Homework 3

Assignments: All parts are to be solved individually (turned in electronically, no ZIP files or tarballs, written parts in ASCII text, NO Word, postscript, PDF etc. permitted unless explicitly stated). All filenames must be exactly what is requested, capitalization included.

Please use the ARC cluster for this assignment. All programs have to be written in C, translated with mpicc/gcc and turned in with a corresponding Makefile.

1. (50 points) Analyze MPI call statistics and devise a method through PMPI instrumentation to generate a communication matrix.
   - (0 points) Familiarize yourself briefly with the LULESH 2.0 Documentation and download the source code of Lulesh V2.0.3. Run the following commands to compile and execute:

     ```
     srun -N8 -n64 -popteron --pty /bin/bash
     tar xvzf lulesh2.0.3.tgz
     export OMP_NUM_THREADS=1
     make
     prun ./lulesh2.0 -s 80 -i 20
     ```

   - (45 points) Write PMPI wrappers (see slide 55 of MPI lecture) in a module named **pmpi.c** to count the number of MPI events that occur in the Lulesh benchmark. Record these values in a communication matrix indexed by MPI rank for [source, destination] of the message.
     - edit **Makefile**
     - update the **SOURCES2.0** variable to include **pmpi.c**

       Hints:

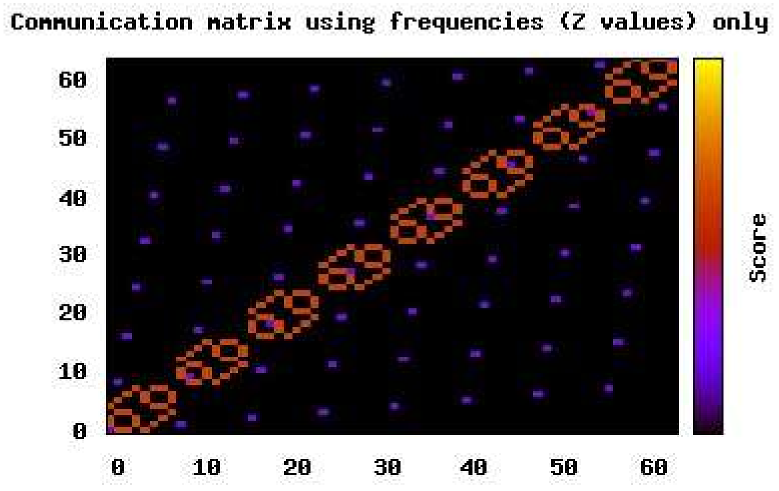
       - Write wrappers for Init, Send, Isend, and Finalize.
       - Use your wrappers to:
         - Call the respective PMPI routines
         - Count the number of times the respective MPI event was called
         - Set up your instrumentation environment (to inquire about your rank etc.)
         - Collect all MPI event counts from remote nodes
       - Rank 0 should print out the entire matrix into a file called **matrix.data**, in the format:

         ```
         [rank i] [No. of Send events to rank 0] [No. of Send events to rank 1] ... [No. of Send
             events to rank n-1]
         ```

   - (5 points) Generate a heat plot.
     - Capture the communication matrix for Lulesh with 64 MPI tasks and -s as 80 and -i as 20

       ```
       gnuplot heatmap.plot
       ```

- Visualize your output. Here are the results for another benchmark, **yours will not look the same**:



Explain your graph, and put your discussion in the **p1.README** file. A (not exhaustive) list of questions to consider are:

- Describe the communication pattern. Where is most of the communication happening (number of calls)?
- What symmetries do you expect to see for Send counts? Are these symmetries observed? Why or Why not?

Hints:

- Each processor should maintain one vector that counts Sends to each node in the communicator. During your MPI_Finalize(), you will gather these vectors onto the root node to form a matrix, and output that matrix.
- The gnuplot script outputs the 2D plot.

Turn in **pmpi.c**, **matrix.data**, and **p1.README**

2. (35 points) Modify the Lake program from HW2 to parallelize for OpenMP.

   An updated version of the serial lake program can be found here. Please use for this assignment. I have also included sample Makefile's for the serial version, OpenMP and OpenACC (next problem). Do NOT modify or submit these Makefile's. I will be using these **exact** Makefile's to grade your submissions!

   - Info on the new code
     - The new Lake now does the entire calculation in **run_sim** (evolve has been removed)
     - All simulations use the same initial pebbles given by the change:

```
//line 277 in original lake.c
void init_pebbles(/*...*/)
{
//...
 srand( time(NULL) );
//...
}
```

to

```
void init_pebbles(/*...*/)
{
//...
 srand( 10 );
//...
}
```

- The new Lake code has the routines

  ```
  start_lake_log(char* logfile);
  lake_log(char *msg, ...);
  stop_lake_log();
  ```

  These routines output a series of logging messages to the file *logfile*. Running times and debugging information are placed in this file.

- Furthermore, all output files (*logfile*,*lake_i/f.dat*) will be placed in the directory that the program is invoked with. So if you run the program as

  ```
  ./lake 128 5 1.0 8
  ```

  The output will go into **./**. If you run with

  ```
  /home/mydir/otherdir/lake 128 5 1.0 8
  ```

  The ouput will go to **/home/mydir/otherdir/**.

  Be aware of this if you are using batch scripts

- V0
  Report the time for the serial code (1 thread) to run with a grid of 1024 and 4 pebbles, up to a time of 2.0 seconds. (NOTE: this could take about 5 minutes on an opteron processor.)

  ```
  srun -n16 -popteron --pty /bin/bash
  make -f Makefile.serial
  ./lake 1024 4 2.0 1
  ```

- V1
  Optimize the **run_sim()** function to remove unnecessary memory copies. Also, update the function to run using OpenMP directives. (Make sure to use **Makefile.omp**!)

  - The code currently uses a call to **memcpy** to update the **u** arrays.

    ```
    /* update the calculation arrays for the next time step */
    memcpy(uo, uc, sizeof(double) * n * n);
    memcpy(uc, un, sizeof(double) * n * n);
    ```

    Find an optimization to achieve the same results without having to copy the memory from one array to another.

    Time and record this optimization using the same parameters as V0. Use your times when discussing your results in the **p2.README** file.

  - To parallelize using OpenMP, use the **nthreads** parameter to define the number of threads to use for running.

    ```
    #pragma omp parallel for ... num_threads(nthreads)
    ```

    Add directives to the finite differencing loops only. You will run using the following 3 methods of parallelization:
    - parallelize the inner loop only
    - parallelize the outer loop only
    - parallelize both loops

    Time and record each method using the same parameters as V0, except this time with **16 threads**. (Make sure you did srun with **-**n16!)

    ```
    make -f Makefile.omp
    ./lake 1024 4 2.0 16
    ```

    Use your times when discussing your results in the **p2.README** file.

    Choose the fastest method and use this method for the rest of the assignment.

- V2
  Update memory initializations to run using OpenMP directives

  - Update the **init()** function to run using OpenMP directives.

    Add directives to the for loops using the fastest method from V1

    Time and record your results using the same parameters as V1. Use your times when discussing your results in the **p2.README** file.

- The code currently uses a call to **memcpy** to initialize the **u** arrays.

  ```
  /* put the inital configurations into the calculation arrays */
  memcpy(uo, u0, sizeof(double) * n * n);
  memcpy(uc, u1, sizeof(double) * n * n);
  ```

  Edit this code and add directives to parallelize this memory copy using OpenMP.

  Time and record your results using the same parameters as V1. Use your times when discussing your results in the **p2.README** file.

- V3
  Update all OpenMP directives to use dynamic scheduling

  - By default OpenMP uses static scheduling. Loops may be parallelized using dynamic scheduling.

    ```
    #pragma omp parallel for ... schedule(dynamic)
    ```

    Change all of your OpenMP directives to use dynamic scheduling.

    Time and record your results using the same parameters as V1. Use your times when discussing your results in the **p2.README** file.

Hints:

- **Be sure to check your output at each step and ensure that your code is producing valid results.**
- Examine your PGI Compiler output (from -Minfo=mp) AND MAKE SURE IT IS CREATING PARALLEL REGIONS!!
- You may not modify the **Makefile**, **lake.h**, or **lake_util.h** that is provided. I will be using the same ones to grade your work!
- OpenMP tutorial/overview, OpenMP 3.0 spec

When you turn in your **p2.README**, consider the following questions:

- How/why does your optimization for removing memory copies work?
- Does which loop you parallelized matter? Why or why not?
- Does parallelizing both loops make a difference? Why or why not?
- Why does parallelizing memory initializations matter?
- Does the scheduling type matter? Why or why not?
- This program is particularly easy to parallelize. Why?
- (Optional) Can you think of other optimizations either in the code or the OpenMP directives that could further speed up the program? Include a thorough discussion of optimizations. If you'd like to implement these optimizations, please do so in separate files from the base code. You may also submit timings from demonstration runs showing the speedup.

Include all timing data and results for V0-V3 in **p2.README**.

Revert your code back to the fastest version. Keep this code as is for the next problem. We will replace the OpenMP compiler *option* with the compiler *option* for OpenACC. This way we only need to carry around (and submit) one set of code. Do NOT remove the OpenMP directives in the code.

Turn in **p2.README**

3. (15 points) Modify the LAKE program from HW2 to parallelize for OpenACC.

   We will use the code from the previous problem. Use **Makefile.acc** for your OpenACC code. In this Makefile, we change from using OpenMP flags (-mp) to OpenACC flags (-acc -ta).
   (Note: since the -ta flag changes based on which type of GPU is on your node, the Makefile will generate this flag for you)

   Update **lake.c** finite differencing loop to include a simple loop accelerator:

   ```
   #pragma acc kernels loop copy(un[:n*n],uc[:n*n],uo[:n*n],pebbles[:n*n])
   ```

   Notice that since all of our arrays are pointers, OpenACC does not know the size of each array to copy to the device. We must explicitly tell OpenACC the size of each array in a copy clause.

   This pragma can be put, for now, on the outer for loop (you will experiment later to determine the best setup for OpenACC).

   Keep the OpenMP directives in place. The options used to compile them have been removed from **Makefile.acc**, so the compiler will just ignore them. We can keep the code portable this way.

   Run your program. Below is a sample of a serial run, and our naive OpenACC run:

   1. Serial *lake.log*:

      ```
      running ./lake with (128 x 128) grid, until 1.000000, with 1 threads
      Initialization took 0.000849 seconds
      Simulation took 0.229231 seconds
      Init+Simulation took 0.230080 seconds
      ```

   2. OpenACC *lake.log*:

      ```
      running ./lake with (128 x 128) grid, until 1.000000, with 1 threads
      Initialization took 0.001212 seconds
      Simulation took 0.788088 seconds
      Init+Simulation took 0.789300 seconds
      ```

   The naive implementation of OpenACC was slower than the serial code! You should notice something similar with your runs.

You will attempt to do the following

- Accelerate the finite-differencing loops.
- Look for ways to optimize.

From here on out, points are given to the degree you are able to optimize your code . You'll notice that after each execution, OpenACC prints out kernel timing statistics to stdout. Use this information to find places to optimize! Some things to consider (these are not questions you need to discuss, but to guide your optimizations):

- Why is the naive implementation slower?
- What could be done to work around this slowdown?
- Can the loops be rearranged to improve memory access on the device?
- Is there any OpenACC setup code that does not involve any simulation variables that could be moved out of **run_sim**?
- Can we manipulate the block/thread scheduling done by OpenACC to our advantage?

Your code is expected, at the very least (that is, for *any* points on this question), to be **3x** faster than the serial version. As a benchmark, use the parameters

```
./lake 512 4 4.0 1
```

In this benchmark OpenACC code should be at least **20x** faster than the serial code in this benchmark to receive full credit. Extra points (up to 5) will be given for code that is (+1) 40x faster, (+5) 80x+ faster.

Report on your results in your **p3.README** file. Please include:

- Your *lake.log* timings and kernel timing statistics (stdout) for each optimization
- Discuss each optimization (one at a time) in detail
- The effect of the problem size (smaller vs. larger grids, short vs. longer simulation times)
- Where your biggest optimization came from (eg thread scheduling? memory management? screaming at the screen louder?)
- Possible improvements in the code, etc.

Hints:

- **Be sure to check your output and that your code is producing valid results.** Example outputs
- OpenACC quickguide
- PGI OpenACC Getting Started Guide
- Example OpenACC program + optimization techniques (this code is pretty similar to what we're doing; lots of good tips on performance analysis and optimizations)
- Examine your PGI Compiler output (from -Minfo=accel) AND MAKE SURE IT IS CREATING KERNEL CODE. The OpenACC code generation can fail, and your code will still compile without any acceleration.

For example, any output like: "Accelerator region ignored" indicates a problem with your acceleration region. Your complier should explicitly indicate that GPU code was generated:

```
Accelerator kernel generated
Generating Tesla code
226, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
228, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

- **There is a bug in PGI OpenACC that makes explicit copying of scalars (that is, double h, double t, int n, ect.) fail.** You have two options: i.) don't worry about copying them or ii.) put these into 1-element arrays. It's up to you if you think any scalars you use need to be explicitly copied onto the GPU, but from my tests it didn't seem to matter.
- Make sure any code that you change for OpenACC does not break your code when compiled for OpenMP!
- Because we are using the OpenMP/OpenACC code in the same file, you may find that you have OpenMP code/functions that do not compile under OpenACC or vice versa. When OpenMP or OpenACC is used in the compilation, the compiler defines

  ```
  #define _OPENMP
  #define _OPENACC
  ```

  So you can put all your OpenMP or OpenACC code inside of preprocessor conditionals

  ```
  #ifdef _OPENMP
  //code for omp
  #endif
  #ifdef _OPENACC
  //code for openacc
  #endif
  ```

  Note: This is not necessary for OpenMP or OpenACC pragmas. For instance, you may want to use functions from "omp.h" or "openacc.h" in your code, so you should import these inside of the preprocessor conditionals. Any functions from these libraries should be inside of preprocessor conditionals as well.

- You may NOT modify the **Makefile's**, **lake.h**, or **lake_util.h** that are provided. I will be using the same ones to grade your work!

Turn in **lake.c** (with both OpenMP and OpenACC directives, the fastest versions), **p3.README**

What to turn in for programming assignments:

- commented program(s) as source code, comments count 15% of the points (see class policy on guidelines on comments)
- Makefiles (if required)
- test programs as source (and input files, if required)
- README (documentation to outline solution and list commands to install/execute)
- in each file, include the following information as a comment at the top of the file, where "username" is your unity login name and the single author is the person who wrote this file:

```
Single Author info:

username FirstName MiddleInitial LastName
```

How to turn in:

Use the "Submit Homework" link on the course web page. Please upload all files individually (no zip/tar balls).

Remember: If you submit a file for the second time, it will overwrite the original file.