Parallel And Distributed Computing

Fall 2014 Assignment 2 : Report

ABSTRACT

The report is case study of how different parallel computation perform by varying the size of problem, number of node and number of threads. The problem taken here is simple algorithm to sort a random distributed. The finally based on the result graph is plotted and compared.

By

Vijay Manoharan
manohara@buffalo.edu
#50097716

Table of contents.

Sequential

Problem: To sort the array of n number using bucket sort.

Algorithm.

The input is taken from command line as ."/sequential n Bucket_numbers". The bucket size is the number of buckets we make and n is the number of values in the array for the problem, for our convention we call that as "size of the problem".

- The input is normal distributed number between 1 and size of the problem.
- The algorithm uses the following to sort the values.
    - Combination of Quick and selection sort called "quickSelect". This does partially sorting in O(n) time complexity.
    - For sorting values inside each bucket we use Mergesort
- Therefore the entire complexity will be O(n)+O(nlogn) which is equal to O(nlogn)
- Distribute the numbers uniformly into all the buckets. Say we have 100 size and 4 buckets then we must divide the 25 values in each bucket.
- For this we must range of values for each bucket called pivots. The bucket one will the values from pivot(0) to pivot(1) and the last bucket i.e., say nth bucket will have the values from pivot(n) to pivot(n+1).
- The values in the buckets are assigned by using QuickSelect.
- Each bucket is sorted one by one using merge sort function.
- Finally the sorted values are printed, and stored in result_sequential_size<size>_bucket<bucketsize>.out.

  For case of comparison with scaling we use the number of buckets equal to the scaling factor. Scaling factor is determined as:
- OpenMP – scaling factor equals number of threads.
- MPI – scaling factor equals the number of nodes.
- OpenMP_MPI – scaling factor equals the number of processors times' the number of threads in each processor.

The comparison is made by varying the input size (100 to 1,000,000) and bucket size in the order of 1,2,4 and 8. The following results are obtained and tabulated.

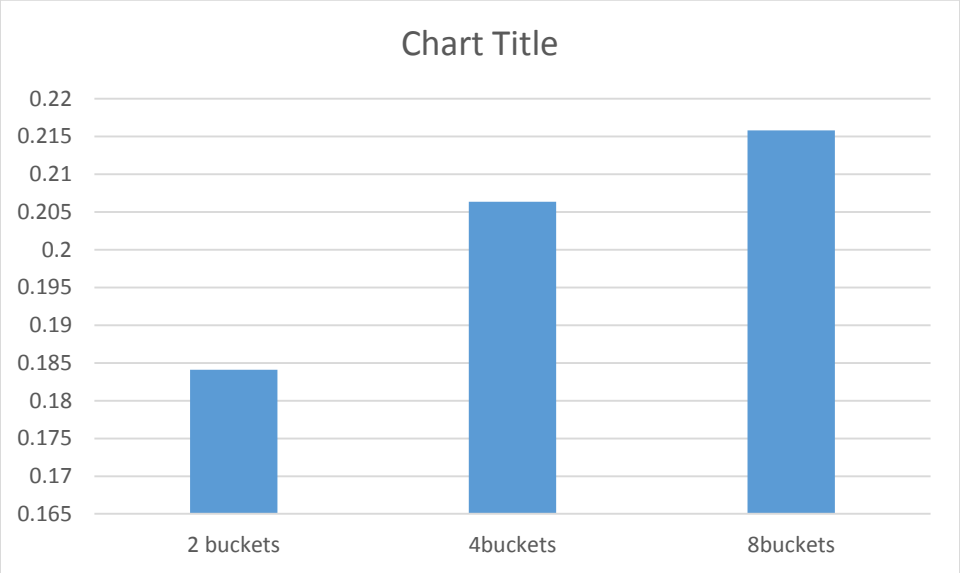
The computation time of the sorting is taken to consideration.

The graph between size of problem and time has plotted.

Graph and table of data collected during the program execution.

10 power x	Time in Secs 2buckets	Time in Secs 4buckets	Time in Secs 8buckets
2	0.000036	0.000037	0.000045
3	0.000158	0.000155	0.000175
4	0.001405	0.001606	0.001701
5	0.016758	0.01822	0.019233
6	0.184074	0.206337	0.215815

The data for varying the data size and time is recorded in the table shown.



The above plot is the time vs number of buckets for a constant size. The lower bucket gives the better running time.

Open MP

The above sequential algorithm is used and made parallel by using OpenMP funtions. The sequential and the OpenMP is similar, expect

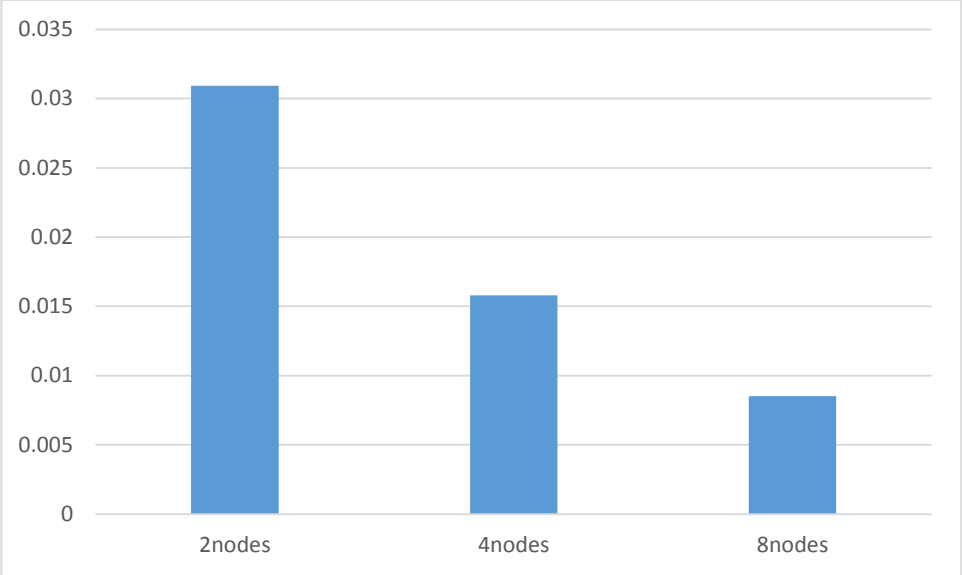
- The bucket size is equal to number of threads which is set in the SLRUM file. The value is equal to the number of tasks.
- The pivots are calculated in parallel using “pragma omp for”, which will span a thread for each iteration.
- The next part is sorting each bucket. We assign one thread for each bucket and sort the values in the bucket using merge sort.

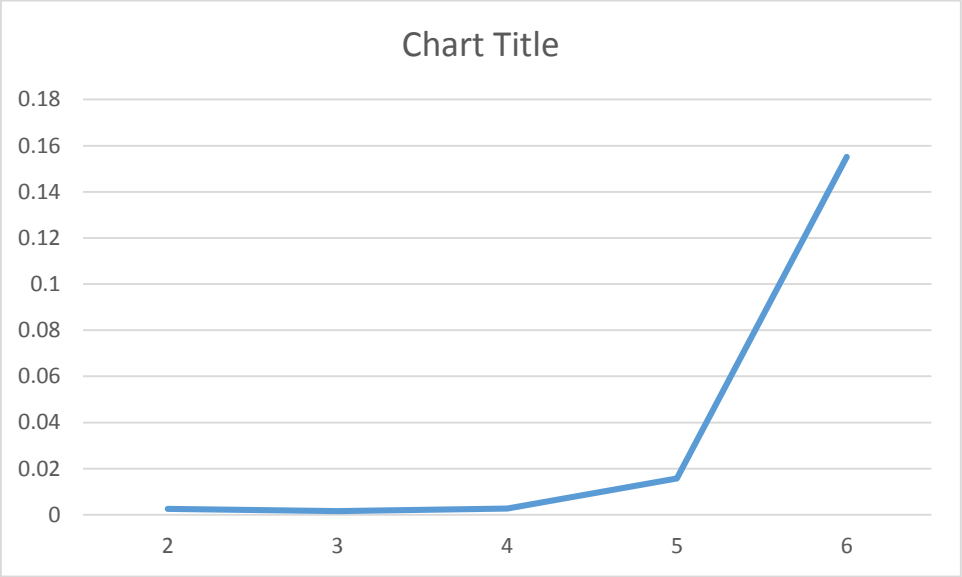
For comparison for the parallelism we run the algorithm for different number of threads in the order of 2, 4 and 8 and varying the problem size in the order of 10, from 1 to 1000,000. The results stored in the “Result_OpenMP_Size<size>_threads<numberofThreads>.out”. The comparison is mostly made on the grounds of computation time of the sorting of the given size.
A tabulation below consisting of the various problem size and number of threads, and the computation of the sorting is listed below. This tabulation is used for comparison and graphs are plotted.

10 power x	2 Thread	4 Thread	8 Thread
2	0.001039	0.002578	0.004732
3	0.000926	0.001594	0.010732
4	0.013287	0.002737	0.036058
5	0.030919	0.015791	0.008527
6	0.103694	0.155167	0.101885

From the graph we can clearly see that the time taken is inversely proportional to the number of threads. This condition is true for larger size, however when noticed for smaller problem size the execution time is longer than for increased number of threads. This is due to the fact that the problem takes more computation time for dividing problem the problem. That is, latency for multithreading, creation of threads and memory access by the threads. There by the processor spends more time in accessing the memory than computation.

The below plot is the graph between number of nodes and the computation time. As the number of nodes increases the computation time decreases.





The above the graph for varying the size of the problem and time. The computation time is more for larger size. But in some cases the computation with more nodes, the computation time is almost equal to the larger size. This is because of the fact the lag that happens in dividing the data amongst the processors.

MPI

The same algorithm is used for MPI as well, here we are take the size of bucket as the number of processors set in the variable “nodes=number of processor” in SLRUM file.

Each processor is given a range of values which is set of pivots and the values are sorted and printed. With the index of value in the array and its values. The output will contain the values sorted by each processor serialized locally but the result will contain datas in non-serialized way. But the index indicates the position.

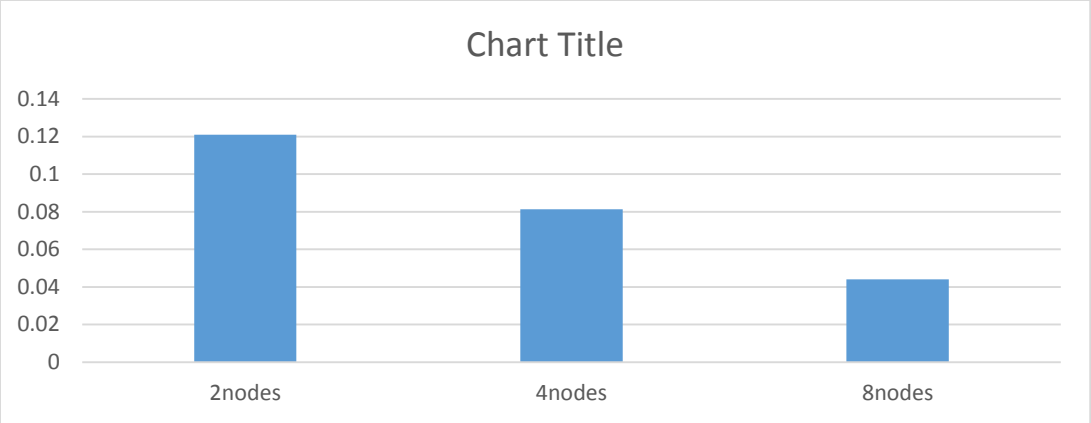
The processor are increased in order of 2, 4 and 8. However the number of threads should not have any effect in this. In MPI only we are varying the nodes and threads should not make any impact.

- We use MPI Bcast to broad cast from the process to other processor initially. The process with process id (p_id=0)will broadcast the values to the other processors and the processors using the value of the p_id finds the value of the pivots it must sort.
    - pivots_value[p_id] = quickSelect(array, 0, size - 1, (pivots[p_id]));
    - Therefore the process id decides the bucket the processor must take.
    - Once the pivot is found then the processor will sort the value in the range from pivot[p_id] to pivot[p_id]-1, using merge sort.
- The processor writes it output of the values sorted in its bucket to STDOUT.

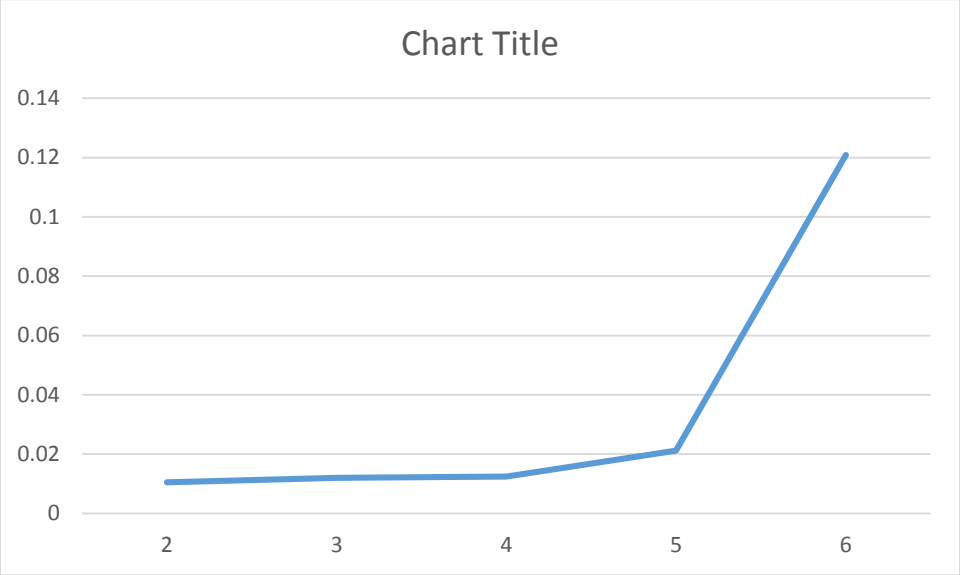
A tabulation below consisting of the various problem size and number of processor/nodes, and the computation of the sorting is listed below. This tabulation is used for comparison and graphs are plotted.

10 power x	Time in Secs 2buckets	Time in Secs 4buckets	Time in Secs 8buckets
2	0.01047	0.012182	0.060988
3	0.01197	0.015278	0.027824
4	0.012391	0.014721	0.025638
5	0.021129	0.044683	0.029407
6	0.120878	0.08126	0.044095

We can see that as the scaling of the problem increases the execution time also increases, but here since the problem is executed in parallel the execution time is reduced in the multiples of scaling factor. Which is clearly seen from the above graph. However for the smaller size we can see that the parallel takes more time, that is because of the latency that is created in dividing the problem is more than the execution of the problem.



The above plot is the graph between number of nodes and the computation time. As the number of nodes increases the computation time decreases. We can clearly see that the values are inversely proportional. The size is constant. Below the graph for varying the size of the problem and time. The computation time is more for larger size. But in some cases the computation with more nodes, the computation time is almost equal to the larger size. This is because of the fact the lag that happens in dividing the data amongst the processors.



Comparison between openmp and MPI.

Let us compare two approached that gave better scaling in execution in terms of the number of parallelism used.

10powerx	2 scaling Factor		4 scaling Factor		8 scaling Factor	
	MPI	OpenMP	MPI	OpenMP	MPI	OpenMP
2	0.01047	0.001039	0.012182	0.002578	0.060988	0.004732
3	0.01197	0.000926	0.015278	0.001594	0.027824	0.010732
4	0.012391	0.013287	0.014721	0.002737	0.025638	0.036058
5	0.021129	0.030919	0.044683	0.015791	0.029407	0.008527
6	0.120878	0.103694	0.08126	0.155167	0.044095	0.101885

Comparison is show below for the larger problem size and different scaling factor. From the comparison, MPI out performs openmp.

OpenMPI(openmp + MPI)

The scaling factors of open and mpi are combined, now the algorithm is made to rum on multithreads and multiprocessors to get much more scaling factor and parallelism.

The algorithm is same as the sequential but the size of bucket is equal to the number of the number of the nodes times the number of threads.

The following changes are made in OpenMP_MPI

- The problem is divided based on the number of processors, in the same it was done in MPI.
- In each processor we run the algorithm in the same way as we did in the OpenMP.
- We calculate the pivot values two times. Once during running MPI and next running OpenMP.
- This will allow us to have problem divided inside the processor for each threads.

  For example we have the 100 values as the size, 2 nodes and 4 threads each. Then we problem divided between two processors say p_id=0 and p_id=1. The pivot values will be 0,50,100.
  The processor p0 will sort the values between 0 to 49. And the processor p1 will sort the values between 50 to 99. Within p0, there will 4 threads and that will divide problem inside the processor into smaller buckets. So t0 will take values from 0 to 12 and t2 from 13 to 25, t3 from 26 to 37 and t4 from 38 to 49. The treads will sort the values in parallel.

Here are the results of the computation, the node of nodes are kept constant and threads are varied to see the execution time. We can see as the number of threads are increased the execution time is reduced to a greater extend. Thus providing multiple scales of parallelism. Here the number of nodes are set to 2.

Special case comparison.

The below table is the comparison of special cases. According to values, we can see that the MPI performs better comparatively. And also Open_MPI is has bad performance because of the heavy overhead in dividing the problem multiple times. First for the MPI and then for each threads in openMP.

Program	Number of cores	Problem size	Time in secs
OpenMP	1 node 2 task-per-node	100,000	0.030919
OpenMP	1 node 2 task-per-node	1,000,000	0.103694
OpenMP	1 node 8 task-per-node	1,000,000	0.101885
MPI	1 node 2 task-per-node	100,000	0.021129
MPI	1 node 2 task-per-node	1,000,000	0.120878
MPI	2 node 8 task-per-node	1,000,000	0.067477
OpenMP_MPI	1 node 2 task-per-node	100,000	0.019347
OpenMP_MPI	1 node 2 task-per-node	1,000,000	0.199851
OpenMP_MPI	2 node 8 task-per-node	1,000,000	0.224309

Running the program.

Before running the program, we set the problem parameters.

- For sequential run.
  - ./sequential <size> <bucketsize>
    - If size is 100 the <size> is 10p2, and for 1,000,000 the <size> is 10p6.
  - The slrum must have nodes=1 and task per node =1.
  - Set out file as Result_BSort_Sequential_size<size>_bsize<buckets>.out
    - If size is 100 the <size> is 10p2, and for 1,000,000 the <size> is 10p6.
  - Output will be saved at the local directory in the form of value in the output file set in the SLRUM file.
- For OpenMP, MPI and OpenMP_MPI
  - ./<type> <size>
    - Type is the how the program is paralleled. For example OpenMP we will have type as OpenMP.
    - If size is 100 the <size> is 10p2, and for 1,000,000 the <size> is 10p6.
  - Set out file as Result_BSort_<type>_size<size>_bsize<buckets>.out
    - If size is 100 the <size> is 10p2, and for 1,000,000 the <size> is 10p6.
  - Output will be saved at the local directory in the form of value in the output file set in the SLRUM file.
  - The values in the SLRUM file for nodes and task per node must be set before running.
- Make all will compile the file and create the object file of the type.
- Sbatch SLRUM_BSort_<type>.sh will create a job to execute with the parameters set in the SLRUM file.
- Once the result is obtained the output will be in the output file of the SLRUM.

Summary

From all the above results we have the following conclusion.

- As the size of the problem increases the execution time increases exponentially.
- As the number of thread increase with the constant problem size, the execution time decreases by the multiple factor of the thread used.
- As the number of cores/nodes/processor increases with the constant problem size, the execution time decreases in the factor of the number of nodes.
- By combining the scaling factors of Openmp and MPI, we get multiple scaling factor. Say if 4 core and 2 threads are used in the program execution, we get 8x parallelism. Thus reducing the execution time by 8 times the sequential execution for the given problem size.

Thus all the result obtained is similar to the theoretically and matches the scaling factor approximately.