CS 6230: Homework 3

Christopher Mertin

Due Date: February 24, 2016

Question 1: Computing the Median . . . .

Using the work-depth language, give an algorithm for computing the median of an unsorted array of n integers. What is the work and depth of your algorithm? Is your algorithm work optimal? Suggest how you would find the k-ranked element in an unsorted array.

[*Hint:* The median is the $k = n/2$-ranked element]

Solution: The algorithm for this is known as *QuickSelect* was taken from [1] and made parallel. The algorithm is a Randomized Algorithm and is able to find the k^{th} element in $\mathcal{O}(n)$ time. The parallel version of `QuickSelect` can be seen in Algorithm 1.

Algorithm 1 QuickSelect(A,k)

```
Input: $A \in \mathbb{R}^{n\times 1}$ and $k^{th}$-element
Output: Index of $k^{th}$-element
 1: $r \leftarrow$ Chosen uniformly at random from $[1, A.size()]$
 2: $pivot \leftarrow A[r]$
 3: $A_1, A_2$ are new arrays {Split into a pile $A_1$ small elements and $A_2$ large elements}
 4: #pragma omp parallel for
 5: for $i = 1$ to $n$ do
 6:   if $A[i] < pivot$ then
 7:     $A_1.append(A[i])$
 8:   else
 9:     if $A[i] > pivot$ then
10:       $A_2.append(A[i])$
11:     else
12:       do nothing
13:     end if
14:   end if
15: end for
16: if $k \leq A_1.size()$ then
17:   return QuickSelect($A_1$, $k$) {It's in the pile of small elements}
18: else
19:   if $k > A.size() - A_2.size()$ then
20:     return QuickSelect($A_2$, $k - A.size() - A_2.size()$) {It's in the pile of large elements}
21:   else
22:     return $pivot$
23:   end if
24: end if
```

By making the for-loop in lines 4–12 parallel, you reduce the work to separating the data by a factor of p, where p is the number of processors. This reduces the work to be $\mathcal{O}(n)$ as you need to perform

n-"binnings" of the data (either the large or small arrays), and the depth is $\mathcal{O}(n/p)$. This is work optimal as parallelizing the for-loop does not require anymore work and just decreases the depth of the algorithm.

Question 2: d-dimensional Hypercube Reduction ....

Generalize the hypercube reduction algorithm for a d-dimensional hypercube for the case in which the number of processors is not a power of two and for any problem size $n \geq p$. Derive an expression for the wall-clock time $T(n,p)$ as a function of the number of processors p and the problem size n. Your estimate should include communication costs in terms of latency, bandwidth, and message size.

Solution: The solution to this problem can be simplified in the following way. You do need 2^d nodes for a d-dimensional hypercube, but when you're looking at the network complexity, it is easy to treat *all network topologies* as you would similar to that of a hypercube and go from there. For example, a 3-dimensional hypercube can be represented as

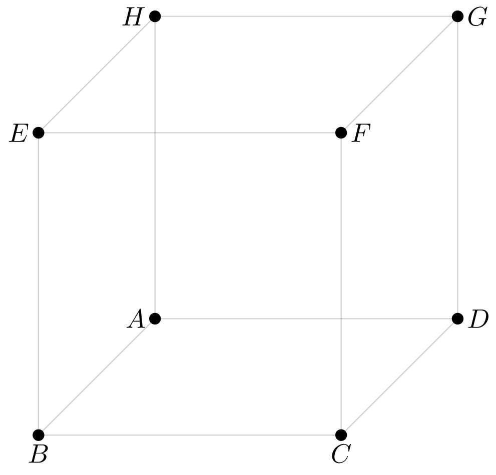

Figure 1: 3-dimensional Hypercube

This can be redrawn in a 2-dimensional plane for *any* d-dimensional hypercube. This doesn't make the compleixty any harder, though does make it easier when designing an algorithm. It can be represented as

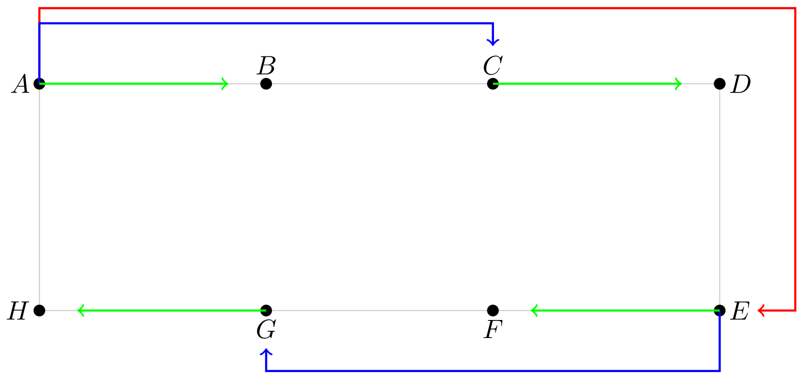

Figure 2: 3-dimensional Hypercube Communication (One-To-All)

which represents the optimal communcation path for a 3-dimensional hypercube. The red represents the data sent during the first clock cycle, the blue is the second clock cycle, and green is the third. While this isn't the *reduction*, the reduction algorithm would simply be the reverse of this.

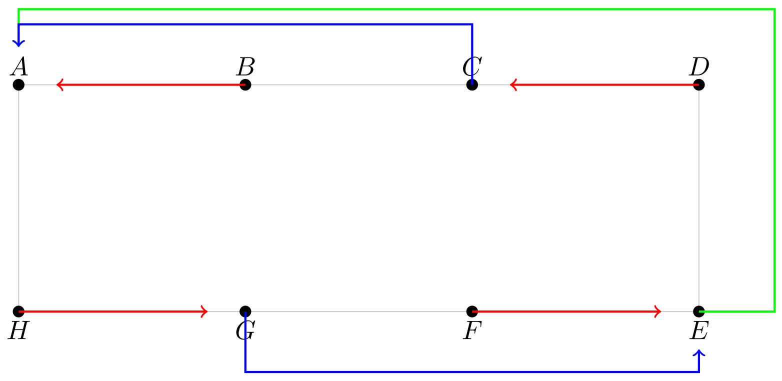

Figure 3: 3-dimensional Hypercube Reduction (All-To-One)

The pseudocode for the d-dimensional hypercube communication can be seen in Algorithm 2. Note: The inner for-loop for the code is called all at once in practice, and it doesn't iterate through the entire list of nodes. However, this was the best way that I could determine to represent it in algorithmic form. For the hypercube reduction, that can be seen in Algorithm 3. For Algorithm 3, we first assume that each node in the network has been given (or already computed) it's share of the values, meaning that each node has n/p values initially. For the *very first clock cycle*, each node will perform the reduction on the data that it already has before entering the for-loop.

Algorithm 2 Hypercube Communication (One-To-All)

```
1: for $i = 1$ to $\log_2(p)$ do
2:     for $j = 1$ to $p$ do
3:         if $p_j$ contains data then
4:             $m = j + 1 + \left\lfloor \frac{p}{2^i} \right\rfloor$
5:             $SendData\,(p_j \to p_m)$
6:         end if
7:     end for
8: end for
```

Algorithm 3 Hypercube Reduction (All-To-One)

```
1: for all nodes do
2:   Reduce $n/p$ data on each node
3: end for
4: for $i = \log_2(p)$ to 1 do
5:   for $j = 1$ to $p$ do
6:     if $p_j$ contains data then
7:       $m = j - 1 - \left\lfloor \frac{p}{2^i} \right\rfloor$
8:       $SendData\,(p_j \to p_m)$
9:     end if
10:  end for
11: end for
```

Again, the inner for-loop is done all at once for those nodes which are supposed to send the data. From this, we can get the time complexity of this algorithm, which we can make the following approximation. Even with $n \geq p$, as long as it isn't such that $n \gg p$ then we can say that the initial reduction is essentially negligable. This is due to the time to reduce the first n/p values would be much less than the communication time. If not, it would be the case that it would take n/p clock cycles which can be added to our final time complexity. From here, we can build the time complexity as being

$$T(n,p) = \log_2(p) \cdot t(s) + \frac{n \cdot m(s)}{bandwidth}$$

where $t(s)$ is the time (in seconds) of the latency, *bandwidth* is the bandwith of the network in *bytes/second*, $m(s)$ is the size of the message in *bytes*. The $\log_2(p)$ term comes about from having to send $\log_2(p)$ messages. The more generalized time complexity for $n \gg p$ can be defined as below where $c(s)$ is *clock cycles/second*

$$T(n,p) = \log_2(p) \cdot t(s) + \frac{n \cdot m(s)}{bandwidth} + \frac{1}{c(s)}\frac{n}{p}$$

Question 3: Finding MPI Bugs . . . .

This archive contains the files `mpi_bug1.c`, `mpi_bug2.c`, ..., `mpi_bug7.c`. These example codes contain bugs, resulting in hangs or other undesirable behavior. Try to find these bugs and fix them. Add a short comment to the code describing what was wrong and how you fixed the problem. Add the solutions to your submission using the naming convention `mpi_solved1.c`, ..., `mpi_solved7.c`. Each problem should be run with 4 MPI tasks.

Solution: The fixed code can be found in the tarball `mpi_solved.tar`. The code is commented where I made changes, but they are also found here

- `mpi_bug1.c`
  - The problem in this instance was that when doing an `MPI_Send` and `MPI_Recv` that the `tag` variable needs to be the same for both function calls on each node. Simply making these the same was the solution to the problem.

- `mpi_bug2.c`
  - When sending and receiving data with MPI, the sent and receiving data need to be of the same size. The problem here was that the root node was sending an integer and the receiving node was trying to receive a float. This caused it to think it had a float when an integer was sent. Simply changing the data type in the `MPI_Recv` call and the data type of the variable `beta` solved the issue.
- `mpi_bug3.c`
  - In this instance, MPI functions were being called but `MPI_Init` and `MPI_Finalize` were not defined at all. These two functions need to be defined at the beginning and end of the program before MPI functions can be used.
- `mpi_bug4.c`
  - For this program, the problem was that the root node wasn't reducing it's data after receiving the results from each task. By simply adding an `MPI_Reduce` to the root node when it's getting the final sum solved this issue.
- `mpi_bug5.c`
  - It wasn't such of a problem as I was running this on a single machine, but the use of `MPI_BYTE` instead of `MPI_CHAR` for the data-type is a *potential problem* if this code was distributed among multiple architectures. `MPI_BYTE` sends a certain number of bytes, but using `MPI_CHAR` converts between sizes for different architectures.
- `mpi_bug6.c`
  - In the code, the arrays were treated as if they were a single array that was shared between the nodes. There was a variable called `offset` that was different for each node to help with saving data. However, this isn't the case and each node actually gets its own version of the array, so by simply setting `offset` to be 0 for each node solved the issue.
- `mpi_bug7.c`
  - Finally, the variable `count` was set to the `taskid` of each node. Therefore, when the root node called `MPI_Bcast`, it set the size of the data to be 0 since `count` was set to the `taskid` which was 0 since it was the root. By changing the value of `count` to be the size of the data being broadcasted (1 since only one value was being sent) resolved this issue.

Question 4: MPI Ring Communication . . . .

Write a distributed memory program that sends an integer in a ring starting from process 0 to 1 to 2 (and so on). The last process sends the message back to process 0.

- Allow for a command line parameter N that specifies how often the message is sent around the ring.
- Start with sending the integer 0 and let every process add its rank to the integer before it is being sent again. Use the result after N loops to check if all processors have properly added their contribution each time they received and sent the message.
- Time your program for a larger N and estimate the latency on your system (*i.e.* the time used for each communication). If possible, try to test your communication ring on more than one machine such that communication must go through the network. Note that if you use MPI on a single processor with multiple cores, the available memory is logically distributed, but messages are

not actually sent through a network. (It depends on the implementation of MPI how it handles sending/receiving of data that is stored in the same physical memory.)

- Modify your code such that instead of a single integer being sent in a ring, you communicate a large array in a ring. Time the communication and use these timings to estimate the bandwidth of your system (*i.e.* the amount of data that can be communicated per second).

Solution: The program was built as it was stated above, and the results can be seen below. The calculated latency was determined to be 0.0001935 s, and the calculated bandwidth was calculated to be 60.9 $^{\text{Gigabytes}}/_{\text{s}}$. The true value uof the bandwidth was 56 $^{\text{Gigabytes}}/_{\text{s}}$, so the program provided a good approximation of the bandwidth.

To calculate the latency, the single integer was sent around the ring to each node N times, and then the average time was taken by calculating $\frac{total\ time}{sends}$ which is displayed as the *Average time* in the code output below.

To calculate the bandwidth, the size of the array was calculated by using the `sizeof()` command. This large array was sent to each node, and then the average time to send to each node was computed. To get the bandwidth, it was simply $\frac{size\ of\ array}{average\ time}$. This produced an approximation of the bandwidth that had an error of only 7.14% of the true value.

```
MPI Ring
--------
Number of nodes:  16
Number of loops:  10
Total:  1200
Run time:  0.0309548 seconds
Average time:  0.000193468 seconds
Number of Sends:  160

Testing for large array
-----------------------
N = 10000000
Average time to send the large array:  0.00393932 seconds
Size of the array:  240000000 bytes
bytes/second:  6.09243e+10
```

Question 5: Parallel Binary Search. . . . .

Implement an MPI+OpenMP parallel binary search algorithm that has a similar signature to the ANSI C `bsearch()` routine but it supports both OpenMP and MPI parallelism and multiple keys. For this reason, it should have three additional arguments compared to the standard `bsearch`: (1) the number of keys, (2) the MPI communicator, and (3) the number of threads per MPI process. Write a driver routine called `search` that takes one argument, the size of the array to be searched (initialized to random integers). On the write-up, give pseudocode for your algorithm, and report the wall clock time/core/n results for $n = \{1M, 10M, 100M, 1B\}$ elements for 1 key, using up to four nodes on Tangent with 1 MPI task per socket. Report weak and strong scaling results using 1 core, 1 socket, and 1, 2, and 4 nodes.

Solution: The implementation of how to use this function can be found in the `README` file that is included. The pseudocode can be seen in Algorithm 4, but a brief summary is that for a given array size N, the number of elements on each node would be $\frac{N}{nodes}$. If a given node finds a key in its array, then it saves its *local index* which is just the index that it was found at in the nodes array segment. At the end, line 22 will update where the keys were found for their *global index* on the whole array. What it does here is if a key was found at a *local index* of 5 on the 3rd node, then the *global index* of that key is going to be $5 + \frac{N}{nodes} \times 2$, since the 3rd node would be dealing with the indices in the range $\left[\frac{N}{nodes} \times 2, \frac{N}{nodes} \times 3\right)$ of the original array.

Algorithm 4 Parallel Binary Search

```
Input: Iterator $begin$, Iterator $end$, $Keys \in \mathbb{R}^{n\times 1}$, $parallelSettings$
Output: $positions \in \mathbb{R}^{n\times 1}$
 1: MPI::Broadcast($Keys$) to every node
 2: $positions \in \mathbb{R}^{n\times 1}$
 3: for $i = 1$ to $n$ do
 4:   $key = Keys[i]$
 5:   $found =$ false
 6:   #pragma omp parallel for
 7:   for $itr = begin$ to $end$ do
 8:     $middle = begin + \frac{end-begin}{2}$
 9:     if $middle \equiv key$ then
10:       $positions[i] = middle.index()$
11:       $found =$ true
12:       #pragma omp flush($found$) {Tells other OpenMP threads to stop looking}
13:     else
14:       if $middle > key$ then
15:         $end = middle$
16:       else
17:         $begin = middle + 1$
18:       end if
19:     end if
20:   end for
21: end for
22: Calculate $global\ indices$ from $local\ indices$
23: $positions =$ MPI::Allgather($positions$)
24: return $positions$
```

This algorithm was run over 10 different instances for the above specifications with 1 CPU, 1 Socket, and 1, 2, and 4 nodes. Table 1 shows the average timing for these instances, where each run was on a different node and then the average over all 10 was taken. Performing this algorithm on 10 different nodes, without overlap, prevented any speed up due to caching.

These values were also plotted in Figure 4. From the table and the figure below, it's easy to see the linear decrease in the time as the array size increased.

Table 1: Average Timing for Nodes (in seconds)

$Nodes$	$N = 1,000,000$	$N = 10,000,000$	$N = 100,000,000$	$N = 1,000,000,000$
1	0.078	0.090	0.163	0.811
2	0.090	0.087	0.092	0.573
4	0.094	0.091	0.091	0.344

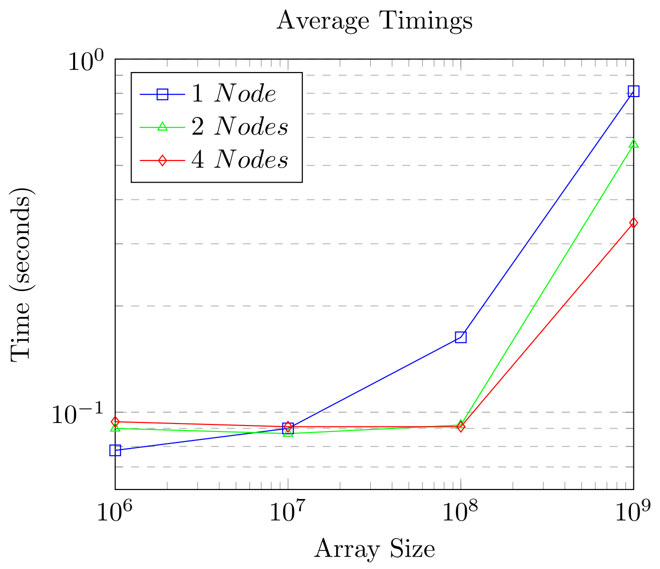

Figure 4: Average Timings for Binary Search

For the case of 10^6, using a single node was faster than the other two cases, which is most likely due to the communication costs at the beginning and end of the algorithm. However, as the size of the array increased, for $N = 10^8$ there was a 2-times speedup for both 2 and 4 nodes when compared to a single node. At $N = 10^9$, there was roughly a 2-times increase in speed for using 2 nodes and roughly a 3.5-times increase in speed for using 4 nodes. This shows the scalability of the algorithm only achieved any usefulness for $N > 10^7$, which is likely due to the $\mathcal{O}(\log(N))$ complexity of the Binary Search algorithm, so there wouldn't be significant gains that would be able to be seen until the constraint $\mathcal{O}(\log(N)) > \mathcal{O}\left(\log\left(\frac{N}{p}\right) + \log(p)\right)$ was met, where the $\log(p)$ term comes about due to the communication costs.

From these results, we can calculate the *Strong Scaling* and *Weak Scaling*, which can be seen in Table 2

Table 2: Strong and Weak Scalability

Nodes	$N = 1,000,000$	$N = 10,000,000$	$N = 100,000,000$	$N = 1,000,000,000$
		Strong Scaling		
2	43.33%	51.72%	88.59%	70.77%
4	20.74%	24.73%	44.78%	58.94%
		Weak Scaling		
2	86.67%	103.45%	177.17%	141.54%
4	82.98%	98.90%	179.12%	235.76%

References

[1] James Aspnes, Yale, `http://www.cs.yale.edu/homes/aspnes/pinewiki/QuickSelect.html`