Big Data Homework 3

Tiago Antunes

March 16, 2021

1 Solution

The given solution applies a logarithmic steps reduction, that is, it reduces by half the amount of transfers needed in each step (similar to a binomial tree reduction).

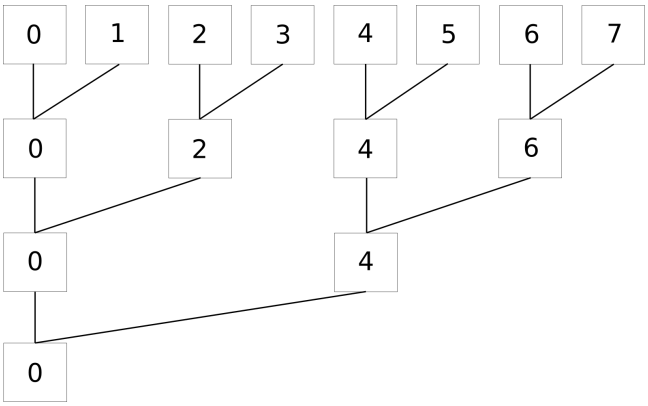

Figure 1: Drawing of the solution with 8 nodes

Figure 1 represents the solution present in the code. Each number in the square is the *rank* of the node and the connections are the reductions that happen. A static planning on how the solution happens is applied and so each node can take the decision and know what to do without requiring a master node to inform it. Listing 2 shows the pseudocode of how the solution works.

The tuning aspect of the algorithm is not shown in it for simplicity. The way to determine the target is via the s, *mod* and *rank* variables. There are two types of nodes through each step of the algorithm: *collector* and *sender*.

The collector of each level is always at positions $2 * s$ and so that's why there's the arithmetic shift and we assign this value to *mod*. Then, all the nodes that are not in multiples of this value are considered senders - on each step, there's only 1 sender per each *chunk*, as it's seen in Figure 1. To determine to whom

to send, there is the *target* variable which is always the first node of the *chunk*. The last *collector* of the algorithm will be node 0, which is the desired result.

To make this algorithm work in non-powers of 2 it has to consider whether or not there are enough nodes, which is done in line 14. Each node is at a distance s, so given *rank* and s the node can find out if it should skip or not that iteration. There is no need for global synchronization since the sends and writes will be the ones indicating whether or not to proceed to the next step of the algorithm.

```
1  fn YOUR_Reduce(sendbuf[], recvbuf[], count)
2      s := 1
3      declare save[] array of size count
4
5      size := MPI_Comm_size()
6      rank := MPI_Comm_rank()
7      save = sendbuf // all values from sendbuf go to save
8      while (s < size) then
9          mod := s << 1 // shift left 1 unit
10         if rank % mod != 0 then
11             target := floor(rank / mod) * mod
12             MPI_Send(save, target)
13             break // the node can exit this function now
14         else
15             if rank + s < size then
16                 source := rank + s
17                 MPI_Recv(recvbuf, source)
18                 save = sum(save, recvbuf)
19
20         s := s << 1
21
22     return save
```

Listing 1: Pseudocode of the blocking solution

After implementing this is C++, I realized that the performance was good but still somewhat slower than the official MPI solution and so I profiled my solution. The conclusions were:

- The program was spending a lot of time in both send and receive calls (I was using blocking calls)
- The sum operation takes some time for bigger calls

After analysing the solution, I found that I could parallelize the sum and receive operations by doing them simultaneously. By using non-blocking MPI calls, I could perform a call to *sum* while the system was also receiving the information. Then, before initiating the next step of the algorithm, a blocking call to **MPI_Wait** can block the program from proceeding until it has received

the required information. This means that on each step t it would be processing the data from step $t-1$.

```
1  fn YOUR_Reduce(sendbuf[], recvbuf[], count)
2      s := 1
3      declare save[] array of size count
4      declare extra[] array of size count
5
6      size := MPI_Comm_size()
7      rank := MPI_Comm_rank()
8      save := sendbuf // all values from sendbuf go to save
9
10     *fill_arr := extra // point to extra
11
12     while (s < size) then
13         mod := s << 1 // shift left 1 unit
14         if rank % mod != 0 then
15             target := floor(rank / mod) * mod
16             MPI_Wait() // wait last recv to finish
17             MPI_Send(save, target)
18             break // the node can exit this function now
19         else
20             if rank + s < size then
21                 source := rank + s
22                 swap(fill_arr, recvbuf, extra)
23                 MPI_IRecv(recvbuf, source)
24
25                 // sum with the non fill_arr from t-1
26                 if fill_arr = recvbuf then
27                     save := sum(save, extra)
28                 else
29                     save := sum(save, recvbuf)
30
31         if rank == 0 then
32             MPI_Wait()
33             save := sum(save, fill_arr)
34
35         s := s << 1
36
37     return save
```

Listing 2: Pseudocode of the non-blocking solution

Due to this new logic, a new array must be allocated. This means that besides the *save* array, there is also an extra one. Because it is only one, the memory complexity from this function continues to be **O(n)** per node, which I consider a good trade-off.

#Nodes	Size	MPI Duration (ms)	Duration (ms)
2	65k	9.054	8.969
	1M	114.128	49.381
	16M	1327.228	383.403
	256M	20507.279	5118.520
4	65k	5.048	8.498
	1M	31.933	84.826
	16M	622.920	553.014
	256M	10210.159	7871.490
8	65k	10.163	14.505
	1M	37.199	100.395
	16M	594.722	857.843
	256M	9365.042	10969.555

Table 1: Table with the results

2 Results

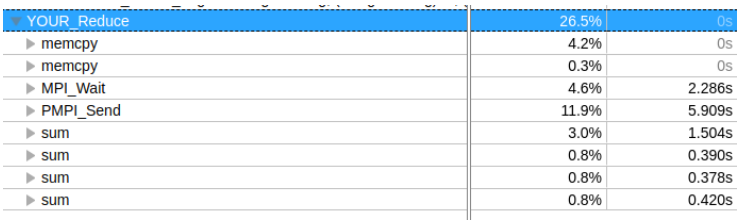

Figure 2: Profiling results, running 3x each size until 128Mb in multiples of 8, 4 nodes

The results obtained in Table 1 show that the algorithm performs not so badly. After profiling the code, it was noticeable that the **MPI_Send** function is taking a long time and might be affecting the overall performance. The results are displayed in Figure 2. This shows how heavy the send and receive functions are in terms of impacting performance and how processing should be done simultaneously to guarantee maximum throughput.

One thing to note is also the difference between the speeds with different nodes. Considering the 256M as reference, MPI took 19s to finish it using 2 nodes (which I found weird and ran again to check, getting the same result) while for a bigger amount of nodes it performed faster. My solution on the other hand is fast for a smaller amount of nodes. This leaves me to think that MPI's Reduce algorithm is different than the one implemented.

One thing to note in Figure 2 is that I ran it locally due to not being able to profile on the server. Due to memory constraints (16GB) I had to change the script so that it could measure decently. The time taken by the sum function is relevant, which can justify the parallelization improvement of this function.

# Nodes	# Threads	MPI Duration (ms)	Duration (ms)
2	1	20342.789	5119.151
	2	21020.924	5113.811
	3	20046.758	3772.181
	4	18976.556	3486.018
4	1	9802.884	7298.245
	2	8864.990	6357.218
	3	10434.312	6093.742
	4	11679.009	6065.121
8	1	7937.878	11552.348
	2	8728.196	8826.472
	3	7220.651	8442.273
	4	8022.704	7666.291

Table 2: Results when running with parallelization within each node, with blocks of 256M elements

3 Bonus

Table 2 contains the results of using OpenMP to parallelize the function *sum* of the given code. Since it's a static amount of sum of integers, I opted for a static scheduler for load balancing, thanks to my past experience in homework 2. The parallelization is a simple *parallel for* optimization from OpenMP. Also, the table includes the MPI duration so that it can show the workload on the server at the time for comparison.

The results displayed on the table show that the sum operation was taking a long time across all numbers of nodes but the performance barely surpassed MPI's implementation for 8 nodes. Assuming that the MPI solution is better given how many more nodes there are, I expect that for 16 nodes my solution would perform worse. This information complements what was mentioned in section 2.