



Peyman Shobeiri & Victor Cruz

IDs: 40327586 & 40136204

Fall 2024




Table of Contents

QUESTION 1	3
QUESTION 2	7

Question 1

Parallel Quicksort on a hypercube topology: One such algorithm for a d-dimensional hypercube is discussed in problem 9.17 of the textbook (page 421). Algorithm 9.9 and Figure 9.21(pages 422 and 423) in the book further elaborate it. The algorithm is based on recursive halving. In this assignment, you will implement this specific quicksort algorithm on the cluster. For doing so, you will assume a virtual hypercube topology on the cluster nodes.

Compare its performance with the best sequential sorting algorithm (e.g., quicksort) to sort a large input sequence (you can generate the numbers randomly, however note that TA may provide the input for demo). As in assignment 1, you should plot speed-up versus number of processes graphs for different input data sizes and explain your findings.

The parallel quicksort algorithm is designed to work efficiently with a number of processes that align with hypercube structures, such as 1, 2, 4, 8, etc. Ideally, the total number of elements should be divisible by the number of processes to optimize performance. This parallel version uses the traditional quicksort as its base sorting method.

The parallel version of the quick sort algorithm is implemented in `qsp_null.cpp` file, which can be compiled with the following command:

```
mpicxx -std=c++17 qsp_null.cpp -o qsp_null.o
```

To run it, you can use this command:

```
mpirun -np 8 --hostfile /media/pkg/apini-scripts/apini_hostfile qsp_null.o
```

Where in this code the 8 specifies the number of processes. The elements are randomly generated using the `gen-rand_input.py` file which be used by the C++ code to run the parallel quicksort algorithm. So, the numbers are generated randomly and are distributed among processes using MPI scatter.

The sequential quicksort algorithm can be compiled with:

```
mpicc quicksort-seq.c -o quicksort-seq.o
```

and can be executed by running:

```
mpirun -np 1 -hostfile /media/pkg/apini-scripts/apini_hostfile quicksort-seq.o
```

Since pivot selection can impact sorting efficiency (poor pivot choices lead to worse runtimes), speedup was calculated based on an average of the execution times for both parallel and sequential versions.

The speedup is given by the formula:

$$\text{Speedup} = \frac{\text{Sequential_Time}}{\text{Paralle_Time}}$$

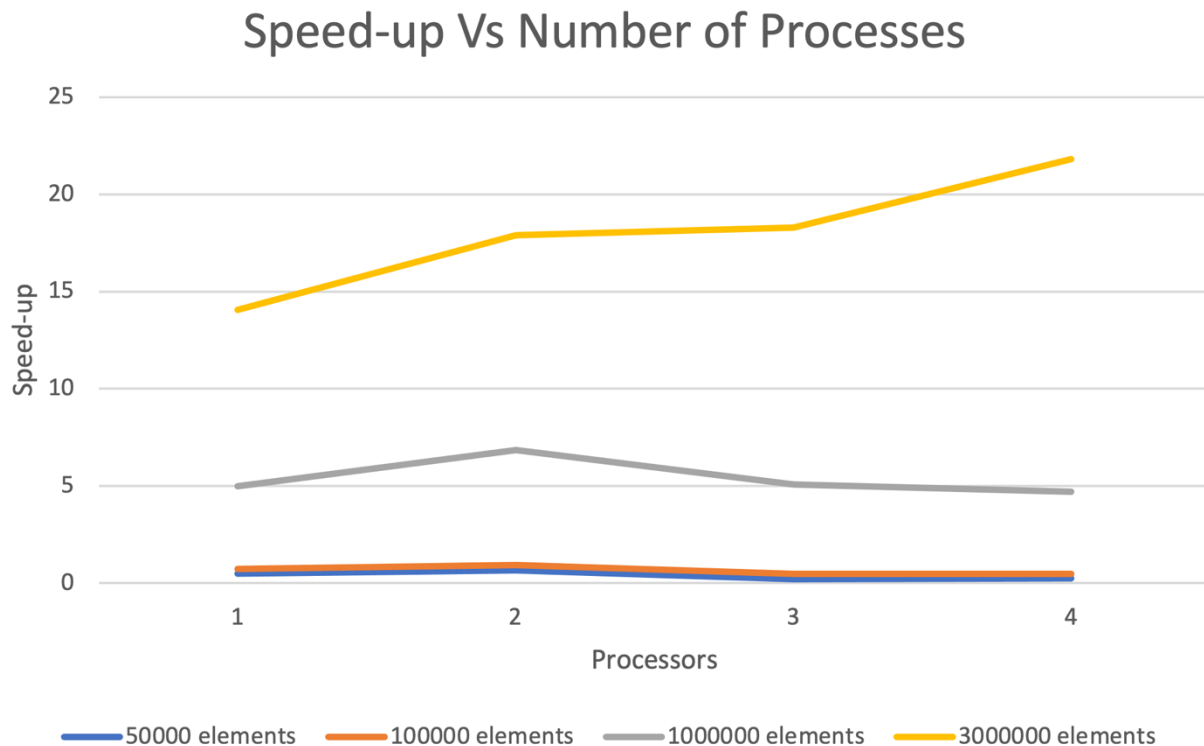
Number of elements in array	Sequential Time (sec)		Number of Processors	Parallel Time (sec)	Average Parallel Time (sec)	Speedup
	Sequential Time	Average				
50000	0.012	0.011	1	0.022	0.022	0.5
				0.023		
				0.022		
	0.012		2	0.017	0.017	0.65
				0.017		
				0.017		
	0.01		4	0.054	0.051	0.22
				0.044		
				0.054		
	0.01		8	0.046	0.045	0.24
				0.045		
				0.043		

Number of elements in array	Sequential Time (sec)		Number of Processors	Parallel Time (sec)	Average Parallel Time (sec)	Speedup	
	Sequential Time	Average					
100000	0.033	0.033	1	0.047	0.046	0.72	
				0.045			
				0.046			
	0.035		2	0.035	0.036	0.92	
				0.034			
				0.04			
	0.035		4	0.072	0.07	0.47	
				0.069			

	0.032			0.068	0.07	0.47
			8	0.065		
				0.062		
				0.07		

Number of elements in array	Sequential Time (sec)		Number of Processors	Parallel Time (sec)	Average Parallel Time (sec)	Speedup
	Sequential Time	Average				
1000000	2.54	2.54	1	0.510	0.51	4.98
				0.511		
				0.513		
	2.51		2	0.39	0.37	6.86
				0.36		
				0.36		
	2.56		4	0.5	0.5	5.08
				0.5		
				0.5		
			8	0.76	0.54	4.7
				0.43		
				0.42		

Number of elements in array	Sequential Time (sec)		Number of Processors	Parallel Time (sec)	Average Parallel Time (sec)	Speedup
	Sequential Time	Average				
3000000	22.87	22.9	1	1.61	1.63	14.05
				1.61		
				1.66		
	23		2	1.28	1.28	17.89
				1.3		
				1.27		
	22.84		4	1.26	1.253	18.28
				1.25		
				1.25		
	22.84		8	1.08	1.05	21.8
				1.07		
				1.01		



Based on the experimental results, we can analyze performance across different processor counts and input sizes with Amdahl's Law (since the problem size is fixed), which defines the maximum speedup achievable through task parallelization. According to this law, the speedup is limited by the sequential portion of the algorithm—tasks that cannot be parallelized, such as data setup and I/O operations. Regardless of how many processors are added, this sequential portion restricts the maximum achievable speedup. Even with an infinite number of processors, maximum speedup is limited to $1/f$ where f represents the sequential part of the algorithm. This explains why the speedup decreases after a certain number of processes. This effect is evident in our results: for smaller input sizes, adding processors initially lowers execution time, but the improvement eventually levels off or even decreases beyond a certain number of processors.

Additionally, the varying results across input sizes highlight the phenomenon of parallel slowdown. As we increase the number of processors, the added communication overhead among them can outweigh the benefits of additional computing power, particularly with smaller datasets. Increased inter-processor communication, necessary for data synchronization, can become a bottleneck. For smaller data sizes, this overhead is more noticeable, leading to diminishing returns as more processors are added. The trend observed across input sizes indicates that larger datasets derive greater benefits from parallelization, as the increased processing power effectively counters the communication overhead.

Question 2

Parallel Sorting using Regular Sampling (PSRS): This is a parallel version of quick sort which is suitable for MIMD machines [1]. The algorithm works in the following steps:

Step 1: Input data to be sorted of size N is initially portioned among the P processes so that each process gets N/P items to sort. Each process initially sorts its own sub-list of N/P items using sequential quick sort.

Step 2: Each process P_i selects P items (we call them regular samples) from its local sorted sub-list at the following local indices: $0, N/P2, 2N/P2, \dots, (P-1)N/P2$, and sends them to process P_0 .

Step 3: Process P_0 collects the regular samples from the P processes (which includes itself). So, it has P^2 total regular samples. It sorts the regular samples using quick sort, and then chooses $(P-1)$ pivot values at the following indices: $P + P/2 - 1, 2P + P/2 - 1, \dots, (P-1)P + P/2 - 1$, and broadcasts the pivots to the P processes.

Step 4: Each process P_i , upon receiving the $P-1$ pivots from process P_0 , partitions its local sorted sub-list into P partitions, with the $P-1$ pivots as separators. Then it keeps i th partition for itself and sends j th partition to process P_j .

Step 5: At the end of step 4, each process P_i has $(P-1)$ partitions from other processes together with its own i th partition. It locally merges all P (sorted) partitions to create its final sorted sub-list.

Compare its performance with the best sequential sorting algorithm, i.e., quick sort, for sorting a large input sequence (you can generate the numbers randomly). As in the previous question, you should plot speed-up versus number of processes graphs for different input data sizes.

The PSRS algorithm can be found in the PSRS.c file. To run the algorithm, first compile it with the command:

```
mpicc PSRS.c -o psrs.o
```

Then, execute it with:

```
mpirun -np 8 psrs.o
```

where 8 specifies the number of processors (this value can be adjusted as needed). You can set the desired number of elements directly in the code. The code includes a function that generates a random array of the specified size for testing purposes.

50k				
	1	2	3	avg
1	0.170611	0.142432	0.136813	0.14995
2	0.036418	0.039415	0.036643	0.03749
3	0.042474	0.042379	0.043778	0.04288
4	0.027656	0.036854	0.02728	0.03060
5	0.038246	0.037168	0.037105	0.03751
6	0.041253	0.037659	0.042265	0.04039
7	0.057762	0.054265	0.057531	0.05652
8	0.031909	0.038925	0.030916	0.03392
9	0.039439	0.048282	0.062236	0.04999
10	0.046705	0.050427	0.048039	0.04839
	base		speedup	
			0.14995	3.99957324
			0.14995	3.4972596
			0.14995	4.90092603
			0.14995	3.99804477
			0.14995	3.71238766
			0.14995	2.65310985
			0.14995	4.42118919
			0.14995	2.99989997
			0.14995	3.09880072

100k				
	1	2	3	avg
1	0.282086	0.556711	0.275956	0.37158
2	0.195573	0.149669	0.078042	0.14109
3	0.310936	0.12099	0.065041	0.16566
4	0.160321	0.08912	0.054164	0.10120
5	0.048255	0.157302	0.049038	0.08487
6	0.053462	0.046287	0.056012	0.05192
7	0.065433	0.058621	0.104798	0.07628
8	0.091123	0.034427	0.06482	0.06346
9	0.146357	0.155901	0.056374	0.11954
10	0.227171	0.228206	0.18902	0.21480
	base		speedup	
			0.37158	2.63358171
			0.37158	2.24311272
			0.37158	3.67172148
			0.37158	4.37853454
			0.37158	7.15681718
			0.37158	4.87106514
			0.37158	5.85571781
			0.37158	3.10834783
			0.37158	1.7299165

	1000000			
	1	2	3	avg
1	4.741423	4.741732	4.727574	4.73691
2	1.509711	1.548359	1.506902	1.52166
3	0.841083	0.848822	0.840182	0.84336
4	0.565169	0.562098	0.58117	0.56948
5	0.436873	0.430855	0.438955	0.43556
6	0.363168	0.369253	0.359672	0.36403
7	0.336045	0.324855	0.335721	0.33221
8	0.250519	0.259139	0.259671	0.25644
9	0.454903	0.462979	0.465855	0.46125
10	0.230971	0.23675	0.269315	0.24568
		base	speedup	
			4.73691	3.11299368
			4.73691	5.61669579
			4.73691	8.31797075
			4.73691	10.875422
			4.73691	13.0123799
			4.73691	14.2589099
			4.73691	18.4715889
			4.73691	10.2698193
			4.73691	19.2809157

	3000000			
	1	2	3	avg
1	29.562492	27.517675	27.693781	28.25798
2	9.553772	9.55449	9.544741	9.55100
3	7.231967	8.693293	5.480557	7.13527
4	3.733057	3.74507	3.72783	3.73532
5	2.808585	2.873071	2.880438	2.85403
6	2.308527	2.31496	2.29508	2.30619
7	1.965515	1.988737	1.941239	1.96516
8	1.656542	1.670679	1.689071	1.67210
9	1.482511	1.495695	1.467049	1.48175
10	1.346315	1.363418	1.32536	1.34503
		base	speedup	
			28.25798	2.95864095
			28.25798	3.96032293
			28.25798	7.56507882
			28.25798	9.90107654
			28.25798	12.2531079
			28.25798	14.3794551
			28.25798	16.8997235
			28.25798	19.0706603
			28.25798	21.0091683

processors	50k	100k	1M	3M
2	3.99957324	2.63358171	0	2.95864095
3	3.4972596	2.24311272	0	3.96032293
4	4.90092603	3.67172148	0	7.56507882
5	3.99804477	4.37853454	0	9.90107654
6	3.71238766	7.15681718	0	12.2531079
7	2.65310985	4.87106514	0	14.3794551
8	4.42118919	5.85571781	0	16.8997235
9	2.99989997	3.10834783	0	19.0706603
10	3.09880072	1.7299165	0	21.0091683

