HW 2—ECE/CS 566, S'23

Due: Wed. 3/17/21, 11:59 pm Instructor: Shantanu Dutt

1. Implement the parallel pi computation algorithm discussed in class and given in the appendix, in which instead of the "MPI_Reduce" procedure call, you need to implement the reduction sum operation using explicit recursive-reduction, and **only** MPI_Send and MPI_Recv (or their non-blocking versions MPI_ISend and MPI_IRecv, respectively) communication routines in MPI-C (MPI in the C language) on the UIC Extreme machine. It is alright to use MPI_Bcast. Further, you are to implement the parallel program assuming a hypercube topology, and the final sum should be collected at processor 0 (so no exchange communication is needed).

   For testing purposes, collect results (final pi value and the parallel time) for $n = 2^{20}$, $2^{23}$and 2^{26}, and $P = 1, 2, 4, 8,$ *and* 16—with the following combination of the number of nodes (*nodes* parameter) and cores (*ppn* parameter) in the script file to set up parameters and run your code: (a) *nodes= 1, ppn=P*, (b) *nodes= P, ppn=1* (or if this is not being allocated by the system, then *nodes= P/2, ppn=2*). Note also that since the program runs have to be submitted in batch mode, you need to remove the interactive inputting of the value of *n*, and instead input *n* as a command line argument of the executable.

   At the end of the computation, processor 0 writes out to stdout (the screen) as well as an output file called *par_pi_op_ your1stname.txt*, the value of *P*, *n*, the *nodes*, *ppn* combination (can be indicated by a command line argument as case "a" or "b"), the final result (pi), the error, and the total runtime taken by processor 0, the speedup and the efficiency (speedup/*P*), one per line in an easy to read and well formulated manner in which you mention which parameter/metric value is being given.
   Note: The way to compute the speedup and efficiency in your code is to first run the code for P=1, write out the above metrics in append mode (which you need to do anyway) to the output file, and then for P > 1 runs you can read the sequential time from the corresponding line in the output file (you will know which line it is on, for either case: counting and not counting initial data distr. time---see below for the latter issues).

   Theoretically analyze the parallel time complexity assuming that message-passing time for 1 unit of data is 10x the time for atomic/scalar arithmetic operations, and plot it for different *n* and *P*. Along with the analytical plots, also plot the empirical parallel times for different *n* and *P* for both *nodes, ppn* combinations (a) and (b) given above, and discuss any differences in *runtime trend* between them and between the two empirical trends and the theoretical one (the exact values will most likely not match, since in Theta notation all constants—programming and technology based—are not taken into account, but the trends—slopes or rates of decrease of parallel times with increasing *P* for each *n*, can be compared). In the above reporting and analysis, provide parallel runtimes (analytical and empirical) that: (i) count and (ii) do not count the initial data distribution (broadcast) time. Finally, also run the code in the Appendix that uses "MPI_Reduce", get its results for the above given values of *n* and *P* and for *nodes* and *ppn* combinations for cases (a) and (b) above, write these out as above to file

par_pi_op_simple_your1stname.txt, plot its parallel runtimes as above for both *nodes, ppn* combinations, and compare to the parallel runtimes to those of the corresponding *nodes, ppn* combination of your program using only *send's* and *receive's*.

You need to submit the following: **a)** On BB: A well-prepared report (pdf only and with proper title and author name), labeled *1stname_hw2_rep_566_s23.pdf*, describing your algorithm (along with a pseudo code) and figures, and presenting the results as mentioned above. Also, include in this report a description of how you've simulated a hypercube (essentially how a processor with integer rank, say, my_id, determines in each round of the reduction operation, which hypercube processor to communicate with; put this description in a section with tittle "Simulating a Hypercube". **b)** submit the following files to the submit directory on Extreme that will be setup by their admin: your source code, executable, makefile (to compile the executable), output files, and script file (for running the executable) appropriately labeled. Your program and executable should be labeled "*par_pi_1stname*" for the code that **does not use** MPI_Reduce and "*par_pi_simple_1stname*" for the code **using** MPI_Reduce. **c)** Besides submitting the above to the submit directory on Extreme, submit the code, makefile and script files [but not the executable] on BB in a .tar file labeled *1stname_hw2_codes_566_s23.rar*.
You will be asked to demo the running and parallel time determination of your code during a subsequent office hours setting or any other mutually convenient time. **500 pts**

Appendix: Parallel PI MPI_C Program using MPI_Bcast and MPI_Reduce

```
#include "mpi.h"
#include <math.h>
int main(argc,argv)
int argc; char *argv[];
{
 int done = 0, n, myid, numprocs, i, rc;
double PI25DT = 3.14159265358979323846264 3;
double mypi, pi, h, sum, x, a;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
if (myid == 0) { printf("Enter the number of intervals: (0 quits) ");      scanf("%d",&n); }
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n >  0)
{
h = 1.0 / (double) n; sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs)
{
x = h * ((double)i - 0.5);      sum += 4.0 / (1.0 + x*x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0) printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
}
MPI_Finalize();
}
```

2. (Optional bonus point problem): Change the above given parallel pi code (using MPI_Bcast and MPI_Reduce) so that each processor sums the same elements as in the above for loop, but in completely reverse order, i.e., the for loop going from $i = n - k$ to $P - k$ where k is particular to the processor id (`myid`) and P = the # of processors (`numprocs` in the code). This can help with reducing floating-point rounding error when performing the sum computation. This needs to be done for the following parameter values: $n = 2^{20}$, $2^{23}$and 2^{26}, and $P = 1, 2, 4$ *and 8,* with *nodes= P, ppn=1.* Then in a well-organized table, compare the pi values and errors for each of the above parameter combinations (basically, n and P) to the pi values and errors of the above given parallel pi code (that you have run as part of problem 1) for the same parameter combinations, and draw some meaningful conclusions on any differences and what it means about whether the reverse order of summation helps in improving floating-point roundoff error accuracy.

   Submit similar files as in problem 1 but with an added suffix of *reverse* in each file name, e.g., the report file for this problem should be *1stname_hw2_rep_566_s23-reverse.pdf*. **100 pts**