PHYS 410 Upgrade

Graham Reid

December 17, 2014

Contents

1 Background

1.1 Motivation

When I first came to UBC I planed on pursuing a masters degree and had no real intention of going on to do a PhD. After applying for and receiving an NSERC CGSD in my first year, I changed my mind and decided to transfer to the PhD program. Unfortunately, after the transfer, the 400 level course I had taken (PHYS 410-Computational Physics) no longer counted towards my degree.

Hoping to find a solution which would reduce my workload for the upcoming year, I approached Dr. Schleich about the possibility of upgrading the course to the graduate level. Dr. Schleich agreed to the possibility of an upgrade provided I submitted an additional graduate level project, preferably relevant to but not related to my research.

This document comprises the final project necessary to upgrade my PHYS 410 credit to PHYS 555 in accordance to the basic outline devised by myself and Dr. Schleich and approved by the UBC Physics department.

1.2 Changes to Project Proposal

The final project differs from the original proposal in a number of ways. Originally, I had planned solve the Schrödinger equation (Equation 1), Poisson equation (Equation 2) and Schrödinger-Newton equations (Equations 3-4) in 3D using `CUDA`:

$$i\frac{\partial \Psi}{\partial t} = -\Delta\Psi + V(x,y)\Psi \tag{1}$$

$$\Delta\Phi = f(x,y) \tag{2}$$

$$i\frac{\partial \Psi}{\partial t} = -\Delta\psi + \Phi\Psi \tag{3}$$

$$\Delta\Phi = \Psi\overline{\Psi} \tag{4}$$

The Schrödinger-Newton equations in particular are topical to my current research as they are the non relativistic limit of a self gravitating Klein-Gordon field [1–3]. If the general relativistic analogs of these equations were solved, they would produce static solutions know as Boson stars which have been widely studied and have been considered as candidates for various astrophysical objects [2, 4–7].

Due to time constraints, I was unable to implement the Schrödinger and Schrödinger-Newton equations in 3D. These equations are however implemented in 1D cartensian coordinates for the case of the Schrödinger equation and spherical symmetry for the Schrödinger-Newton Equations (see section 4 for the complete list of implemented problems).

The second major change is that the multigrid utility (`MG_OMP`) developed for the purpose of this project does not make use of graphics card acceleration

through either CUDA or OpenACC. When the single core/OpenMP version of the utility was developed, I determined that I had made a number of design choices which prevented the efficient use of CUDA and OpenACC. Rather than redesign the utility, I implemented a simple 2D wave equation in Fortran (Wave Utility) and accelerated it using OpenMP, MPI and OpenACC. The performance comparisons between OpenMP, MPI and OpenACC presented in section 5 are thus based on this code rather than MG_OMP.

1.3 Document Plan

The document may be broken down as follows:

- Section 2 provides a basic introduction to parallel programming and outlines outlines four parallel programming paradigms (MPI, CUDA, OpenMP and OpenACC).
- Section 3 provides a broad description of the MG_OMP utility for solving PDE's using the multigrid method.
- Section 4 outlines the sample problems provided with the MG_OMP utilitity and demonstrates the convergence of the problems based on equations 2-4.
- Section 5 tests the performance of the MG_OMP utility and Wave Utility for various grid dimensions and number of threads.
- Section 6 states conclusions and discusses the various observations made in completing this project.

2 Parallelization

Parallelization of single processor code can enable enormous speedups if the code is inherently parallelizable. As finite difference methods compute derivative using stencils of neighbouring grid points, these codes are often easily and efficiently parallelized. Below, the three methods of parallelization used in this project are discussed.

2.1 MPI

The Message Passing Interface (MPI) is the current standard for communication among parallel processes on distributed memory systems such as large compute clusters. MPI uses language independent communications protocol for sharing information between parallel processes enabling communication between processes independent of the language and architecture being used.

In C, Fortran and Java, this communication is achieved through function calls which transmit contiguous chunks of memory from one process to another [8]. These routines are implemented both by the open source community and through proprietary compilers such as those made by Intel® Corporation which typically perform far more efficiently [9].

2.2 OpenMP

OpenMP (Open Multi-Processing) is an API which supports shared memory parallel processing shared memory systems such as those found on a single compute node. OpenMP is supported in C, C++ and Fortran where it is enabled by inserting compiler specific directives into existing code which allow the compiler to parallelize the program. In the case of a for loop, this would involve entering the parallel region, spawning a number of new threads (an expensive operation), and having each thread loop over a different section of code. As with MPI, both proprietary and open source implementations exist.

2.3 CUDA and OpenACC

CUDA (Compute Unified Device Architecture) is freeware developed by Nvidia® Corporation for accelerating code on GPUs [10]. CUDA takes advantage of the huge computing power of modern Nvidia GPUs ($\approx$ 4TFLOPS for a Tesla [11]) compared to CPUs ($\approx$ 20 GFLOPS for a single core) to achieve large performance increases with parallel codes.

CUDA is available through compiler directives in a method almost identical to OpenMP through OpenACC (Open Accelerators), or through extensions to existing languages such as CUDA C [12]. Due to the simplicity of OpenACC versus CUDA C, I opted to use OpenACC for this project.

3 Utilities

This project broadly consists of two utilities, the first of which (MG_OMP) solves general 1D, 2D and 3D PDE's via the FAS Multigrid Method. The second utility is the Wave Utiltiy which is implemented using OpenMP, OpenACC and MPI and is used for determining the relative efficiency of the various parallel programming techniques.

Unless otherwise specified, MG_OMP was compiled using g++ with -Ofast enabled on a dual core i5 system. The OpenMP and OpenACC versions of the Wave Utility were compiled on the Parallel cluster on WestGrid [13] using pgfortran [14] with optimizations enabled. The MPI version of the code was likewise compiled and run on Parallel, but uses mpif90 with -Ofast enabled.

3.1 Multigrid Method

Ordinary methods for solving elliptic equations such as Jocbi itteration and Successive-Over-Relaxation are computationally inefficient requiring $O(n^f) > O(n)$ operations to compute an approximate solution on an n point grid [15]. Conversely, the multigrid method can solve such problems in $O(n)$ operations [15, 16] and therefore represents a far superior computational tool.

The multigrid algorthim presented below is designed to solve problems of the form $Lu = f$ on a series of l grids where the finest has spacing h (u^l) and the coarsest has spacing $2^{l-1}h$ (u^1). Here, L is some non linear operator, u are

the fields to be solved for and f is a source function which is often moved into the definition of L. By successively interpolating and restricting the problem to grids of different resolutions using the interpolation operator I_{i-1}^{i} and restriction operator I_{i}^{i-1}, the multigrid method is able to accelerate to convergence of high frequency modes in the elliptic problem.

A full description of the Full Approximation Storage (FAS) multigrid method for non linear PDEs may be found in [15] and [16], but for convenience the vCycle algorithm is summarized in the pseudo code below [15]. This algorithm typically reduces the error of the solution by about an order of magnitude each iteration.

---

procedure VCYCLE(l, p, q)
  initialize source on finest grid
  $s^l = f^l$
  cycle from fine grid to coarse
  for i=l; $i \geq 2$; $i- = 1$ **do**
    apply smoothing relaxation iterations
    for $j = 1$; $j \leq p$; $j+ = 1$ **do**
      $u^i = \text{relax}(u^i, s^i, h^i)$
    end for
    restrict to coarse grid problem
    $s^{i-1} = L^{i-1} I_i^{i-1} u^i - I_i^{i-1} L^i u^i + I_i^{i-1} s^i$
    $u^{i-1} = I_i^{i-1} u^i$
  end for

  solve coarsest level problem
  while $|r^1| > errorTol$ **do**
    $u^1 = \text{relax}(u^1, s^1, h^1)$
  end while

  cycle from coarse grid to fine
  for i=2; $i \leq l$; $i- = 1$ **do**
    apply correction from coarse grid
    $u^i = u^i + I_{i-1}^{i} \left(u^{i-1} - I_i^{i-1} u^i \right)$
    apply smoothing relaxation iterations
    for $j = 1$; $j \leq q$; $j+ = 1$ **do**
      $u^i = \text{relax}(u^i, s^i, h^i)$
    end for
  end for
end procedure

3.2 Multigrid Utility

The multigrid (`MG_OMP`) utility developed for the purpose of this project implements the FAS multigrid method to solve 1D, 2D and 3D PDEs of hyperbolic,

elliptic or parabolic character. The relaxation routine is a iterative Jacobi solver, thus the update scheme for a PDE of the form $\mathrm{L}\mathbf{u} = 0$ with approximation $\tilde{u}$ such that $L\tilde{\mathbf{u}} = \tilde{\mathbf{r}}$ is

$$\tilde{\mathbf{u}}^{n+1} = \tilde{\mathbf{u}}^{n} - f\tilde{\mathbf{r}}\mathrm{J}^{-1} \tag{5}$$

$$\mathrm{J} = \mathrm{diag}\left(\frac{\partial \mathrm{L}_i}{\partial \mathbf{u}_i}\right) \tag{6}$$

$$0 \leq f \leq 1 \tag{7}$$

Since only the diagonal terms of the jacobian are used in the update, systems which are not diagonally dominant will not converge. However, in many cases it is possible to diagonalize the equations by hand as can be seen in section 4. The factor f in the above equations is used to smooth the update process. By choosing $f \approx 0.8$ superior convergence behaviour can be achieved.

`MG_OMP` is written in `C++` and consists broadly of three classes, `Mesh`, `Coord` and `Hierarchy`. A instance of `Hierarchy` consists of a collection of `Mesh` and `Coord` instances defined on l levels along with the methods necessary to implement the vCycle algorithm. A `Mesh` instance represents a single field defined on a given level of the hierarchy. It consists of a single data array along with a collection of pointers for manipulating the array as a 1D, 2D or 3D `C++` array. Additionally, `Mesh` provides methods for interpolating or restricting a given `Mesh` instance onto a different level of the hierarchy. Finally, the `Coord` class represents a set of coordinates on a given level. For a n-dimensional hierarchy, an instance of `Coord` consists of n 1D `Mesh` instances populated as coordinates.

To use `MG_OMP`, the user must supply two functions `f_jacobi` and `f_opp` (named however the user wishes). Examples demonstrating the implementation and use of such functions are shown in section 4.

One further item of note is that launching threads using `OpenMP` is a computationally expensive process; i.e. for small loops, the cost of creating a `OpenMP` parallel region greatly outweighs any possible speedup **cite**. For this reason, whenever `MG_OMP` encounters a loop smaller than `MIN_GRID_SIZE` as defined in the `Mesh` class, a serial version of the iteration is automatically used for all `Mesh` and `Hierarchy` methods. To take full advantage of this speedup, the user supplied `f_jacobi` and `f_opp` must make similar checks (This is shown in the Schrödinger-Newton example).

3.3 Wave Utility

Due to memory structure of `MG_OMP`, it could not be efficiently ported to either `OpenACC` or `MPI`, Therefore, in order to make a fair comparison of `OpenMP` and `OpenACC`, a small program for solving the 2D wave equation was written and subsequently ported to `OpenMP` and `OpenACC`. The wave utility solves

$$\frac{\partial^2 \phi}{\partial t^2} = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \tag{8}$$

using the following Crank-Nicolson discretization (where i and j indicate indices on the grid)

$$\sigma = \phi \tag{9}$$

$$\psi = \frac{\partial \phi}{\partial t} \tag{10}$$

$$0 = \frac{\sigma_{i,j}^{n+1} - \sigma_{i,j}^{n}}{\Delta t} - \frac{\psi_{i,j}^{n+1} + \psi_{i,j}^{n}}{2} \tag{11}$$

$$0 = \frac{\psi_{i,j}^{n+1} - \psi_{i,j}^{n}}{\Delta t} - \frac{\sigma_{i-1,j}^{n+1} - 2\sigma_{i,j}^{n+1} + \sigma_{i+1,j}^{n+1}}{2\Delta x^2} - \frac{\sigma_{i-1,j}^{n} - 2\sigma_{i,j}^{n} + \sigma_{i+1,j}^{n}}{2\Delta x^2} - \frac{\sigma_{i,j-1}^{n+1} - 2\sigma_{i,j}^{n+1} + \sigma_{i,j+1}^{n+1}}{2\Delta y^2} - \frac{\sigma_{i,j-1}^{n} - 2\sigma_{i,j}^{n} + \sigma_{i,j+1}^{n}}{2\Delta y^2} \tag{12}$$

The full source code for the `Wave Utility` is included in the repository [17].

4 Sample Problems

The following section demonstrates how to solve PDEs using `MG_OMP`. A step-by-step breakdown of the code is given for the 1D Poisson Equation on the line and the Schrödinger-Newton equation in spherical symmetry.

4.1 Poisson 1D

The Poisson equation in 1D is:

$$\frac{\partial^2 \phi}{\partial x^2} = f(x, \phi) \tag{13}$$

For our example we will solve,

$$\frac{\partial^2 \phi_1}{\partial x^2} = 0 \tag{14}$$

$$\phi_1(-1) = 1 \tag{15}$$

$$\phi_1(1) = 1 \tag{16}$$

$$\frac{\partial^2 \phi_2}{\partial x^2} = \phi_2^2 \tag{17}$$

$$\phi_2(-1) = 1 \tag{18}$$

$$\phi_2(1) = 1 \tag{19}$$

Using a centered second order discretization, we define the interior residuals,

$$r_{1i} = \frac{\phi_{1i-1} - 2\phi_{1i} + \phi_{1i+1}}{\Delta x^2} \tag{20}$$

$$r_{2i} = \frac{\phi_{2i-1} - 2\phi_{2i} + \phi_{2i+1}}{\Delta x^2} - \phi_{2i}^2 \tag{21}$$

and exterior residuals,

$$r_{10} = \phi_{10} - 1.0 \tag{22}$$
$$r_{1nx-1} = \phi_{1nx-1} - 1.0 \tag{23}$$
$$r_{20} = \phi_{20} - 1.0 \tag{24}$$
$$r_{2nx-1} = \phi_{2nx-1} - 1.0 \tag{25}$$

The Corresponding Jacobians are:

$$J_{1i} = \frac{-2}{\Delta x^2} \tag{26}$$
$$J_{10} = 1 \tag{27}$$
$$J_{1nx-1} = 1 \tag{28}$$
$$J_{2i} = \frac{-2}{\Delta x^2} \tag{29}$$
$$J_{20} = 1 \tag{30}$$
$$J_{2nx-1} = 1 \tag{31}$$

With these residuals and Jacobians defined, we can write the `f_jacobi` and `f_opp` methods for this problem. Starting with `f_opp`, we must first retrieve the coordinate and field arrays from the method arguments:

```
void f_laplace_opp_1d(double dt, Coords* coords, Mesh ** level) {
   int i, nx, num_f;
   double f, dx;

   double* x = coords->mesh[0]->f1d;

   num_f = 2;
   double* phi1 = level[0+Hierarchy::u_i*num_f]->f1d;
   double* phi2 = level[1+Hierarchy::u_i*num_f]->f1d;

   double* phi1_res = level[0+Hierarchy::res_i*num_f]->f1d;
   double* phi2_res = level[1+Hierarchy::res_i*num_f]->f1d;

   dx = x[1] - x[0];
   nx = coords->size[0];
```

Here `Hierarchy::u_i=0`, `Hierarchy::res_i=1` and `Hierarchy::jac_i=2` are constants indicating the location of the function, residual and jacobian fields. Next, we enter the loop region. Since we are using `OpenMP` we enclose the loop region in a `parallel` pragma:

```
#      pragma omp parallel firstprivate(i, nx, f, dx, x, phi1, phi1_res, \
        phi2, phi2_res)
```

```
      {
         double phi1_xx, phi2_xx;

#         pragma  omp for nowait
         for (i = 1; i < nx - 1; i++) {
            f = phi1[i] * phi1[i];
            phi1_xx = (phi1[i - 1] - 2.0 * phi1[i] + phi1[i + 1]) / (dx * dx);
            phi1_res[i] = phi1_xx - f;

            f = 0.0;
            phi2_xx = (phi2[i - 1] - 2.0 * phi2[i] + phi2[i + 1]) / (dx * dx);
            phi2_res[i] = phi2_xx - f;
         }
      }
```

The clause `firstprivate(...)` tells the compiler that each thread should get its own copy of these variables. Strictly speaking, it is not necessary to add the arrays themselves to this clause as the default behaviour for externally defined variables inside the `parallel` region is to make them shared across each processor. Since we are not modifying the array pointers it makes essentially no difference, and I find that it helps me keep track of variable scope.

Finally we apply the boundary conditions and finish off the method:

```
      phi1_res[0] = phi1[0] - 1.0;
      phi1_res[nx - 1] = phi1[nx - 1] - 0.0;

      phi2_res[0] = phi2[0] - 1.0;
      phi2_res[nx - 1] = phi2[nx - 1] - 1.0;
   }
```

The Jacobian method is very much the same:

```
   void f_laplace_jac_1d(double dt, Coords* coords, Mesh ** level) {
      int i, nx, num_f;
      double f, dx;

      double* x = coords->mesh[0]->f1d;

      num_f = 2;
      double* phi1 = level[0+Hierarchy::u_i*num_f]->f1d;
      double* phi2 = level[1+Hierarchy::u_i*num_f]->f1d;

      double* phi1_jac = level[0+Hierarchy::jac_i*num_f]->f1d;
      double* phi2_jac = level[1+Hierarchy::jac_i*num_f]->f1d;

      dx = x[1] - x[0];
      nx = coords->size[0];
```

```
#      pragma omp parallel firstprivate(i, nx, f, dx, x, phi1, phi2, \
      phi1_jac, phi2_jac)
      {
         double dphi_phi1_xx, dphi_phi1;
         double dphi_phi2_xx, dphi_phi2;

#         pragma  omp for nowait
         for (i = 1; i < nx - 1; i++) {
            dphi_phi1_xx = (-2.0) / (dx * dx);
            dphi_phi1 = -2.0 * phi1[i];
            phi1_jac[i] = dphi_phi1_xx + dphi_phi1;

            dphi_phi2_xx = (-2.0) / (dx * dx);
            dphi_phi2 = 0.0;
            phi2_jac[i] = dphi_phi2_xx + dphi_phi2;
         }
      }

      phi1_jac[0] = 1.0;
      phi1_jac[nx - 1] = 1.0;

      phi2_jac[0] = 1.0;
      phi2_jac[nx - 1] = 1.0;
   }
```

Now all we have to do is set up the problem and run it. With `MG_OMP` this requires only a few lines of code. First, define the number of functions (2), number of levels in the multigrid hierarchy (11), dimension of the problem (1) boundaries of the problem (-,1,1) and size of the smallest grid (3):

```
int num_l;
int num_f;
int dim;
int i, j, n;
int size_base[1];
double bbox[2];

num_l = 11;
num_f = 2;
dim = 1;
size_base[0] = 3;
bbox[0] = -1;
bbox[1] = 1;
```

For a 2D problem this might change to:

```
int num_l;
int num_f;
int dim;
int i, j, n;
int size_base[2];
double bbox[4];

num_l = 11;
num_f = 2;
dim = 1;
size_base[0] = 3;
size_base[1] = 3;
bbox[0] = -1;
bbox[1] = 1;
bbox[2] = -1;
bbox[3] = 1;
```

Then we set up the hierarchy and solve the problem:

```
Hierarchy* hierarchy = new Hierarchy(num_l, num_f, dim, size_base, bbox);
resid = 1.0;
while (resid > 1e-8) {
   hierarchy->vCycle(4, 0, f_laplace_opp_1d, f_laplace_jac_1d);
   printf("change %e, residual %e\n ", hierarchy->change[num_l - 1],
           hierarchy->resid[num_l - 1]);
   resid = hierarchy->resid[num_l - 1];
}
```

The declaration for `vCycle` is `vCycle(int smoothing_itterations, dt, f_opp, f_jac)`. Solutions can then be accessed through `hierarchy->fields[num_l][i]->f1d` where `i=0` corresponds to ϕ_1 and `i=1` corresponds to ϕ_2.

4.2 Schrödinger Newton

The Schrödinger-Newton equations (Equations 3-4) can be rewritten as:

$$\Psi_1 = \Re\Psi \tag{32}$$

$$\Psi_2 = \Im\Psi \tag{33}$$

$$0 = \frac{\partial\Psi_1}{\partial t} + \frac{2}{r}\frac{\partial\Psi_2}{\partial r} + \frac{\partial^2\Psi_2}{\partial r^2} - V\Psi_2 \tag{34}$$

$$0 = \frac{\partial\Psi_2}{\partial t} - \frac{2}{r}\frac{\partial\Psi_1}{\partial r} - \frac{\partial^2\Psi_1}{\partial r^2} + V\Psi_1 \tag{35}$$

$$0 = \frac{2}{r}\frac{\partial V}{\partial r} + \frac{\partial^2 V}{\partial r^2} - \Psi_1^2 - \Psi_2^2 \tag{36}$$

With boundary and initial conditions:

$$\frac{\partial \Psi_1(0,t)}{\partial r} = 0 \tag{37}$$
$$\frac{\partial \Psi_2(0,t)}{\partial r} = 0 \tag{38}$$
$$\frac{\partial V(0,t)}{\partial r} = 0 \tag{39}$$
$$\Psi_1(r_{max},t) = 0 \tag{40}$$
$$\Psi_2(r_{max},t) = 0 \tag{41}$$
$$V(r_{max},t) = 0 \tag{42}$$
$$\Psi_1(r,0) = 10\exp(-r^2) \tag{43}$$
$$\Psi_1(r,0) = 0 \tag{44}$$

The boundary conditions presented above are obviously unsuitable for long time evolution; at large times we expect outgoing portions of the field to reflect off the boundary at r_{max}. This could be fairly simply resolved by introducing compactified coordinates $x = r/(1+r)$ and effectively moving the boundary to infinity.

Unfortunately, performing a straight forward Crank-Nicolson discretization on these equations, results in a method that is unstable for $\Delta t \gtrsim \Delta x^2$. Although the Crank-Nicolson method itself is unconditionally stable for the Schrödinger equation [18], here we have encountered a problem due to the fact that Equations 34 and 35 are not diagonally dominant in Ψ_1 and Ψ_2 for large time steps.

This can be rectified by solving the discretized versions of Equations 34 and 35 for $\Psi^{n+1}_{1i,j}$ and $\Psi^{n+1}_{2i,j}$ and substituting into the opposite expression to obtain Poisson like equations for $\Psi^{n+1}_{1i,j}$ and $\Psi^{n+1}_{2i,j}$ which are independent of the advance time level (at (i,j)) for the opposite field.

Due to this field diagonalization, the expressions for the residuals and Jacobians are quite involved and I will not repeat them here. The interested reader should examine the Schrödinger-Newton code to see exactly how the time levels are handled in the `f_jacobi` and `f_opp` methods.

The main loop of the previous program becomes:

```
int main() {
   int num_l;
   int num_f;
   int dim;
   int i, j, k, n;
   int size_base[1];
   double bbox[2];
   double *s1_n, *s2_n, *V_n, *s1_np1, *s2_np1, *V_np1;
   double* x, dx;

   num_l = 11;
```

```
num_f = 6;
dim = 1;
size_base[0] = 3;
bbox[0] = 0;
bbox[1] = 10;

Hierarchy* hierarchy = new Hierarchy(num_l, num_f, dim, size_base, bbox);

n = hierarchy->fields[num_l - 1][0]->n;
s1_n   = hierarchy->fields[num_l - 1][0]->f1d;
s2_n   = hierarchy->fields[num_l - 1][1]->f1d;
V_n    = hierarchy->fields[num_l - 1][2]->f1d;
s1_np1 = hierarchy->fields[num_l - 1][3]->f1d;
s2_np1 = hierarchy->fields[num_l - 1][4]->f1d;
V_np1  = hierarchy->fields[num_l - 1][5]->f1d;
x = hierarchy->coords[num_l - 1]->mesh[0]->f1d;
dx = x[1] - x[0];
for (i = 0; i < n; i++) {
   s1_n[i]   = 10.0*exp(-x[i]*x[i]);
   s1_np1[i] = 10.0*exp(-x[i]*x[i]);
}
```

Unlike Ψ_1 and Ψ_2 we are not free to choose V at $t = 0$. Before we can time advance the system we must solve for V

```
double resid = 1;
while(resid > 1e-8) {
   hierarchy->vCycle(4, 0.0, f_schrodinger_newton_opp_initial_1d,
           f_schrodinger_newton_jac_initial_1d);
   printf("change %e, residual %e\n", hierarchy->change[num_l - 1],
           hierarchy->resid[num_l - 1]);
   resid = hierarchy->resid[num_l - 1];
}
printf("\n");

n = hierarchy->fields[num_l - 1][0]->n;
V_n    = hierarchy->fields[num_l - 1][2]->f1d;
V_np1  = hierarchy->fields[num_l - 1][5]->f1d;
for (i = 0; i < n; i++) {
   V_n[i] = V_np1[i];
}
```

Here the `f_opp` and `f_jac` methods solve the 36 on the advance time step. The time stepping is then handled by iterating until convergence then switching the pointers to the time levels:

```
double dt = 0.02 * dx;
```

```
double time = 0.1;
for (j = 0; j < time/dt; j++)
{
   resid = 1;
   while(resid > 1e-8) {
      hierarchy->vCycle(4, dt, f_schrodinger_newton_opp_1d,
              f_schrodinger_newton_jac_1d);
      printf("change %e, residual %e\n", hierarchy->change[num_l - 1],
              hierarchy->resid[num_l - 1]);
      resid = hierarchy->resid[num_l - 1];
   }
   printf("\n");

   hierarchy->switchFields(0,3);
   hierarchy->switchFields(1,4);
   hierarchy->switchFields(2,5);
}
```

Evolutions of a gaussian pulse at the origin can be seen in Figures 1 and 2. Due to the larger gravitational potential, the evolution shown in Figure 1 implodes inwards while that of Figure 2 disperses.

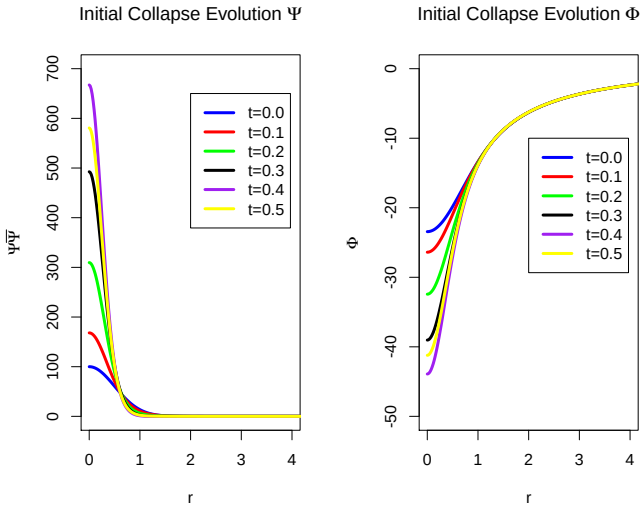

Figure 1: Evolution of an initially collapsing gaussian pulse. At later times the pulse rebounds.

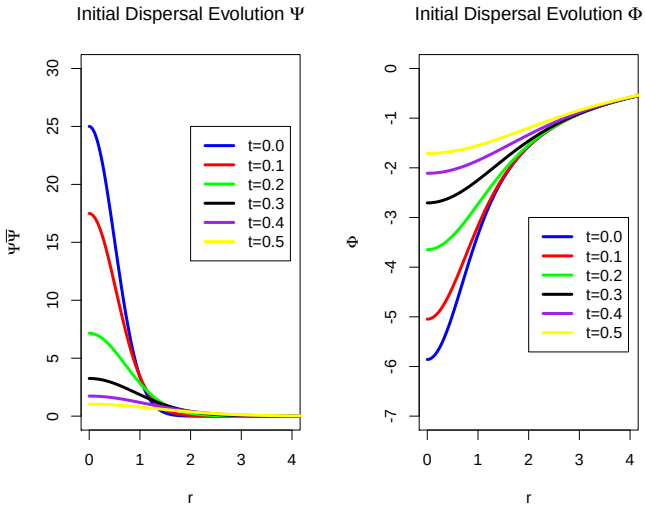

Figure 2: Evolution of an initially dispersing gaussian pulse. The gravitational potential was insufficiently strong to bind the Schrödinger field.

4.3 Other Samples

In addition to the Schrödinger-Newton equation in spherical symmetry and Poisson Equation on a line, the repository [17] contains sample code for the 3D Poisson equation in cartesian coordinates and 1D Schrödinger equation on a line. As the structure on the code is similar to the code presented above, the code is not presented for the sake of conciseness.

4.4 Convergence

Along with the `f_opp` and `f_jac` methods for each of the sample programs, there is also an independent residual evaluator for each sample program. These functions use alternate (not necessarily stable) discretizations of the PDEs which are of at least the same order as the original discretization. When the residuals of these alternate discretizations converge to zero at the expected rate, it demonstrates that the correct problem is being solved correctly [15]. Figure 3 demonstrates the convergence of each of the sample programs.

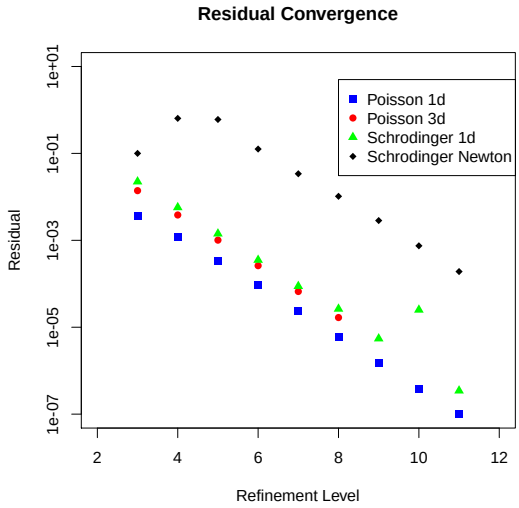

Figure 3: Independent residual convergence of the various sample programs. Each of the programs is observed to converge at second order as expected.

5 Performance

This section quantifies the performance of `MG_OMP` and the `Wave Utility` by running each program in various configurations. Wherever the term "refinement level" is used, it indicates the number of grid points on an edge grid. For example, a refinement level of 10 for the Schrödinger-Newton `MG_OMP` example indicates a 1025 point line, while a refinement level of 8 for the `Wave Utility` indicates a 256x256 point grid.

5.1 `MG_OMP`

`MG_OMP` was tested using the Schrödinger-Newton code for a short term evolution. Each successive refinement level doubles the number of grid points and halves the time-step so that the overall error is $O(\Delta x^2, \Delta t^2) \propto O(\Delta x^2)$. As such, each successive refinement level requires roughly four times the computational resources of the previous level. Figure 4 shows the relative timing performance of `MG_OMP` running in `OpenMP` mode for various number of threads, while Figure 5 demonstrates the relative efficiency of `OpenMP` versus single core execution.

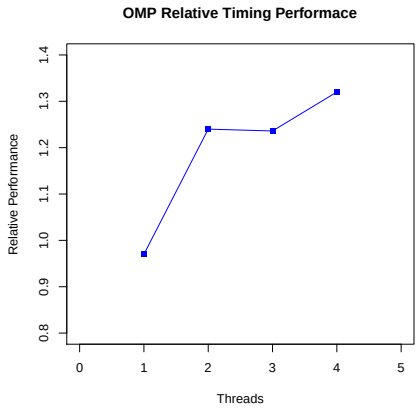

Figure 4: Relative timing performance of the multigrid utility running in `OpenMP` mode versus single core execution on a 1025 point grid. Running the Schrödinger-Newton example on a dual-core i5 processor, the maximum performance increase is $\approx$ 1.32x. This is significantly below the 2x speed increase we might hope to achieve for a dual core processor. For smaller grids, the performance increase is smaller.

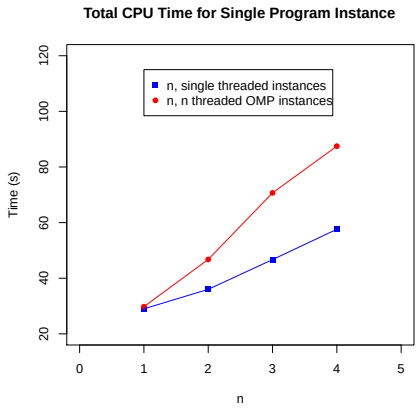

Figure 5: Total CPU time to execute a single instance of the same program as Figure 4. CPU time for `OpenMP` is calculated as *number of threads* x *time to execute*. Compared to running four instance of the single core version simultaneously, the `OpenMP` version with four threads achieves only 65% efficiency. Compared to running four instances of the single core version sequentially, the `OpenMP` version with four threads achieves only 33% efficiency.

5.2 `Wave Utility`

The `Wave Utility` was tested for a wide variety of configurations for `OpenMP`, `MPI` and `OpenACC` versions. Figure 6 demonstrates the performance of the `OpenMP` version of the code while Figures 7 and 8 show the performance of the `MPI` and `OpenACC` versions respectively. The performance in these Figures indicates the timing performance relative to single core execution.

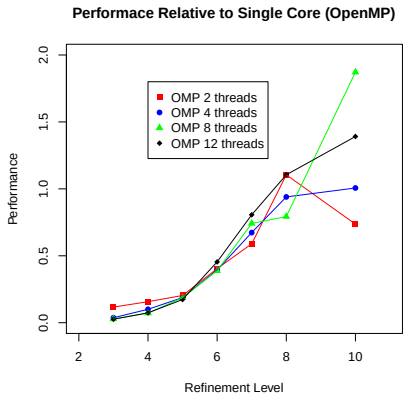

Figure 6: Performance of the Wave Utility relative to a single core. These tests were run on the Parallel cluster at West Grid on a single node consisting of two, six core processors at 2.53GHz and 3 Nvidia Tesla M2070 GPUs. `OpenMP` was unable to break the 2x speed up barrier.

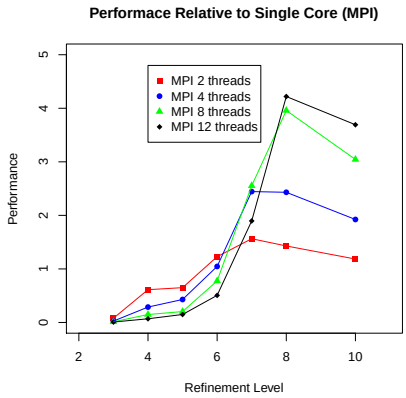

Figure 7: Performance of the Wave Utility relative to a single core. The relative performance of the `MPI` code plateaus around refinement level at approximately 4x the performance of a single core. The relative decrease in performance around refinement level 10 is likely due to the size of the arrays being manipulated; above 512x512, the arrays cannot simultaneously fit in L3 cache.

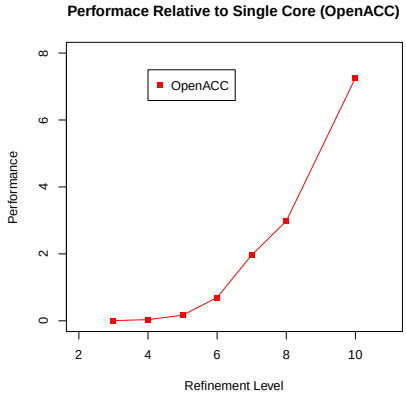

Figure 8: Performance of the Wave Utility relative to a single core. For these tests, only a single GPU was used. As shown in Figure 9, the relative performance of the `OpenACC` code plateaus around refinement level 11 at approximately 8x the performance of a single core.

Figure 9 indicates the amount of time spent processing each grid point relative to the time spent at the eighth refinement level. For larger problems, the `OpenACC` code becomes progressively more efficient while the `MPI` code undergoes a sharp decrease in efficiency around refinement level 10. This decrease is likely due to inefficient caching; at 1024x1024, the grids take up roughly 50Mb

and the L3 cache is only 12Mb. Since my `MPI` code is quite memory inefficient, it is probably possible to ameliorate this issue with better coding practices.

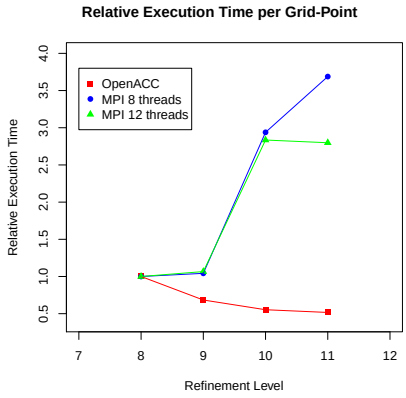

Figure 9: Relative time spent per grid point for different refinement levels. Due to time constraints, I was unable to progress past refinement level 11 (2048x2048), but it can be observed that the Tesla performance will plateau around refinement level 11 or 12 corresponding to about an 8x speed up over single core execution.

Finally, Figure 10 demonstrates the performance of the `MPI` code for a large number of nodes, each running with eight processors active (requesting only 8 or 12 processors seems to permit much faster progression through Parallel's queuing system). In contract to `OpenMP`, it is observed that `MPI` scales extremely efficiently.

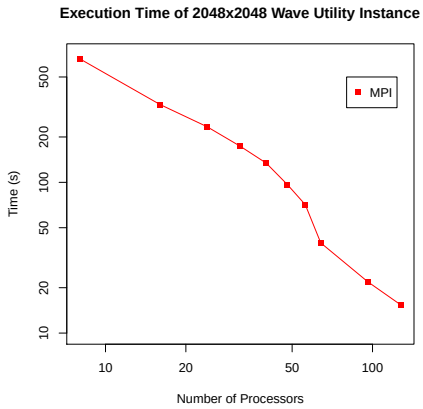

Figure 10: Time scaling of the Wave Utility for large numbers of processors. For a 2048x2048 grid, the execution time scales efficiently up to at least 128 processors. The sharp increase in efficiency around 32 processors is probably due to the grids becoming small enough to fit in L3 cache. For 128 processors, the `MPI` code performs about 100 times faster than the single processor code compiled with `PGI` compilers and 175 times faster than the `MPI` code running on a single processor with `Intel` compilers.

6 Discussion

6.1 `MG_OMP`

`MG_OMP` is an easy to use utility for solving PDEs using the FAS Multigrid Method. Section 4 demonstrates that it is fairly robust and the included samples show how it may be used to solve time evolution problems as well as 1D, 2D and 3D elliptic problems.

As shown in Section 5, enabling `OpenMP` gives a fairly negligible performance increase that could easily be overshadowed by better coding practices or different compiler options. To rectify this shortcoming, I may develop a second version of this utility in `Fortran` which uses `MPI` and makes a distinction between hyperbolic PDEs (which do not generally require a vCycle to solve) and elliptic PDEs. On the other hand, our group already has a code (`PAMR` [19]) which does this. In this light, the only reason to do this would be to eliminate to overhead due to `PAMR`'s adaptive mesh refinement and to expand my own knowledge base.

6.2 `Wave Utility`

Section 5 demonstrates that `OpenMP` is of very limited use for the purpose of easily accelerating PDE code. The maximum speed increase observed was less than two times the speed of a single threaded version of the program and it

can only be run on shared memory devices. As shown with `MG_OMP`, it is far more efficient to run multiple concurrent instances of the single processor code than to use `OpenMP`. This is in line with the performance increases I have seen in typical example code online which seem to typically result in a performance increase of less than 180%.

`OpenACC` appears to be a good comprimize between performance and ease of use. Running on a single node with a high power processor performance increase of 5-10x seem fairly typical [20]. As shown in Section 5, this is probably better than what you would get from using `MPI` due to the memory access bottle neck that accompanies running large grids in parallel on a single node. In addition, `OpenACC` it is quick to setup requiring about as much work as `OpenMP` for `Fortran` code (it is considerably more difficult in C++ as the memory structures can be much more complex). If you have a few powerful GPUs, such as Teslas [11] or Titans [21], `OpenACC` is an attractive alternative to `MPI` for medium sized problems.

Finally we come to `MPI` which Section 5 demonstrates is a powerhouse which scales extremely efficiently. It is more difficult to program than either `OpenMP` or `OpenACC` but the performance increase can be well worth it for large problems.

It appears that there is really no reason to write a GPU version of our group's `PAMR` [19] code or a hybrid model which uses both `MPI` and `OpenACC`. As shown in Section 5, `OpenACC` delivers a performance increase approximately equal to that of MPI when running on all processors of a single node. For this reason there is no real benefit to using a hybrid model.

In closing, I would like to make a statement to any student entering a degree program in computational physics; you should really just go ahead and learn `Fortran` and `MPI`, it will save you a lot of pain over trying use `C++`.

References

[1] D. Giulini and A. Groardt, "The schrödingernewton equation as a non-relativistic limit of self-gravitating kleingordon and dirac fields," *Classical and Quantum Gravity*, vol. 29, no. 21, p. 215010, 2012. [Online]. Available: http://stacks.iop.org/0264-9381/29/i=21/a=215010

[2] F. S. Guzmán and L. A. Ureña López, "Evolution of the schrödinger-newton system for a self-gravitating scalar field," *Phys. Rev. D*, vol. 69, p. 124033, Jun 2004. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevD.69.124033

[3] D.-I. Choi, "Numerical Studies of Nonlinear Schrödinger and Klein-Gordon Systems: Techniques and Applications," Ph.D. dissertation, UT Austin, 1998.

[4] E. Seidel and W.-M. Suen, "Dynamical evolution of boson stars: Perturbing the ground state," *Phys. Rev. D*, vol. 42, pp. 384–403, Jul 1990. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevD.42.384

[5] ——, "Formation of solitonic stars through gravitational cooling," *Phys. Rev. Lett.*, vol. 72, pp. 2516–2519, Apr 1994. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevLett.72.2516

[6] W. Hu, R. Barkana, and A. Gruzinov, "Fuzzy cold dark matter: The wave properties of ultralight particles," *Phys. Rev. Lett.*, vol. 85, pp. 1158–1161, Aug 2000. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevLett.85.1158

[7] D. F. Torres, S. Capozziello, and G. Lambiase, "Supermassive boson star at the galactic center?" *Phys. Rev. D*, vol. 62, p. 104012, Oct 2000. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevD.62.104012

[8] W. Gropp. Tutorial on mpi: The message-passing interface.

[9] Intel mpi library. [Online]. Available: https://software.intel.com/en-us/intel-mpi-library

[10] Parallel programming and computing platform. [Online]. Available: http://www.nvidia.ca/object/cuda_home_new.html

[11] Supercomputer workstations with tesla k40 and k20 gpus. [Online]. Available: http://www.nvidia.com/object/tesla-workstations.html

[12] Openacc: Directives for accelerators. [Online]. Available: http://www.openacc-standard.org/

[13] (2014) Parallel. [Online]. Available: https://www.westgrid.ca/support/quickstart/parallel

[14] Pgi compilers tools. [Online]. Available: http://www.pgroup.com/index.htm

[15] M. Choptuik, "Lectures for VII Mexican School on Gravitation and Mathematical Physics," 2006. [Online]. Available: http://laplace.physics.ubc.ca/People/matt/Teaching/06Mexico/mexico06.pdf

[16] A. Borzi, "Introduction to multigrid methods." [Online]. Available: http://laplace.physics.ubc.ca/People/matt/Teaching/06Mexico/mexico06.pdf

[17] Phys 410 upgrade.

[18] I. Puzynin, A. Selin, and S. Vinitsky, "A high-order accuracy method for numerical solving of the time-dependent schrödinger equation," *Computer physics communications*, vol. 123, no. 1, pp. 1–6, 1999.

[19] F. Pretorius. Pamr reference manual.

[20] M. Harris. An openacc example(part 2).

[21] C. Lupo. Geforce gtx titan - the ultimate cuda development gpu. [Online]. Available: http://users.csc.calpoly.edu/~clupo/media/GTXTitan.pdf