Homework #3 – MPI Programming

Yoke Kai Wen, 2020280598

March 12, 2021

1 Introduction

In this assignment, I implemented a reduction algorithm on a cluster of nodes using MPI, in single-threaded and multi-threaded implementations. The task is to sum up all the arrays input to each process and return the combined value in the output buffer of process 0.

2 Detailed description of implementation

I implemented a parallel reduction algorithm with `MPI_Send` and `MPI_Recv`. Suppose number of nodes $N = 8$. The vanilla unoptimised version would involve a for loop of 7 iterations, with node 0 receiving the arrays from each of the other 7 nodes and adding them to the output array in each iteration. On the other hand, the optimised tree-based version involves only $log_2 N = 3$ iterations in the for loop, leading to better time performance. The comparison between the optimised and unoptimised versions is shown in Figure 1.

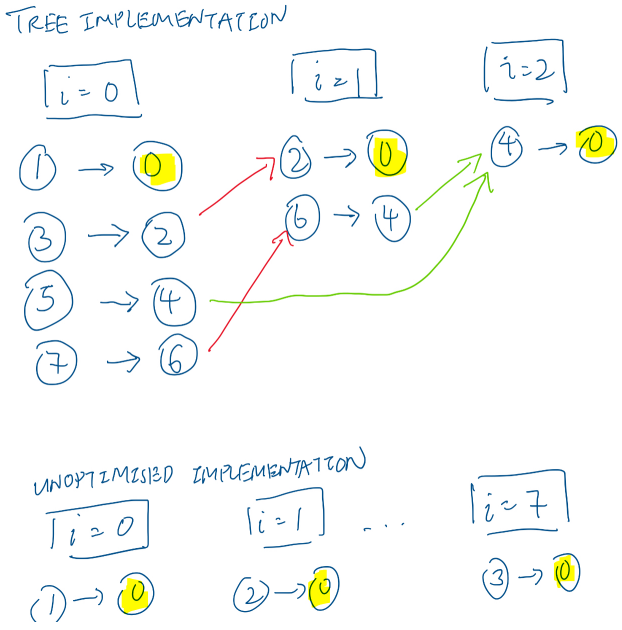

Figure 1: Diagram showing tree implementation vs unoptimised implementation of reduction algorithm for 8 nodes

Instead of waiting for process 0 to communicate with each of the other processes individually, the tree implementation allows the accumulation of arrays to occur in other intermediate processor nodes, before being passed on to the root process 0, thus speeding up the computation. In the first iteration, half the nodes send their arrays to the other half of the nodes to be summed at each receiver node; in the second iteration, out of the receiver nodes in the first iteration, half of them send their accumulated arrays to the other half and this goes on until all the arrays are accumulated in one node (0). This is done by two nested for loops, with the outer loop iterating over $log_2(N)$ times, and the inner loop incrementing relative to the outer loop variable, in a way such that the sender and receiver can be chosen correctly to allow for the optimal communication for the reduction task. The code snippet below demonstrates the tree implementation, and only works when number of nodes is a power of 2.

```
void YOUR_Reduce(const int* sendbuf, int* recvbuf, int count) {
  int rank, size;
  int i, j, k;
  int tag = 0;

  int *tmpbuf;
  // initialize
  tmpbuf = new int [count];
  for (i=0; i<count; i++) {
    recvbuf[i]=sendbuf[i];
  }

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // binary tree implementation
  for (i = 0; i < log2(size); i++) {
    for (j = 0; j < size; j += 1 << (i + 1)) {
      const int receiver = j;
      const int sender = j + (1 << i);
      if (rank == receiver) {
        MPI_Recv(tmpbuf, count, MPI_INT, sender, tag, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
        for(k=0; k<count; k++) {
          recvbuf[k]+=tmpbuf[k];
        }
      }
      else if (rank == sender) {
        MPI_Send(recvbuf, count, MPI_INT, receiver, tag, MPI_COMM_WORLD);
      }
    }
  }
}
```

2.1 Bonus detailed implementation

I added OMP `#pragma omp parallel for schedule (static)` directive in front of the `for` loop that performs the summation operation between two arrays, allowing this for loop to be executed via multithreading. This is because for large arrays, parallelising the computation of this `for` loop can be much faster than the serial single-thread implementation.

```
#pragma omp parallel for schedule (static)
        for(k=0; k<count; k++) {
          recvbuf[k]+=tmpbuf[k];
```

```
4         }
```

3 Time results

Table 1 shows the 12 time results of 3 configurations of NP (2,4, and 8 nodes) and for 4 different array sizes (64k, 1M, 16M, and 256M integers).

		Number of nodes		
		2	**4**	**8**
Array size	**64k**	(10157) 4112	(5892) 9479	(7006) 13366
	1M	(93335) 51324	(29715) 90111	(476082) 103296
	16M	(1453383) 259727	(622246) 490120	(476082) 707304
	256M	(17162147) 3504660	(9706293) 6997394	(7206965) 9927390

Table 1: Table showing time results in us for 2,4,8 nodes, for different array sizes of 64k, 1M, 16M and 256M integers, with official MPI_Reduce implementation time shown in brackets

Table 2 shows the 12 time results of 3 configurations of NP (2,4, and 8 nodes) and for 4 different numbers of threads (1,2,3 and 4 threads per node) implemented with OMP.

		Number of nodes		
		2	**4**	**8**
Number of threads	**1**	(21080107) 3510567	(9364050) 7138020	(7672271) 9800140
	2	(20238105) 2379197	(8366491) 5387243	(9852543) 7622929
	3	(20898887) 2077628	(7640353) 4421534	(8634925) 7066168
	4	(19204921) 2091700	(8240319) 4588794	(8046237) 6702875

Table 2: Table showing time results in us for 2,4,8 nodes and 1,2,3,4 threads, for array size of 256M integers, with official MPI_Reduce implementation time shown in brackets