HW 2: Basic Parallel Computing Data Reduction and API Calls

Adam Ross

*adro4510@colorado.edu*

POINT OF THIS: BLOCKING RECV and DATA REDUCTION BLOCKING/INEFFICIENT

**Abstract**

*The Paper addresses basic MPI message passing, it’s effects on a program and some of the nuances of moving an algorithm between serial and simple parallel computing with MPI. I first discusses the blocking nature of MPI\_RECV and the effects this has on program flow. Then I compare the differences and benefits of post processing all to one data reduction versus post processing distributed data reduction which leads to how I conceptually compare the overhead and idle time of each method.*

1. **Introduction**

We have a very basic knowledge of MPI at this point and mostly need some experience working with the code itself. Coding with the fundamental building blocks of MPI teaches through experience the top level nature of the MPI library. I examine the OpenMPI RECV call and the semantics about it. I then begin to look at MPI data reduction algorithms and the motivation for further investigation.

1. **Overview**

The openMPI Recv api call is blocking and as such, in order for our program not to block, there must be a send for every receive. Specifically there must be a send before every single receive for our program to not block on some level. This information directs thought to the manner in which we call send and receive and as a repercussion the source and destination management for each of these calls. It needs to be carefully done as not to put too much load on a single process to compile the data and not too complex that we cannot manage our calls and block the program as a whole. Edge cases were also a delicate case that needed though. If your MPI code did not handle it properly then the code would not work.

For example the required trapezoid integral estimation program calculated the sources and destination for the MPI send and receive calls as such:

source = my\_rank + 1;

dest = (my\_rank == 1 ? 0 : (my\_rank - 1));

This means if we have a single process my\_rank = 0. The dest will be calculated (0+1)mod1 = 0. The source will be calculated (0==0) -> True -> (1 - 1) = 0. We will send and receive from the same process which works as intended.

Each process starting from the highest ranked process would send its data to the process one rank below it and sum the data being passed through. This would occur until process 0 summed the final result which was the value of our integral estimation. This enables us to spread the aggregation and receiving work out, increasing efficiency of the program.

1. **Verification**

A program was written that included a send only after a receive call. The program hung and accomplished nothing. The program was then rewritten to send first with destination in a wrapping fashion to the process rank above it. This ran fine and did not block.   
  
 Another program was then written to do a calculation and send data to the process one rank below it to sum data down to the first process rank. This program started with all but one process to be blocking on receive where a single send would trigger a receive send chain to sum the data and distribute the actual summation and receiving work and overhead.

The final program written performed Simpson’s rule given an interval, a more complex operation that requires more per process management and more complicated edge cases. It is verified by computing the edge cases.

Code Output:

**Only one interval across 10 processes (low interval and odd interval amount)**

mpiexec -np 10 parallel\_simpson -i 1 -v

An odd interval was given, which basic simpson's rule does not work with. Adding 1 interval.

[ 1 ] [ 4 ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ 1 ]

The parallel sum for Simpson's rule from 0.000000 to 1.000000 is 0.333333

**Many intervals across one process**

mpiexec -np 1 parallel\_simpson -i 100 -v

[ 1 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 1 ]

The parallel sum for Simpson's rule from 0.000000 to 1.000000 is 0.333333

**Many intervals across moderate processes**

mpiexec -np 8 parallel\_simpson -i 100 -v

[ 1 4 2 4 2 4 2 4 2 4 2 4 2 ] [ 4 2 4 2 4 2 4 2 4 2 4 2 4 ] [ 2 4 2 4 2 4 2 4 2 4 2 4 2 ] [ 4 2 4 2 4 2 4 2 4 2 4 2 4 ] [ 2 4 2 4 2 4 2 4 2 4 2 4 ] [ 2 4 2 4 2 4 2 4 2 4 2 4 ] [ 2 4 2 4 2 4 2 4 2 4 2 4 ] [ 2 4 2 4 2 4 2 4 2 4 2 4 1 ]

The parallel sum for Simpson's rule from 0.000000 to 1.000000 is 0.333333

**Many intervals across many processes**

mpiexec -np 32 parallel\_simpson -i 100 -v

[ 1 4 2 4 ] [ 2 4 2 4 ] [ 2 4 2 4 ] [ 2 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 1 ]

The parallel sum for Simpson's rule from 0.000000 to 1.000000 is 0.333333

1. **Conclusion**

It is clear that in order to optimize highly parallel processing we must also manage and distribute MPI calls and their overhead as well as the aggregation work itself. As was said in class there is basicly no way to completely get rid of all serialized work to achieve a true parallel processing.