Assignment 5: Sparse Matrices and Relaxation
due: March 6, 2020 – 11:59 PM

GOAL: This problem introduces relaxation as a way to solve linear algebra problems with sparse matrices. Linear algebra on sparse matrices is a problem at the core of a majority of high performance computing applications—scientific computing, machine learning, and big data! At the same time, the method of relaxation is representative of a wide class of *iterative* methods, where the solution of a problem (in principle) gets more and more accurate with more iterations, as opposed to direct solution methods. Because this is such a representative problem, we're also going to look at parallelizing it using MPI. In the future we will compare this with a treaded code using either openACC or openMP.

Relaxation comes up in a variety of real world applications, such as electrostatics and, perhaps more physically intuitive, heat flow. We're going to start with a one-dimensional application: a metal rod being hit by a blow torch. We'll then move on to two-dimensional problems. These problems will give you the basic building blocks for one possibility for the class projects. We will want to start to discuss forming teams and defining projects after the Spring Break on March 17, 2020.

I Coding up a one-dimensional Laplace solver

Write a program to solve the one-dimensional finite difference heat equation with fixed boundaries using the methods of Gauss-Seidel and Red-Black iterations. These algorithms are described in depth, albeit in the two-dimensional case, in the class notes. For your convenience, we've provided a base code which solves this problem with Jacobi iterations on the class GitHub in the file `MPI_OPM_example/Jacobi/jacobi_scal.cpp`. Remember, each algorithm is nearly identical, with only a few changes in the iterative loop!

In the reference code, we accumulate a solution in the array `double T[N +1]` with values $T[i]$ spanning $i = 0, ..., N$. The fixed boundary values live in the sites $T[0] = T[N] = 0$. The source now lives at the mid-point, $i = N/2$. The code tracks the iterative solution by outputting the residual $r = b - AT$ every 1000 iterations, that is, printing the norm of the vector

$$r[i] = b[i] - AT[i] = b[i] - (T[i] - \frac{1}{2}[T[i-1] + T[i+1]]). \tag{1}$$

(Note we have chosen to define A by delving by 1/2 relative to the discrete one-dimensional Laplace matrix – a very simple "preconditioner".)

The program stops iterating when the relative residual reaches a target $\epsilon = 10^{-6}$, that is, the constraint

$$\frac{\sqrt{\sum_{i=1}^{N-1} r[i] * r[i]}}{\sqrt{\sum_{i=1}^{N-1} b[i] * b[i]}} < 10^{-6} \tag{2}$$

Since the input is at a single point when you increas N a better metric for convergence migh the rout means square (RMS) error in the residuals: $\sqrt{N^{-1} \sum_{i=1}^{N-1} r[i] * r[i]} < 10^{-6}$.

The goal of this assignment is to compare how each method converges by tracking the relative residual as a function of the iteration count at large N. You should produce a new source file for each method: `gauss_seidel_scal.cpp` and `red_black_scal.cpp`. For this part of the assignment, keep N fixed at the default value of 512. You should also make a plot which shows the relative residual as a function of the iteration count, one curve per method, all overlaid on the same figure.

As a next step, parallelize the code using MPI! As we discussed, Gauss-Seidel can't be parallelized well, so we'll skip that. Instead, just parallelize the Jacobi iterations using the appropriate MPI instructions. We start by explain the full solved 1d example using mpi at : `MPI_OPM_example/Jacobi/jacobi_mpi.`

I.1 The one-dimensional Laplace equation with MPI

Before we jump into the two-dimensional Laplace equation, let's cover some key concepts with the one dimensional Laplace equation. We've included a reference Jacobi code, parallelized with MPI, on the class GitHub in `MPI_OPM_example/Jacobi/jacobi_scal.cpp`. 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 one-dimensional 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. 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.
- 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, 64 to 16384 in multiplicative steps of 4. Add the timing routines in the example in `MPI_OMP_example/n01TimingMPI_OMP` Try using 4, 8, and 16 processes, using the submit flag `mpi_4_tasks_per_node`, and the appropriate analogs for 8 and 16. You should use submit scripts because this will take a while! 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 one-dimensional 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, `jacobi_mpi.c`, and a representative submit script. You should also make a plot of your effective clock cycles, like we've done previously, 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!

II The two-dimensional Laplace equation with MPI

Next up, generalize the MPI code to two dimensions. 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. (See Chapter 7 in the notes for more deails on the Temperature application.) 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 $L + 2$ 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 `T[X][0] = T[X][L+1] = 0` for x. The relaxation routine has a very simple iterative scheme:

$$T_c[x][y] = \frac{1}{4}(T_c[x+1][y] + T_c[x-1][y] + T_c[x][y+1] + T_c[x][y-1]) + b_0[x][y] \tag{3}$$

Now put in a hot source in the midle of the plate $b_0 = 0$ except $b_0[L/2][b[L/2] = 100$.

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 the one-dimensional code using Jacobi. Sweep over a range of problem sizes, $L = 16$ to $L = 512$, in multiplicative 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 tasks 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 one dimensions of a single source in the middle is very simple to write

down. (Can you guess it?) In two dimensions 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 one and two dimensions.

The deliverables for this part of the assignment is your source code, your submit script, and curves of the effective clock cycles as a function of `L` for 4, 8, and 16 processes—the same as we were looking for in the previous part of this homework.

For extra credit or to keep you from getting bored 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 probably be part of the problem set after Spring Break you are just getting ahead of the class.