CSE 708: Hypercube based Parallel Quicksort using MPI

Sai Vishwanath Venkatesh

University at Buffalo
Graduate School of Education

Outline

1. Overview of Traditional Quicksort Algorithm
2. About Hypercube
3. Overview of Hyperquicksort
4. Algorithm Complexity breakdown
5. Testing Considerations
6. Observations, Results and Discussion
    a. Runtime
    b. Speedup
    c. Efficiency
7. Problems identified and areas of improvement

Traditional Quicksort Algorithm

1. Chose a pivot element
2. Place all elements smaller than pivot to a smallList
3. Place all elements larger than pivot to a bigList
4. Do steps 1-3 recursively for smallList and bigList
   a. Quicksort(smallList)
   b. Quicksort(bigList)
5. Stitch smallList , pivot and bigList

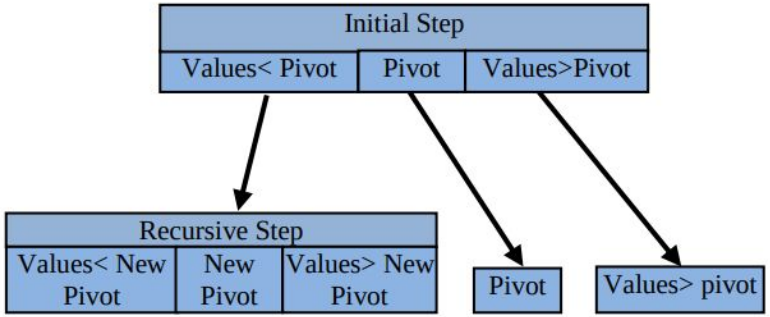


Fig 1: Simple graphical representation of the Sequential quick sort algorithm

Figure : Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithms - Rajput IS et al.

Traditional Quicksort Algorithm : Why not just do this??

- This works for a shared memory setting. Imagine data across several processing elements .
- Now the solution would need random movement of values across processing elements - involves expensive communication

  (OR)

  pooling all elements to one processor - which would flood one processor and not balance work for very large problems

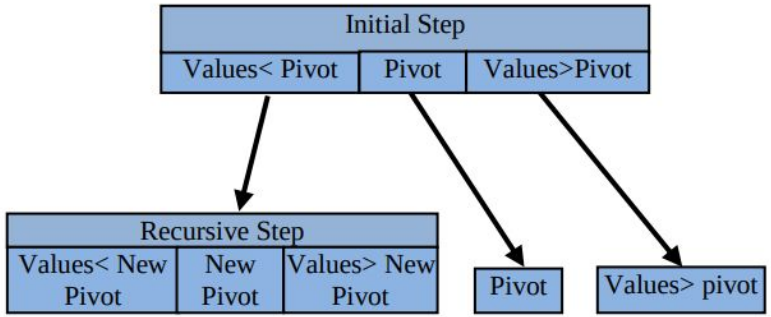


Fig 1: Simple graphical representation of the Sequential quick sort algorithm

Figure : Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithms - Rajput IS et al.

Hypercubes: What are they?

- a hypercube of size ***n*** consists of n processors indexed by the integers 0,1, … , $n - 1$, where n is an integral power of 2.
- Processors A and B are connected if and only if the unique $\log_2 n$-bit represent their indices differ in exactly 1 position.

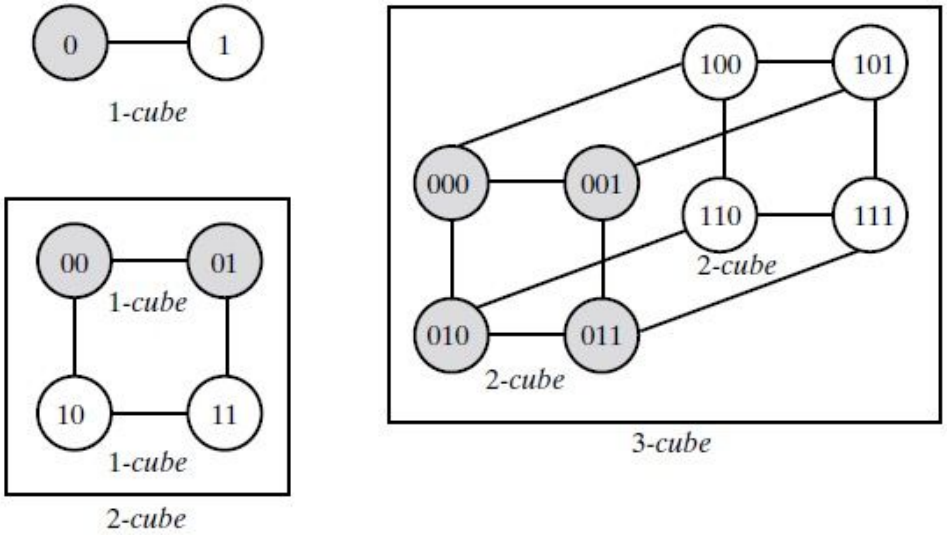


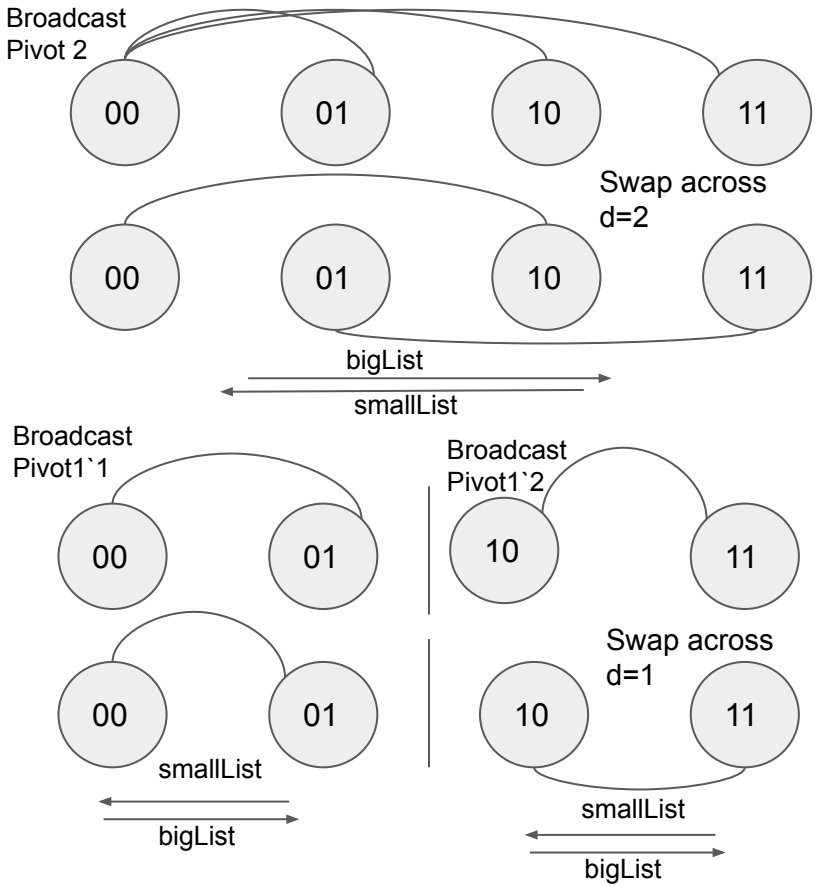
Figure : Algorithms, Sequential and Parallel: A Unified Approach – Russ Miller and Laurence Boxer. 3rd Edition.

Hyperquicksort

1. Assume n/p data across each of the 2^d processors
2. Sort data on each processor and broadcast median(pivot) from p0 to other processors in group
3. Locally on all processors use binary search to find pivot and make smallList ([0:pivot]) and bigList([pivot+1::])
4. Swap smallList and bigList across pair processors in subcube across degree

   Pair Processor = (di) XOR (rank)
5. Repeat 2-4 for next degree of subcube until you have exhausted hypercube dimensions
6. Sort individual processors (d=0)

Example for p = 2², Degree = 2 hypercube



Algorithm Theoretical Analysis

1. Sort Locally
2. Send Group
   a. Grouping - (Identify who would broadcast pivot) and MPI SendRecv
3. Make and send small and big lists
   a. Binary search received pivot (largest index after pivot)
   b. Split sorted list based on the above. Send List sizes to pair processors
   c. Send Lists

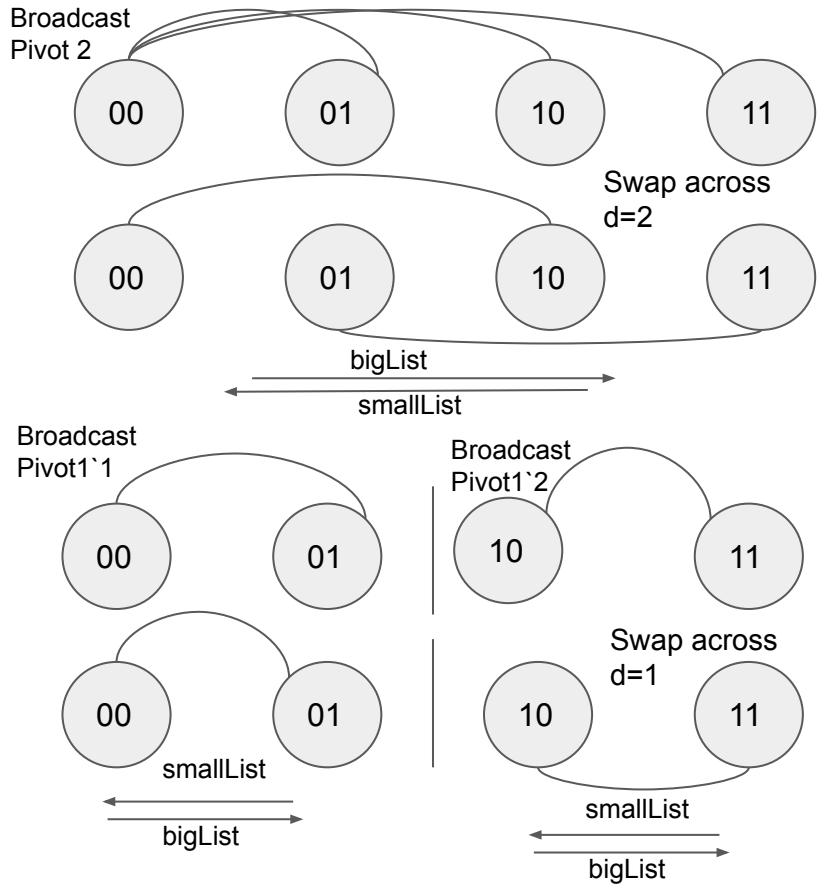
Steps 1-3 occur for d steps. (d is the tot.hypercube degree)

4. Sort Locally

Average Case time :

log (p)* [n/p log(n/p) + (Time for broadcast) + log(n/p) + (Time for data swap MPI SendRecv)]

Example for p = 2², Degree = 2 hypercube



Testing Considerations and SLURM specifications

- Number of Processing Elements

  = [1, 2, 4, 8, 16, 32, 64, 128, 256]

- Number of Nodes (corresponding to above)

  = [1, 1, 1, 1, 1, 2, 4, 8, 16]

- Hypercube dimensions (corresponding to above)

  = [0, 1, 2, 3, 4, 5, 6, 7, 8]

- Problem Sizes (in GB)

  = [1, 2, 4, 10, 20, 60]

- Partition

  General-Compute

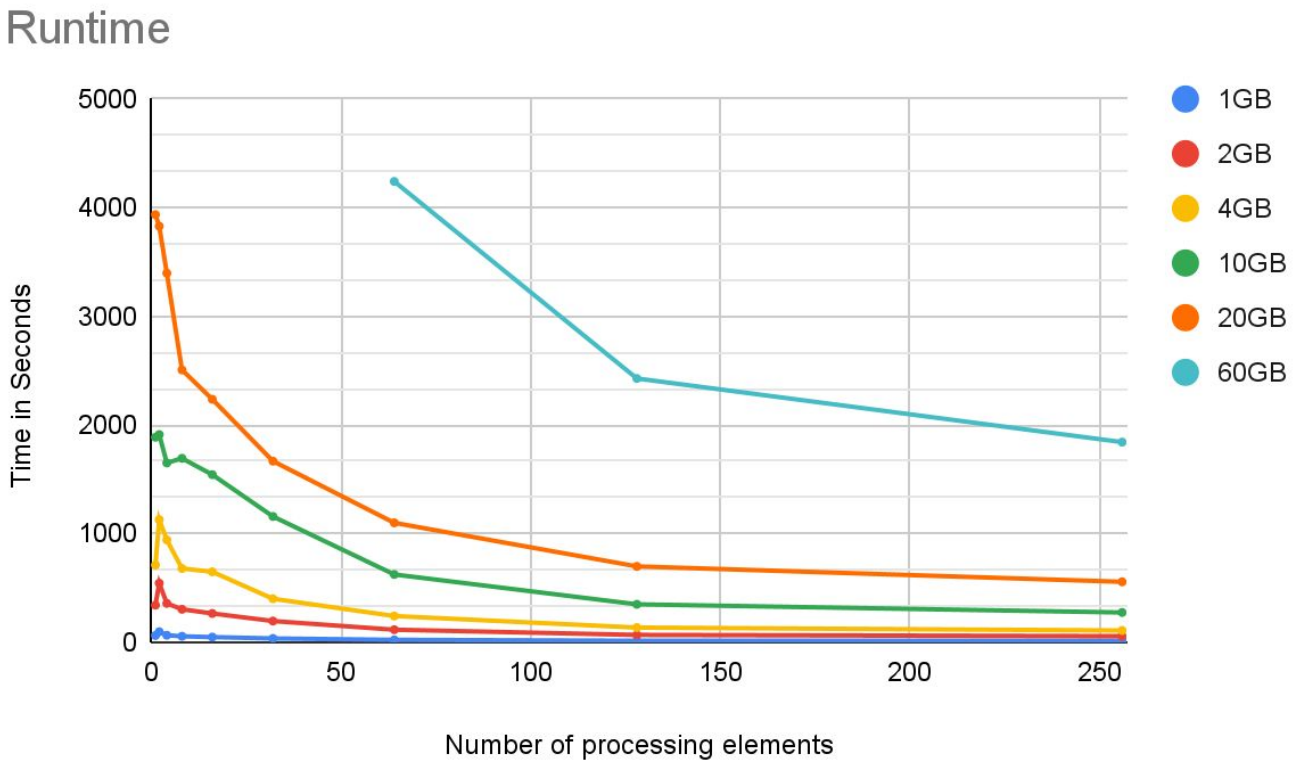
- Memory Allocation

  64 GB per node

Observations - Runtime

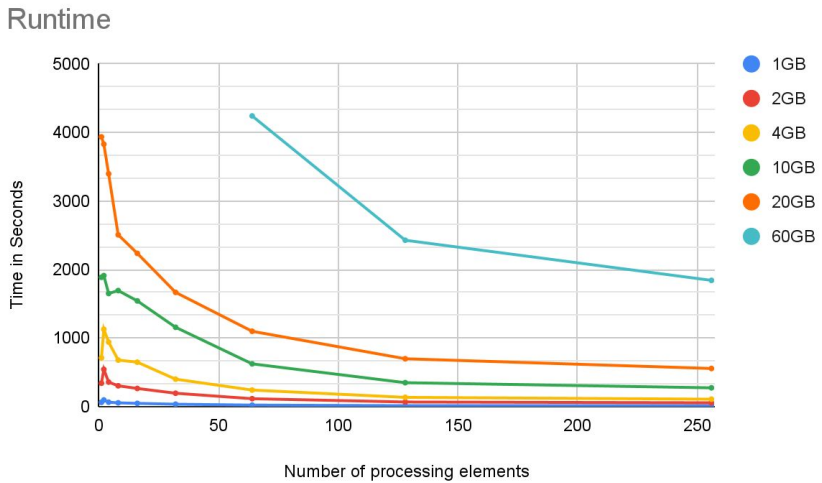
p\n	1 GB	2GB	4GB	10GB	20GB	60GB
1	62.7239	341.8013	710.964	1887.16	3935.94	SlurmTimeOut
2	97.3456	544.2685	1129.17	1911.79	3829.21	SegFault
4	65.1068	358.4353	942.285	1650.88	3397.395	SegFault
8	55.3394	303.9763	679.479	1693.35	2507.14	SegFault
16	47.8395	265.3903	648.714	1543.02	2236.65	SegFault
32	35.5152	195.7346	400.773	1158.03	1667.355	SegFault
64	22.6964	115.9353	241.771	624.37	1099.485	4241.36
128	14.2269	68.0167	135.585	349.58	698.759	2428
256	14.4909	56.1877	108.853	274.6	556.83	1842.5

Observations - Runtime



Observations and Discussion - Runtime

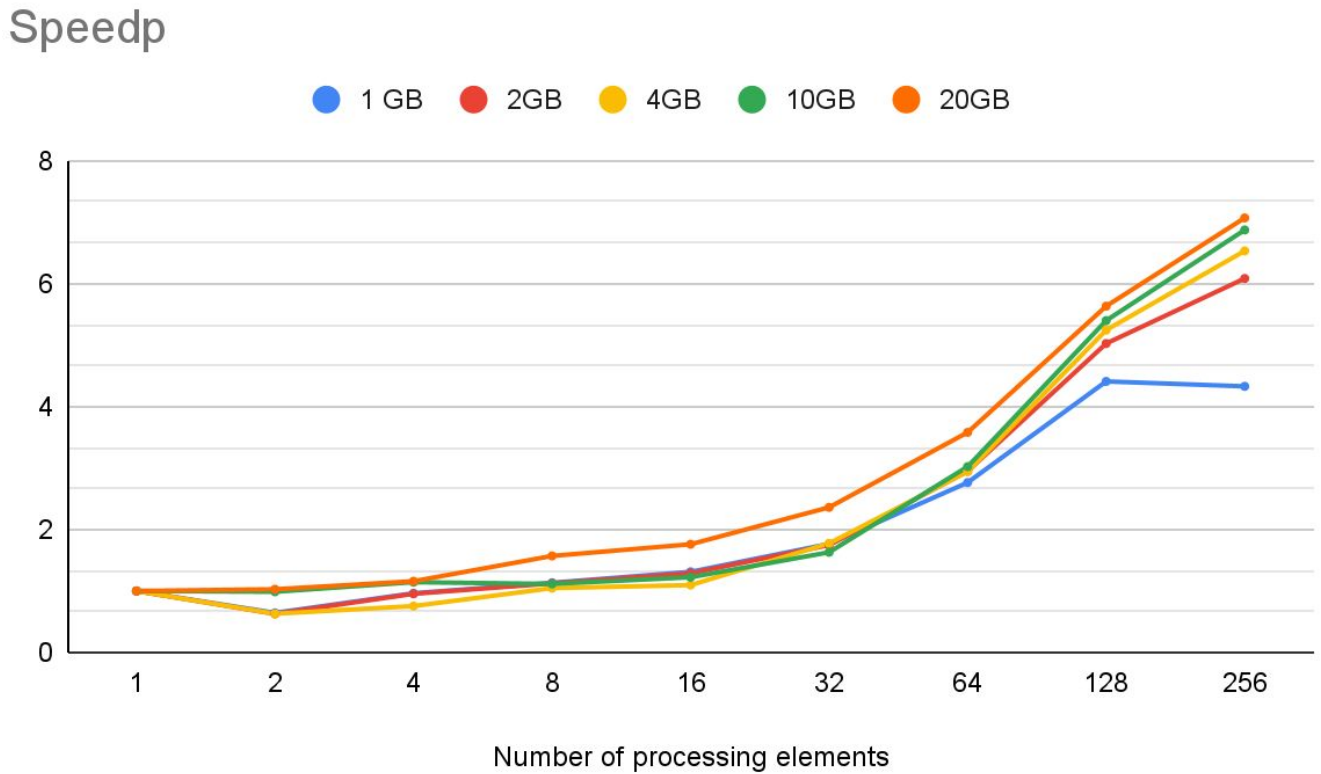
- Runtime reduction is minimal for small problems, larger as the problem size increases. Arguably displays good reduction between 2-64 PEs for the large problems.
- Cases where it has even spiked (from 2-4) compared to sequential solution. (Since communication takes longer than sorting for smaller problems)
- 1-32 PEs fails with Segmentation Fault for 60GB problem.



Observations - Speedup

p\n	1 GB	2GB	4GB	10GB	20GB	60GB
1	1	1	1	1	1	-
2	0.6443424253	0.6280012531	0.6296341561	0.9871167858	1.02787259	-
4	0.9634001364	0.9535927404	0.754510578	1.143123667	1.15851704	-
8	1.133440189	1.124434043	1.046336973	1.114453598	1.569892387	-
16	1.311132014	1.28791934	1.095959082	1.223030162	1.759747837	-
32	1.766114227	1.746248747	1.77398178	1.62962963	2.360589077	-
64	2.763605682	2.948207319	2.94065045	3.022502683	3.579803272	1
128	4.408824129	5.025255562	5.243677398	5.398363751	5.632757503	1.746853377
256	4.328502715	6.083205043	6.531413925	6.872396213	7.068476914	2.301959294

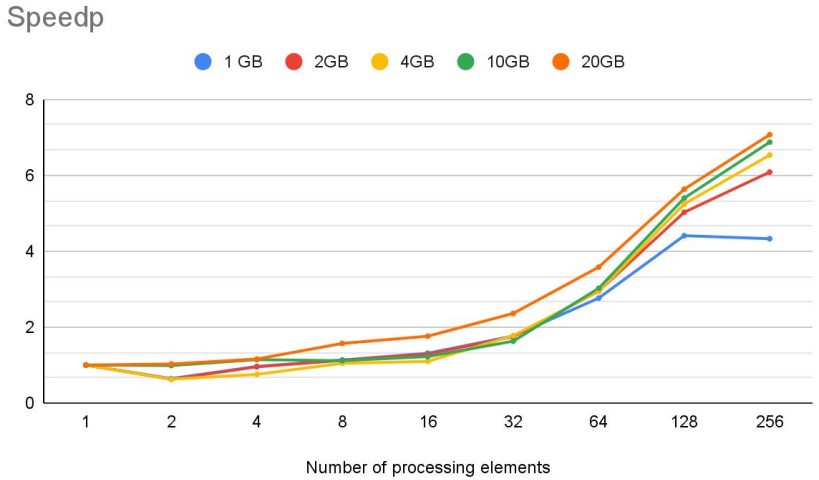
Observations - Speedup



Observations and Discussion - Speedup

- In theory the ideal speedup would be **p * [(logn)/ log(n/p)]** which you could approximate to **p**.
- We hardly even get half of the mentioned ideal speedup (1-4 PEs is the closest to ideal)
- Also the idea we got from runtime analysis that “the algorithm was improving for larger problems” is not true as the speed up is not too different with problem sizes.

  This can also be attributed to the increase in communication with more PEs

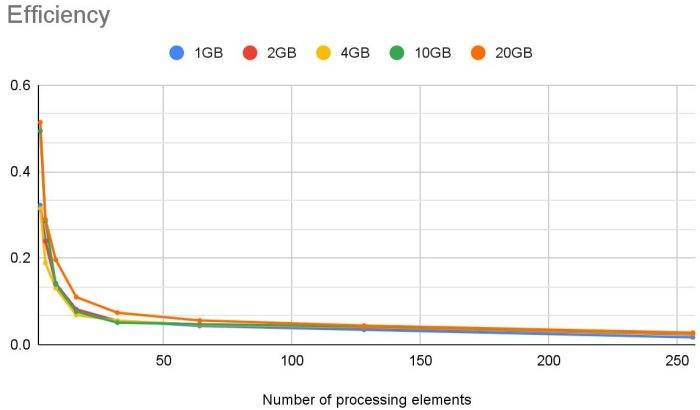


Observations - Efficiency

p\n	1 GB	2GB	4GB	10GB	20GB
1	1	1	1	1	1
2	0.3221712127	0.3140006265	0.314817078	0.4935583929	0.513936295
4	0.2408500341	0.2383981851	0.1886276445	0.2857809168	0.2896292601
8	0.1416800236	0.1405542554	0.1307921216	0.1393066997	0.1962365484
16	0.0819457509	0.08049495875	0.06849744263	0.0764393851	0.1099842398
32	0.0551910696	0.05457027334	0.05543693063	0.05092592593	0.07376840865
64	0.04318133878	0.04606573936	0.04594766328	0.04722660442	0.05593442612
128	0.03444393851	0.03925980908	0.04096622967	0.0421747168	0.04400591799
256	0.01690821373	0.0237625197	0.02551333565	0.02684529771	0.02761123795

Discussion - Efficiency

- This efficiency is very bad and cascades from our discussion in speedup. The best performance can witnessed between 1 - 4 processors for the larger 20GB problem with a megre 51%.
- The drastic drop in efficiency with the increase in processing elements and nodes suggests that my algorithm could be breaking due to high communication overheads.
- Also there is a chance for the data partitioning to be bad and cause the lack of PE use. (This could have also caused the terrible efficiency)



Discussion - Problems and Areas identified for improvement

Problems in hyperquicksort algorithm

- Arriving at a bad pivot can cause uneven data partitioning to result in a less use of some PEs - since this is followed by a sequential sort step. (thus low efficiency)
- High communication overhead in data swap step.
- Can only work with processors in the powers of 2 - Which means the num. processors increase exponentially if we need more PEs and therefore the communication overhead also increases in an exponential scale.

Major Coding optimisations to do to better the speedup and efficiency

- Heavy list merge: Can use *std::vector.reserve()* before *std::vector.insert()* to avoid dynamic resizing.
- Can use MPI profiling tools to better identify where the communication overhead is most.

References

1. Rajput, I.S., Kumar, B. and Singh, T., 2012. Performance comparison of sequential quick sort and parallel quick sort algorithms. International Journal of Computer Applications, 57(9).
2. Algorithms, Sequential and Parallel: A Unified Approach – Russ Miller and Laurence Boxer.
3. https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_sorting.htm

University at Buffalo
Graduate School of Education

Questions or Comments or Suggestions?

saivishw@buffalo.edu