Project 5 : Parallel Quicksort

Duane Johnson

Mar 12, 2009

Introduction

The quicksort algorithm can be parallelized using a hypercube algorithm. While it can be a great boon to very large number sets, its efficiency for smaller sets is quite poor. In this paper, a parallel quicksort algorithm is described and empirically tested on large data sets, and then compared with an a priori mathematical model of performance.

Procedure

The algorithm tested requires the use of an even 2^n nodes so that approximately "half" of the data can be swapped between nodes whenever a pivot occurs. The main process for sorting is as follows:

1. A large set of integers is evenly distributed among all nodes in the network.
2. Node 0 broadcasts its median to all nodes in its communication group
3. Each node splits its set into integers less than and greater than the median
4. Smaller nodes keep the "less thans" and larger nodes keep the "greater thans"
5. The group is split into two, and the process repeats
6. When all nodes have the correct set of integers, a serial quicksort begins

Experimental Setup

The message-passing library, MPI, was used for the implementation of the parallel quicksort algorithm in C.

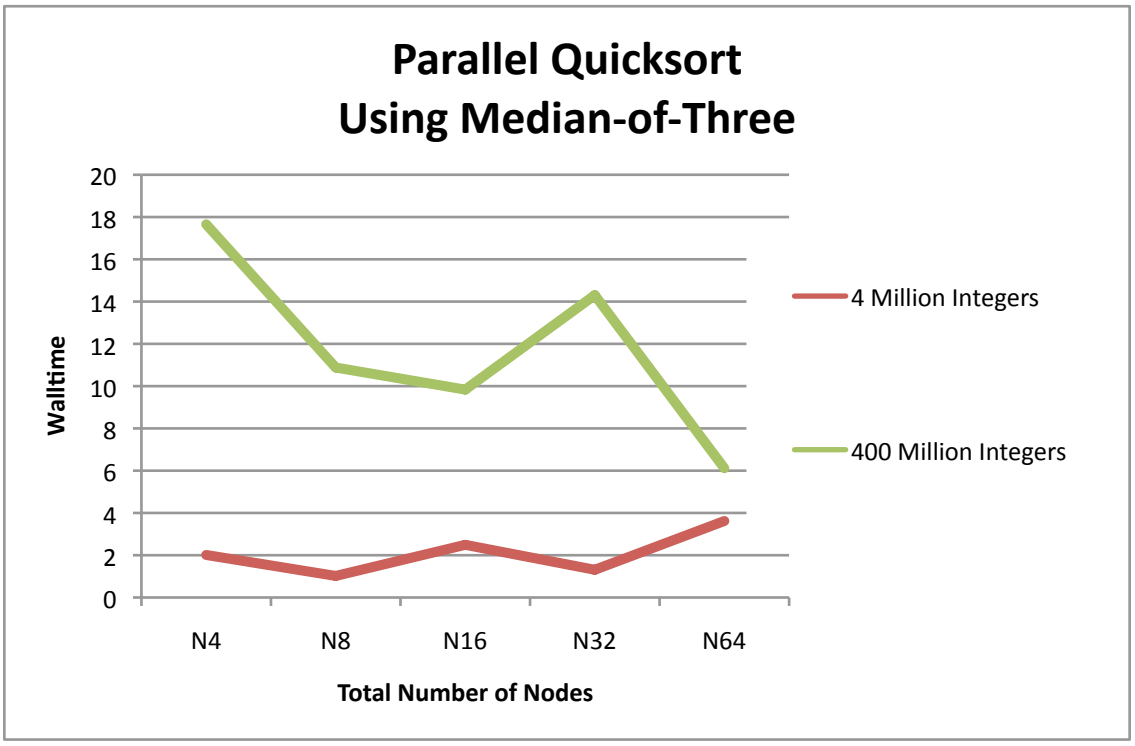
Timing of the solutions were performed on a supercomputer at BYU configured with 2.6GHz Xeon EM64T dual-core processors and 8 GB of fully-buffered RAM. Processes were distributed across a network of nodes using "infiniband" for communication.

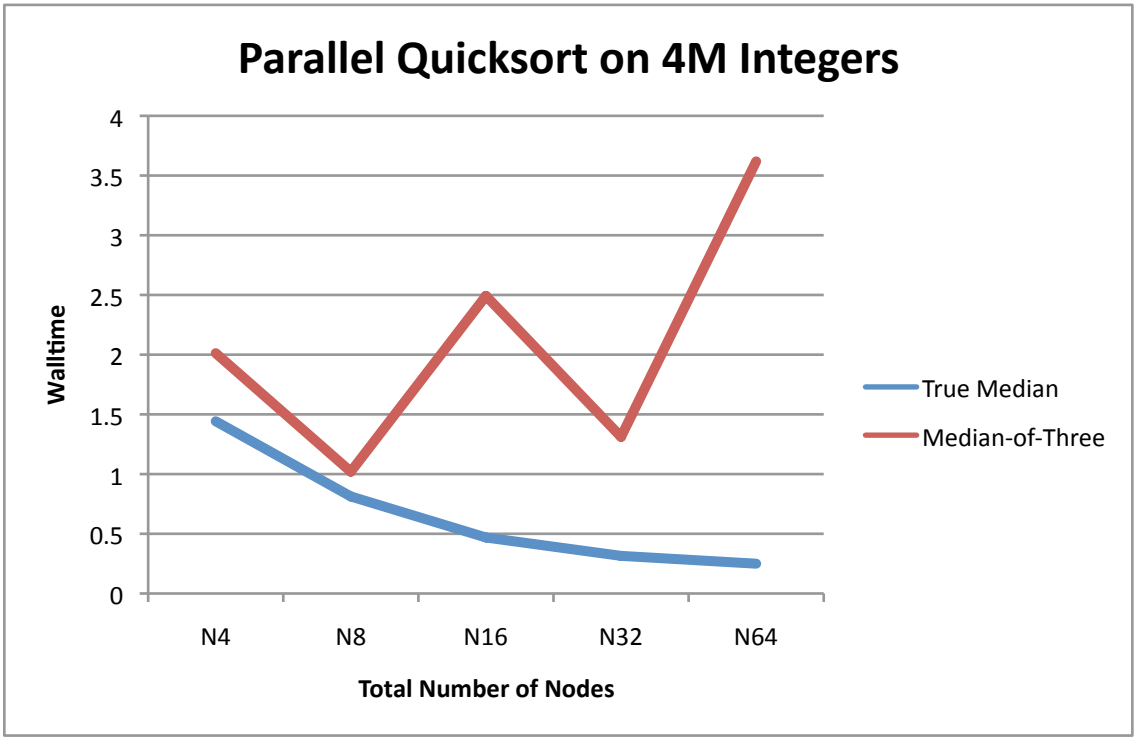
Results

Measurements were made 16 times for each data point, and the median chosen from the varying walltimes. Therefore, a total of 240 runs were made to compile the data in the following charts.

In the first chart, a "median of three" algorithm is used during the pivot procedures to compare the timings of 4 million integers and 400 million integers. Because the data are random, the use of "median of three" is not particularly helpful. As can be seen, the timings are fairly unsteady.

In the second chart, the "median of three" algorithm is compared with a true median algorithm for choosing the pivot point. This causes a much more even distribution of integers between the nodes, and thus, the results are much steadier in the true median case.


Parallel Quicksort
Using Median-of-Three
Walltime
20
18
16
14
12
10
8
6
4
2
0
N4
N8
N16
N32
N64
Total Number of Nodes
4 Million Integers
400 Million Integers


Parallel Quicksort on 4M Integers
Walltime
4
3.5
3
2.5
2
1.5
1
0.5
0
N4
N8
N16
N32
N64
Total Number of Nodes
True Median
Median-of-Three

Conclusion

For random numbers, finding the true median may be worthwhile since it causes the load distribution to be fairly even when compared with the median-of-three method. On the other hand, finding the true median is an expensive operation that would be of little benefit on semi-sorted or sorted data.

An a priori estimate of the parallel quicksort algorithm was given as follows:

Execution Time = n/p * log2(n/p) * CompareTime +
log2(p) *
((latency + 1/bandwidth) +
2*(latency + n/(p*bandwidth) +
(CompareTime * 2*n/p)

Using the following values, the comparison graph below shows a similar trend:

CompareTime = 1 * 10e-9
latency = 1 * 10e-4
bandwidth = 1000000
n = 4000000