Experiences Tuning SMG98 — a Semicoarsening Multigrid Benchmark based on the *hypre* Library

Guohua Jin
Dept. of Computer Science
Rice University
Houston, TX
jin@cs.rice.edu

John Mellor-Crummey
Dept. of Computer Science
Rice University
Houston, TX
johnmc@cs.rice.edu

ABSTRACT

LLNL's *hypre* library is an object-oriented library for the solution of sparse linear systems on parallel computers. While *hypre* facilitates rapid-prototyping of complex parallel applications, our experience is that without careful attention to temporal data locality, node performance of applications developed using *hypre* will fall significantly short of peak performance on architectures based on modern microprocessors. In this paper, we describe our experiences analyzing and tuning the performance of SMG98, a benchmark that exercises *hypre*'s semicoarsening multigrid solver. In the original code, the lack of temporal data reuse in the registers and caches significantly hurts performance. We describe a variety of techniques we applied to hand-tune the performance of *hypre*'s semicoarsening multigrid solver. We expect that similar strategies will be applicable to other solvers and codes based on *hypre* as well. We present performance measurements of SMG98 on both SGI Origin and Compaq Alpha platforms. Overall, our optimizations improve the node performance of SMG98 by nearly a factor of two on large problems.

Categories and Subject Descriptors

G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*multigrid and multi-level methods*; D.1.4 [**Software**]: Programming Techniques—*performance*

General Terms

Algorithms, performance, design.

Keywords

Multigrid, memory hierarchy, performance tuning, stencils, time skewing, cache blocking.

1. INTRODUCTION

Multigrid methods are a class of techniques for performing fast, iterative solves of linear systems in linear time and space. These methods improve upon the convergence rate of classic iterative methods by using a hierarchy of grids at different resolutions. Coarser levels of the grid hierarchy are effective for quickly eliminating low frequency components of solution error. As a result, the total computational work required by a multigrid method to achieve a prescribed level of accuracy is proportional to the grid size.

Multigrid methods were originally developed for efficient solution of linear elliptic partial differential equations (PDEs); however, they have been successfully applied to a broad spectrum of problems including differential problems such as those that arise in flows, electromagnetism, magnetohydrodynamics, quantum mechanics and structural mechanics as well as non-differential problems such as those that arise in geodesy, image reconstruction, pattern recognition, design and optimal control [2]. Many techniques have been proposed for improving the robustness of multigrid methods when applied to problems with degenerate coefficients, such as problems with highly discontinuous and anisotropic coefficients [1, 10, 16]. Because of their remarkable range of applicability and fast convergence, multigrid methods are being used increasingly by scientific applications for solving large-scale problems. For this reason, the performance of multigrid methods on modern architectures is of significant interest.

Achieving top performance with scientific applications on modern computer systems has become more difficult with each new generation of hardware. In large part, this effect is due to the widening gap between processor and memory speeds. Most, but not all, modern computer systems provide multi-level memory hierarchies to help hide this gap. Typically systems provide two or more levels of caches in addition to registers. Today, efficiently using caches is widely considered to be critical for achieving high performance. By obtaining data from caches instead of memory, each processor can reduce the time it spends waiting for data and achieve a higher utilization. Getting the most performance benefit from a cache requires reusing cache-resident data as many times as possible before it is evicted. When optimizing for cache-based systems, one must be particularly careful to structure data so that data will not be brought into a cache unless it is about to be used. Bringing unnecessary data into a cache not only squanders memory hierarchy bandwidth, but also may displace data that is still needed.

```
1. Pre-relax on $A^h U^h = F^h$ by performing $v_{pre}$ red/black
   sweeps based on an initial guess $u^h$.
2. If grid is not the coarsest
   2.1. Set $F^{2h} \leftarrow I_h^{2h}(F^h - A^h u^h)$.
   2.2. Solve $A^{2h} U^{2h} = F^{2h}$ by applying the algorithm
        recursively.
   2.3. Correct $u^h \leftarrow u^h + I_{2h}^h u^{2h}$.
3. Post-relax on $A^h U^h = F^h$ by performing $v_{post}$ red/black
   sweeps based on initial guess $u^h$.
```

Figure 1: 3D semicoarsening multigrid algorithm.

Several researchers have been active in evaluating and optimizing cache performance of iterative methods [12, 19, 18, 20]. Douglas [12] investigated the effect caches can have on performance of multigrid algorithms and described how to develop cache-aware multigrid algorithms from an algorithm designer's point of view. No experimental results were reported. Weiß *et al.* [19] investigated cache optimizations for red-black Gauss-Seidel relaxation and a full-coarsening multigrid V-cycle. However, further research in this area is needed, both to improve the generality of techniques for improving performance of this class of applications and to study their applicability to whole codes.

In this paper, we discuss performance issues for SMG98, a code that exercises a semicoarsening multigrid solver implemented in the *hypre* library, and describe techniques to improve its cache and overall performance. In Section 2, we first briefly discuss the semicoarsening multigrid method and its potential performance problems. In Section 3, we describe *hypre*, LLNL's object-oriented library for solving sparse linear systems. In Section 4, we present *run-time stencil factoring*, a technique to improve data locality of stencil computations. In Section 5, we present *global temporal skewing and blocking*, an optimization for increasing data reuse across multiple sweeps of the data domain. In Section 6, we discuss a data exchange strategy for improving both sequential and parallel performance. We present experimental results in Section 7 and conclude with some observations about our experiences.

2. SEMICOARSENING MULTIGRID

Semicoarsening multigrid was developed to avoid costly but necessary alternating plane relaxations used in the "black box" full-coarsening three-dimensional multigrid algorithm developed by Dendy [7, 8, 9, 10]. In contrast to full-coarsening multigrid, the semicoarsening approach coarsens grids more gradually, namely, along only one dimension at each coarsening step. Semicoarsening multigrid was originally implemented by Schaffer and Jones [15], and independently by Dendy [11]. Schaffer recently improved the approach with a more sophisticated definition of the prolongation and restriction operators [16] and this approach was implemented for distributed memory machines by Brown, Falgout, and Jones [4].

In semicoarsening multigrid, planes and lines of a grid are colored red and black in an alternating fashion. Red and black plane (line) relaxations are performed in two separate sweeps. Assume $AU = F$ is the given linear system to solve, where the unknown U and right-hand side F are vectors defined on a given grid space. A is a symmetric, positive definite matrix of various point stencil forms. A recursive description of a 3D semicoarsening multigrid, based on one given by Brown, Falgout, and Jones [4], is shown in Figure 1. The superscripts and subscripts of h and $2h$ in the algorithm represent a finer grid and a coarser grid, respectively. I_{2h}^h is the interpolation operator to transfer an error correction from a coarser grid to a finer grid. I_h^{2h} is the restriction operator to transfer residuals from a finer grid to a coarser grid. The matrix A on the coarse grid is defined as $A^{2h} = I_h^{2h} A^h I_{2h}^h$.

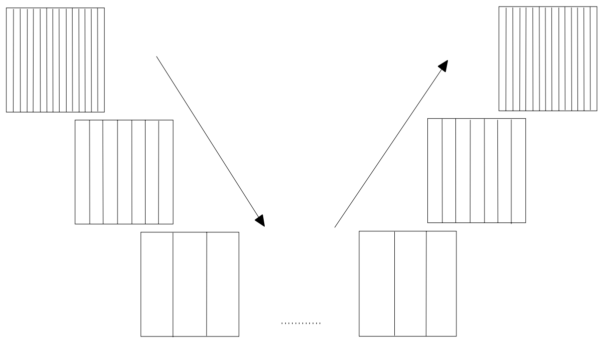


Figure 2: A V-cycle along y-dimension (the horizontal dimension).

Beginning with the finest grid in the z-dimension, the algorithm walks down to the coarsest grid recursively and then works its way back up to the finest grid. The 3D solve at each coarsening level along the z-dimension is accomplished by computing a 2D plane solve for each xy-plane in the volume. Each 2D xy-plane solve is computed using recursive coarsening along the y-dimension. Figure 2 shows a schematic view that illustrates a hierarchy of plane coarsenings along the y-dimension. Because of the order in which the hierarchy of grids is traversed, this is known as a V-cycle. At each coarsening level along the y-dimension, a 1D line solve is computed directly for each x-line in the plane using cyclic reduction. At each coarsening level in the z- and y-dimensions, red/black relaxation is employed. First, all of the red points are updated to satisfy their equations, then all of the black. Along the z-dimension, red/black plane relaxation is used; along the y-dimension, red/black line relaxation is used.

At each coarsening level, stencil computations are performed as part of red/black relaxation as well as to compute residuals as the basis for computation at the next coarsening level. Only "nearest neighbor" stencils, are employed. The red/black relaxation steps use stencils of sizes 1, 2, 3, 5, 6 and 10 points. The residual calculations performed during coarsening use stencils of 5, 7, 9, and 15 points.

The *hypre* implementation of semicoarsening multigrid is written in ISO-C. It is an SPMD code based on the MPI [17] message passing library. Parallelism is achieved by data decomposition. Each processor is assigned a vertical slice of the hierarchy of grids. The implementation is memory intensive. For 3D problems, the memory it requires is roughly 54 times the local problem size plus some miscellaneous overhead (e.g. for storing ghost points). All values in the hierarchy of grids are represented as double precision quantities. The space overhead for ghost-zones relative to regular grid points is quite high on a single processor, but grows roughly logarithmically as the number of processors increases [13].

3. THE *HYPRE* LIBRARY

The *hypre* library was developed to facilitate solution of large sparse linear systems on parallel computers [6]. It provides abstractions of multi-dimensional Cartesian grids, grid hierarchies, and iterators, along with support for domain decomposition of these grids among the processors in a parallel machine. The primary goal of *hypre* is to provide users with modern powerful and scalable preconditioners. The design of *hypre* enables it to be used as both a solver package and a framework for algorithm development.

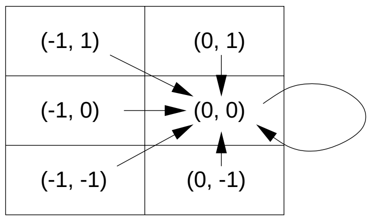
The *hypre* library provides several families of preconditioner algorithms. It includes "grey-box" algorithms that use more than just matrices to solve certain classes of problems more efficiently than general-purpose libraries. An example of this is structured multigrid. The *hypre* library also provides several of the most commonly used Krylov-based iterative methods to be used in conjunction with its scalable preconditioners. It includes methods for nonsymmetric systems such as GMRES and methods for symmetric matrices such as conjugate gradient. *hypre* supports multiple interfaces including stencil-based structured and semi-structured interfaces, a finite element based unstructured interface, and a linear algebra based interface. Through these grid-centric interfaces, users do not have to use complicated data structures to access advanced solvers. *hypre* is designed as an object-oriented library, although many of the abstractions provided by internal interfaces are implemented using macros rather than procedures for flexibility and efficiency. Central to *hypre*'s design is the use of interfaces to enable algorithm developers to mix and match solvers, data layouts, and linear system types. *hypre* can be used from both Fortran and C.

4. OPTIMIZING STENCIL COMPUTATION

A performance analysis of SMG98 using a single-processor benchmark on two different architectures (see Tables 1–2 in Section 7) showed that it spent 40–50% of its execution time calculating residuals using stencil-based computation. Such stencil computations often account for a significant fraction of execution time in scientific applications that solve partial differential equations over rectilinear domains. Stencil computations involve updating a value associated with each element in a data domain as a function of data values associated with a set of elements in its neighborhood.

On a rectilinear domain, a stencil can be described as a set of offsets relative to a target element. Each offset indicates an element in the target's neighborhood that will contribute to its update. An n-element stencil S applied to an m-dimensional data domain can be described as a set of n offsets $o_1, o_2, ..., o_n$, with each offset o_i, $1 \leq i \leq n$, consisting of an m-vector. Each position $o_{i[j]}$, $1 \leq j \leq m$, in m-vector o_i represents the offset in dimension j with respect to the target element. Figure 3 shows a 6-point stencil in two dimensions. At the bottom of the figure is the representation of the stencil as a vector of offsets. The picture shows each offset represented as a cell in its position relative to the target cell at coordinates (0,0). The arrows depict each cell in the stencil contributing to the update of the target cell.

Stencil computations have long been recognized as opportunities for optimization. Bromley et. al. [3] built a special purpose compiler to compile stencil computations for Thinking Machine's Connection Machine 2 (CM-2). This tool would transform a stencil computation written using Fortran 90 array syntax into a control template that specifies what register reference patterns should be used to parameterize a library of hand-crafted microcode routines. To improve on-processor performance, they constructed "multistencils," which compute a result for more than one target element at a time. Multistencils are constructed so that they fit entirely into registers. They are formed by applying a combination of strip-mining, unrolling and scalar replacement to the original stencil code. Multistencils exploit the overlap of adjacent stencils to reduce the number of redundant loads. A variety of other techniques were also employed by the stencil compiler to cope with idiosyncrasies of the CM-2 microsequencer.



S = {(-1,1),(-1,0),(-1,-1),(0,1),(0,0),(0,-1)}

Figure 3: A 2D 6-point stencil.

```
for all boxes $B$ on grid $G$
    for each stencil offset $o_p (p = 1, 2, ..., n)$
        Let $A$ be the coefficient matrix for stencil $o_p$
        for $k = klb_B, kub_B, kstep$
            for $j = jlb_B, jub_B, jstep$
                for $i = ilb_B, iub_B, istep$
                    R$(k, j, i)$ = R$(k, j, i)$ - A$(k, j, i)$ *
                                X$(k + o_{p[3]}, j + o_{p[2]}, i + o_{p[1]})$
```

Figure 4: A sketch of SMG98's stencil-based residual computation.

Unlike the "static stencils" compiled by the CM-2 stencil compiler, the residual calculation in SMG98, performed by routine `hypre_SMGResidual`, uses the same code to apply an arbitrary stencil unknown to it until the routine is invoked. As described in Section 2, SMG98 uses a variety of stencils in the course of its computation. To accommodate application of arbitrary stencils, `hypre_SMGResidual` applies a stencil to a 3D domain using the approach sketched in the pseudo code shown in Figure 4. For each stencil element, the code iterates over the entire data domain and computes its contribution to the residual R. The flexibility of being able to use this code to apply an arbitrary stencil comes at a cost in performance. In Figure 4, the statement in the innermost loop contains three loads and a store for each multiply-accumulate operation. There is no reuse of floating point values in the registers. The ratio of memory accesses per floating point operation that a processor can perform (the machine balance) imposes an upper limit on the loop's performance. On an MIPS R12000 the machine balance is

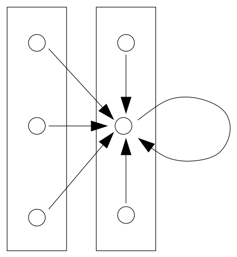


Figure 5: Factoring a 2D 6-point stencil into stride-adjacent triplets.

one, whereas the machine balance on a Compaq Alpha EV67 is two. Since this loop nest performs a sweep over the entire data domain for each stencil element it applies, when the size of the domain is sufficiently large, all of the data accessed while processing one stencil element will be evicted before it can be reused when processing other stencil elements.

The key to improving `hypre_SMGResidual`'s performance is to exploit temporal reuse of data. We introduce *run-time stencil factoring* as a technique for boosting temporal data reuse and efficiency when applying stencils unknown at library development time. Stencil factoring aims to exploit three key types of reuse. First, performing the residual computation for multiple stencil elements in a single instance of the inner loop reduces the number of times the computation will have to sweep over the data and stream it through the memory hierarchy. Second, when a group of stencil elements are applied together, only one load and store of each $R(k, j, i)$ point is needed for the group of stencil elements rather than a load and store of $R(k, j, i)$ for each element. Third, whenever there are stencil elements that are *stride adjacent*, namely, their offsets differ only in the leftmost position and there by *istep* (the stride of the inner loop), then we can achieve register reuse of elements of X across adjacent loop iterations. We next explain our strategy for achieving register reuse for stride-adjacent stencil elements more precisely.

A pair of stencil elements with offsets o_i and o_j in an m-dimensional data domain are stride adjacent if $|o_{i[1]} - o_{j[1]}| = istep$ and $o_{i[k]} = o_{j[k]}$ for $k = 2, 3, ..., m$. Figure 5 shows a factoring of a 6-point stencil in two dimensions. Stencil elements are grouped into two stride-adjacent strips as shown by the boxes around stencil element triplets.

When processing a stride-adjacent strip from a stencil, the value loaded for the leading edge of the strip can be retained in a register and reused in subsequent iterations for each position in the strip. At the source code level, we can arrange to achieve reuse of X in the registers by using a technique known as *scalar replacement* [5], which involves computing with array values stored temporarily in scalar variables. A compiler's register allocator typically will assign a register as the storage location for a scalar temporary variable.

An analysis of the stencils used by SMG98 showed that the bulk of its stencil-based computation involved stencils that could be factored into stride-adjacent triplets. Figure 6 shows a skeletal version of the code we developed to improve the performance of these stride-adjacent triplets for the case when the stride of the inner loop is one. Statement nesting in control constructs is indicated by indenting. This code

```
// rp is the residual array
// xp is the data array
// Ap1, Ap2, and Ap3 are coefficient arrays
// Ai, xi, and ri have initial values
// xioff = xi - Ai, rioff = ri - Ai
for (loopk = 0; loopk < hypre__nz; loopk++ )
   for (loopj = 0; loopj < hypre__ny; loopj++ )
      // fill x3 and x4 before the first iteration
      xi = Ai + xioff; ri = Ai + rioff;
      x3 = xp[xi- 2];
      x4 = xp[xi - 1];
      for (loopi = 0; loopi < hypre__nx - 1; loopi+=2 )
         xi = Ai + xioff; ri = Ai + rioff;
         x1 = x3; x2 = x4; x3 = xp[xi]; x4 = xp[xi + 1];
         tmp = rp[ri];        tmp2 = rp[ri + 1];
         tmp -= Ap1[Ai] * x1; tmp2 -= Ap3[Ai + 1] * x4;
         tmp -= Ap2[Ai] * x2; tmp2 -= Ap1[Ai + 1] * x2;
         tmp -= Ap3[Ai] * x3; tmp2 -= Ap2[Ai + 1] * x3;
         rp[ri] = tmp;        rp[ri + 1] = tmp2;
         Ai += 2;
      if (loopi == hypre__nx - 1)
         // handle a trailing singleton iteration
         xi = Ai + xioff; ri = Ai + rioff;
         x1 = x3; x2 = x4; x3 = xp[xi];
         tmp = rp[ri];
         tmp -= Ap1[Ai] * x1;
         tmp -= Ap2[Ai] * x2;
         tmp -= Ap3[Ai] * x3;
         rp[ri] = tmp;
         Ai += 1;
      Ai += hypre__jinc1;
   Ai += hypre__kinc1;
```

Figure 6: Skeletal code fragment for applying stride-adjacent stencil element triplets.

updates a residual value based on a stride-1 triplet. The inner loop is unrolled to interleave the update of two residual values to reduce the exposed floating point pipeline latency that arises in the computation because the three multiply-accumulates for a single residual are involved in a recurrence (the intermediate value produced by each multiply-accumulate is an input to the next). Each iteration of the loop loads two values of X, six distinct coefficient values and updates two residual points. In the code shown in Figure 6, the ratio of memory accesses to multiply-accumulate instructions is two-to-one, considerably better than the four-to-one ratio of the original code shown in Figure 4.

To increase the performance of stencil computations, we wrote a run-time library routine to factor stencils into pieces that could be processed efficiently by `hypre_SMGResidual` either as stride-adjacent groups of stencil elements and or as groups of random stencil elements. To factor a stencil, we sort stencil elements lexicographically by their stencil offset vectors. Next, we identify stride-adjacent groups of stencil elements and mark them to be processed in groups of 2 or 3 elements by code similar to that shown in Figure 6. These stride-adjacent groups are most efficient because they improve the loop balance of the residual calculation by reusing values in the registers. Finally, we collect remaining stencil elements into a group that will be processed by the residual code in groups of 6, 4, 2, or 1 elements at a time, with a preference for the largest group possible. (Experimentally, we determined that using groups of more than 6 stencil elements induced register spilling, which made larger groups inefficient.)

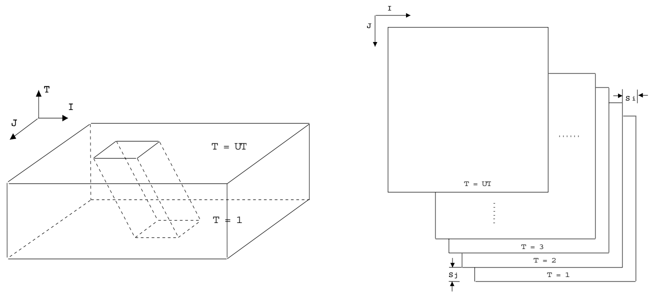
```
for (t = 1; t <= nstep; t++)
  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        a[i][j] = c1 * a[i][j]
                + c2 * (a[i-1][j] + a[i+1][j] +
                        a[i][j-1] + a[i][j+1])
```

(a)

```
for (iout = 1; iout <= n; iout += blkI)
  for (jout = 1; jout <= n; jout += blkJ)
    for (t = 1; t <= nstep; t++)
      for (i = max(1, iout-t+1;
           i <= min(n, iout+blkI-t); i++)
        for (j = max(1, jout-t+1;
             j <= min(n, jout+blkJ-t); j++)
          a[i][j] = c1 * a[i][j]
                  + c2 * (a[i-1][j] + a[i+1][j] +
                          a[i][j-1] + a[i][j+1])
```

(b)

Figure 7: An example.



(a) (b)

Figure 8: 2D temporal skewing.

5. GLOBAL TEMPORAL SKEWING AND BLOCKING

Iterative methods perform multiple sweeps over a data domain. Ideally, between sweeps data can be kept in caches to avoid the delay that occurs when data is fetched out of main memory. For large problems, however, the data for the entire domain will not fit in cache and thus there can be no data reuse between a pair of separate sweeps. Previously, we proposed a compiler strategy that combines loop skewing with recursive blocking to enhance data reuse in iterative computations [14]. This strategy interleaves iterations to bring uses of the same data elements closer together in time. In this section, we describe how we apply temporal skewing and blocking globally to multiple sweeps of *hypre*'s semicoarsening multigrid solver and address issues of efficient computation of hypre box information.

In general, temporal skewing and blocking carves an iteration space over a data domain into a set of skewed time-space prisms in which one or more spatial dimensions is skewed by the index of a time loop [18, 20].

As an example, Figure 7 illustrates a straightforward 2D Gauss-Seidel implementation and an optimized version that was created by applying temporal skewing and blocking to the original. In the original code shown in Figure 7(a), each time step of t completes a full sweep over the spatial domain. Each element of the array a is updated based on its value at the previous time step and the values of its four neighbors, two from the previous time step and two from the current time step. In the optimized code shown in Figure 7(b), the inner loops t, i, and j sweep through a time-space prism and the outer loops enumerate prisms that cover the original iteration space. To preserve the data dependences carried by t, the i and j loops are skewed with respect to t. Blocking factors *blkI* and *blkJ* should be chosen so that each time-space prism is small enough to fit into the primary cache.

Figure 8(a) shows the rectangular solid time-space iteration volume for the code shown in Figure 7 and a prismatic tile within that time-space volume in which each spatial dimension has been skewed by the temporal dimension as in the optimized code shown in Figure 7. All prismatic tiles of the time-space iteration volume are skewed in the same way. Boundary tiles are clipped to prevent them from extending outside of the iteration space. A temporal skewing can be represented as a triple (t, $\mathbf{L}$, $\vec{s}$), where t is the time loop, $\mathbf{L}$ is a set of spatial loops to be skewed with respect to t, and $\vec{s}$ is a vector of skew factors for the loops in $\mathbf{L}$. Based on the data dependences carried by the time loop, temporal skew factors are chosen so that a skewed portion of the domain iteration space for time t can safely execute after a portion of the iteration space finishes execution for time $t-1$. Figure 8(b) shows a temporal skewing with skew factor vector $\vec{s} = (s_i, s_j)$. Suppose $[lb_i : ub_i, lb_j : ub_j]$ is the spatial cross section of the prism evaluated at time step t. At time $t+1$, the spatial cross section executed will be $[lb_i - s_i : ub_i - s_i, lb_j - s_j : ub_j - s_j]$. This process continues until it reaches the face of the prism. Boundary prisms are skewed in the same way, except that they are clipped by lower and upper bounds of the spatial loops.

Applying temporal skewing to the semicoarsening multigrid solver in *hypre* is not as straightforward as to a simple kernal for three reasons. First, it includes a complete solve with a setup phase and convergence test, and it uses recursion to perform a solve, coarsening grids along only one dimension at a time. Second, the algorithm employs multiple sweeps over different level of grids and the implementations of these sweeps is spread across several routines. Third, the code is based on an object-oriented library that was developed for a broad range of targets. Because of this, we need not only to determine how to apply the temporal skewing to achieve the most data reuse, but also how implement these transformations without destroying the structure of the object-oriented *hypre* library. We apply temporal skewing to the residual computation and cyclic reduction at the level where grids are coarsening along the j-dimension. This enables us exploit temporal data reuse at three levels: across multiple pre- and post- relaxations, across red and black sweeps and across multiple phases within a V-cycle.

In each relaxation step, SMG98 performs a red line relaxation followed by a black line relaxation. Each line sweep computes residuals and calls a direct solver. Figure 9 shows a simplified description of the original red/black line sweeps, where r is the relaxation loop and s represents the red/black sweep loop. Both the residual calculation and cyclic reduction consist of multiple phases. The residual calculation has a copy phase followed by a computation phase. The cyclic reduction calculation includes a full V-cycle with fine-grid relaxation, restriction, and interpolation phases. The code

```
for (r = 0; r < max_iter; r++)
   for (s = 0; s < num_spaces; s++)
       residual(p1, ..., pm)
       cyclic_reduction(q1, ..., qn)
```

Figure 9: Red/black line sweeps in SMG98.

```
for (kbox = 0; kbox <= bz-1; kbox++)
  for (jbox = 0; jbox <= by-1; jbox++)
    for (nextZ = 0, kLeft = kLen; kLeft > 0;
         kLeft -= blkK, nextZ += blkK)
      for (nextY = 0, jLeft = jLen; jLeft > 0;
           jLeft -= blkJ, nextY += blkJ)
        for (r = 0; r < max_iter; r++)
          for (s = 0; s < num_spaces; s++)
              is = space_ranks[s];
              myNextY = nextY - r - is * s;
              residual(p1, ..., pm, jbox, kbox, jLeft,
                  myNextY, kLeft, nextZ, r, s, is)
              cyclic_reduction(q1, ..., qn, jbox, kbox,
                  jLeft, myNextY, kLeft, nextZ, r, s, is)
```

Figure 10: Transformed red/black line sweeps in SMG98.

after applying temporal skewing and blocking is shown in Figure 10, where `nextY` and `nextZ` are used to march the tiles and `myNextY` indicates where the tile for the current relaxation step and red/black sweep should start. After the transformation, the residual and cyclic reduction computations operate on only a portion of the data domain. Figure 11 shows a code fragment for a tiled version of the original simple residual computation. Loop sizes for the j and k dimensions are set to `blkJ` and `blkK`. Their lower bounds are adjusted accordingly. After the residual computation, the loop sizes and lower bounds are reset to the original values. For simplicity, we only present a code skeleton for a central tile. Code for boundary tiles is more complicated. Without periodic boundary conditions, the upper and lower bounds of tiles are clipped at the edges of the data domain. Tiles along box boundaries include not only computation for the current box, but also computation for neighboring boxes as well. To exploit temporal data reuse across the relaxation steps and red/black line sweeps, the size of prismatic tiles must be carefully chosen so that they can fit into cache.

While temporal skewing and blocking can improve cache performance in SMG98 by increasing data reuse across sweeps, these techniques can introduce substantial overhead if they are not carefully applied to the code which manipulates hypre boxes. In the original program, box information is computed in each of the phases directly before each sweep. Blocking increases the number of boxes and proportionally increases the overhead of recomputing this information. To amortize the overhead of this computation, we precompute box information once before outside the tile loop and reuse box information for tiles in each of the sweeps in the tile loop. By doing this, we are able to minimize the overhead of calculating box information for tiles.

To date, we have implemented temporal skewing and blocking within a single processor. Additional changes need to be made to the code so that the temporal skewing can be applied when the data domain is partitioned among processors.

```
// set start and loop_size
hypre_IndexY(loop_size) = blkJ;
hypre_IndexZ(loop_size) = blkK;
hypre_IndexY(start) += myNextY*hypre_IndexY(base_stride);
hypre_IndexZ(start) += nextZ*hypre_IndexZ(base_stride);
for (loopk = 0; loopk < hypre__nz; loopk++ )
    for (loopj = 0; loopj < hypre__ny; loopj++ )
        for (loopi = 0; loopi < hypre__nx; loopi++ )
            rp[ri] -= Ap[Ai] * xp[xi];
            Ai += iinc1; xi += iinc1; ri += iinc1;
        Ai += jinc1; xi += jinc2; ri += jinc3;
    Ai += kinc1; xi += kinc2; ri += kinc3;
// restore start and loop_size;
```

Figure 11: Code fragment of a transformed internal sweep.

6. DATA EXCHANGE BLOCKING

A hypre box is one of the key building blocks in both the *hypre* library and SMG98. Hypre boxes are used to represent computation, data, and communication sets. While data boxes can be divided among multiple processors to achieve parallelism, the use of multiple data boxes within each processor can also potentially improve data locality. Each box has its own ghost regions for saving data received from neighboring boxes, which may or may not reside on the same processor. Carefully choosing the size of ghost regions and extending them to pad array dimensions (as needed) can also reduce cache conflict misses [19].

In SMG98, local data exchange is handled the same way as data exchange between processors. Data that needs to be exchanged within a processor or between processors are represented as *send* and *recv* sets that are organized as a set of boxes. When updated partial solutions are needed, a global pass of communication is inserted to finish all of the data exchanges between neighboring boxes within and across processors. Figure 12(a) shows two dimensions of a data box $\times$ and its data exchanges with neighboring boxes. To compute data values in each of the gray regions requires data from a neighboring box in either the east, south, west, or north. To compute data values in each of the black regions also requires data from one of its other four diagonal neighbors. Applying temporal skewing and blocking without adjusting the communication would cause the box boundaries to be communicated in their entirety for each tile. In the transformed code, we block the iteration space as well as the data exchange between neighboring boxes, and transfer only the subset of data needed for performing the computation on the current tile. Figure 12(b) shows the original data box $\times$ blocked into three subboxes $\times 1$, $\times 2$, $\times 3$. The grey area shows the data exchange necessary for completing the computation of the central subbox. $myNextY$ and $myNextY + blkJ - 1$ represent the lower and upper bounds of the blocked data exchange along the j-direction. It is important to note that which data elements need to be exchanged is determined by the shape of the stencil that will be applied when performing the tile's computation. The exchanged data must include all elements from neighboring boxes around the tile boundary that are needed to apply the stencil to any point within the tile's boundary.

To date, we have implemented this data exchange blocking strategy for a domain consisting of multiple boxes on a single processor. When the domain is partitioned among multiple

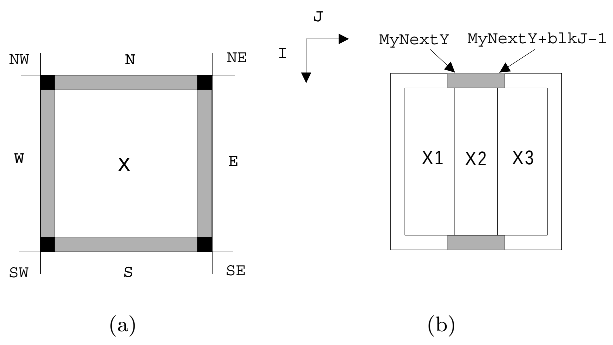


Figure 12: Local data exchange around boundaries of a 2D hypre box.

processors, the processors' computation and data exchange have to be coordinated to ensure that each processor can obtain the updated values it needs from neighbors at the proper time. Elaboration of this strategy is future work.

7. EXPERIMENTAL RESULTS

Experiments with SMG98 were performed on a single processor of a Compaq ES40 or an SGI Origin 2000. Our Compaq ES40 contains four 667MHz Compaq Alpha EV67 (21264A) processors. An EV67 processor can issue up to four instructions per cycle, which may include at most two memory accesses and one floating-point multiply-accumulate. Floating-point arithmetic operations have a 4-cycle latency. The ES40's memory hierarchy consists of a 64KB 2-way set associative primary cache and an 8-MB Dual Data Rate cache. Primary cache latency is three cycles for integer loads and four cycles for floating-point loads, while the latencies to the secondary cache and memory are 12 and 80 cycles, respectively. Our SGI Origin 2000 contains 16 300MHz MIPS R12000 processors, each with a 32KB 2-way set associative cache. Each node in the system contains a pair of processors, which share an 8MB unified secondary cache. Each R12000 can issue four instructions per cycle, which may include at most one memory access and one floating-point multiply-accumulate. Latency penalties for L1 and L2 misses are approximately 8 and 100 cycles each.

We hand-transformed the SMG98 semicoarsening multigrid code to incorporate the optimizations described in Sections 4–6. Performance results were measured with SpeedShop on a MIPS R12000 and with DCPI on an Alpha EV67. Execution time, retire delay, and cache misses were collected separately using hardware performance counters. Programs were compiled with MIPSpro C V7.3.1.1 using optimizations "-O3 -mips4 -r12000" on the SGI Origin and with Compaq C V6.3 with optimizations "-O3 -arch ev67" on the Compaq ES40. We measured performance for two problem sizes, 64^3 and 128^3, with coefficients 2.0, 3.0 and 40.0. For both cases, each V-cycle of the multigrid performs two sweeps of pre-relaxations and post-relaxations.

7.1 Results and Discussion

Tables 1 and 2 present the sequential performance of the original SMG98 semicoarsening multigrid code that served as the starting point for our study. The tables show the breakdown of execution time, floating point computation and memory hierarchy utilization on the two platforms for four major routines[1] in the program. The first row of Table 1 shows the total number of L1 cache misses, L2 cache misses, graduated floating-point instructions, and CPU cycles on a MIPS R12000, while the first row of Table 2 shows the total number of board cache misses, retire delay, and cycles on Alpha EV67. The remaining rows of both tables show the relative percentage of each total that is attributed to each of the four main routines of the multigrid code.

The tables show that about 90% of the total execution time is spent in two routines which are `hypre_SMGResidual` and `hypre_CyclicReduction`. The two remaining routines, `hypre_SMGIntAdd`, which incorporates error corrections from coarser grids into finer grids, and `hypre_SMGRestrict`, which transfers residuals from finer to coarser grids, account for another 4–6% of the total execution time. A setup phase (not shown in the table) accounts for most of the remaining time. On the MIPS R12000, we observed 1.09 and 9.03 billion primary cache misses for the small and large problem sizes (about four misses for every five graduated floating-point operations); of these, 4.2% and 12.7% caused further secondary cache misses for the small and large problem sizes, respectively. Over 90% of the cache misses occurred in `hypre_SMGResidual` and `hypre_CyclicReduction`. As a result, the number of cycles per floating-point operation is as high as 24 on the MIPS R12000; results on the Alpha EV67 are similar. Ineffective cache utilization causes high retire delay which accounts for about 95% of the total cycles.

Table 3 shows the performance metrics for the residual and cyclic reduction computations broken down for each of the major phases in these computations to help us better understand the factors contributing to their run-time cost. The cost for residual computation is broken down into a copy phase and a stencil computation phase. The cost for cyclic reduction is separated into each of its five phases: a copy phase, two down-cycle phases, and two up-cycle phases. The similar cache miss statistics for each of the phases of cyclic reduction shows that there is not substantial reuse between phases; otherwise, the later phases would have less misses than the first.

The residual computation of `hypre_SMGResidual` accounts for 44% of the total floating point operations in the program, and in the original code, this computation accounts for up to about 50% of the execution time on both processors. Overall, the original residual computation requires about 15 cycles per FLOP for the small problem size on MIPS R12000, yielding 20 MFLOPS. On EV67, it requires about 14 cycles per FLOP for the same problem, yielding 48 MFLOPS. Performance for the large problem size on both architectures is worse because of severe secondary cache misses. By appling stencil factoring to `hypre_SMGResidual`, we were able to reduce its primary cache misses by over 36% on MIPS R12000 and secondary cache miss by over 21% for large problem size on both processors as shown in Table 4. The overall performance of the optimized residual computation using stencil factoring is 48% and 89% faster on MIPS R12000 and EV67 for the small problem size, yielding 30 MFLOPS and 91 MFLOPS. The optimized kernel that updates a residual point for any three stencil-element strip achieves 155

[1]We use shorter names *residual*, *cyc_red*, *intadd*, and *restrict* in tables to represent routines `hypre_SMGResidual`, `hypre_CyclicReduction`, `hypre_SMGIntAdd`, and `hypre _SMGRestrict`.

Code	*64 × 64 × 64*				*128 × 128 × 128*			
	L1 misses	*L2 misses*	*Grad. fp*	*Cycles*	*L1 misses*	*L2 misses*	*Grad. fp*	*Cycles*
total	1093M	46M	1.4B	21B	9028M	1143M	11B	268B
residual	48.7%	46.2%	44.2%	42.8%	47.7%	52.0%	44.2%	49.6%
cyc_red	43.8%	36.6%	45.6%	44.8%	45.5%	38.2%	45.8%	41.8%
intadd	2.9%	4.1%	4.5%	3.5%	2.9%	3.8%	4.5%	3.4%
restrict	1.5%	5.1%	2.7%	2.6%	1.5%	2.3%	2.7%	1.6%

Table 1: Cache misses(in millions), graduated floating-point instructions(in billions), and CPU cycles(in billions) of the original code on an SGI Origin 2000 (MIPS R12000 processor).

Code	*64 × 64 × 64*			*128 × 128 × 128*		
	Bmisses	*RetDelay*	*Cycles*	*Bmisses*	*RetDelay*	*Cycles*
total	104M	18B	19B	2277M	214B	225B
residual	44.3%	50.1%	47.8%	53.9%	52.7%	51.1%
cyc_red	40.7%	43.5%	44.6%	36.3%	42.4%	43.6%
intadd	3.3%	2.3%	2.5%	3.7%	2.4%	2.6%
restrict	4.7%	1.1%	1.2%	2.2%	1.1%	1.1%

Table 2: Board cache misses(in millions), retire delay(in billions), and CPU cycles(in billions) of the original code on a Compaq ES40 (Alpha EV67 processor).

Code	*L1 misses %*		*L2 misses %*		*Grad. fp %*		*Cycles %*	
	64^3	128^3	64^3	128^3	64^3	128^3	64^3	128^3
residual	48.7	47.7	46.2	52.0	44.2	44.2	42.8	49.6
copy phase	6.0	5.8	15.9	10.2	0.0	0.0	8.6	7.5
stencil	42.6	41.9	30.3	41.7	44.2	44.2	34.0	42.1
cyc_red	43.8	45.5	36.6	38.1	45.6	45.8	44.8	41.8
copy phase	4.3	4.2	1.5	4.2	0.0	0.0	2.6	4.0
down:relax	8.6	8.6	13.3	9.2	6.7	6.6	13.5	9.4
down:restr	10.9	10.8	20.4	12.9	12.9	13.1	14.1	10.8
up:inter	6.2	6.4	0.2	3.6	0.0	0.0	2.9	4.3
up:relax	13.5	15.6	0.9	8.2	25.9	26.1	11.2	13.2

Table 3: Cache misses and execution time breakdown of the original code on an SGI Origin 2000 (MIPS R12000 processor).

MFLOPS, or 4.3 cycles per FLOP on EV67. Speedups for the large problem size are 1.4 and 2.13 on MIPS R12000 and EV67 as shown in Table 4, where *orig* represents the original code and *opt* represents the optimized code. The overall performance is lower because this kernel for handling stride-adjacent triplets only applies to 72% of FLOPS in the residual stencil computation. The remaining 28% of the FLOPS are for stencil elements that are not members of any stride-adjacent strip. We apply such stencil elements in groups to save on the load/stores of the residual values. Most of the computation is handled for pairs of stencil elements, though about a quarter is handled four-elements at a time or six elements at a time (the largest group before register resources were exceeded on the MIPS R12000).

To improve temporal data reuse across multiple sweeps, we applied the global temporal skewing to `hypre_SMGResidual` and `hypre_CyclicReduction` at the level where grids are coarsening along the j-dimension. We applied a 2D temporal skewing along the j- and k-dimensions, while keep the i-dimension unchanged to preserve the original semantics. We chose a 16×1 blocking for j- and k-dimensions. Table 5 compares the performance of the code optimized using global time skewing against the original code. On the MIPS R12000, global time skewing reduced the total number of primary cache misses by 23% and 18%, and total number of secondary cache misses by 9% and 46% for the two problem sizes. The significant reduction of secondary cache misses of both `hypre_SMGResidual` and `hypre_CyclicReduction` for the large problem size comes from increased temporal reuse across the two pre- and post-relaxations as well as multiple phases in the V cycle. Improved cache utilization yields an overall performance speedup of 1.75. On Alpha EV67, secondary cache misses is reduced by about 15% and 45% for the small and large problem sizes, leading to an overall performance improvement of 12% and 44%.

Although dynamic stencil factoring and global time skewing share the same goal of increasing temporal data reuse, their performance benefits largely come from improving different aspects of the multigrid computation. Stencil factoring increases immediate temporal reuse in the registers for the residual computation, while global time skewing increases temporal cache reuse of both the residual computation and the direct solver by reusing data across multiple global sweeps. The combination of these two optimizations yields better performance than either one alone as shown in Table 6. The optimized code achieves over 30% reduction

Code	*Origin 2000*							*Alpha 21264a*						
	L1 misses M		*L2 misses* M		*Cycles* B			*Bmisses* M		*RetDelay* B		*Cycles* B		
	orig	*opt*	*orig*	*opt*	*orig*	*opt*	*spd*	*orig*	*opt*	*orig*	*opt*	*orig*	*opt*	*spd*
	64 × 64 × 64													
residual	532	338	21	21	9.0	6.1	1.48	46	48	8.8	4.5	8.9	4.7	1.89
total	1093	904	46	48	21	18	1.17	104	112	18	14	19	15	1.27
	128 × 128 × 128													
residual	4306	2681	594	469	133	95	1.40	1226	937	113	51	115	54	2.13
total	9028	7209	1143	1039	268	238	1.13	2277	2074	214	143	225	155	1.45

Table 4: Performance comparison before and after applying stencil factoring.

Code	*Origin 2000*							*Alpha 21264a*						
	L1 misses M		*L2 misses* M		*Cycles* B			*Bmisses* M		*RetDelay* B		*Cycles* B		
	orig	*opt*	*orig*	*opt*	*orig*	*opt*	*spd*	*orig*	*opt*	*orig*	*opt*	*orig*	*opt*	*spd*
	64 × 64 × 64													
residual	532	430	21	20	9.0	6.3	1.43	46	43	8.8	8.1	8.9	8.5	1.05
cyc_red	479	314	17	14	9.4	7.5	1.25	42	29	7.6	6.7	8.3	6.7	1.24
total	1093	839	46	42	21	16	1.31	104	88	18	16	19	17	1.12
	128 × 128 × 128													
residual	4306	3872	594	375	133	74	1.80	1226	771	113	85	115	86	1.34
cyc_red	4110	2823	435	131	112	58	1.93	826	276	91	52	98	54	1.81
total	9028	7391	1143	622	268	153	1.75	2277	1251	214	150	225	156	1.44

Table 5: Performance comparison before and after applying global temporal skewing and blocking.

Code	*Origin 2000*							*Alpha 21264a*						
	L1 misses M		*L2 misses* M		*Cycles* B			*Bmisses* M		*RetDelay* B		*Cycles* B		
	orig	*opt*	*orig*	*opt*	*orig*	*opt*	*spd*	*orig*	*opt*	*orig*	*opt*	*orig*	*opt*	*spd*
	64 × 64 × 64													
residual	532	314	21	19	9.0	5.0	1.80	46	39	8.8	4.3	8.9	4.5	1.98
cyc_red	479	317	17	13	9.4	7.9	1.19	42	28	7.6	6.3	8.3	6.7	1.24
total	1093	730	46	41	21	16	1.31	104	84	18	12	19	13	1.46
	128 × 128 × 128													
residual	4306	2654	594	279	133	51	2.61	1226	562	113	41	115	43	2.67
cyc_red	4110	2829	435	131	112	58	1.93	826	279	91	51	98	54	1.81
total	9028	6197	1143	527	268	132	2.03	2277	1076	214	106	225	114	1.97

Table 6: Performance comparison before and after applying stencil factoring and global temporal skewing and blocking.

of primary cache misses of the whole application and up to 54% reduction of secondary cache misses for the large problem size. The overall performance improvement is about a factor of 2 for each architecture.

We also evaluated the optimizations in a multiple hypre box case where there are 2 × 2 × 2 boxes and size of each box is 64^3. Preliminary results on both processors are shown in Table 7. The optimized code using stencil factoring and the global time skewing has about 27% less primary cache misses on MIPS R12000 and 51% and 46% less secondary cache misses on MIPS R12000 and Alpha EV67. Overall speedups of the optimized code against the original code are 1.57 and 1.62 on two processors. Compared with the single hypre box cases, we believe that the less significant speedup comes from less efficient box boundary copying and extra cost for computing box information.

8. CONCLUSIONS

We have proposed and demonstrated the effectiveness of several strategies for improving the single-processor performance of SMG98 by improving the memory hierarchy utilization of *hypre*'s semicoarsening multigrid solver. Overall, these optimizations yield improvements ranging from 31% to about a factor of two. While the stencil factoring optimization for the residual computation is applicable to both sequential and parallel executions of the code, at present the temporal skewing techniques we applied can only be used in a single-processor execution of the code. The temporal skewing required significant adjustments to the code. Extending the time skewing technique effectively for multiple processors requires proper reordering of computation and data exchanges between neighboring boxes within a processor and across processors. It is a subject of our current work.

The structure of the internal interfaces of the *hypre* library was in many cases not well-suited to accommodate our optimizations. The abstractions in *hypre* were designed for ease of use, but their interfaces, defined by a collection of macros for manipulating hypre boxes, did not provide the flexibility necessary to implement these optimizations without reworking the abstractions or not using them at all. Our experiences in tuning *hypre*'s semicoarsening multigrid solver show that while *hypre*'s high-level abstractions such

Code	*Origin 2000*							*Alpha 21264a*						
	L1 misses M		*L2 misses* M		*Cycles* B			*Bmisses* M		*RetDelay* B		*Cycles* B		
	orig	*opt*	*orig*	*opt*	*orig*	*opt*	*spd*	*orig*	*opt*	*orig*	*opt*	*orig*	*opt*	*spd*
	64 × 64 × 64 × 2 × 2 × 2													
residual	4322	2611	462	255	119	66	1.80	918	542	104	41	108	45	2.40
cyc_red	4145	3265	499	137	128	75	1.71	928	320	91	63	96	68	1.41
total	9356	6791	1098	533	280	178	1.57	2115	1134	211	124	224	138	1.62

Table 7: Performance comparison before and after applying stencil factoring and global sweep optimizations for multiple boxes on a single processor.

as a macro supporting iteration over the data in a hyprebox can be useful for rapid prototyping, they only get in the way when trying to optimize performance by operating on a patch of data elements at once such as the stencil factoring did. Studying the ways in which *hypre*'s abstractions failed to support our optimizations could lead to improvements in *hypre*'s interfaces to make it easier to use it as a building block for high performance codes.

9. ACKNOWLEDGMENTS

This research was supported by Department of Energy's Accelerated Strategic Computing Initiative under research subcontract B347884 and the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23 as part of the prime contract (W-7405-ENG-36) between the Department of Energy and the Regents of the University of California. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

We thank Rob Falgout for his comments on this work.

10. REFERENCES

[1] A. Behie and P. A. Forsyth, Jr. Multigrid solution of three-dimensional problems with discontinuous coefficients. *Appl. Math. Comput*, 13:366–386, 1983.

[2] A. Brandt. Multilevel computations: Review and recent developments. In S. F. McCormick, editor, *Multigrid Methods: Theory, Applications, and Supercomputing*, pages 35–62. Marcel Dekker, New York, 1988.

[3] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[4] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM J. Sci. Comput*, 21(5):1823–1834, 1999.

[5] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[6] E. Chow, A. J. Cleary, and R. D. Falgout. Design of the hypre preconditioner library. In M. Henderson, C. Anderson, and S. Lyons, editors, *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, Yorktown Heights, Oct. 1998.

[7] J. E. Dendy, Jr. Black box multigrid. *J. Comput. Phys.*, 48:366–386, 1982.

[8] J. E. Dendy, Jr. Black box multigrid for nonsymmetric problems. *Appl. Math. Comput*, 13:261–283, 1982.

[9] J. E. Dendy, Jr. Black box multigrid for systems. *Appl. Math. Comput*, 19:57–74, 1986.

[10] J. E. Dendy, Jr. Two multigrid methods for three-dimensional problems with discontinuous and anisotropic coefficients. *SIAM J. Sci. Stat. Comput*, 8:673–685, 1987.

[11] J. E. Dendy, Jr., M. P. Ida, and J. M. Rutledge. A semicoarsening multigrid algorithm for simd machines. *SIAM J. Sci. Stat. Comput*, 13:1460–1469, 1992.

[12] C. C. Douglas. Caching in with multigrid algorithms: Problems in two dimensions. *Parallel Algorithms and Applications*, 9:195–204, 1996.

[13] R. D. Falgout and J. E. Jones. Multigrid on massively parallel architectures. Technical Report UCRL-JC-133948, Lawrence Livermore National Laboratory, 2000.

[14] G. Jin, J. Mellor-Crummey, and R. Fowler. Increasing temporal locality with skewing and recursive blocking. In *Proceedings of SC2001*, Denver, CO, Nov 2001.

[15] J. Jones. A semicoarsening multigrid algorithm for elliptic partial differential equations. Master's thesis, Mathematics Department, New Mexico Tech, 1989.

[16] S. Schaffer. A semicoarsening multigrid method for elliptic partial differential equations with highly discontinuous and anisotropic coefficients. *SIAM J. Sci. Comput*, 20(1):228–242, 1998.

[17] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[18] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[19] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Proceedings of SC'99: High Performance Networking and Computing*, Portland, OR, Nov. 1999.

[20] D. Wonnacott. A general algorithm for time skewing. Technical Report DCS-TR-449, Dept. of Computer Science, Rutgers University, July 2001. To appear in *International Journal of Parallel Programming*, June 2002, 181–221.