Homework #2 Assignment
EEL6763 Parallel Computer Architecture
Spring Semester 2016

Part A: Monte-Carlo Integration and MPI

1. One method of numerically estimating integrals is by using Monte-Carlo simulation. Consider the following integral g(a,b) and its estimate h(x):

$$g(a,b) = \int_a^b f(x)dx \qquad h(x) = \frac{(b-a)}{N}\sum_{i=1}^{N} f(x_i)$$

The numerical solution to g(*a*,*b*) can be estimated using a uniform random variable *x* that is evenly distributed over [*a*, *b*]. The estimate h(*x*) will converge to the correct solution as the number of samples *N* grows. Since each sample is independent, the calculation can be easily parallelized. Write a short MPI program that will use many samples to calculate:

$$\int_a^b \frac{8\sqrt{2\pi}}{e^{(2x)^2}}\,dx$$

using this method. For this problem you must provide code for two functions, estimate_g(…) and collect_results(…), which will be used by the following main(…) function:

```
int main(int argc, char **argv)
{
  MPI_Init(&argc,  &argv);

  float lower_bound  = atof(argv[1]);
  float upper_bound  = atof(argv[2]);
  long long int N = atof(argv[3]);

  initialize_data();

  result = estimate_g(lower_bound, upper_bound, N);
  collect_results(&result);

  MPI_Finalize();
  return 0;
}
```

- The following function prototypes must be used for your functions:
  - double estimate_g(double lower_bound, double upper_bound, long long int N);
  - void collect_results(double *result);
- The total number of samples to calculate *N* (to be split among all MPI nodes) as well as the bounds of the integral *a* and *b* will be provided through command-line arguments.
- Every MPI node will generate its own random numbers. Ensure that each node uses a different starting seed for its random number generator. This should be the only thing that occurs in initialize_data(…).
- Each node should return a single value, which should be combined on the root node in order to compute the final integral.

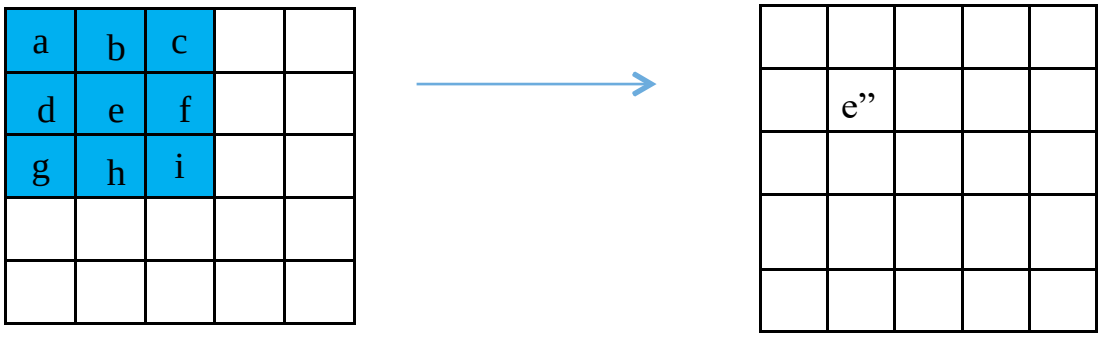
- Use only the following functions:
  - MPI_Init
  - MPI_Comm_rank
  - MPI_Comm_size
  - MPI_Send
  - MPI_Recv
  - MPI_Finalize
- Your code should be submitted as a file named **hw2_a1.c**.

2. Rewrite the previous functions to use MPI_Reduce instead of MPI_Send and MPI_Recv. Submit this version of your code as **hw2_a2.c**.

3. Comment on the performance of both programs. Use several different numbers of samples, *N*. Additionally, vary the system size (number of MPI nodes, or cores) from 1 to 32. This type of code is sometimes called embarrassingly parallel. Why is that?

Part B: Matrix Mask Operation

1. A Mask operation (neighborhood weighted-averaging filter) is commonly used for image processing. The basic concept is to recalculate the value of each pixel on the basis of the adjacent pixel values and value of the current pixel. For our purposes consider a grayscale image as a matrix with each pixel having a value of 0 to 10000. A mask operator is defined over a matrix as shown in the figure. The operator takes the average of adjacent matrix values defined as
   e" = (a+b+c+d+2e+f+g+h+i)/10.



BEFORE PROCESSING (A) AFTER PROCESSING (Ap)

- Implement the mask operation on a Matrix using MPI.
- The operation is performed on a square matrix of size NxN.
- The matrix must be initialized in one of the nodes using rand() function.
- Use Scatterv / Gatherv along with any other MPI functions required
- To keep program simple, it is not required to process the first and last rows and columns.
- All MPI nodes must perform the mask operation.

- You will provide code for three functions (distribute_data, compute_mm, and collect_data) with the following prototypes:
  void scatter_data(int *A, int N);
  void mask_operation(int *A, int N, int *Ap);
  void gather_results(int *Ap, int N);
  *Ap stands for processed matrix

- Use the following pseudocode for your main function. N is size of Matrix NxN

```
int main(int argc, char **argv)
 {
  MPI_Init(&argc,  &argv);
  int N = atof(argv[1]);
  initialize_data(&A,&Ap);
    scatter_data(A,  N);
    mask_operation(A,  N, Ap);
    gather_results(Ap,  N);
  MPI_Finalize();
  return 0;

 }
```

- Submit the code as **hw2_b1.c**

2. Quantitatively analyze the performance and scalability of the program that you just created. How does performance scale with system size (number of MPI nodes, or cores) from 1 to 32? How does performance scale with problem size?

Submission guidelines

- Submit a single PDF containing answers through Canvas.
- Can be done **in teams 1 or 2 students.**
- Only ONE team-member has to submit PDF using Canvas (NOT BOTH)
- Your answers should include source code (copy-paste the complete source code from your source files), timing information from experiments, and your observations.
- Your PDF must contain the code corresponding to **hw2_a1.c, hw2_a2.c, hw2_b1.c**
- Please make sure source code is consistently indented and commented.
- All code and answers MUST be your own, original, and NOT copied from the Internet or elsewhere**.**