Experiment On Implementing PSRS Algorithm By MPI

Lingbo Tang

lingbo

Introduction:

MPI refers to Message Passing Interface. It is primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process. [1] Unlike implementing PSRS algorithm with shared memory (like Pthread), MPI tries to distribute data evenly to multiple address spaces. It become intuitively suitable for PSRS algorithm. And since MPI has five major advantages on handling data distribution, such as standardization, flexible portability, performance opportunities, it is interesting and important to use this interface to research load balancing problems. [2]

Implementation and Process:

MPI has a list of functions that can be directly use for Collective Communication Routines. For example, MPI\_Scatter and MPI\_Scatterv can be used to scatter a distinct subset of data to different process. The portion of data is not necessarily to be scattered evenly, but for convenient purpose, I still used evenly scattered set of data to be scattered. I used those two function to scatter unsorted data in phase 1. Then after sorting regular samples, I used the MPI\_Gather and MPI\_Gatherv to gather samples into main task, they are used as inverse function in Phase1. After sorting pivots candidates and picking up the final pivots in Phase 2, I will use MPI\_Bcast to broadcast the same pivots set to all process space. Unlike using sequential search to find pivot index in Assignment 1, I used binary search in Phase 3, and theoretically, it will reduce the time complexity from to . Then I used MPI\_Gather twice to gather both divided index and segment lengths. In Phase 4, I implemented multi-way merge instead of sequential 2-way merge which will improve the time complexity to where L is the sum of segment length and the space complexity becomes 2L+N. Finally, merge all the sorted segment into main task MPI\_Gather again. I also used the MPI\_Barrier after each collective communication routines function had been called. On the remote side, I setup 4 identical machines, each with 2 virtual CPUs and 2GB RAM. So in total I have 8 cores to use and 8GB memory to allocate. My local machine has 4GB RAM and 4CPUs.

Data Analysis:

The followed graph is the speed up that I got from the remote VM measured by the total time spent. The x-axis in the line graph is the number of processors, and the y-axis is the speed up value. For all number of keys that I have been tested, we can see that s(p) < p, which means I have a sub-linear speed up on each (Number of Keys, Number of Processors) test pair. And compare to the theoretical 1,2 ,4, 8 speed up, I have a very descent number. For example, the best value I have is from (16000000,2), which it is nearly 0.97 time as the theoretical 2 times speed up. Also theoretically, the larger number of keys we give in our test, the more speed up we should expect. But it turns out that my 16000000 and 64000000 keys testing are slightly faster than the 8000000 and 32000000 keys. But the difference is small, so I am assuming that even 64000000 keys didn't even maximize the granularity enough. For example, each key is long type with 4 bytes in memory, so 6.4\*10^7 keys means only approximately 244MB data are processed through in total 8GB memory. Although the pipeline traffic might affect the speed up, 244MB will not make significant difference compared to 30MB (8000000keys). I noticed that when comparing The phase 2 time differentiate significantly, I think it was caused by the pipe line traffic. Because phase 2 has to wait for all subarray being blocked by the MPI\_Barrier function.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| #Keys | 1PE | 2PE | 4PE | 8PE |
| 8000000 | 1 | 1.738313551 | 2.979803987 | 4.287258497 |
| 16000000 | 1 | 1.94355741 | 3.316060936 | 5.101369546 |
| 32000000 | 1 | 1.423855488 | 2.522154162 | 3.822800884 |
| 64000000 | 1 | 1.811793804 | 2.923161464 | 4.906049617 |

Speed up table of remote VMs Figure.1

Speed up chart of remote VMs. Figure 2.

The results that I get from local machine is better when I distribute them into 2 and 4 process. However, since I only have 4 process on my local machine, the results that I get from running by 8 processes get less speed up than the results I get from running by 8 processes on remote VM.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| #Keys | 1PE | 2PE | 4PE | 8PE |
| 8000000 | 1 | 1.885548915 | 3.219213306 | 4.07502977 |
| 16000000 | 1 | 1.8393921 | 3.53184265 | 4.459534328 |
| 32000000 | 1 | 1.82575802 | 3.61291243 | 4.38423023 |
| 64000000 | 1 | 1.813180783 | 3.322304722 | 4.112381573 |

Speed up graph and Chart on Local Machine. Figure 3

Again, x-axis are number of processors and y-axis are speed up value in the above speed up graph. I think there’re two major facts that make the difference between the results of local and remote. First, the memory architecture is not the same. When we run MPI on remote VM, we are using distributed memory model. But on local machine, we are using shared memory model or hybrid.

As for phase by phase analysis, first we want to verify the speed up for each phase are reasonable. I’ll give an example from 64000000 keys. (Showing in Figure 4) They almost follow the linear speed up except for phase 2. Some times the network between each VMs are bad, so waiting to gather all samples from Phase 1 takes more time. And if no waiting is needed, phase 2 time will be reduced significantly. This happens very soon when processor number are small, and original data is large.Therefore you can see that phase 2 time differentiate a lot. How MPI communicate between two processors is simple. A processor will hold a process on itself, the process will have two main components, application buffer and system buffer. Process A will send data from its application buffer to process B’s system buffer. Since I’m using MPI collective functions that ensures the safety of, I think there are some overhead and latency in their function that does similar job as MPI\_Wait. (Collective Communication Routines insures Synchronization safety, data movement safety, and Collective Computation).

|  |  |  |  |
| --- | --- | --- | --- |
| Time | 64, 2P (s) | 64, 4P (s) | 64, 8P (s) |
| Phase1 | 6.8694012 | 3.1511192 | 1.8756426 |
| phase2 | 1.5528912 | 0.572702 | 0.149348 |
| Phase3 | 1.4214852 | 1.0292306 | 0.8535678 |
| Phase4 | 0.5362346 | 0.3485866 | 0.2256308 |
| Total | 10.3800122 | 5.1016384 | 3.1041892 |

Phase by phase analysis on 64 million keys example Figure 4.

Phase by phase real time chart Figure.5

x-axis: #processors, y-axis: time (s)

Phase by Phase Real Time Percentage. Figure 6.

As what wee can see from Figure5 and Figure6, all phase time reduced significantly with slightly the same speed up rate when number of processors increased. And both phase 3 phase 4 took less time in all 4 phases. Phase 4 has almost same absolute execution time for each run. I can conclude that phase 3 and phase 4 both reached their optimization criteria by using binary search and multi-way merge algorithm. Phase1 takes most of the execution time, because it has to execute quick sort in each divided array (The amount of keys to sort in each processor is still large, if number of processor is 4, then the number of keys that we have to sort in phase1 is 16000000 on each processor). Interestingly, phase 3 increased time percentage and phase 2 decreased time percentage when number of processors increased. Because, the change of amount of sending and receiving regular samples are small enough to be ignored (if #procs changed from 2 to 4, then # of samples changed from 4 – 16). But the amount of sending and receiving data changes larger than that. (Number of index stored changed, number of chunk lengths changed.)

When executing test on my local machine, the results is a little different from what I got from the remote VM.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 64,2P | 64,4P | 64,8P |
| Phase1 | 1.631082571 | 1.133556143 | 1.132455 |
| Phase2 | 0.001798714 | 0.032830286 | 0.010731286 |
| Phase3 | 0.310147286 | 0.379632571 | 0.435460286 |
| Phase4 | 0.294290571 | 0.255551 | 0.438718 |
| total | 2.237319143 | 1.80157 | 2.017364571 |

Local phase by phase time table Figure 7.

Local phase by phase real time chart Figure 8.

Local phases by phase real time percentage Figure 9.

Phase 2 takes much less time than executing on the remote VM because by using the local memory, there’s almost no communication waiting in phase2. Not surprisingly, phase 1 still takes most of the execution time. Phase 4 is stable as expected. However, when we increase the amount of processors, phase 3 started to increase its execution time. It is because the communication waiting between each process in my local machines increased with a more significant amount than the waiting on the remote side. (MPI collective function have to ensure the communication safety for shared memory model too). The total time still demonstrate a sub-linear speed up, which is what I expected. But the absolute amount of time being used to execute the algorithm is less than the execution time on the remote VM.

Conclusion:

By analyzing the total time on local and remote side, we can conclude that both sides demonstrate a descent sub-linear speed up and granularity. In addition, with the analysis I’ve made why phase 2 and phase 3 time changes differently on local side and remote side, we can conclude that MPI collective function handles better load balancing on distributed memory model than shared memory model, because it reduced more time of communication waiting and distributed data more evenly on different instances.

**Reference:**

[1] <https://computing.llnl.gov/tutorials/mpi/>

[2] <https://computing.llnl.gov/tutorials/mpi/>