Rui Huang rhhq7

For homework three, the task was to accelerate the process of calculating distance matrix. I parallelized two parts. One part was assigning each element to buckets. Another was calculating the distance matrix. For the first part, I didn't get much time benefit from increasing the amount of thread. For the second part, the time it cost decreased apparently when the number of thread increased. The following part of this report will discuss the details about the parallelization and analysis the time trend and also what I learned from this homework.

1. Assigning Elements to buckets

   In this part, I loaded the data from csv file to a two-dimension array (following of this report will called this array "data") firstly. Then I parallelized the processes of building argArray, bucketsSize, and assigning elements to buckets. ArgArray was a one-dimension array whose ith element was the argmax of the ith row of data. BucketsSize was a one-dimension array whose ith element was the size of the ith bucket. The reason I built the array bucketsSize was that I could't create a dynamic array of buckets in C, which meant I need to know the size of each bucket before I defined it.

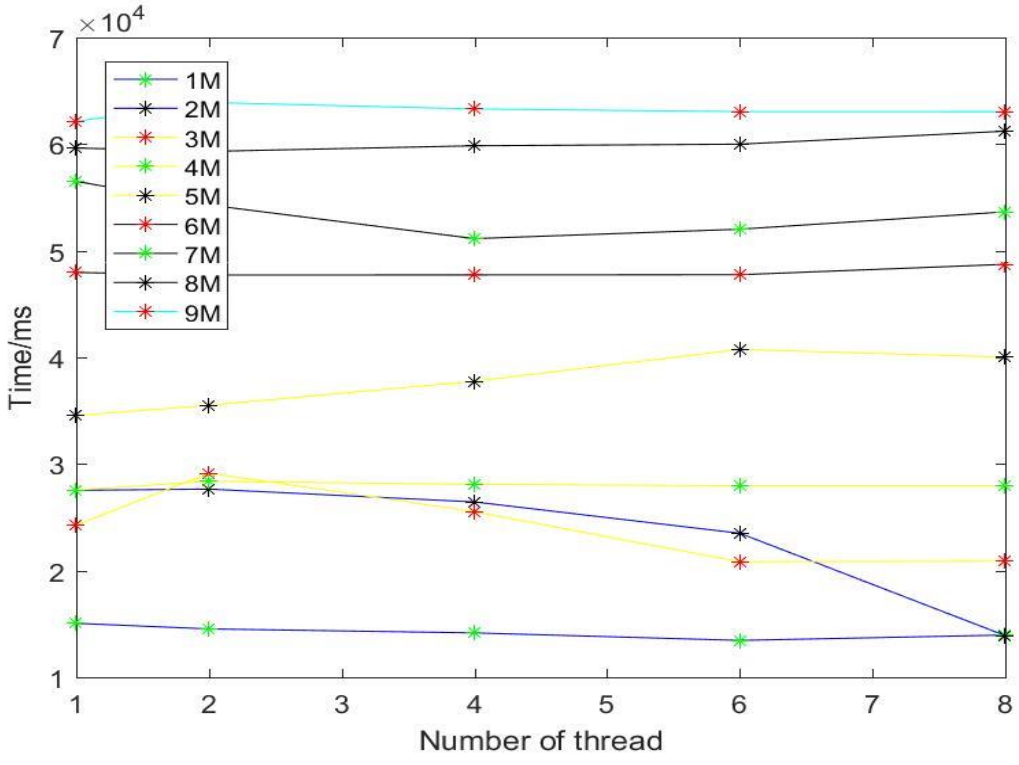
   To accelerate the process of building argArray and bucketsSize, I used the row interleave pattern. I had a for loop, the times it repeated was equal to the number of thread. In each iteration, I created a thread and assigned a function and a struct of parameter to the thread. The parameter struct contains the ID of the thread, number of thread, the pointer to data, the pointer to argArray etc. Once a thread began to run, it knew the ID of itself and how many thread there was, these informations would be enough for this thread to run without any confliction with other threads. For example, there was 3 threads, for the first thread, it processed the 1th, 4th, 7th row of data. After it finished ith row, it set the ith element of argArray to the argmax of this row and increased the ith element of bucketsSize by 1.

   After I built bucketsSize(mentioned above), I could use malloc to apply memory for each bucket and then began putting elements to them. Also, I had an array called curSize whose ith element was the number of elements in ith bucket currently. To accelerate the process of assigning elements to buckets. I used an array of mutex (21 mutexes) to make sure that threads worked without confliction. I also used row interleave pattern in this part. For example,

Rui Huang rhhq7

If there were 3 threads. The first thread process 1th, 4th, 7th rows of data and create structs that corresponding that rows and put them to buckets. The similar happen with the second and third threads. Each time a thread put an element to ith bucket, it increased curSize[i] by 1. The reason I used mutex was that if two threads accessed the same bucket simultaneously, over-write would happen, which meant an element put in the bucket by one thread would be covered by other element put by other thread. The idea to solve this was that I used 21 mutexes, each mutex was for one bucket. If the ith bucket was accessed by a thread, the thread locked it firstly, and then increased curSize[i] and put an element to this bucket and then unlocked it.

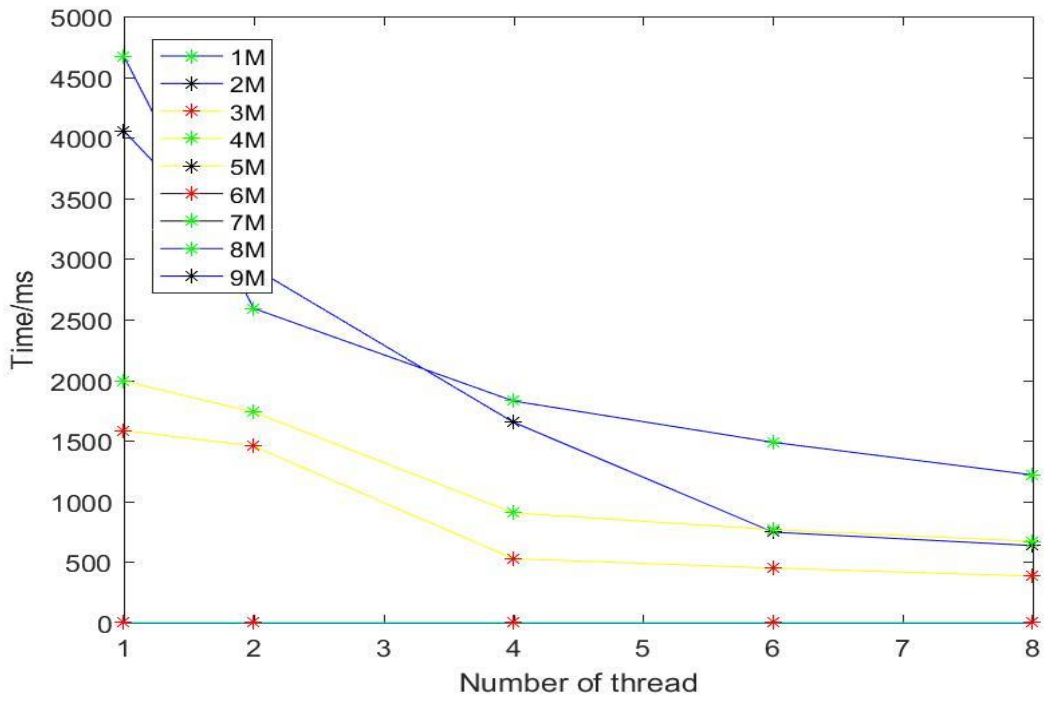
The result of this whole part (building argArray and bucketsSize, assigning elements to buckets) was not what I expected. The time it cost almost remained the same when the number of thread increase. The reason was that it cost some time for OS to manage 21 mutexes probably. Another reason was that it cost some time for one thread to wait for another thread to release the resources. Below is the picture that show the time trend along with the number of thread.



Rui Huang rhhq7

2. Calculate distance matrix and average distance.

For this part, I used row interleave pattern. All threads worked on the same matrix and then after finishing, all threads moved on to work on other matrix. For example, if there were 3 threads, the first thread would work in the 1th, 4th, 7th row, the second thread worked on 2th, 5th, 8th row, and third thread worked on 3th, 6th, 9th row. Another thing that needed to mention was that only upper and right corner of the distance matrix was calculated because the other part will be the same. I had a for loop, the iteration times was the same as number of threads. For each iteration, I used pthread_create to create a thread, and also passed a function and an struct to that thread. The struct contained some information like number of thread and threadID and sum etc. What sum meant was that the sum of the element that thread went through. For example, if there were 3 threads, for thread 1, the sum is the sum of the 1th, 4th,5th ……row of data. After all thread run over, I added all the sum of each thread and I could get the sum of each element in the distance matrix. In this way I could get the average distance. The time it cost decreased when the number of thread increased. Below is the picture that shows the time trend along with the number of thread.



Rui Huang rhhq7

3. What I learn from homework three

   The most important that I learned was how to parallelize program using multi-threads. Also, I learned how to use mutex to make threads work without conflictions. Another thing I learned was the way to know how much time a multi-threads function cost. At first, I use the function clock (), and I didn't realize that there was something wrong. Later, from stack overflow, I knew that, when the function was a multi-threads function, the value we got from the difference between start and end is the sum of time cost on each thread. Start is the return value of clock () called right before the function and end is the return value of clock() right after the function. So, if I use clock(), the time multi-thread cost we get is usually much larger than what it is actual.