زیرآرایه‌های راست و چپ یک ترد تشکیل می‌دهیم.

برای این‌کار دو `struct thread_data` می‌سازیم و اطلاعاتی که هر ترد برای کار خود نیاز است را در آن قرار می‌دهیم، سپس دو ترد جدید می‌سازیم و آنها را join میکنیم تا با هم تمام شوند سپس ادامه‌ی کد اجرا شود.

همچنین اگر طول کل آرایه از مقدار `SUB_ARRAY_LEN_CUTOFF` کمتر باشد دیگر نیازی به اجرای موازی و تحمل سربارهای آن نیست و کد به صورت سریال اجرا می‌شود.

```
void *quickSortP(void *arg_struct)
{
    struct thread_data *args;
    args = (struct thread_data *) arg_struct;
    int left = args->st_idx, right = args->en_idx;
    int i = left, j = right;

    pthread_t first_thread;
    pthread_t second_thread;

    struct thread_data index_left, index_right;


    if ( ((right-left)<SUB_ARRAY_LEN_CUTOFF) )
    {
        partition(arrayP, i, j);
        quickSortS(arrayP, left, j);
```

```
        quickSortS(arrayP, i, right);

    }
    else
    {
        partitionP(arrayP, i, j);
        index_left.st_idx = left;
        index_left.en_idx = j;

        index_right.st_idx = i;
        index_right.en_idx = right;
        pthread_create(&first_thread, NULL, quickSortP,
                                (void *) &index_left);

        pthread_create(&second_thread, NULL, quickSortP,
                            (void *) &index_right);
        pthread_join(first_thread, NULL);
        pthread_join(second_thread, NULL);
    }
    pthread_exit(NULL);
}
```

در تابع main اولین ترد را می‌سازیم و کل آرایه را به آن می‌دهیم. همچنین برای اینکه بتوانیم از تردهای‌مان joinable باشند PTHREAD_CREATE_JOINABLE را ست می‌کنیم.

```
//Parallel
    gettimeofday(&start2, NULL);
    pthread_t first_thread;
    pthread_attr_t attr;

    struct thread_data index_main;
    void *status;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    index_main.st_idx = 0;
    index_main.en_idx = ARRAY_SIZE - 1;
    pthread_create(&first_thread, &attr, quickSortP,
                            (void *) &index_main);
    pthread_join(first_thread, &status);
```

```
   pthread_attr_destroy(&attr);

   gettimeofday(&end2, NULL);
```

سپس نتایج دو روش را از نظر صحت و برابری با همدیگر مقایسه می‌کنیم.

```
   for (int i=0; i<ARRAY_SIZE; i++)
   {
       if ((i+1 != ARRAY_SIZE) && (arrayS[i] > arrayS[i+1]))
       {
           valid_result = 0;
           printf("Results are not valid. arrayS[%d] = %f > arrayS[%d] =
%f\n", i, arrayS[i], i+1, arrayS[i+1]);
           break;
       }
       if (arrayS[i] != arrayP[i])
       {
           printf("arrayS[%d] = %f, arrayP[%d] = %f\n", i, arrayS[i], i,
arrayP[i]);
           same_results = 0;
       }
   }
   if (valid_result && same_results)
       printf("Results are valid and the same.\n");
```

زمان‌های اجرا و مقدار سپیدآپ را محاسبه می‌کنیم.

```
   long seconds1 = (end1.tv_sec - start1.tv_sec);
   long micros1 = ((seconds1 * 1000000) + end1.tv_usec) -
(start1.tv_usec);

   long seconds2 = (end2.tv_sec - start2.tv_sec);
   long micros2 = ((seconds2 * 1000000) + end2.tv_usec) -
(start2.tv_usec);

   int valid_result = 1;
   int same_results = 1;
```

```
printf ("Serial Run time = %ld \n", micros1);
printf ("Parallel Run time = %ld \n", micros2);
printf ("Speedup = %4.2f\n", (float) (micros1)/(float) micros2);
```

نتیجه‌ی اجرا: