Homework 2: Ring Hello World, Trapezoidal Rule, Simpsons Rule

Brandon Bell

***Abstract*—Here I present my latest interludes into parallel programming with MPI. I wrote a Hello World program utilizing a ring style of process communication, modified the reduce method of a Trapezoidal Rule numerical integration program, explored numerical integration using Simpsons Rule in both a serial and parallel format. These projects provided for increasing sophistication in inter-processes communication, parallel algorithm design, and parallel program I/O.**

I. Introduction

Effective inter-process communication is critical to the development of successful parallel programs. MPI allows for this communication with two fundamental functions, MPI_Send() and MPI_Recv(). Care must be taken in the use of MPI_Send() and MPI_Recv() as they proved blocked communication. A process that calls MPI_Recv() waits for a message to be received before continuing execution. If the Sends and Recvs of a program are not well coordinated it can cause inefficiencies due to idle processes or a program a crash.

II. Hello World Ring communication

I modified the Hello World program from HW1 to pass its messages around a ring of processes rather than having every process sending a message to process 0. Process 0 starts by sending a message to process 1, who then prints process 0s message and then sends its own message to process 2 and so on until, the last process sends its message to process 0 to print. This scheme requires that process 0 first send a message and begging listening for a message from the last process. All other processes must first listen for incoming messages before sending their own. This ensures the orderly passing of messages from process 0, through all the other processes and then back to zero, forming a ring. If process 0 was to listen first, then the message chain would not begin and the program would hang. If the other processes sent their messages first, youd have to rely on the systems buffering of messages for orderly delivery and the likelihood of a missed message hanging the program increases. If the program is run with only a single process, process 0 simply prints its own message and the program concludes. When started with 8 process, the program yields,

```
% mpirun -n 8 ./HW2-Part1
Greetings from process 0
Greetings from process 1
Greetings from process 2
Greetings from process 3
Greetings from process 4
Greetings from process 5
Greetings from process 6
Greetings from process 7
```

III. Trapezoidal Rule Integration With Linear Reduce

I simply modified the all the one reduction method of the original Trapezoidal Rule code from the books source code to use a linear reduce. The highest ranked process (p) starts the reduce by sending the to total of its integration region to process (p-1) which, then adds the total of process p to its own and send the new running total to process (p-2) and so until p0. Process 0 receives the running total, adds its region to the total and prints the results. I think this linear reduce is is about as efficient as the orginal all-to-one reduce as they are both scale as O(P). However, in this regime of small latency dominated messages, I think that there may be a small advantage to the all-to-one reduce since the overhead of sending a message is palatalized. When run with 8 processes my program finds,

```
% mpirun -n 8 ./HW2-Part2
With n = 1024 trapezoids, our estimate
of the integral from 0.000000
to 1.000000 = 0.333333
```

which agrees with the finding of the original program when run with 8 processes.

```
% mpirun -n 8 ./get_data
Enter a, b, and n
0 1 1024
With n = 1024 trapezoids, our estimate
of the integral from 0.000000
to 1.000000 = 0.333333
```

IV. Simpson's Rule

I started out by writing a serial version of Simpsons Rule with a fixed function of $f(x) = x^2$, fixed region of integration [0,1], and a fixed number of intervals, 1024. My implementation of Simpsons Rule.

```
// Take care of the starting and end points.
integral = f(a) + f(b);
// Loop through the n strips in the region.
for ( int i=1; i<(2*n - 1); i++ )
{
// increment x by a half interval at a time (we
x += h / 2;
if ( i % 2 == 0 )
integral += ( 2 * f(x) );
else
integral += ( 4 * f(x) );
}
integral *= ( h / 3 );
```

Then to serialize this implementation, I worked this integration function into the the linear reduce version of the Trapezoidal rule code from part 2.

This parallel Simpsons Rule code also incorporates command line arguments to set the number of intervals used and and to print out the interval partition assigned to each process. All of the output is handled by process 0 but, all the processes are aware of any arguments given. This implementation of Simpson's Rule requires that the number of intervals be even so, the code checks the parity of a user defined interval number and if its odd, the code increments it by one to make it even and then carries on with the new even interval count. If the interval value entered is not a valid base 10 int value a warning is issued about incorrect arguments, command line argument parsing stops and the program carries on with the hard coded interval count of 1024. Each process determines a new interval count from the command line arguments, if present, so no time consuming message passing is required.

To implement to the verbose option to print out the interval coefficients being handled by each process, process 0 calculates the partition scheme from the variable of integration and does not depend of communication from the other processes. At present my code has process 0 do this calculation after it has received the total from all the other process. It would probably be more optimal the partitions where determined and and output while process 0 was waiting for the linear reduce to complete, thereby reducing idle process time, but the difference would be minimal at this level I expect. There is a known bug that i've been unabel to track down, setting -v before -i results in the new interval value being read as 0 regardless of specified value. In the case that an interval of 0 is found the code falls back to the hard coded value of 1024.

Examples:

```
% mpirun -n 4 ./HW2-Part3-Parallel
With n = 1024 intervals, our estimate
of the integral from 0.000000 to 1.000000 = 0.664229

% mpirun -n 4 ./HW2-Part3-Parallel -i 11
With n = 12 intervals, our estimate
of the integral from 0.000000 to 1.000000 = 0.480710

% mpirun -n 4 ./HW2-Part3-Parallel -i 11 -v
Process 0: [ 1 4 2 ]
Process 1: [ 2 4 2 ]
Process 2: [ 2 4 2 ]
Process 3: [ 2 4 1 ]
With n = 12 intervals, our estimate
of the integral from 0.000000 to 1.000000 = 0.480710
```