

High Performance Multigrid for Poisson's Equation in 3D

S. Nicholas Barkas

Negar Safinianaini

May 27, 2009

HPC Summer School 2008

Reviewer: Michael Hanke

Abstract

We introduce a model Poisson equation in 3D, and describe the multigrid algorithm we wish to use for solving this problem. An initial program is written, then, through testing and profiling, we iteratively improve the areas where we find performance bottlenecks. Experimentation is done with highly tuned vector functions from level 1 BLAS in an attempt to reduce run time further. Finally, computationally intensive parts of the program are parallelized using shared memory to run on separate CPU cores, with OpenMP. Overall floating point performance is measured and compared with the theoretical maximum performance on our test hardware, and additional considerations with respect to memory and cache usage, etc. are discussed.

1 Introduction

Multigrid methods are very fast algorithms for solving linear (and also sometimes non-linear) systems of equations. Typically, these are systems arising from discretization of partial differential equations, especially elliptic PDEs, but algebraic multigrid methods can be used for solving more general algebraic systems of equations. In this project we are focused on the former type of system, specifically Poisson's equation in three dimensions.

One can solve this system using a number of different methods. Some choices include iterative methods like Gauss-Seidel, which runs in $\mathcal{O}(n^2)$ time, and direct solvers like sparse Cholesky, which runs in $\mathcal{O}(n^{3/2})$ time. Even better are fast Poisson solvers based on the fast Fourier transform, which run in $\mathcal{O}(n \log n)$ time [2]. We still would like a faster method though, given that in 3D the number of grid points we have grows very rapidly; if we have N grid points in each dimension, the total number of unknowns $n = N^3$. Multigrid provides us with this faster method. It runs in linear time, $\mathcal{O}(n)$, and is thus an optimal algorithm for this problem.

In the following sections we shall provide a brief overview of how the multigrid algorithm works for a model 3D Poisson equation, followed by a discussion of how we implemented a high speed, multi-threaded multigrid solver in C with OpenMP. Then we shall finish up with some performance analysis of solving our model problem with this code using an 8-core system, one of PDC's nodes in the Ferlin cluster.

1.1 Model Problem

As a model problem to try our multigrid algorithm on, we shall solve the following 3D Poisson equation:

$$\begin{aligned} -\Delta u(x, y, z) &= \sin(x) + y^2 - 3z, \quad (x, y, z) \in [0, 1] \times [0, 1] \times [0, 1], \\ u &= 0 \text{ on boundaries.} \end{aligned} \tag{1}$$

If we discretize the Laplacian operator with second order finite differences, we get a linear system of equations

$$A\mathbf{x} = \mathbf{b}. \tag{2}$$

\mathbf{b} is a vector containing the right hand side of (1) evaluated at each non-boundary grid point, in lexicographical order. If we have N grid points in each dimension with equal grid point spacing $h = 1/(N + 1)$, and we write the right hand side as function $f(x, y, z) = \sin(x) + y^2 - 3z$, then $\mathbf{b} = (f(h, h, h), f(h, h, 2h), \dots, f(h, h, Nh), f(h, 2h, h), \dots, f(Nh, Nh, Nh))^T$. \mathbf{x} is our unknown solutions at the grid points ordered the same way as \mathbf{b} , and A is a sparse banded matrix that represents the discrete Laplacian operator in 3D. One relatively concise way to describe A is with Kronecker products of simpler matrices. If we write the $N \times N$ tridiagonal matrix

$$T = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix},$$

then

$$A = \frac{1}{h^2}(T \otimes I \otimes I + I \otimes T \otimes I + I \otimes I \otimes T), \tag{3}$$

where I is the $N \times N$ identity matrix.

2 Algorithms

Multigrid algorithms are made up of several different components, including iterative solvers for linear systems, grid restriction, and interpolation operations. We shall give an overview of these components before moving on to describing the multigrid algorithms themselves.

2.1 Iterative Solvers

As mentioned in the introduction section, it is possible to solve a linear system such as (2) with a number of different methods. Some of these methods work iteratively, improving the estimated solution with each additional iteration of the algorithm. Two of the most basic iterative algorithms are the weighted Jacobi and Gauss-Seidel methods. Both of these methods are good at getting rid of high frequency errors in a solution in just a few iterations, but take many more iterations to remove low frequency errors. For this reason, these types of solvers are referred to as smoothers when one is speaking in the context of multigrid—a few weighted Jacobi or Gauss-Seidel iterations smooth out high frequency errors.

Weighted Jacobi and Gauss-Seidel methods are both commonly used as smoothers in multigrid. A Gauss-Seidel iteration converges in half as many iterations as weighted Jacobi, but Jacobi is easiest to parallelize. We have implemented a Gauss-Seidel smoother using red-black ordering, which also is possible to parallelize. Much more detail about iterative solvers can be found in [2] and numerous other resources.

2.2 Grid Restriction and Interpolation

The next multigrid components to discuss are the grid transfer operators, restriction and interpolation. Multigrid gets its name from the fact that it utilizes grids with differing mesh sizes to solve problems on one of those grids. As such, it is necessary to transfer values from fine grids to coarser grids, and from coarse grids to finer ones. The former operation can be referred to as restriction, while the latter is often called interpolation.

In our implementation, when transferring from a fine grid to a coarse one, we use a full-weighting restriction operator. Instead of just copying the fine grid points that line up with coarse grid points to their corresponding place on the coarse grid, we set each coarse grid point to a weighted average of the corresponding fine grid point and its nearest neighbours. For transferring from a coarse grid to a fine one, we use a trilinear interpolation operator. Coarse grid points that correspond directly to a fine grid point are copied directly, while those fine grid points that lie between points on the coarse grid are set by averaging over the corresponding nearest neighbours from the coarse grid.

2.3 Multigrid Algorithms

Now we move on to describing the multigrid algorithms themselves. There are two algorithms we discuss here, the multigrid v-cycle and the full multigrid method. The multigrid v-cycle is used within full multigrid, but it also can be used on its own as an iterative solver. Iteratively solving with multigrid v-cycles is not quite as fast as full multigrid, though, so we have implemented a full multigrid solver.

Algorithm 1 gives a pseudo-code description of how the multigrid v-cycle works. First, we do one or more pre-smoothing steps using our iterative solver, or smoother, to smooth out high frequency errors in the initial estimated solution (or computed solution from the previous iteration). Our specific implementation uses two of these pre-smooth steps. Next we calculate the residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}^*$ using our approximate solution \mathbf{x}^* . With this we can write a new linear system, $\mathbf{A}\mathbf{e} = \mathbf{r}$, where \mathbf{e} is the error in our approximate solution \mathbf{x}^* : $\mathbf{e} = \mathbf{x} - \mathbf{x}^*$. But rather than trying to solve the system for the error directly, we restrict the residual down to a coarser grid to solve for the coarse grid error: $A_{coarse}\mathbf{e}_{coarse} = \mathbf{r}_{coarse}$. The matrix A_{coarse} is formed the same way as (3), but h is doubled to account for the coarser grid spacing. We can achieve the same effect by multiplying the coarse grid right hand side \mathbf{r}_{coarse} by four. Solving for the coarse grid error is done through a recursive call to the multigrid v-cycle function itself, or if we are unable to coarsen the grid any further, we solve the system using a more traditional solver. In our

Algorithm 1: Multigrid v-cycle: improves solution \mathbf{x} to linear system $A\mathbf{x} = \mathbf{b}$. Call $\text{mgv}(x, b)$

Arguments: Solution grid x , right hand side grid b

Parameters: Matrix A , number of pre-smoothing steps ν_1 , number of post-smoothing steps ν_2

```
// Pre-smoothing
for  $i \leftarrow 1$  to  $\nu_1$  do
    smooth( $x, b$ )
end
// Calculate residual and coarsen it
 $r \leftarrow b - Ax$ 
 $r_{coarse} \leftarrow \text{restrict}(r)$ 
 $e_{coarse} \leftarrow 0$ 
if Coarsest grid then
    // Solve coarse grid system exactly  $A_{coarse}e_{coarse} = r_{coarse}$ 
    solve( $e_{coarse}, r_{coarse}$ )
else
    // Recursively call ourself
    mgv( $e_{coarse}, r_{coarse}$ )
end
 $e \leftarrow \text{interpolate}(e_{coarse})$ 
// Correct using interpolated coarse error
 $x \leftarrow x + e$ 
// Post-smoothing
for  $i \leftarrow 1$  to  $\nu_2$  do
    smooth( $x, b$ )
end
```

Algorithm 2: Full multigrid: solves linear system $A\mathbf{x} = \mathbf{b}$. Call $\text{fmg}(x, b, n)$

Arguments: Solution grid x , right hand side grid b , number of levels n

Parameters: Number of fine grid v-cycles ν_3

```
// Repeatedly coarsen right hand side  $b$  down to coarsest grid
 $b_0 \leftarrow b$ 
for  $i \leftarrow 1$  to  $n$  do
     $b_i \leftarrow \text{restrict}(b_{i-1})$ 
end
// Solve coarsest grid system exactly
solve( $x_{coarse}, b_n$ )
// Repeatedly interpolate coarse solution to fine grid and do v-cycles
for  $i \leftarrow n - 1$  to  $0$  do
     $x \leftarrow \text{interpolate}(x_{coarse})$ 
    mgv( $x, b_i$ )
     $x_{coarse} \leftarrow x$ 
end
// Do  $\nu_3$  extra v-cycles on the fine grid solution
for  $i \leftarrow 1$  to  $\nu_3$  do
    mgv( $x, b$ )
end
```

implementation this is done with an iterative Gauss-Seidel solver. Gauss-Seidel will converge quickly if we are able to make the coarsest grid have very few points.

These steps correspond to the downward arm on the left side of the letter V, hence the name v-cycle. Now that we have the error on the coarsest grid, we begin to go up on the right side of the V, unravelling the recursion stack to finer and finer grids. We interpolate the coarse grid error back to the next finer grid, and add it to the approximate solution \mathbf{x}^* to bring it closer to the true solution \mathbf{x} on that grid. Finally, we do a few steps (again, two in our implementation) of post-smoothing to further improve the solution.

Full multigrid is shown in algorithm 2. After restricting down the right hand side and finding the solution on the coarsest grid possible, one can imagine a series of V shapes ever increasing in size that represent doing a v-cycle on grids of increasing fineness, then interpolating its solution to the next finest. Finally, we may need to do a few more v-cycles to further improve the solution on the finest grid. In our implementation, we use five of these extra v-cycles to hone our solution. More information about multigrid algorithms, as well as restriction and interpolation operators, can be found in [1], [2], and [5], or for a more in depth treatment, [7].

3 Serial Implementation Details

We first implemented a serial multigrid solver in C, focusing more on code that was easy to understand and produced correct results than on speed. James Demmel’s sample Matlab code [3] for a 2D multigrid Poisson solver was used to help get started with this process. After our first version was working properly, we profiled our program and iteratively tried to improve the slow functions to decrease overall run time. In all cases for these tests, we ran the code on a single node in the Ferlin cluster at PDC. Each of these nodes has two 2.66GHz quad core Intel Xeon E5430 processors with 8GB of RAM¹, and runs Linux. We stored grid values as single precision floating point numbers, so that each grid cell would take up only four bytes of memory rather than the eight bytes used by double precision. This allowed us to use less memory and provides other advantages, as shall be discussed in more detail later.

Initially running our program without any optimization besides those from the compiler’s (gcc 4.1.2) -O3 flags and disabling assertions with -DNDEBUG, we got the run times shown in table 1 for $N \times N \times N$ grids with N ranging in value from 15 to 511. On each of these grids, the size was chosen such that the coarsest grid has only a single non-boundary point. The number of levels used by full multigrid (and in the finest grid’s v-cycle) is $\log_2(N + 1)$.

Table 1: Run times for untuned code at various grid sizes

N	Wall time	Ratio to previous time	Residual norm
15	0.004	N/A	5.727×10^{-7}
31	0.052	11.9	1.241×10^{-6}
63	0.313	6.0	3.424×10^{-6}
127	2.657	8.5	9.608×10^{-6}
255	21.720	8.2	2.667×10^{-5}
511	175.478	8.1	6.819×10^{-5}

As we increase the grid size we see that the size of the residual increases slightly, but remains quite small, indicating that our solution is fairly accurate. We see also that the ratio of run times to the run time with the previous smaller grid size settles down to around 8 when we get to the larger grids. We are roughly doubling the number of grid points in each dimension as we go to larger grids. This doubling is done in three dimensions, so the total number of grid points increases also by a factor of 8. This demonstrates that the asymptotic computational cost of our multigrid code does seem to be linear, as we desire.

¹ <http://www.pdc.kth.se/resources/computers/ferlin/ferlin-hardware>

3.1 Profiling

We used the `gprof` profiler on our code to determine which functions were taking up the most time. With our initial code using a grid of size $N = 255$, the functions taking more than 1% of CPU time are shown in table 2.

Table 2: Amount of time spent in most time-consuming functions of untuned code, according to `gprof`. Grid size $N = 255$ in each dimension.

% time	total time (s)	function name
51.33	11.01	<code>grid_get()</code>
20.98	4.50	<code>smooth_rbgs()</code>
9.09	1.95	<code>grid_interpolate_trilinear()</code>
7.04	1.51	<code>grid_residual()</code>
4.48	0.96	<code>grid_diff_l2_norm_sq()</code>
2.94	0.63	<code>grid_restrict_fw()</code>
1.54	0.33	<code>grid_set()</code>

3.2 Algorithm Improvements

By far the slowest function shown in table 2 is `grid_get()`. This function is relatively simple, but it is called each time we need to read a grid value from any of the grids for the solution, right hand side, error, or residual. We store our grids in a simple data structure that contains the number of grid cells in each dimension (rows, columns, and sheets) along with a one dimensional array that stores each grid cell's value, ordered lexicographically. `grid_get()` and `grid_set()` make it more convenient to access or set these grid points by computing the offset in the array of the desired grid point according to its coordinates i , j , and k .

In this initial version of our program, we do not store the boundary grid points explicitly. Instead, `grid_get()` checks if a boundary point is requested and returns a zero if it is, to satisfy the homogeneous Dirichlet boundary conditions specified in (1). Because this function is called so often, the cost of branching for the if statement gets quite expensive. We modified our code base to instead store the grid points at the boundaries explicitly so that we could get rid of this range check. This requires extra memory to store two extra grid points in every dimension, but the difference is not that large and the speed improvement is substantial. On a grid with 511 points in each dimension, the total runtime of our code improved from the approximately 175 seconds shown in table 1 to about 115 seconds. At this point both `grid_get()` and `grid_set()` are very simple, one line functions, so we decided to replace them with C preprocessor macros. This removed the overhead of making so many function calls, and further reduced the running time to an even greater degree. Again on our grid of 511 points in each dimension, the run time was now down to about 47 seconds.

Most of the other time consuming functions were not easy to speed up by any obvious changes to the algorithms used, but we were able to rework `grid_interpolate_trilinear()` to loop over coarse grid cells, setting multiple fine grid cells on each iteration, rather than looping over each fine grid cell. This further reduced run time on a grid of size 511 down to about 40 seconds. After all of these changes, the most time consuming functions reported by `gprof` are shown in table 3.

3.3 BLAS

The next thing tried to make our code faster was to replace some of our code with calls to highly tuned library functions. The two most time consuming functions shown in 3, `smooth_rbgs()` and `grid_residual()`, both were written originally using three nested loops to do their computations. The inner loops could be replaced fairly easily with some calls to level 1 BLAS functions that operate on entire vectors: `axpy()`, `copy()`, and `scal()`. We linked with the latest version (11.0.074) of the Intel Math Kernel Library installed on Ferlin in order to have access to the needed BLAS functions.

Table 3: Amount of time spent in most time-consuming functions of code where boundary points are stored explicitly, `grid_get()` and `grid_set()` are replaced by macros, and `grid_interpolate_trilinear()` has been changed to do more work in each loop iteration, with about eight times less loop iterations done. Grid size $N = 255$ in each dimension.

% time	total time (s)	function name
43.91	1.93	<code>smooth_rbgs()</code>
32.08	1.41	<code>grid_residual()</code>
9.56	0.42	<code>grid_interpolate_trilinear()</code>
5.92	0.26	<code>grid_restrict_fw()</code>
4.55	0.20	<code>grid_add_to()</code>

It was found that using the BLAS calls in both `smooth_rbgs()` and `grid_residual()` actually slowed down our program, especially on the smaller grids. On a grid of size 63, the code vectorized with BLAS calls took 0.173 seconds to run, while the non-vectorized code we had at the end of the previous section ran on the same grid size in 0.052 seconds. For the largest grid we could try, with 511 non-boundary points in each dimension, the vectorized code took about 70 seconds to run while, as mentioned previously, the non-vectorized code ran in about 40 seconds. The slowdown is less significant for larger grids, so perhaps BLAS would eventually result in a speed increase if we were able to increase the grid size further, or if we avoided using the BLAS functions when operating on coarser grids within our multigrid algorithms. We did not test this hypothesis, partly because—as will be discussed further in later sections—we ran into difficulties combining BLAS with OpenMP.

We were not able to determine what exactly in our BLAS function calls led to the slowdown. `gprof` was not able to provide any information about the functions, because presumably Intel’s MKL was not compiled with profiling information included. Through trial and error, however, we discovered that the BLAS calls in `smooth_rbgs()` were apparently the ones causing the slowdown. Leaving in place only the calls to BLAS functions in `grid_residual()` led to a slight increase in the program’s speed. With a grid of 511 points in each dimension, the run time went from about 40 seconds in the BLAS-free code down to about 36.9 seconds. As this speedup was not very significant, we did not bother trying to replace code in any of the other less time consuming functions with BLAS calls.

The level 1 BLAS functions we used are mostly dependent on memory speed. One approach we have not tried would be to use matrix-matrix and matrix-vector operations from BLAS level 3 and 2. These BLAS routines could completely replace all of our for loops in the Gauss-Seidel smoother and residual calculation, and should be able to make very effective use of caches, possibly leading to greater performance improvements [4]. A single Gauss-Seidel iteration for improving the solution to (2) can be written using matrices as follows:

$$x_{m+1} = (D - \tilde{L})^{-1} \tilde{U} x_m + (D - \tilde{L})^{-1} b$$

where D is the diagonal of A , \tilde{L} is the strictly lower triangular part, and \tilde{U} is the strictly upper triangular part [2]. We could also calculate the residual, $b - Ax$, using one of the matrix-vector functions from level 2 BLAS. These matrices are all sparse and very large for grids of any substantial size, so it would be a good idea to make use of the sparse BLAS functions as well (these are included with the Intel MKL).

3.4 Compilers and Options

The next thing we tried to speed up our program was to simply use different compilers and command line arguments to them. So far, we had only been using `gcc 4.1.2` with the flags `-Wall -O3 -DNDEBUG`. We tried a number of different compiler flags such as `-fomit-frame-pointer`, `-fstrict-aliasing`, `-momit-leaf-frame-pointer`, `-falign-loops`, and targeting the CPUs in Ferlin more specifically with `-march=core2`, but none had much effect on run time. There was little difference between specifying `-O3` and `-O2`, but both produced much faster code than if only `-O1` or no optimization flags at all were used. Automatic vectorization of instructions using `-ftree-vectorize` did not help since `gcc` reported that it was unable to vectorize any loops. We also found that omitting `-DNDEBUG`, which disables assertions,

did not slow the program’s run time down, so we stopped using it. Keeping our assertions in place will provide extra error checking at run time.

Next we tried a more recent version of `gcc` that is also installed on Ferlin, version 4.3.2. Using the same combinations of options as with the older version, we found that most did not speed our program up much, either. However, we did find that with this version, `-O3` produced faster code than `-O2`, and using `-march=core2` sped things up additionally. The automatic vectorizer (`-ftree-vectorize`) in this version of `gcc` was able to vectorize some loops, but this still did not appreciably affect run time.

Thirdly, we tried using the Intel C compiler. Several versions of this compiler are available on Ferlin. We tried the default version (10.0.2007-06-06) and the latest installed version (11.0.2008-12-16) with different options such as `-O2`, `-O3`, and `-fast`, which enables several other options meant to speed up execution. We found that with Intel’s compilers, disabling assertions with `-DNDEBUG` did slightly speed up our program. The Intel compilers also try to automatically produce code with vectorized instructions, but this did not seem to help. No matter what options we tried with `icc`, the binaries it produced ran somewhat slower than the code produced with `gcc 4.3.2`.

Ultimately, through a combination of algorithm improvements, making calls to BLAS vector operations in `grid_residual()`, and compiling with `gcc 4.3.2` using flags `-Wall -O3 -march=core2`, the fastest serial code we produced has run times for various grid sizes shown in table 4. One strange thing to note in these results is that the code runs significantly slower on the smallest grid than it does on the next one. We are unsure as to the cause of this, but it only happened sometimes when our the program was the first thing to be run on a newly assigned Ferlin node. We suspect it is something going on with the operating system that is outside the control of our program. In any case it is not a cause for much concern as we are mostly interested in the results on larger grids.

Table 4: Run times for fastest serial version of program at various grid sizes. These results are the averages from five separate runs.

N	Wall time	Ratio to previous time	Residual norm
15	0.070	N/A	5.130×10^{-7}
31	0.009	0.1	8.165×10^{-7}
63	0.060	6.5	2.171×10^{-6}
127	0.535	8.9	6.036×10^{-6}
255	4.329	8.1	1.660×10^{-5}
511	36.382	8.4	4.490×10^{-5}

Comparing with the results in table 1, we see for the largest grid with 511 interior grid points in each dimension, our optimized code is approximately 4.8 times faster than the initial version of our program.

4 Parallelization

To further increase the performance of our program, we next moved on to parallelization. Multigrid is not as straightforward to parallelize as some algorithms because of the differing grid sizes used. One must partition the grid differently depending on its current size. However, as shown previously we have only a few functions where the majority of the running time is being spent, and it is possible to parallelize all of those. Each of the slow functions involve nested loops without data dependencies, so we decided the simplest approach would be to try shared memory parallelism with OpenMP. This allowed us to avoid much of the communication complexity that would likely be involved with using a distributed memory method like MPI.

4.1 OpenMP

The functions we decided to parallelize are `smooth_rbg()`, `grid_residual()`, `grid_restrict_fw()`, and `grid_interpolate.trilinear()`. These are the four slowest functions we had in our serial code, as shown

in table 3². All four of these functions, as mentioned above, have nested loops without data dependencies. We instructed the compiler to use OpenMP to run the outermost loops in each of these functions, the loop iterating over the grid rows, in parallel, with thread private variables used for the loop iterators as well as some temporary variables used in some of the functions. The large variables, the grid data structures, were shared amongst all threads. We parallelized the outer loops rather than either of the inner ones because there is an implicit barrier at the end of each parallel section. Barriers are expensive, so we wish to do it as few times as possible.

A problem we ran into right away was that our code no longer was producing accurate results for the smaller grids, and the inaccuracy of the results seemed to change fairly randomly. We narrowed the problem down to `grid_residual()`. On suspicion that the problem was related to the BLAS function calls, we disabled these and found that the problem went away. We assume it is not possible to inspect the Intel Math Kernel Library’s source code to investigate the problem, and were unable to find any information in [4] about using OpenMP with the provided BLAS functions. We opted to not allow both BLAS and OpenMP to be used at the same time in this function via preprocessor directives, and for the rest of our work on parallelization have ceased using BLAS altogether.

OpenMP allows one to specify a minimum size of a loop before determining whether to parallelize it or not. Sometimes if a loop has few enough iterations, the cost of parallelizing is greater than any parallel speed gains one can attain. We tested using a minimum loop size of one, two, or four rows per thread with two, four, and eight threads for various grid sizes. These result are shown in figure 1. There is not a significant difference in run time as compared to the total run time for the larger grids, no matter the number of threads or the minimum number of rows per thread. On the smaller grids, however, using a larger minimum number of rows per thread as a condition for parallelizing the loops does lead to faster execution. As is easier to see in the next section than in the plots here, we get better performance with four threads than two or eight, so we chose to use two as the minimum number of rows per thread as a condition for parallelizing.

4.2 Parallel Speedup and Efficiency

Figure 2 shows the parallel speedup and efficiency we get using one to eight threads. The absolute speedup is computed according to the time our program ran on the largest grid in table 4. This time is slightly faster than our parallel code with a single thread, used for computing relative speedup, because it speeds up `grid_residual()` with BLAS calls. The speedup is best when we use four threads, but efficiency is already quite low with this many threads. Speedup decreases as we use more threads, causing efficiency to decrease at an even greater rate. This poor scalability indicates that this program ought not be run on more than four threads, at least with the problem size used here. We suspect this poor speedup and scalability could be due to the overhead of forking off new threads being fairly substantial compared to the amount of time needed by the functions to do computations. The majority of our running time is spent inside the functions we have chosen to parallelize, but those functions are themselves called several times in the code. The run time of an individual call to even `smooth_rbgs()` ought to be quite short, so the OpenMP overhead may be significant.

5 Floating Point Performance

To determine the total number of floating point operations executed by our full multigrid code, it is easiest to start with finding the cost of small components of the code and add them up. Table 5 shows the number of floating point operations done in the various functions that make up our multigrid v-cycle (`mg_vcyc1e()`) and full multigrid (`mg_full()`) functions.

`mg_vcyc1e()` on a single grid level needs to do four smoothing operations (we do two each of pre-smoothing and post-smoothing), one residual, one restriction, one interpolation, one grid addition, and a scalar multiplication on a grid with one eighth the size of the current grid level. This last operation is

²Additional small speed-ups from using BLAS and a different compiler were achieved after the profiling data in this table was recorded, but in this faster code the top four most time consuming functions remained the same. We just did not include a table of the `gprof` results for the final version of the serial code.

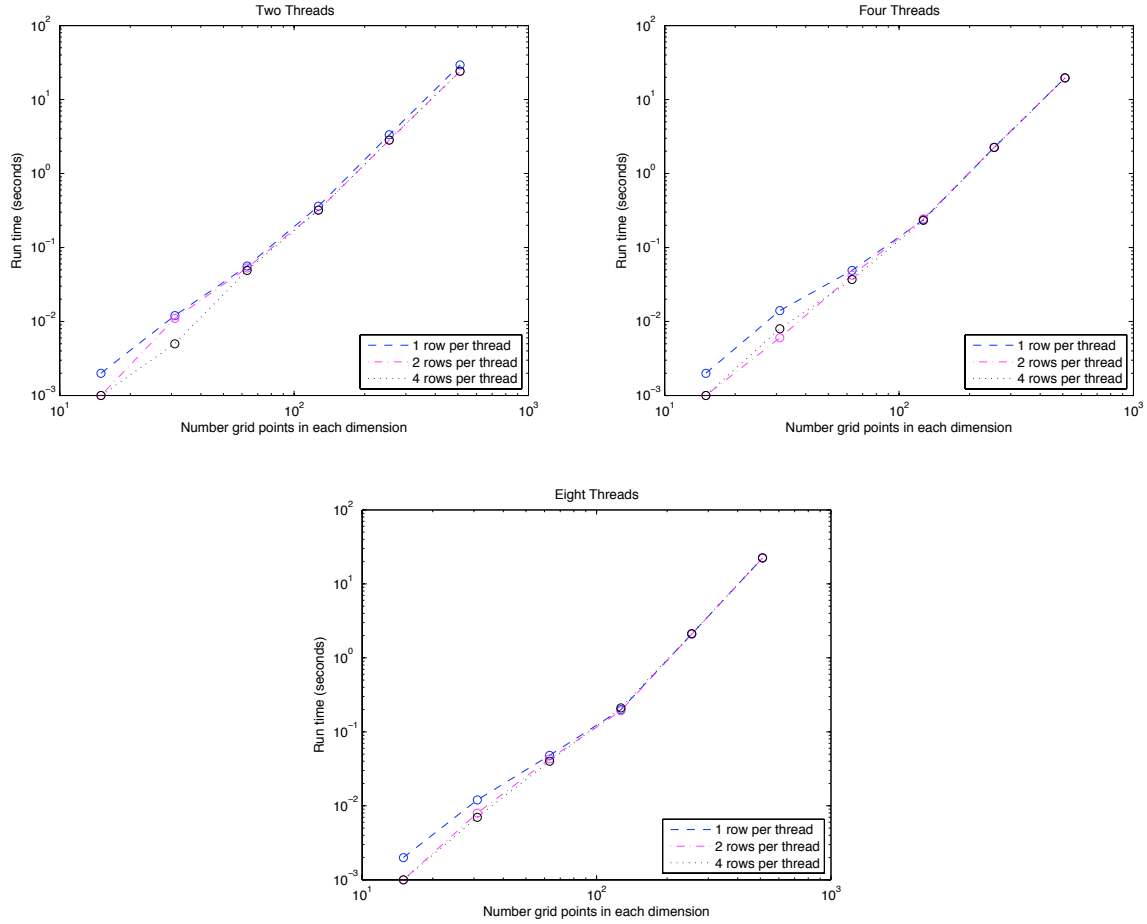


Figure 1: Run times using two, four, and eight threads with various different grid sizes. Each line in the plots is for a different minimum number of rows per thread used as a condition of whether or not to parallelize.

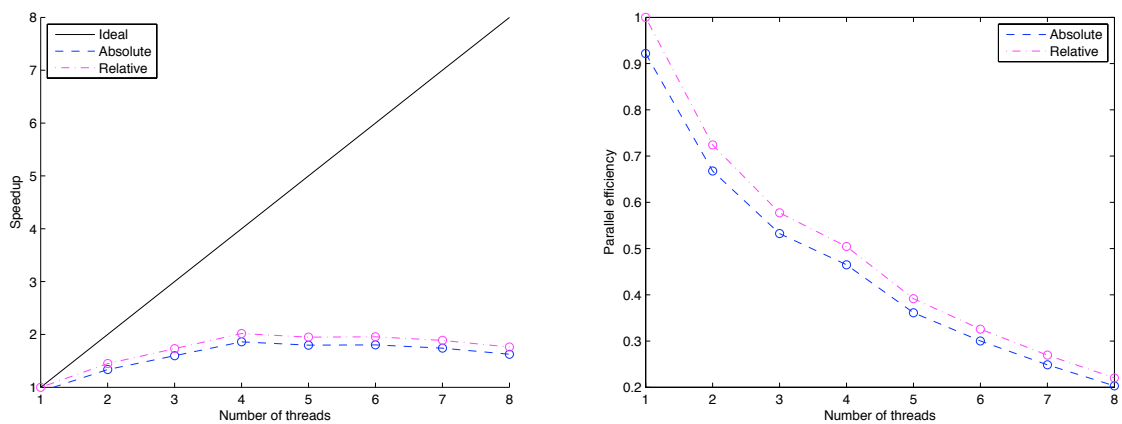


Figure 2: Relative and absolute speedup (left) and efficiency (right) of the parallel program. The grid used for these computations has 511 non-boundary points in all three dimensions, and the minimum number of rows per thread as a condition to parallelizing for loops is two.

because we must multiply the coarse grid residual, which contains about one eighth the number of points as the fine grid, by four to account for the grid spacing h doubling on the coarse grid. If the fine grid has

Table 5: Number of floating point operations for individual operations on a grid with a total of n non-boundary grid points

Function	Number of operations	Details per grid point
<code>smooth_rbggs()</code>	$7n$	6 add., 1 mult.
<code>grid_residual()</code>	$8n$	6 add., 1 sub., 1 mult.
<code>grid_restrict_fw()</code>	$3.75n$	26 add., 4 mult. on coarse grid ($n/8$)
<code>grid_interpolate_trilinear()</code>	$3.25n$	19 add., 7 mult. on coarse grid ($n/8$)
<code>grid_add_to()</code>	n	1 add.
<code>grid_multiply_scalar()</code>	n	1 mult.

a total of n non-boundary points, this totals to

$$4(7n) + 8n + 3.75n + 3.25n + n + 0.125n = 44.125n$$

floating point operations. The v-cycle algorithm is recursive though, and we must do the same operations on coarser grids. As just mentioned, each time we coarsen the grid the number of grid points goes down by roughly a factor of eight. We can then estimate the total number of operations done by `mg_vcyc1e()` with the sum $44.125n(1 + 1/8 + 1/64 + 1/512 + \dots)$. The series in parentheses depends on a finite n —we cannot coarsen the grid any further when there is only one grid point left—but we can estimate its sum with the geometric series

$$\sum_{i=0}^{\infty} \frac{1}{8^i} = \frac{8}{7}.$$

Substituting this back in gives us a total floating point operation count for `mg_vcyc1e()` of $44.125n(8/7) \approx 50.4n$.

Now we move on to our full multigrid function, `mg_full()`. The first thing done by this function is to restrict the right hand side grid down to the coarsest grid. At each level this involves one restriction and one scalar multiplication of the coarse grid, again to account for h doubling. Using the same trick as above for estimating the total size of all the grids operating on with the geometric series sum, this is $8/7$ restriction operations and $1/7$ scalar grid multiplications. The finest grid does not need to be scaled, so we're only left with the cost of scaling coarse grids. Then, from the coarsest grid back up, we must repeatedly interpolate and do v-cycles. Using our series trick again, this is $8/7$ each grid interpolations and multigrid v-cycles. Finally, we do five multigrid v-cycles at the end to further refine our solution. The total number of floating point operations needed by the full multigrid code for a grid with n total points is then

$$\frac{8}{7}(3.75n) + \frac{n}{7} + \frac{8}{7}(50.4n + 3.25n) + 5(50.4n) \approx 318n. \quad (4)$$

For our largest grid with 511 grid points in each dimension, $n = 511^3$ and the total number of floating point operations is approximately $318 \times 511^3 = 4.24 \times 10^{10}$. Dividing by our fastest serial time shown in table 4, we get $4.24 \times 10^{10} / 36.382s = 1.17\text{Gflop/s}$. A single Ferlin core is capable of about 10.6 Gflop/s theoretical peak performance³, so our serial code is getting about 11% of this.

We believe a great deal of the additional computation time is taken up waiting for memory accesses. This largest grid needs about 540MB of memory (see section below on memory usage), which is certainly too large to fit entirely in L2 cache.

³According to <http://www.pdc.kth.se/resources/computers/ferlin/ferlin-hardware>, the entire cluster is capable of 57.88TF theoretical peak performance, and it consists of 5440 cores

6 Further Discussion

6.1 Cache Effects

One interesting thing of note in the run times for our problem given in tables 1 and 4 is that the ratio of the run time to the previous grid's run time jumps when we go from grids of size $N = 63$ to 127 ⁴. The time jump from $N = 127$ to the next grid is close to a factor of eight, which satisfies the expected linear run time of our algorithm. For a grid with 63 non-boundary points, we have $65^3 \approx 275000$ total floating point numbers (it is 65^3 rather than 63^3 because we must include two boundary grid points at each edge). Each of these floats takes up four bytes of memory, for a total of about 1.1 million bytes of memory to store the grid. For the next larger grid, we have $4(129^3) \approx 8.5$ million bytes of memory. Each Intel Xeon E5430 CPU in a Ferlin node has 12MB of L2 cache⁵, so we can fit both the solution and right hand side grids of our linear system completely in that L2 cache on the grid of size 63. This is not the case with the next larger grid, so we believe the additional memory accesses necessary for grids of size 127 and greater account for the ratio in run times for $N = 127$ to 63 being somewhat larger than the 8 expected.

6.2 Memory Usage

Observing our program working on the largest grid size used, 511 points in each dimension, we witnessed the system using up to about 1.9GB of memory. This measurement is not at all precise—we simply ran the program and checked the resident size repeatedly with the command `ps ux`. The largest grid used, including its boundary points, contains 513^3 floating point numbers. With four bytes for each float, this adds up to about 540MB. We may keep up to three of the largest grids in memory at once: the solution, right hand side, and one grid for the fine grid residual that is reused for the error. This adds up to about 1.6GB of memory. The coarse grid right hand sides stored by `mg_full()` and other temporary coarse grids add a few more megabytes, so our observed memory usage is not that much higher than expected. We are unable to solve on the next larger grid, with 1023 points in each dimension, because the eightfold increase in memory that would be required is more than the total physical memory (8GB) in a Ferlin node. To confirm this, we attempted to run our program with this next larger grid, and it exited early due to an inability to allocate enough memory.

We have also chosen to use single precision floating point numbers in this problem, mostly because we would need twice as much memory using double precision. We have written the code such that one can switch to double precision by defining the `USE_DOUBLE` macro at compile time, however. A double precision version of the code produces more accurate results according to a smaller residual at the end, but the run time is substantially longer. We suspect this is due to being able to store only half as many grid points in the cache, and the memory bandwidth being saturated with fewer grid points as well.

6.3 Non-Optimal Grid Sizes

Throughout this report, we have used only a few different grid sizes. Our multigrid algorithm works best on grids where the number of non-boundary points in each dimension is of the form $2^n - 1$. Grids of this size can be repeatedly divided into coarser grids by taking the size $N = 2^n - 1$ in each dimension, subtract 1, divide by 2, then add 1 to get the size of the coarse grid. This coarsening can be repeated all the way down to the coarsest possible grid, with just one non-boundary point. We must completely solve the linear system on the coarsest grid with an iterative solver (Gauss-Seidel in our case), so it is advantageous to make this coarsest grid as small as possible. Our program can handle grids of other sizes, but in general it must make many more Gauss-Seidel iterations at the coarsest grid than an optimally sized grid. The time to do these computations can quickly add up. For example, if the coarsest grid we can restrict to has 10 grid points in each dimension, this amounts to a linear system with $10^3 = 1000$ unknowns, and Gauss-Seidel can then take 1000 iterations to converge to a solution.

⁴We suspect the unexpected time ratios on smaller grids are due to factors such as the time intervals being so short and other overhead in the program having a greater impact on run time than the computation time needed by the multigrid algorithm itself.

⁵<http://processorfinder.intel.com/details.aspx?sSpec=SLANU>

We can improve this situation by using a faster solver for the coarsest grid. Since the matrix A in (2) is symmetric and positive definite, the coarse grid matrix will be as well, and we can use the conjugate gradient method. Also, we could improve our multigrid algorithms so that other coarsening strategies are possible besides just grid size doubling.

6.4 Concluding Remarks

Although our parallel efficiency and speedup are not particularly impressive, we feel satisfied with the serial performance of our multigrid program. If more time were available, we would like to investigate the decrease in speedup observed when we go over four threads. It could also be useful to do further experimentation with OpenMP for each function we have used it in. Perhaps it would pay off to parallelize more parts of the program, or perhaps instead some of the places we have parallel loops are not contributing to speedup and should be made serial again. Additionally, we are curious as to why the use of BLAS functions in our Gauss-Seidel smoother caused such a slowdown, and would like to determine if it possible to combine the use of BLAS library functions with OpenMP.

In the future, it would also be interesting to try a distributed memory parallel version of this program. This would allow one to spread the memory requirements for even larger grids across several nodes, rather than requiring a single system to have enough memory to store three grids of the desired size. It seems rather challenging to program a distributed memory parallel version of the multigrid algorithm, though, due to the need to divide up the grid amongst compute nodes differently at different grid coarseness levels. However, these sorts of parallel multigrid programs are possible and discussed in [7] and elsewhere.

References

- [1] William L. Briggs. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics, 1987. ISBN 0898712211.
- [2] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997. ISBN 0898713897.
- [3] James W. Demmel. Matlab programs for applied numerical linear algebra. <http://www.eecs.berkeley.edu/~demmel/ma221/Matlab/>, 2004.
- [4] Intel Corporation. *Intel Math Kernel Library Reference Manual*, 8 2008. Document Number 630813-029US.
- [5] Harald Köstler. *A Multigrid Framework for Variational Approaches in Medical Image Processing and Computer Vision*. PhD thesis, Universität Erlangen-Nürnberg, 2008.
- [6] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill Higher Education, 2004. ISBN 0072822562.
- [7] Ulrich Trottenberg, Cornelis Oosteele, and Anton Schüller. *Multigrid*. Academic Press, 2001. ISBN 0-12-701070-X.