Homework #2

(Due: February 26)

Task. [**50 Points**] Consider the following 3D order-$2k$ stencil (also known as a $(6k+1)$-point stencil) used for 3D finite difference computations, where $c_0, c_1, \ldots, c_k$ are prespecified constants.

$$d_{t+1}(x,y,z) = c_0 d_t(x,y,z) + \sum_{i=1}^{k} c_i \left(\begin{array}{r} d_t(x-i,y,z) + d_t(x+i,y,z) \\ +d_t(x,y-i,z) + d_t(x,y+i,z) \\ +d_t(x,y,z-i) + d_t(x,y,z+i) \end{array} \right)$$

Given positive integers n_x, n_y, n_z and t_{final}, and values of $d_0(x,y,z)$ for all $\langle x,y,z\rangle \in [1,n_x] \times [1,n_y] \times [1,n_z]$, our goal is to compute all $d_{t_{final}}(x,y,z)$. We assume that $d_t(x,y,z) = 0$ when $\langle x,y,z\rangle \notin [1,n_x] \times [1,n_y] \times [1,n_z]$ and $t \geq 0$. In this homework, we will also assume for simplicity that $k = 3$ and $n_x = n_y = n_z = n = 2^q$ for some integer $q \geq 0$.

(a) [**5 Points**] Implement an iterative algorithm (using serial ***for*** loops) for solving the problem on a serial machine. Analyze its running time T. [*see Appendix 2 for input and output formats*]

(b) [**5 Points**] Modify your implementation in part (a) for running on shared-memory parallel machines (e.g., using `cilk++`). Analyze its running time T_p on p cores. [*Hint: simply use parallel* ***for*** *loops*].

(c) [**25 Points**] Implement an algorithm for solving the problem on distributed-memory parallel machines (e.g., using `MPI`). Analyze its computation complexity t_{comp} and communication complexity t_{comm}. [*Hint: follow the simple idea used in lecture 6 for parallelizing the 2D heat equation*]

(d) [**10 Points**] Combine your implementations in parts (b) and (c) to solve the problem on distributed shared-memory parallel machines, e.g., on a network of multicores. Analyze t_{comp} and t_{comm}. [*Hint: see Appendix 1 to get an idea on how to run MPI on top of Cilk*]

(e) [**5 Points**] Find the largest value of n (say, n_{max}) for which your implementation in part (a) terminates in less than 15 minutes (assuming $t_{final} = n$). Report the running times of your implementations in parts (a), (b), (c) and (d) for n_{max} and $t_{final} = n_{max}$. For parts (c) and (d) run your code on a network of 4 compute nodes, that is, use "-pe 12way 48" in your jobscript on `Lonestar` and "-pe 16way 64" on `Ranger`.

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. First create a shared library named `libncr.so` from `ncr.cilk`, and then compile `ncr-mpi.cpp` and link 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 and Output Formats

The input will start with a line containing two positive integers: n and t_{final}. The second line will contain four floating point numbers giving the values of c_0, c_1, c_2 and c_3, respectively. The next n^2 lines will contain n floating point numbers each. The i-th ($1 \leq i \leq n^2$) such line will contain values for $d_0(x, y, 1), d_0(x, y, 2), \ldots, d_0(x, y, n)$, where $x = \left\lceil \frac{i}{n} \right\rceil$ and $y = i - n(x - 1)$.

The output will contain n^2 lines giving the values of $d_{t_{final}}(x, y, z)$ for $x, y, z \in [1, n]$ in exactly the same format as d_0 in the input.