Homework 2: Parallel IRL and the Helium Atom

Andrew Emerick

December 14, 2013

1 Parallel IRL Algorithm

In irl_p.cpp (with the accompying irl_p.h header file), I employ a parallel IRL algorithm computed utilizing the GSL library to serially solve for the eigenvalues of H , and to perform the QR decomposition of H_{k+p}. The serial version of this code (not included) is as close to a translation of the matlab irl.m to C++ as possible. The primary differences between the matlab version and the C++ version were the many extra loops needed to properly multiply, add, or subtract vectors. In addition, function calls became slightly more complex (needing to pass more parameters, as I tried to avoid using global variables). For the GSL eigensolver and QR decomposition, I wrote two functions ('eigensolve' and 'qrdecomp') that are essentially wrappers on the GSL functions. These wrappers allow me to pass matrices constructed with std::vector, and be returned matrices as std::vector in the same format as the matlab code. This limits the use of GSL specific variable types/classes to these functions.

The parallelization comes by spreading the v matrix (and thus vp and evector) over the number of processors (essentially dividing the total volume amongst the number of processors). The code as written cannot take an arbitary processor number. It requires that the number of processors be a perfect cube (e.g. 1, 8, 27), and that the number of global grid points in each dimension is evenly divisble by the number of processors in each dimension (e.g. if 27 processors are used, global grid points in each dimension must be divisble by 3). Once the volume is constructed (more on this later), and proper accounting is employed for proper sharing across processors, the process of solving for the eigenvalues occurs simultaneously on each procesor, rather than having only the rank 0 processor do the computation, and then having to distribute the information across all processors for the next iteration.

For proper accounting, I construct three matrices: p_xyz, nbr, and nbrfc. p_xyz has a number of rows equal to the number of processors, so row n corresponds to the n-1 lexical order processor number (the rank). The first three columns contain the 3 dimensional processor coordinate of the rank n processor (called px, py, pz), and the following 6 contain the lexical order processor number (rank) of the adjacent processors in x+, x-, y+, y-, z+, and z-. This matrix is used to properly set with processors should be sending and receiving data from any given processor.

The v matrix contains buffers at its end to take in values from the adjacent processors. The total length of the buffer is 6 times the number of grid points on each given face (or 6.0*L*L). The first L*L indices are set to contain the values of v for the processor adjacent towards the x- direction. Each subsequent set of L*L elements correspond to the x+, y-, y+, z-, and z+ values. The nbrfc matrix is used to record the lexical order values in v of all grid points sitting on a face. For a given processor, nbrfc[0] contains a list of all indeces in v corresponding to grid points sitting on the x+ side of the cube. Each row after this (1-5) contains x-, y+, y-, z+, and z- respectively. This matrix is necessary to properly send information in the MP_Isend calls.

The nbr and nbrfc matrices are set in the set_bc_nbr_nbrfc function (similar to the serial function set_bc_nbr from the matlab code), while the p_xyz matrix is set in the p_xyz function. The sharing of information across processors occurs immediately before any call of the Av matrix times vector function, and occurs in the share_v function. v is the only matrix that needs to be shared and contains buffers. Any other sharing that occurs in the algorithm occurs as MPI_Allreduce's for when a dot product or norm is taken over the entire volume.

An example output for irl_p.cpp is included (answer_p8_8.txt) for the algorithm run on 8 processors solving for 8 eigenvalues using kmax = 12 and pmax = 22. The global number of grid points in each dimension is 40. This took 8.2 minutes, compared to 13.3 minutes on 1 processor.

2 Helium Atom

Utilizing the IRL and CG solvers, I compute the ground state energy of the Helium Atom following the matlab code from Mikael Kuisma. I will give a summary of results first, then go into how the code is structured. After our discussion on Thursday, I successfully figured out the normalizations and have produced the correct ground state energy of Helium. I have included an output run on 1 processor with L=40 in each dimension, run with 10 DFT iterations. As you can see, it converges to -2.678 after only 7 iterations. Unfortunately, however, I have somehow introduced a bug in making these changes. Previoulsy my singe and multiple processor runs converged together, but now they are converging to different values. I have included an output for L=40 run on 8 processors over 10 DFT iterations. Unfortunately, I have to table finding the bug until later. If I do, I will send you an update.

The code is tructured as follows. The irl_func.cpp code contains the irl solver function and associated functions needed. cg_func.cpp is contains the conjugate gradient solver and associated functions. parallel_functions.cpp contains functions needed to set up the volume, including setting boundary conditions, setting up the local neighbor matrix, and setting up the global processor neighbor matrix. These three are employed in the helium.cpp file, which computes the energies, following the matlab function. The exchange, hartree, and external energies are calculated within the main function, with Vh originating from the conjugate gradient solver and psi from the IRL algorithm. The compensation (ncomp) is employed directly from the matlab code. The external potential is set initially in a separate function, along with ncomp. The loop calculates kinetic energy with a special function, since it involves matrix times vector operations.