Homework 1: Conjugate Gradient and Poisson's Equation

Andrew Emerick

October 14, 2013

1 Conjugate Gradient Solver

I successfully completed the conjugate gradient solver in both a serial and parallel implementation. The serial implementation can be found in Emerick_cg-serial.cpp , while the parallel version in Emerick_cg-mpi.cpp . The two were implemented on the same simple test case, and the output for each are stored in the respective .dat files. The test case was N=3, so the parallel version ran comparatively slow because commmunication is much more expensive relative to computation for such small N. However, the proof of concept still stands.

I will be honest, I did not spend much time cleaning up my parallel cg solver code. It is not written as a stand alone function meant to be incorporated into a larger program. Doing so would not be challenging, but I had spent too much time elsewhere trying to get the Poisson solver working. It operates by splitting A between processors (lines 144-153) and making the multiplication A * p in parallel (lines 158-168). Then it regjins the resulting Ap_vec (lines 171-183) to the rank=0 processor. From here everything is run in serial. In my own testing I found it was ultimately ineffecient to have any more than this multiplication be run in parallel. Theoretically more aspects could be parallelized, and the means of parallelization could be more creative, but this version works fine.

2 Poisson Solver

2.1 3D Solver - Serial

The Poisson equation solver in three dimensions running on a single processor is suprisingly simple, assuming one can readily construct the necessary Laplacian matrix. Doing this from scratch turned out to be somewhat challenging for me (it took me some time to identify the right pattern and get the correct if statements to fill the sparse matrix with the correct values). However, once this is done, the solution is readily found by passing the sparse matrix and the right hand side of the Poisson equation through the conjugate gradient solver. I do not include the serial code here, and discuss the parallel version and the code included in the next section. Suffice to say however, the motivation for obtaining a parallelized version to split the problem into chunks is readily understood, as the Laplacian matrix quickly gets obsurdly large for even a modest resolution, as A is NxN for $N=n^3$ grid points (for n=64 grid points in each dimension, A contains over 68 billion elements).

2.2 3D Solver - Parallel 1D Processor Grid

For notation purposes, I am solving the equation $\nabla^2 U(x,y,z) = b(x,y,z)$ for $b(x,y,z) = -4.0 \exp\left(-2.0\sqrt{x^2+y^2+z^2}\right)$. The equation is solved with the Laplacian sparse matrix A. The variables for the conjugate gradient algorithm are the same as given in the handout.

After much tribulation, I finally obtained the parallel 3D Poisson equation solver, that works on an arbitrary number of processors for an arbitrary number of grid points in each dimension provided thet: 1) the number of grid points is the same in each dimension (i.e. a cubic volume), 2) the number of processors does not exceed the number of grid points in the z dimension, and 3) that the number of grid points in the z dimension is evenly divisble by the number of processors used. Though the last point could be fixed by writing an algorithm to smartly divide up the remainder to a number of processors, doing this robustly is not trivial.

I had two main issues in writing the parallelized solver. The first, which we had discussed, was that I had forgotten to take the global sums needed to properly calculate the α and β parameters for the conjugate gradient. The second, was that I conceptually understood how information needed to be passed between processors for adjacent faces, but I did not implement this appropriately for the conjugate gradient. I naively passed information between faces for the potential (or U in A*u = b as in my code), even though a glance at the algorithm would reveal this does nothing — it is the p vector that needs to share information across processors. The final issue I faced was information loss during processer communication. I had e-mailed you about this previously, but for some reason (with both blocking and non-blocking send/receive) a given processor would only receive information from one adjacent face, but not both. The workaround was alternating the send/receives so a given processor is either sending or receiving at a time, but not both.

An outline of the code is as follows. Line 64 gives the function used to construct the Laplacian matrix. This matrix is no longer square. It contains a square portion equal to the number of grid points on a given processor, but on either side (left/right) of this part are elements used to account for the adjacent z faces. Thus the number of rows is n*n times the number of z slices per processor, and the number of columns is this plus the width of the buffers used to share information between faces (i.e. 2*n*n). Line 110 is the density function, or in general the function used to make the R.H.S of the Poisson equation.

Lines 121-174 declare all variables and initialize memory for the arrays. The size of each array is n*n times the number of z slices per node, where n is the grid points per dimension. As alluded to above, both the A matrix and the p array are larger than this to include the buffers used to share information between faces (located at the first and last n*n elements of p). This comes into play in the p·A· calculation for the cojugate gradient algorthim. b is filled, while p, r, U are initialized in lines 191-206. Line 212 begins the algorithm while loop, while the loop begins by passing information between adjacent faces in an alternating fashion. Each processor only sends or receives up/down at a given time, but not both send and receive. This prevents information loss. Lines 278 onward are the rest of the conjugate gradient algorithm. The only difference here between the serial version is that global sums (MPI_Allreduce) are needed to calculate α and β; this information is shared between all processors. Lines 323 onward contain code to construct the output in a nice format by having each processor send its U and b array contents to the rank=0 processor for output.

I ran the program on 16 processors using n=32 grid points in each dimension, with each side of the simulated cube having a length L=4. The results are plotted in Fig. 2.2. The data file is not included here (2.5 MB), but the program finished with in 1.33 minutes with an error of 8.77×10^{-10}.

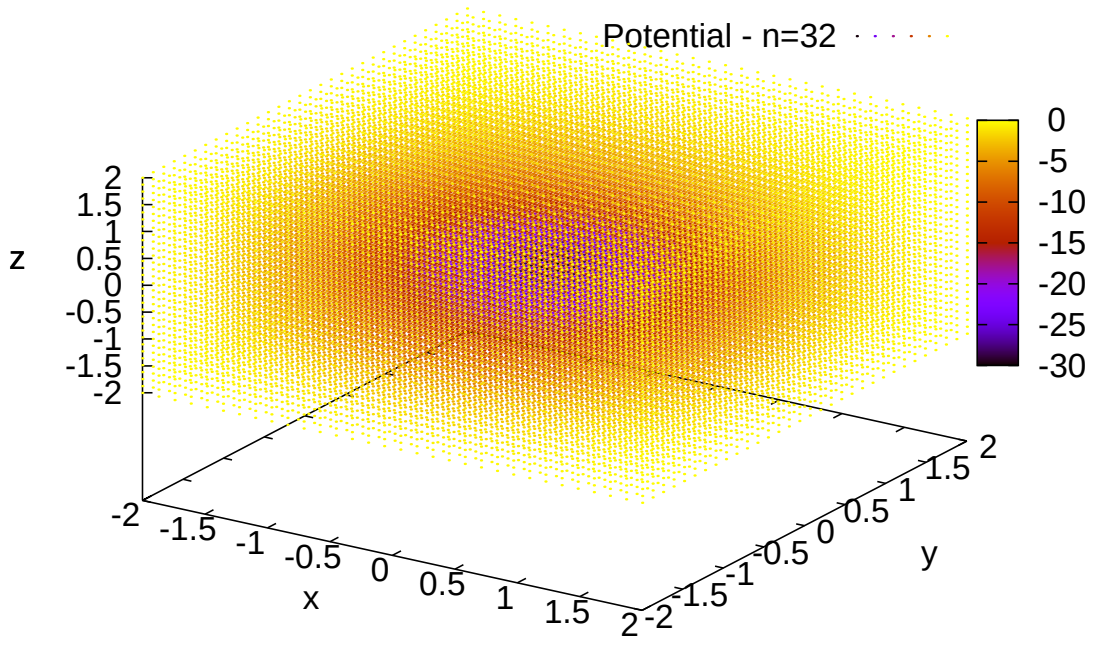

Figure 1: Shown is the potential U(x,y,z) obtained via the Poisson equation solver for n=32 grid points in each dimension, on 16 processors.