CS 635 - Spring 2012
Project 2
This homework assignment is due by 10 pm on April 20, 2012.

SampleSort Using MPI and OpenMP

Background

One of the parallel sorting algorithms that we studied earlier this semester was called SampleSort. If you recall (see our lecture notes and textbook, pp. 412–213), the algorithm was targeted for a distributed memory message passing environment and can be classified as a parallel bucket sort.

Several computation and collective communication steps are required for this distributed memory parallelization– these steps determine splitter points and the use them sort the input data:

1. Partition the n data items into p blocks of size n/p, one block per computing node
2. **Sort** each block locally on all nodes
3. Choose p evenly spaced values in each block
4. Collect the whole set of p^2 values to form the sample
5. Sort the sample of p^2 values and determine $p-1$ evenly spaced splitters from it
6. Each compute node does a **binary search** for each of the $p-1$ splitters within its already sorted local data to determine the beginning and end of the p blocks
7. Each node sends its $p-1$ blocks to the appropriate nodes, keeping its own designated block
8. Each node **merges** the incoming and resident data to form a single local sorted list.

Your Tasks

The goal of this project is to explore the application of the MPI distributed memory message-passing paradigm with OpenMP threads. This includes experimental evaluation that compares the combined MPI-OpenMP algorithm with MPI-only or OpenMP-only implementations. Please be aware that this means you will need to write *at least two* parallel codes for this project.

- The MPI library routines are to be used to pass messages between tasks that are running on separate nodes in the Medusa cluster. Note you can find a skeleton version of the MPI code in our textbook, pages 270-272.)
- You must use the multi-threading directives of OpenMP to use threads on each individual cluster node to accomplish the local **binary search** and **merging tasks** outlined in steps (6) and (8). Note that this may involve *more* than just randomly inserting OpenMP loop pragmas if concurrent reads or writes are required.

  Ideally, the local sorting of step (2) should also be implemented using multiple threads. This requires writing your own local multi-threaded sorting algorithm. You can use OpenMP to parallelize any sorting algorithm that you like, including a sequential version of SampleSort or Insertion Sort, instead of parallelizing quicksort which the class and textbook version of SampleSort use. Beware of the use of multi-threading and recursion.

- You will need to perform quantitative experiments to measure the total time used by your two programs for the following values of n, p and t for data set size, number of message passing nodes, and number of local threads per node, respectively. (You can also use larger data sets.)

Data set: n	Nodes: p	Local Threads: t
64	4 & 8	4 & 8
256	4 & 8	4 & 8
1024	4 & 8	4 & 8
4096	4 & 8	4 & 8
16384	4 & 8	4 & 8

  Note that the Medusa nodes are dual dual-core processors; however, you can use virtual threads in OpenMP, so it is suggested that you try your implementations using 4 and 8 threads per node to investigate the costs associated with virtual threads in OpenMP.

  As before, make sure you do not include the I/O times (i.e. scanf, printf commands) within your timing measurements.

- Interpret your MPI-OpenMP hybrid program results and compare them to at least one MPI-only or OpenMP-only implementation of SampleSort.

  This includes a display of your quantitative results in table form: for example,

  - For each fixed n and p, list the measured and estimated total execution time as a function of t.
  - For each fixed n and t, list the measured and estimated total execution time as a function of p.
  - For each fixed p and t, list the measured and estimated total execution time as a function of n.

  What appears to be the best choices for n and p?

Turn In:

- A Project Report that includes:
  - a brief overview of your MPI-OpenMP implementation that includes the following:
    - A description and justification of your data distribution and communications strategies for the MPI aspects of your program
    - A description and justification of the multi-threading strategy you used in the OpenMP parts of your implementation
  - A summary of the pure MPI-only or OpenMP-only parallelization of SampleSort against which you compared your MPI-OpenMP implementation.
  - Quantitative evidence as to whether or not you observed speedups in your MPI-OpenMP implementation. You should compare your results to the pure MPI and/or OpenMP codes. Your summary should include a discussion of where/when bottlenecks that affected performance occurred.
  - Output results from an execution of your program with $n = 64$, $p = 4$, and $t = 4$ to verify that your implementation is correct. The initial input array should contain an input list of random data. Your program output should also include the measured times for communication and computation.

- Source code:
  - A copy of your MPI-OpenMP hybrid SampleSort code.
  - A copy of your MPI-only or OpenMP-only SampleSort code.
  - All listings should have short comment sections delineating the major computation and communication steps of your implementations. Include any special instructions needed for compiling and/or executing your code on Medusa.

  Make sure your code runs on the Medusa cluster if you developed it somewhere else. It will be tested.

- Electronic submission: Your Project Report and your source code should be uploaded to the Moodle website *before* the due date/time.

  Incomplete projects will likely be assigned low grades. Please consult the Project Score Sheet (available on Moodle) to see how points will be allocated.

  Start early! Late projects will not be accepted.

Additional Information

There are two sample programs attached below that combine MPI and Open-MP. Remember that the command to compile such programs on Medusa nodes would be:

```
mpicc -o foo foo.c -fopenmp
```

Sample code 1

```
#include "mpi.h"
#include <stdio.h>
#include "omp.h"
int main(argc, argv)
        int argc;
        char *argv[];
{
        int rank, size;
        MPI_Init(&argc, &argv); /*Initialize MPI State */
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        #pragma omp parallel num_threads(8)
        {
        int thread= omp_get_thread_num();
        printf ("Hello World! I am thread %d of process %d of %d\n",
            thread, rank, size);
        }
        MPI_Finalize(); /*Clean up MPI State */
        return 0;
}
```

Sample code 2

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include "omp.h"

#define CHUNKSIZE 10
#define N 100

void openmp_code() {
   int nthreads, tid, i, chunk;
   float a[N], b[N], c[N];

   for (i=0; i< N; i++) a[i] = b[i] + i*1.0;
   chunk = CHUNKSIZE;
   #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
   {
      tid= omp_get_thread_num();
      if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf ("Number of threads= %d\n", nthreads);
      }
      printf ("Thread %d starting\n", tid);

      #pragma omp for schedule(dynamic,chunk)
      for (i=0; i< N; i++) c[i] = a[i] + b[i];
```

```
   }
}

int main(int argc, char **argv) {
        char message[20];
        int i, rank, size, type=99;;
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

       if (rank == 0) {
          strcpy(message, "Hello, world");
          for (i=1;i<size;i++)
             MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
       }
       else  MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);

       if (1) { //* do if this machine has multiple cores
          openmp_code();
       }

       printf ("Message from process =%d : %.13s\n", rank, message);
       MPI_Finalize(); /*Clean up MPI State */
}
```