1. Abstract

The goal of this task is to accelerate segmentation of (X, Y) tuples into 21 buckets based on argmax, and computation of the distance matrices from the Shared Memory segment(s).

The techniques used in this task include parallel programming with multiple processes, inter-process communication, as well as SLURM.

Note that since the distance matrices are too large, I have to cut down sizes of the matrices to fit my memory. In this circumstance, comparisons of timings in creating the distance matrices for different csv files will not make sense, since no matter how large the csv files are, the distance matrices have to keep small. Hence, let’s focus on the segmentation part.

1. Why got out of memory error in creating distance matrices?

Take the file hpc\_1m.csv for example. After calculating number of (X, Y) tuples in each bucket based on the argmax rules, I get the following statistics.

1. # of (X, Y) tuples in each bucket are:
2. bucket 0:  221618
3. bucket 1:  2158
4. bucket 2:  6736
5. bucket 3:  156780
6. bucket 4:  69206
7. bucket 5:  10958
8. bucket 6:  44578
9. bucket 7:  33263
10. bucket 8:  33186
11. bucket 9:  31190
12. bucket 10: 1430
13. bucket 11: 30797
14. bucket 12: 13380
15. bucket 13: 14581
16. bucket 14: 3543
17. bucket 15: 1723
18. bucket 16: 124294
19. bucket 17: 101354
20. bucket 18: 46650
21. bucket 19: 11492
22. bucket 20: 41083
23. Total: 1000000

Then sum of the number of float type elements in 21 distance matrices is: . Suppose each element takes up 4 bytes, then we need at least to store the 21 distance matrices. Checking memory info. on TC with the command *cat /proc/meminfo*, I got only memory, far less than the requirement of 428*GB*. Thus I have to cut down the distance matrix size for (X, Y) tuples in each bucket, to prevent the out of memory error. So even though number of (X, Y) tuples increase when parsing larger files, the matrix size cannot grow, and we have no way to compare timings in creating the matrices as their sizes growing up along with the larger csv files.

1. Pseudocode
2. Overview of the whole program

My program is structured as a result of the requirement for this task.

1. Load data, *D*, from file into Memory.
2. # In Parallel, up to *P*:
3. Segment *D* from Memory into collections, *Ci*, in Shared Memory.
4. end **for**
5. # In Parallel, up to *P*:
6. **for** Each  **do**
7. Compute *Mi*
8. Store the computed *Mi* into a std::map data structure
9. end **for**
10. Free and Release Shared Memory
11. Report Timing Statistics

First, load the data from csv files using the parser created in task 0, and store the data in a map structure.

Then, segment the whole data into 21 buckets based on the argmax rule. In this part, *P* processes were used to do the work in parallel. The procedure of dropping data into the same bucket has to be mutually exclusive for each process, and I used semaphore to realize the mutex.

Third, compute distance matrices for (X, Y) tuples for all the buckets. Of course, I have to limit the matrix size before the computation. The computation was also done by *P* processes in parallel. Since different processes deals with data in different buckets, we don’t need to set up a mutex for this procedure.

In this end, write distance matrices data back to parent process, release the shared memory and mutexes, and report timings for different procedures.

1. File parsing into a map structure

This part copies the code developed in task 0. The program walks down the csv file and manage the whole data in a *std::map* structure, where the key are (X, Y) tuples, denoting coordinate of points on this planet, and the value are vectors of 21 features. The only difference is that we don’t need to consider the header in this task.

1. Forking children processes

The procedure of forking several children processes from the same parent process is nontrivial. I tried 2 methods in forking processes.

1. pid\_t pid;
2. **int** i;
3. **for**(i = 0; i < P; i++)
4. {
5. pid = fork();
6. **if**(pid == 0 || pid == -1)  // only parent process forks
7. **break**;
8. }
9. **if**(pid == -1)
10. {
11. perror("fail to fork!\n");
12. exit(1);
13. }
14. **else** **if**(pid == 0)
15. {
16. // child process goes here
17. exit(0);
18. }
19. **else**
20. {
21. // parent process goes here
22. }

The 1st method created *P* children processes using a for loop. For each child process, the value of *i* is different, so I can use *i* to differentiate each child process in the child process code. In the for loop, I need to check *pid* after forking with an *if* statement, to make sure only the parent process forks. Otherwise, child process will create grandchild processes. After forking *P* children processes, the parent and children processes can be separated by an *if else* statement.

1. **for**(**int** i = 0; i < P; i ++)
2. {
3. pid\_t pid = fork(); // fork, which replicates the process
4. **if** (pid < 0)
5. {
6. std::cerr << "Could not fork!!! ("<< pid2 <<")" << std::endl;
7. exit(1);
8. }
9. **else** **if**(pid == 0)
10. {
11. // child process goes here
12. exit(0);
13. }
14. **else**
15. {
16. // parent process goes here
17. }
18. }

The 2nd method is more straightforward. Fork the children process within a for loop, and then separate parent and children processes by referencing different *pid* values.

For both methods, I need exit(0) after the child process is done; otherwise, child process will go beyond the brace and execute the code that follows the *for* statement, which will mess up my logics.

In my code, core work of segmenting data into buckets and creating distance matrices are done by children processes. The parent process only deals with the configuration and timing job. To make it consistent, a child process is forked and do the core work even if *P* = 1 when running the executable.

1. Setting up the shared memory

In the work of segmenting data into buckets based on argmax, I put the 21 buckets in shared memory, so that all the children processes have access to the buckets. Along with the buckets, I allocated shared memory for *bucket\_size*[21], which records count of (X, Y) tuples in each bucket. When one child process drop a tuple into certain bucket, increase *bucket\_size* for that bucket.

In the work of generating distance matrices, I put the 21 distance matrices in shared memory, so that after the matrices are created, they can be copied back to the parent process.

1. Setting up the semaphore

In the work of segmenting data into buckets, I need a mutex for each bucket, so 21 in total, to prevent children processes from dropping different (X, Y) tuples into the same bucket, and increasing *bucket\_size* for that bucket at the same time. The code is adapted from the example code of semaphore provided on Canvas.

In the work of generating distance matrices, it is unnecessary to use semaphores. Since children processes are assigned to generate matrices for data in different buckets, there is no way for different processes to access the same data at the same time if the code is right.

1. Share the burden among children processes

After segmenting data into 21 buckets, I found sizes of buckets could be very different. For example, after parsing the hpc\_1m.csv file, bucket 0 has 221618 (X, Y) tuples, but bucket 1 has only 2158 tuples. That means distance matrix 0 could be more than 10,000 times larger than distance matrix 1. And it is possible that all the other children processes have done their work, and wait for very long time for the one dealing with distance matrix 0. Thus, we need some way to share the burden among children processes.

I came up with one way to make burden for different children processes as similar as possible. The pseudocode goes as follows.

1. get the number of tuples in each bucket *Ni*
2. Calculate size of each distance matrix *Di = Ni\*Ni*
3. sort *D* from largest to smallest, we get
4. initialize *P* empty sets
5. **for** i = 0...*P*-1
6. add  to *Pi*, that is *Pi* =
7. end
9. **for** i = P...21-1
10. find *Pj* which sum(*Pj*) is smallest
11. *Pj* = {*Pj*, }
12. end

After parsing the file hpc\_1m.csv, suppose we have 4 children processes, the assigned tasks for each process will be:

* child process 0 will tackle bucket: 0
* child process 1 will tackle bucket: 3
* child process 2 will tackle bucket: 16 6 7 8 11 1 15 10
* child process 3 will tackle bucket: 17 4 18 20 9 13 12 19 5 2 14

where numbers denote indices of of the original bucket sequence.

1. Warping up

After the work is done, shared memory and semaphores must be freed and released, otherwise they will stay in the operating system even though my executable is done. We can use the command *ipcs* to have a double check.

1. Timing analysis
2. Time in parse different files

From Figure1 we can see time in parsing files has an almost linear relationship with file sizes. Since only the parent process deals with the procedure of parsing file and creating the map structure, this result fits my expectation.

 

Figure 1 Trend of time in parsing files with file size growing

Figure 2 Trend of time in segmenting data as parallel process count grows

1. Time in segmenting data into buckets

Figure 2 shows the time consumption in segmenting the whole data into 21 buckets.

When the file has only 1 million lines, using more processes will slightly increase the time. Perhaps the time in creating and deleting children processes, managing the process control block, and swappng processes in and out outweigh the time saved by mutli-processes.

When the file has more lines, we can see the time consumption goes down as number of processes in the segmenting work grows. But the time decreasing degree gets less and less as the number of processes grows. Finally, the time curve goes flat, and increasing processes cannot decrease the time any more. At that point, the portion of program that is not parallelized will limit the overall speed up available from parallelization. This is also what the Amdahl's Law says.

When view the curves in a vertical way, say, let’s keep the process count to be 4, when file size gets larger, the time cost in segmenting the data goes higher, too.