PCSE Spring 2016 Homework 4: Exchange-Sort Exercise

Neil Klenk - nlk322

2/23/16

0.1 Introduction

In this exercise, the exchange-sort algorithm was implemented between many processors. Each processor was given a single number, and compared with its neighboring processor to determine if their values needed to be swapped or not. The algorithm was split into two stages: Even stage: processors 2i and 2i + 1 compare and swap. Odd stage: processors 2i + 1 and 2i + 2 compare and swap. by repeating this P/2 times, where P is the number of processors, the processor numbers in ascending order would contain the corresponding number in ascending order.

0.2 Theory

1. In total the parallel code should take P/2 iterations where P is the number of processors being used. 2.The algorithm takes P/2 sequential steps. T1: 0.00549 sec. This each comparison on the initial node took approximately 0.000011 seconds. For an array consisting of 32 values, it would take 0.00549 seconds to perform all of those comparisons on one processor. Tp: 0.067 sec for 32 processors (2 nodes). T(infinite): 0.000177 sec. Since each processor only containes one value, and this comparison is performed directly between a processor and its partner, there is no opportunity to gain parallelism once you leave the first node. Sp: 0.082 sec. This was determined by dividing T1 by the actual time it took (0.067 sec). This low value demonstrated the effects of network latency. Ep: 0.002563 sec. Average amount of parallelism: 31.0169. This value was obtained by dividing the time it would take on one processor by the amount of time it took when constrained to one node. 3. T1 is the sequential time (Time for one processor to work though it). Thus Sp and Ep would remain the same.

0.3 Writeup

0.3.1 The Algorithm and Corresponding Code

This section of the code takes care of the edge cases that appear when the partner rank is set to either -1 or ”size”. This occurs in the odd stage where the partner to rank = 0 is rank = -1, when the partner to rank = 31 is rank = 32. In each case the processor does not exist, so they send and receive from and to ”MPI_PROC_NULL” to prevent the code from hanging.

```
if(rank == 0)
prevProc = MPI_PROC_NULL;
else
```

```
prevProc = rank - 1;

if(rank == size - 1)
nextProc = MPI_PROC_NULL;
else
nextProc = rank + 1;
```

This section of the code is the meat of the program. It functions by first evaluating the stage that the program is in, either even or odd. Depending on if it is the even or odd stage, the swap and subsequent compare happen between different processors. If it is the even stage, the even processor compares with the following processor. The odd processor keeps the larger of the two values while the even processor keeps the smaller of the two values. If it is in the odd stage, the even processor compares with the preceding processor. The even processor keeps the larger value while the odd processor keeps the smaller value.

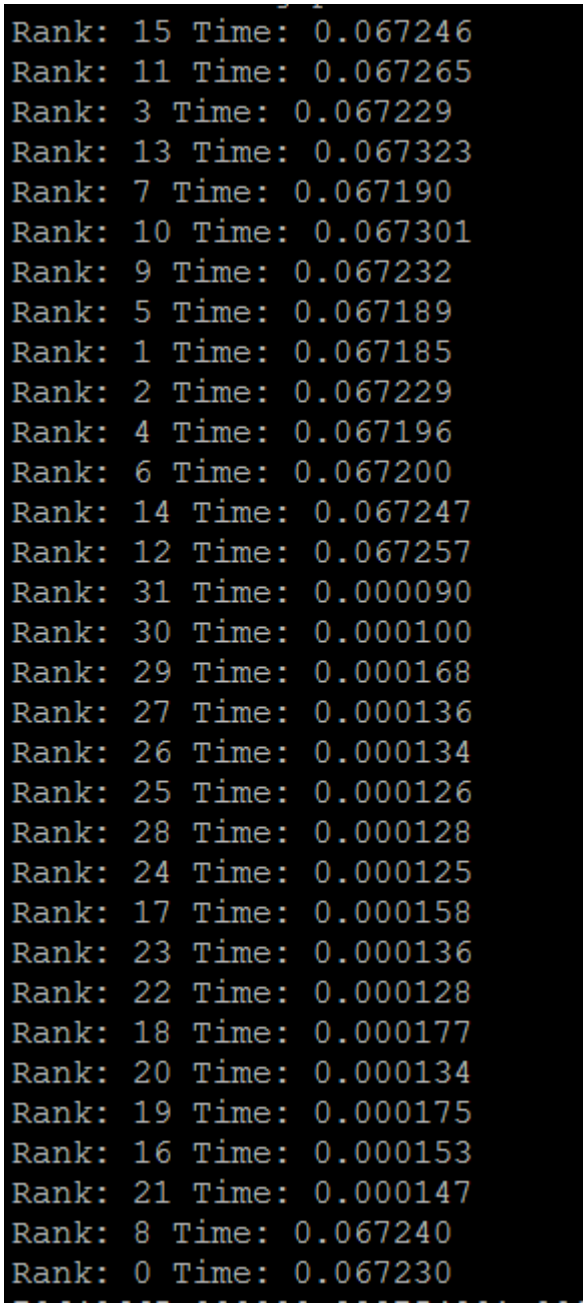
```
while(stage_count <= size/2){
other_num = process_num;
  if(stage_count % 2 == 0){ //even stage
    if(rank % 2 == 0){ //if even
    MPI_Send(&process_num,1,MPI_DOUBLE,nextProc,10,MPI_COMM_WORLD);
    MPI_Recv(&other_num,1,MPI_DOUBLE,nextProc, 20,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    process_num = (process_num < other_num) ? process_num:
        other_num;
    }
    else{ //if odd
    MPI_Recv(&other_num,1,MPI_DOUBLE,prevProc, 10,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&process_num,1,MPI_DOUBLE,prevProc,20,MPI_COMM_WORLD);
    process_num = (process_num > other_num) ? process_num:
        other_num;
    }
   }
  else{ //odd stage
    if(rank % 2 != 0){ //if odd
    MPI_Send(&process_num,1,MPI_DOUBLE,nextProc,30,MPI_COMM_WORLD);
    MPI_Recv(&other_num,1,MPI_DOUBLE,nextProc, 40,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    process_num = (process_num < other_num) ? process_num:
        other_num;
    }
    else{ //if even
    MPI_Recv(&other_num,1,MPI_DOUBLE,prevProc, 30,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&process_num,1,MPI_DOUBLE,prevProc,40,MPI_COMM_WORLD);
```

```
    process_num = (process_num > other_num) ? process_num:
        other_num;
    }
  }
stage_count++;
}
```

0.3.2 Theoretical Issues

Theoretical issues with this implementation include the potential for the MPI_Sendrecv be effected by its blocking properties.

0.3.3 Timing Results



The two concentrations of times indicates that some of the processors had to communicate across the nodes. Inter node communications are much faster than intra node communication, and explains why there is a concentration of times at both 0.00017 and 0.067.