1. Abstract

Multi-threads parallelization is use in dropping data into 21 buckets, generating distance matrices for buckets with sizes less than 5001, and calculating averages for generated distance matrices. For csv files with larger than 4 million data, no distance matrices are generated and no averages are calculated.

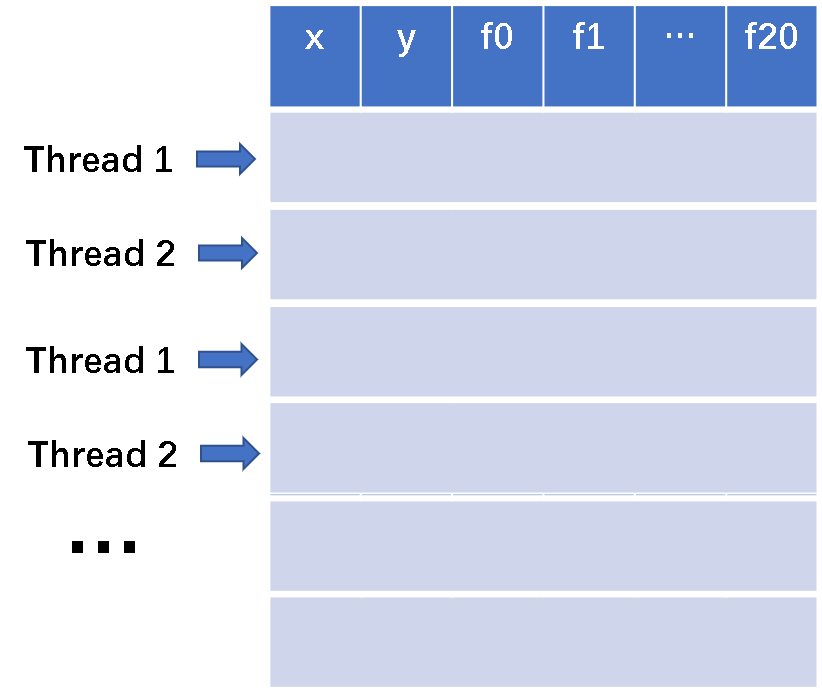
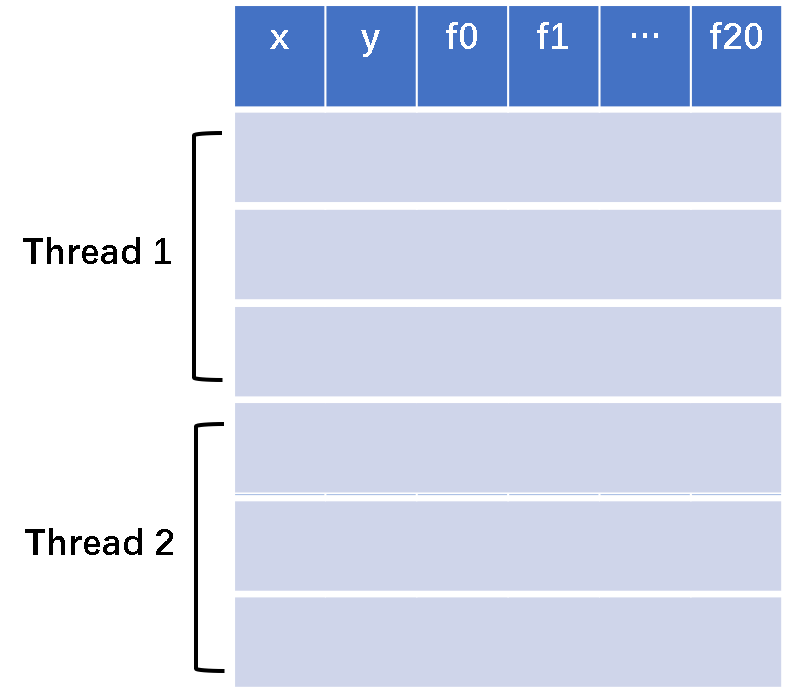
Parallelization of interleaving and blocking fashions are compared and analyzed in the processes of dropping data into buckets and generating distance matrices.

The timings are gathered by running the executables in command line. It seems when running with *srun*, the timing result is the total execution time of all the threads rather than the parallel execution time.

1. Segmenting data into buckets

Two methods of parallelism were tried in segmenting data into 21 buckets based on argmax. The 1st one is to cut the whole data into small chunks and let threads work chunk by chunk in an interleave fashion. In this way, I have to use a mutex to protect the incremental index of current chunk. The 2nd one is to evenly cut the whole data into big blocks, where the number of blocks is equal to number of threads. In this way, I don’t have to use a mutex in reading data.

Of course, in both methods, I have to use 21 mutexes for 21 buckets, and put the code of dropping data into buckets in the critical section.

(a) interleaving (b) blocking

Figure 1 two methods of parallelism tried in segmenting data into buckets

Since in the interleaving method, we have more mutex sections, I expect the running time is longer than that in blocking fashion. And the experiment of segmenting the data from hpc\_1m.csv validates my guess, see Figure 2 for details. When there is only 1 thread, the 2 methods achieves similar time; but with more than 1 threads, less time is achieved with the blocking method. So I kept the blocking method in segmenting data in this project.



Figure 2 timings of interleaving and blocking methods in segmenting data

(a) run executable in command line (b) run executable with *srun*



(c) segmenting data with multi-processes (figure from multi-processes impl.)

Figure 3 timings in segmenting data in case of different file sizes

Figure 3(a) shows comparisons of execution time in cases of different thread numbers and file sizes. It is easy to see as thread increases, time in segmenting data gets smaller, but the decrease amount gets smaller, which conforms the Amdahl’s Law and Gustafson’s Law.

The timing results got by running executables with *srun* go beyond my expectation. Rather than going down with thread count increasing, the time goes slightly up. It seems the time gathered is a total execution time by all threads rather than the CPU time in parallel computing. I guess the slight up in time is due to overhead of creating and managing more threads.

Comparing Figure 3(a) and Figure 3(c), it is easy to see creating a thread has less overhead than creating a process. And it is obvious that threads are more light-weighted than processes and can achieve more speedup.

1. Generating distance matrices

Once the data are dropped into the right buckets, we are ready to generate distance matrices for buckets with sizes less than 5001. The scheme shown in Figure 4 is used in the parallel computation.

First, compute the right half matrix. Each thread is responsible for a block or segment of the matrix. After finishing its current work, it will go to the back and pick up new work. So in essence, this scheme is an interleaving fashion.

After the right half matrix is created, go ahead and fill up the left half by copying the values at the symmetric position relative to the diagonal, with the same scheme.

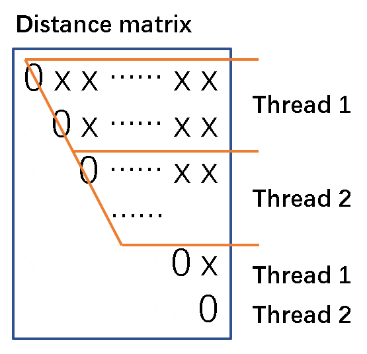
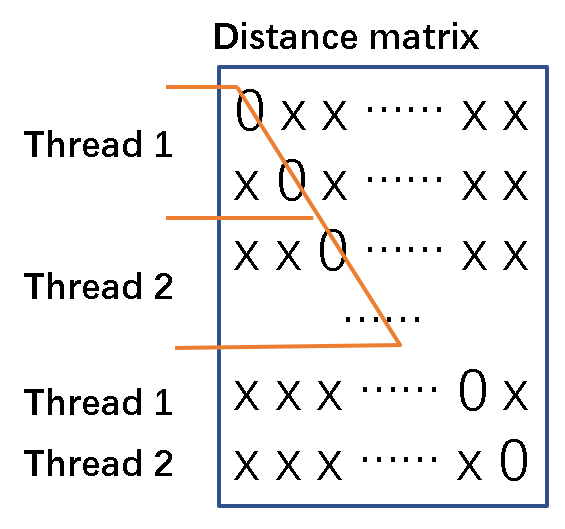
 

Figure 4 scheme in generating distance matrices

Now the question is, how many lines should I put in a block? If only few lines in a block, threads will go to the critical section very frequently to pick up their next assignment, which will definitely harm the parallelization; if too many lines in a block, there is a risk that a thread takes too much burden and everyone else has finished work and has wait for the working thread.

Figure 5 compares the time in generating distance matrices for the file hpc\_1m.csv. Cases with 2 and 8 working threads were experimented. Both cases show the time goes down then goes up again as block size increases, which conforms with my expectation. Comparing these two cases, we can see the minimum time with 2 working threads happens at a larger block size than the 8 working threads scenario. It makes sense. Smaller block size could more evenly distribute the burden among different threads.



Figure 5 trend of time in generating distance matrix as segment size increases



Figure 6 trend of time in generating distance matrix as thread count goes up

Figure 6 shows the trend of time in generating distance matrices as thread count goes up. The statistics is only for the files with less than or equal to 4M lines, since there is no bucket smaller than 5001 for the rest files. It is clear that time decreases as working thread number going up, but the decrease proportion gets smaller and smaller, since the portion of code that cannot be parallelized limits the speedup of parallelization, which conforms with Amdahl’s law.

Note that comparison of the execution time in generating distance matrices for different files does not make sense, because of the difference in matrix number and matrix size.

1. Computing average of distance matrix

In computing average of distance matrix, the same pattern of parallelism with generating distance matrix is used, that is, threads work in interleaving fashion.

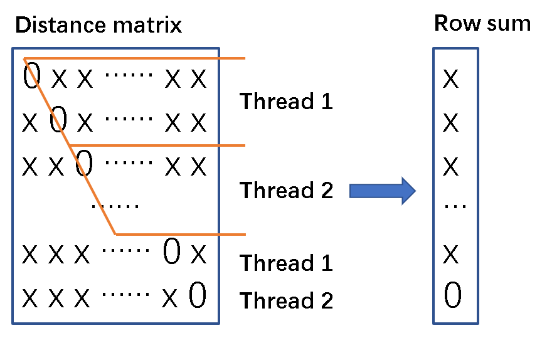


Figure 7 parallel scheme in computing average of a distance matrix

Like shown in Figure 7, each thread is assigned a block in the right half matrix, and do the summation of each line. After finishing one block, go ahead and pick up a next block. After the Row sum vector is obtained, sum up the elements in the vector and multiply by 2 to get the final summation. Then average of the distance matrix is computed as the summation divided by size of the matrix.

There is a same problem of choosing block size here. In this project, the block size in computing averages is fixed at 100 lines.



Figure 8 time trend of computing average of distance matrix as threads increases

From Figure 8 it is easy to see that time in calculating matrix average decreases when thread count goes up, but the decrease proportion get smaller and smaller, indicating that the speedup by parallelism is limited by the mutex section.

1. Lessons learnt from this project

In project 2 and 3, I achieved task speedup with multi-processes and multi-threads. I got a better recognition of the concepts of multi-processes and multi-threads, came to know the usage of functions and methods related to them, and had a great time debugging multi-processes and mutli-threads programmings.

I have to say that parallel programming is a huge topic, and there is still a lot to learn and practice with mutli-processes and multi-threads programning.

I don’t know why a majority of computer code is still not written to exploit the multi-core architecture. Parallel programming is too hard to realize or easy to make mistakes?