I Assignment 5: Timing, More Parallelization, and Even More Relaxation

due: March 1, 2017 – 11:59 PM (Before Spring Break!)

GOAL: In extreme cases (thus extreme-scale computing), some problems can't even be solved on a single node. In this case, we need to break out multi-node computing methods. In this case, we're going to investigate MPI(Message Passing Interface) in the context of two dimensional relaxation. We're also going to start timing our code using timers.

Within a single "node" with a unified address space one can use threads and openMP to break up a code into concurrent pieces. However, on a network, each node has its own memory and to exchange data you must send a "message" with all of the required information to other nodes with sufficient information so that it can be used on that node.

This problem set is split into three parts.

In the first part, we're going to look at a sample, simple timing code. To make sure you understand how to use it, you're going to modify an old homework assignment and time the different methods.

In the second part, we're going to introduce MPI, starting with a simple Hello World program. There won't be much work to do here, but we are going to have to get on SCC to try it out!

In the third part, we're going to look at a 1D Laplace equation code modified to use MPI. You'll then have to extend it to a 2D code, implementing it as scalar code, modifying it to use OpenMP, and last modifying it to use MPI based on the 1D example.

II Timers

We've put a sample timing code on the class github in the directory `RefHwCode/HW5/timing`. Download this code and give it a look—we're not going to get that into the details, but we'll point out a few small points. This code times performing a dot product of various lengths. A few features of note:

- Line 6: We've had to include a new header, `#include <time.h>`. This contains the timing functions and structures we'll need.
- Lines 11 to 23: Timing is based on a structure called `timespec`, which gets filled with a number of seconds and nanoseconds. If you fill this structure *b*efore and *a*fter calling a function you want to time, the difference between the times in each structure gives you the timing for the function! `diff` just computes this difference safely.
- Lines 67 and 73: The function `clock_gettime` fills the aforementioned `timespec` function with the current time in seconds and nanoseconds.

- Line 76: The function `diff`, as noted before, takes the difference between the times.
- Line 79: This line just saves the timing as the number of seconds plus nanoseconds stored as a `double` as opposed to as long integers.

Make sure you understand this code! You compile it with the function:

```
g++ timing.cpp -lm -lrt -o timing
```

The addition of the flag `-lrt` adds the "real timer" library to the executable, which allows us to perform the precise timing we want for this reference.

For you assignment, you're going to modify your code from Part 1 of Homework 2. As a reminder, in this program you modified the code `search.cpp` to test linear sort, binary sort, and dictionary sort as a function of the number of elements `N`. To make sure you understand how to use timers, you're going to modify the code to also time each implementation.

Your expected deliverables for this assignment are:

- An updated program, `search.cpp`, which adds timers to your code from HW2 and prints timings for all three sorts as a function of `N`.
- A plot of these timings for each sorting algorithm, again as a function of `N`. Make sure you label your axes appropriately!

III Hello, MPI!

There are no deliverables for this exercise, just make sure you can reproduce it on SCC!

We've included a reference MPI "Hello, world!" code on the github in the directory `RefHwCode/HW5/hello`. (There are also reference scalar and OpenMP codes.) You should grab the code directly from the github, but for the purpose of discussion, here's the code:

```
// Based on the tutorial from mpitutorial.com/tutorials/mpi-hello-world/

#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv)
{
   int world_size; // number of MPI processes
   int world_rank; // rank of MPI process
   char processor_name[MPI_MAX_PROCESSOR_NAME]; // name of processor
   int name_len; // length of name string.

```

```
    // Initialize MPI
    MPI_Init(&argc, &argv);

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    MPI_Get_processor_name(processor_name, &name_len);

    printf("Hello world from rank %d of %d! My name is %s!\n",
                        world_rank, world_size, processor_name);

    // Clean up
    MPI_Finalize();
    return 0;
}
```

You'll notice there are no pragmas here. Similar to the OpenMP code, though, we can learn the total number of processes (`MPI_Comm_size`, analogous to `omp_get_num_threads`), and the id of the current process (`MPI_Comm_rank`, analogous to `omp_get_thread_num`). You'll notice there's no barrier here, but the functionality is there. MPI runs more appropriately with Send/Receive calls, which we'll get to when we discuss one dimensional relaxation.

First, we need to discuss how to compile and run MPI code. Instead of using `gcc`, we use `mpicc`:

```
mpicc mpi_hello.c -o mpi_hello
```

The `mpicc` compiler links the appropriate libraries to run MPI jobs automatically—you'll notice we didn't have to add any library flags like we did for the timing code. (By the way, the timing code does work with MPI, and with OpenMP at that.) There's also a special way to run MPI programs, `mpirun`:

```
mpirun -np 2 ./mpi_hello
```

`mpirun` does the appropriate system-specific setup before the executable itself runs. For example, `mpirun` knows how to tell the application how many processes has. It's still our job to tell `mpirun` how many processes there are: the flag `-np 2` means to run the program on 2 processes.

You don't have to worry about job submission on SCC to test this application, you can just run it on the login node! (Up to 4 processes, for a short time—Hello, world is more than short enough.)

Now that we're done with this quick introduction, check out section 6 of the posted lecture notes, titled **"Lecture Notes: Parallelization Methods"**. Some of the material should be familiar, but there's also content on submitting parallel jobs to the job queuing system, which will become essential for the next problem!

As noted before, there are no deliverables for this assignment, but you'll need to be familiar with these concepts to succeed at the next problem!

IV The one-dimensional Laplace equation with MPI

Before we jump into the 2D Laplace equation, let's cover some key concepts with the 1D Laplace equation. We've included a reference 1D Laplace equation using Jacobi only (so don't think this saves you on HW4!) on the github in the directory `RefHwCode/HW5/jacobi`, both for scalar code and MPI code. Let's look at some of the key features line by line. Something that'll come up is this code has special boundary conditions: the solution is fixed to be zero on the far left and right sides.

- Line 9 gives the 1D length of the 1D Laplace problem. 512 is very small, we'll bump this up!
- Lines 12, 15, and 18 give the maximum number of iterations (always an important safety to keep a relaxation code from chugging on forever), a frequency on how long to print the residual, and the target residual of the program.
- Lines 38, 41, and 44 should look familiar from the hello world program, make sure you understand it!
- Line 47 is the first important new line: since nodes can't easily communicate, each node needs to figure out the size of the local problem they're solving! If the length of the dimension is 512, and there are 4 processes, each process handles 512 divided by 4, or 128 sites. (In the balanced case, of course—there are some cases you may want to keep it inbalanced.) Line 49 just handles any overflow if the number of processes doesn't divide equally into the total size of the problem (say, if the number of processes is 6).
- Lines 61 and 63 are a case of where it's important that a process knows what sites it controls. We want the right-hand side to have a single point at the center, at `N/2`. Line 61 figures out what process that point lives on, and then line 63 puts it in place.
- Line 66 computes the magnitude of the right-hand side—this helps us compute the tolerance, as given in HW4. We'll dig into this code below.
- Lines 73 to 86 do the real work, performing Jacobi iterations. You'll notice there aren't any `MPI` calls in here! They're all hiding in the functions `jacobi` and `getResid`. You'll notice on lines 81 to 83 we're making sure only one process prints anything.
- The function starting on line 96, `magnitude`, is a dot product routine in disguise. It has a few extra quirks because we have our zero boundary conditions.

  - Lines 101 and 102 handle these quirks: the first process handles the zero boundary condition on the left hand side, and the last process handles the zero boundary on the right hand side. These lines set the lower and upper limits of the local dot product to avoid these extra zeros. (Of course, this isn't important for zero boundary conditions, but it is for other boundary conditions!)
  - Lines 107 to 110 does the dot product, avoiding the boundaries if necessary.
  - Line 115 is the first bit of MPI magic: the dot product we've done so far is *local* to each node. The issue is we need the dot product summed over every node! That's what `MPI_Allreduce` is for. It takes the local value, `bmag`, sums it over all nodes (thus `MPI_SUM` and `MPI_COMM_WORLD`), and stores the global result in `global_bmag`.

- The function starting on line 120, `jacobi`, performs `RESID_FREQ` iterations of jacobi on the 1D problem. This is where we start dealing with *communication*.
  - Lines 133 and 134 are the first time we need to think about communications. An iteration of Jacobi averages a site over its two neighbors. What happens if the neighbor of a site lives on a different node? We need to *communicate* that value of the network. That's what `left_buffer` and `right_buffer` help with—they'll hold the values exchanged from the other nodes. This is the start of the idea of a `halo`—the nearest-neighbor values on other processes that get exchanged over the network.
  - Lines 146 through 153 handle this exchange *asynchronously*, thus the `Isend` and `Irecv`. Why would we do this asynchronously? The values of `x` don't get updated on the fly; values get updated into an array `tmp`, then afterwards get copied back into `x`. This means we can exchange values at the boundary, do work on all of the sites that don't depend on these boundary values, and *then* actually wait for the results on the network if they haven't shown up yet. This is the idea of overlapping communications with computation, and it's a huge time saver in extreme-scale computing.
  - In the spirit of the above discussion, lines 158 to 161 are where we do local work, and line 164 is the MPI call, `MPI_Waitall`, that waits for the boundary values on the network.
  - Lines 168 through 174 actually do the updates with the boundary values.
  - Line 188 lives outside of this main loop—this is just a nice sanity check to make sure all of the nodes are synced up before we leave the function call doing the work. `MPI_Barrier` is conceptually the same as barriers in OpenMP, except it's distributed over the network.

- The function `getResid` combined the two functions into one go: the residual is defined by computing the norm of the residual, that is, `|b - Ax|`. This requires both applying a Jacobi iteration (applying `A`) and then doing a reduction. Look through the function. Make sure you understand it! Compare parts of it to the previous functions, `jacobi` and !magnitude!. If you have questions, ask!

Once you understand this program, your job is to modify it and time it. The program is currently fixed to a length `N` of 512. Make this a variable and sweep over a range of problem sizes, 16 to 16384 in steps of 4, and time it. Try using 4, 8, and 16 processes. Use the flag `mpi_4_tasks_per_node` just to keep that part fixed. You may need to step away and let this run for a while for the larger values of `N`, so it's a good idea to use submit scripts like what we discussed in the notes. You'll need to make the walltime larger! What's going on, why is this taking so long? It's absurd that this problem has so many issues with a target residual of 10^{-2}, that's hardly accurate at all.

The 1D Laplace problem is a great example of the phenomena of *critical slowing down*, which is a super-linear divergence of the time to solution as the problem size scales. Critical slowing down is a part of why we need bigger machines, but also (and far more importantly) why we need better *algorithms*, like multigrid algorithms. We'll get to that on the next problem set!

For this assignment, submit your modified jacobi code, `my_jacobi_mpi.c`. You should also make a plot of your timings as a function of `N` with different curves for the different number of processes. Think about your axes and if you need to make a log plot or a log-log plot!

V The two-dimensional Laplace equation with MPI

Next up, generalize the MPI code to 2D. To have a more physical intuition for the problem, let's call the variable temperature $T[x][y]$ on a hot plate (or surface of a processor chip), represented by an $L \times L$ grid with $N = L^2$ points and $x, y = 0, 1, 2, \cdots, L-1$ in the interior. The boundaries are lines just off the edge of the square with $x = -1, L$ for all y and $y = -1, L$ for all x. These are sometime called *ghost zones*. Including the ghost zones, this means you need roughly $L + 2$ sites in each dimension—for this reason, it's convenient in C to layout the arrays to include these buffer zones: `double T[L+2][L+2]`.

With that memory layout convention, the zero boundaries lead to `T[0][y] = T[L+1][y] = 0` for all y, and likewise for x. The relaxation routine has a very simple iterative scheme:

$$T_c[x][y] = \frac{1}{4}(T_c[x+1][y] + T_c[x-1][y] + T_c[x][y+1] + T_c[x][y-1]) + b_0[x][y] \tag{1}$$

What values of x and y should you loop over?

With this in mind, the first part of the assignment is to write a serial code in 2D to find the solution to the temperature profile. This is a small modification of `RefHwCode/HW5/jacobi/scal_jacobi.c`. Make variable and sweep over a range of problem sizes, $L = 16$ to $L = 512$ in steps of 2, and time it. (Remember, the volume grows with L^2, which is why we aren't going to as high of values as we did for the 1D case.)

Next break this up into MPI task using 4, 8, and 16 processes. The simplest modification of `RefHwCode/HW5/jacobi/scal_jacobi.c` is to split the x-axis into equal segments of length $L_x = L/\text{world\_size}$. Note is useful to have for each edge of the vertical slices *ghost zones* so the local slice has $x = -1, 0, 1, cdots, L$. The values at $x = -1$ and $x = L$ are shared buffers for the internal edges between processors. These will be used for the MPI send and receive bufferes.

Comment: The solution in 1D of a single source in the middle is very simple to write down. (Can you guess it?) In 2D it is an approximation to $\log[r]$ where r is the distance from the source $r^2 = (x - L/2)^2 + (x - L/2)^2$. If you want to see this, plot the solution in 1D and 2D.

For extra credit (or fun over the Spring Break) you can modify this to break $L \times L$ into squares for each processor, where $N = \text{world\_size} \times L_x \times L_y$ with $L_x = L_y$. This will be part of the problem set after Spring Break you are just getting ahead of the class.

VI Submitting Your Assignment

This assignment is due at 11:59 pm on Monday, February 27. Please e-mail a **tarball** containing the assignment to the class e-mail, bualghpc@gmail.com. Include your name in the tarball filename.

If you're not familiar with tar, here's a sample instruction that perhaps I would use:

```
tar -cvf evan_weinberg_asgn5.tar [directory with files]
```

You may want to include other files. **Do NOT include a compiled executable!** That's a dangerous, unsafe practice to get into.