Homework #2

(Due: April 6)

Task 1. [250 Points] Distributed Sample Sort with Multithreaded Merge Sort inside Each Compute Node.

This task asks you to implement a distributed sample sort algorithm that runs a multithreaded merge sort algorithm inside each compute node. You will be sorting double precision floating point numbers. We will assume that all numbers in the input array are distinct. For simplicity generate the input array as follows. First initialize the array by storing the number i (as a double precision float) at location i of the array, where $1 \leq i \leq n$. Then choose two random locations of the array and swap their contents, and repeat this swaping step n times. Use this permuted array as input to your sorting implementations. When timing the implementations do not include the time needed to generate the input array.

(a) [**30 Points**] Implement and optimize the shared-memory parallel merge sort algorithm (Par-Merge-Sort-SM) shown in Figure 1 which uses the standard serial algorithm (Merge) for merging two sorted sequences.

Optimize for the base case size m_1, that is, empirically find a value of m_1 that gives you the best or close to the best performance for large values of n, and use that value of m_1 in all subsequent runs of the algorithm. For this task it is OK to check only powers of 2 for the potential best value of m_1. Use all cores during this base case optimize phase.

Let N be the largest power of 2 such that this algorithm can sort N numbers in less than 5 minutes when run on a single processing core. Find the value of N.

(b) [**30 Points**] Implement and optimize the shared-memory parallel merge sort algorithm (Par-Merge-Sort-PM) of Figure 2 which uses a parallel merge algorithm (Par-Merge).

First optimize for the base case size m_2 of Par-Merge, and then use that value of m_2 when you optimize for the base case size m_3 of Par-Merge-Sort-PM.

(c) [**30 Points**] Plot the running times of Par-Merge-Sort-SM and Par-Merge-Sort-PM for 10 equispaced points between $n = 1000$ and $n = N$ when run on all cores of the compute node you are using. Compare and explain the results.

(d) [**30 Points**] Generate Cilkview scalability plots for Par-Merge-Sort-SM and Par-Merge-Sort-PM assuming $n = N$. Compare and explain the results.

(e) [**50 Points**] Implement the distributed-memory parallel sample sort algorithm (Distributed-Sample-Sort) shown in Figure 3. Make two implementations. In one implementation use the merge sort algorithm (Par-Merge-Sort-SM) you implemented in part (a) for the local sorts in steps 2 and 5, and call this Distributed-Sample-Sort-SM. In another implementation use the merge sort algorithm (Par-Merge-Sort-PM) you implemented in part (b) and

```
MERGE( A, p, q, r )                    {Merge sorted segments A[p : q] and A[q + 1 : r], and store in A[p : r].}
1. n1 ← q − p + 1,  n2 ← r − q
2. create array L[1 : n1 + 1] and R[1 : n2 + 1]
3. L[1 : n1] ← A[p : q],  R[1 : n2] ← A[q + 1 : r]
4. L[n1 + 1] ← ∞,  R[n2 + 1] ← ∞
5. i ← 1,  j ← 1
6. for k ← p to r do
7.     if L[i] ≤ R[j] then A[k] ← L[i],  i ← i + 1
8.     else A[k] ← R[j],  j ← j + 1

PAR-MERGE-SORT-SM( A, p, r )           {Sort the numbers in A[p : r]. All numbers in A are assumed to be distinct.}
1. n ← r − p + 1
2. if n ≤ m1 then                      {m1 is the global base case size for PAR-MERGE-SORT-SM}
3.     sort A[p : r] using insertion sort
   else
4.     q ← ⌊(p + r)/2⌋
5.     spawn PAR-MERGE-SORT-SM( A, p, q )
6.           PAR-MERGE-SORT-SM( A, q + 1, r )
7.     sync
8.     MERGE( A, p, q, r )             {merge A[p : q] and A[q + 1 : r], and store in A[p : r]}
```

Figure 1: Parallel merge sort with serial merge.

call this DISTRIBUTED-SAMPLE-SORT-PM. In DISTRIBUTED-SAMPLE-SORT-SM, execute the two recursive calls to PAR-MERGE-SORT-SM in steps 5–7 of PAR-MERGE-SORT-SM serially (i.e., remove the ***spawn*** and ***sync*** keyword) instead of calling them in parallel. But do not change anything inside the PAR-MERGE-SORT-PM implementation used in DISTRIBUTED-SAMPLE-SORT-PM.

(f) [**40 Points**] Plot the running times of DISTRIBUTED-SAMPLE-SORT-SM and DISTRIBUTED-SAMPLE-SORT-PM for 10 equispaced points between $n = 1000$ and $n = N$ when run on 5 compute nodes. For DISTRIBUTED-SAMPLE-SORT-PM use all cores inside each compute node. Compare and explain the results. Also plot the speedups w.r.t. PAR-MERGE-SORT-SM run on a single core of a single compute node.

(g) [**40 Points**] Repeat part (f), but without including the time needed for the initial distribution of input (step 1 of DISTRIBUTED-SAMPLE-SORT) and the final collection of output (step 6) in the running time.

```
PAR-MERGE( T, p1, r1, p2, r2, A, p3 )   {Merge disjoint sorted segments T[p1 : r1] and T[p2 : r2], and store
                                         result in A[p3 : r3], where r3 = p3 + n1 + n2 − 1, n1 = r1 − p1 + 1,
                                         and n2 = r2 − p2 + 1. All numbers in T are assumed to be distinct.}
1.  n1 ← r1 − p1 + 1,  n2 ← r2 − p2 + 1,  r3 = p3 + n1 + n2 − 1
2.  if n1 + n2 ≤ m2 then                 {m2 is the global base case size for PAR-MERGE}
3.      use the standard serial merge algorithm to merge T[p1 : r1] and T[p2 : r2], and store result in A[p3 : r3]
    else
4.      if n1 < n2 then p1 ↔ p2,  r1 ↔ r2,  n1 ↔ n2
5.      q1 ← ⌊(p1 + r1)/2⌋
6.      q2 ← BINARY-SEARCH( T[q1], T, p2, r2 )   {each number in T[p2 : q2 − 1] is smaller than T[q1]
                                                   and each in T[q2 : r2] is larger than T[q1]}
7.      q3 ← p3 + (q1 − p1) + (q2 − p2)
8.      A[q3] ← T[q1]
9.      spawn PAR-MERGE( T, p1, q1 − 1, p2, q2 − 1, A, p3 )
10.             PAR-MERGE( T, q1 + 1, r1, q2 + 1, r2 − 1, A, q3 )
11.     sync

PAR-MERGE-SORT-PM( A, p, r )   {Sort the numbers in A[p : r]. All numbers in A are assumed to be distinct.}
1.  n ← r − p + 1
2.  if n ≤ m3 then                       {m3 is the global base case size for PAR-MERGE-SORT-PM}
3.      sort A[p : r] using insertion sort
    else
4.      q ← ⌊(p + r)/2⌋
5.      spawn PAR-MERGE-SORT-PM( A, p, q )
6.             PAR-MERGE-SORT-PM( A, q + 1, r )
7.      sync
8.      T[p : r] ← A[p : r]              {T is a global array of the same size as A}
9.      PAR-MERGE( T, p, q, q + 1, r, A, p )   {merge T[p : q] and T[q + 1 : r], and store in A[p : r]}
```

Figure 2: Parallel merge sort with parallel merge.

DISTRIBUTED-SAMPLE-SORT(n, p, q) *{Sort n distinct numbers using p processors. One node contains all n input numbers initially, and also the numbers in the final sorted order.}*

1. **Initial Distribution:** The master node scatters the n keys among p processing nodes as evenly as possible.
2. **Pivot Selection:** Each node sorts its local keys, and selects $q-1$ evenly spaced keys from its sorted sequence. The master node gathers these local pivots from all nodes, locally sorts those $p(q-1)$ keys, selects $p-1$ evenly spaced global pivots from them, and broadcasts them to all nodes.
3. **Local Bucketing:** Each node inserts the global pivots into its local sorted sequence using binary search, and thus divides the keys among p buckets.
4. **Distribute Local Buckets:** For $1 \leq i \leq p$, each node sends bucket i to node i.
5. **Local Sort:** Each node locally sorts the elements it received in step 4.
6. **Final Collection:** The master node collects all sorted keys from all nodes, and for $1 \leq i < p$, places all keys from node i ahead of all keys from node $i+1$.

Figure 3: Distributed sample sort.

APPENDIX 1: Calling Cilk++ Functions from MPI Code

ncr.cilk

```
#include <cilk.h>

int nCr( int n, int r )
{
   if ( r > n ) return 0;
   if ( ( r == 0 ) || ( r == n ) ) return 1;

   int x, y;

   x = cilk_spawn nCr( n - 1, r - 1 );
   y = nCr( n - 1, r );

   cilk_sync;

   return ( x + y );
}

extern "C++" int nCr_CPP( int n, int r )
{
   return cilk::run( nCr, n, r );
}
```

ncr-mpi.cpp

```
#include <mpi.h>

extern "C++" int nCr_CPP( int n, int r );

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

  int rank;
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  printf( "C( %d, %d ) = %d\n", 30, 15 + rank,
          nCr_CPP( 30, 15 + rank ) );

  MPI_Finalize( );

  return 0;
}
```

In `ncr.cilk` we have a `Cilk++` function called `nCr` which we would like to call from within the `MPI` code `ncr-mpi.cpp`. Since we do not have a `cilk_main` function in `ncr-mpi.cpp`, we do not have a `Cilk++` context, and so `nCr` cannot be called directly from within `ncr-mpi.cpp`. Instead we create a function (named `nCr_CPP`) callable from `C++` which starts a `Cilk++` environment through

`cilk::run` and calls `nCr`.

You can compile and link the files as follows on Lonestar/Stampede. The first command creates a shared library named `libncr.so` from `ncr.cilk`, and the second one compiles `ncr-mpi.cpp` and links it with `libncr.so`.

```
cilk++ -m64 -fPIC -shared -o libncr.so ncr.cilk
mpicxx ncr-mpi.cpp -L. -L$CILKHOME/lib64 -Wl,-rpath=.
       -lncr -lcilk_main -lcilkrts -lcilkutil
```

The resulting `MPI` program (`a.out`) can be run as follows (from your job script).

```
ibrun tacc_affinity a.out
```

If you want to run your `MPI` program on t compute nodes on Lonestar, and launch $k \in \{1, 2, 3, 4, 6, 12\}$ parallel processes on each node, then include the following line in your job script with $m = 12t$.

```
#$ -pe kway m
```

If k parallel processes are launched on each node, then `Cilk++` functions called from each process will be able to launch at most $12/k$ concurrent threads. Recall that when multiple processes are launched on the same node then the total memory is divided among the processes and no process is able to access the memory allocated to other processes, but all threads running under a process share the memory allocated to that process.

APPENDIX 2: What to Turn in

One compressed archive file (e.g., zip, tar.gz) containing the following items.

- Source code, makefiles and job scripts.
- A PDF document containing all answers and plots.

APPENDIX 3: Things to Remember

- **Please never run anything that takes more than a minute or uses multiple cores on TACC login nodes.** TACC policy strictly prohibits such usage. They reserve the right to suspend your account if you do so. All runs must be submitted as jobs to compute nodes (even when you use Cilkview or PAPI).
- Please store all data in your work folder ($WORK), and not in your home folder ($HOME).
- When measuring running times please exclude the time needed for reading the input and writing the output. Measure only the time needed by the algorithm.