PDE Solvers for Laplace's Equation

Izhar ul Hassan, 830118-6535

March 25, 2011

1 Introduction

Partial differential equations are used to formulate and model a variety of physical systems such as fluid flow, propagation of waves and weather prediction. Simple PDE's in simple geometric regions can be solved analytically, however, for more practical problems, the equations are solved by discretizing the problem domain using iterative numerical methods.

In this project we have solved Laplace's equation in two dimensions using Jacobi Iterations, Gauss-Seidel Scheme (GS) and using Multigrid Methods. Laplace equation is a second order PDE which is used in the modeling of electricity, fluid flow and steady head conduction. A typical Laplace equation is represented as shown in Equation 1.

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 \tag{1}$$

Function ϕ represents some unknown potential such as heat or stress. The system is represented computationally by a matrix where the spatial region and the values for the boundaries are given and the goal is to approximate the values for the interior points. For our convenience, we have initialized the boundaries to being all ones and the interior points being all zeros. The steady-state solution is then calculated by repeated iterations to find the values of the interior points. We have assumed the grid to be an n x n square and it is surrounded by a boundary of all ones. The computation terminates when the change in the new value from the old value becomes smaller than a given EPSILON.

We have developed sequential programs for solving the PDE. Parallel implementation of these three methods have been implemented with Pthreads library using shared-memory programming model.

2 Implementation

2.1 Sequential Jacobi Iteration

Jacobi iteration uses a stencil to iterate through the matrix. New values for each grid point are calculated as the average of the old values of the grid points to the left, right, above and below it. Since we have assumed the grid to be an n x n square, and accommodating the boundary layers the dimensions would become n+2 x n+2. We would require two matrices to represent the grid and the newly calculated values. We initialize both these matrices to the initial values and the boundary conditions. The main computation for Jacobi Iteration is gives as follows:

```
 while (True)
 {
        //compute the new inner points
        for (i=1; i<n-1; i++)
        {
            for (j=1; j<n-1; j++)

            {
                new[(i*n)+j] = (grid[((i+1)*n)+j] + grid[((i-1)*n)+j] +
                grid[(i*n)+j-1] + grid[(i*n)+j+1])/4
            }

        }
        iters++;
        //compute the maximum difference
        for (i=1; i<n; i++)
        {
        for (j=1; j<n; j++)
        {
            maxdiff = max(maxdiff, absolute(1 - grid[(i*n)+j]));
        }
    }
    //check for termination
    //copy new to grid to prepare for next update
}
```

2.2 Optimizations

The performance of the above code can be greatly improved by applying some optimizations. The first for loop is executed n^2 times and there are n^2 additions and divisions by 4. We start with *strength reduction* by replacing the division with multiplication by 0.25 which takes fewer machine cycles as compared to division. The code to calculate the maximum difference is executed on every iteration of the while loop. We move this part of the code out of the main loop and replace the while loop with a for loop for a fixed number of iterations. The number of iterations can always be increased to suit the required level of accuracy. Next we consider the part of code that copies the new matrix to the old one. We could avoid this copying by introducing a 3-dimensional n x n x 2 and represent the grid and new matrix in this 3d matrix. Now, instead of copying the new matrix to grid, we could simply swap the roles of the dimensions. This however, results in calculating extra indices on every iteration. A better idea to avoid this could be *loop unrolling.* We repeat the loop body one more time and divide the number of iterations by 2. This would also take care of the copying as seen in the code snippet below.

```
for (k=0; k<(num_iter/2); k++)
{
        //compute the new inner points
        for (i=1; i<n-1; i++)
        {
            for (j=1; j<n-1; j++)
            {
                new[(i*n)+j] = (old[((i+1)*n)+j] + old[((i-1)*n)+j] +
                old[(i*n)+j-1] + old[(i*n)+j+1])*0.25;
            }
        }
        //loop unrolling
        for (i=1; i<n-1; i++)
        {
            for (j=1; j<n-1; j++)
            {
                old[(i*n)+j] = (new[((i+1)*n)+j] + new[((i-1)*n)+j] +
                new[(i*n)+j-1] + new[(i*n)+j+1])*0.25;

            }
        }
}
```

2.3 Parallel Implementation of Jacobi Iteration

Let P be the number of processors available and n be the dimensions of the grid. We assume that n is much larger than P and that n may not be a multiple P. We divide the grid into a number of horizontal strips and assign them to the processors. If n is not perfectly divisible by the number of processors, the residue is assigned to the last process, as shown in the code.

```
residue = n % numWorkers;
    height = (n-residue) / numWorkers;
    myFirst = (thread_id * height) +1;
    myLast = (thread_id * height)+height;
    if(thread_id==numWorkers-1)
        myLast = myLast+residue;
```

Each processor updates its strip of points. We used the optimized code from the serial version of Jacobi Iteration and modified it so that the strips are divided among the processors and added thread synchronization by using barriers as show in the code.

```
for (k=0; k<(num_iter/2); k++)
{
        //compute the new inner points
        for (i=myFirst; i<=myLast; i++)
        {
            for (j=1; j<n-1; j++)
            {
                new[(i*n)+j] = (old[((i+1)*n)+j] + old[((i-1)*n)+j] +
                old[(i*n)+j-1] + old[(i*n)+j+1])*0.25;
            }
        }
        Barrier();
        //loop unrolling
        for (i=myFirst; i<=myLast; i++)
        {
            for (j=1; j<n-1; j++)
            {
                old[(i*n)+j] = (new[((i+1)*n)+j] + new[((i-1)*n)+j] +
                new[(i*n)+j-1] + new[(i*n)+j+1])*0.25;
            }
        }
}
```

Maximum Difference is calculated by each thread calculating a local maximum difference and storing the result in a shared array. Once all threads are finished, we find the maximum difference from the shared array. Jacobi Iteration is slow to converge. We have also developed *Serial and Parallel versions of Gauss Seidel Scheme*. The Gauss-seidel method converges faster and uses less space as it does not depend on the old values.

```
grid[(i*n)+j] = (grid[((i+1)*n)+j] + grid[((i-1)*n)+j] +
                grid[(i*n)+j-1] + grid[(i*n)+j+1])*0.25;
```

2.4 Serial Multigrid Method

The execution time of approximating a PDE depends upon the size of the grid. The greater the size of the grid, the longer it will take for the computation to finish. Smaller grids are solved more readily but they do not capture the same level of details as a finer grid. Multigrid method is an approach where the idea is to use grids of different sizes and to switch between them to increase the convergence rate. It consists of the following steps:

- Smoothing: Start with the finest grid and update the grid using one of the relaxation methods e.g. jacobi.
- Restriction: Restrict the fine grid to a coarser grid by using a smoothing operator e.g. a coarser grid could have points which are twice as far apart as the fine grid.
- Solution: Solve the coarse grid to the desired accuracy.
- Interpolation: Interpolate the coarse grid back to the fine grid using an interpolation operator. We used bilinear Interpolation.
- Smoothing: Update the fine grid for a few iterations again.

There are several kinds of multigrid methods each using a different pattern of coarse grid correction. In this project we have used the single correction which starts with the fine grid, iterates for a few iterations, restrict to a coarse grid to solve the problem to desired accuracy and then interpolate back to fine grid and smooth the fine grid. This is called a V-Cycle. We have used several such V-cycles to converge the solution.

2.5 Parallel Multigrid

Multigrid methods converge more rapidly as compared to other basic iterative methods but they are much more harder to program as they require extensive book-keeping. In particular, the parallel version is even more difficult as the work has to be sub-divided between the threads. We used the same ideas from our parallel implementations of Jacobi Iterations and Guass-Seidel Scheme to parallelize multigrid methods. The grids were divided into strip assigned to different processes.

Table 1: Serial versions of Jacobi Iteration and Multigrid Methods

Method	Grid Size	Iterations	Time (sec)	Max Diff.
Jacobi	100	1000	1.011	0.906
Jacobi	200	1000	4.156	0.999
Gauss-Seidel	100	1000	1.014	0.607
Gauss-Seidel	200	1000	4.305	0.994
Multigrid	100	1000	2.009	0.000
Multigrid	200	1000	8.177	0.030

3 Performance Evaluation

We executed the programs on a shared-memory multiprocessing (smp) system called key.pdc.kth.se. Key consists of 32 1.6 GHz cores of IA64 (Intel) type with 18 MB cache. The total main memory is 256 GB. Fig.1 shows the timings for Jacobi Iteration and Multigrid methods. Both Jacobi and Multigrid programs were executed with grid sized of 100 and 200 points. The number of iterations for Jacobi were 1000 while those for the multigrid were 1000 on the coarse grid which was executed 8 times (8 v-cycles). From fig.1 Jacobi Iteration is faster than multigrid methods because of the fact that multigrid method has more iterations to take care of. As we increase the number of threads, the timing tends to decrease. We have quite similar curves for Jacobi Iterations as well. This is due to the fact that multigrid methods use jacobi iterations for the solution of coarse grids.

The speedup analysis for both of these algorithm is shown in fig.2. As we can observe that the Jacobi Iteration with maximum load shows the best speedup, almost linear. With a smaller payload the Jacobi Iteration still achieves good speedup. Although we can see that the speedup will decrease if we increased the number of threads beyond 4. Multigrid methods are much more harder to program and it is not easy to achieve good speedup due to the extensive book-keeping. The only case where multigrid methods would outperform faster is when the payload is very large. With smaller grid size the speedup is quite poor. This can be easily observed in fig.2 where the multigrid method with the

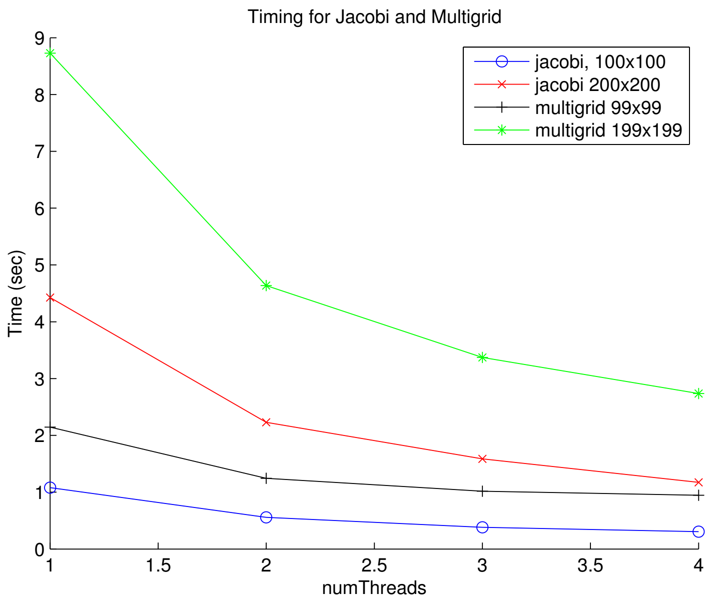

Figure 1: Timing for Jacobi and Multigrid

smaller grid size performs very poor. This is even more obvious if we look at fig.3 and fig.4. We get a small speedup with grid size of 24 with 2 threads, but an increase in the number of threads result in slower performance as the payload is not big enough to test a higher number of processors.

4 Conclusion

In this project we have solved the Laplace's equation with Jacobi Iteration, Gauss-Seidel Scheme and Multigrid methods. Jacobi Iteration and Gauss Seidel scheme are the easiest to program and we achieve very good speedups with parallel versions of these programs. Gauss-Seidel scheme in particular is much faster to converge and it requires much smaller space. The multigrid methods are even faster to solve the problem however they are much more difficult to program. They require extensive book-keeping which is hard to manage especially when the number of coarse grids is increased. The book-keeping also proves to be a hinderance in achieving a good speedup. This is particularly obvious when the size of the grid is not big enough. Multigrid methods would always be the algorithm of choice for huge grid sizes.

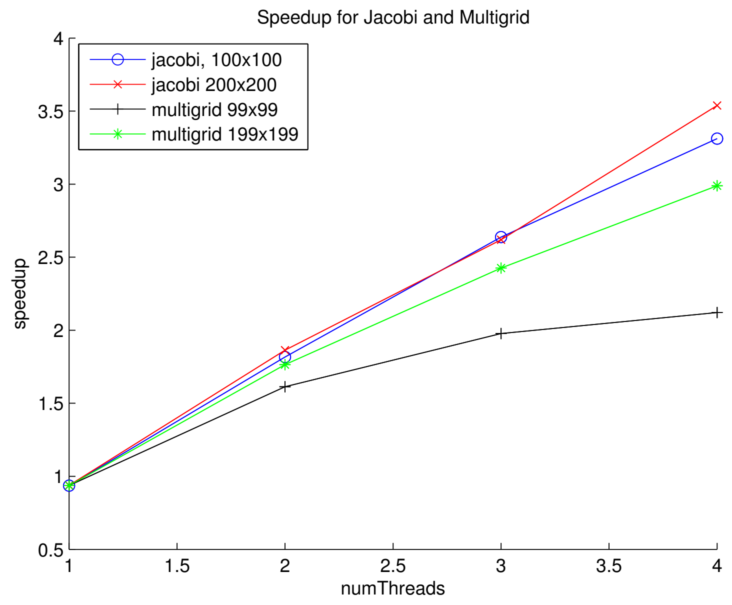

Figure 2: Speedup for Jacobi and Multigrid

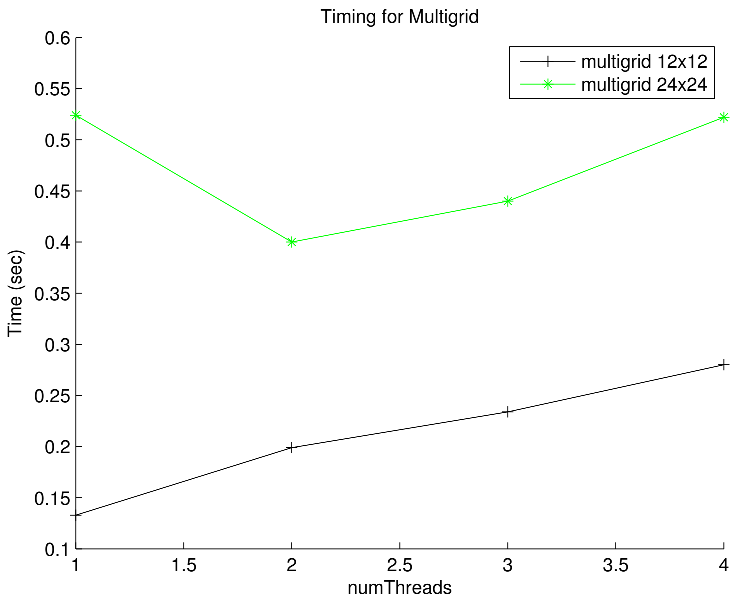

Figure 3: Timing for required smaller grid sizes

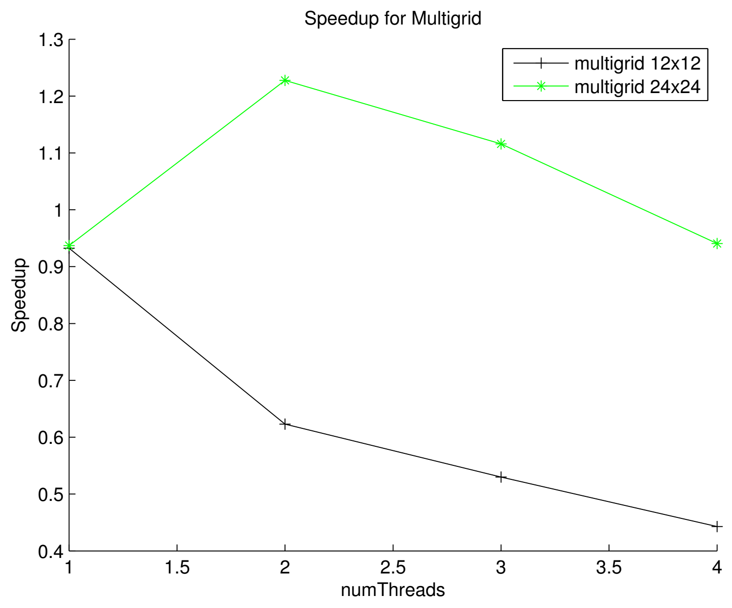

Figure 4: Speedup for required smaller grid sizes