Implementation of Parallel Quick Sort using MPI

CSE 633: Parallel Algorithms
Dr. Russ Miller

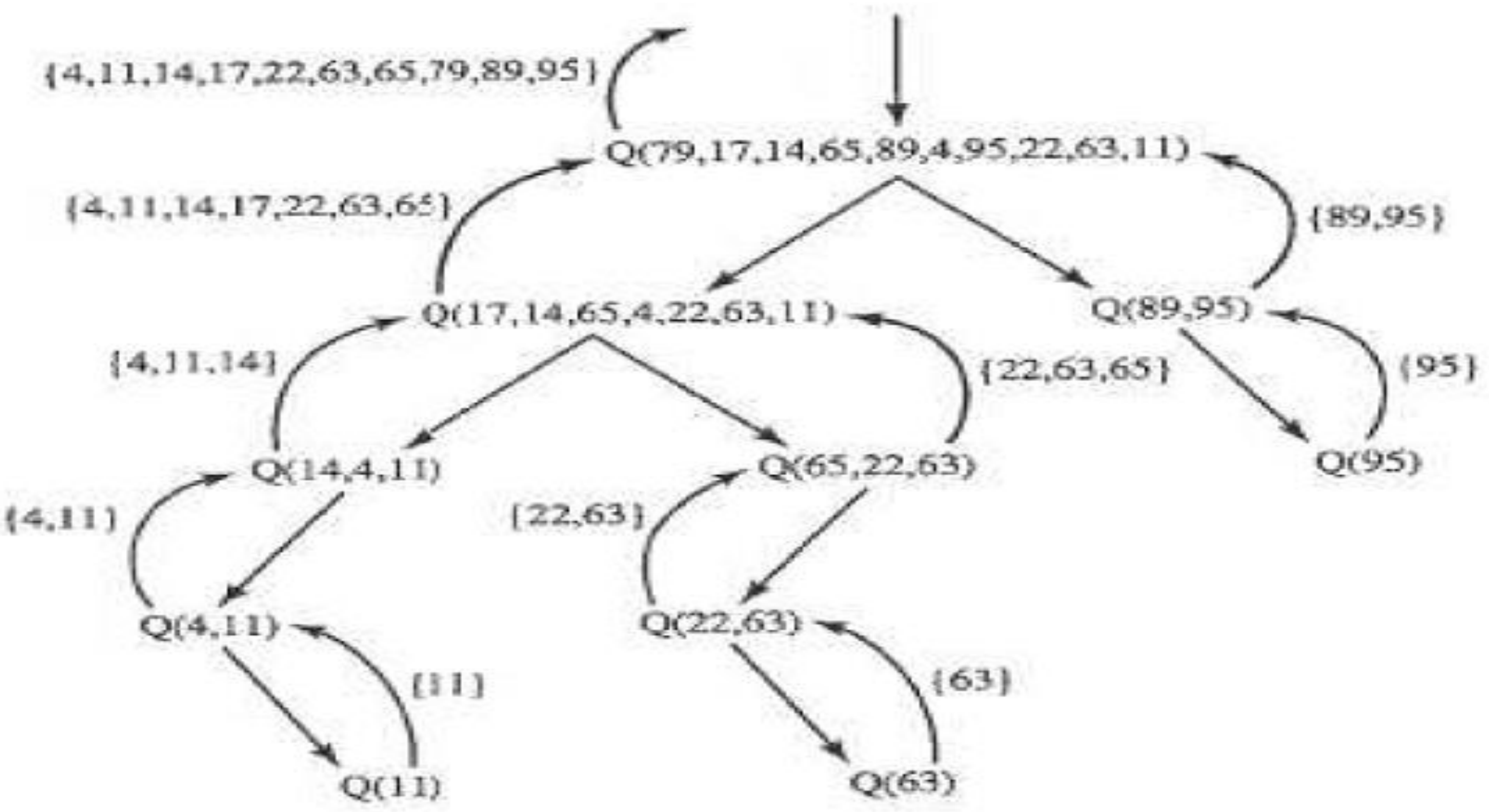
Deepak Ravishankar Ramkumar
50097970

Recap of Quick Sort

- Given a list of numbers, we want to sort the numbers in increasing or decreasing order.
- On a single processor the unsorted list is in its primary memory, and at the end the same processor would contain the sorted list in its primary memory.
- Quicksort is generally recognized as the fastest sorting algorithm based on comparison of keys, in the average case.
- Quicksort has some natural concurrency.

Sequential quicksort algorithm:

- Select one of the numbers as pivot element.
- Divide the list into two sub lists: a “low list and a “high list”
- The low list and high list recursively repeat the procedure to sort themselves.
- The final sorted result is the concatenation of the sorted low list, the pivot, and the sorted high list


{4,11,14,17,22,63,65,79,89,95}
Q(79,17,14,65,89,4,95,22,63,11)
{4,11,14,17,22,63,65}
{89,95}
Q(17,14,65,4,22,63,11)
Q(89,95)
{4,11,14}
{22,63,65}
{95}
Q(14,4,11)
Q(65,22,63)
Q(95)
{4,11}
{22,63}
Q(4,11)
Q(22,63)
{11}
{63}
Q(11)
Q(63)

Algorithm for Parallel Quick Sort

- Start off assuming that the number of processors are a power of two.
- At the completion of the algorithm,
- (I) the list stored in every processor's memory is sorted, and
- (2) the value of the last element on processor *Pi* is less than or equal to the value of the first element on Pi+1.

Parallel quicksort algorithm

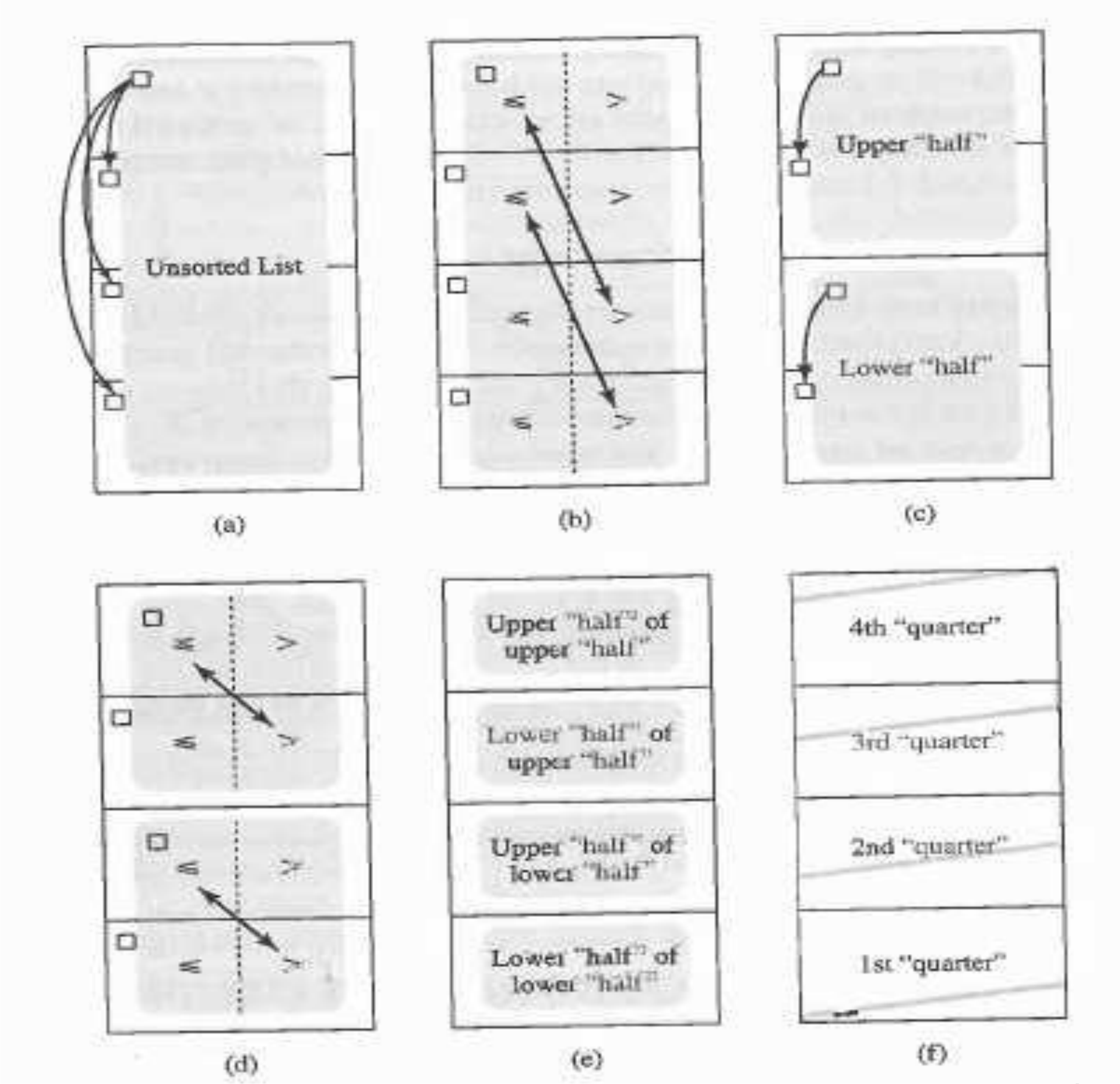
- Randomly choose a pivot from one of the processes and broadcast it to every process.
- Each process divides its unsorted list into two lists: those smaller than (or equal) the pivot, those greater than the pivot
- Each process in the upper half of the process list sends its “low list” to a partner process in the lower half of the process list and receives a “high list” in return.
- The processes divide themselves into two groups and the algorithm is recursive.

- Now, the upper-half processes have only values greater than the pivot, and the lower-half processes have only values smaller than the pivot.

- After log P recursions, every process has an unsorted list of values completely disjoint from the values held by the other processes.

- The largest value on process i will be smaller than the smallest value held by process i + 1.

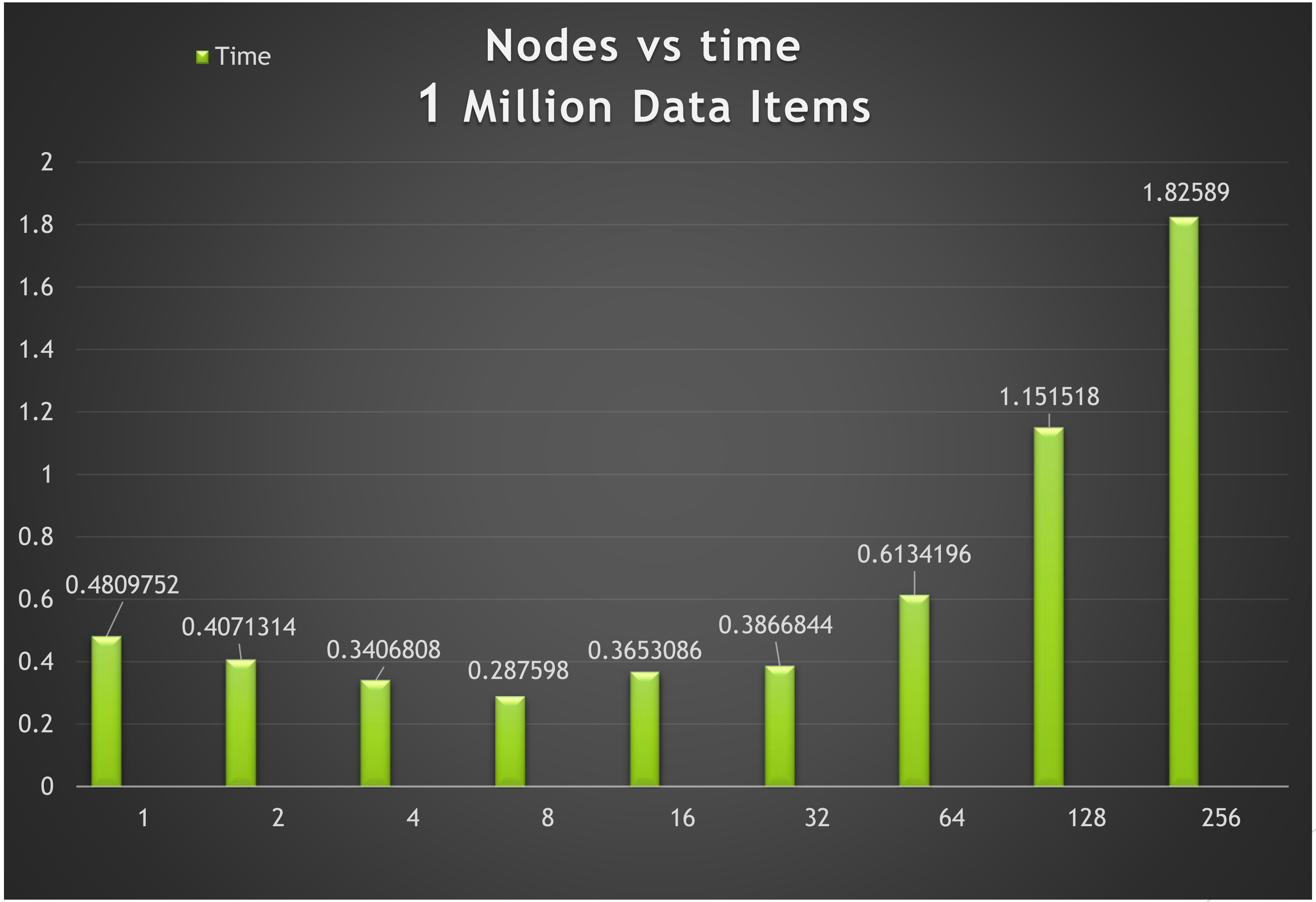
- Each process now can sort its list using sequential quicksort.



Unsorted List
≤
>
Upper "half"
Lower "half"
Upper "half" of upper "half"
Lower "half" of upper "half"
Upper "half" of lower "half"
Lower "half" of lower "half"
4th "quarter"
3rd "quarter"
2nd "quarter"
1st "quarter"
(a)
(b)
(c)
(d)
(e)
(f)

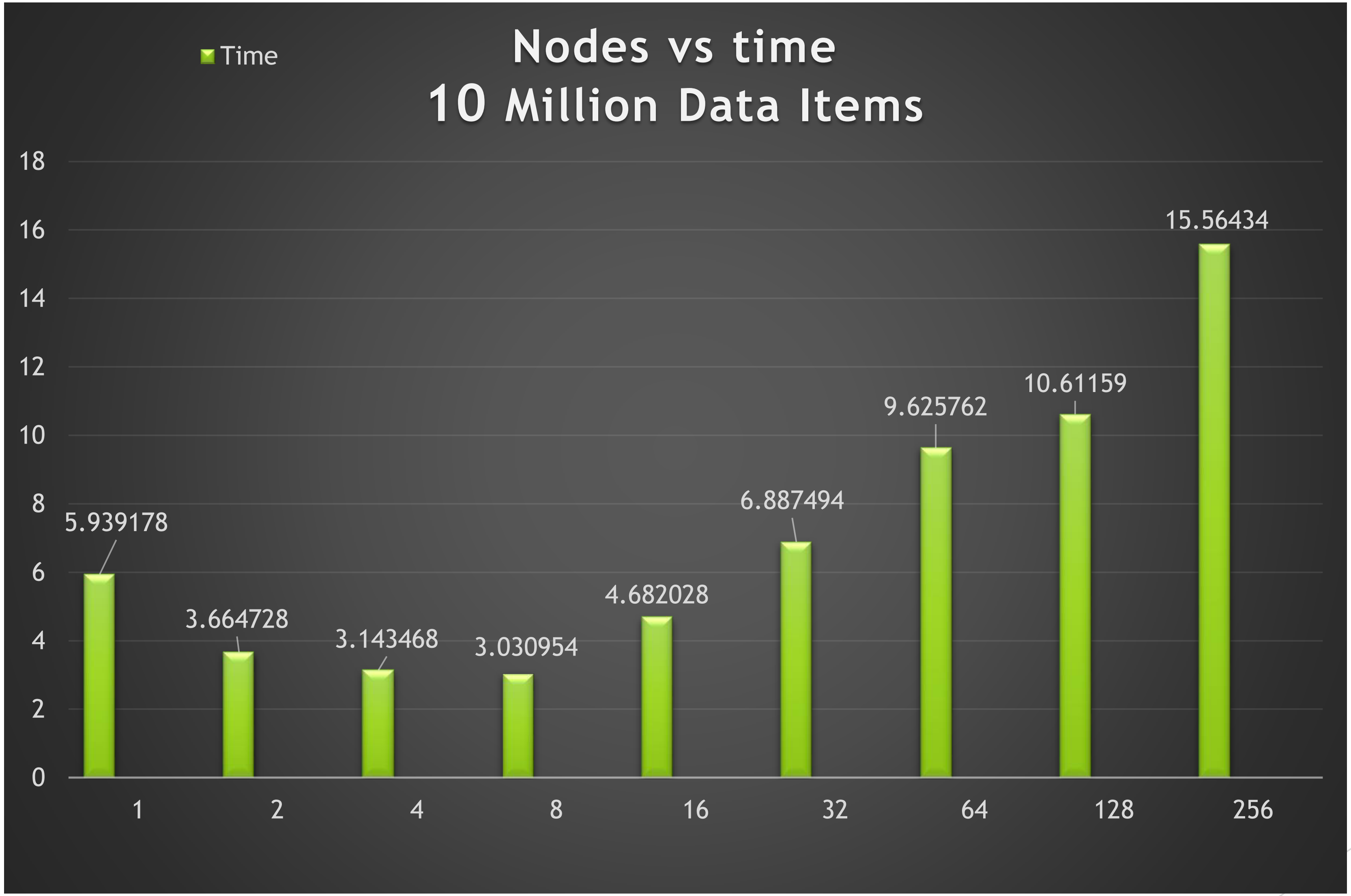
Readings for 1 Million Numbers

Num of Processors	Reading 1	Reading 2	Reading 3	Reading 4	Reading 5	Average
1	0.486137	0.479304	0.498915	0.459345	0.481175	0.48098
2	0.405326	0.406831	0.407008	0.412961	0.403531	0.40713
4	0.337033	0.357055	0.333783	0.342317	0.333216	0.34068
8	0.291766	0.280861	0.296402	0.281548	0.287413	0.2876
16	0.36568	0.374642	0.362062	0.365493	0.358666	0.36531
32	0.649314	0.298465	0.435157	0.272094	0.278392	0.38668
64	0.621614	0.578988	0.612795	0.633433	0.620268	0.61342
128	1.09468	1.24556	1.1024	1.0973	1.21765	1.15152
256	1.96445	1.87924	1.79648	1.77215	1.71713	1.82589


Time
Nodes vs time
1 Million Data Items
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0.4809752
0.4071314
0.3406808
0.287598
0.3653086
0.3866844
0.6134196
1.151518
1.82589
1
2
4
8
16
32
64
128
256

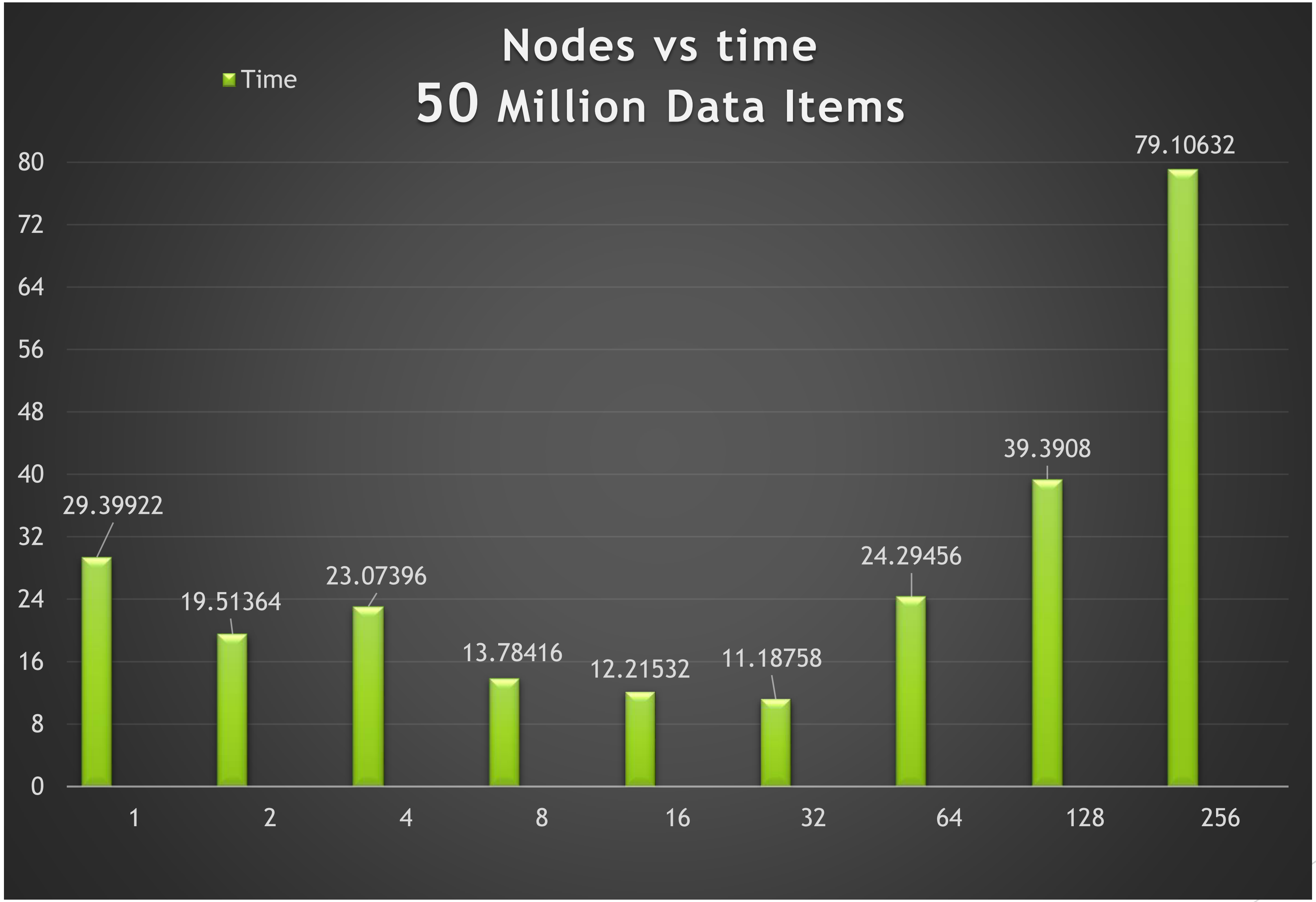
Readings for 10 Million Numbers

	A	B	C	D	E	F	G
1	Num of Processors	Reading 1	Reading 2	Reading 3	Reading 4	Reading 5	Average
2	1	5.63154	7.55708	5.59229	5.2965	5.61848	5.93918
3	2	3.63402	3.65614	3.60379	3.7325	3.69719	3.66473
4	4	3.08463	3.08168	3.19005	3.17345	3.18753	3.14347
5	8	3.23096	2.84125	2.91948	3.2931	2.86998	3.03095
6	16	2.79442	4.19635	4.16732	7.66447	4.58758	4.68203
7	32	4.09503	7.22516	6.9171	8.04961	8.15057	6.88749
8	64	10.9961	5.66727	15.8492	5.90791	9.70833	9.62576
9	128	9.76145	10.1217	10.9627	9.4038	12.8083	10.6116
10	256	15.4053	15.7564	15.4962	15.616	15.5478	15.5643
11							


Time
Nodes vs time
10 Million Data Items
18
16
14
12
10
8
6
4
2
0
5.939178
3.664728
3.143468
3.030954
4.682028
6.887494
9.625762
10.61159
15.56434
1
2
4
8
16
32
64
128
256

Readings for 50 Million Numbers

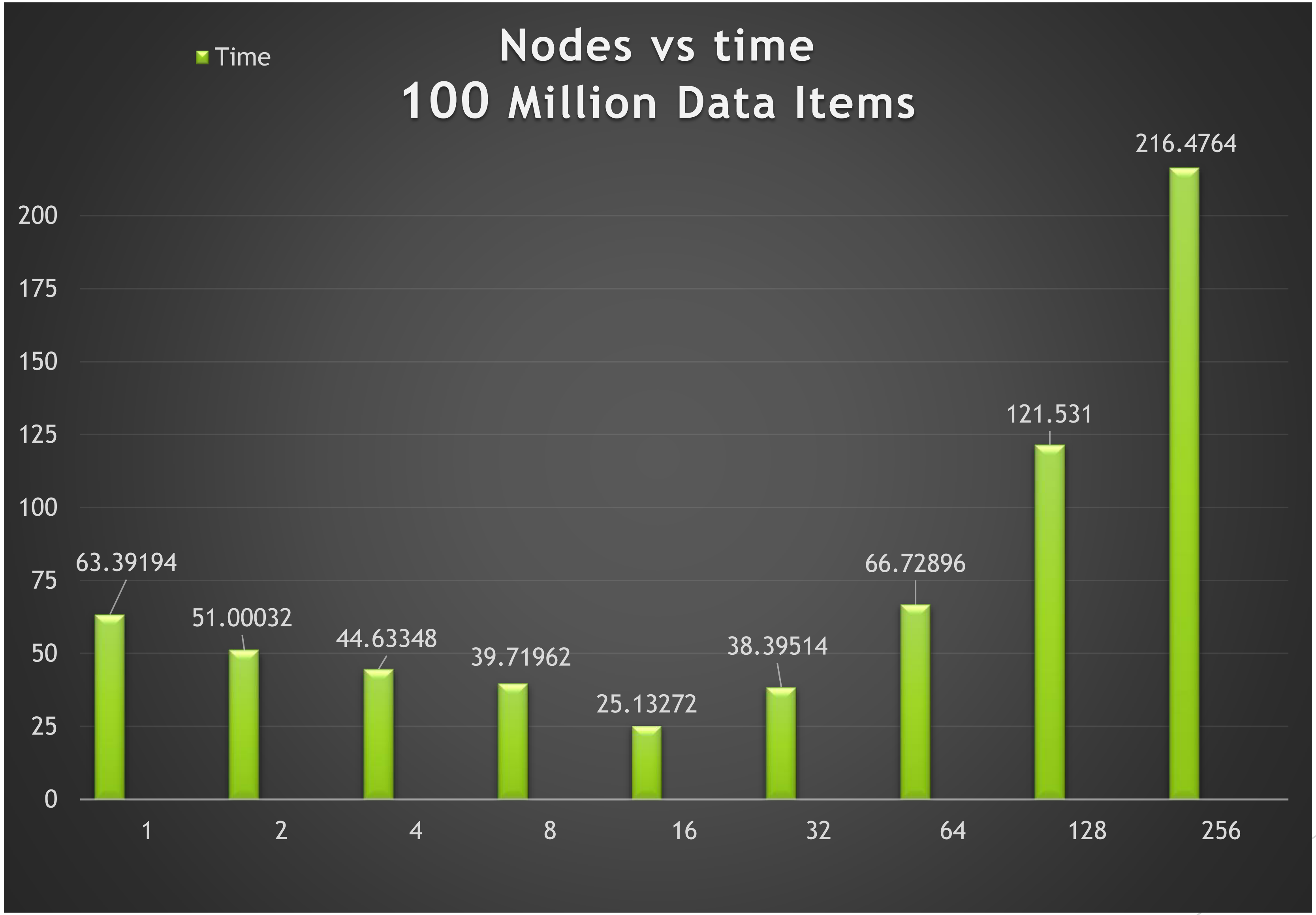
Num of Processors	Reading 1	Reading 2	Reading 3	Reading 4	Reading 5	Average
1	30.9067	28.4611	28.1991	28.4471	30.9821	29.3992
2	19.6237	19.1829	20.4774	19.1715	19.1127	19.5136
4	23.9516	23.3604	22.8615	22.5307	22.6656	23.074
8	14.1024	13.6956	13.6535	13.7426	13.7267	13.7842
16	12.3328	12.1905	12.1798	12.1843	12.1892	12.2153
32	11.5322	11.1226	11.2357	11.0757	10.9717	11.1876
64	24.3776	24.0729	24.2584	24.5778	24.1861	24.2946
128	39.7297	39.2354	39.0507	39.0437	39.8945	39.3908
256	78.6667	79.8285	78.2345	80.0237	78.7782	79.1063



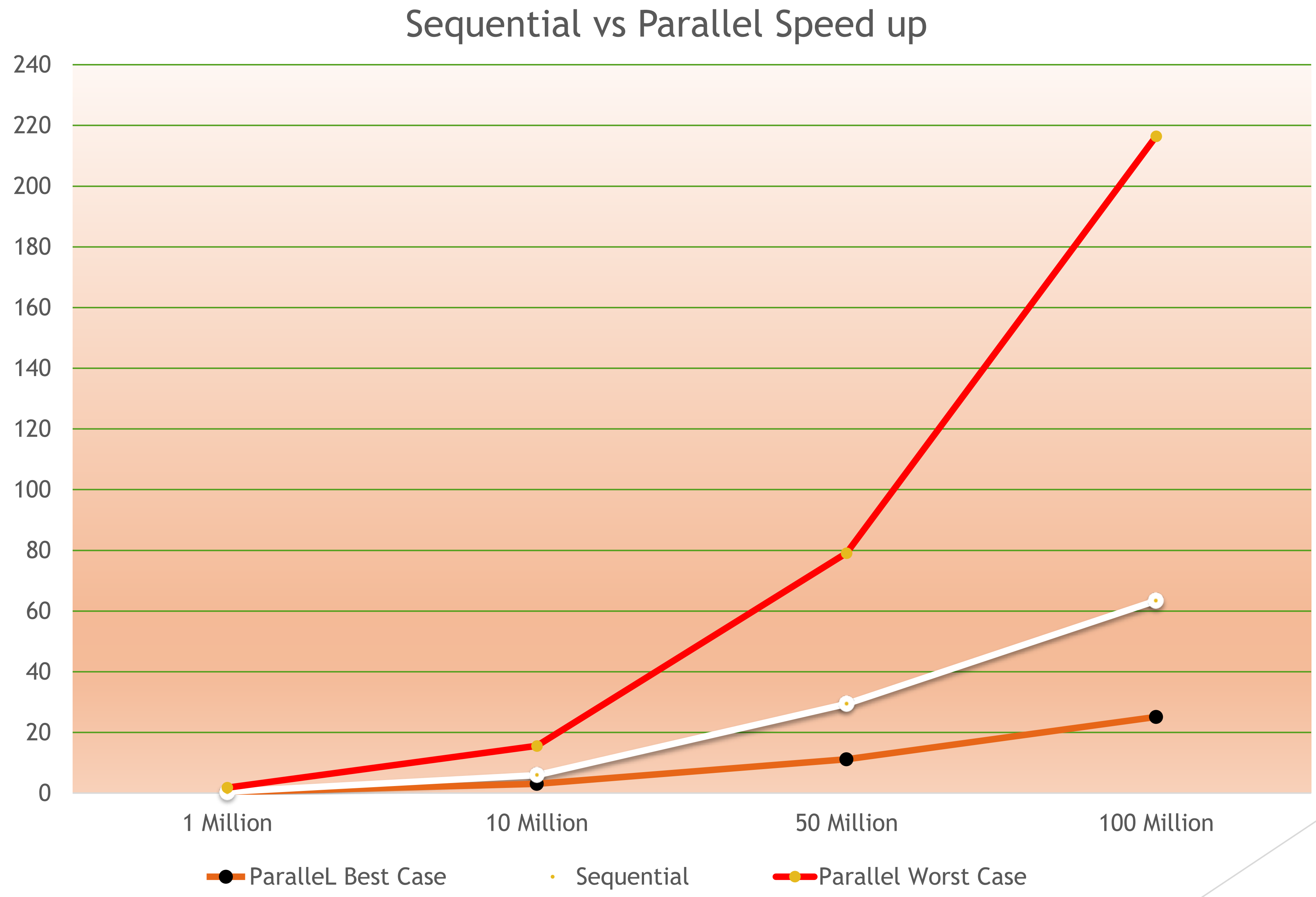
Nodes vs time
50 Million Data Items
Time
80
72
64
56
48
40
32
24
16
8
0
29.39922
19.51364
23.07396
13.78416
12.21532
11.18758
24.29456
39.3908
79.10632
1
2
4
8
16
32
64
128
256

Readings for 100 Million Numbers

Num of Processors	Reading 1	Reading 2	Reading 3	Reading 4	Reading 5	Average
1	69.3125	60.4611	60.4283	62.8341	63.9237	63.3919
2	51.3782	50.4934	50.5294	52.3765	50.2241	51.0003
4	44.8693	44.8474	44.7109	44.4998	44.24	44.6335
8	41.5838	38.9771	39.1323	39.1884	39.7165	39.7196
16	31.5832	23.0936	23.1187	24.7625	23.1056	25.1327
32	41.4321	37.7752	36.2429	38.8732	37.6523	38.3951
64	65.2272	66.792	68.235	66.145	67.2456	66.729
128	113.61	126.206	128.456	117.64	121.743	121.531
256	214.938	234.765	254.26	203.567	174.852	216.476



Time
Nodes vs time
100 Million Data Items
216.4764
200
175
150
125
121.531
100
63.39194
75
66.72896
51.00032
44.63348
50
39.71962
38.39514
25.13272
25
0
1
2
4
8
16
32
64
128
256


Sequential vs Parallel Speed up
240
220
200
180
160
140
120
100
80
60
40
20
0
1 Million
10 Million
50 Million
100 Million
ParalleL Best Case
Sequential
Parallel Worst Case

Take away

- This parallel quicksort algorithm is likely to do a poor job of load balancing.
- Even if one processor has work to do all the other processes have to wait for it to complete.
- Also faced deadlock problems and had to make sure that the blocking functions in MPI were used correctly.
- In order to achieve better performance its critical to identify the optimal number of processors that would be required for any given computation.

References

- Miller Algorithms Sequential and Parallel A Unified Approach 3rd edition
- Parallel Programming in C with MPI and OpenMP by Michael J. Quinn

- *Thank you.*