Mathematics Research Report No. MRR 038–97

TECHNIQUES FOR IMPROVING THE DATA LOCALITY OF ITERATIVE METHODS

LINDA STALS, ULRICH RÜDE

October 1997

ABSTRACT. The numerical solution of partial differential equations leads to large, sparse systems of equations with up a several millions of unknowns. Fast iterative algorithms for the solution of these systems are typically based on the multilevel principle. Unfortunately, some of the commonly used programming techniques lead to a high overhead on many advanced computer architectures. A fundamental problem arises from hierarchical memory architectures with several layers of caches. Their effective use requires programs with data access locality. Unfortunately, iterative solvers are typically implemented by using global sweeps over the whole data set, and thus their performance is essentially limited by the speed of the memory system. This article introduces techniques to improve the data locality and therefore the efficiency of multigrid algorithms.

1991 *Mathematics Subject Classification*: primary 65M55; secondary 68-04.

Postal address: Institut für Mathematik, Universität Augsburg, Germany

Techniques for Improving the Data Locality of Iterative Methods

Linda Stals[*] Ulrich Rüde[*]

October 13, 1997

1 Introduction

Current semiconductor technology is (1997) capable of producing microprocessors operating with more than 500 MHz clock frequency and a peak performance of beyond 1 GFlop (10^9 floating point operations) per second. The continuing evolution is expected to yield further performance increases of estimated 60% annually (see e.g. Patterson [16]).

The main bottleneck in todays computers is memory access times. Memory access times are decreasing at a comparatively low rate of about 5-10% per year [5]. Consequently the CPUs are unable to get the data quick enough to keep busy and only a small percentage of the performance potential can be realised.

One way of hiding such a bottleneck is to use a *memory hierarchy*. There are many different ways of implementing this idea, but the consistent theme is that there is a large cheap slow memory at the bottom of the hierarchy and a small expensive high speed memory at the top of the hierarchy. This high speed memory is called the *cache* and will contain copies of the main memory to speed up access to frequently needed data.

An access to data not residing in the cache has to pay the extra cost of copying the information from the main memory to the cache. Caches thus help to speed up programs only when cached data is being re-used. For programs that do not re-use data, the speed of processing is limited by the poor latency and bandwidth values of the main memory. For many compute intensive applications it has therefore become important to write cache-aware programs.

The situation can be made even worse, since caches usually operate not on single words of memory, but on blocks of a fixed size, the so-called *cache lines*. A typical cache line contains four double precision floating point numbers. If an array is now accessed with stride 4, only one of the words in a cache line is used, but all four must be copied from the main memory to the cache.

Additionally, the different cache organisation principles may have an impact on the performance. In direct mapped or set-associative caches, the main memory blocks of a certain class compete for a limited number of cache lines. If data is accessed frequently from more of these blocks than there are corresponding cache lines, the effectivity of caching is again reduced.

[*]Institut für Mathematik, Universität Augsburg, D-86159 Augsburg, Germany

There is a trend to use an even more refined memory hierarchy. Besides a cache memory integrated into the CPU chip, which is fast, but naturally of limited size, many current designs employ a second level of external cache. This cache is built with fast and expensive synchronous memory (SRAM) chips. The latency and bandwidth of the external cahce is significantly better than that for the main memory which is built from cheap dynamic memory (DRAM) chips of various types. However, the second level external cache is usually already significantly slower than the on-chip cache.

A step further is taken by the curent Digital Alpha 21164 processors which employ two levels of on-Chip cache of size 8KB and 96KB, respectively. Additionally, a third level of SRAM based cache off chip can be used. The motivation behind this design are the high clock rates of the Digital chip, it would have been impossible to provide latencies of one cylce even on the chip when the cache were much larger than 8KB. Note that the successor of the 21164 chip will return to only two layers of cache, however, this comes at the price of even the on-chip cache having a two-cycle latency.

Effects from hierarchical memory are not new. For example, the software design for the dense linear algebra package LAPACK [1] is influenced by efficiency considerations in the presence of caches. On the other hand, little work has yet been done with iterative methods, particularly in our area of interest which is the multigrid method [3, 12] for the solution of partial differential equations (PDE).

Multigrid is one of the most attractive algorithms for the solution of large sparse systems of equations that arise in the solution of PDEs. The efficiency of multigrid algorithms has been shown theoretically and in numerous practical applications. In terms of the number of arithmetic operations, multigrid methods are *asymptotically optimal*, that is the number of operations is only proportional to the number of unknowns, see e.g. [12]. In standard problems, like scalar elliptic PDE in two or three space dimensions, multigrid can compute a solution within 100 floating point operations per unknown, and in special situations like Poisson's equation even in below 30 operations per unknown. On a GFlops machine, we would thus expect to solve a system of one million unknowns in a tenth of a second or below. Unfortunately this figure is much too optimistic, since all multigrid codes known to the authors fail to reach the peak performance on cache based machines by a large margin.

In this paper we shall therefore show how non-standard cache-aware implementations of the multigrid method improve the out-of-cache performance significantly. After a brief introduction to multigrid methods, we will discuss various cache-oriented programming techniques and their effect on multigrid performance.

2 The Multigrid Method for the Solution of Elliptic Partial Differential Equations

2.1 Problem Description

In this article, we consider the finite difference approximation of a model elliptic partial differential equations defined on the square domain $\mathcal{M} = [0,1] \times [0,1]$ with Dirichlet boundary conditions. The finite difference method uses a discrete grid $\mathcal{M}_k$ given by the set of points

$$\mathcal{M}_k = \{(x_i, y_j) = (i/h_k, j/h_k) : 0 \le i, j \le 1/h_k, i, j \in \mathbb{N}\},$$

with meshsize $h_k = 2^{-k}$. The multigrid method will use a sequence of grids $\mathcal{M}_k$ for $k = 1, \ldots, n$.

We assume that the finite difference stencil is constant on each grid level and may be stored in compact 9-point form,

$$\begin{bmatrix} s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 \\ s_7 & s_8 & s_9 \end{bmatrix}.$$

This restricts the discussion in the present paper to linear PDE with constant coefficients, though of course many of the basic ideas can be carried over to more general situations e.g. involving variable coefficients.

If $s_5 = -4$, $s_2 = s_4 = s_6 = s_8 = 1$ and $s_1 = s_3 = s_7 = s_9 = 0$ the 9-point stencil is equivalent to the standard 5-point finite difference discretisation for the Laplacian operator. To simplify the algorithm descriptions we let `Su[i, j]` denote the result of applying the 9-point stencil to some grid point `u[i, j]` $= u(x_i, y_j)$

$$\texttt{Su[i, j]} = \begin{array}{lllllll} s_1\texttt{u[i+1, j-1]} & + & s_2\texttt{u[i+1, j]} & + & s_3\texttt{u[i+1, j+1]} & + \\ s_4\texttt{u[i, j-1]} & + & s_5\texttt{u[i, j]} & + & s_6\texttt{u[i, j+1]} & + \\ s_7\texttt{u[i-1, j-1]} & + & s_8\texttt{u[i-1, j]} & + & s_9\texttt{u[i-1, j-1]}. \end{array}$$

For the description of the Gauss Seidel algorithm it is also convenient to introduce

$$\texttt{S}_{\texttt{-5}}\texttt{u[i, j]} := \texttt{Su[i, j]} - s_5\texttt{u[i, j]}.$$

In most of this paper we will use the compact 9-point *Mehrstellen*-stencil of the form

$$\frac{1}{6}\begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}.$$

In our context the Mehrstellen operator has two interesting features. If additionally to using the *Mehrstellen*--operator the right hand side is preprocessed by applying the averaging operator

$$\bar{R}_k = \frac{1}{6}\begin{bmatrix} 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 4 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 \end{bmatrix},$$

then an approximation accuracy of $O(h^4)$, where h is the meshsize of the grid, is obtained provided the solution is smooth enough, see e.g. Hackbusch [12]. Clearly, the 9-point discretisation is more expensive in terms of operations required. For smooth enough solutions, however, its better convergence order leads to more efficient algorithms. Additionally, our results indicate that the compact 9-point stencil results in a higher Mflop rate than the 5-point stencil (see section 7.2). This effect partly compensates the higher work count, making the method additionally attractive in our context.

Thus this paper concentrates on 9-point stencils, except in section 7.2 where we present a comparison between the compact 9-point stencil and the 5-point stencil to substantiate the above claim. To guarantee that this comparison is as fair as possible, we use a 5-point stencil to store the 5-point finite difference grid. In other words, we do not explicitly multiply `u[i+1, j-1]`, `u[i+1, j+1]`, `u[i-1, j-1]` and `u[i-1, j+1]` by s_1, s_3, s_7 and s_9 respectively as $s_1 = s_3 = s_7 = s_9 = 0$.

2.2 Multigrid Method

The multigrid method is an iterative method for the solution of (discretised) partial differential equations. Multigrid methods use multiple layers of grids with each grid layer removing different frequency components of the error.

We have implemented the so-called standard *V-cycle* and the *full multigrid* (FMV) scheme. For detailed information about these algorithms, in particular their mathematical properties, we refer to the standard literature [2, 3, 4, 12, 13, 15]. Here we just describe the basic algorithms so that we have a basis to discuss our programming methodology in section 3.

Consider the problem of solving the system of equations $Au = f$, where A is the result of applying some discretisation method to an elliptic partial differential equation. Then, given a nested sequence of grids, $\mathcal{M}_1 \subset \mathcal{M}_2 \subset ... \subset \mathcal{M}_n$, the algorithm for the V-cycle is given in figure 1 while the algorithm for the FMV scheme is given in figure 2.

$\texttt{V\_cycle}_k(\mu_1, \mu_2, u_k, f_k)$

1. Iterate μ_1 times over $A_k u_k = f_k$.
2. If k = 0 then goto step 8.
3. Calculate the residual $r_k := f_k - A_k u_k$.
4. Restrict the residual $r_{k-1} := R_k^{k-1} r_k$.
5. Solve the coarse grid equation $u_{k-1} := \texttt{V\_cycle}_{k-1}(\mu_1, \mu_2, 0, r_{k-1})$
6. Interpolate the coarse grid correction $v_k := I_{k-1}^k u_{k-1}$.
7. Correct the current estimate $u_k := u_k + v_k$.
8. Iterate μ_2 times over $A_k u_k = f_k$.

Figure 1: The V-cycle algorithm.

$\texttt{FMV\_scheme}_k(\mu_1, \mu_2, u_k, f_k)$

1. If k = 0 then goto step 4.
2. Use the coarse grid to find a good initial guess $u_{k-1} := \texttt{FMV\_scheme}_{k-1}(\mu_1, \mu_2, 0, f_{k-1})$
3. Interpolate the coarse grid initial guess $u_k := \widetilde{I}_{k-1}^k u_{k-1}$.
4. Solve the fine grid equation $u_k := \texttt{V\_cycle}_k(\mu_1, \mu_2, u_k, f_k)$

Figure 2: The FMV-scheme algorithm.

In the algorithm, we use bilinear interpolation

$$I_{k-1}^{k} = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix},$$

and set the restriction operator to be the transpose of the interpolation operator $R_k^{k-1} = (I_{k-1}^k)^T$.

The interpolation operator $\widetilde{I}_{k-1}^k$ used in the FMV-scheme has to be of high order to make sure that we do not introduce high frequency errors to the fine grid, see [2, 3]. Therefore we use fourth order polynomial interpolation to ensure that we do not loose the $O(h^4)$ convergence offered by the compact 9-point stencil.

3 Programming Environment and Data Structures

3.1 Language

The programs were written in the literate programming language called FWEB [14]. The aim of such programming languages is to treat the documentation and code as a whole, thus blurring the distinction between the two. An FWEB program consists of a collection of sections, with each section containing a documentation and a code part. The code part can be written in either C^{++}, C, Fortran, Pascal or Ratfor. The documentation part can be written in either LaTeX or TeX.

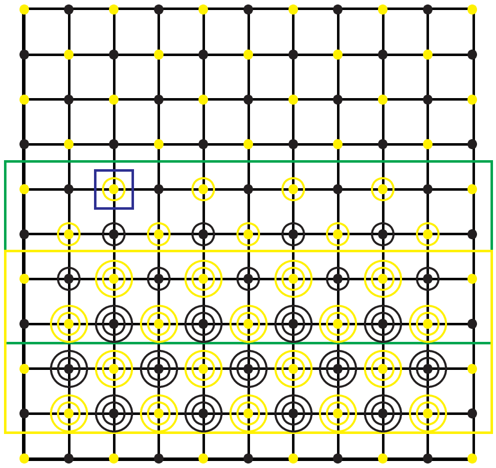
FWEB programs are passed through the FWEAVE or FTANGLE preprocessors. The FWEAVE preprocessor outputs a LaTeX or TeX document with the code pretty-printed. For example, figure 3 shows the code which updates two pairs or rows within the red/black Gauss-Seidel algorithm (section 4.2 explains how we group the rows into pairs to improve the out-of-cache performance). The FTANGLE preprocessor extracts the code part from the FWEB document and outputs compilable code. We have written all of the code in C.

Our original reason for choosing FWEB was to make sure that the program was well documented. By the nature of the project we must try many different approaches and it is important to keep track of the results (successes as well as failures). We also found that FWEB's extended macro features were of great benefit. For example, in the code shown in figure 4 we have avoided the need to unroll the loop by hand by using the `DO` macro defined within FWEB; the FWEB preprocessor FTANGLE will expand the code defined within each loop.

Another advantage of using FWEB is that you can segment the code without using procedures. Compilers are not very good at inter-procedural code optimisations, so we would like to avoid their use for small sections of code. Unfortunately though FWEB does not understand local and global scope so when putting the different sections together you have to be careful that the variables have been initialised in the correct order, etc. The need to have a clear understanding of how all of the sections should be put together detracts some of the benefits of segmenting the code.

§7.4 [#125] `Multigrid algorithm` Calculation with fused red/black loops 20

[`sor.web`] To ensure that the data dependences are not violated we must update the rows in pairs. For example, in the following figure we could not update the indicated red point before we updated the black point above it.



⟨ do two rows of RB SOR 7.4 ⟩ ≡
{ // update the black points

start_column = 1;
⟨ do one row of paired RB SOR 7.5 ⟩

bfloor = *bottom*;
bottom = *centre*;
centre = *top*;
top += *fine*.*no_columns*;

// update the red points

start_column = 2;
row++;
⟨ do one row of paired RB SOR 7.5 ⟩

top −= 3∗*fine*.*no_columns*;
centre −= 3∗*fine*.*no_columns*;
bottom −= 3∗*fine*.*no_columns*;
bfloor −= 3∗*fine*.*no_columns*;
}

This code is used in sections 7.1, 7.2, 7.3, #166, #167, #168, #172, #173, #174, #177, #178, #183, #184, #185, and #186.

Figure 3: A page extracted from an FWEB document.

§7.7 [#128] **Multigrid algorithm** Calculation with fused red/black loops 22

[`sor.web`] Unroll the loop over the black points. The black points should sit below the red points defined in *UNROLL_RED*.

```
"multi.c" 7.7 ≡
  @m UNROLL_BLACK(unroll_size)
      {
      $DO(I, 0, unroll_size)
        {
        tmp1[I] = fine_stencil.a*centre[j - 1 + 2*I] + fine_stencil.c*centre[j + 1 + 2*I]
              + fine_stencil.g*bfloor[j - 1 + 2*I] + fine_stencil.i*bfloor[j + 1 + 2*I];
        tmp2[I] = fine_stencil.d*bottom[j - 1 + 2*I] + fine_stencil.f*bottom[j + 1 + 2*I]
              + fine_stencil.b*FINE(row, j + 2*I) + fine_stencil.h*bfloor[j + 2*I];
        }
      $DO(I, 0, unroll_size)
        {
        tmp3[I] = FINE_F(row - 1, j + 2*I) - tmp1[I] - tmp2[I];
        }
      $DO(I, 0, unroll_size)
        {
        FINE(row - 1, j + 2*I) = tmp3[I]/fine_stencil.e;
        }
      }
```

Figure 4: Example use of macros within a FWEB document. The preprocessor FTANGLE will expand the code within each loop so the programmer does not have to do it by hand.

3.2 Architecture

The test examples given here were run on the SGI Power Indigo with R8000 processor. The specifics of the machine are

SGI.

Clock Cycle: 75MHz.

Peak Performance: 300 Mflops.

Cache: First-Level Data Cache: 16 Kbytes, Secondary Cache: 2 Mbytes.

Processor: R8000.

Compiler: CC

3.3 Compiler Options

On the **SGI** machines the different compiler options which were tested include

`-O1`, `-O2` and `-O3`.

`-Ofast1`: `-Ofast=ip26`.

`-Ofast2`: `-Ofast=ip26 -OPT:alias=restrict`.

The `-O1`, `-O2` and `-O3` compiler options are optimisation levels 1, 2 and 3. Level 1 performs local optimisations, level 2 performs global optimisations which do not change the floating point accuracy while level 3 allows more aggressive optimisations (i.e. the floating point accuracy may be changed). The `-Ofast=ip26` option sets the fastest compiler options for the target platform ip26. Finally, the `-OPT:alias=restrict` option tells the compiler that different pointers will always point to different memory locations and thus means that it can make less conservative optimisations.

Figure 5 shows the Mflops for the standard implementation (no loop unrolling etc) of the lexicographical Gauss Seidel method using the different compiler settings for the **SGI** machines. The documentation for the **SGI** machines says that the level 3 optimisation may hurt the performance, which is true in this case.

As the results show, it is very important to choose the right compiler settings. This observation is not new, the point which surprised us though is the extent to which the results depend upon the compiler options. For example, in figure 5 the cache effects are not even visible until the higher compiler options are used.

If is often stated by the machine vendors that in order to get the best performance out of the machine you must use the right compiler options. Unfortunately however they do not seem to be so forthcoming in saying what the right compiler options are. Using the options given in the LINPACK [6] benchmarks seems to be a good start, but these are for dense matrix solvers and are not necessarily the best choice for iterative solvers. Wading through the documentation trying to find which switches to turn on or off is a considerable source of frustration.

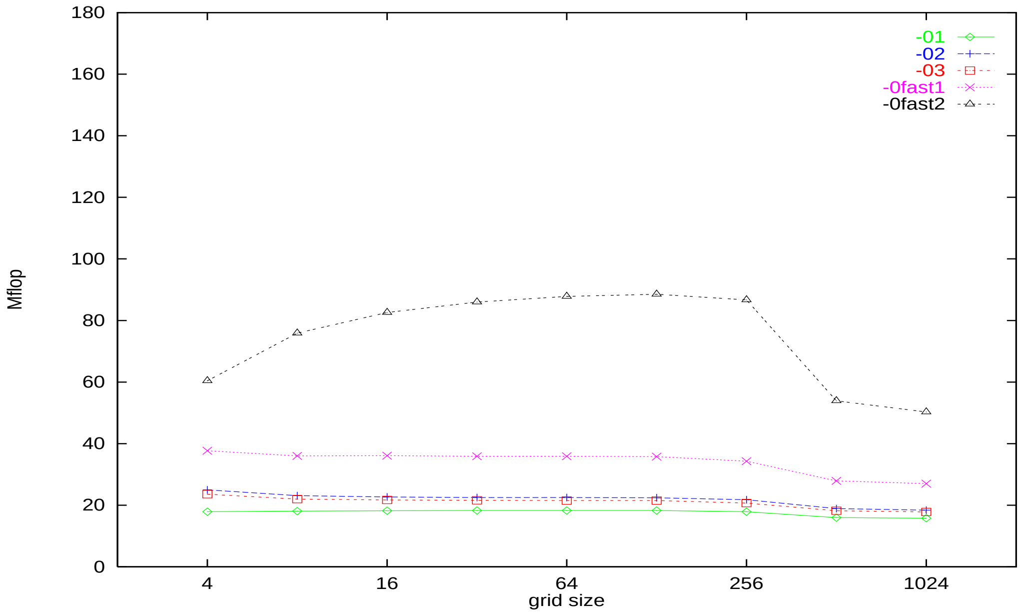


Figure 5: The Mflops rate for the standard Gauss Seidel method with different compiler settings on the **SGI**. The grid size = $1/h_n$.

3.4 Data Structure

The data layout can have a direct effect on the Mflops rate, therefore we feel that it is important to give an overview of our data structure.

The multigrid levels are stored in an array of grids. The array is of a fixed size given by a `#define` variable which is set at compile time. The grids are stored in structures with the following format

```
typedef struct
    {
    Counter number_nodes;
    Counter number_columns;
    Counter total_number_nodes;
    Real *u;
    Real *f;
    } Grid;
```

The variables `u` and `f` are allocated space of size (`number_columns` $\times$ `number_columns`) at run time. We use this dynamic structure because one of our future goals is to look at adaptive refinement. Although this data structure is giving us good results, more testing and experimentation still needs to be done to ensure that it is the optimal data structure.

When accessing a two dimensional grid the usual practise is to use something like

`grid.u[i*number_columns+j]`.

We have observed that the performance is slightly better if the grid is accessed by the rows. For example, let

$$\texttt{top = \&grid.u[(i+1)*number\_columns]},$$

$$\texttt{centre = \&grid.u[i*number\_columns]}$$

and

$$\texttt{bottom = \&grid.u[(i-1)*number\_columns]},$$

then we can use statements like

$$\begin{aligned}\texttt{sum} &= s_1\texttt{top}[\texttt{j}-1] + s_2\texttt{top}[\texttt{j}] + s_3\texttt{top}[\texttt{j}+1] \\ &+ s_4\texttt{centre}[\texttt{j}-1] + s_5\texttt{centre}[\texttt{j}] + s_6\texttt{centre}[\texttt{j}+1] \\ &+ s_7\texttt{bottom}[\texttt{j}-1] + s_8\texttt{bottom}[\texttt{j}] + s_9\texttt{bottom}[\texttt{j}+1].\end{aligned}$$

The 9-point stencils are also stored in an array of fixed size (the same size as the multigrid array). We have allowed the stencil to vary from grid to grid (but not within a grid) because we are also looking at a hierarchical multigrid method [10]. Our studies of this method are still preliminary and are not described in this report.

The stencils are stored in a structure which holds nine double precision variables. We had originally stored the stencils within the grid structure but found that that decreased the Mflops rate.

When carrying out the runs with the 5-point finite difference stencils described in section 7.2 we do not view the 5-point stencil as something which can sit within a 9-point stencil, but rather use a structure which only holds 5 double precision variables.

4 Red/Black Gauss Seidel

4.1 In-Cache Optimisations

To smooth the error on a grid level we used the red/black Gauss Seidel algorithm. As the smoother is one of the most computationally expensive parts of the multigrid algorithm it shall be described in some detail.

Figure 6 shows the Mflops rate for different implementations of the red/black algorithm. The line labelled `red black(1)` is the result of what we would consider to be a standard implementation of the red/black Gauss Seidel method. In other words we used some code like that given in algorithm 1.

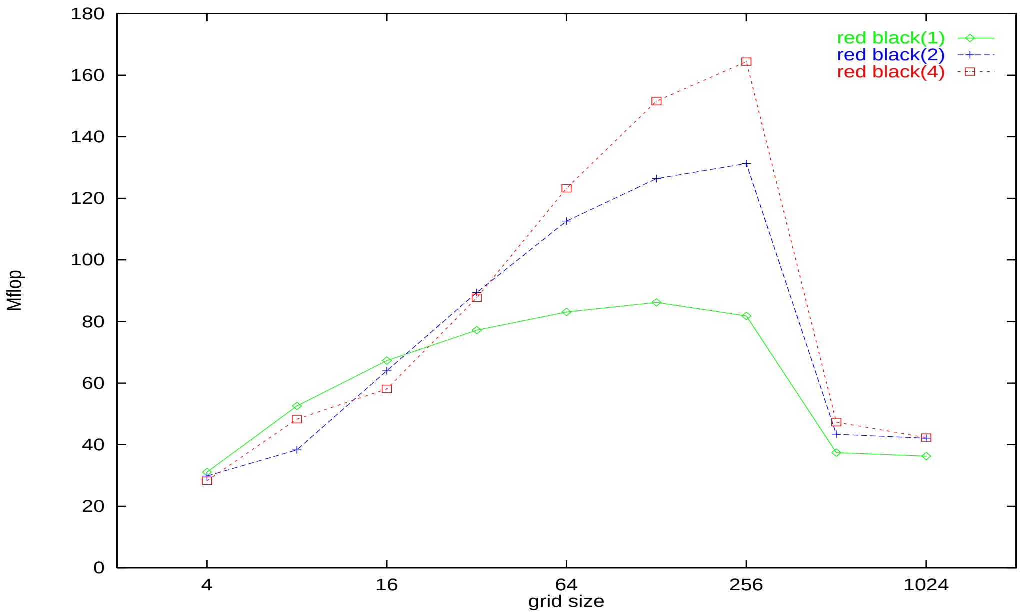


Figure 6: Mflops rate for different implementations of red/black Gauss Seidel .

Algorithm 1
Standard Red/Black Gauss Seidel

```
{
 update the red points
 for (i = 1; i < grid size; i++)
     for (j = 2 - i%2; j < grid size; j++)
        u[i, j]= (f[i, j] − S_5u[i, j])/s_5
 update the black points
 for (i = 1; i < grid size; i++)
     for (j = 1 + i%2; j < grid size; j++)
        u[i, j]= (f[i, j] − S_5u[i, j])/s_5
}
```

The lines labelled `red black(2)` and `red black(4)` were obtained by using an optimisation technique similar to loop unrolling. This is like a technique suggested by Fried [8] and is also used in [9] and is clearly effective as it doubles the maximum Mflops rate. A pseudo-code description of the technique for two levels of loop unrolling is shown in algorithm 2, although in FWEB we used the `DO` loop in a similar way to that shown in figure 4 to handle loop unrolling of any given level (as long as it is an even number).

Algorithm 2
Optimised Red/Black Gauss Seidel

```
{
  update the red points
 for (i = 1; i < grid size; i++)
     start column = 2 - i%2
     Update One Row of Gauss Seidel (Algorithm 3)
 update the black points
 for (i = 1; i < grid size; i++)
     start column = 1 + i%2
     Update One Row of Gauss Seidel (Algorithm 3)
 }
```

Algorithm 3
Update One Row of Gauss Seidel

```
 {
  for (j = start column; j < grid size - 2; j+= 4)
     tmp1[0] = s1 u[i+1, j-1] + s3 u[i+1, j+1] + s7 u[i-1, j-1] + s9 u[i-1, j+1]
     tmp2[0] = s4 u[i, j-1] + s6 u[i, j+1] + s2 u[i+1, j] + s8 u[i-1, j]
     tmp1[1] = s1 u[i+1, j+1] + s3 u[i+1, j+3] + s7 u[i-1, j+1] + s9 u[i-1, j+3]
     tmp2[1] = s4 u[i, j+1] + s6 u[i, j+3] + s2 u[i+1, j+2] + s8 u[i-1, j+2]
     tmp3[0] = f[i, j]- tmp1[0] - tmp2[0]
     tmp3[1] = f[i, j+2] - tmp1[1] - tmp2[1]
     u[i, j]= tmp3[0]/s5
     u[i, j+2] = tmp3[1]/s5
 if (start column%2 == 0)
     j = grid size - 1
     tmp1[0] = s1 u[i+1, j-1] + s3 u[i+1, j+1] + s7 u[i-1, j-1] + s9 u[i-1, j+1]
     tmp2[0] = s4 u[i, j-1] + s6 u[i, j+1] + s2 u[i+1, j] + s8 u[i-1, j]
     tmp3[0] = f[i, j]- tmp1[0] - tmp2[0]
     u[i, j]= tmp3[0]/s5
 }
```

Figure 6 unmistakably shows the cache effect, once the grid is too large to fit in the cache ($> 256 \times 256$) the performance decreases dramatically. In the next subsection we shall show ways of modifying the algorithm to improve the out-of-cache performance. It may seem unnecessary to spend time on improving the in-cache performance when we are mainly interested in the results for large grids, however all methods of improving the out-of-cache performance involve some sort of blocking technique. That is, we try to break the large grids up into smaller grids so if the performance for the small grids is poor the performance for the large grids will be poor.

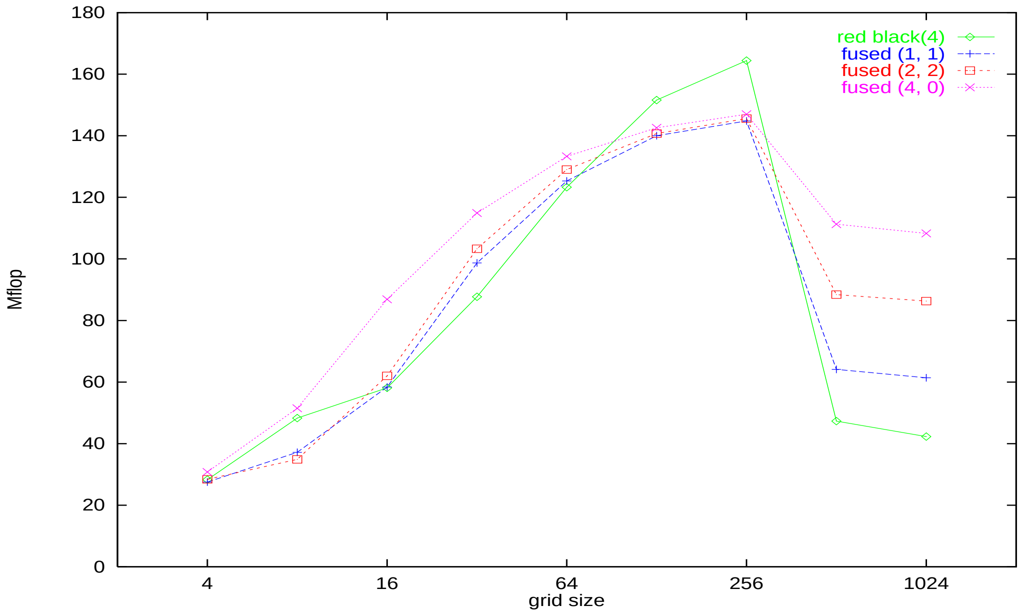


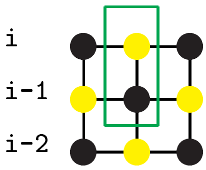
Figure 7: Mflops rate for different implementation of red/black Gauss Seidel .

4.2 Out-of-Cache Optimisations

In the last section we saw how the cache effect can severely degrade the performance of the red/black Gauss Seidel algorithm. Figure 7 shows the results of melting the loops together to improve the out-of-cache performance. This technique is explained in the following text.

To get good out-of-cache performance we found that we needed to use 2 levels of loop unrolling (instead of 4), that is why the best in-cache results shown in figure 7 are not as good as those in figure 6 (however the overall results are far better). We would like to point out that the results show that in order to have effecient programs it is indeed necessary to write cache aware code.

When implementing the standard red/black Gauss Seidel algorithms (algorithms 1 and 2) the usual practise is to do one complete sweep of the grid updating all of the red nodes and then one complete sweep updating all of the black nodes. The first point to note is that as a red node is updated the black node directly underneath it may be updated. Figure 8 shows how we can group the red and black nodes together. If a 9-point stencil is placed over one of the black points then we can see that all of the red points it touches are up to date (as long as the red node above it is up to date).



Consequently, we work in pairs of rows; once all of the red nodes in one row have been updated, all of the black nodes in the previous row are updated. So instead of doing a red sweep and then a black sweep we just do one sweep of the grid updating the red and black

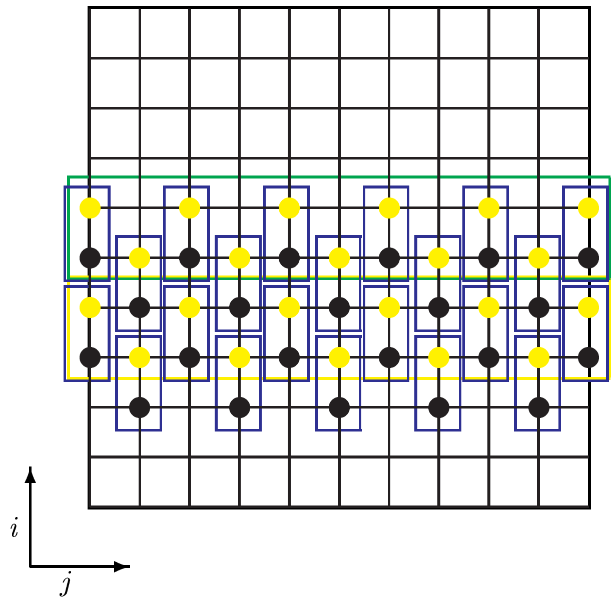


Figure 8: The red and black nodes can be updated in pairs.

nodes as we move through. The line labelled `fused(1,1)` in figure 7 shows how this can improve the Mflops rate.

In the next step we melt the sweeps together. For example, instead of doing two sweeps, and updating the nodes once during each sweep, we do one sweep and update the nodes twice. Let $\mu = \mu_1 \vee \mu_2$ be the number of sweeps, then a high level view of the algorithm is given in algorithm 4. We have ignored the top and bottom boundaries which have to be handled separately to avoid the problem of accessing data which is sitting outside of the grid, but the idea behind code used for the top and bottom boundaries is the same as that given in algorithm 4.

Algorithm 4
Melt Red/Black Gauss Seidel

```
{
 for (k = 2*(μ + 1); k < grid size - 1; k+= 2)
     for (i = k; i > k - 2μ; i-= 2)
        start row = i-1
Do Two Rows of Red/Black Gauss Seidel          (Algorithm 5)
}
```

.

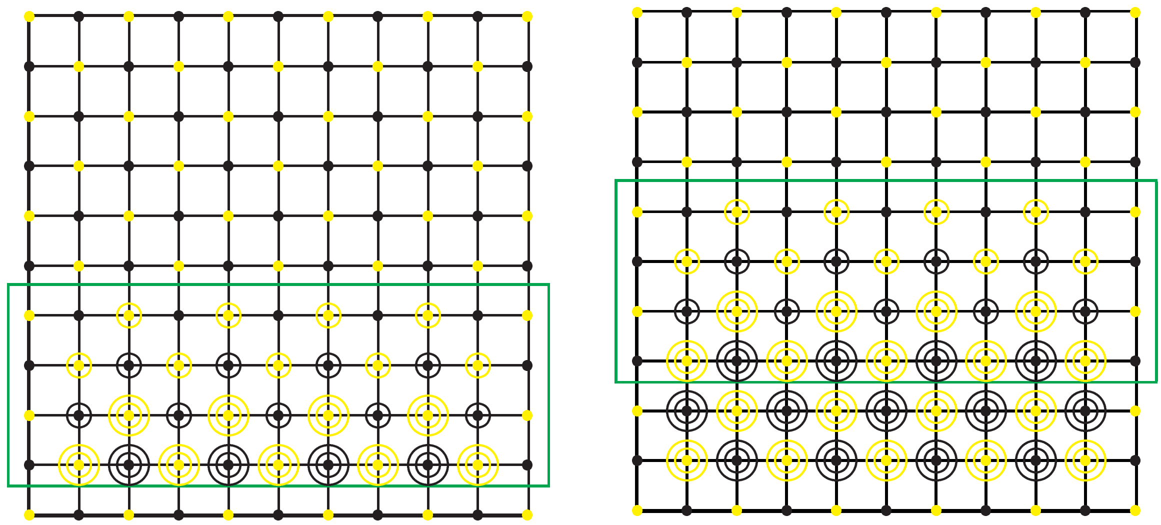


Figure 9: Example of fusing two sweeps together. Each circle around a node shows how many times it has been updated. Within each strip we updated the rows in pairs, starting from the top moving down.

Algorithm 5
Do Two Rows of Red/Black Gauss Seidel

```
{
 start column = 1
row = start row
Do One Row Of Paired Red/Black
start column = 2
row = start row + 1
Do One Row Of Paired Red/Black
}
```

The purpose of the `k` iteration is to cover the grid in strips of size 2μ. Figure 9 shows how a strip is moved up the grid.

Within each strip we update the rows in pairs, starting from the top moving down. Since the strips overlap, the algorithm may update the same nodes more then once. In the two sweep example shown in figure 9 the nodes with two circles around them have the same value they would have if we did two complete (non-melted) sweeps of the grid.

The calculations must start at the top of the strip to make sure that the values agree with those obtained through the standard red/black Gauss Seidel method. See figure 10.

The 'saw tooth' appearance of the updates shown in figure 9 is a consequence of the need to update the red points before the black points sitting below them. In algorithm 5 we update the nodes in row `i-1` before those in row `i`. If we updated row `i` first then we would run into the problem shown in the following figure where the black node in row `i-1` is updated before the red nodes in row `i-1`

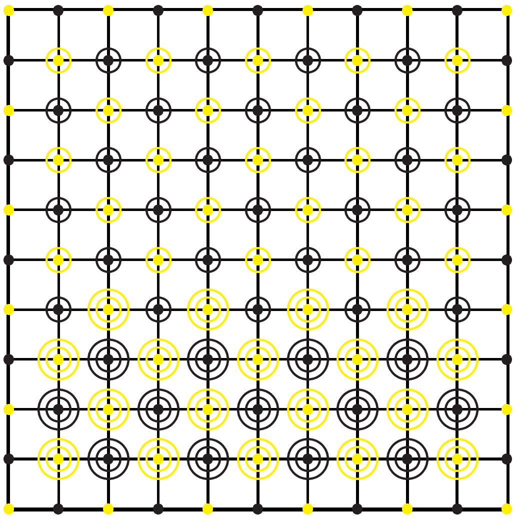
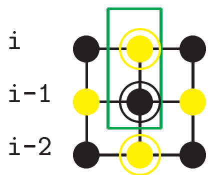


Figure 10: Order of updates when the red/black nodes are fused together. Note that the nodes in row `i+1` are updated in sweep $\alpha - 1$ before those in row `i` during sweep α.



The line labelled `fused(2,2)` in figure 7 shows the result of two calls to the melt algorithm with $\mu = 2$. We are trying to simulate the case in the multigrid algorithm where we would do a pre and post smoothing sweep with $\mu_1 = \mu_2 = 2$. The line labelled `fused(4, 0)` in figure 7 shows what happens if we melt 4 sweeps together ($\mu_1 = 4, \mu_2 = 0$).

It is interesting to note that there are the same number of operations in `fused(2,2)` as there are in `fused(4, 0)` but the Mflop rate for `fused(4, 0)` is far better. This is because we do fewer sweep through the grid (2 compared to 1) and thus reduces the number of times we must copy data into the cache.

5 V-cycle

We now focus on the performance of the V-cycle. The V-cycle consists of the following four modules

1. smoothing,
2. calculation of the residual,
3. restriction of the residual and

4. interpolation (coarse grid correction).

The smoother was described in section 4. To calculate the residual we used an algorithm like the one given in algorithm 6. A high level view of the restriction algorithm is given in algorithm 7 and the interpolation algorithm is shown in algorithm 8.

Algorithm 6
Calculate The Residual

```
{
 for (i = 1; i < grid size; i++)
    for (j = 1; j < grid size; j++)
       r[i, j]= f[i, j] – Su[i, j]
}
```

Algorithm 7
Restrict The Residual

```
{
 for (i = 2; i < grid size; i+=2)
    for (j = 2; j < grid size; j+=2)
       tmp1 = 0.25*(r[i+1, j+1] + r[i+1, j] + r[i-1, j+1] + r[i-1, j-1])
       tmp2 = 0.5*(r[i, j+1] + r[i, j-1] + r[i+1, j] + r[i-1, j])
       c[i/2, j/2]= r[i, j] + tmp1 + tmp2
}
```

Algorithm 8
Correct The Current Estimate

```
{
 for (i = 2; i < grid size; i+=2)
    for (j = 2; j < grid size; j+=2)
       u[i, j]+= c[i/2, j/2]
       u[i+1, j]+= 0.5*c[i/2, j/2]; u[i-1, j]+= 0.5*c[i/2, j/2]
       u[i, j+1]+= 0.5*c[i/2, j/2]; u[i, j-1]+= 0.5*c[i/2, j/2]
       u[i+1, j+1]+= 0.25*c[i/2, j/2]; u[i+1, j-1]+= 0.25*c[i/2, j/2]
       u[i-1, j+1]+= 0.25*c[i/2, j/2]; u[i-1, j-1]+= 0.25*c[i/2, j/2]
}
```

5.1 In-Cache Optimisations

To optimise the residual and restriction calculations we used a loop unrolling technique similar to the one shown in algorithm 2. The results for two pre and post smoothers after unrolling the smoother, restriction and residual calculations by four levels are given by the line labelled `optimised RB` in figure 11.

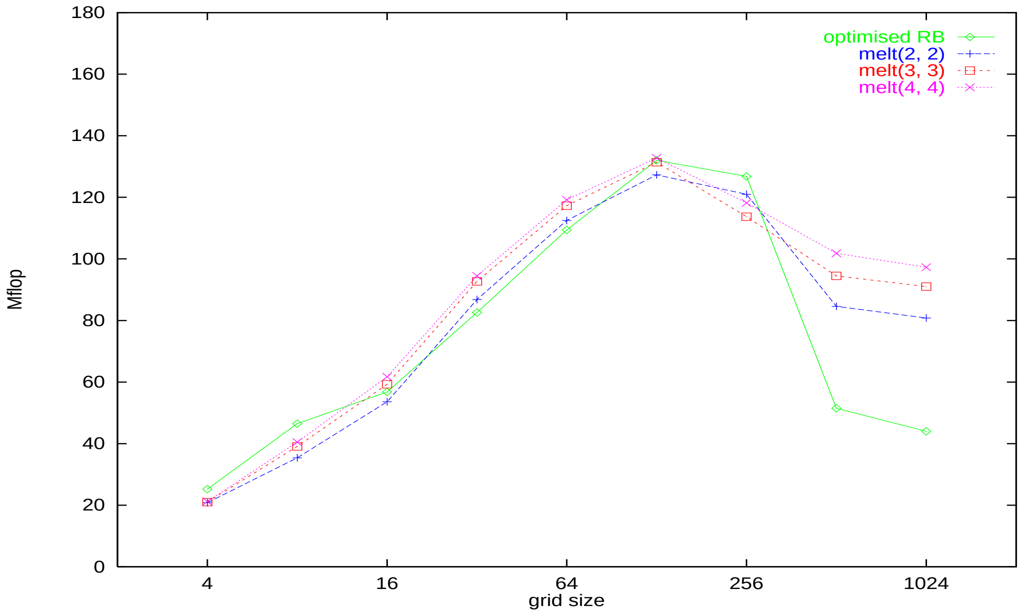


Figure 11: Mflops rate for different implementations of the V-cycle.

5.2 Out-of-Cache Optimisations

The multigrid modules can be grouped into pre coarse grid operations (smoother, residual calculation and restriction) and post coarse grid operations (interpolation and smoother). To improve the out-of-cache performance we melted the pre coarse grid operations together and the post coarse grid operations together. The other lines in figure 11 show the results, where `melt`(μ_1, μ_2) represents the number of pre and post smoothers. A more detailed description of the techniques are given in the following sections (to obtain the results in figure 11 we used approach 3 from section 5.2.1 and approach 2 from section 5.2.2). In all of the algorithms given in this section we have ignored the top and bottom boundaries; for technical reasons these have to be handled slightly differently, but the ideas are still the same.

The reason why the Mflops increase as the number of pre and post smoothers is increased is that more work is done before moving down to the coarse grids. When the algorithm moves down to the coarse grids it looses any fine grid information sitting in the cache. These results are interesting because increasing the number of pre and post smoothers also increases the convergence rate and thus gives a more effecient algorithm, which raises the question of what is the optimal number of pre and post smoothers. We plan to look into this in more detail.

5.2.1 Melting Pre Coarse Grid Operations

To reduce the number of cache-misses we tried melting the pre coarse grid operations together. We studied three different approaches. In the first one, the residual was calculated as soon as possible and then a weighting was added to the coarse grid. In the second approach the restriction operator was not called until (roughly speaking) three lines of the residual had been calculated. In the third approach the residual was calculated for as many lines as will fit into the cache and then the restriction operator was applied to those lines. Each approach will now be explained in more detail.

Approach 1 Given the coarse grid node `c[i/2, j/2]` (`i, j` is even) and the residual, `r[i, j]`, calculated at the fine grid node `u[i, j]`, the restriction operator is defined as

$$\begin{aligned}\texttt{c[i/2, j/2]} &= 0.25 * (\texttt{r[i+1, j-1]} + \texttt{r[i+1, j+1]} + \texttt{r[i-1, j-1]} + \texttt{r[i-1, j+1]}) \\ &+ 0.5 * (\texttt{r[i+1, j]} + \texttt{r[i, j-1]} + \texttt{r[i, j+1]} + \texttt{r[i, j+1]}) \\ &+ \texttt{r[i, j]}, \qquad (1)\end{aligned}$$

which may be rewritten as

$$\begin{aligned}\texttt{tmp1} &= 0.25 * (\texttt{r[i-1, j-1]} + \texttt{r[i-1, j+1]}) + 0.5 * \texttt{r[i-1, j]} \\ \texttt{tmp2} &= 0.5 * (\texttt{r[i, j-1]} + \texttt{r[i, j+1]}) + \texttt{r[i, j]} \\ \texttt{tmp3} &= 0.25 * (\texttt{r[i+1, j-1]} + \texttt{r[i-1, j+1]}) + 0.5 * \texttt{r[i+1, j]} \\ \texttt{c[i/2, j/2]} &= \texttt{tmp1} + \texttt{tmp2} + \texttt{tmp3}. \qquad (2)\end{aligned}$$

In the calculations we use a weighted restriction operator, $4 * R_k^{k-1}$, so that we do not have to explicitly divide the finite difference matrix through by h_{k-1}^2 (i.e. we use $h_{k-1}^2 A_{k-1}$). Note that the factor 4 is a consequence of $h_{k-1} = 2 * h_k$.

When using the restriction operator as defined in equation (1) we have to keep the residual calculations in a temporary storage area since the fine grid values along row `i-1` are needed to calculate the coarse grid values for row `i/2` and row `i/2-1` (`r[i-1, j]= r[(i-2)+1, j]`). Therefore, in the first approach we tried using equation 2 where the residual along row `i-1` is calculated and added to rows `i/2` and `i/2-1`. That is, we calculate the values for, say, `tmp1` as shown in equation (2) and then add `tmp1` to `c[i/2, j/2]` and to `c[i/2-1, j/2]`. A high level view of the procedure is given in algorithm 9.

Algorithm 9
Melt Pre Coarse Approach 1

```
{
 for (k = 2*(μ + 1); k < grid size - 1; k+= 2)
     smoother strip
     for (i = k; i > k - 2μ; i-= 2)
        start row = i-1
        Do Two Rows of Red/Black Gauss Seidel (Algorithm 5)
     melt restriction and residual calculations
     i = k - 2μ
     Restrict Two Rows (Algorithm 10)
}
```

The residual is always calculated on the row below the strip as enforced by the data dependencies. For example, the residual for the indicated point in figure 12 can not be calculated because the above black point has not been updated and it will not be updated until the strip has moved two places.

Algorithm 10 shows how the residual and restriction calculations are melted together. In the first `j` loop the residual for the odd numbered rows is calculated (i.e. `tmp1` and `tmp3` in equation (2)) where `left = r[i-1, j-1] = r[(i-2)+1, j-1]`, `middle = r[i-1, j] = r[(i-2)+1, j]` and `right = r[i-1, j+1] = r[(i-2)+1, j+1]`.

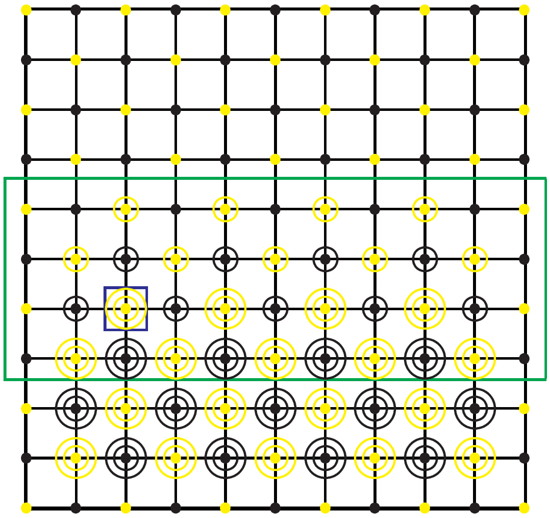


Figure 12: In the first approach, the residual calculation can not take place within the strip. The residual for the indicated nodes can not be calculated because some of the neighbouring nodes are not up to date.

In the second `j` loop the residual along the even numbered rows is calculated (i.e. `tmp2` in equation 2).

Algorithm 10
Restrict Two Rows

```
{
 odd rows
right = f[i-1, 0] - Su[i-1, 0]
for (j = 2; j < grid size; j+= 2)
    left = right
    middle = f[i-1, j] - Su[i-1, j]
    right = f[i-1, j+1] - Su[i-1, j+1]
    c[i/2, j/2] = 0.25 left + 0.5 middle + 0.25 * right
    c[(i-2)/2, j/2] += 0.25 left + 0.5 middle + 0.25 * right
even rows
right = f[i, 0] - Su[i, 0]
for (j = 2; j < grid size; j+= 2)
    left = right
    middle = f[i, j] - Su[i, j]
    right = f[i, j+1] - Su[i, j+1]
    c[i/2, j/2] += 0.5 left + middle + 0.5 * right
}
```

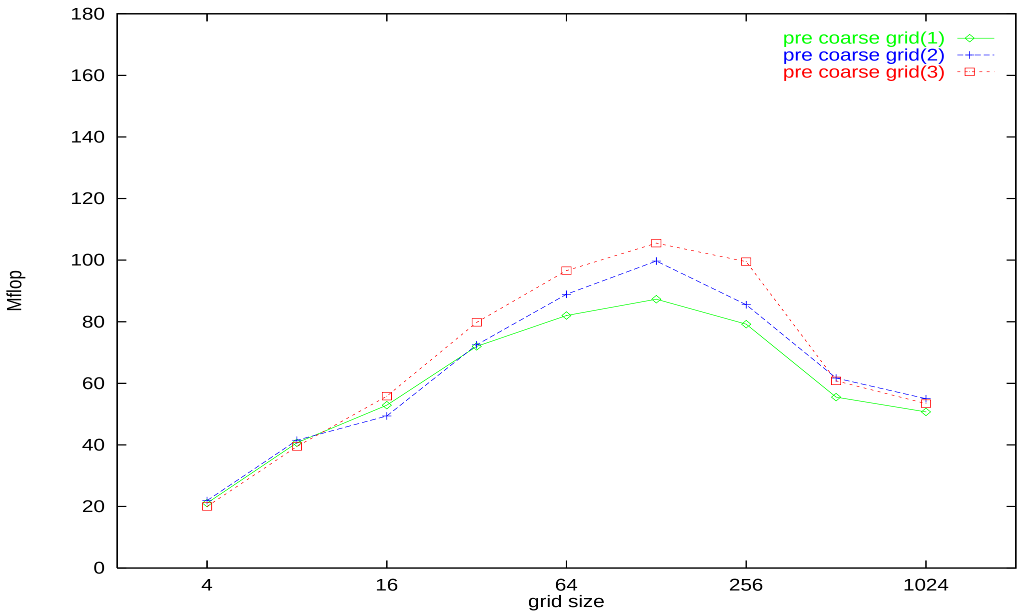


Figure 13: Mflops rate for different approaches of melting the pre coarse grid operations. $\mu_1 = \mu_2 = 2$.

The Mflops rate for approach 1 is shown by the line labelled `pre coarse grid(1)` in figure 13. Note that we have not melted (or optimised) the post coarse grid operations as we are just interested in the pre coarse grid operations in this case, which is why the Mflops rates in figure 13 are lower then those in figure 11.

Approach 2 The disadvantage of using equation 2 as opposed to equation 1 is that it has a higher operation count. For example, instead of calculating `0.5*(r[i+1, j] + r[i, j-1] + r[i, j+1] + r[i, j+1])` we calculate `0.5*(r[i+1, j]+ r[i, j-1])+ 0.5*r[i, j+1] + 0.5*r[i, j+1]`. In the second approach we actually use equation 1 and calculate (roughly speaking) three rows of residuals before restricting the values down to the coarse grid. A high level view of the algorithm is given in algorithm 11.

Algorithm 11
Melt Pre Coarse Approach 2

```
{
 for (k = 2*(μ + 3); k < grid size - 1; k+= 2)
     smoother strip
     for (i = k; i > k - 2μ; i-= 2)
        start row = i - 1
        Do Two Rows of Red/Black Gauss Seidel (Algorithm 5)
     melt restriction and residual calculation
     i = k - 2(μ+1)
     Restrict Three Rows (Algorithm 12)
}
```

In this version we must do the residual/restriction calculations two rows behind the cur-

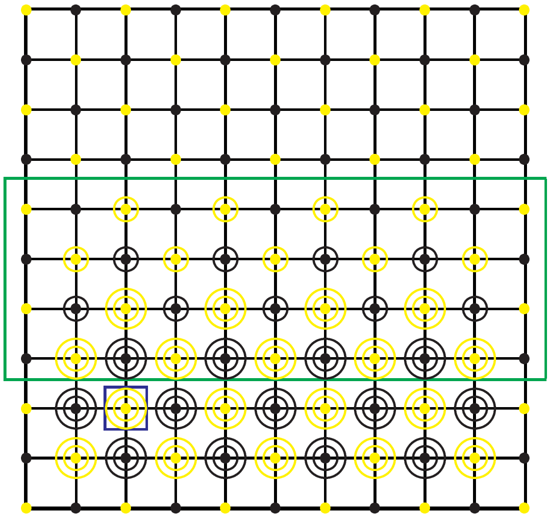


Figure 14: In the second approach, the residual calculation must lag two rows behind the smoother strip. If the indicated node also belongs to the coarse grid, then we could not restrict to this node because the residual for the black node sitting above it can not be calculated.

rent smoothing strip. For example, in the figure 14 we could not restrict down to the coarse grid node because the residual calculations for the node above it can not be completed.

Algorithm 12
Restrict Three Rows

```
{
 swap(top array, bottom array) (only swaps pointers not the actual values)
 for (j = 1; j < grid size; j++)
     centre array[j] = f[i, j]- Su[i, j]
 for (j = 1; j < grid size; j++)
     top array[j] = f[i+1, j] - Su[i+1, j]
 for (j = 2; j < grid size-2; j++)
     c[i/2, j/2]= 0.25*(top array[j-1] + top array[j+1]
           + bottom array[j-1] + bottom array[j+1])
        + 0.5(top array[j] + bottom array[j]
           + centre array[j-1] + centre array[j+1])
        + centre array[j]
}
```

The results for approach 2 are shown by the line labelled `pre coarse grid(2)` in figure 13. This method works slightly better then the method described in the previous section. We believe this is due to the reduced number of operations.

Approach 3 The final approach is the easiest to implement. We smooth over p rows and then calculate the residual for p rows and finally apply the restriction operator to p rows. The size of p is chosen so that p rows will fit into the cache and is given by a `#define` variable set a run time. We do not give a high level algorithm in this case because the code is essentially the same as that given in section 5.1, the only difference is that the restriction calculations must lag $2(\mu + 1)$ steps behind the smoother (`k` iteration as in algorithm 11).

The Mflops rate obtainable through the third approach is shown by the line labelled `pre coarse grid(3)`. In terms of the out-of-cache performance, there is not much difference between the three approaches, however we favour approach 3 as we can simply build upon the code designed to maximise the in-cache performance.

5.2.2 Melting Post Coarse Grid Operations

This section looks at how we melted the coarse grid correction and the red/black Gauss Seidel method. We tried two different approaches. In the first approach, the smoother was called as soon as enough information had been interpolated back up to the fine grid. In the second approach several coarse grid rows were interpolated and then the post smoother was called.

Approach 1 In the first approach, shown in algorithm 13, we used the coarse grid to correct one line on the fine grid and then applied the smoother to as many lines as possible. The smoother has to lag two rows behind the correction operator because the values on the fine grid row `i+1` (`i` is even) are not up to date until they have received corrections from the coarse grid rows `i/2` and `i/2+1`.

Algorithm 13
Melt Post Coarse Approach 1

```
{
 for (k = 2*(μ + 1); k < grid size - 2; k+= 2)
    interpolation
    i = k+2
    for (j = 2; j < grid size; j+=2)
       u[i, j]+= c[i/2, j/2]
       u[i+1, j]+= 0.5*c[i/2, j/2], u[i-1, j]+= 0.5*c[i/2, j/2]
       u[i, j+1]+= 0.5*c[i/2, j/2], u[i, j-1]+= 0.5*c[i/2, j/2]
       u[i+1, j+1]+= 0.25*c[i/2, j/2], u[i+1, j-1]+= 0.25*c[i/2, j/2]
       u[i-1, j+1]+= 0.25*c[i/2, j/2], u[i-1, j-1]+= 0.25*c[i/2, j/2]
    smoother strip
    for (i = k; i > k - 2μ; i-= 2)
       start row = i-1
       Do Two Rows of Red/Black Gauss Seidel(Algorithm 5)
}
```

The results are shown by the line labelled `post coarse grid(1)` in figure 15. We used approach 3 to melt the pre coarse grid operations.

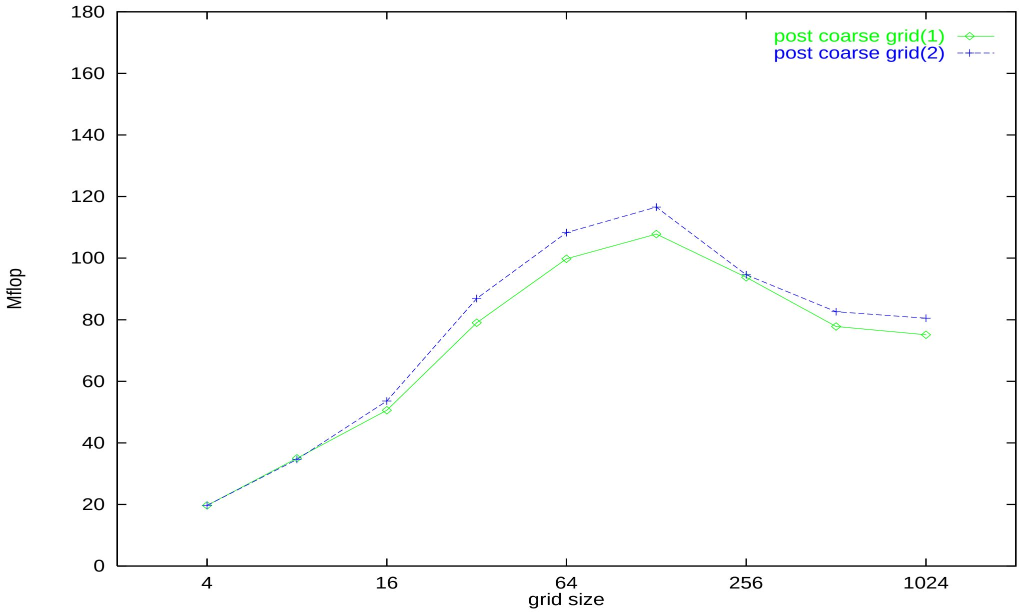


Figure 15: Mflops rate for different approaches of melting the post coarse grid operations. $\mu_1 = \mu_2 = 2$.

Approach 2 The idea behind this approach is similar to approach 3 described in section 5.2.1. Namely, we apply the interpolation to p rows and then smooth p rows. The smoother must lag two rows behind the coarse grid correction.

The results for approach 2 are given by the line labelled `post coarse grid(2)` in figure 15. Once again, it seems that the easier approach worked the best.

6 FMV-Scheme

This section looks at the Mflops rate for the FMV-scheme. As shown in figure 2, the FMV-scheme is built upon the V-cycle, the only exception is the interpolation operator $\widetilde{I}^k_{k-1}$. From the classical multigrid theory [3] it is known that if $\widetilde{I}^k_{k-1}$ is of sufficiently high order, a fixed number of V-cycles is sufficient to compute a solution to the level of truncation errors. In our case, to maintain the $O(h^4)$ accuracy offered by the compact 9-point stencil it was necessary to use a polynomial interpolation of degree 4.

Suppose the grid size is $h_3 = 1/8$ and define the two operators $\mathtt{I_i}$ and $\mathtt{I_j}$ as

$$\mathtt{I_i[i,j]} = \begin{cases} \frac{35}{128}\mathtt{u[i,0]} + \frac{35}{32}\mathtt{u[i,2]} + \frac{-35}{64}\mathtt{u[i,4]} + \frac{7}{32}\mathtt{u[i,6]} + \frac{-5}{128}\mathtt{u[i,8]} & \text{if } \mathtt{j} = 1 \\ \frac{-5}{128}\mathtt{u[i,0]} + \frac{15}{32}\mathtt{u[i,2]} + \frac{45}{64}\mathtt{u[i,4]} + \frac{-5}{32}\mathtt{u[i,6]} + \frac{3}{128}\mathtt{u[i,8]} & \text{if } \mathtt{j} = 3 \\ \frac{3}{128}\mathtt{u[i,0]} + \frac{-5}{32}\mathtt{u[i,2]} + \frac{45}{64}\mathtt{u[i,4]} + \frac{15}{32}\mathtt{u[i,6]} + \frac{-5}{128}\mathtt{u[i,8]} & \text{if } \mathtt{j} = 5 \\ \frac{-5}{128}\mathtt{u[i,0]} + \frac{7}{32}\mathtt{u[i,2]} + \frac{-35}{64}\mathtt{u[i,4]} + \frac{35}{32}\mathtt{u[i,6]} + \frac{35}{128}\mathtt{u[i,8]} & \text{if } \mathtt{j} = 7 \\ \mathtt{u[i,j]} \text{ otherwise} & \end{cases}$$

$$
\mathtt{I_j[i,j]} = \begin{cases} \frac{35}{128}\mathtt{u[0,j]} + \frac{35}{32}\mathtt{u[2,j]} + \frac{-35}{64}\mathtt{u[4,j]} + \frac{7}{32}\mathtt{u[6,j]} + \frac{-5}{128}\mathtt{u[8,j]} & \text{if } \mathtt{i} = \mathtt{1} \\ \frac{-5}{128}\mathtt{u[0,j]} + \frac{15}{32}\mathtt{u[2,j]} + \frac{45}{64}\mathtt{u[4,j]} + \frac{-5}{32}\mathtt{u[6,j]} + \frac{3}{128}\mathtt{u[8,j]} & \text{if } \mathtt{i} = \mathtt{3} \\ \frac{3}{128}\mathtt{u[0,j]} + \frac{-5}{32}\mathtt{u[2,j]} + \frac{45}{64}\mathtt{u[4,j]} + \frac{15}{32}\mathtt{u[6,j]} + \frac{-5}{128}\mathtt{u[8,j]} & \text{if } \mathtt{i} = \mathtt{5} \\ \frac{-5}{128}\mathtt{u[0,j]} + \frac{7}{32}\mathtt{u[2,j]} + \frac{-35}{64}\mathtt{u[4,j]} + \frac{35}{32}\mathtt{u[6,j]} + \frac{35}{128}\mathtt{u[8,j]} & \text{if } \mathtt{i} = \mathtt{7} \\ \mathtt{u[i,j]} \text{ otherwise} \end{cases}
$$

If the `u[i, k]` terms in $\mathtt{I_i}$ are replaced by `c[i/2, k/2]` for even `i` and even `k` then $\mathtt{I_i}$ is just polynomial interpolation of degree 4 with respect to the x-direction. Similarly, if the `u[k, j]` terms in $\mathtt{I_j}$ are replaced by `c[k/2, j/2]` for even `j` and even `k` then $\mathtt{I_j}$ is just polynomial interpolation of degree 4 with respect to the y-direction.

The interpolation operator in each direction can then be used as shown in algorithm 14 to find the interpolated values on the 2D grid with size h_3.

Algorithm 14
FMV Interpolation

```
{
 for (i = 2; i < 8; i+=2)
     FMV Interpolation Even Row (Algorithm 15)
 for (i = 1; i < 8; i+=2)
     FMV Interpolation Odd Row (Algorithm 16)
}
```

Algorithm 15
FMV Interpolation Even Row

```
{
  even columns
 for (j = 2, j < 8; j+=2)
     u[i, j] = c[i/2, j/2]
 odd columns
 for (j = 1, j < 8; j+=2)
     u[i,j] = I_i[1,j]
}
```

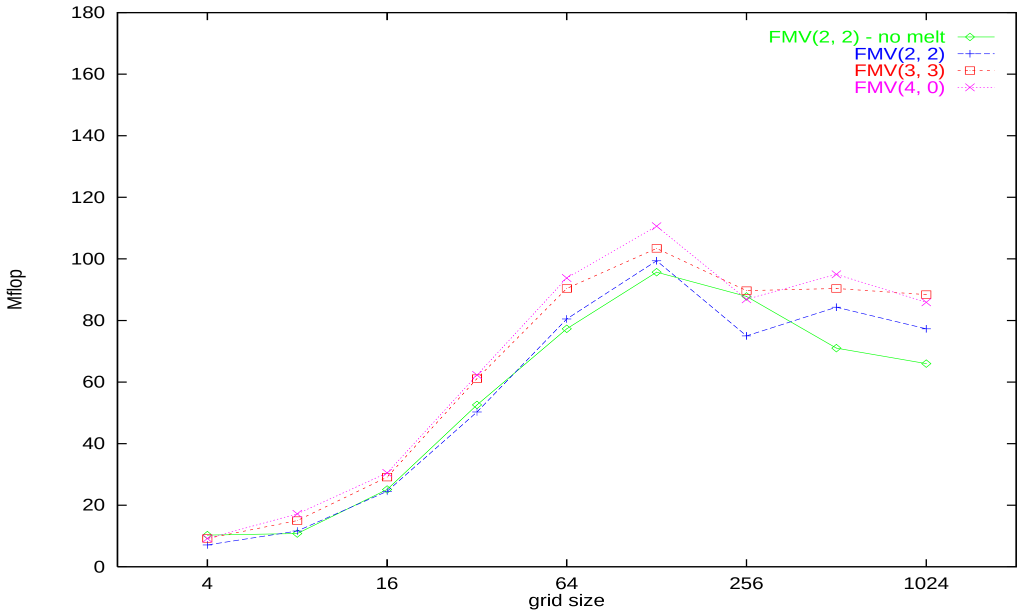


Figure 16: Mflops rate for different number of pre and post smoothers.

Algorithm 16
FMV Interpolation Odd Row

{
 even columns
 `for (j = 2, j < 8; j+=2)`
  $\texttt{u}[\texttt{i},\texttt{j}] = \texttt{I}_\texttt{j}[\texttt{i},\texttt{j}]$
 odd columns
 `for (j = 1, j < 8; j+=2)`
  $\texttt{u}[\texttt{i},\texttt{j}] = \texttt{I}_\texttt{i}[\texttt{i},\texttt{j}]$
}

To handle general grids, we use piecewise interpolation, that is, we simply view the grid as partioned in a set of patches each with 5×5. grid lines.

The Mflops rate for the FMV-scheme is given in figure 16. The lines labelled $\texttt{FMV}(\mu_1, \mu_2)$ correspond to different numbers of pre and post smoothers. The operations in the $\texttt{V\_cycle}_k$ part of the algorithm have been melted as described in section 5.2. We could melt the polynomial interpolation with the pre coarse grid operations, but for simplicity we have not done this and the results in figure 16 imply that it has not severely affected the out-of-cache performance.

7 Numerical Results

We now interpret the results from a more mathematical point of view. The main aim of this section is to compare the performance of the compact 9-point stencil and the 5-point stencil. When developing finite difference software the 5-point stencil is usually preferred because of its simplicity and smaller operation count. Mathematical studies of the finite difference method also tend to focus on the 5-point stencil. However, we have found that the compact 9-points has a higher Mflops rate and combined with the higher convergence rate this suggests that it is the more efficient method.

The question of how much the architecture should affect the mathematical method is one of great interest to us. The driving force behind this project is the development of software which makes more efficient use of the cache.

7.1 Test Problems

The test problems we considered include

1.
$$\Delta u = -\sin(\pi x)\sin(\pi y),$$
on the unit square with $u = 0$ on the boundary.

2.
$$\Delta u = f.$$
on the unit square with boundary conditions and f such that $\sin(\sinh(3x)) * \sin(\sinh(3y))$ is the exact solution.

3.
$$\Delta u = f,$$
on the unit square with the boundary conditions and f such that $\cos(32\pi((x-1/2)^2 + (y-1/2)^2))\exp(-8(x-1/2)^2 - 8(y-1/2)^2)$ is the exact solution.

7.2 5-Point Stencil v's Compact 9-Point Stencil

7.2.1 V-Cycle

Figure 17 shows the error for the three test problems when using the V-cycle with the 5-point and compact 9-point stencils. In all cases we used 2 pre and 2 post smoothers. The numbers in brackets indicate how many V-cycles it required to reach the truncation error level on the largest grid size. The y-axis uses a $\log_{16}$ scaling while the x-axis uses a $\log_2$ scaling. Since the 9-point stencil is $O(h^4)$ we would expect to see its results fit a line with slope 1. The results for the 5-point stencil should look like a line with slope 1/2.

The graphs show that we get the expected $O(h^2)$ convergence rate for the 5-point stencil and $O(h^4)$ for the 9-point stencil. The increase in the error for the compact 9-point stencil with grid size = 1024 in test problem 1 is due to rounding errors.

In figure 18 we compare the Mflops rate for the 5-point stencil and the 9-point stencil. To do the calculations with the 5-point stencil we replaced the structure which holds the 9-point stencil with one which holds a 5-point stencil. We also had to remove all occurrences of the variables corresponding to `u[i+1, j-1]`, `u[i+1, j+1]`, `u[i-1, j-1]` and `u[i-1, j+1]` from the code. Such a change decreased the computation time, but as figure 18 shows it also

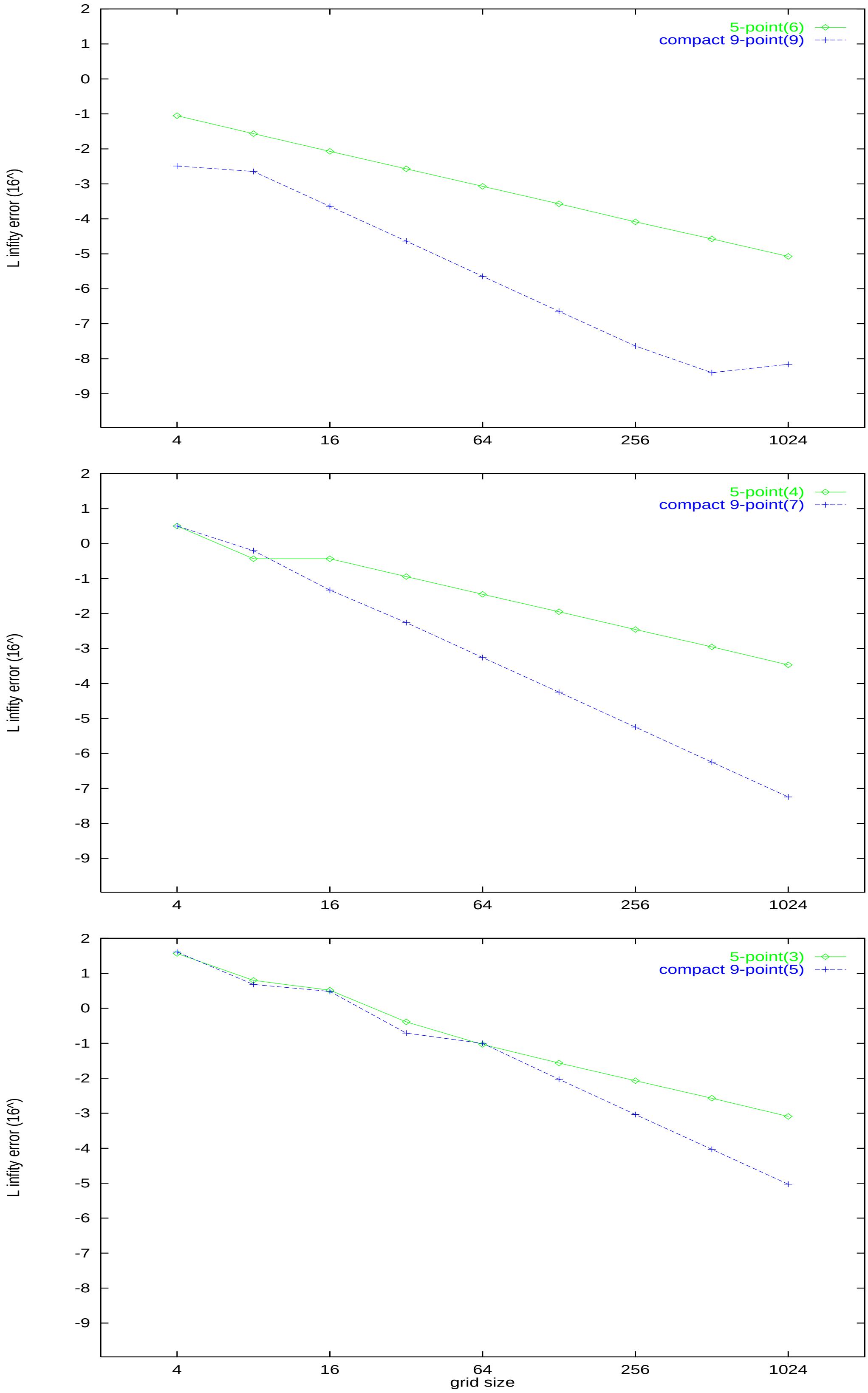


Figure 17: Error for test problems 1, 2 and 3 using the V-cycle with $\mu_1 = \mu_2 = 2$.

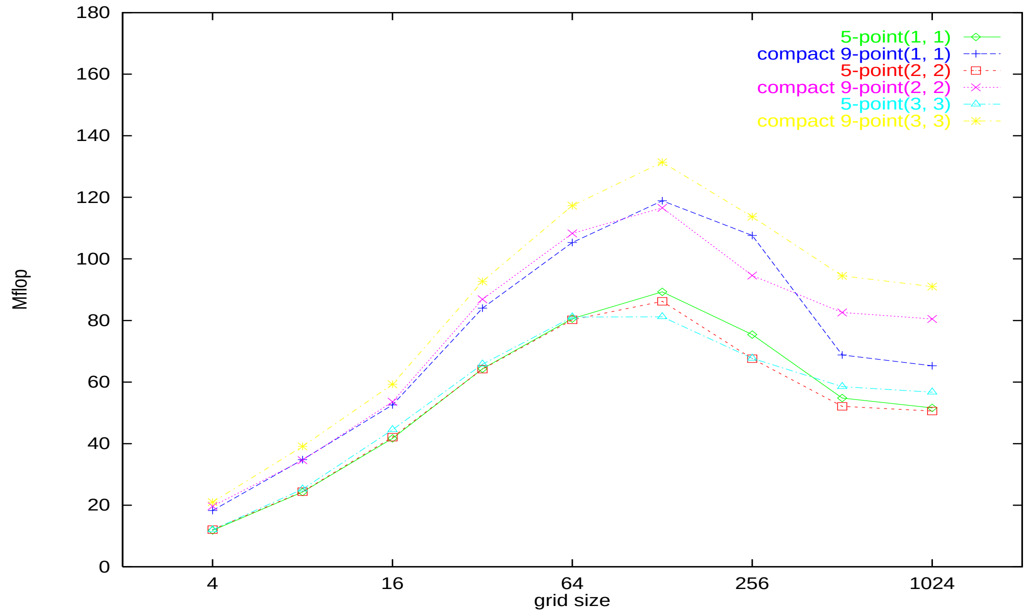


Figure 18: Mflops rate for different number of pre and post smoothers of the 5 point and 9 point stencils.

decreased the Mflops rate. We believe this is because the 9-point stencil makes better use of data locality.

A, possibly, more useful comparison is given in figure 19. Here we look at how much the error is reduced within a certain time. Once again, the numbers in brackets show how many V-cycles it required to reach the order of convergence on the largest grid size. We used a $\log_{16}$ scaling for the y-axis and a $\log_4$ scaling for the x-axis. The reason for the $\log_4$ scaling is that when the grid size is halved the total number of nodes is increased by 4 and therefore the time is roughly multiplied by 4.

Clearly the compact 9-point stencil outperforms the 5-point stencil. We have taken into account that that the compact 9-point requires more iterations to reach the convergence rate. Furthermore, we used the fact that the residual on the black nodes after a red/black Gauss Seidel sweep is 0 for the 5-point stencil to reduce the number of residual calculations.

7.2.2 FMV-scheme

Figure 20 shows the error for the three test problems when using the FMV-scheme. The advantage of the FMV-scheme is that it only needs 1 iteration to reach the truncation error. The figure shows that we get the expected convergence rates. For the 9-point stencil a fourth order polynomial was used as described in section 6, however for the 5-point stencil we only needed to use quadratic interpolation.

Figure 21 looks at the reduction in error verses the time. Notice that for the larger grid sizes the time to complete one iteration of the FMV-schem is roughly the same for both the 9-point stencil and 5-point stencil. Therefore, after taking the difference in convergence rates into account it becomes clear the the 9-point stencil far outperforms the 5-point stencil.

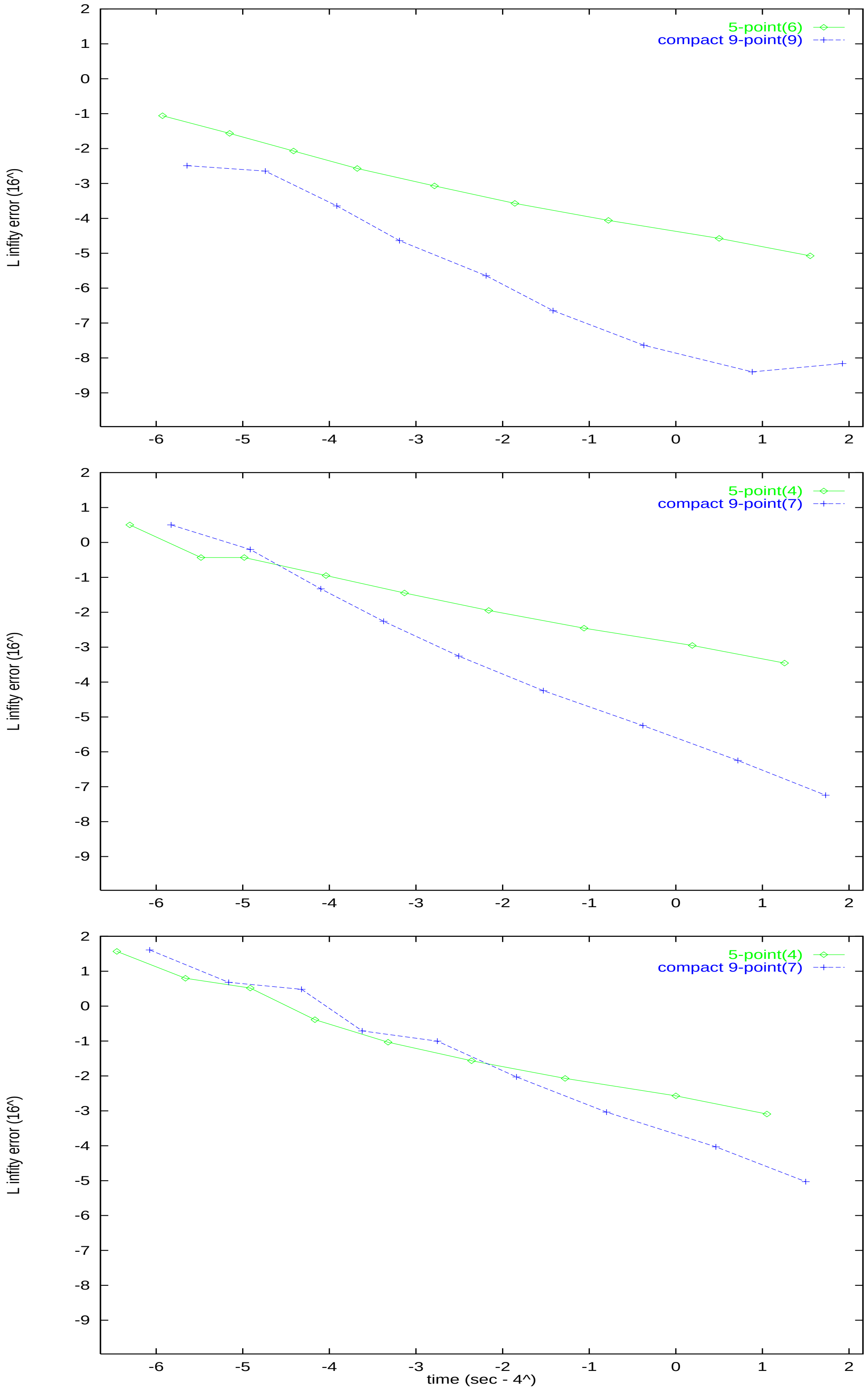


Figure 19: Error v's time for test problem 1, 2, 3 using the V-cycle with $\mu_1 = \mu_2 = 2$

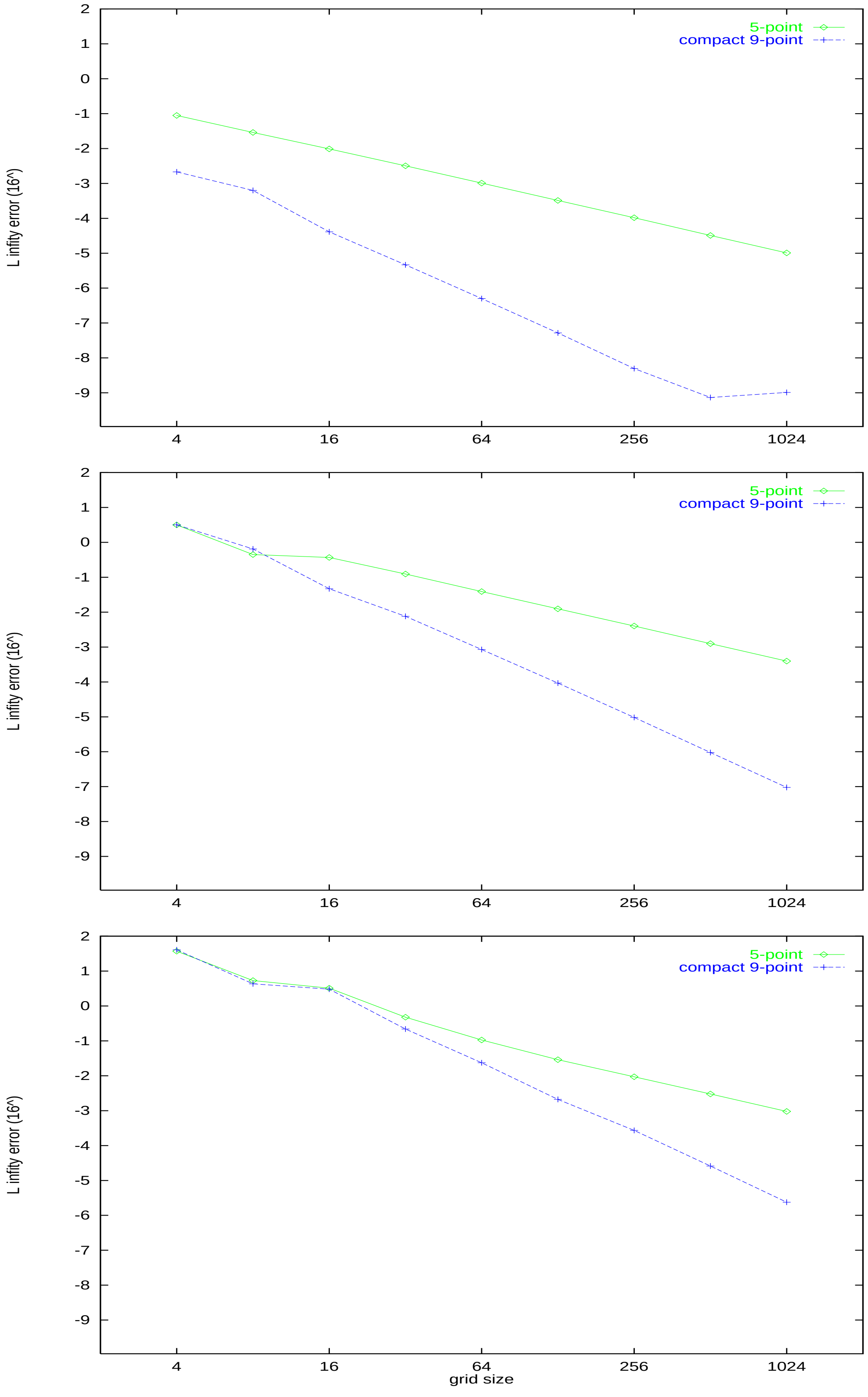


Figure 20: Error for test problems 1, 2 and 3 using the FMV-scheme with $\mu_1 = \mu_2 = 2$.

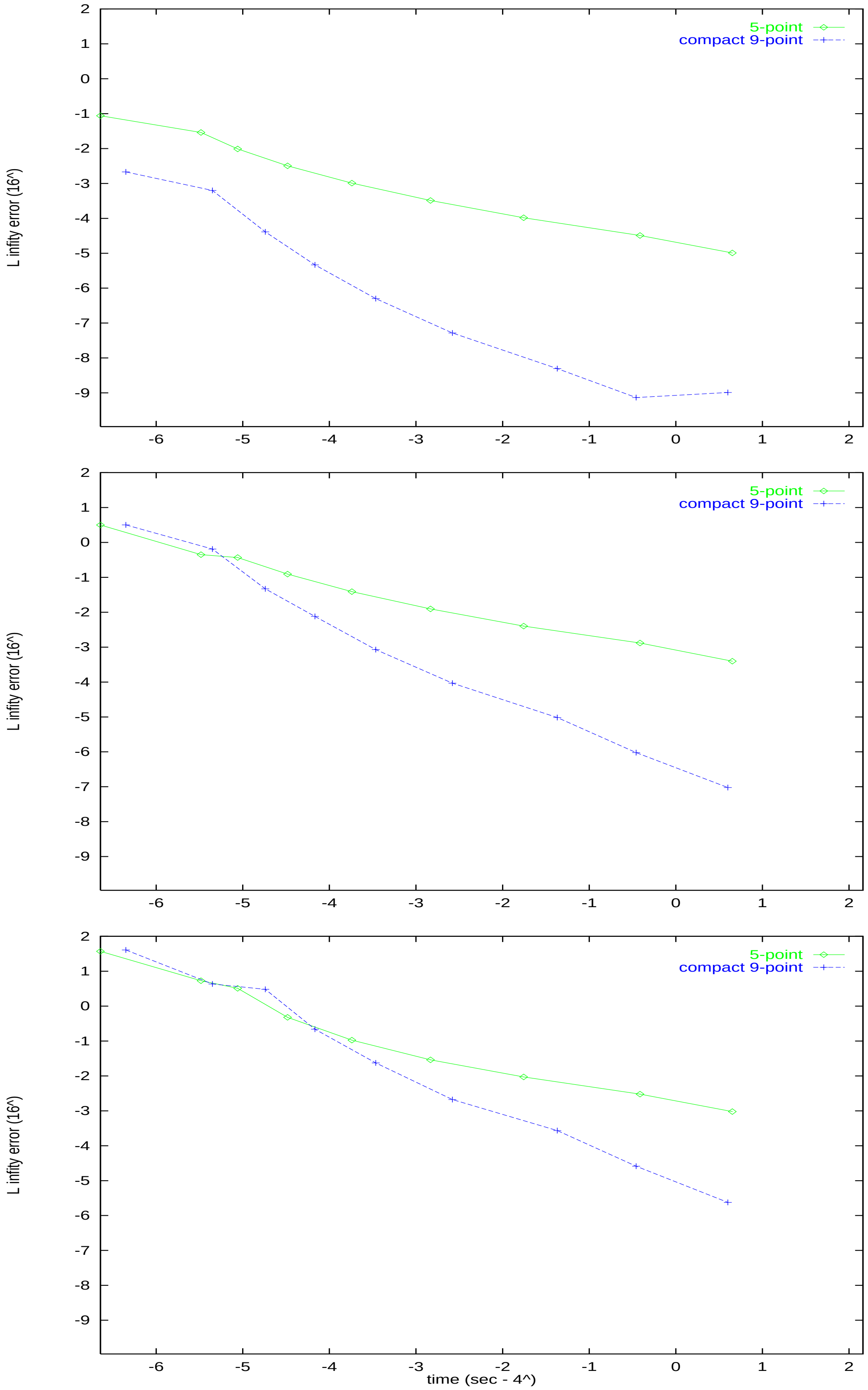


Figure 21: Error v's time for test problem 1, 2, 3 using the V-cycle with $\mu_1 = \mu_2 = 2$ (NOTE: The timings for the 5-point are incorrect).

Acknowledgements

This work is partly being funded by the Deutsche Forschungsgemeinschaft (grant number RU 422/7-1) as a joint project between the authors and Hermann Hellwagner and Christian Weiss of the Technische Universität München who have contributed part of the numerical experiments.

This report has been completed while the second author was visiting the School of Mathematical Sciences of the Australian National University in September/October 1997. He gratefully acknowledges the support and the many interesting and stimulating discussions on various aspects of this work.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson. *LAPACK user's guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[2] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Math Comput.*, 31(138):333–390, April 1977.

[3] A. Brandt. *Multigrid techniques: 1984 guide with applications to fluid dynamics.* GMD-Studien, Nr. 85. Gesellschaft für Mathematik und Datenverarbeitung MBH, Bonn, May 1984.

[4] W. Briggs. *A multigrid tutorial.* SIAM, 1987.

[5] D. C. Burger, J. R. Goodman, and A. Kägi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical Report TR-95-1261, University of Wisconsin, Dept. of Computer Science, Madison, 1995.

[6] J. Dongarra. Performance of various computers using standard linear equations software. Computer Science Department, University of Tennessee, Ridge. CS-80-85, July 1997.

[7] C. Douglas. Caching in with multigrid algorithms: Problems in two dimensions. IBM Research Report RC 20091, IBM, Yorktown Heights, NY, 1995.

[8] S. Fried. A tutorial on NDP Alpha compiler driver and optimizations FIND OUT, December 1996.

[9] S. Goedecker and A. Hoisie. Achieving high performance in numerical computations on RISC workstations and parallel systems. FIND OUT, June 1997.

[10] M. Griebel. *Zur Lösung von Finite-Differenzen und Finite-Element-Gleichungen mittels der Hierarchischen-Transformations-Mehrgitter-Methode.* PhD thesis, Fakultät für Mathematik und Informatik, Technischen Universität München, Deutschland, 1989.

[11] W. Hackbusch *Elliptic differential equations: theory and numerical treatment.* Springer Series in Computational Mathematics 18. Springer-Verlag, 1992.

[12] W. Hackbusch. *Multigrid methods and applications.* Springer Verlag, Berlin, 1985.

[13] W. Hackbusch and U. Trottenberg, editors. *Lecture notes in mathematics, 960.* Springer-Verlag, 1982.

[14] J. A. Krommes. FWEB: A WEB system of structured documentation for multiple languages. WWW address : `http://w3.pppl.gov/~krommes/fweb_toc.html`

[15] S. F. McCormick. *Multigrid Methods.* SIAM Frontiers In Applied Mathematics, 1987.

[16] D. Patterson. *Microprocessors in 2020.* Scientific American, September 1995.