Homework #3

(Due: May 5)

Task 1. [30 Points] Shared-Memory Quicksort.

(a) [**5 Points**] Implement the parallel prefix sums algorithm covered in the class (see slides 10–14 of lecture 7) in `Cilk++`. Optimize your code as much as possible (e.g., memory allocation, space usage, base case size, etc.).

(b) [**5 Points**] Use your parallel prefix sums implementation from part (a) to implement the parallel partitioning algorithm covered in the class (see slides 1–10 of lecture 9). Optimize your code.

(c) [**10 Points**] Use your implementation of the parallel partitioning algorithm from part (b) to implement the randomized looping quicksort algorithm (Par-Randomized-Looping-Quicksort) from Task 3 of HW1. Check if instead of recursing down to arrays of size 1, stopping as soon as the array size becomes smaller than some base case size and using serial insertion sort[1] to sort that small array reduces running time. If so, find and report the base case size that gives you the best running time.

(d) [**5 Points**] For each test case (see Appendix 2) report the running time of your quicksort algorithm from part (c) when run on a single core as well as on all cores. Also report the speedup values.

(e) [**5 Points**] Replace your parallel partitioning function with a serial partitioning function[2] in your implementation of Par-Randomized-Looping-Quicksort. Now compare this implementation with your original implementation (with parallel partitioning) in part (c). Report the running times of both algorithms when run on all cores. Explain your findings.

Task 2. [70 Points] Distributed Sample Sort with Shared-Memory Quicksort.

(a) [**15 Points**] Consider the distributed sample sort algorithm given in Figure 1 where each process uses a shared-memory parallel sorting algorithm for sorting local keys. Prove that in step 5 each process sorts at most $\frac{n}{p} + \frac{n}{q}$ keys in the worst case provided $p, q \geq 1$ and $n \geq p^2q^2$.

(b) [**25 Points**] Implement the algorithm given in Figure 1 using `MPI` and `Cilk++`. Appendix 1 tells you how to run `MPI` over `Cilk++`.

(c) [**10 Points**] Analyze the computation and communication complexities of your implementation in part (b).

[1] see Chapter 2, Section 2.1 of "Introduction to Algorithms", 3rd Ed., by Cormen et al.

[2] use the algorithm on p. 171 or the one on p. 185 of "Introduction to Algorithms", 3rd Ed., by Cormen et al.

DISTRIBUTED-SHARED-MEMORY-SORT($A[1:n],\ q$)

(Input is an array A of n distinct numbers, and a parameter q used for deciding the number of local pivot elements to be selected by each process. The output is A in sorted order.)

1. **Initial Distribution:** The master process distributes the n keys among all p processes (including itself) as evenly as possible.
2. **Pivot Selection:** Each process sorts its own set of keys using the shared-memory quicksort algorithm from part (c) of Task 1, and selects $q-1$ evenly spaced keys as *local pivots* from this sorted sequence. The local pivots divide the sorted local keys into q segments of equal (or almost equal) length. Each process sends its local pivots to the master process which sorts these $p(q-1)$ keys using the shared-memory quicksort, and selects $p-1$ evenly spaced pivot elements from the sorted sequence. The master process broadcasts these *global pivot* elements to all processes.
3. **Local Bucketing:** Each process inserts the global pivots into its local sorted sequence using binary search, and thus divides its local keys into p buckets.
4. **Distribute Local Buckets:** For each $i \in [1, p]$, each process sends its i-th bucket to process i.
5. **Local Sort:** Each process sorts the keys it received in step 4 using the shared-memory quicksort algorithm from part (c) of Task 1.
6. **Final Collection:** The master process collects the sorted keys from all processes.

Figure 1: Distributed sample sort using shared-memory quicksort for local sorting.

(d) [**10 Points**] Assuming $q = kp$ for integer $k \geq 1$, find and report the value of k that gives you the best running time for your implementation in part (b). Use three compute nodes[3] and one process per node for this optimization. Include a plot showing how the running time changes as k varies.

(e) [**10 Points**] For each test case (see Appendix 2) report the running time of your implementation from part (d) when run on three compute nodes but the number of processes per node is varied (chosen from $\{1, 2, 6, 12\}$). Report running times with and without including steps 1 (Initial Distribution) and 6 (Final Collection) in all cases. Explain your findings.

[3] as the "development queue" on Lonestar does not allow the use of more than 3 compute nodes in parallel, and the "normal queue" has a long scheduling delay

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. 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: Input/Output Format

- **Input Format:** The first line of the input file will contain the number of integers (n) in the file. Each of the next n lines will contain one integer. All integers in the file will be distinct.
- **Output Format:** The output will consist of n lines. Line $i \in [1, n]$ will contain the i-th integer in sorted (increasing) order.
- **Test Input:** Folder `/work/01905/rezaul/CSE613/HW3/test` on Lonestar.

APPENDIX 3: What to Turn in

Please email one compressed archive file (e.g., zip, tar.gz) containing the following items to `cse613hw@cs.stonybrook.edu`.

- Source code, makefiles and job scripts for both tasks.
- A PDF document containing all answers.

APPENDIX 4: 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.
- 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.