CSE 708: Hypercube based Parallel Quicksort using MPI

Sai Vishwanath Venkatesh
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

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

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 $0, 1, \ldots, n - 1$, 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)
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:
\[ \log(p) \cdot [ n/p \log(n/p) + (\text{Time for broadcast}) + \log(n/p) + (\text{Time for data swap MPI SendRecv}) ] \]
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
<table>
<thead>
<tr>
<th>קובץ</th>
<th>1 GB</th>
<th>2 GB</th>
<th>4 GB</th>
<th>10 GB</th>
<th>20 GB</th>
<th>60 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>62.7239</td>
<td>341.8013</td>
<td>710.964</td>
<td>1887.16</td>
<td>3935.94</td>
<td>SlurmTimeOut</td>
</tr>
<tr>
<td>2</td>
<td>97.3456</td>
<td>544.2685</td>
<td>1129.17</td>
<td>1911.79</td>
<td>3829.21</td>
<td>SegFault</td>
</tr>
<tr>
<td>4</td>
<td>65.1068</td>
<td>358.4353</td>
<td>942.285</td>
<td>1650.88</td>
<td>3397.395</td>
<td>SegFault</td>
</tr>
<tr>
<td>8</td>
<td>55.3394</td>
<td>303.9763</td>
<td>679.479</td>
<td>1693.35</td>
<td>2507.14</td>
<td>SegFault</td>
</tr>
<tr>
<td>16</td>
<td>47.8395</td>
<td>265.3903</td>
<td>648.714</td>
<td>1543.02</td>
<td>2236.65</td>
<td>SegFault</td>
</tr>
<tr>
<td>32</td>
<td>35.5152</td>
<td>195.7346</td>
<td>400.773</td>
<td>1158.03</td>
<td>1667.355</td>
<td>SegFault</td>
</tr>
<tr>
<td>64</td>
<td>22.6964</td>
<td>115.9353</td>
<td>241.771</td>
<td>624.37</td>
<td>1099.485</td>
<td>4241.36</td>
</tr>
<tr>
<td>128</td>
<td>14.2269</td>
<td>68.0167</td>
<td>135.585</td>
<td>349.58</td>
<td>698.759</td>
<td>2428</td>
</tr>
<tr>
<td>256</td>
<td>14.4909</td>
<td>56.1877</td>
<td>108.853</td>
<td>274.6</td>
<td>556.83</td>
<td>1842.5</td>
</tr>
</tbody>
</table>
Observations - Runtime

Runtime

Time in Seconds

Number of processing elements
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

<table>
<thead>
<tr>
<th>( p \backslash n )</th>
<th>1 GB</th>
<th>2GB</th>
<th>4GB</th>
<th>10GB</th>
<th>20GB</th>
<th>60GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.6443424253</td>
<td>0.6280012531</td>
<td>0.6296341561</td>
<td>0.9871167858</td>
<td>1.02787259</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>0.9634001364</td>
<td>0.9535927404</td>
<td>0.754510578</td>
<td>1.143123667</td>
<td>1.15851704</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>1.133440189</td>
<td>1.124434043</td>
<td>1.046336973</td>
<td>1.114453598</td>
<td>1.569892387</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>1.311132014</td>
<td>1.28791934</td>
<td>1.095959082</td>
<td>1.223030162</td>
<td>1.759747837</td>
<td>-</td>
</tr>
<tr>
<td>32</td>
<td>1.766114227</td>
<td>1.746248747</td>
<td>1.77398178</td>
<td>1.62962963</td>
<td>2.360589077</td>
<td>-</td>
</tr>
<tr>
<td>64</td>
<td>2.763605682</td>
<td>2.948207319</td>
<td>2.94065045</td>
<td>3.022502683</td>
<td>3.579803272</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>4.408824129</td>
<td>5.025255562</td>
<td>5.243677398</td>
<td>5.398363751</td>
<td>5.632757503</td>
<td>1.746853377</td>
</tr>
</tbody>
</table>
Observations - Speedup

Speedp

- 1 GB
- 2GB
- 4GB
- 10GB
- 20GB

Number of processing elements
Observations and Discussion - Speedup

- In theory the ideal speedup would be \( p * \frac{\log(n)}{\log(n/p)} \) 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

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

- This efficiency is very bad and cascades from our discussion in speedup. The best performance can be witnessed between 1 - 4 processors for the larger 20GB problem with a mere 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


Questions or Comments or Suggestions?

saivishw@buffalo.edu