COMP5112 Final Report

LI Yang
20750699
ylikp@connect.ust.hk

Ⅰ Introduction

This is the final report of course COMP5112 which aims to review and describe my past three programming assignments: MPI, Pthread, and CUDA. The description may include the design and implementation of the programs, how parallelism is achieved in the programs, and performance improvements or lessons learned if any.

These three programs are trying to solve the same problem in graph clustering but the sequential versions have some differences. The sequential programs of Pthread and CUDA are the same which have two separate stages. In the first stage, it will find the pivots and then expand from these pivots to their neighbors to find the clusters in stage2. The sequential program for MPI is different with only one stage to find the final results.

The remaining part of this report is organized as follows: section II~IV is the description of the three programming assignments, and section V is the comparison and discussion part.

Ⅱ Description of MPI Programming Assignment

a) Design and Implementation

I divide my MPI assignment into four main steps as below:

The first step of the MPI program is to broadcast the number of graphs that need to be dealt with in each process. In this step, I use the *MPI_Bcast* function to broadcast the *num_graphs* value and then each process calculates the *local_num_graph* value. We have the assumption (only for assignment1) that the total number of graphs can be evenly divided by the number of processes so we don't need to do more actions in this step.

Then the second step is to define a derived type for *GraphMetaInfo* because it is not a native data type for MPI. In this part, I use two *MPI_Aint* values to build a new *MPI_Datatype* and use *MPI_Scatter* to distribute this new type of data to each process.

Now we need to distribute the *nbrs* and *nbr_offs* to each process but they have different sizes. So I first calculate the numbers to be sent for each process, store them in some arrays and broadcast these arrays to processes. Finally, I use *MPI_Scatterv* to distribute the values. And I also calculate the size of data to be sent back from each process later.

In the final step, we can do the SCAN algorithm in each process so I first allocate space for local results and do the *clustering* function in each process. A difference with the sequential version in this part is that I use a pointer to store the local results rather than an array. After doing the SCAN algorithm, I use *MPI_Scatter* and *MPI_Scatterv* to collect the local results to the main process.

The total flow of the MPI program is as below:

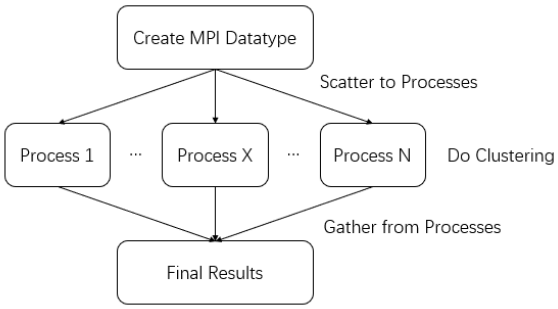

Figure 1: Flow of the MPI program

b) Analysis and Discussion

MPI works with distributed memory rather than shared memory so the main parts of my assignment are about memory allocation, data distribution, and data collection. The parallelism is achieved by distributing the data to different processes and do the SCAN algorithm inside each process. Processes did their own work in parallel and we collect the results back to the main process.

Some useful knowledge I have learned in this assignment is that we need to be careful about the data size to be distributed or collected. If the data size for each process is the same we can use *MPI_Scatter* and *MPI_Gather*. And if they are different we need to have an array to store the data size and then use *MPI_Scatterv* and *MPI_Gatherv*.

In my submission, I forgot to change the number of distributing from a certain number (*3*) to *local_num_graphs* in two *MPI_Scatter* functions. I use *3* in these two lines because I tested the program with a certain test file & setting at the beginning and I only changed the later parts. I need to pay more attention to the details the generalization of my programs.

III Description of Pthread Programming Assignment

a) Design and Implementation

Similar to the first step in MPI, I calculate the first and last graph number my_first, *my_last* that a thread will deal with later. In this case, we need to consider the situation that the total number is not divided by the number of threads.

Then each thread will do the first stage of the SCAN algorithm which is to find the pivots. In this small part, it is the same as the sequential version.

After stage1, we need to synchronize different threads and I use a condition variable together with a mutex here to do this task. First I use *pthread_mutex_lock* to lock the mutex and then use a variable *counter* to count how many threads have finished their tasks. When all threads have finished stage1, *pthread_cond_broadcast* will invoke all threads again to the next stage.

In stage2 of the SCAN algorithm, the basic idea is to start from every pivot and expand to find its clustering neighbors. In the sequential version, there is a global array *visited* to store whether one vertex has been visited. But here different threads may start from different pivots and reach a certain vertex many times. From the SCAN algorithm, the clustering result for each vertex should be the smallest number of its clustering neighbors. If one vertex has been assigned a label from another vertex with a large label and the array *visited* has stored this result, the final clustering result will have many mistakes. So I use *local_visited* arrays for each

thread to store this visiting result locally.

Then the condition for each thread to change the label has also changed. Each thread need to check if the *clustering_result* for this vertex is larger than its label (has been changed by a larger vertex) or equal to *-1* (has never been changed). After this check, we will start the recurrent invoking of function *expansion*.

In function *expansion*, it will expand from one vertex to its neighbors and do it recurrently. Here I also set the same check condition as above when it tries to change the clustering result. In addition, I put a read-write lock here and use the write lock *pthread_rwlock_wrlock* & *pthread_rwlock_unlock* to ensure that no mistakes will happen when different threads are changing the same vertex's result.

After all these parts, the final result is stored in the *clustering_result* array to be sent back to *main.cpp* for later usage. The total flow of the Pthread program is as below:

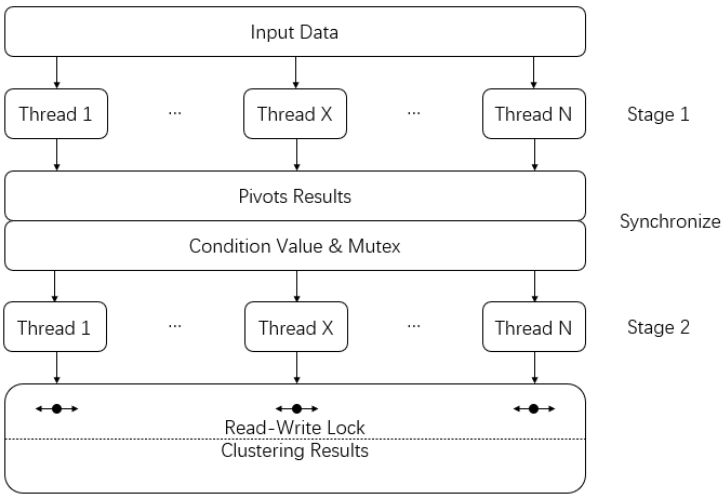

Figure2: Flow of the pthread program

b) Analysis and Discussion

Pthread works with shared memory so we don't need to distribute and collect. We define the global variables for the main thread and local variables for other threads.

The parallelism in this program is reached by assigning different start and end points for each thread. Then each thread will read data from different locations of the same global array and put the result in different locations as well. In this situation, we need to pay attention to the situation when reading and writing of the same position. Also, we need to synchronize different threads between stage1 and stage2 because they may have different arrangement orders by CPU.

The SCAN algorithm used in this program requires starting from one vertex and expanding to other vertices so it may cross the data boundary for each thread. To prevent the potential mistakes I set the *local_visted* array and use the read-write lock in stage2. This operation can ensure the final label of each pivot will be the smallest one.

Ⅳ Description of CUDA Programming Assignment

a) Design and Implementation

The CUDA programming assignment is different from the previous two assignments

because it includes two devices: CPU and GPU, to do the task together. I divide this program into three main parts.

The first step is to do the data preparation and distribution. We read the input variables from files into main memory and we need to send it to GPU first. I named my variables with the head of *Host_* for those in main memory and *Device_* for those in GPU memory. In this step, I also calculate the size of each array in the two-dimensional array *sim_nbrs*. It is named *Host_size_index* and will be copied to *Device_size_index* and later we will use it in GPU memory. In summary, I use *cudaMalloc* and *cudaMemcpy* to allocate space for device data and copy some data to the global memory of GPU in this step.

The next step is to find the pivots in GPU with the device data we have got in the previous step. This device function is named as *__global__ void stage_1* to be called by both host and device. Another function used in *stage_1* is *__device__ int get_num_com_nbrs* and can only be called by the device. I first calculate *my_thread_rank* for each thread to determine its task location. Then I use the coalesced access way to do the iteration in stage1. The remaining part is similar to the sequential version except for treating *sim_nbrs* as a one-dimensional array with the help of *size_index* array.

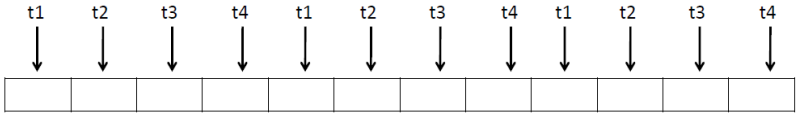

Figure3: Coalesced access used in stage1

After finding the pivots in stage1 I first use *_syncthreads* and *cudaDeviceSynchronize* to synchronize different threads. Three result arrays *Device_pivots*, *Device_num_sim_nbrs* and *Device_sim_nbrs* are copied back from GPU to CPU. Then I apply the stage2 in CPU so it will be the same as the sequential version. Finally, I do the memory-free for both CPU and GPU memory. The total flow of the CUDA program is as below:

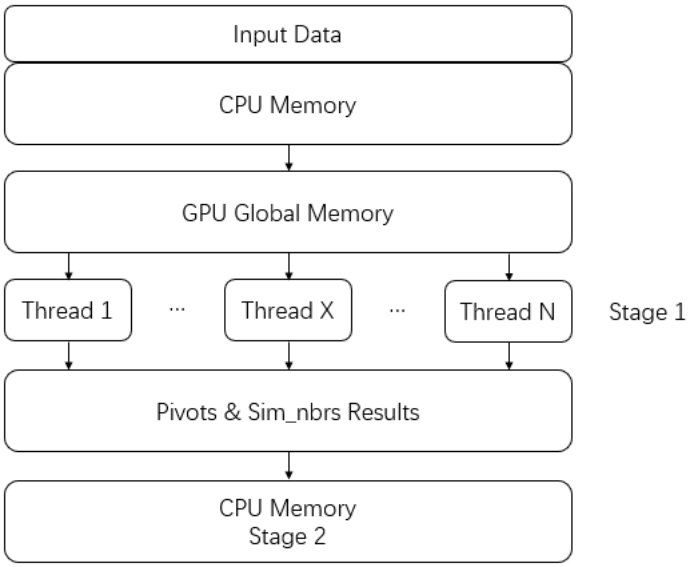

Figure 4: Flow of the CUDA program

b) Analysis and Discussion

CUDA program works with two different memories and the GPU memory's operation is different from CPU memory. We need to allocate the GPU memory and copy the inputs there in advance.

The parallelism is achieved by dividing the tasks into different threads in stage1. Each thread in GPU will read their data from global memory, do their own tasks, and finally put the results back into global memory.

I don't know how to use dynamic two-dimensional arrays in GPU memory so I transfer a two-dimensional array to one dimension. Then it is easy to use in GPU memory. In addition, I only parallel stage1 in GPU and do the stage2 in CPU as the sequential version because there are no mutexes or locks to avoid the conflict. If there is a potential way to do stage2 in parallel with CUDA, it would be a good improvement.

Ⅴ Comparison and Discussion

To have a clear comparison and discussion of these three programming assignments, I will focus on two parts: similarities and differences.

a) Similarities

The basic idea of achieving parallelism is very similar in these three programs: divide the task and let each process/thread do its own task. These tasks belong to the data parallelism type and they share the similar work flow: 1) set index/distribute data for each process/thread, 2) each process/thread work with their own paces, 3) collect/store the final result.

b) Differences

Although the basic idea is similar, their implements are very different.

In MPI, each process has its own memory space so we need to distribute the data to each process and collect results back at last. In addition, MPI only supports several basic value types so you need to define your own MPI value types if you want to send/collect structs like C/C++. And each process works in their own space so there is no need to pay attention to the read/write conflict.

In Pthread, different threads share the same memory space so we don't need to distribute and collect data. We define global variables to be used for all threads and local variables for a specific thread. The problems followed with this feature are: 1) there may be different threads trying to read & write the same location in memory, 2) different threads may finish their own work at different time points. A read-write lock and a condition value are used in my assignment to handle this problem. Without these actions the program may cause mistakes or finish with different results for each running time.

In CUDA, the CPU and GPU use different memory spaces and the GPU memory needs to be allocated in host codes. CUDA does not offer mutexes or locks to deal with the conflict in writing so it loses some parallel functions. Another problem is that you can not use dynamic two-dimensional arrays in GPU memory as easily as in CPU memory. So I use an array to store the indexes to solve this problem.