Automated Memory Analysis: Improving the Design and Implementation of Iterative Algorithms

by

John M. Dennis

BA, University of Colorado, 1993

MS, University of Colorado, 1998

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2005

This thesis entitled:
Automated Memory Analysis:
Improving the Design and Implementation of Iterative Algorithms
written by John M. Dennis
has been approved for the Department of Computer Science

Elizabeth R. Jessup

Professor William M. Waite

Professor Xiao-Chuan Cai

Dr. Steven J. Thomas

Professor Henry M. Tufo

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Dennis, John M. (Ph.D., Computer Science)

Automated Memory Analysis:

Improving the Design and Implementation of Iterative Algorithms

Thesis directed by Professor Elizabeth R. Jessup

Historically, iterative solvers have been designed to achieve the best numerical accuracy for a given number of floating-point operations. However, this approach ignores the cost of memory access, which has not seen nearly as rapid of an improvement as floating-point costs. To reduce the time to solution, we need to address both the numerical efficiency and memory efficiency of an iterative algorithm. We contend that it is possible to evaluate the memory efficiency of an iterative algorithm during the design process. There are two techniques for *a priori* evaluation of memory efficiency: manual and automated memory analysis. **Manual memory analysis**, which involves the derivation of analytical expression for data movement, is a laborious, error-prone process that is too complex to perform on a regular basis. Automated memory analysis is possible through the use of the Sparse Linear Algebra Memory Model (SLAMM) language processor. The SLAMM language processor accepts as input Matlab code and outputs a suitably transformed Matlab source that contains blocks of code that predict data movement. We demonstrate that the SLAMM language processor accurately predicts the amount of data loaded from the memory heirarchy to the L1 cache ($Mbytes_{L1}$) to within 20% error for numerous small kernels and complete iterative algorithms on three different compute platforms. SLAMM reduces the time to perform memory analysis from as long as several days to 20 minutes. SLAMM provides the ability to rapidly evaluate the memory efficiency of particular design choices during the design phase of an iterative algorithm. Additionally, we demonstrate how SLAMM is used to improve the memory efficiency of a pre-existing solver.

Dedication

To my lovely and patient wife, without whom I night never have finished.

Acknowledgements

I would like to acknowledge and thank the National Center for Atmospheric Research for its continuing financial support and for the sabbatical necessary to complete my studies. I also thank Professor Jessup for her support and patience over the years. I would like to thank and acknowledge my entire thesis committee, in particular Dr. Thomas, who initially suggested the use of a shifted algorithm, and Professor Waite, who provided invaluable insight into techniques to address Matlab's syntactical ambiguities. Additional thanks go to Amik St-Cyr for suggesting the mass form of the Helmholz operator and the use of right preconditioning. Finally, I would like to thank Rich Loft, whose various forms of encouragement over the years were instrumental.

Contents

Tables

Figures

Figure

Chapter 1

Introduction

Many important scientific and engineering computational problems involve the solution of a set of partial differential equations (PDEs). For example, PDEs are used to simulate airflow over an airfoil [4], blood flow through a heart [108], and weather patterns [73]. Frequently, the cores of these applications involve solving a large, sparse system of linear equations. The solution of the linear system by an iterative linear solver is often the single most costly component of the application [42, 94]. Historically, iterative solvers have been designed to achieve the best numerical accuracy for a given number of floating-point operations [79, 95, 90]. This approach is based on the assumption that the time to perform floating-point operations dominates the overall cost of the iterative solver. This assumption is no longer true, because while advances in computer architecture have significantly reduced the cost of a floating-point operation, the memory access cost has not seen nearly as rapid an improvement [82]. The time to solution is no longer strictly a function of floating-point operations, but rather a combination of the costs of floating-point arithmetic and memory access [51].

We should not focus solely on reducing the number of iterations required to converge, but also on reducing the time to solution. For example, during the implementation of a conjugate gradient solver in a prototype atmospheric model, it was observed that the use of a simple and memory-efficient preconditioner greatly reduced the time to solution versus a numerically efficient but memory-intensive alternative [66]. However, it should be emphasized that memory efficiency does not not always imply a reduction in numerical efficiency. It was demonstrated in [9] that a block variant of GMRES achieves improved memory efficiency without loss of numerical efficiency.

The creation of new algorithms that demonstrate both numerical and memory efficiency [102, 22, 8] is a non-trivial problem that has not been addressed extensively or in a systematic fashion. Sparse linear solvers are designed in either an *a priori* or an *a posteriori* manner. The *a priori* design approach integrates memory efficiency into the algorithm from the start and requires extensive knowledge of computer architecture and software engineering. The implementation is frequently a time-consuming and error-prone process, and the memory efficiency of the resulting algorithm is often unknown until it is fully implemented in a compiled language like C/C++ or FORTRAN. A memory-efficient algorithm that is poorly implemented can be mistakenly discarded.

The all-too-common *a posteriori* approach is to ignore memory efficiency until the algorithm is fully implemented in a compiled language. Modifications for memory efficiency are therefore an afterthought, not an integral component of the design process. The *a posteriori* approach places unnecessary limitations on the variety and efficiency of possible solvers. Both methodologies are unnecessarily limiting. A design technique is needed that simplifies the creation of memory-efficient sparse linear solvers.

We therefore contend that evaluation of the impact of an iterative algorithm on the memory hierarchy must be an integral component of the design process. **Memory analysis** concentrates on determining the required data movement for each component of an iterative algorithm. For memory analysis to improve the memory efficiency of iterative solvers, we need the ability to evaluate the numerical and memory efficiency of an algorithm, during its design phase. We address this need through the development of the Sparse Linear Algebra Memory Model (SLAMM) language processor to automate memory analysis. The SLAMM language processor provides the ability to rapidly evaluate the memory efficiency of particular design choices during the design phase of an iterative algorithm. Further, SLAMM can be used to improve the memory efficiency of a pre-existing solver. The SLAMM language processor simplifies the creation of memory-efficient sparse linear solvers.

In Chapter 2, we review the background of memory-efficient programming and describe the multivector optimization, a technique for improving memory efficiency. In Chapter 3, we describe several Krylov subspace algorithms, whose data movement is analyzed using both manual and automated techniques in subsequent chapters. In Section 3.1, we describe the conjugate gradient (CG) algorithm. In Section 3.2, we describe a new variant of CG, the multishifted conjugate gradient algorithm (mCG) which is based on the QD transform [38]. In Section 3.3, we describe the generalized minimum residual algorithm (GMRES) [92]. Finally, in Section 3.4, we describe a block variant of the GMRES algorithm (B-LGMRES).

In Chapter 4 we describe a manual memory analysis of the B-LGMRES algorithm implemented as part of the Portable Extensible Toolkit for Scientific Computing (PETSc) [11]. We evaluate the B-LGMRES algorithm with and without the multivector optimization. We demonstrate that it is possible to predict the impact of different implementation choices on performance using *a priori* information derived from manual memory analysis.

Manual memory analysis is a laborious and error-prone process that is too complex to perform on a regular basis. In Chapter 5, we therefore describe the development of the SLAMM language processor. The SLAMM language processor uses compiler techniques to analyze input code and generate transformed code that predicts data movement. The transformed code is subsequently executed by the Matlab interpreter to complete the automated memory analysis process. In Section 5.1, we provide background on related efforts for automated performance prediction and compiling Matlab code. Next we describe the main computational tasks of a compiler, which are lexical analysis in Section 5.2, syntactical analysis in Section 5.3, semantic analysis in Section

5.4, and transformation in Section 5.5. In Section 5.6, we describe the Eli compiler construction suite [50], which is used to develop the SLAMM language processor. In Section 5.7, we describe several of Matlab's syntactical peculiarities that complicate the development of the language processor. In Section 5.8, we describe the memory analysis computations. Finally, in Section 5.9, we provide an operational description of the SLAMM language processor.

In Chapter 6, we evaluate the ability of the SLAMM language processor to accurately predict the amount of data loaded from the memory hierarchy to the L1 cache ($Mbytes_{L1}$) for a collection of benchmarks. In Section 6.1, we describe the test configurations and methodology. In Section 6.2, we evaluate a series of linear algebra kernels. In Sections 6.3 and 6.4, we examine several conjugate gradient and GMRES algorithms, respectively. We find that SLAMM accurately predicts $Mbytes_{L1}$ for nearly all benchmarks on three different compute platforms to within 20% error. We demonstrate in Chapter 7 how a prediction within 20% error is sufficient to allow for easy identification of sections of code with excessive data movement.

In Chapter 7, we demonstrate how SLAMM is used as a valuable design tool to improve the memory efficiency of iterative algorithms. In particular, we use SLAMM to evaluate data movement in the iterative solver of the High Order Method Model Environment (HOMME). HOMME, a scalable prototype atmospheric model [104], is described in Section 7.1. In Section 7.2, we verify that an mCG-based HOMME successfully reproduces a standard atmospheric test problem. In Section 7.3, we examine the performance impact of alternative Krylov solvers in HOMME. In Section 7.3.1 we demonstrate how SLAMM is used to performance tune the existing mCG implementation in HOMME. In Section 7.3.2 we compare the mCG and CG algorithm in HOMME. The mCG algorithm demonstrates the critical impact memory access costs have on the time to solution. In Section 7.3.3 use SLAMM to predict the data movement required by several nonsymmetric solvers in the context of HOMME. The results of Chapter 6 and 7 demonstrate that SLAMM can be used as an *a priori* design tool to evaluate the impact on memory efficiency during an algorithm's design phase. Finally, in Chapter 8, we summarize our work and describe several avenues for further investigation.

Chapter 2

Background

We begin this chapter by reviewing basic computer architecture concepts and the existing work on memory-efficient programming. In Section 2.1, we review the memory hierarchy followed by Section 2.2, in which we describe the fundamental memory efficiency issues associated with iterative solvers. In Section 2.3, we describe existing work and techniques to improve the memory efficiency of existing linear algebra algorithms. In Section 2.4, we derive analytical expressions, which form the basis of memory analysis, to describe data movement. Finally, we use the analytical expressions from Section 2.4 to evaluate the multivector optimization in Section 2.5.

2.1 Computer Hardware

Modern microprocessor computer systems are composed of a central processing unit (CPU), a memory hierarchy, and a collection of peripherals. Because iterative solvers typically operate on data in the "core" or main memory of a system, we ignore the impact of peripherals and disk drives for the purposes of this work. The memory hierarchy therefore consists of registers, caches, and main memory. The CPU loads operands from the memory hierarchy into registers, performs computations, and stores the results back to memory. Moving outward from the CPU toward main memory, each level of the memory hierarchy has both increased capacity and increased access time.

The fastest pieces of the memory hierarchy are the registers, which are accessible in a single CPU clock cycle. However, register space is extremely limited and may only provide storage for 32 to 128 data values. The Level 1 cache (L1) is significantly larger, typically 16 to 64 Kbytes in size with an access time of 1 to 3 cycles. The Level 2 cache (L2) varies from 512 Kbytes to 8 Mbytes in size and has access times of 10 to 30 cycles. Level 3 caches (L3) are becoming increasingly common and vary from 2 to 32 Mbytes in size and have access times of 20 to 150 CPU cycles. Finally, main memory typically varies from 256 Mbytes to 16 Gbytes and is accessed in 40 to 200 CPU cycles.

Data is moved through the memory hierarchy in units of cache lines. Cache lines vary from 16 to 128 bytes in size and may be different for each level of cache. The links, or buses, between the different caches each have an associated width and clock rate. For example, a bus between an L1 and L2 cache that operates at 400 Mhz and has width of 64 bits or 8 bytes can transfer data at a maximum rate of 3200 Mbytes/sec or 3.2 Gbytes/sec. Section 6.1 contains a detailed description of the experimental compute platforms used for this paper.

2.2 Generic Iterative Solver Issues

We next describe the algorithmic issues associated with iterative solvers. Iterative solvers find an approximate solution x for the equation

$$Ax = b,$$

where $A \in \mathbb{R}^{n \times n}$ is a sparse coefficient matrix and $x, b \in \mathbb{R}^n$ are dense vectors. The matrix A is stored in a sparse format due to memory limitations. All iterative solvers perform two fundamental types of operations. The first type is the calculation of matrix-vector products and involves accessing the sparse coefficient matrix A. The second type of calculation involves dense vector operations. Because the issues associated with dense linear algebra are well known [33, 31, 18, 63], we concentrate on the first type of calculation. Improving the memory characteristics of compressed sparse row (CSR), which is the basic storage format for sparse matrices, has been the focus of a large body of research [1, 62, 75]. Figure 2.1 illustrates the code necessary to calculate $y = Ax$, where x and y are vectors and A is a matrix in CSR format. The variable $Aval$ contains the matrix nonzero values, ia is the column pointer, and ja is the row pointer. The number of nonzeros is nnz and the matrix order is n. The array x is indexed using indirect addressing, while y is indexed using direct addressing. The use of indirect addressing significantly increases access time. Attempts to eliminate it or reduce its impact are the fundamental motivation for all research into improving sparse storage formats.

```
int nnz;                  /* Number of nonzeros */
int n;                    /* Matrix order */
double Aval[nnz]          /* Nonzero values */
int ia[n+1];              /* Column pointers */
int ja[nnz];              /* Row pointers */
double x[n],y[n];         /* Vectors */

for (i=0; i<n; i++) {
        is = ia[i];
        ie = ia[i+1];
        tmp = 0.0;
        for (j=is; j< ie; j++) {
         tmp = tmp + Aval[j]*x[ja[j]];
        }
        y[i]=tmp;
   }
```

Figure 2.1: Code to perform a matrix-vector multiply for a coefficient matrix A stored in CSR format.

2.3 Memory-Efficient Programming

Memory-efficient programming attempts to minimize the impact of the memory hierarchy on the execution time of a piece of code. Memory efficiency is achieved by increasing a code's **spatial** or **temporal memory reference locality**. A code has spatial locality if successive memory references tend to access adjacent words of memory.

A code has temporal locality if memory references that are close in time repeatedly access the same pieces of data. To understand the distinction, consider the calculation of a dot product of two vectors with a stride-one access pattern. The dot product calculation uses direct addressing, so successive words of each vector are accessed in order, generating a memory access pattern with good spatial locality. The dot product calculation does not itself have good temporal locality because the operand vectors are accessed only once. However, if the dot product calculation were used as part of a matrix-matrix multiply, the code would have both good spatial and temporal locality, because the vectors would be accessed repeatedly. A result of the dot product's good spatial locality is that it uses cache lines efficiently, because all or most of the words in the cache line are used by the CPU as operands. Conversely, a code employing indirect addressing may exhibit poor spatial locality. Poor spatial locality results in inefficient use of cache lines and increases the required memory bandwidth. However, it is possible to increase the temporal locality of any code by grouping computations on the same operands together.

Increasing memory reference locality reduces an algorithm's dependence on memory latency, bandwidth, or both. Memory latency refers to the time between when a CPU issues a load for an operand and when the operand arrives in the registers. Latency depends on the location of the operand in the memory hierarchy. The load latency is considerably longer for operands in main memory than for those located in the L1 cache. Prefetching operands is a general technique for addressing load latency [2, 106, 61, 76]. Prefetching involves the loading of operands into the L1 cache before they are needed by the CPU and is implemented in either software or hardware. Both types of prefetching require an accurate prediction of the load access time. Furthermore, care must be taken to prevent the prefetched load from removing other pieces of data currently in use, unintentionally resulting in increased memory traffic.

Existing data structure techniques also reduce the impact of memory latency for the two memory loads necessary for indirect addressing. The first load determines the address of the operand, while the second loads the actual operand. The blocked AIJ approach reduces the amount of indirect addressing required by changing the underlying data structures of the coefficient matrix A to be a collection of small dense blocks. Indirect addressing is therefore only needed to locate the first element of the block. The blocked AIJ approach, which frequently requires the matrix to be reordered to locate small dense blocks, has been successfully applied by [106, 56, 83, 114] to improve spatial memory reference locality. Its effectiveness depends on the nonzero structure of the matrix. A related technique involves locating identical nonzero structures in successive rows (inodes) of the matrix [11]. While this approach does not address the spatial locality of the code, it does increase temporal locality.

In addition to memory latency, bandwidth limitations also impact an algorithm's performance. Memory bandwidth is the rate at which data is moved through the memory hierarchy. We estimate the impact of memory bandwidth on a particular algorithm by comparing the total storage requirement of an algorithm and its working set size to the size of the cache. The total storage requirement is the total size in bytes of all variables required by an algorithm. The working set size is the size in bytes of all variables loaded from the memory hierarchy during a particular section of the algorithm. Different implementations of the same algorithm might have the same total storage requirement but different working set sizes. The impact of the memory hierarchy on an algorithm's performance is minimized when the total storage requirement or working set size are smaller than the cache size. A technique that matches the working set size to cache size is loop blocking [47, 18, 63]. Loop blocking breaks up large code blocks or loops into smaller blocks to improve data reuse. Loop blocking is performed by the optimized Level 2 and 3 BLAS routines [33, 31]. The ATLAS project [116] demonstrated that the optimal size of blocks can be determined experimentally. Loop blocking primarily addresses the spatial locality of the code. Finally, the multivector optimization, which increases both spatial and temporal locality, allows the calculation of four matrix-vector products in only 50% more time than needed to perform a single matrix-vector product [51, 52]. We examine the multivector optimization in detail in Section 2.5.

2.4 Quantifying Memory Efficiency

The impact that the memory hierarchy has on sparse linear algebra was first analyzed by Temam and Jalby [102]. Their work takes a probabilistic approach to model the impact of cache and the nonzero pattern of a general sparse matrix-vector multiply. Unlike Temam and Jalby, we do not concentrate on the form of the nonzero pattern, but rather on the amount of data loaded through the memory hierarchy. We approximate data movement based on the total storage requirement (SR), amount of data accessed size (AS) and the working set load size (WSL) or the amount of data loaded from the memory hierarchy for an algorithm. We calculate the total storage requirement by summing the size in bytes for all variables required by a section of an algorithm. Consider the matrix-vector multiply algorithm in Figure 2.1. The total storage requirement for the matrix-vector multiply is

$$SR \equiv sizeof(A) + sizeof(x) + sizeof(y),$$

where sizeof() is a function that returns the size of its argument in bytes. The matrix A in CSR format requires an array of double precision floating-point values of length nnz, an integer array ia of length nnz, and an integer array ja of length n. The arrays ia and ja represent the column and row pointers from Figure 2.1, respectively. Therefore, the size of the matrix A is

$$sizeof(A) = sizeof(double) \cdot nnz + sizeof(int) \cdot (n + nnz), \qquad (2.1)$$

where *double* is a double precision value and *int* is an integer value. Because the size in bytes of the vectors x, y from Figure 2.1 are $sizeof(double) \cdot n$, the total storage requirement for the code in Figure 2.1 is

$$SR \equiv (nnz + 2n)L_d + (nnz + n)L_i,$$

where $L_d = sizeof(double)$ and $L_i = sizeof(int)$.

We now derive the expressions for AS and WSL for the calculation of a single element of the vector y using the matrix-vector multiply routine from Figure 2.1. The calculation of the i^{th} value of y or $y(i)$ requires two values of ia, values of $Aval$, ja, x, and $y(i)$. All variables except $y(i)$ must be loaded from the memory hierarchy; therefore, it is easy to determine AS and WSL. The AS and WSL to calculate $y(i)$ are

$$AS_{y(i)} = (2 \cdot rowlen_i + 2)L_d + (rowlen_i + 2)L_i$$

$$WSL_{y(i)} = (2 \cdot rowlen_i + 1)L_d + (rowlen_i + 2)L_i,$$

where $rowlen_i = (ia(i+1) - ia(i) + 1)$. While the difference between the values for $AS_{y(i)}$ and $WSL_{y(i)}$ is minor in this case, it is not always so. For example, when the result of the computation is similar in size to the operands, AS and WSL are significantly different. Because the goal of memory analysis is to accurately predict required data movement, and the AS value includes the size of variables that also must be stored to the memory hierarchy, we use WSL for comparison with measured values hereafter.

2.5 Multivector Optimization

The multivector optimization is a technique that involves the use of special data structures to locate corresponding members of different vectors next to each other in memory. This arrangement is described in [51, 8] as an interlaced storage format. Consider a set of s vectors $V = [v_1, v_2, \ldots, v_s]$, where $v_i \in \mathbb{R}^{n\times 1}$. Let $v_i(j)$ be the j^{th} element of vector v_i. Use of the interlaced storage format places the j^{th} elements of $v_i \in V : i = 1, s$ next to each other in memory. The interlaced storage format contrasts with the non-interlaced or standard approach, which places successive elements of a vector

v_i next to each other. For example, the first r elements where $r < s$ of a multivector V are $v_1(1), v_2(1), \ldots, v_r(1)$ versus $v_1(1), v_1(2), \ldots, v_1(r)$ for the standard approach. The rearrangement of data necessary for the interlaced storage format directly increases spatial locality.

Use of the interlaced storage format requires rewriting all fundamental linear algebra operations required by an iterative solver. For example, an iterative solver for $AX = B$, where X and B are multivectors, requires rewriting the matrix-multivector multiply and other common linear algebra operations such as dot product, axpy, and others. The advantage of this data structure arrangement is that it increases both spatial and temporal locality, resulting in greater data reuse. For example, consider the calculation of the dot product $W = X^T V$ of two multivectors $X, V \in \mathbb{R}^{n\times s}$ and $W \in \mathbb{R}^{s\times s}$, where X and V each contain constituent vectors $x_i, v_j \in \mathbb{R}^{n\times 1}, i = 1 : s, j = 1 : s$. The storage requirement of the dot product operation is

$$SR \equiv sizeof(X) + sizeof(V) + sizeof(W) = (2n + s)sL_d.$$

Let $multiDot(X, V)$ be the multivector dot product, while $Dot(x_i, v_j)$ is the standard dot product. We consider the standard approach first. The standard approach involves forming the dot product $Dot(x_i, v_j)$ for vectors $x_i, v_j, i = 1 : s, j = 1 : s$ with s^2 calls to DOT(). Each call to DOT() requires accessing $2 \cdot n$ data values. The WSL of a single call to the DOT() operation is

$$WSL_{DOT()} \equiv 2nL_d.$$

Therefore, the calculation of $DOT(x_i, v_j), i = 1 : s, j = 1 : s$ has a working set size of

$$WSL_{DOT(x_i,v_j)} \equiv 2ns^2 L_d.$$

A single call to the $multiDot(X, V)$ routine calculates all s^2 dot products of the constituent vectors $x_i, v_j, i = 1 : s, j = 1 : s$ simultaneously. The multivector dot product therefore accesses each x_i, v_j once because of the rearrangement of data structures. Consequently, the working set load size is reduced to

$$WSL_{multiDot(X,V)} \equiv 2nsL_d.$$

The potential impact of this rearrangement on data movement is determined by comparing the total storage requirement versus the size of the cache. If $SR < sizeof(cache)$, then the use of the interleaved storage format has no impact on data movement because both the standard and multivector implementations have similar data reuse. However, if $SR > sizeof(cache)$, then all operands do not fit into cache. In this case, the working set size of an implementation determines the amount of data movement. An implementation with the smaller WSL has greater data reuse and reduced data movement. In the dot product example from the previous paragraph, we see that if $SR \gg sizeof(cache)$, then a single call to $multiDOT(X, V)$ reduces data movement versus $DOT(x_i, v_j)$ by a factor of s because

$$\frac{WSL_{Dot(x_i,v_j)}}{WSL_{multiDot(X,V)}} = \frac{2ns^2}{2ns} = s.$$

The advantages of multivectors are not limited to dot products but also hold for other linear algebra operations. In fact, the ratio of working set sizes for $s = 2$ is demonstrated in Chapter 4 for an entire iterative solver.

To evaluate the advantage of the multivector optimization on an entire iterative solver, we must first determine the impact of the multivector optimization on several other common linear algebra operations, including matrix-vector multiplication and axpy.

Let $multiMxV(A, V)$ be the matrix-multivector multiplication, while $MxV(A, v_i)$ is the standard matrix-vector multiplication. We consider the standard approach first. The standard approach involves forming each matrix-vector production $MxV(A, v_i)$ for $i = 1 : s$ with s calls to $MxV()$. Because $WSL_{MxV()} = sizeof(A) + sizeof(v_i)$, using

(2.1) the working set load size for a single call to $MxV()$ is

$$WSL_{MxV()} \equiv (nnz + n)L_d + (nnz + n)L_i.$$

Therefore, the calculation of $MxV(A, v_i)$ for $i = 1 : s$ has a working set size of

$$WSL_{MxV(A,v_i)} \equiv s \cdot (nnz + n)L_d + s \cdot (nnz + n)L_i.$$

A single call to the $multiMxV(A, V)$ routine calculates all s matrix-vector products of the constituent vectors simultaneously. The matrix-multivector multiply therefore only accesses A once and the working set size is reduced to

$$WSL_{multiMxV(A,V)} \equiv (nnz + ns)L_d + (nnz + n)L_i.$$

If $SR > sizeof(cache)$, then all operands do not fit into cache and the working set size of an implementation determines the amount of data movement. So if $SR \gg sizeof(cache)$ and $nnz \gg n \cdot s$, then a single call to $multiMxV(A, V)$ reduces data movement versus $MxV(A, v_i)$ by as much as a factor of s because

$$WSL_{MxV(A,v_i)}/WSL_{multiMxV(A,V)} \approx s.$$

Let $multiAXPY(U, V)$ be the multivector axpy operation that evaluates the expression $U = U + V \cdot \Gamma$, where $U, V \in \mathbb{R}^{n \times s}$ and $\Gamma \in \mathbb{R}^{s \times s}$. Let $AXPY(u, v)$ represent the standard approach that evaluates the expression $u = u + \gamma v$, where $u, v \in \mathbb{R}^n$ and $\gamma \in \mathbb{R}$. We consider the standard approach first. The working set size for a single call to $AXPY()$ is

$$WSL_{AXPY()} \equiv (2n + 1)L_d.$$

Therefore, the calculation of $AXPY(u_i, v_j)$ for $i = 1 : s, j = 1 : s$ has a working set load size of

$$WSL_{AXPY(u_i,v_j)} \equiv (2n + 1)s^2 L_d.$$

Because a single call to the $multiAXPY(U, V)$ routine calculates all s^2 axpy operations simultaneously, the working set size is

$$WSL_{multiAXPY(U,V)} \equiv (2n + s)sL_d.$$

If $SR \gg sizeof(cache)$ and $n \gg s$, then a single call to $multiAXPY(U, V)$ reduces data movement versus $AXPY(u_i, v_j)$ by as much as a factor of s because

$$WSL_{AXPY(u_i,v_j)}/WSL_{multiAXPY(U,V)} \approx s.$$

The storage requirements and working set sizes for both non-multivector and multivector versions of the dot product, axpy, and matrix-vector multiply are summarized in Table 2.1. Note that each of the standard routines must be executed the number of times indicated in column 2 of Table 2.1 to calculate the equivalent result of a single call to the multivector equivalent of size s.

Table 2.1: Total storage requirements, and working set sizes for the non-multivector and multivector implementations of the dot product, axpy, and matrix-vector multiply routines.

Operation	# of calls	SR	WSL
	Non-Multivector Operations		
$DOT(u_i, v_j)$	s^2	$(2n + s)sL_d$	$2ns^2L_d$
$AXPY(u_i, v_j)$	s^2	$(2n + s)sL_d$	$(2n + 1)s^2L_d$
$MxV(A, u_i)$	s	$(nnz + 2ns)L_d$	$(nnz + n)sL_d$
		$+(nnz + n)L_i$	$+(nnz + n)sL_i$
	Multivector Operations		
$multiDOT(U, V)$	1	$(2n + s)sL_d$	$2nsL_d$
$multiAXPY(U, V)$	1	$(2n + s)sL_d$	$(2n + s)sL_d$
$multiMxV(A, U)$	1	$(nnz + 2ns)L_d$	$(nnz + ns)L_d$
		$+(nnz + n)L_i$	$+(nnz + n)L_i$

Chapter 3

Krylov Subspace Algorithms

In Section 2.5 we derived the analytical expressions that describe the data movement required for several linear algebra operations. Because iterative solvers are composed of a collection of linear algebra operations, we can also derive the corresponding analytical expression for an entire iterative algorithm. In Section 3.4, we provide a derivation of the analytical expressions for a Krylov iterative algorithm, which is used for the manual memory analysis in Chapter 4. However, we do not provide derivations for all iterative algorithms described in this chapter. Instead, we use automated memory analysis on the remaining iterative algorithms in Chapter 6, a technique that does not require the derivation of any analytical expressions. We next describe several Krylov algorithms on which memory analysis is applied in subsequent chapters.

Krylov subspace solvers are a family of iterative algorithms that approximate a solution to

$$Ax = b, \tag{3.1}$$

where $A \in \mathbb{R}^{n \times n}$ is a sparse coefficient matrix and $x, b \in \mathbb{R}^n$ are dense vectors. The solution x is of the form $x \in x_0 + \mathcal{K}_m(A, r_0)$, where

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2 r_0, \dots, A^{m-1} r_0\}$$

is a Krylov subspace of size m. Here x_0 is an initial guess and the initial residual r_0 is $r_0 = b - Ax_0$. The m^{th} estimate for the solution x_m must satisfy the Galerkin condition

$b - Ax_m \perp \mathcal{L}_m$, where $\mathcal{L}_m$ is another m dimensional subspace. Different members of the Krylov subspace family have different subspaces $\mathcal{L}_m$. The iterative process continues until convergence is achieved.

Two Krylov subspace algorithms are used extensively in practice. The Conjugate Gradient (CG) algorithm [54] was originally developed in 1952 as a direct solver, where A is a symmetric positive definite matrix. The CG algorithm was discarded, because it had inferior numerical properties and equivalent operation counts to Gaussian Elimination with pivoting. It was later advanced by [38, 88] as a iterative solver. Saad [92] developed the general minimum residual (GMRES) algorithm when A is nonsymmetric. Like CG, GMRES is stable and guaranteed to converge in at most n iterations.

In Section 3.1, we describe the basic CG algorithm along with a version of CG with a merged inner-product [28]. In Section 3.2, we describe a multishifted Conjugate Gradient (mCG) algorithm whose memory efficiency we compare with the merged inner-product CG algorithm in Chapter 7. We next examine the GMRES algorithm in Section 3.3, followed by a block variant of GMRES in Section 3.4. We do not examine the convergence issues associated with each existing variant here, but rather provide the basis for evaluating the memory efficiency of each algorithm.

3.1 Conjugate Gradient

The conjugate gradient method (CG) is a Krylov subspace algorithm where $\mathcal{L}_m = \mathcal{K}_m$ and A is symmetric positive definite. As with all Krylov subspace methods, the initial residual $r_0 = b - Ax_0$ and the approximate solution $x_m \in x_0 + \mathcal{K}_m$ satisfy the Galerkin condition $b - Ax_m \perp \mathcal{K}_m$. If $v_1 = r_0 / \parallel r_0 \parallel_2$, where $\beta = \parallel r_0 \parallel_2$, then Arnoldi's method constructs a orthogonal basis V_m such that

$$V_m^T A V_m = H_m.$$

Then because $x_m = x_0 + V_m y_m$

$$\begin{aligned} r_m &= b - Ax_m \\ &= b - A(x_0 + V_m y_m) \\ &= r_0 - AV_m y_m. \end{aligned}$$

By the Galerkin condition $V_m^T r_m = 0$, $V^T r_0 = V^T A V_m y_m = H_m y_m = \beta e_1$ holds because $V^T r_0 = \beta e_1$. The approximate solution x_m after m iterations is

$$\begin{aligned} x_m &= x_0 + V_m y_m \\ y_m &= H_m^{-1}(\beta e_1). \end{aligned}$$

Because A is symmetric, H_m is also symmetric and therefore equal to a tridiagonal matrix T_m. The matrix T_m can be decomposed by LU factorization, which allows the new approximation x_j to be calculated by a simple recurrence relationship [91]. The complete conjugate gradient algorithm is provided in Figure 3.1, where $A \in \mathbb{R}^{n \times n}$, $r_j, p_j, w, x_j \in \mathbb{R}^n$, and $\alpha_j, \beta_j, \phi_j \in \mathbb{R}$. The Conjugate Gradient algorithm in Figure 3.1 requires two inner-products: the first (w, p_j) in line 4 and the second in line 8 of Figure 3.1. Because an update of the residual vector r_{j+1} is both dependent on the first inner-product and required for the second inner-product, both inner-products must be calculated separately.

A variation of the base Conjugate Gradient algorithm with a merged inner-product [28] allows all inner-products to be calculated simultaneously. The merged inner-product CG is particular well-suited for parallel applications, an environment in which the calculation of separate inner-products is particularly expensive. The merged inner-product version of CG, which is based on a rearrangement of the loop structure, is provided in Figure 3.2, where $A \in \mathbb{R}^{n \times n}$, $r_j, p_j, s_j, v_j, x_j \in \mathbb{R}^n$, and α_j, β_j, γ_j, ϵ_j, δ_j, $\sigma_j \in \mathbb{R}$. A single iteration of the base CG algorithm is performed in lines 1 and

```
1. $r_0 = b - Ax_0, p_0 = r_0, \phi_0 = (r_0, r_0)$
2. for $j = 0, 1, \ldots$ until convergence
3.         $w = Ap_j$
4.         $\delta_j = (w, p_j)$ /* inner-product */
5.         $\alpha_j = \phi_j / \delta_j$
6.         $x_{j+1} = x_j + \alpha_j p_j$
7.         $r_{j+1} = r_j - \alpha_j w$
8.         $\phi_{j+1} = (r_{j+1}, r_{j+1})$ /* inner-product */
9.         $\beta_j = \phi_{j+1} / \phi_j$
10.        $p_{j+1} = r_{j+1} + \beta_j p_j$
11. end
```

Figure 3.1: Conjugate Gradient (CG) algorithm.

2 to initialize the iteration. Note that three inner-products lines 6 to 8 can be calculated simultaneously. The memory efficiency of the merged inner-product CG algorithm along with the multishifted Conjugate Gradient algorithm described next is examined in detail in Chapter 7.

3.2 Multishifted Conjugate Gradient

We next examine an iterative algorithm that solves a shifted set of linear equations

$$(A + \sigma^{(k)} I) x^{(k)} = b, \tag{3.2}$$

where $\sigma^{(k)} \in \mathbb{R}$, $A \in \mathbb{R}^{n \times n}$ is symmetric positive definitive and I is the identity matrix. The value $\sigma^{(k)}$ is a diagonal shift of the coefficient matrix A. Shifted Kyrlov iterative algorithms are possible because $\mathcal{K}(A, r_0) = \mathcal{K}(A + \sigma I, r_0)$, where $\mathcal{K}(A, r_0) = span\{r_0, A\,r_0, A^2 r_0, \ldots\}$. It is therefore possible to approximate multiple solutions $x^{(k)} = x_0^{(k)} + \mathcal{K}(A, r_0)$ by calculating a single Krylov subspace.

Several multishifted Krylov algorithms exist [44, 58, 41, 109, 46], for the solution of shifted linear systems. The multishifted Conjugate Gradient Least Squares algorithm (CGLS) of van den Eshof and Sleijpen [109] is based on the stationary QD transform of Rutishauser [38]. The QD transform allows the calculation of the $L^\sigma D^\sigma L^{\sigma T}$ factorization

```
1. $r_0 = b, \gamma_0 = (r_0, r_0), p_0 = r_0, v_0 = Ap_0, s_0 = v_0$
2. $\sigma_0 = (p_0, v_0), x_1 = \gamma_0 p_0 / \sigma_0$
3. for $j = 1, 2, \dots$ until convergence
4.          $z_j = M^{-1} r_j$
5.          $s_j = A z_j$
            /* inner-products */
6.          $\gamma_j = (z_j, r_j)$
7.          $\delta_j = (z_j, s_j)$
8.          $\epsilon_j = (r_j, s_{j-1})$
            /* calculate scalars */
9.          $\beta_j = \gamma_j / \gamma_{j-1}, \sigma_j = \delta_j + \beta_j \epsilon_j, \alpha_j = \gamma_j / \sigma_j$
            /* update vectors */
10.         $p_j = z_j + \beta_j p_{j-1}$
11.         $v_j = s_j + \beta_j v_{j-1}$
12.         $x_{j+1} = x_j + \alpha_j p_j$
13.         $r_{j+1} = r_j - \alpha_j v_j$
14. end
```

Figure 3.2: Merged Inner-Product Conjugate Gradient algorithm.

of a shifted system based on the LDL^T of a base system, where $L^\sigma D^\sigma L^{\sigma T} = LDL^T + \sigma I$. However, we are not interesting in solving (3.2) for $x^{(k)}$ for a single right-hand side but for different right-hand sides $b^{(k)}$. We must therefore solve

$$(A + \sigma^{(k)} I) x^{(k)} = b^{(k)}. \tag{3.3}$$

A possible technique to solve (3.3) is by the use of a multishifted block Conjugate Gradient algorithm. We first solve the block system $AX' = B$ where $X' = [x'^{(1)}, x'^{(2)}, \dots, x'^{(s)}]$ and $B = [b^{(1)}, b^{(2)}, \dots, b^{(s)}]$ using a block Conjugate Gradient [35, 79, 80] algorithm. We use the QD transform to apply the shift $\sigma^{(k)}$ and calculate $x^{(k)}$. However, a multishifted block Conjugate Gradient algorithm, also unnecessarily solves

$$(A + \sigma^{(l)} I) x^{(l)} = b^{(k)}, \tag{3.4}$$

where $l \neq k$. While a block Conjugate Gradient algorithm potentially reduces iteration count, we do not believe the reduction in iteration count is sufficient to defray the additional cost of solving (3.4).

We therefore combine the Conjugate Gradient algorithm in Figure 3.1 with the QD transform technique to create our multishifted Conjugate Gradient (mCG) algorithm. While we must compute multiple Krylov subspaces, our mCG algorithm allows for other potential advantages. In particular, this algorithm requires only a single coefficient matrix A and thus a single preconditioner for all shift values. However, the potential utility of the multishifted iterative algorithm is limited because preconditioners must maintain the shifted structure [58]. Consider the preconditioned shifted system

$$M^{-1}(A + \sigma^{(k)} I)x^{(k)} = M^{-1}b^{(k)}, \tag{3.5}$$

where M is a preconditioner with shifted structure. We use right preconditioning to transform (3.5) while maintaining its shifted structure,

$$\begin{aligned} (M^{-1}A + \sigma^{(k)} M^{-1})x^{(k)} &= M^{-1}b^{(k)} \\ (M^{-1}AMM^{-1} + \sigma^{(k)} M^{-1})x^{(k)} &= M^{-1}b^{(k)} \\ (M^{-1}AM + \sigma^{(k)})\tilde{x}^{(k)} &= M^{-1}b^{(k)}, \end{aligned}$$

where $\tilde{x}^{(k)} = M^{-1}x^{(k)}$. The final form is

$$(M^{-1}\tilde{A} + \sigma^{(k)} I)\tilde{x}^{(k)} = M^{-1}b^{(k)}, \tag{3.6}$$

where $\tilde{A} = AM$. The multishifted Conjugate Gradient algorithm with right preconditioning is illustrated in Figure 3.3. Because the mCG algorithm is currently based on the CG algorithm in Figure 3.1, it possesses the same inner-product structure. The mCG algorithm is therefore less desirable for parallel computing environments. However, we believe that it is possible to apply the QD transform technique to the merged inner-product version of CG.

3.3 GMRES

GMRES is a Krylov subspace method where $\mathcal{L}_m = A\mathcal{K}_m$. The Krylov subspace $\mathcal{K}_m$ has an orthonormal basis $V_m = \{v_1, v_2, \ldots, v_m\}$, where $v_1 = r_0/\beta$, and $\beta = \| r_0 \|_2$.

```
1.  $r_0 = b, p_0 = r_0, z_0 = M^{-1}r_0, \phi_0 = (z_0, r_0), \tilde{A} = AM$
2.  $\tilde{x}_0^\sigma = 0, p_0^\sigma = z_0, \gamma_0^\sigma = 1, t_0^\sigma = \sigma$
3.  for $j = 0, 1, \ldots$ until convergence
4.          $w_j = \tilde{A}p_j$
5.          $\alpha_j = \phi_j/(w_j, p_j)$
6.          $r_{j+1} = r_j - \alpha_j w_j$
7.          $z_{j+1} = M^{-1}r_{j+1}$
8.          $\phi_{j+1} = (z_{j+1}, r_{j+1})$
9.          $\beta_{j+1} = \phi_{j+1}/\phi_j$
10.         $p_{j+1} = z_{j+1} + \beta_{j+1}p_j$
             /* constants for shifted system */
11.         $l^\sigma = 1 + \alpha_j t_j^\sigma$
12.         $t_{j+1}^\sigma = \sigma + \beta_{j+1}t_j^\sigma/l^\sigma$
13.         $\gamma_{j+1}^\sigma = \gamma_j^\sigma l^\sigma$
            /* update vectors for shifted system */
14.         $\tilde{x}_{j+1}^\sigma = \tilde{x}_j^\sigma + \alpha_j p_j^\sigma/\gamma_{j+1}^\sigma$
15.         $p_{j+1}^\sigma = z_{j+1} + \beta_{j+1}p_j^\sigma/l^\sigma$
16. end
17. $x_{j+1}^\sigma = M\tilde{x}_{j+1}^\sigma$
```

Figure 3.3: Preconditioned Multishifted Conjugate Gradient (mCG) algorithm.

The orthonormal basis V_m is created by Arnoldi's method, which constructs an upper Hessenberg matrix H_m such that

$$AV_m = V_{m+1}H_m, \tag{3.7}$$

where $V_m \in \mathbb{R}^{m\times n}$ and $H_m \in \mathbb{R}^{(m+1)\times m}$.

The solution x to (3.1) is approximated after m iterations by $x_m = x_0 + V_m y$ where $y \in \mathbb{R}^m$. For convergence, we want to minimize

$$\| r_m \|_2 = \| b - Ax_m \|_2 = \| b - A(x_0 + V_m y) \|_2 .$$

Using (3.7) we get,

$$\begin{aligned} b - A(x_0 + V_m y) &= r_0 - AV_m y \\ &= \beta v_1 - V_{m+1}H_m y \\ &= V_{m+1}(\beta e_1 - H_m y), \end{aligned}$$

where e_1 is the canonical vector. We can find a y that minimizes $\| \beta e_1 - H_m y \|_2$ by solving the $(m+1)\times m$ least squares problem. The full GMRES algorithm is illustrated in

```
1.  $r_0 = b - Ax_0, \beta = \| r_0 \|_2, v_1 = r_0/\beta$
2.  for $j = 1 : m$
3.      $w = Av_j$
4.      for $i = 1 : j$
5.          $h_{i,j} = (w, v_j)$
6.          $w = w - v_i h_{i,j}$
7.      end
8.      $h_{j+1,j} = \| w \|_2$
9.      $v_{j+1} = w/h_{j+1,j}$
10. end
11. $H_m = \{h_{i,j}\}_{1 \le i \le j+1; 1 \le j \le m}$
12. find $y_m$ s.t. $\| \beta e_1 - H_m y \|_2$ is minimized
13. $x_m = x_0 + V_m y_m$
```

Figure 3.4: GMRES.

Figure 3.4. Note that a dot product indicated by (w, v_j) is required in line 5 of Figure 3.4 as well as an axpy operation in line 6. Lines 4 to 7 of Figure 3.4 represent the modified Gram-Schmidt orthogonalization in the Arnoldi process. The size of the orthogonal basis V_m increases linearly as the number of iterations j increases, while the amount of orthogonalization work increases exponentially. If the number of iterations required to converge m_c is small ($m_c \ll n$), then the linear increase in work may be acceptable. If however $m_c \approx n$, the amount of orthogonalization work and required storage becomes untenable. This characteristic of GMRES has prompted the development of a restarted version of GMRES. Restarted GMRES with a restart size of m (GMRES(m)) [92] limits the size of the orthogonal basis to m vectors. If the convergence criterion is not satisfied, the approximate solution x_m is used as the initial guess x_0 for another cycle of GMRES. This process is repeated until convergence is achieved.

3.4 Block LGMRES

Next we examine a block Krylov subspace algorithm. Block algorithms have long been advocated as an approach to reduce the time to solution for a series of linear systems with an identical coefficient matrix and multiple right-hand sides. The equation

$$Ax^{(k)} = b^{(k)}, \tag{3.8}$$

for $k = 1, \ldots, s$ right-hand sides is rewritten as

$$AX = B, \tag{3.9}$$

where $X = [x^{(1)}, x^{(2)}, \ldots, x^{(s)}]$ and $B = [b^{(1)}, b^{(2)}, \ldots, b^{(s)}]$. A block Krylov subspace after m iterations is

$$\mathcal{K}_m(A, R_0) = span\{r_0^{(1)}, \ldots, r_0^{(s)}, Ar_0^{(1)}, \ldots, Ar_0^{(s)}, \ldots, A^{m-1}r_0^{(1)}, \ldots, A^{m-1}r_0^{(s)}\},$$

where $R_0 = B - AX_0$ for some set of initial guesses X_0. The approximate solution X_m after m iterations is $X_m \in X_0 + \mathcal{K}_m(A, R_0)$.

We concentrate on one block Krylov subspace algorithm in particular, the block LGMRES (B-LGMRES) algorithm [10]. LGMRES ("Loose" GMRES) is a variant of restarted GMRES that accelerates convergence by augmenting the Krylov subspace with error approximations. The algorithm LGMRES(m,k) creates a Krylov subspace of size $m + k$ where there are m vectors formed in the standard way augmented with k error approximations. The i^{th} error approximation is $z_i = x_i - x_{i-1}$, where x_i and x_{i-1} represent the approximate solution after the i^{th} and $(i-1)^{th}$ restart cycles respectively. Error approximations from the k previous restart cycles are appended to the Krylov subspace. The Krylov space after the i^{th} restart cycle, where $i > k$ for LGMRES(m,k), is thus

$$\mathcal{K}(A, r_{i-1}) = span\{r_{i-1}, Ar_{i-1}, A^2r_{i-1}, \ldots, A^{m-k-1}r_{i-1}, z_{i-k-1}, \ldots, z_{i-1}\},$$

where r_{i-1} is the residual from the $(i-1)^{st}$ restart cycle. The inclusion of error approximations from previous restart cycles reduces the loss of residual information from previous restart cycles [10].

```
1. $r_i = b - Ax_i$, $\beta = \|r_i\|_2$
2. $R_i = [r_i, z_i, \ldots, z_{i-k+1}]$
3. $R_i = V_1\hat{R}$
4. for $j = 1 : m$
5.     $U_j = AV_j$
6.     for $l = 1 : j$
7.         $H_{l,j} = V_l^T U_j$
8.         $U_j = U_j - V_l H_{l,j}$
9.     end
10.    $V_j = U_{j+1} H_{j+1,j}$
11. end
12. $W_m = [V_1, V_2, \ldots, V_m]$, $H_m = \{H_{l,j}\}_{1 \le l \le j+1; 1 \le j \le m}$
13. find $y_m$ s.t. $\|\beta e_1 - H_m y_m\|_2$ is minimized
14. $z_{i+1} = W_m y_m$
15. $x_{i+1} = x_i + z_{i+1}$
```

Figure 3.5: B-LGMRES(m, k) for restart cycle i.

Appending the z_i error approximations to the end of the Krylov subspace is one technique to add error approximation information into the solution space. Adding the z_i approximations as additional right-hand side vectors creates a block formulation. For the block implementation of LGMRES (B-LGMRES(m,k)), set $B = [b, z_{i-k-1}, \ldots, z_{i-1}]$ and $X = [x_i, 0, \ldots, 0]$, where $X, B \in \mathbb{R}^{n \times s}$ and $s = k + 1$. The solution X is approximated after m iterations by $X \in X_0 + \mathcal{K}_m(A, R_0)$, where

$$\mathcal{K}_m(A, R_0) = span\{r_{i-1}, Ar_{i-1}, \ldots, A^{m-k-1}r_{i-1}, \ldots, A^{m-k-1}z_{i-k-1}, \ldots, A^{m-k-1}z_{i-1}\}.$$

The block formulation of B-LGMRES in Figure 3.5 allows the use of the multivector optimization described in Section 2.5.

We analyze the impact the multivector optimization techniques described in Section 2.5 have on the block algorithm by analyzing the required data movement on a line-by-line basis. Table 3.1 provides the analytical expressions for both the total storage requirement (SR) and working set size (WSL), both with the multivector optimization (MV) and without (non-MV). Additionally, we also provide totals for three sections of the B-LGMRES algorithm. We denote the matrix-vector product in line 5 the MatMult section. Lines 6 to 9 in Figure 3.5 are the modified Gram-Schmidt orthogonalization (MGS), while the remaining lines 10 and 12 to 15 are denoted Other.

Table 3.1: Total storage requirements and working set sizes for both the non-multivector (non-MV) and multivector (MV) approachs for each line of the B-LGMRES algorithm in Figure 3.5, where $s = k + 1$.

Line	Operation	SR	WSL (bytes)	
			non-MV	MV
		MatMult		
5	$MxV()$	$(nnz + 2sn)L_d$ $+(nnz + n)L_i$	$s\,(nnz + n)L_d$ $+s\,(nnz + n)L_i$	$(nnz + sn)L_d$ $+(nnz + n)L_i$
		MGS		
7	$j \times DOT()$	$(2n + s)sL_d$	$(2n + 1)s^2 L_d$	$2nsL_d$
8	$j \times AXPY()$	$(2n + s)sL_d$	$(2n + 1)s^2 L_d$	$(2n + s)sL_d$
10	$DOT()$	$(2n + s)sL_d$	$(2n + 1)s^2 L_d$	$2nsL_d$
	Total MGS	$(2\,n\,s + \mathcal{O}(s^2 m^2))L_d$	$((4n + 2)m^2 s^2$ $+(2n + 1)ms^2)L_d$	$((4n + s)m^2\,s$ $+2\,n\,m\,s)L_d$
		Other		
12	Update Hess.	$\mathcal{O}(s^2\,m^2)L_d$	$\mathcal{O}(s^2\,m^2)L_d$	$\mathcal{O}(s^2\,m^2)L_d$
13	$MAXPY()$	$(n +\; n\,s\,m)L_d$	$n\,s^2\,(m + 1)L_d$	$n\,s\,(m + 1)L_d$
15	$AXPY()$	$2nL_d$	$2nL_d$	$2nL_d$
	Total Other	$(2n + n\,s\,m$ $+\mathcal{O}(s^2\,m^2))L_d$	$(n(s^2 m + s^2 + 2)$ $+\mathcal{O}(s^2\,m^2))L_d$	$(n(s\,m + s + 2)$ $+\mathcal{O}(s^2\,m^2))L_d$

We delineate the MatMult, MGS, and Other sections of the B-LGMRES algorithm to allow direct comparison with experimental results in Chapter 4 using manual memory analysis techniques. For example, to consider the impact the multivector optimization has on the MatMult section of the B-LGMRES algorithm, we calculate $SR_{MatMult}$ and compare it with the sizes of the L1 and L2 caches. If $SR_{MatMult} \gg sizeof(cache_{L1})$, then the amount of data moved between the L1 and L2 caches would be related to the value of $WSL_{MatMult}$ for both the non-MV and MV implementations. Inspection of Table 3.1 reveals that the MV implementation has a lower $WSL_{MatMult}$ than the non-MV implementation. Let $WSL^{non-MV}_{MatMult}$ and $WSL^{MV}_{MatMult}$ indicate the working set sizes for the non-MV and MV version of the MatMult section of the B-

LGMRES algorithm, respectively. Then the ratio

$$\frac{WSL_{MatMult}^{non-MV}}{WSL_{MatMult}^{MV}}$$

provides the expected reduction in data movement in the MatMult section of code for the multivector implementation versus the standard implementation.

We find in Chapter 4 through a detailed comparison for both the MatMult and MGS sections of the B-LGMRES algorithm that data movement through the memory hierarchy is accurately predicted based on ratios of working set sizes. Further we find that reduction in data movement correlates well with reduction in execution time.

Chapter 4

Manual Memory Analysis

In this chapter, we examine how manual memory analysis is used to improve the implementation of a Krylov iterative algorithm. Manual memory analysis is performed by using the analytical expressions derived in Chapter 3 to calculate the predicted data movement for an iterative algorithm. We then compare the predicted data movement to the measured data movement for the B-LGMRES algorithm. We describe in detail the manual memory analysis process to provide a comparison to automated memory analysis described in Chapter 6. We first describe the test configuration in Section 4.1. We demonstrate the critical importance of the matrix-multivector optimization in Section 4.2. We provide results from monitoring the data movement through the memory hierarchy during the B-LGMRES solve to explain differences in execution time between different implementations of B-LGMRES in Section 4.3. We demonstrate that a reduction in execution time for our test problems correlates more strongly to reduction in data movement between levels of cache than to a reduction between main memory and cache. The following chapter represents joint work [9] and has been accepted for publication.

4.1 Test Configuration

We implemented B-LGMRES in C using PETSc 2.1.5 (Argonne National Laboratory's Portable, Extensible Toolkit for Scientific Computation) [11]. The PETSc 2.1.5

libraries contain the tools for storing a multivector in the interlaced format, which is referred to as *multi-component vectors* in the PETSc manual [11], is described in Section 2.5. PETSc provides a matrix-vector multiply routine for multiplying a matrix in AIJ format by a multi-component vector (the *MatMult* function on a matrix created with the *MatCreateMAIJ* function). The PETSc AIJ matrix storage format is equivalent to compressed sparse row storage described in Section 2.2. We also modified our local installation of PETSc to include multivector versions of the PETSc routines *VecDot*, *VecAXPY*, *VecMAXPY*, and *MatSolve*. VecDot and VecAXPY are the vector dot product and axpy routines respectively, and correspond to the DOT and AXPY notation introduced in Section 2.4. VecMAXPY adds a scaled sum of vectors to a vector. MatSolve performs a forward and back solve for use with the ILU preconditioner.

We use test problems from the University of Florida Sparse Matrix Collection [27], the Matrix Market Collection [77], and the PETSc test collection [11]. The test problems used for this chapter are listed in Table 4.1. All performance results provided are measured on a single processor of the SUN Ultra II compute platform described in detail in Table 6.1.

4.2 Multivector Optimization and B-LGMRES

Recall from Section 2.5 that the multivector interlacing scheme places corresponding elements of its constituent vectors in the same cache line. As a result, the matrix-multivector multiply routine uses a higher fraction of elements from each cache line for each access of a nonzero element of the coefficient matrix A than a non-multivector routine. Therefore, the number of floating-point operations performed per byte of data read from memory is increased. The advantages of interlacing data items in general are explained in detail in [52]. To demonstrate the benefit of the multivector optimization, we implemented B-LGMRES in PETSc both with and without multivectors. We refer to our implementation of B-LGMRES with multivectors as the MV implementation. The

Table 4.1: List of test problems together with the matrix order (n), number of nonzeros (nnz), preconditioner, and a description of the application area (if known).

	Problem	n	nnz	Preconditioner	Application Area
1	pesa	11738	79566	none	
2	epb1	14734	95053	none	heat exchanger simulation
3	memplus	17758	126150	none	digital circuit simulation
4	zhao2	33861	166453	none	electromagnetic systems
5	epb2	25288	175027	none	heat exchanger simulation
6	ohsumi	8140	1456140	none	
7	aft01	8202	125567	ILU(0)	acoustic radiation, FEM
8	memplus	17758	126150	ILU(0)	digital circuit simulation
9	arco5	35388	154166	ILU(0)	multiphase flow: oil reservoir
10	arco3	38194	241066	ILU(1)	multiphase flow: oil reservoir
11	bcircuit	68902	375558	ILUTP(.01,5,10)	digital circuit simulation
12	garon2	13535	390607	ILUTP(.01,1,10)	fluid flow, 2-D FEM
13	ex40	7740	458012	ILU(0)	3-D fluid flow (die swell)
14	epb3	84617	463625	ILU(1)	heat exchanger simulation
15	e40r3000	17281	553956	ILU(2)	2-D fluid flow (driven cavity)
16	scircuit	170998	958936	ILUTP(.01,.5,10)	digital circuit simulation
17	venkat50	62424	1717792	ILU(0)	2-D fluid flow

B-LGMRES implementation without multivectors, referred to as non-MV, represents the best non-multivector implementation possible with the tools available in PETSc 2.1.5. Both implementations were written so as to eliminate any coping of data from one data structure to another and represent a best coding effort.

Table 4.2 shows the impact of the multivector optimization on execution time by comparing the MV and non-MV implementations of B-LGMRES for 10 restart cycles. The problems in Table 4.2 are a subset of the test problems in Table 4.1 with a range of numbers of nonzeros (nnz). The percentage improvement of the MV over the non-MV implementation is given in the right-most column of Table 4.2, and, as expected, the MV implementation has the lower execution time for each problem. In fact, the MV implementation is about twice as fast as the non-MV implementation for B-LGMRES(15, 1) which has multivectors of size $s = 2$. Furthermore, the percentage improvement is independent of nnz. For the remainder of this section, we detail how the multivector optimization impacts various sections of the code, and explore the relationship between data movement through the memory hierarchy and execution time.

Table 4.2: A comparison of execution times in seconds for the MV and non-MV implementations of B-LGMRES(15, 1) for 10 restart cycles. The matrix order (n), number of nonzeros (nnz), and percentage improvement of the MV over the non-MV implementation are also listed.

				Execution Time		Relative
	Problem	n	nnz	MV	non-MV	improvement
1	pesa	11738	79566	1.7	3.5	51%
3	memplus	17758	126150	2.5	5.1	51%
10	arco3	38194	241066	17.2	30.5	44%
11	bcircuit	68902	375558	26.1	48.4	46%
13	ex40	7740	458012	9.0	20.4	51%
14	epb3	84617	463625	32.5	63.5	49%
16	scircuit	170998	958936	80.7	151.6	47%
17	venkat50	62424	1717792	61.6	113.6	46%

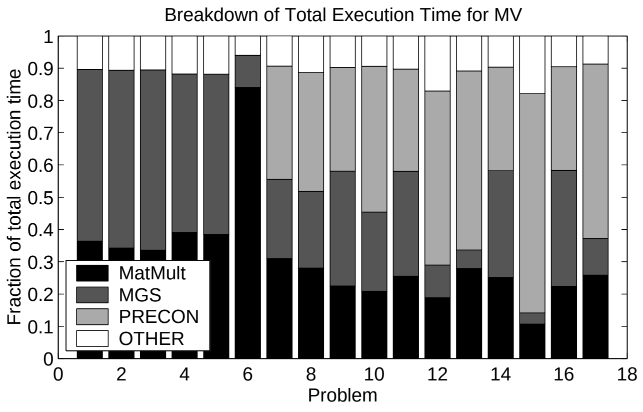


Figure 4.1: Percentage of time for each section of code of the MV implementation of B-LGMRES(15,1).

The impact of the multivector optimization is not limited to a single section of the code but rather pervades the entire algorithm. For example, because U and V in the B-LGMRES algorithm given in Figure 3.5 are multivectors, the orthogonalization in lines 6 - 9 as well as the matrix-vector multiply in line 5 require modification. Three primary sections of the B-LGMRES code are impacted by the multivector optimization: the matrix-vector multiply (MatMult), the modified Gram-Schmidt orthogonalization (MGS), and the application of the preconditioner (Precon), if required. Because the Precon section of code shows similar characteristics to the MatMult section, we only discuss the MatMult and MGS sections of code. For reference, Figure 4.1 gives the percentage of time spent in each of the three primary sections for the MV implementation of the B-LGMRES algorithm. The Other category represents the difference between the total time and the sum of times for the three sections shown. For our 17 test problems, execution time is not consistently dominated by a single section of the B-LGMRES code.

The MatMult section of the code is the matrix-vector multiply in line 5 of Figure 3.5. For the non-MV implementation, successive calls are made to the matrix-vector multiply routine for each individual vector in V_j. In contrast, the MV implementation utilizes a single call to the PETSc matrix-multivector multiply routine. The matrix-multivector multiply groups computations on the same data, which allows more floating-point operations per byte of data loaded through the memory hierarchy. The analysis in Section 3.4 indicates that we expect to reduce the amount of data moved through the memory hierarchy by a factor of s, where s is the number of vectors in the multivector. Based on the results from [9], we concentrate on a multivector of size $s = 2$ that corresponds to B-LGMRES(m, k) with $k = 1$.

As seen in Figure 4.1, the MGS section of code (lines 6-10 in Figure 3.5) often contributes significantly to the overall cost of the B-LGMRES algorithm. In fact, this section of code consumes more than 50% of the time to solution for several of the test problems. The MGS section of code requires the creation of multivector versions of the PETSc routines *VecDot* and *VecAXPY*. Following the PETSc use of *Stride* to denote multivector versions of common functions, we refer to the new versions of these routines as *VecStrideDot* and *VecStrideAXPY*, which correspond to the multiDOT() and multiAXPY() of Section 2.5, respectively. These routines represent the majority of the total time for the MGS section.

The VecStrideAXPY subroutine is an important component of the MGS section and was written using loop temporaries and loop unrolling to aid compiler optimization. The use of loop temporaries allows a compiler to identify data reuse at the register level. Loop unrolling further helps register reuse and allows different iterations of the loop to occur simultaneously. In Figure 4.2, we illustrate these optimization techniques applied to the inner loop of VecStrideAXPY. Recall that each multivector consists of two vectors of length n. Version A in Figure 4.2 is a “naive” implementation of the loop (without loop temporaries and unrolling). In Version B, we use loop temporaries, which means that all references to the *alpha* and x arrays are replaced with scalars. Version C incorporates loop unrolling; the inner loop in Version B is unrolled by a factor of 2.

```
/*------------ version A ------------------------------*/
stride=2; m=stride*n;
for (i=0; i<m; i+=stride) {
        y[i]   = y[i]   + alpha[0]*x[i] + alpha[1]*x[i+1];
        y[i+1] = y[i+1] + alpha[2]*x[i] + alpha[3]*x[i+1];}

/*------------ version B ------------------------------*/
stride=2; m=stride*n;
a0=alpha[0]; a1=alpha[1]; a2=alpha[2]; a3=alpha[3];
for (i=0; i<m; i+=stride) {
        x0=x[i]; x1=x[i+1];
        y[i]   = y[i]   + a0*x0 + a1*x1;
        y[i+1] = y[i+1] + a2*x0 + a3*x1;}

/*------------- version C -----------------------------*/
stride=2; m=stride*n;
unroll=2; step=unroll*stride; mm=m/step; rem=m\%stride
a0=alpha[0]; a1=alpha[1]; a2=alpha[2]; a3=alpha[3];
for (i=0; i<(mm*step); i+=step) {
        x0=x[i]; x1=x[i+1]; x2=x[i+2]; x3=x[i+3];
        y[i]   = y[i]   + a0*x0 + a1*x1;
        y[i+1] = y[i+1] + a2*x0 + a3*x1;
        y[i+2] = y[i+2] + a0*x2 + a1*x3;
        y[i+3] = y[i+3] + a2*x2 + a3*x3;}
if (rem)
   for (i=mm*step; i<m; i+=stride) {
        y[i]   = y[i]   + a0*x[i] + a1*x[i+1];
        y[i+1] = y[i+1] + a2*x[i] + a3*x[i+1];}

/*------------- version D -----------------------------*/
/*      Unrolled version of B where unroll=4           */
```

Figure 4.2: Code to perform a multivector AXPY operation. Successive versions add additional optimization techniques.

Version D is not shown in Figure 4.2, but is similar to C except it is unrolled by a factor of 4. In Table 4.3, the time to perform each version of the loops in Figure 4.2 as well as the functional equivalents for the non-MV implementation are provided in μsec for a subset of the test problems with a range of matrix orders. Note how in each case successive optimization either has no impact or reduces the execution time. The combination of optimizations in version D of the loop results in a 40% reduction on average in execution time for the VecStrideAXPY routine over its non-MV equivalent. Therefore, loop version D was used in all subsequent timings in this chapter. Because the MGS section can consume a large percentage of total execution time, simple optimizations such as those in VecStrideAXPY have as significant an impact on overall execution time of the solver as the use of the matrix-multivector multiply routine.

Table 4.3: Execution times in μsec for a single call to VecStrideAXPY for the non-MV implementation and MV implementations with loop versions A - D in Figure 4.2. Relative improvement of version D versus non-MV is also listed. Problems are listed in increasing order of matrix size (n).

			VecStrideAXPY time (μsec)					Relative
	Problem	n	non-MV	A	B	C	D	Improvement
13	ex40	7740	1.8	1.3	1.3	1.2	1.1	39%
3	memplus	17758	4.4	2.9	2.8	2.7	2.5	43%
11	bcircuit	68902	17.2	11.3	10.3	10.3	9.4	45%
14	epb3	84617	20.4	14.8	12.7	12.7	12.7	38%

Having described the multivector modifications to the MatMult and MGS sections, we now determine the effects of these changes by comparing the execution times for both sections of code in the non-MV and MV implementations. In Figure 4.3, the y-axis indicates the ratio of execution times for the non-MV implementation to the the MV implementation for both the MatMult and MGS sections of code. The x-axis contains the 17 test problems. A value greater than one indicates that the execution time for MV is less than that for non-MV. The MV implementation reduces the execution time of the MatMult section by a factor 1.4 to 2.7 over the non-MV implementation.

The least improvement in execution time for the MatMult section occurs for problems 4 and 9. These problems have neither the largest nor smallest n or nnz, but they do have the lowest average number of nonzeros per row: 4.9 and 4.3, respectively. However, it is unclear what impact matrix density has on the effectiveness of the multivector optimizations in general. The execution time for the MGS section shows an even greater improvement of MV over non-MV on average. In fact, the MV implementation of MGS reduces execution time by a factor of 2.3 to 2.7 over the non-MV implementation.

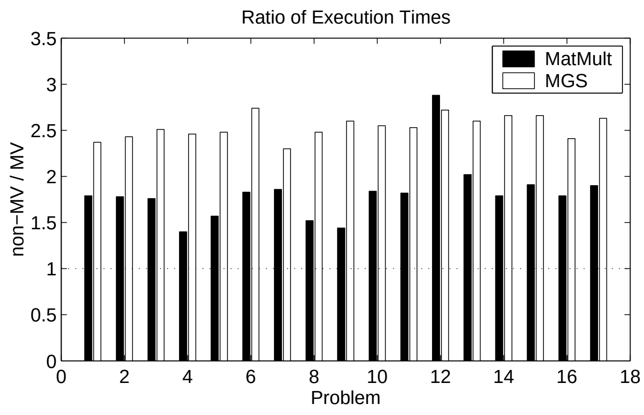


Figure 4.3: A comparison of execution time for the MatMult and MGS sections with the non-MV implementation of B-LGMRES(15,1) versus the MV implementation for 10 restart cycles.

4.3 Reducing Data Movement in B-LGMRES

For the remainder of this chapter, we explain reductions in execution time due to the multivector optimization using data from hardware performance counters. In particular, we use the Htrace utility described in Section 6.1 to measure data movement through the memory hierarchy. We denote the amount of data loaded from the memory hierarchy to the L2 cache as $Mbytes_{L2}$ and the amount loaded from the memory hierarchy to the L1 cache as $Mbytes_{L1}$.

To determine the specific source of performance gains for the MV implementation, we first examine data movement from main memory to the L2 cache for the non-MV and MV implementations. In Figure 4.4, the y-axis indicates the ratio of $Mbytes_{L2}$ for non-MV to MV for both the MatMult and MGS sections. Bars extending above 1 indicate that the non-MV implementation required greater data movement. In general, the MV implementation has a smaller $Mbytes_{L2}$ than the non-MV implementation.

For the MatMult section, we expect an approximate factor of two reduction in $Mbytes_{L2}$ for test problems with an $SR_{MatMult}$ larger than the L2 cache. Based on equations in Table 3.1 , we determined that eight test problems have $SR_{MatMult}$ larger than the 4 Mbytes L2 cache of our test system and these problems (6 and 11-17) all have ratios from 1.75 to 2.0 in Figure 4.4. Using the equations in Table 3.1 we can predict the expected values for the experimental results in Figure 4.4. In Figure 4.5 we provide the expected and measured ratios for the eight test problems. The predicted and experimental ratio of reduction in $Mbytes_{L2}$ with the MV implementation correlate well with the expected reductions in $Mbytes_{L2}$. The average relative error between the predicted and experimental ratio is 3.9%. These ratios of reduction in $Mbytes_{L2}$ with the MV implementation also correlate well with factor of 2 reductions in execution time for those problems seen in Figure 4.3. In other words, if a reduction in $Mbytes_{L2}$ is responsible for an improvement in execution time, then Figures 4.3 and 4.4 should show similar ratios for all problems.

For the MGS section of the algorithm, we expected $Mbytes_{L2}$ to be impacted only if the storage requirement SR_{MGS} is significantly greater than the L2 cache size. Only problem 16 has a $SR_{MGS} = 5.2\, Mbytes$, greater than the 4 Mbyte L2 cache size, and it shows a ratio of $Mbytes_{L2}$ for non-MV to MV of 1.4. This ratio is not the largest in Figure 4.4 and does not correlate with the improvement in execution time in Figure 4.3. In fact, other inconsistencies exist in Figure 4.4 with respect to Figure 4.3; several problems show an increase in $Mbytes_{L2}$ for the MGS and MatMult sections even though the MV implementation is faster. These inconsistencies indicate that a reduction in $Mbytes_{L2}$ does not accurately predict a reduction in execution time for most of the test problems. We show subsequently that the multivector optimization impacts a different part of the memory hierarchy for most of the test problems.

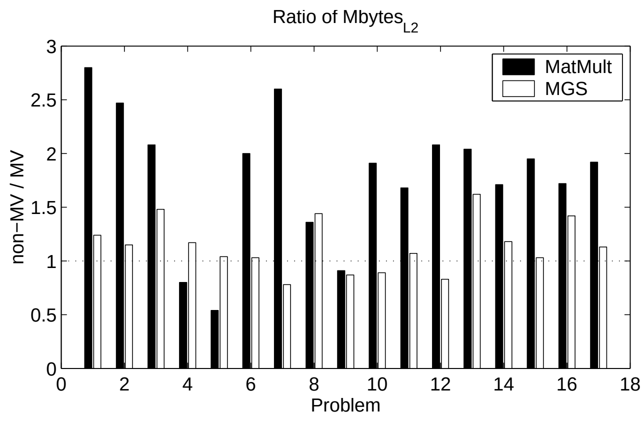


Figure 4.4: A comparison of data movement from main memory to L2 cache in the MatMult and MGS sections for the non-MV implementation of B-LGMRES(15,1) versus the MV implementation for 10 restart cycles.

Now we consider data loaded from the memory hierarchy to the L1 cache. Analogously to the plot in Figure 4.4, Figure 4.6 shows the ratio of $Mbytes_{L1}$ for the non-MV to MV implementations for the MatMult code section, while Figure 4.7 shows the same ratio for the MGS section of code. For the MatMult section of code, we expected test problems with $SR_{MatMult} \gg sizeof(cache_{L1})$ to show a reduction in $Mbytes_{L1}$. Test problem 1 has the smallest $SR_{MatMult}$ at 978 Kbytes, which is significantly larger than the 32 Kbyte L1 cache. Because all tests problems satisfied this size criterion, Figure 4.6 shows ratios of non-MV to MV that do in fact range from 1.5 to 1.9. Additionally, the predicted ratios using equations from Table 3.1 correspond well with the experimental measured ratios. The average relative error between predicted and experimental is 5.7%.

Similarly, for the MGS section of code, test problems with SR_{MGS} significantly greater than the size of the L1 cache should also show a reduction in $Mbytes_{L1}$. Be-

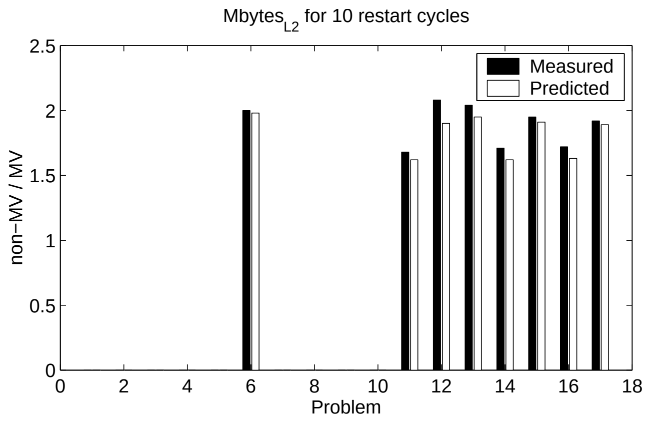


Figure 4.5: A comparison of experimental and predicted data movement from main memory to L2 cache in the MatMult section for the non-MV implementation of B-LGMRES(15,1) versus the MV implementation for 10 restart cycles.

cause the smallest SR_{MGS} is 241 Kbytes (test problem 12), all of the test problems in Figure 4.7 have experimental ratios of non-MV to MV that range from 1.6 to 2.0 and correspond well with the predicted ratios. The average relative error between predicted and experimental is 1.9%. Furthermore, the reduction in $Mbytes_{L1}$ is consistent with the reduction in execution time seen in Figure 4.3.

Now we compare reductions in total execution time and data movement for B-LGMRES. The y-axis in the top panel in Figure 4.8 indicates the execution time and $Mbytes_{L1}$ (the left and right bars, respectively) of the non-MV implementation divided by that of the MV implementation. The closer the two bars are in value for each problem, the stronger the correlation between the reduction in $Mbytes_{L1}$ and in execution time for that problem. The top panel in Figure 4.8 clearly shows the correlation between reduction in execution time and total reduction in $Mbytes_{L1}$. A similar plot in the bottom panel of Figure 4.8 for $Mbytes_{L2}$ does not show such a correlation between $Mbytes_{L2}$ for most of the test problems. Therefore, we conclude that a reduction in $Mbytes_{L1}$ is a better predictor than $Mbytes_{L2}$ for the reduction in execution time achieved by the multivector optimizations for our test problems. For much larger test

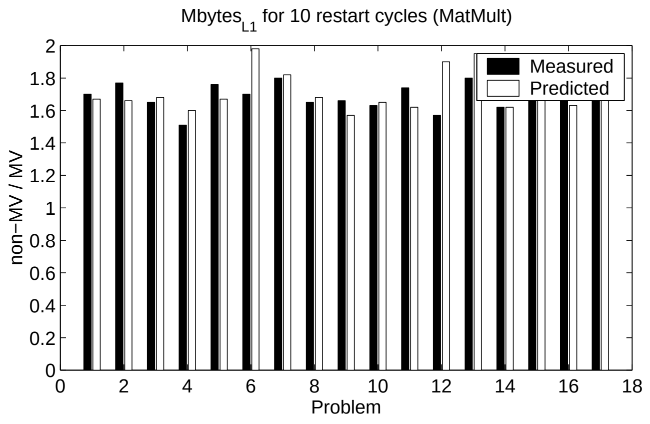


Figure 4.6: A comparison of experimental and predicted data movement from L2 cache to L1 cache in the MatMult section for the non-MV implementation of B-LGMRES(15,1) versus the MV implementation for 10 restart cycles.

problems, for example, where matrix A and SR_{MGS} are larger than the L2 cache size, we expect that the reduction in $Mbytes_{L2}$ would more strongly correlate to execution time.

The results presented in this chapter demonstrate that use of multivectors enables an efficient implementation of a block iterative solver that reduces data movement. We also demonstrate that reduction in data movement correlates well with reduction in execution time. While we demonstrate that the reduction in data movement is accurately predicted using an *a priori* memory analysis of data movement, the process is clearly limited. Because the manual memory analysis involves the comparison of various ratios, multiple implementations of a single algorithm are required. Furthermore, the complexity of the manual memory analysis process is excessive and a significant impediment to regular usage. In the next chapter we examine the development of a tool that automates memory analysis.

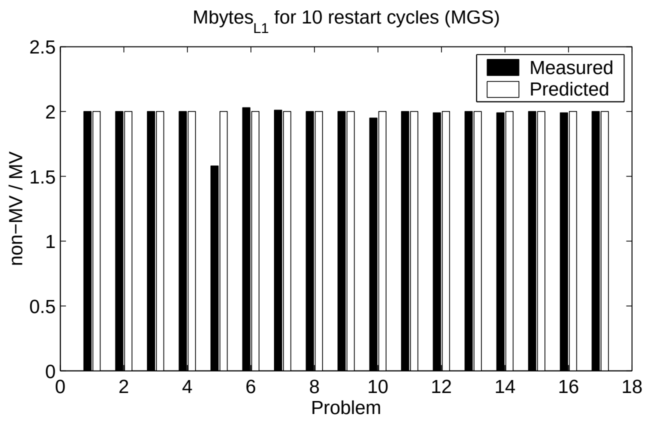


Figure 4.7: A comparison of experimental and predicted data movement from L2 cache to L1 cache in the MGS section for the non-MV implementation of B-LGMRES(15,1) versus the MV implementation for 10 restart cycles.

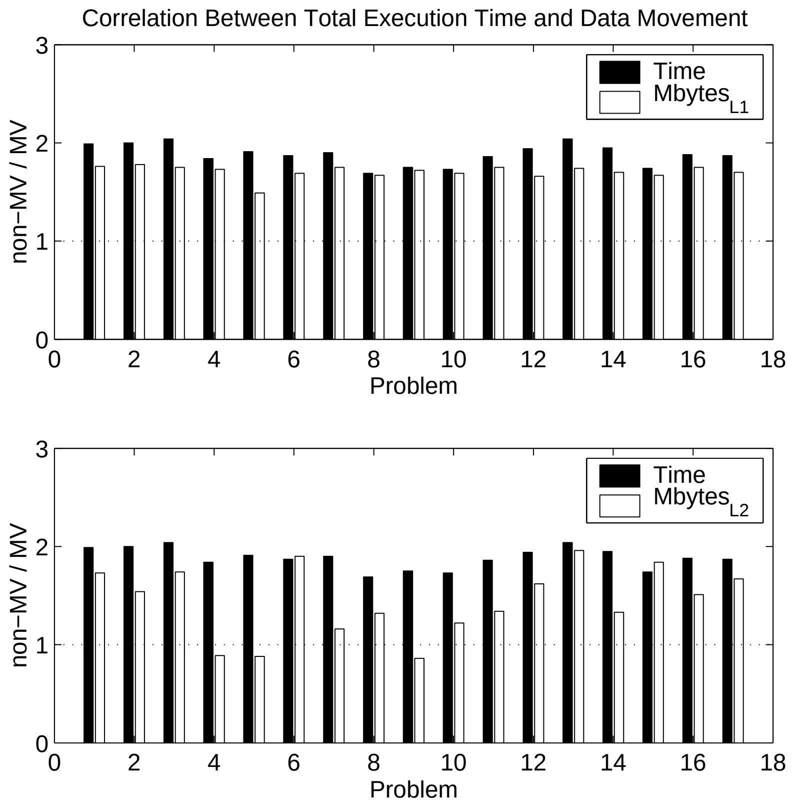


Figure 4.8: The upper panel is a comparison of data movement from L2 to L1 cache in the MatMult and MGS sections for the non-MV implementation of B-LGMRES(15,1) versus the MV implementation for 10 restart cycles. The lower panel compares data movement from main memory to L2 cache.

Chapter 5

Sparse Linear Algebra Memory Model Language Processor

The memory analysis performed on the B-LGMRES algorithm described in Chapter 4 is a time-consuming, laborious, and error-prone process. Manual memory analysis requires numerous steps that include conversion from mathematical form into an executable form, line-by-line analysis to determine storage requirements and working set sizes, and selection of the machine and a representative problem configuration. The process is too complex to perform on a regular basis. We therefore have developed the Sparse Linear Algebra Memory Model (SLAMM) language processor to automate the memory analysis process. The SLAMM language processor consists of an executable problem, utility functions, and a set of directives to control memory analysis. SLAMM inputs Matlab [69] source code and outputs equivalent source code along with statements to calculate the required data movement. We chose Matlab because it is a ubiquitous programming language in numerical analysis research.

We begin our discussion of the SLAMM language processor with the description of related work predicting code performance and translating Matlab code in Section 5.1. In the next four sections, we describe the basic computational tasks for a language processing problem. In Section 5.2, we describe lexical analysis, followed by discussion of syntactical analysis in Section 5.3. In Section 5.4, we describe several different semantic analysis tasks, followed by Section 5.5 in which we describe code transformation. The development of the SLAMM language processor was greatly simplified by the compiler construction suite Eli [50, 115] briefly described in Section 5.6. The choice of Matlab as the language on which to perform the memory analysis has several drawbacks that stem from its weak typing and ambiguous syntax. We describe the difficulties encountered during the development of SLAMM and our resulting solutions in Section 5.7. In Section 5.8, we provide specifics on the memory analysis computations. Finally, in Section 5.9, we provide a operational description of the SLAMM language processor.

5.1 Related Work

We begin by discussing related efforts to the SLAMM language processor. In Section 5.1.1 we describe work to model the impact of scientific applications on the memory hierarchy using compiler techniques. These techniques use a combination of source code analysis and instrumentation to predict both execution time and cache miss-rates. In Section 5.1.2, we describe efforts to translate Matlab code into other representations.

5.1.1 Automated Performance Prediction

Cascaval [19] uses the Polaris compiler to predict the performance of serial FORTRAN and C codes. The input code is analyzed by the Polaris compiler, which adds analytical expressions that predictions execution time and cache miss-rates. The predictions of execution time and cache miss-rates, which are within an average 20% error, are used by the Polaris compiler to improve code optimization.

Fahringer [39] developed the Parameter based Performance Prediction Tool (P^3T) which is integrated with Vienna Fortran compiler. P^3T automatically estimates cache performance for both sequential and data parallel FORTRAN using simple cache model.

The PAMELA compiler [111, 112] adds symbolic cost expressions for data parallel problems, which in combination with a simple machine model, allows it to predict execution time to within 10% error.

Unlike the related work, SLAMM does not analyze FORTRAN or C source code but rather analyzes Matlab code. The use of Matlab allows the analysis of an algorithm before implementation. SLAMM therefore provides feedback to the algorithm developer during design phase. Further, an analysis of a Matlab version of an algorithm is not biased by the implementation details present in a FORTRAN or C implementation. A SLAMM analysis predicts the minimum possible data movement needed by an algorithm versus the required data movement.

5.1.2 Compiling Matlab Code

There have been many efforts to convert or compile Matlab code into other representations [12, 29, 57, 86, 60]. The MCC compiler [57] supplied with Matlab and the FALCON compiler [29] are the most relevant to the SLAMM language processor efforts. The MCC compiler converts Matlab code into C code, which can then be compiled into a stand-alone executable program. Examination of the C code generated by MCC reveals that it converts each Matlab statement into a separate subroutine call. While the overhead of this technique for large linear algebra operations like a matrix-matrix multiply may be minimal; the overhead for a single scalar assignment is excessive. The subroutine approach is used to address several of the difficulties of compiling Matlab code that stem from its weak typing. A language is weakly typed if it does not require the programmer to specify the type, shape, or size of variables. Matlab's weak typing makes it both easy to use and difficult to compile.

The subroutine approach of the MCC compiler addresses Matlab's weak typing by checking type, shape, and size, and if necessary, changing the configuration of the variable during each subroutine call. The FALCON compiler, which converts Matlab into FORTRAN90 code, performs much more extensive type analysis during the compilation phase. FALCON uses several techniques to address Matlab's weak typing, including multiple forward type analysis passes and a value prediction phase [30].

5.2 Lexical Analysis

We begin our discussion of the main computational phases of a compiler with lexical analysis. Both lexical and syntactical analysis consist of processing or parsing the input text. In particular, lexical analysis converts input text into a collection of tokens. A **token** is a string of one or more characters with a particular format specified by the lexical specification. Character strings read in from the input text by a *scanner* are classified into tokens. Valid tokens are passed to the syntactical processor whose operation is described in Section 5.3. We illustrate the discussion of lexical analysis with Figure 5.1, a simple C code with a collection of coding errors.

```
#include <stdio.h>                 /* line 1 */
  main()                           /* line 2 */
{                                  /* line 3 */
    int index, 2ii;                /* line 4 */
    double best                    /* line 5 */
    double grav, result3;          /* line 6 */
    grav = 1.0g-5;                 /* line 7 */
    result3 = grav*be@t + y[2ii];  /* line 8 */
}                                  /* line 9 */
```

Figure 5.1: C code with a number of coding errors.

Because lexical analysis is concerned with the legality of input tokens, we examine the illegal tokens, or lexical errors, in Figure 5.1. The first lexical error occurs in line 4 of Figure 5.1 with the declaration of the int variables “index” and “2ii”. **Identifiers** are used as names of variables or functions. In C, identifiers consist of one or more letters or underscores followed by one or more numerals, letters, or underscores. The token “2ii” does not begin with either a letter or an underscore and is therefore an invalid identifier token. Identifier token “be@t” in line 8 is invalid because of the presence of the @ symbol. Lexical analysis is not limited to verifying the form of identifiers. Another lexical error occurs in line 7 because the token “1.0g-5” is not a valid form of floating-point constant. The lexical analysis scanner either generates an error for characters that are not present in the lexical specification in the case of the “@” character or passes partial tokens to the syntactical processor for further analysis.

5.3 Syntactical Analysis

Syntactical analysis, like lexical analysis, is a component of the processing of input text. While the lexical specification of a language is a description of all legal tokens, the syntactical specification describes the order in which the tokens may appear in the input text. The syntactical analysis processor forms a parse, a collection of input tokens. A decision is made by the syntactical analysis processor to either continue the parse or reduce the parse to concrete syntax tree representation [84]. The decision is based on the current parse and the next token in the input text, the **lookahead** token. If the lookahead token is not of an expected type, a syntax error is generated. The first syntax error, an omission of a semicolon, occurs on line 5 in Figure 5.1 in the declaration of the identifier “best”. The rules for token order are specified by a context-free grammar. One possible context-free grammar, or concrete syntax specification, and the associated syntax tree for identifier declaration is provided in Figure 5.2.

A concrete syntax tree is comprised of a collection of terminal, non-terminal, and constant nodes. A terminal node, indicated by a circle in Figure 5.2, and a constant, indicated by a half circle, are leaves in the syntax tree. A non-terminal node, indicated by a rectangle in Figure 5.2, consists of a collection of other non-terminal, terminal, or constant nodes. The node *Ident* in Figure 5.2, which corresponds to the identifier token described in Section 5.2, is an example of a terminal node. A constant is one or more characters with a specific meaning in the language and is delimited by apostrophes in the concrete syntax specification. The root node of Figure 5.2, denoted *Declaration*, is a non-terminal node comprised of two non-terminal nodes: *TypeDenoter* and *IdentList*, followed by a constant. It is apparent that while line 6 of Figure 5.1 matches the form

```
Declaration:  TypeDenoter IdentList ';' .
TypeDenoter: 'int' / 'float' / 'double' .
IdentList: Ident / IdentList ',' Ident .
```

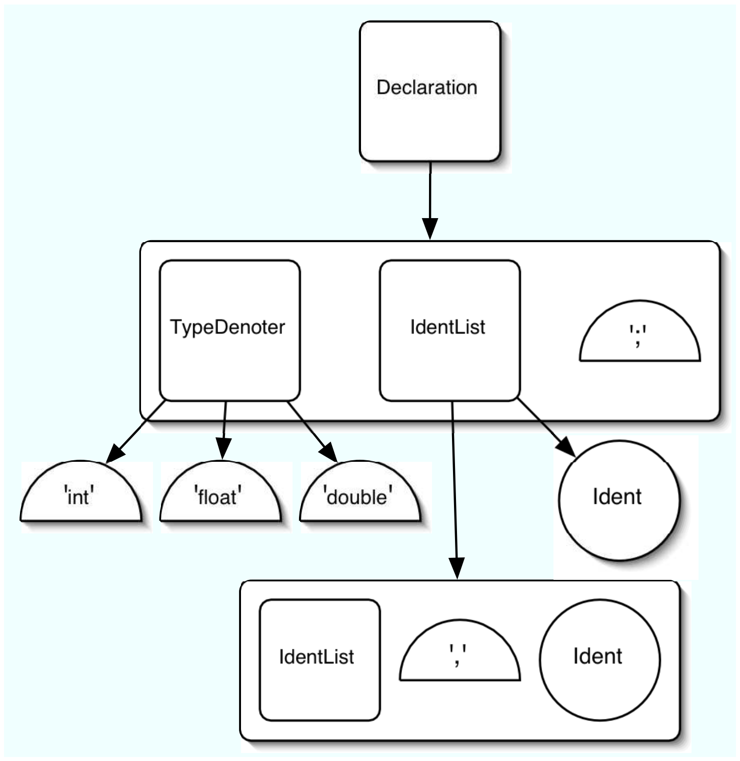


Figure 5.2: A simplified concrete syntax for variable declaration in C in both specification and tree form.

of *Declaration* node, line 5 of Figure 5.1 does not. The missing semicolon in line 5 of Figure 5.1 is a syntax error.

Based on the concrete syntax tree specification, the lexical scanners and syntactical processors combine to convert the input text into an **abstract syntax tree**. Computations on the abstract syntax tree, or semantic analysis, are described in the next section.

5.4 Semantic Analysis

Recall that the abstract syntax tree is constructed by the lexical scanners and syntactical processors based on the content of input text. The semantic analysis phase is characterized by computations on the nodes of the abstract syntax tree. The exact nature of the computations on the abstract syntax tree varies widely and is dependent on the purpose of the application. For example, consider several different applications: a pretty printer, a lint processor, and a compiler that converts C code into assembly language. The pretty printer application, which transforms input text into output text with particular formatting rules, requires only an unparser, and not a semantic analysis phase. An unparser transforms the abstract syntax tree into textual output as described in Section 5.5. A lint application, which checks the validity of the input text, requires a semantic analysis component, but not an unparser. A C to assembly language compiler requires both extensive semantic analysis and unparser components. We describe several common computational tasks on abstract syntax trees: name analysis, type analysis, and tree parsing.

5.4.1 Name Analysis

Name analysis is concerned with determining the meaning of identifiers and whether particular language properties hold. Determining the **scope** or extent of an identifier is a typical name analysis problem. For example, in C, the placement of a variable's declaration determines its scope. A variable declared outside a function has a global scope, while a variable declared inside a function has a local scope. Each occurrence of an identifier in the input text is classified as either a define or a use occurrence. An occurrence of an identifier within a declaration statement is a define occurrence, while all other occurrences are of type use. Note that the identifier “grav” in Figure 5.1 occurs three times. The occurrences of “grav” in line 6 of Figure 5.1, which declares “grav” to be a variable of type “double”, is a define occurrence. The occurrence of identifier “grav” in lines 7 and 8 of Figure 5.1 represents use occurrences. The proper classification of each occurrence and its scope allows the name analysis component to determine whether an identifier is defined. Clearly the identifier “grav” is defined. The identifier “y” in line 8 of Figure 5.1 is never defined and would therefore represent a semantic error.

5.4.2 Type Analysis

The task of type analysis, like name analysis, performs computations on the nodes of the abstract syntax tree to verify certain properties of the input text and provide more information for subsequent phases of the compiler. Type analysis focuses on determining the type of an identifier and the implications of its type on additional computations. The type of identifier “result3” in Figure 5.1 is “double”, as are the identifiers “best” and “grav”. The type of “y” is unknown because it is never defined and is subsequently ignored. A typical type analysis task verifies that the “*” operator is defined for operands of type “double”. If the operator is not defined, type analysis determines whether the operand can be converted or coerced. In addition to verifying certain language properties, type analysis also provides information for subsequent compiler phases. For the C to assembly language compiler example, type analysis determines that a floating-point instruction should be generated for line 8 of Figure 5.1 versus an integer instruction.

5.4.3 Tree Parsing

Tree parsing is a semantic analysis task that involves the transformation of one abstract syntax tree into another. The creation of a second abstract syntax tree is useful when a particular computation is not well suited to the initial abstract syntax tree. For example, consider a C to assembly language compiler. The structure of the input language C differs significantly from the structure of the generated assembly code. Tree parsing converts the input abstract syntax tree into a tree with a structure similar to the assembly code. This transformation simplifies the computations necessary to generate assembly code. Another application of tree parsing that addresses ambiguities with input syntax is described in detail in Section 5.7.3.

5.5 Transformation

The final phase of a compiler is the transformation of the abstract syntax tree representation into the output text. The output language may be the same as the input, as in the case of the pretty printer application described in Section 5.4, or it may be a separate language as in the C compiler example. The unparser component of a compiler performs the transformation from tree form into a structured output. The form of the text output for each node in the abstract syntax tree is based on the computations performed during the semantic analysis phase.

5.6 Eli Compiler Construction Suite

The Eli compiler construction suite [50, 115] is a collection of tools that solve particular tasks necessary for the construction of a compiler. The tools in the suite are accessible through the use of several high-level specification languages. Using specification languages reduces development time and simplifies maintenance of the resulting compiler [96]. The specifications are converted by the toolkit into ANSI C code that is then compiled into an executable. Eli differs from products like lex and yacc [64] in that it provides tools or modules for all phases of compiler development, not just the lexical and syntactic analysis phases. In addition to providing tools for lexical and syntactic analysis, Eli also provides tools to perform semantic analysis and transformation into output text. Further, all tools are integrated into a single development environment.

While Eli provides numerous languages and their associated translators to describe particular compiler tasks, perhaps the most important is the LIDO language. **LIDO** is the specification language in which all tree computations are described in Eli. We describe several of LIDO's fundamental programming constructs to clarify further discussion of tree computations. At the highest level of abstraction is the symbol construct. There are two types of symbols: a tree symbol, which represents a node in the abstract syntax tree, and a class symbol, which is a computational method. Both types of symbols contain a collection of objects. Both tree and class symbols can inherit a class symbol and all associated objects. Objects consist of attributes, properties, and computations. Attributes and properties are data structures on which computations occur. Attributes apply to all nodes in a tree, while properties only apply to particular nodes. Eli also provides centralized storage of properties.

Properties are data structures that are associated with particular nodes in the abstract syntax tree. A fundamental property in Eli is the symbol id, an integer that corresponds to a unique character string in the string table. A symbol id is generated by the lexical scanner for each valid token added to the string table. When an identifier is defined, the symbol id corresponding to the identifier is entered into the symbol table and the **symbol key** is returned. A symbol key is a data structure that associates a defined identifier with a scope.

A scope is the extent over which an identifier is defined. For example, consider the case where two separate functions use a locally defined variable "tmp". Because each variable "tmp" has a scope local to its own function, the two declarations of "tmp" refer to different variables. While both “tmp” variables share a single symbol id, each has an unique symbol key because of its unique scope. Properties are not limited to a particular node in the abstract syntax tree. Eli’s centralized storage scheme for properties is indexed by the symbol keys. The property storage module is used to accumulate information about the input text. For example, the number of times a symbol appears in a particular context is determined by incrementing a counter associated with its symbol key.

While attributes are data structures that are defined for all nodes in the abstract syntax tree, they are only associated with nodes where they are used. A common application of attributes is the generation of output text. Support for the generation of output text in Eli is provided by associating a structure of type PTGNode to every node of the abstract syntax tree. The PTGNode structure contains a string of characters and a pointer to a formatting function. The form of the PTGNode structure for each non-terminal node is determined by an attribute computation on the PTGNodes of its children nodes. The PTGNode structure of the root node of the abstract syntax tree contains the complete output text.

5.7 Matlab-Specific Difficulties

The weak typing and flexible syntax features of the Matlab language complicates compiler development. The weak typing difficulties described in Section 5.1.2 are troublesome only if Matlab is converted into a language with stronger typing like C or FORTRAN. We avoid the Matlab type analysis problem in the SLAMM language processor by not translating the input text into a stronger-typed language. Instead we transform the original Matlab code into a second Matlab code with the addition of the statements necessary to perform the memory analysis. Whenever possible, we use Matlab-specific features to simplify the development and execution of the SLAMM language processor. Unfortunately, the problems created by its flexible syntax are unavoidable and are de-

```
b = [1 - 2 3],
b2 = [1 - 2, 3],
c = [1 -2 3],
c2 =[1, -2, 3],
```

Figure 5.3: Matlab code that illustrates the use of invisible commas

scribed in detail in the following sections. We first describe the difficulties encountered in the lexical and syntactical analysis of Matlab in Sections 5.7.1 and 5.7.2. In Section 5.7.3, we describe the difficulties encountered in the name and type analysis phase and how tree parsing is used to address the syntactical ambiguities.

5.7.1 Invisible Commas

A prime example of Matlab's flexible syntax is seen in its use of commas. In Matlab, commas are used to separate a list of items. For example, commas are used to separate the indices of an array or the arguments to a function. Commas are also used to separate individual Matlab statements for which the result should be printed to standard output.

Further, Matlab supports a concept called invisible commas, where in certain contexts, other tokens are syntactically equivalent to commas. For example, in the case of statement separation, a newline character or several blank characters and a newline are both equivalent to a comma. Another example of invisible comma usage in Matlab is provided in Figure 5.3. Figure 5.3 is Matlab code that sets the variables b, $b2$, c, and $c2$ to be arrays of integers where the variable $b2$ is equal to b and $c2$ is equal to c.

We examine the meaning of blank space in the case of the assignment of variable c. Both occurrences of blank spaces, which are usually irrelevant, are syntactically equivalent to a comma within the context of the brackets in line 3 of Figure 5.3. For the assignment of the variable b in line 1 of Figure 5.3, the blank space between the brackets has multiple meanings. In this case, the first two blank spaces are irrelevant, while the third is equivalent to a comma. The subtle and context-dependent meaning of blank space in Matlab creates an invisible comma classification problem.

One possible way to address the invisible comma problem illustrated in Figure 5.3 is to define blank space as a terminal node in the abstract grammar. However, the addition of such a node and the required associated context would significantly increase the complexity of the abstract syntax tree. Instead of pushing the added complexity into the syntax tree, we choose to address it by including additional token definitions during lexical processing. Consider four different types of tokens: *Blank*, *Neg*, *NegSpace*, and *Integer*. Token *Blank* is defined as one or more blank spaces; *Neg* is defined as a minus sign; *NegSpace* is defined as zero or more blank spaces, a minus sign, then one or more blank spaces; while *Integer* is defined as one or more integer characters. Consider the order of tokens that are classified when parsing within the brackets for the assignment of *b* in Figure 5.3. The first character "1" matches to an *Integer* token, while the next set of characters " - " matches to a *NegSpace* token. Note that in the case where multiple tokens match, the token classification with the greatest length is used. The next three tokens are *Integer*, *Blank*, and *Integer*. With the match of the *Blank* token, a state variable is checked to determine context. If it is enclosed in brackets, the *Blank* is accepted by the abstract syntax as a comma. If the *Blank* token is not contained within brackets, it is ignored.

A similar examination of the order of tokens classified when parsing within the brackets for the assignment of *c* in Figure 5.3 reveals a key difference. The first character "1" matches to an *Integer* token, while the next character matches to a *Blank*, followed by a *Neg* token. The *NegSpace* token is not matched because in this case one or more blank spaces do not trail the minus sign. The remaining tokens match as before. Because both of the *Blank* tokens appear within brackets, they are properly classified as commas. The use of additional token types during lexical analysis prevents the complexity of invisible commas from affecting the form of the abstract syntax tree.

```
b = a'
msg = 'This is a string constant'
```

Figure 5.4: Matlab code that illustrates the use of both transpose operators and string constants.

5.7.2 Transpose Operator

In the previous section, we illustrated how the use of multiple different tokens to represent a single component of the Matlab grammar causes difficulty for syntactical analysis. We next examine the case where a single character, the apostrophe, is used in semantically different constructs. A single apostrophe applied as a postfix to an identifier is the Matlab transpose operator. String variables in Matlab, like Pascal, are delimited by apostrophes. We illustrate the difficulty the dual use of the apostrophe creates by examining a Matlab code in Figure 5.4 that includes both the transpose operator and string variables.

It should be noted that the processing of a character string token occurs differently than for all other tokens. When a beginning delimiter is encountered, the text scanner searches forward in the input text for the end delimiter, accepting all intervening characters. A text scanner would therefore incorrectly interpret the transpose operator in the first line of Figure 5.4 as the beginning delimiter of a character string that concludes at the first apostrophe on the second line of Figure 5.4. A series of identifier tokens would next be classified followed by an unterminated character string. The resulting incorrectly constructed collection of input tokens would result in a syntax error. To differentiate between a character string deliminator and a transpose operator, we use a repair function.

The repair function "Reparatur" in Eli allows the reclassification of an input token. Recall from Section 5.2 that once a token is classified, it is passed to the syntactical analysis processor, which either accepts it into the abstract syntax tree or rejects it and

```
[a, b],
[a, b] = qr(A),
```

Figure 5.5: Matlab code that illustrates the multiple uses of brackets.

generates an error message. The repair function allows a token rejected by the syntactical processor to be reclassified and the pointer to the input text to be reset. The text scanner resumes input, and the reclassified tokens are passed again to the syntactical analysis processor.

The apostrophe identification problem is addressed by classifying all apostrophe characters as transpose operators with a secondary or repair classification as a string constant. It is clear that the first apostrophe in line 1 of Figure 5.4 is correctly classified as a transpose operator. The first apostrophe in line 2 of Figure 5.4 does not adhere to the syntax rules for a transpose operator and results in a call to the repair function. The subsequent reclassification of the token as the beginning deliminator of a character string is correct and results in the formation of a correct abstract syntax tree.

5.7.3 Classification of Identifiers

Recall from Section 5.7.1 that we addressed the invisible comma problem by moving computations typically performed at a later compiler phase into an earlier phase where we had sufficient information for proper classification. We next describe a technique that delays the construction of certain components of the abstract syntax tree to a later compiler phase. The delayed construction of certain nodes in the abstract syntax tree allows the correct classification of *identifiers*. As in previous sections, we begin by providing a motivating example.

In Section 5.7.1, we illustrate one possible use of brackets in Matlab as an array constructor. Brackets are also used in Matlab to delimit the output arguments of function calls. Both uses of brackets are illustrated in Figure 5.5.

The Matlab statement, in line 1 of Figure 5.5 requests that an array composed of identifiers “a” and “b” be printed to standard output. Line 2 of Figure 5.5 indicates that the identifiers “a” and “b” are the output arguments of the function “qr”. While the first six characters of each line are identical, the occurrence of identifiers “a” and “b” represent different semantic meanings in each line, which must be expressed differently in the abstract syntax tree. In line 1 of Figure 5.5, the identifiers “a” and “b” are a use occurrence, while in line 2 they are a define occurrence. Further, note that in order for the identifier token “a” to be properly accepted by the syntactical processor, several additional tokens must be scanned and classified, including the last comma in line 1 and the equal sign in line 2. However, the additional required tokens are not possible because tokens are classified and accepted into the abstract syntax tree based on the previous accepted tokens and a single lookahead token. Matlab’s use of brackets as both array constructors and to delineate the output arguments of function calls creates difficulties in the proper semantic analysis of identifiers.

We address the identifier classification problem by applying the tree parsing technique described in Section 5.4.3. While tree parsing is typically used to construct entirely new abstract syntax trees, we only apply it to selected pieces of the existing tree. We choose to apply tree parsing sparingly for several reasons. First, the output language is the same as the input, so we have no fundamental need to create an additional syntax tree. Second, the addition of a nearly identical abstract syntax tree would result in an increase in the size of the necessary compiler specifications and a decrease in its reliability and maintainability.

Our approach in cases in which the classification of identifiers is ambiguous, is to accept the tokens as unknown identifier nodes. Once the initial abstract syntax tree has been constructed, we parse the particular branches of the tree that contain unknown identifiers to create a new tree branch. The new tree branch containing the proper classification of the unknown identifiers is grafted into the original tree. All tree computations necessary for memory analysis and formation of the transformed output text occur on the newly grafted branches.

5.8 Memory Analysis Computations

The techniques described in Section 5.7 address particular difficulties in the syntax of Matlab. These techniques combine to provide an accurate representation of the input text in abstract syntax tree form on which to base further computations. Additional tree computations are primarily concerned with calculating memory analysis statistics by using a combination of static and dynamic techniques. The SLAMM language processor performs static analysis of the input text to generate additional code blocks for output generation. The Matlab interpreter subsequently executes the transformed code to complete the memory analysis. Memory analysis statistics include both the number of times a variable is loaded from the memory hierarchy and the total storage requirement for all variables.

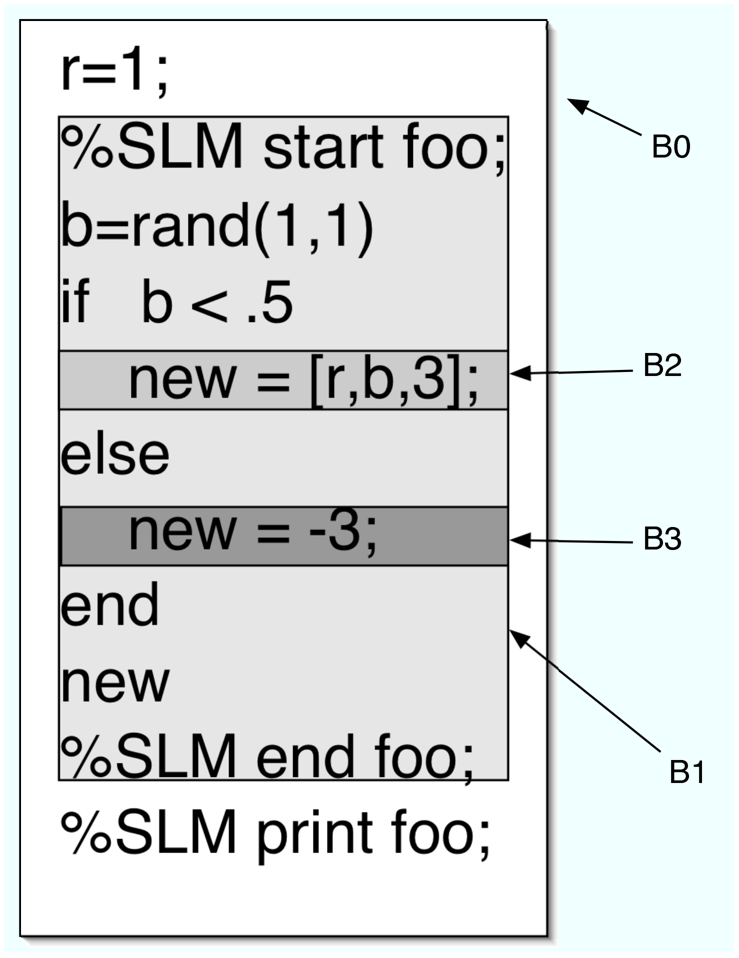
Section 5.8.1 describes how the memory analysis statistics are based on inclusive and exclusive counts of identifiers. While identifier counts provide a basis for the memory analysis, several corrections must be applied to the base identifier counts. The corrections necessary to improve the accuracy of the memory analysis are described in Section 5.8.2. In Section 5.8.3, we describe the various code blocks that the SLAMM language processor generates. In Section 5.8.4, we describe some code transformations necessary to provide function support. Finally, in Section 5.8.5, we describe the memory analysis output.

5.8.1 Inclusive and Exclusive Counting

Recall the concept of scope, which is the extent or range over which an identifier is known. In Section 5.4.1, we described how scope is used to determine whether an identifier is properly defined within a function or collection of functions. While this is the most common use of scope, it is not its only application. We consider using scope over a much smaller range. For purposes of this discussion, we define a piece of code with unique scope, even if it is as small a single statement, to be a **body** of code. Note that because symbol keys are properties of a node, we can maintain the associated keys for both the function and body applications of scope. Use of body level scoping allows a fine-grain counting of particular program characteristics, providing the basis for all memory analysis computations.

A Matlab code that contains multiple scopes is provided in Figure 5.6. Note that we have indicated the unique scopes with the labels *B0* to *B3*. SLAMM directives, indicated by the prefix "%SLM", are also included. The directives "%SLM start foo;" and "%SLM end foo;" delimit and provide a symbolic name, in this case "foo", to a body of Matlab code on which to perform memory analysis. The third directive in Figure 5.6 "%SLM print foo;" requests the printing of the memory analysis for body "foo". A complete description of all SLAMM directives is provided in Section 5.9. Note that the directives create a separate nested scope *B1* that is contained in the original root scope *B0*. The "if" statement also defines two additional bodies, *B2* and *B3*, nested within body *B1*. The call to the Matlab function $rand(1,1)$ generates a single random floating-point value between 0.0 and 1.0.

We next describe inclusive and exclusive counts for identifiers. An **inclusive identifier count** is the number of times an identifier occurs in the current scope and all included scopes. For example, the inclusive count for identifier "r" in the scope B1 is equal to 1 because it occurs in the included scope B2. An **exclusive identifier count** is the number of times an identifier occurs in the current scope. The exclusive count for identifier "r" in the scope B1 is equal to 0 because "r" occurs only in the B0 and B2 scopes. Both types of identifier counts are calculated through the use of symbol keys.



```
r=1;
%SLM start foo;
b=rand(1,1)
if   b < .5
     new = [r,b,3];
else
     new = -3;
end
new
%SLM end foo;
%SLM print foo;
```

Figure 5.6: Matlab code that illustrates multiple scopes.

```
n = size(A,1);
m = size(r_m1'*r);
alpha = r'*r;
r_m1 = r;
r = A * w;
```

Figure 5.7: Matlab code that illustrates the need for corrections to the base identifier counts.

Because each scope and symbol id combination has a unique key, we easily determine exclusive counts using a simple increment of an integer counter. Inclusive counts require a somewhat subtler approach because the counters for both the current and all parent scopes must be incremented. The total amount of storage required for a particular body of code is determined by summing the storage requirement for all variables with inclusive identifier counts greater or equal to one. However, the exclusive identifier counts provide only an estimate of the number of times a variable is loaded from the memory hierarchy. Several corrections to the identifier counts described in the next section are necessary to accurately characterize data movement.

5.8.2 Corrections to Identifier Counts

The goal of the SLAMM language processor is to predict the data movement for an efficiently implemented algorithm in a compiled language. While the use of identifier counts as the basis of the memory analysis is a good approximation, as described in the previous section, it is not always accurate. In particular, the occurrence of an identifier does not necessarily indicate that the corresponding variable must be loaded from the memory hierarchy. Fortunately, it is possible to improve the accuracy of the base identifier counts through a series of corrections. These corrections represent a translation from the literal analysis of the input Matlab code to an estimate of how the code would be efficiently implemented in a compiled language. A small Matlab code is provided in Figure 5.7 to motivate the discussion, where $r, r\_m1, w \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$.

The first line of Figure 5.7 uses the built-in Matlab function *size* to set the variable n to be extent of the first dimension of the matrix A. While line 1 of Figure 5.7 represents a use of identifier A, its implementation does not require access to the entire matrix “A”. It is possible to provide the functionality necessary for line 1 in Figure 5.7 in a compiled language implementation by accessing a single integer variable. The **function call correction** addresses the counting mismatch by decrementing the counts for those identifiers that occur within a function call argument list. We are careful not to decrement the count for those identifiers that occur in an expression in an argument list. For example, the function call correction does not apply to those identifiers “r_m1” and “r” in line 2 of Figure 5.7.

The calculation of the dot product “alpha” in line 3 of Figure 5.7 represents the need for another type of correction. In this case, the identifier “r” occurs twice in a single expression. However, the vector r is only loaded once from the memory hierarchy. The duplicate occurrence of “r” represents cache reuse that cannot be ignored in the memory analysis calculations. To address cache reuse, we decrement the identifier counts for any identifiers that occur multiple times within a single statement. The **duplicate correction** does not address the possibility of cache reuse between multiple statements.

Line 4 of Figure 5.7 represents a copy of the vector r to vector r_m1. Memory copies are necessary in Matlab for renaming purposes because Matlab lacks a pointer construct. Single variable assignments are implemented as either a memory copy or a pointer assignment in a compiled language. We assume that for an efficiently implement algorithm, variables with a large storage requirement, for example vectors, use pointer assignments. Variables with a small storage requirement, for example a single floating-point value, use memory copies. Because the cost of memory copy for small variables is insignificant, we assume for the purposes of memory analysis that the assignment of one variable to another is a pointer assignment. The **copy correction** decrements the identifier counts to properly address pointer assignment.

Line 5 of Figure 5.7 represents the need for a different form of correction. The previous corrections involved decrementing counts to properly account for particular identifier occurrences. Proper memory analysis of the statement on line 5 of Figure 5.7 requires additional information about the types of the “A” and “w” identifiers. For example, if “A” is a sparse matrix, the amount of data movement, which is examined in Section 6.2.3, is potentially larger than the sum of the storage requirements of “A” and “w”. However, type-dependent computations are not possible for the SLAMM language processor because it has no knowledge of type. Type information is only available to the Matlab interpreter. We address our lack of type information by transforming certain operators into function calls. We then apply the standard techniques developed for functions described in Section 5.8.4. For example, to apply the **special operator correction** to line 5 of Figure 5.7, we transform the expression $r = Aw$ to the equivalent $r = mtimes(A, w)$. A profiled version of $mtimes$, described in Section 5.8.4, allows the proper calculation of data movement based on the determination of type at runtime.

5.8.3 SLAMM Output

The SLAMM language processor generates output text, based on the results of the tree computations described in Sections 5.8.1 and 5.8.2. The output text typically contains the input text with the addition of various code blocks that calculate the memory analysis. When the input text contains calls to certain functions, additional code transformation is required. We next describe the case where no input code transformation is required. This is followed by the transformation case in Section 5.8.4.

The SLAMM language processor generates several different types of additional code blocks. We illustrate the types and placement of the code blocks that are generated in Figure 5.8 if the Matlab code in Figure 5.6 is processed by the SLAMM language processor. The oblong objects in Figure 5.8 represent code blocks added by the language processor. We describe each type of code block in detail.

A code block of type **Header** is inserted as a prefix to the original Matlab code. The Header code block contains the definitions and initialization of Matlab structures used to accumulate the information for each scope and identifier. A code block of type **SizeOf** typically contains a single call to the Matlab *whos* function. The *whos* function returns a structure that describes the type, extent, and required memory storage of its argument. A SizeOf block is generated by SLAMM each time an identifier is assigned. When an identifier requires definition through a SLAMM directive, a condition not present in Figure 5.8, a SizeOf block is generated before the first use of the identifier. If the input Matlab code includes access to a subsection of an array, a condition also not illustrated in Figure 5.8, an additional statement in the SizeOf code block is necessary. Because the *whos* function does not accept subscripted arrays as arguments, the array subsection is first copied to a temporary variable whose characteristics are subsequently determined by the *whos* function.

We next describe the contents of an **exclusive memory analysis** code block. In particular we examine in detail the exclusive memory analysis block for the B2 body in Figure 5.8. The generated Matlab code for body B2 is provided in Figure 5.9. Note that the SizeOf block for the assignment of the identifier "new" is included for completeness and that the "." represents an access to a component of a Matlab structure. The exclusive memory analysis code block in Figure 5.9 includes calculations for the size of variables loaded from the memory hierarchy, as indicated by the structure field *wsl*, the size of variables stored to the memory hierarchy *wss*, and the total storage size *sr*. The *bytes* field of the structure created by the *whos* function is used to calculate the proper values for *wsl*, *wss*, and *sr*. The string "ifLn5" is SLAMM's name for the B2 body.

While all bodies of code require an exclusive memory analysis code block, only those bodies that contain other bodies require an **inclusive memory analysis** code

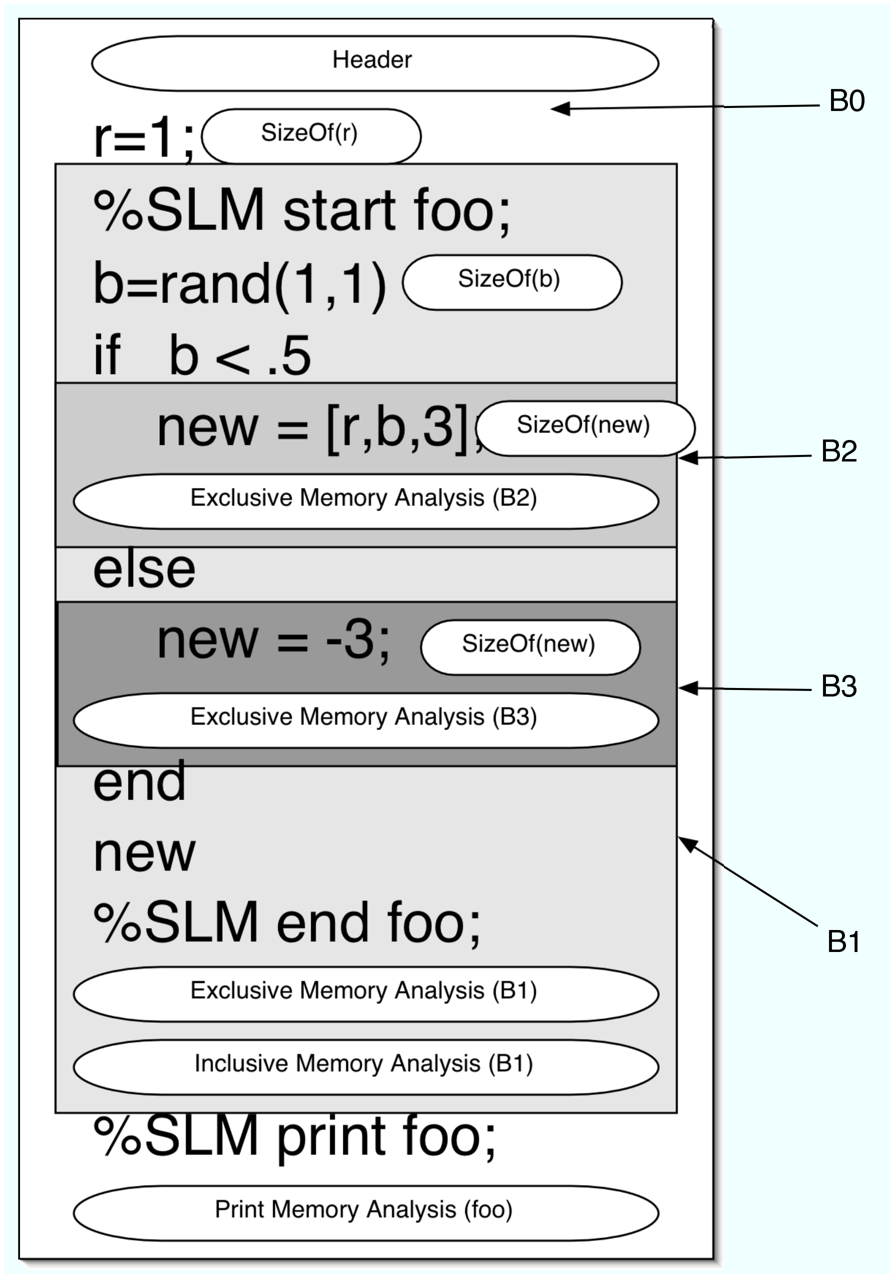


Figure 5.8: A diagram of SLAMM output using the Matlab code in Figure 5.6 as input.

```
new =[r,b,3];
[slm_new] = whos('new');
%---------------SLM memory Analysis ----------------------
% Exclusive memory analysis information
slm_ifLn5.wsl = slm_ifLn5.wsl + slm_b.bytes + slm_r.bytes;
slm_ifLn5.wss = slm_ifLn5.wss + slm_new.bytes;

% Total storage size
slm_ifLn5.sr = slm_b.bytes + slm_r.bytes + slm_new.bytes;
%-------------------------------------------------------;
```

Figure 5.9: The SLAMM generated exclusive memory analysis code block for scope B2 of Figure 5.6

.

block. The inclusive memory analysis code block for B1 contains the sum of all included bodies for the fields *wsl* and *wss*. Note that the fields *wsl* and *wss* must be summed because they are based on exclusive identifier counts. The value for *sr*, which is based on an inclusive identifier count remains unchanged. **Print memory analysis**, the final type of code block, consists of a call to the provided Matlab function *SlmPrtAnalysis* with the proper structure as an argument. The form of the output of the Matlab function *SlmPrtAnalysis* is described in the next section.

5.8.4 Functions

We next describe how SLAMM supports function calls. Using the techniques described in Section 5.7.3, SLAMM differentiates between identifiers that correspond to functions and identifiers that correspond to variables. Functions are further classified into those that provide a significant contribution to data movement versus those that do not. We refer to those functions that provide a significant contribution to data movement as **profiled functions**. SLAMM maintains the proper classification for a collection of commonly used built-in Matlab functions. For example, the Matlab function *size*, which determines the extent of a variable, does not contribute to data movement,

```
[slm_L1C5, slm_sin__L1C5] = SLMsin(r);
[slm_L1C14,slm_cos__L1C14] = SLMcos(z);
t = slm_L1C5+slm_L1C14,
```

Figure 5.10: The SLAMM transformed Matlab code for the expression $t = sin(r) + cos(z)$.

however the function qr, which calculates the QR factorization of an input matrix does contribute to data movement. SLAMM provides a directive for the classification for user-supplied functions, as described in Section 5.9. We concentrate on two types of code transformations necessary for profiled functions. The first type of transformation involves changing the actual function call, while the second involves changes to the function itself. We examine the function call transformation first.

SLAMM transforms a function call to a profiled function call by prefixing the string "SLM" to the name of the function and adding an additional output argument or return value. The additional output argument is a Matlab structure containing the SLAMM calculated memory analysis. The profiled function's contribution to data movement is subsequently accumulated in the appropriate inclusive memory analysis code block. For profiled functions that return multiple output arguments, the code transformation only requires the addition of an extra output argument. For profiled functions that return a single output argument, additional code transformation may be necessary. For example, a single expression that uses the output argument of a profiled function as an operand requires the generation of multiple sub-expressions. All sub-expressions are linked by temporary variables. An example of the SLAMM transformed Matlab for the expression $t = sin(r) + cos(z)$, where $r, z, t \in \mathbb{R}^n$ is provided in Figure 5.10.

In Figure 5.10, the single original expression $t = sin(r) + cos(z)$ is broken into three separate statements. The temporary variables slm_L1C5 and slm_L1C14 are the original output arguments of the *sin* and *cos* functions respectively and are used to calculate the expected result t. The structures slm_sin__L1C5 and slm_cos__L1C14 contain the memory analysis for the *sin* and *cos* functions respectively. The transformation in Figure 5.10 allows both the calculation of the correct result t and the proper memory analysis.

We next describe the transformation of profiled functions. The function call transformation requires the generation of new versions of the profiled functions. SLAMM provides a collection of profiled functions for all necessary built-in Matlab functions. The provided profiled functions consist of a call to the original function and the assignment of the memory analysis structures. For user-supplied Matlab functions, the SLAMM language processor generates the various memory analysis code blocks described in Section 5.8.3 and alters the name and output arguments appropriately.

5.8.5 Memory Analysis Output

The output of the supplied Matlab function *SlmPrtAnalysis* is the end result of the automated memory analysis process. An example output is provide in Figure 5.11. The first line of Figure 5.11 indicates that it is the memory analysis for a body of code with name “Blgmres”. The next two lines provide the total storage requirement (SR) and working set load size (WSL) in Mbytes. For the WSL prediction, a most likely value and error bounds are provided. The error bounds address the inability of the SLAMM language processor to accurately predict data movement for certain programming constructions as described in detail in Chapter 6. The WSL value is subdivided into different types of data movement. Dense matrix-matrix operations that can be implemented as a call to the BLAS routine DGEMM [31] as well as those involving sparse matrix-vector operations are indicated separately. SLAMM also provides a prediction for the upper and lower bounds on execution time. The accuracy of both the WSL and execution time predictions are analyzed throughly in Chapter 6.

```
SLAMM Memory Analysis for Body: Blgmres
       TOTAL: Storage Requirement  Mbytes (SR) : 7.69
       TOTAL: Loaded from L2 -> L1 Mbytes (WSL): 549.87 +- 10.01
                          DGEMM    Mbytes      : 0.00 +- 0.00
                     Sparse Ops    Mbytes      : 129.18 +- 10.01
       Predicted etime for SUN Ultra II msec: [580.418 3311.795]
```

Figure 5.11: The output of a call to the *SlmPrtAnalysis* function.

5.9 Using the SLAMM Language Processor

Finally, we provide an operational description of the user interface to the SLAMM language processor. The SLAMM language processor consists of an executable program and a collection of utility functions written in Matlab. ANSI C source code for the SLAMM executable was generated by the Eli compiler construction suite described in Section 5.6. The actions of the language processor are controlled by a collection of directives. All SLAMM directives include a prefix, a *command*, and a terminating semicolon. The prefix *%SLM* is treated by the Matlab interpreter as a comment. The *command* component of the directive consists of a keyword followed by one or more parameters. The legal keyword and parameter combinations are:

{Start | FuncStart | End | FuncEnd} SymName Directives with the Start and End keywords delineate the beginning and ending of a body of Matlab code. Similarly, directives with the FuncStart and FuncEnd keywords delineate the beginning and end of a Matlab function. SLAMM requires that the first line of all input Matlab scripts or functions contains a Start or FuncStart directive respectively. Similarly, SLAMM requires that the last line of all input Matlab scripts or functions contains an End or FuncEnd directive respectively. Corresponding Start and End directives are matched using the SymName parameter. The SymName parameter is a user-supplied name and allows easy identification of pieces of code.

Var {VarName | VarNameList, VarName} The Var directive provides the ability to classify one or more identifiers as variables. The VarName parameter is the name of the variable. The VarNameList parameter is one or more VarName parameters separated by a comma.

{ Func | IncFunc } { FuncName | FuncNameList, FuncName} Directives with the Func keyword classify a function identifier for which data movement is ignored, while the IncFunc keyword indicates profiled functions that contain significant data movement. The FuncName parameter is the name of the function. The FuncNameList parameter is one or more FuncName parameters separated by a comma.

Print SymName The Print directive requests that a Print Memory Analysis code block for the body SymName be generated. The Print directive must be placed outside the scope of the SymName code body.

The SLAMM language processor is accessed from either the UNIX command line or through the Matlab interpreter. Command line access is particularly useful for generating the profiled functions described in the previous section. SLAMM is accessed through the Matlab interpreter by calling the supplied function slmexe. The slmexe function has a single input argument, a string variable containing the name of the Matlab script. The slmexe function spawns a shell process that executes SLAMM on the input Matlab script and subsequently executes the output using the Matlab interpreter.

The computational cost to perform a SLAMM memory analysis has two components: the cost to process the input text and the overhead of the memory analysis calculation code. The cost to process the input text is minimal and typically requires several tenths of a second. The SLAMM generated code however typically increases the Matlab execution time by a factor of 2 to 10. The more extensively the input Matlab uses array subscripting, the larger the overhead. Ironically, the SLAMM generated Matlab has an increased execution time due to the large number of memory copies SLAMM generates to properly predict data movement.

Chapter 6

Automated Memory Analysis

We demonstrated in Chapter 4 that a manual line-by-line memory analysis of a mathematical description of a Krylov iterative algorithm predicts the differences in data movement for two different implementations. In particular, we predict the performance of a multivector versus non-multivector based implementation by comparing ratios for the working set size and measured data movement. Furthermore, we are able to observe a strong correlation between the ratios of data movement through the memory hierarchy and execution time. However, manual memory analysis is time consuming, laborious, and error prone process. Our results are a proof of concept of the importance of memory analysis, not a general solution.

In Chapter 5, we described the development of the SLAMM language processor that automates the memory analysis procedure. The SLAMM language processor accepts Matlab code as input and outputs a modified version. The modified Matlab output code contains both the original code and new blocks of code that calculate the memory usage properties.

The SLAMM-based memory analysis represents a significant advance over manual memory analysis in terms of both speed and accuracy. A manual memory analysis that might require as long as multiple days is now achieved in 20 minutes by SLAMM. Further, the language processor accurately predicts actual data movement through the lowest level of the memory hierarchy for most input code. For the remainder of the chapter, we examine the accuracy of SLAMM-based memory analysis by evaluating a collection of benchmarks. Additionally, we provide examples of how to use SLAMM to assist algorithm design and implementation. In Section 6.1, we describe our approach to the validation process, including the compute platforms, test methodology, and test matrices. In Section 6.2, we evaluate the ability of SLAMM to properly analyze linear algebra computational kernels. In Section 6.3, we examine several different Matlab subroutines that implement the Conjugate Gradient algorithm and determine if different coding styles affect the accuracy of the SLAMM memory analysis. In Section 6.4, we examine a trio of GMRES algorithms. The standard restarted GMRES(m), LGMRES(m,s), and B-LGMRES(m,s) variants are all analyzed.

6.1 Test Configuration

To evaluate the accuracy of SLAMM-based memory analysis, we compare the predicted data movement for a benchmark written in Matlab to the actual data movement for the corresponding benchmark written in C. We use PETSc 2.1.6 (Argonne National Laboratory's Portable, Extensible Toolkit for Scientific Computation) to provide a high-performance, single-processor implementation of the various benchmarks. Whenever possible, we configure PETSc to take advantage of optimized math libraries. We instrument the benchmark C code with the locally developed performance profiling library Htrace, which is based on the PAPI hardware performance counter API [81]. Htrace calculates data movement by tracking the number of cache lines moved through the different components of the memory hierarchy. We focus on three primary microprocessor compute platforms that provide counters for cache lines loaded from the memory hierarchy to the L1 cache: Sun Ultra II [74] (Ultra II), IBM POWER 4 [16] (Pwr4), and MIPS R14K [113] (R14K). We include two additional compute platforms, which do not provide hardware counters of cache lines loaded from the memory hierarchy to the L1 cache, Intel P4 Xeon [23, 24] (P4), and the Motorola G4 (G4) to determine if we can predict execution time based on the SLAMM-predicted data movement. These five compute platforms represent a wide range of cache sizes, system bandwidths, and execution rates. A description of the cache configurations for each compute platform is provided in Table 6.1.

Table 6.1: Description of the microprocessor compute platforms and their cache configurations. The total size of the cache (size), length of cache line (cline), associativity (assoc), and location of the cache (loc) is provided (if known).

CPU Company Version		Ultra II SUN	Pwr4 IBM	R14K SGI	P4 Intel	G4 Motorola 7447A
Mhz		400	1300	500	2000	1500
L1 D-cache	size	32KB	32KB	32KB	8KB	32KB
	cline	16 bytes	128 bytes	32 bytes	64 bytes	–
	assoc	–	2-way	2-way	4way	–
	loc	on-die	on-die	on-die	on-die	on-die
L2 cache	size	4 MB	1440 KB	8 MB	512 KB	512 KB
	cline	64 bytes	128 bytes	128 bytes	64 bytes	–
	assoc		8-way	2-way	8-way	–
	loc	off-die	on-die	off-die	on-die	on-die
L3 cache	size	–	32 MB	–	–	–
	line	–	512 bytes	–	–	–
	assoc	–	8-way	–	–	–
	loc	–	off-die	–	–	–

Because the SLAMM language processor has limited accuracy under certain conditions, we provide error bounds in addition for the predicted values. To simplify presentation, we report only the predicted value of data loaded from the memory hierarchy to the L1 cache ($Mbytes_{L1}$) for comparison with the measured value.

In addition to the prediction of $Mbytes_{L1}$, we also examine the ability of the SLAMM language processor to accurately predict execution time. Execution time prediction is based on the assumption that the cost of data movement through the memory hierarchy dominates execution time. It also assumes that the cost of data movement is determined by memory bandwidth. While a valid assumption for certain codes, this simplification neglects a host of other factors, including memory latency, TLB misses, floating-point costs, and the quality of the executables instruction stream. Execution time prediction is further complicated by the fact that the SLAMM language processor has no general technique to accurately determine the location of a operand in the memory hierarchy.

We can, however, estimate bounds on the execution time. The lower bound represents the lowest execution time the algorithm could achieve, in which case all data loaded from the memory hierarchy is located in the L2 cache. The equation

$$T_L = Mbytes_{L1}/bandwidth_{L2}, \tag{6.1}$$

where $bandwidth_{L2}$ corresponds to the load bandwidth between the L1 and L2 cache represents this case. The upper bound corresponds to the case where all data loaded from L2 to L1 is located in the compute platform's main memory. The upper bound is calculated using the equation

$$T_U = Mbytes_{L1}/bandwidth_{MM}, \tag{6.2}$$

where $bandwidth_{MM}$ corresponds to the load bandwidth between the L1 cache and main memory.

We experimentally determine $bandwidth_{L2}$ and $bandwidth_{MM}$ for each compute platform using a test code similar to the STREAMS benchmark [71]. The test code times the execution of the axpy operation $x = x + \alpha y$, where $x, y \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$. The value of n is chosen such that $sizeof(cache_{L1}) < SR_{axpy} < sizeof(cache_{L2}) - 2 \times sizeof(cache_{L1})$, where SR_{axpy} is the total storage requirement for the axpy operation. To determine $bandwidth_{L2}$, a single axpy operation is executed followed by a flush of the L1 cache. The L1 cache is flushed by accessing and modifying a variable besides x, y, or α whose storage requirement is twice the size of the L1 cache. While it is possible that small pieces of x and y may be removed from the L2 cache and flushed to main memory, it is highly unlikely that a significant amount is displaced because L2 caches are typically significantly larger than L1 caches. With the variables x, y, and α located in the L2 cache, a second axpy operation is timed. We calculated $bandwidth_{L2}$ based on the average of 400 timings. The value for $bandwidth_{MM}$ is calculated in the same fashion, except all caches are flushed prior to the execution of the second axpy. Table 6.2 provides the experimentally determined bandwidths for each compute platform in Mbytes/sec based on an axpy with vectors of dimension 12000.

Table 6.2: Experimentally determined memory hierarchy bandwidths for each compute platform in Mbytes/sec.

	Ultra II	Pwr4	R14K	P4	G4
$bandwidth_{L2}$	870	8200	1400	7800	2100
$bandwidth_{MM}$	170	1500	420	850	420

The execution time bound provides a rough estimate on which to judge the memory efficiency of a particular implementation of a Matlab algorithm. A compiled implementation of an algorithm with an execution time greater than the upper bound has significant performance problems. A matrix-matrix multiplication, containing nested loops that do not respect storage order is a common example of this type of implementation. Conversely, an implementation with an execution time near or equal to the lower bound is extremely memory efficient. The memory efficiency of an implementation whose execution time lies near the middle of the bounds can not be inferred.

We use input matrices from the University of Florida Sparse Matrix Collection [27] and the Matrix Market Collection [77]. The s1rmq4m1b matrix was created by removing the several hundred explicitly stored zero values in the original s1rmq4m1 matrix.

.

6.2 Fundamental Linear Algebra Operations

We begin our evaluation of the accuracy of SLAMM memory analysis by examining several linear algebra kernel benchmarks. We chose the kernel benchmarks because

Table 6.3: List of test problems with the matrix order (n), number of nonzeros (nnz), matrix density (ρ), and description of the application area (if known).

Matrix	n	nnz	ρ	Application Area
nos5	468	5172	2.4e-2	finite element approximation of building
gr3030	900	7744	9.6e-3	nine point stencil on a 30 x 30 grid
sherman5	3312	20793	1.9e-3	fully implicit oil simulator
pesa	11738	79566	5.7e-4	
epb1	14734	95053	4.3e-4	heat exchanger simulation
ex15	6867	98671	2.1e-3	3-D fluid flow
memplus	17758	126150	4.0e-4	digital circuit simulation
zhao2	33861	166453	1.4e-4	electromagnetic systems
epb2	25288	175027	2.7e-4	heat exchanger simulation
wang3	26064	177168	2.6e-4	electron continuity for 3D diode
s1rmq4m1b	5489	262411	8.7e-3	cylindrical shell 30x30 quad mesh
bcsstk17	10974	428650	3.6e-3	stiffness matrix - elevated pressure vessel
finan512	74752	596992	1.1e-4	financial portfolio optimization

they form the basis of all linear algebra algorithms. Different inputs for each kernel benchmark are selected such that the storage requirement is larger than the L1 cache for all compute platforms. We first examine vector kernel benchmarks in Section 6.2.1. Next we examine two benchmarks with matrices as operands: a dense matrix benchmark in Section 6.2.2 and a sparse matrix benchmark in Section 6.2.3.

6.2.1 Vector Benchmarks

We create four kernel benchmarks to test the ability of SLAMM to properly analyze dense linear algebra with vector operands. The benchmarks are DotProd, Axpy, Maxpy, and Combo. The **DotProd** benchmark calculates the dot product of two disjoint vectors: $\alpha = (x, y)$, where $x, y \in \mathbb{R}^n$, and $\alpha \in \mathbb{R}$. The **Axpy** benchmark calculates $x = x + \alpha y$, where $x, y \in \mathbb{R}^n$, and $\alpha \in \mathbb{R}$. The **Maxpy** benchmark calculates $x = x + yz$, where $x \in \mathbb{R}^n$, $y \in \mathbb{R}^{n\times s}$, and $z \in \mathbb{R}^s$. The **Combo** benchmark contains two common linear algebra operations and calculates $\alpha = (r, r)$ and $x = x + \alpha y$, where $r, x, y \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$.

The DotProd, Axpy, and Maxpy benchmarks involve the streaming of operands from the memory hierarchy to the processor with no possibility of reuse. A streaming access pattern should be easily analyzed by the SLAMM language processor. In the case of the Combo benchmark, the reuse of the vector r complicates the analysis. However, the proper recognition of data reuse is achieved by the duplicate identifier correction described in Section 5.8.2.

Each benchmark is executed for several different values of n, and both the L1 and L2 caches are flushed to prevent L1 reuse. The SLAMM-predicted value of $Mbytes_{L1}$, the working set load size (WSL_P), the measured value (WSL_M) in Kbytes, and the corresponding relative error for each input size and benchmark are provided in Table 6.4. There is some variability between the measured values for different compute platforms. For example, WSL_M ranges from 397 to 468 Kbytes when $n = 12000$ for the Axpy benchmark. To account for the compute platform variability, we consider any prediction within 20% of measured to be an accurate prediction for $Mbytes_{L1}$. With the exception of the Combo benchmark on the Pwr4 platform for $n = 24576$, all predicted and measured values for $Mbytes_{L1}$ agree within 20%. The reason for the lone discrepancy is unclear but may be related to the particular value of n that contains a large power of 2 (1024) as one of its factors. As expected, SLAMM successfully recognizes the reuse of r in the Combo benchmark.

Given the ability of the SLAMM language processor to accurately predict $Mbytes_{L1}$ for the dense linear algebra kernel benchmarks, we now examine the accuracy of execution time prediction. Recall that in general, execution time prediction is inherently inaccurate because SLAMM has no information about the location of the required variables in the memory hierarchy. However, we can use the cache flushing techniques described in Section 6.1 to predict execution time under controlled conditions. To evaluate the upper bound on execution time, we flush all caches prior to timing the benchmark. All variables required by the benchmarks are therefore located in the main memory. To evaluate the lower bound, we initially warm the caches followed by a flush of the L1 cache. As a result, all variables required by the benchmarks are located in the L2 cache.

Table 6.4: $Mbytes_{L1}$ for the vector kernel benchmarks. WSL_P, calculated by the SLAMM language processor and measured values of WSL_M are in Kbytes.

OP	n	WSL_P	Ultra II WSL_M	Ultra II err	Pwr4 WSL_M	Pwr4 err	R14K WSL_M	R14K err
DotProd	3000	46.9	49.7	-5.6%	53	-11.5%	51	-8.0%
	7000	109.4	114.2	-4.2%	105.1	4.1%	115.1	-5.0%
	24576	384	398.8	-3.7%	451.6	-15.0%	396.2	-3.1%
Axpy	3000	46.9	49.7	-5.6%	54.1	-13.3%	51.4	-8.8%
	7000	109.4	114.2	-4.2%	108.9	0.5%	115.4	-5.2%
	12000	187.5	194	-3.4%	199.1	-5.8%	195.2	-3.9%
	24576	384	397	-3.3%	468.2	-18.0%	396.6	-3.2%
Maxpy	3000	117.2	122.9	-4.6%	118.1	-0.8%	125.3	-6.5%
	7000	273.5	338.3	-19.2%	249.7	9.5%	283.4	-3.5%
(s=4)	1200	468.8	482.6	-2.9%	451.4	3.9%	483.6	-3.1%
	24576	960	987.6	-2.8%	851.3	12.8%	986.6	-2.7%
Combo	3000	70.3	74.3	-5.4%	72.9	-3.6%	76.1	-7.6%
	7000	164.1	171.1	-4.1%	150.6	9.0%	172.3	-4.8%
	12000	281.3	290.6	-3.2%	275	2.3%	292.2	-3.7%
	24576	576	596.2	-3.4%	1185	-51.4%	594.1	-3.0%

We examine the overall accuracy of execution time prediction for all dense linear algebra benchmarks first, followed by a detailed look at some specific instances. While we expect the upper bound on execution time to be accurately predicted for all benchmark input set combinations, we expect only those benchmarks with sufficiently small storage requirements to be accurately predicted for the lower bound. In particular, we expect any benchmark with

$$SR < sizeof(cache_{L2}) - 2 \times sizeof(cache_{L1}), \tag{6.3}$$

where $2 \times sizeof(cache_{L1})$ accounts for the storage requirement of the array used to flush the L1 cache, to be accurately predicted. Using the measured bandwidths for each compute platform from Table 6.2 as input, SLAMM predicts the upper bound (T_U) and a lower bound (T_L) on execution time. We compare the measured execution time (T_M) to the SLAMM predictions for each benchmark where (6.3) holds. In Table 6.5, we provide a percentage of execution times predicted to within 20% error for all five compute platforms. For example, Table 6.5, indicates that on the P4 compute platform, 93% of the T_U values the all benchmarks are predicted within 20% error.

Table 6.5: Overall accuracy of execution time prediction for the vector benchmarks.

	Accurately Predicted	
	T_L	T_U
Ultra II	47%	100%
Pwr4	53%	73%
R14K	69%	93%
P4	80%	93%
G4	54%	100%
Total	61%	92%

Table 6.5 indicates that the overall accuracy for the upper bound on execution time is excellent for all but the Pwr4 compute platform. The upper bounds on execution time is accurately predicted for all benchmarks on the G4, a compute platform for which we have no ability to measure data movement. The overall accuracy of the lower bound with the exception of the P4 is marginal. However, the total accuracy across all platforms indicates that, while the prediction of execution time is not entirely trustworthy, it does appear to work in general. We next examine some specific instances of the execution time prediction.

The predicted upper bound on execution time (T_U) and actual execution time (T_M) for each of the vector linear algebra benchmarks and the relative errors for all five compute platforms are provided in Table 6.6. For two of the compute platforms, Ultra II and G4, the predicted and measured execution time agree to within 9% for all benchmark and input size combinations. The execution time predictions for the P4 are nearly as accurate, exceeding the 20% accuracy threshold for only the Combo benchmark for $n = 24576$. However, T_U is not as accurate for the R14K and Pwr4 compute platforms. For the R14K, the measured execution time is outside the accuracy threshold for three benchmarks with the relative error ranging from -21 to 23%. We also observe accuracy errors larger than the acceptable threshold for the small input sets for the DotProd, Axpy, and Combo benchmarks for the Pwr4 compute platform.

Despite these discrepancies, the upper bound on execution calculated by SLAMM is accurate for 92% of the benchmark, input set, and compute platform combinations.

The predicted lower bound (T_L) and measured (T_M) execution time for each of the vector benchmarks for all five compute platforms and the relative error are provided in Table 6.7. We include only values that we expect to be accurately predicted based on (6.3). While the goal of SLAMM-based memory analysis is not to explain the peculiarities of a particular compute platform, it is useful to examine specific results in detail to verify correct operation. We therefore examine the accuracy of the execution time prediction in detail for the Ultra II and Pwr4 compute platforms.

We begin with the Ultra II. Interestingly, two of the benchmarks, DotProd and Axpy, have consistently low relative errors (2-4%), while the other two benchmarks, Maxpy and Combo, have consistently high relative errors (46% and 21% respectively). Examination of the PETSc source reveals that the DotProd and Axpy benchmarks are implemented as calls to vendor-optimized versions of BLAS subroutines for the Ultra II. The Maxpy benchmark is implemented in source code, which is likely less optimized than either the DotProd or Axpy benchmarks. As a result, the execution time of Maxpy is greater than the SLAMM-predicted value, which is based on experimental timings of an axpy operation. Similary, the Combo benchmark, which is composed of two separate calls to optimized BLAS subroutines, also have execution times greater than predicted by SLAMM due to subroutine call overhead. It is therefore apparent that the discrepancies between predicted and measured execution time for the Ultra II are due to implementation-specific details and not to the prediction methodology.

As with the Ultra II, the Pwr4 compute platform implements all benchmarks in a similar manner. It is therefore not surprising that the relative error for the Pwr4 platform is consistently high for both the Maxpy and Combo benchmarks as well. In addition, the higher-than-expected execution times for the Pwr4 on the Axpy and Combo benchmarks for $n = 24576$ are consistent the with correspondingly higher-than-expected value for $Mbytes_{L1}$ shown in Table 6.4.

6.2.2 Dense Matrix Benchmark

The **MxM** benchmark is the multiplication of two long skinny matrices $Z = X^T Y$ where $X, Y \in \mathbb{R}^{n \times s}$, and $Z \in \mathbb{R}^{s \times s}$. We choose $s = 4$ as a typical configuration encountered in block Krylov methods. It is possible to implement the PETSc version of MxM benchmark as a call to either the BLAS *DGEMM* subroutine [31] or a handwritten subroutine that takes advantage of the particular size of the operands. In Table 6.8 we provide the SLAMM predicted value (WSL_P) and the measured $Mbytes_{L1}$ for a vendor optimized BLAS DGEMM subroutine (WSL_M^B) and a handwritten subroutine (WSL_M^H). The handwritten implementation corresponds to the VecStrideDOT subroutine described in Section 4.2.

Table 6.8 indicates that SLAMM predicts $Mbytes_{L1}$ to within 9% error for the handwritten implementation of the MxM benchmark. However, SLAMM does not accurately predict $Mbytes_{L1}$ for the BLAS implementation of the MxM benchmark for both the Ultra II and Pwr4 compute platforms. The BLAS implementation of the MxM benchmark requires 15 to 25% and 38 to 44% more data movement than necessary on the Ultra II and Pwr4 compute platforms, respectively. The BLAS implementation on the R14K and the handwritten version of MxM are memory-efficient because they load only the minimum amount of data from the memory hierarchy. We believe that the additional data movement required on the Ultra II and Pwr4 compute platforms is due to cache conflict misses in the L1 cache.

For larger values of s the compute platform dependence of WSL_M for the MxM benchmark becomes even more pronounced. For example for $n = 4500$ and $s = 20$, the SLAMM predicted value $WSL_P = 1406$ Kbytes is significantly different than the measured values of 2777, 7968, and 10822 Kbytes for the BLAS implementation of MxM benchmark on the Ultra II, Pwr4 and R14K compute platforms, respectively.

Table 6.6: The predicted upper bound (T_U) versus measured execution time (T_M) in μsec for the vector benchmarks.

	n	SR	Ultra II			Pwr4			R14K			P4			G4		
OP		Kbytes	T_M	T_U	err	T_M	T_U	err	T_M	T_U	err	T_M	T_U	err	T_M	T_U	err
DotProd	3000	46.9	.292	.276	5%	.047	.030	36%	.126	.110	13%	.056	.055	2%	.114	.112	2%
	7000	109.4	.665	.643	3%	.066	.071	-8%	.275	.257	7%	.125	.129	-3%	.257	.260	-1%
	24576	384	2.251	2.259	0%	.181	.248	-37%	.869	.904	-4%	.409	.452	-11%	.892	.914	-2%
Axpy	3000	46.9	.288	.276	4%	.046	.030	35%	.123	.110	11%	.059	.055	7%	.104	.112	-8%
	7000	109.4	.651	.643	1%	.082	.071	13%	.248	.257	-4%	.135	.129	4%	.258	.260	-1%
	12000	187.5	1.113	1.103	1%	.121	.121	0%	.442	.441	0%	.222	.221	0%	.451	.446	1%
	24576	384	2.249	2.259	0%	.247	.248	0%	.749	.904	-21%	.416	.452	-9%	.944	.914	3%
Maxpy	3000	117.2	.682	.690	-1%	.078	.076	3%	.359	.276	23%	.141	.138	2%	.273	.279	-2%
	7000	273.5	1.625	1.609	1%	.147	.176	-20%	.803	.643	20%	.337	.322	4%	.651	.651	0%
(s=4)	12000	468.8	2.722	2.758	-1%	.251	.302	-20%	1.366	1.103	19%	.577	.552	4%	1.105	1.116	-1%
	24576	960	5.532	5.647	-2%	.542	.619	-14%	2.740	2.259	18%	1.050	1.129	-8%	2.286	2.286	0%
Combo	3000	70.3	.452	.414	8%	.069	.045	35%	.188	.165	12%	.080	.083	-4%	.173	.167	3%
	7000	164.1	1.011	.965	5%	.119	.106	11%	.402	.386	4%	.176	.193	-10%	.413	.391	5%
	12000	281.3	1.708	1.654	3%	.171	.181	-6%	.665	.662	0%	.301	.331	-10%	.688	.670	3%
	24576	576	3.450	3.388	2%	.379	.372	2%	1.343	1.355	-1%	.562	.678	-21%	1.467	1.371	7%

Table 6.7: The predicted lower bound (T_L) versus measured execution time (T_M) in μsec for the vector benchmarks.

	n	SR	Ultra II			Pwr4			R14K			P4			G4		
OP		Kbytes	T_M	T_L	err	T_M	T_L	err	T_M	T_L	err	T_M	T_L	err	T_M	T_L	err
DotProd	3000	46.9	.050	.048	4%	.006	.006	0%	.031	.034	-10%	.007	.006	14%	.021	.022	-5%
	7000	109.4	.115	.113	2%	.012	.013	-8%	.054	.080	-48%	.016	.014	13%	.046	.052	-13%
	24576	384	.411	.396	4%	.047	.045	4%	.141	.282	-100%	.058	.050	14%	.270	.183	32%
Axpy	3000	46.9	.050	.048	4%	.007	.006	14%	.042	.034	19%	.007	.006	14%	.029	.022	24%
	7000	109.4	.115	.113	2%	.013	.013	0%	.086	.080	7%	.015	.014	7%	.058	.052	10%
	12000	187.5	.197	.193	2%	.022	.022	0%	.138	.138	0%	.025	.024	4%	.089	.089	0%
	24576	384	.412	.396	4%	.084	.045	46%	.282	.282	8%	.062	.050	19%	.216	.183	15%
Maxpy	3000	117.2	.226	.121	46%	.032	.014	56%	.103	.086	17%	.014	.015	-7%	.077	.056	27%
	7000	273.5	.527	.282	46%	.073	.032	56%	.225	.201	11%	.033	.035	-6%	.165	.130	21%
(s=4)	12000	468.8	.908	.483	47%	.125	.055	56%	.371	.345	7%	.120	.061	49%			
	24576	960	1.864	.990	47%	.262	.113	57%	.797	.706	11%						
Combo	3000	70.3	.092	.072	22%	.011	.008	27%	.054	.052	4%	.013	.009	31%	.044	.033	25%
	7000	164.1	.213	.169	21%	.023	.019	17%	.116	.121	-4%	.030	.021	30%	.084	.078	7%
	12000	281.3	.369	.290	21%	.038	.033	13%	.196	.207	-6%	.052	.037	29%	.167	.134	20%
	24576	576	.762	.594	22%	.143	.068	52%	.388	.424	-9%						

Table 6.8: $Mbytes_{L1}$ for the MxM benchmark. WSL_P, calculated by the SLAMM language processor and the measured value for the BLAS (WSL_M^B) and hand optimized (WSL_M^H) implementations are in Kbytes. Relative error between predicted and measured is provided in parenthesis.

size		Ultra II		Pwr4		R14K	
n,s=4	WSL_P	WSL_M^B	WSL_M^H	WSL_M^B	WSL_M^H	WSL_M^B	WSL_M^H
3000	188	223 (-16%)	193 (-3%)	332 (-44%)	207 (-9%)	199 (-6%)	195 (-4%)
7000	438	583 (-25%)	452 (-3%)	750 (-42%)	463 (-5%)	455 (-4%)	452 (-3%)
12000	750	884 (-15%)	773 (-3%)	1265 (-41%)	771 (-3%)	775 (-3%)	772 (-3%)
24000	1500	1756 (-15%)	1546 (-3%)	2422 (-38%)	1444 (4%)	1543 (-3%)	1539 (-2%)
24576	1536	1836 (-16%)	1583 (-3%)	2471 (-38%)	1492 (3%)	1579 (-3%)	1576 (-2%)

Because the aim of SLAMM is to predict data movement for a memory-efficient implementation, we provide both a prediction of the minimum required data movement and an estimate of the additional data movement required due to L1 cache conflicts. We approximate cache conflicts in the MxM benchmark, by assuming they cause an additional load of the X and Y operands from the L2 cache. Tighter bounds for the required data movement could be estimated using ATLAS [116]. While the SLAMM language processor does not provide an accurate estimate of $Mbytes_{L1}$ for dense matrix matrix multiplication at this time, it does indicate the relative impact an memory-efficient dense matrix-matrix multiply subroutine may have on an algorithm's implementation.

6.2.3 Sparse Matrix Benchmark

We use the MxV benchmark to characterize sparse linear algebra computations. The **MxV** benchmark is the multiplication of a sparse matrix times a vector $y = Ax$, where $x, y \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$. Similar to the vector benchmarks, the MxV benchmark involves the streaming of data from the memory hierarchy into the processor with little chance of reuse. Unlike the vector benchmarks that impose a stride-one access pattern, the MxV benchmark includes indirect addressing as well.

We examine the MxV benchmark using 13 different input matrices whose total storage requirements range from 70 Kbytes to more than 8 Mbytes in size. The accurate prediction of $Mbytes_{L1}$ for the MxV benchmark is nontrivial in that it is dependent on the structure of the nonzero pattern of the A matrix. In particular, the loading of the x vector and storing of the resulting y vector may interfere in the cache. Further, the use of indirect addressing for the x vector may result in a larger $Mbytes_{L1}$ than necessary due to inefficient use of cache lines. While the interaction of nonzero patterns and cache size on data movement was analyzed by Temam and Jalby [102], we require a simpler approach. In particular, we do not want the analysis of the nonzero pattern of A to significantly increase the execution time of the SLAMM-transformed code. Based on an examination of experimentally measured $Mbytes_{L1}$, we therefore approximate the impact the nonzero pattern on data movement for the MxV benchmark by adding an additional vector of length n to WSL_P. This approximation is reasonably accurate across the entire set of input matrices. The total storage requirement, the predicted and measured $Mbytes_{L1}$, and the relative error for each input matrix for each primary compute platform are provided in Table 6.9.

Table 6.9: $Mbytes_{L1}$ for the MxV benchmark. SR and WSL_P, calculated by the SLAMM language processor, and the measured values of WSL_M are in Kbytes.

			Ultra II		Pwr4		R14K	
matrix	SR	WSL_P	WSL_M	err	WSL_M	err	WSL_M	err
nos5	70	70	73.4	2.5%	77.5	-3.0%	75.5	-0.4%
gr3030	108	108	110.3	7.9%	111	7.2%	115.4	3.1%
sherman5	308	308	322	-4.3%	292.9	5.2%	339.7	-9.3%
pesa	1162	1162	1441	-17.6%	1521	-23.6%	1517	-23.4%
epb1	1402	1402	1483	-5.5%	1403	-0.1%	1530	-8.4%
ex15	1290	1290	1267	1.8%	1298	-0.6%	1272	1.4%
memplus	1825	1825	2150	-15.1%	2308	-20.9%	2235	-18.3%
zhao2	2612	2612	3197	-18.3%	3321	-21.3%	3411	-23.4%
epb2	2544	2544	2848	-10.7%	2679	-5.0%	2786	-8.7%
wang3	2585	2585	2948	-12.3%	2583	0.1%	3064	-15.6%
s1rmq4m1b	3182	3182	3216	1.0%	3324	-2.3%	3135	3.6%
bcsstk17	5238	5238	4816	11.4%	4871	10.2%	4752	12.9%
finan512	8456	8456	9314	-9.2%	8578	-1.4%	9319	-9.3%

Table 6.9 reveals that WSL_M is consistent between the three compute platforms, and that in general, the SLAMM language processor accurately predicts $Mbytes_{L1}$

within 20% error with only a few exceptions. For example, the predicted values for $Mbytes_{L1}$ are less accurate for the pesa, memplus, and zhao2 input matrices, where they are 20-25% too low for several platforms. The remaining matrices for which the predictions are within the 20% threshold are either consistently low (wang3 and finan512), consistently high (gr3030 and bcsstk17), or within the compute platform's variability of WSL_M (sherman5, s1rmq3m1b, nos5, and ex15). We therefore conclude that SLAMM, in general, predicts $Mbytes_{L1}$ within 20% of measured for the MxV benchmark. The largest source of error is dependent on the nonzero pattern of the matrix. We address the matrix-dependent uncertainty with the addition of error bounds.

We establish in the previous section that the SLAMM language processor can accurately predict an upper bound on execution time for 92% of the dense linear algebra benchmarks. Unlike the dense kernel benchmarks, with the MxV benchmark, we do not have control over the placement or size of the required variables. We can, however, verify that the measured execution time is contained within the upper and lower bounds of the predicted execution time. In contrast to previous benchmarks, we time the MxV benchmark without flushing cache and use the average execution time for 100 matrix vector multiplies. The measured and predicted bounds on execution time are provided in Table 6.10. The measured execution time is contained within the predicted bounds for 62 of 65 possible input matrix and compute platform combinations. Two of the discrepancies, the s1rmq4m1b matrix on the R14K and the finan512 matrix on the G4 have T_M that are outside the predicted range by an insignificant 2.5% and 2.2% respectively. The third discrepancy, where the T_M for the zhao2 matrix is 17.5% greater than expected on the G4 compute platform, is consistent with the low prediction for $Mbytes_{L1}$ in Table 6.9.

The results presented in this section demonstrate that the SLAMM language processor accurately predicts $Mbytes_{L1}$ to within 20% for most dense linear algebra benchmarks with vector operands. The only exceptions are due to particular input size

Table 6.10: The measured (T_M), predicted lower bound (T_L), and upper bound (T_U) on execution time in μsec for the MxV benchmark.

Matrix	SR	Ultra II			Pwr4			R14K			P4			G4		
	Kbytes	T_M	T_L	T_U	T_M	T_L	T_U	T_M	T_L	T_U	T_M	T_L	T_U	T_M	T_L	T_U
pesa	1162	3.69	1.20	6.83	.48	.14	.75	1.19	.85	2.73	.67	.15	1.37	2.47	.55	2.77
wang3	2585	8.06	2.66	15.21	1.14	.30	1.67	2.60	1.90	6.08	1.50	.33	3.04	5.77	1.23	6.16
memplus	1825	4.73	1.88	10.74	.90	.21	1.18	1.88	1.34	4.29	1.12	.23	2.15	4.15	.87	4.35
sherman5	308	.75	.32	1.81	.13	.04	.20	.30	.23	.73	.08	.04	.36	.21	.15	.73
bcsstk17	5238	6.51	5.53	31.57	2.24	.63	3.46	8.63	3.95	12.63	3.85	.69	6.31	10.61	2.56	12.78
s1rmq4m1b	3182	3.37	3.35	19.10	1.32	.38	2.09	2.33	2.39	7.64	2.30	.42	3.82	7.23	1.55	7.73
nos5	70	.20	.08	.44	.03	.01	.05	.07	.06	.18	.02	.01	.09	.04	.04	1.79
gr3030	108	.33	.12	.70	.04	.01	.08	.10	.09	.28	.02	.02	.14	.06	.06	.28
ex15	1290	2.29	1.33	7.60	.41	.15	.83	1.01	.95	3.04	.73	.16	1.52	2.81	.61	3.07
finan512	8456	32.91	8.72	49.74	4.08	.99	5.45	13.58	6.22	19.90	6.05	1.08	9.95	20.57	4.03	20.13
epb1	1402	4.33	1.44	8.25	0.60	.17	.90	1.35	1.03	3.30	.79	.18	1.65	3.36	.67	3.36
epb2	2544	8.66	2.62	15.00	1.14	.30	1.64	2.68	1.87	5.99	1.46	.33	2.99	5.68	1.21	6.06
zhao2	2612	8.31	2.69	15.36	1.30	.31	1.68	3.22	1.92	6.15	1.95	.34	3.07	7.31	1.24	6.22

and compute platform combinations that interact negatively with the cache configurations. The ability to accurately predict $Mbytes_{L1}$ for the MxM benchmark is limited by the compute platform and implementation variability. It was also demonstrated that in general SLAMM predicts $Mbytes_{L1}$ accurately for the sparse matrix-vector multiplication as well. These results provide the foundation on which to examine complete Krylov iterative algorithms, which are composed of basic linear algebra operations.

6.3 Conjugate Gradient

Based on the successful prediction of $Mbytes_{L1}$ for the kernel benchmarks, we next examine the ability of SLAMM to accurately predict an entire Krylov subspace algorithm. We examine the Conjugate Gradient algorithm first because it is a simple combination of the fundamental linear algebra operations described in the previous section. Because the cost of the Conjugate Gradient algorithm is dominated by the cost of the sparse matrix vector multiply, we expect SLAMM to predict data movement with an accuracy similar to that of the MxV benchmark. Before we evaluate the accuracy of SLAMM on the CG benchmark for a collection of input matrices, we first determine if the accuracy of the memory analysis is dependent on a particular Matlab programming style. We evaluate the impact of programming style in Section 6.3.1 by comparing the results from three different CG implementations. Next we evaluate the accuracy for a single CG subroutine on a selection of input matrices in Section 6.3.2. Finally, in Section 6.3.3, we analyze the cost of one particular feature of one of the CG subroutines.

6.3.1 Programming Style

We evaluate the impact of programming style on accuracy by comparing the results obtained on three Matlab codes written by different authors who implemented the same non-preconditioned CG algorithm. We demonstrate that the four corrections described in Section 5.8.2 eliminate the impact of programming style on the accuracy of the memory analysis. The three Matlab codes selected are: **Pcg**, the standard CG routine supplied by MathWorks; **CgNL**, a simple CG routine from the templates collection [14] available from the Netlib repository [78]; and **Mycg**, a CG routine written by the author. We modify all CG codes with the addition of a handful of SLAMM directives to control memory analysis. The Pcg code includes an optional parameter for the specification of a preconditioner. The CgNL version requires a preconditioner for correct execution. The Mycg version does not support preconditioning. For consistency, the CgNL code was modified to support an optional preconditioner. A single if statement was added to the Pcg code to remove the execution of a stagnation test not present in any of the other CG codes in Matlab or PETSc. Information on the three Matlab codes, including author, number of source lines, and number of lines modified to performed the memory analysis is provided in Table 6.11.

Table 6.11: Code statistics for several CG algorithms written in Matlab.

		Source Lines		Modifications	
Name	Author	Original	Final	%SLM	Other
Pcg	MathWorks	312	334	5	17
CgNL	Netlib/templates	66	81	6	13
Mycg	J. Dennis	36	36	5	0

To evaluate programming style, we chose a single input matrix: s1rmq4m1b. We compare the average measured $Mbytes_{L1}$ for all three compute platforms of 41.1 Mbytes to the predicted value for the three Matlab codes for several different SLAMM configurations. Different SLAMM configurations include different combinations of the memory analysis corrections described in Section 5.8.2. We present the predictions of SLAMM in $Mbytes_{L1}$ and the relative error for five different configurations in Table 6.12. The abreviations used for the memory analysis corrections in Table 6.12 are: for function calls (fcall), for duplicate identifiers (dup), for copy removal (copy), and for special operator transformations (sops). Table 6.12 indicates that the fcall correction has the largest impact on the Pcg code, reducing the error from 14.8 to -1.6%. The reason for this large improvement in accuracy is due to Pcg's matrix-free support. The Pcg code allows its first argument "A" to be either a sparse matrix or a pointer to a function call. The Pcg code therefore uses a large number of utility function calls to determine the nature of the argument "A." The application of the fcall correction therefore allows SLAMM to accurately predict $Mbytes_{L1}$. The remaining corrections do not, for this set of Matlab codes, have as significant an impact on accuracy. However, the inclusion of all four corrections does provide the most consistent estimate of $Mbytes_{L1}$ for all three Matlab codes. We use configuration 5 for all remaining analyses because this configuration allows the SLAMM language processor to be insensitive to programming style.

Table 6.12: Impact of various corrections to baseline memory analysis for CG algorithm with the s1rmq4m1b matrix. The average of measured value WSL_M for each primary compute platform of 41.1 Mbytes is used for comparison.

	Corrections				Pcg		CgNL		Mycg	
config	fcall	dup	copy	sops	WSL_P	err	WSL_P	err	WSL_P	err
1					47.17	14.8%	40.03	-2.6%	39.61	-3.6%
2	x				40.45	-1.6%	39.53	-3.8%	39.19	-4.6%
3	x	x			40.45	-1.6%	39.53	-3.8%	38.73	-5.8%
4	x	x	x		39.07	-4.9%	39.06	-5.0%	38.69	-5.9%
5	x	x	x	x	39.53	-3.8%	39.53	-3.8%	39.61	-3.6%

6.3.2 Accuracy for CG

We next examine the ability of SLAMM to accurately predict $Mbytes_{L1}$ for the CG benchmark on a collection of input matrices. We chose to analyze the Pcg code supplied by MathWorks, which includes an optional stagnation test, whose impact is determined in the next section. We analyze Pcg on a collection of input matrices whose total storage requirement of a single vector of length n is greater than the size of the L1 cache. The predicted and measured $Mbytes_{L1}$ for 10 iterations of the Conjugate

Gradient algorithm in Mbytes, and the relative error for the three primary compute platforms are provided in Table 6.13.

Table 6.13: $Mbytes_{L1}$ for CG benchmark for 10 iterations. SR and WSL_P, calculated by SLAMM, and measured values of WSL_M are in Mbytes.

			Ultra II		Pwr4		R14K	
matrix	SR	WSL_P	WSL_M	err	WSL_M	err	WSL_M	err
ex15	1.5	19.68	20.80	-5.4%	21.8	-9.7%	21.8	-9.7%
s1rmq4m1b	3.3	39.61	40.8	-2.9%	41.9	-5.5%	40.7	-2.7%
bcsstk17	5.6	66.94	64.0	4.6%	63.2	5.9%	64.6	3.6%
finan512	10.5	154.1	176.2	-12.6%	170.0	-9.4%	186.2	-17.2%

Because the cost of the sparse matrix vector product dominates the total cost of the CG algorithm, the accuracy of the CG prediction is similar to the accuracy of the MxV benchmark for the corresponding matrices. In particular, finan512 the matrix for which the $Mbytes_{L1}$ is low for the MxV benchmark, has a corresponding low $Mbytes_{L1}$ as well for the CG benchmark. However, the relative errors for all input matrices on all compute platforms are within the threshold of 20%.

6.3.3 Stagnation Test

In the previous section, we demonstrated that the SLAMM language processor accurately predicts $Mbytes_{L1}$ for the CG benchmark. We now examine how SLAMM is used to evaluate the impact of a particular algorithm feature. As mentioned in Section 6.3.1, the Pcg subroutine contains a test for stagnation of the solution. The stagnation test is included to check that the update to the new approximate solution $\alpha_j p_j$ in line 5 from Figure 3.1 has a norm greater than machine epsilon. Without this property, the accuracy of the approximate solution x_{j+1} never changes. While the stagnation test is present in the Pcg code written in Matlab, it is not present in the PETSc implementation of Conjugate Gradient. We use SLAMM to analyze the cost of the stagnation test in data movement. Table 6.14 contains the predicted $Mbytes_{L1}$ with and without the stagnation test in Pcg as well as the percentage increase for $Mbytes_{L1}$ for all input matrices. Table 6.14 indicates that the stagnation test increases $Mbytes_{L1}$ by 5 to 16%. The impact of the stagnation test is related to matrix density (ρ) which is provide in Table 6.3. The stagnation test has the lowest impact for the s1rmq4m1b matrix, which has the highest matrix density ($\rho = 8.7e - 3$), and the greatest impact for the finan512 matrix, which has the lowest matrix density ($\rho = 1.1e - 4$).

Table 6.14: Predicted increase in $Mbytes_{L1}$ for stagnation test in Pcg version of CG.

	Stagnation Test		
matrix	without	with	increase
ex15	19.68	21.93	11.4%
s1rmq4m1b	39.53	41.33	4.6%
bcsstk17	66.94	70.54	5.4%
finan512	154.1	178.7	15.9%

6.4 Generalized Minimal Residual Methods

Unlike the Conjugate Gradient algorithm, the restarted GMRES(m) algorithm, where m is the restart size, is not necessarily dominated by the cost of a sparse matrix-vector multiply but rather has a significant cost associated with vector computations. Because of the increased presence of vector operations, we expected the GMRES(m) benchmark to be more accurately predicted by SLAMM than is the CG benchmark. The Matlab codes for this section are: **GmresNL**, a GMRES code downloaded from the Netlib repository; **Gmres**, a GMRES code supplied by MathWorks; **Lgmres**, a LGMRES code that uses QR factorization to approximate the residual; **LgmresROT**, a LGMRES code that uses Givens rotation to approximate the residual; and **Blgmres**, a Matlab version of the B-LGMRES block algorithm [9]. The three LGMRES codes were supplied by Baker [7]. All codes are modified through the addition of SLAMM directives. Additionally, the GmresNL code was modified to include an optional preconditioner as with the corresponding CG code. Several lines of the Gmres code were modified to address a difficulty with the recognition of cache reuse for array notation. Optional components of the Lgmres and LgmresROT codes were turned off to match the PETSc versions. Table 6.15 provides a listing of the author of the code, the original number of lines, the final number of lines, and the number of lines modified with SLAMM directives and for other reasons.

Table 6.15: Code statistics of a family of GMRES algorithms written in Matlab. For the GMRES(30) algorithm, the type of Arnoldi process are indicated as either modified Gram-Schmidt (MGS) or Householder (House).

			Source Lines		Modifications	
	Algorithm	Author	Original	Final	%SLM	Other
GmresNL	GMRES(30) w/ ILU (MGS)	Netlib	93	136	7	40
Gmres	GMRES(30) w/ ILU (House)	Mathworks	472	491	10	11
Lgmres	LGMRES(29,1)	A. Baker	195	202	5	8
LgmresROT	LGMRES(29,1)	A. Baker	187	192	5	3
Blgmres	B-LGMRES(15,1)	A. Baker	149	155	5	3

6.4.1 GMRES

We use the GmresNL code to evaluate the accuracy of the SLAMM memory analysis because both the GmresNL code and the PETSc supplied GMRES are based on the same Arnoldi process. The Mathworks Gmres code uses a different Arnoldi process which impacts data movement. We evaluate the accuracy of the SLAMM memory analysis for the GMRES benchmark first, followed by an examination of the cost of different Arnoldi implementations.

As with the previous benchmarks, we execute the GMRES benchmark multiple times to reduce the impact of system variation and flush all caches prior to each execution. We record the cache line movement for one restart cycle of GMRES(30). The measured and predicted $Mbytes_{L1}$ and relative error for the GMRES(30) benchmark both with and without an incomplete LU factorization preconditioner are provided in Table 6.16 for a collection of input matrices. Note that the preconditioned results in

Table 6.16 include $MBytes_{L1}$ only for the application of the preconditioner and not for its calculation. As expected Table 6.16 indicates that SLAMM predicts data movement within 13% error.

Table 6.16: $Mbytes_{L1}$ for one restart cycle of the GMRES(30) benchmark. SR and WSL_P, calculated by the SLAMM language processor, and measured WSL_M are in Mbytes.

				Ultra II		Pwr4		R14K	
matrix	precon	SR	WSL_P	WSL_M	err	WSL_M	err	WSL_M	err
pesa	-	10.6	213.0	232.0	-8.2%	245.2	-13.1%	239.2	-11.0%
memplus	-	16.1	324.6	351.8	-7.7%	363.7	-10.8%	359.5	-9.7%
epb1	-	13.3	265.7	281.8	-5.7%	283.9	-6.4%	289.7	-7.3%
zhao2	-	29.9	590.9	642.3	-8.0%	590.9	0.0%	657.7	-10.2%
epb2	-	22.9	459.5	492.4	-6.7%	440.0	4.4%	498.7	-7.9%
wang3	ILU(0)	28.1	662.9	663.2	0.0%	687.2	-3.5%	696.4	-4.8%
sherman5	ILU(0)	3.5	82.4	78.1	5.6%	90.1	-8.5%	73.9	11.5%
epb1	ILU(0)	15.7	369.0	359.9	2.5%	379.6	-2.8%	373.3	-1.1%
epb2	ILU(0)	27.4	645.7	635.8	1.6%	654.7	-1.4%	656.3	-1.6%

Having verified that SLAMM accurately predicts $Mbytes_{L1}$ for the GMRES(30) algorithm, we now evaluate the impact certain algorithm design choices have on $Mbytes_{L1}$. In particular, we determine the impact of the choice of Arnoldi process on the $Mbytes_{L1}$ for GMRES(30). The Arnoldi process in the GmresNL code is based on modified Gram-Schmidt, while the Gmres code is Householder based. The Householder variant, while more accurate, does require additional flops [91]. The SLAMM language processor can analyze the cost in data movement. The predicted $Mbytes_{L1}$ for the modified Gram-Schmidt and Householder variants are provided in Table 6.17.

Note that the additional cost of the MGS-based versus the Householder-based Arnoldi process is consistent for all input matrices. Table 6.17 indicates that the Householder variant requires approximately 14% more data movement for non-preconditioned GMRES(30) and 11% more data movement for preconditioned GMRES(30).

Table 6.17: Predicted increase in $Mbytes_{L1}$ for Gram-Schmidt versus Householder based GMRES(30).

		Arnoldi Process		
Matrix	Precon	MGS	Householder	Increase
pesa	-	213.0	242.0	14%
memplus	-	265.7	302.0	14%
epb1	-	324.6	368.5	13%
zhao2	-	590.9	675.5	14%
epb2	-	459.5	521.9	14%
sherman5	ILU(0)	82.4	91.5	11%
wang3	ILU(0)	662.9	734.5	11%
epb1	ILU(0)	369.0	409.6	11%
epb2	ILU(0)	645.7	714.8	11%

6.4.2 LGMRES

We next examine LGMRES(m,s), a GMRES variant developed by Baker [10]. LGMRES(m,s), which augments the Krylov subspace with error approximation vectors, is described briefly in Section 3.4. The LgmresROT code, which is used in the comparison with the PETSc version of LGMRES, estimates the residual using Givens rotations, while the Lgmres version uses a QR factorization of the Hessenberg matrix. We evaluate the accuracy of the SLAMM memory analysis first, followed by a comparison between the two residual estimation schemes.

We record the cache line movement for one restart cycle of LGMRES(29,1). The measured and predicted $Mbytes_{L1}$ and relative error for the LGMRES(29,1) benchmark are provided in Table 6.18 for a collection of input matrices. We provide results only for non-preconditioned problems because we have previously established that preconditioning does not affect the accuracy of the memory analysis. As with the GMRES(30) benchmark, SLAMM accurately predicts $Mbytes_{L1}$ for LGMRES(29,1) to within 11% error.

We now examine the impact of residual estimation on data movement. The GMRES algorithm does not explicitly provide the residual for each iteration. The residual is approximated using either a QR factorization of the Hessenberg matrix or Givens ro-

Table 6.18: $Mbytes_{L1}$ for one restart cycle of the LGMRES(29,1) benchmark. SR and WSL_P, calculated by the SLAMM language processork, and measured WSL_M are in Mbytes.

			Ultra II		Pwr4		R14K	
Matrix	SR	WSL_P	WSL_M	err	WSL_M	err	WSL_M	err
pesa	4.4	215.8	233.8	-7.7%	246.3	-12.4%	241.9	-10.8%
epb1	5.5	269.1	284.6	-5.5%	286.9	-6.2%	290.5	-7.4%
memplus	6.8	328.6	354.7	-7.4%	366.7	-10.4%	363.6	-9.6%
zhao2	12.1	598.6	648.9	-7.7%	600.3	-0.3%	666.1	-10.1%
epb2	9.6	465.3	496.9	-6.4%	447.1	4.1%	504.6	-7.8%

tations. Givens rotations replace the more expensive factorization with multiplication by small 2x2 rotation matrices. The impact of the Givens rotations on data movement is easily quantified with the SLAMM language processor for the LGMRES(29,1) algorithm. In Table 6.19, we compare the predicted $Mbytes_{L1}$ for the Lgmres code, which is based on the QR factorization, versus the LgmresROT code, which is based on the Givens rotations. It is apparent from Table 6.19 that the Givens rotations have a significant impact on data movement, reducing total $Mbytes_{L1}$ by approximately 17%.

Table 6.19: Predicted decrease in $Mbytes_{L1}$ for a Givens rotation versus a QR factorization for residual estimation in the LGMRES(29,1) algorithm.

	Residual Estimation		
Matrix	QR fact	Givens rot	decrease
pesa	259.9	215.8	17.0%
epb1	324.4	269.1	17.1%
memplus	395.4	328.7	16.9%
zhao2	725.9	598.6	17.5%
epb2	560.1	465.3	16.9%

6.4.3 B-LGMRES

Finally, we examine B-LGMRES(m,s), a block version of the LGMRES(m,s) algorithm, which is described in Section 3.4. This is the same iterative algorithm for which the manual memory analysis is described in Chapter 4. We revisit the algorithm to verify the accuracy of the SLAMM-based memory analysis.

We provide both the predicted and measured $Mbytes_{L1}$ for the B-LGMRES(15,1) algorithm and the relative accuracy for each of the primary compute platforms in Table 6.20. The results in Table 6.20 demonstrate that as with all the other GMRES variants, the SLAMM language processor accurately predicts $Mbytes_{L1}$ within 20% accuracy for the B-LGMRES(15,1) benchmark.

Table 6.20: $Mbytes_{L1}$ for one restart cycle of the B-LGMRES(15,1) benchmark. SR and WSL_P, calculated by the SLAMM language processor, and measured values of WSL_M are in Mbytes.

			Ultra II		Pwr4		R14K	
Matrix	SR	WSL_P	WSL_M	err	WSL_M	err	WSL_M	err
pesa	3.5	237.6	280	-15.1%	285	-16.6%	289	-17.7%
epb1	4.3	295.4	336	-12.1%	318	-7.1%	342	-13.6%
memplus	5.3	362.8	426	-14.8%	427	-15.0%	435	-16.6%
zhao2	9.3	647.6	781	-17.1%	729	-11.1%	805	-19.5%
epb2	7.5	512.9	588	-12.7%	551	-6.9%	597	-14.1%

Chapter 7

Reducing Solver Costs in HOMME

Chapter 6 revealed that the automated memory analysis provided by the SLAMM language processor accurately predicts $Mbytes_{L1}$ for both simple linear algebra kernels and complete iterative algorithms. We next provide a case study on how we apply SLAMM to examine the impact of different solvers on data movement in the Higher Order Method Modeling Environment (HOMME) [105]. HOMME is a prototype spectral element atmospheric model that uses a preconditioned Conjugate Gradient algorithm to solve a Helmholz problem for each vertical level. The existing Helmholtz problem is modified to isolate the vertical and horizontal components, which allows the use of the multishifted Conjugate Gradient (mCG) algorithm described in Section 3.2. The mCG algorithm allows the use of a single level independent preconditioner, which has the potential to reduce data movement. Unfortunately, HOMME currently supports only an inexpensive diagonal scaled preconditioner and therefore mCG does not significantly reduce data movement versus the existing CG solver.

However, recent developments indicate that a restrictive optimized Schwarz preconditioner significantly reduces iteration counts in HOMME versus the diagonal scaled preconditioner. Because we anticipate significant changes to the HOMME solver, we apply SLAMM to predict the impact on data movement of a multishifted solver and a nonsymmetric solver in HOMME. Because the implementation of a multishifted nonsymmetric solver in HOMME is beyond the scope of this thesis, we evaluate the mCG solver as an example of a multishifted solver.

We first describe the Helmholtz problem and the modifications necessary to take advantage of mCG algorithm in Section 7.1. In Section 7.2, we present numerical results for HOMME based on the existing CG and mCG algorithms. In Sections 7.3.1 and 7.3.2, we provide a case study of how SLAMM is used to reduce data movement of the mCG algorithm. Finally, in Section 7.3.3, we predict the impact of several nonsymmetric Krylov algorithms on data movement in HOMME.

7.1 HOMME

The spectral element method is a promising horizontal discretization scheme for the construction of dynamical cores of climate models. The computational domain is subdivided into spectral elements where the model fields are approximated by high order polynomials. C^0 continuity is imposed along element boundaries that share degrees of freedom. In HOMME, which is based on the spectral element atmospheric model originally developed by Taylor et al. [101], the sphere is tiled with rectangular elements by subdividing the six faces of the cube that circumscribes the sphere, then a gnomonic projection maps these elements onto the surface of the sphere.

The cubed-sphere computational domain is illustrated in Figure 7.1, where each cube face contains an array of $N_e \times N_e$ quadrilateral spectral elements. Global variables are defined on a single pressure grid (N_p^2 grid points per spectral element). The semi-implicit, time-stepping version of HOMME, which is based on a combination of Crank-Nicholson and an explicit leap frog scheme, uses a preconditioned CG algorithm to solve a "Helmholtz like" problem for a horizontal level that contains a total of $6 \times N_e^2 \times N_p^2$ grid points. Note that this is not a proper Helmholtz problem because we never form the Laplacian, but rather a "Laplacian-like" operator. With a slight abuse of convention, we refer to the "Helmholtz-like" problem as the Helmholtz problem hereafter.

There are several versions of the vertical formulation. We concentrate on the

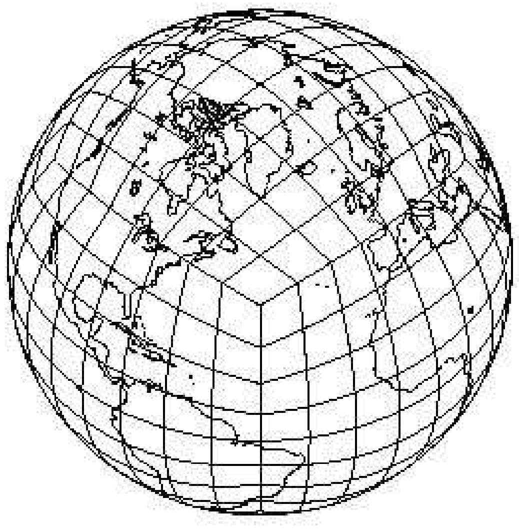


Figure 7.1: The cubed-sphere with continental outline for $N_e = 8$

primitive equation version [37] that contains multiple coupled vertical levels discretized with a finite difference approximation. We do not provide a derivation of the linearized Helmholtz operator for the primitive equations, but rather refer the interested reader to [105]. Instead, we concentrate on two different forms of the Helmholtz problem. The mass form of the Helmholtz problem (mForm) for the k^{th} vertical level is

$$(\mathcal{M} - \Delta t^2 \lambda_k \mathcal{M} \nabla^2)\Gamma_k = \mathcal{M} C_k, \tag{7.1}$$

where Δt is the timestep in seconds, λ_k is an eigenvalue of the vertical structure matrix, $\mathcal{M}$ is the mass matrix, Γ_k is a generalized pressure variable, and $C_k \in \mathbb{R}^{6{N_e}^2{N_p}^2}$ is defined in [105]. Note that the level-dependent variables are subscripted by k.

The structure of (7.1) is however not compatible with the mCG algorithm because the horizontal and vertical components of the Helmholtz problem are not separated. While it is possible to separate the horizontal and vertical components by factoring out the mass matrix, preliminary numerical results indicate the mass matrix acts as a preconditioner for the system. However, because $\mathcal{M}^{-1}$ exists, it is possible to use right preconditioning to transform (7.1) into

$$(I - \Delta t^2 \lambda_k \mathcal{M} \nabla^2 \mathcal{M}^{-1}) \tilde{\Gamma_k} = \mathcal{M} C_k, \tag{7.2}$$

where $\tilde{\Gamma_k} = \mathcal{M}\Gamma_k$. A Helmholtz problem suitable for the mCG algorithm is obtained by dividing (7.2) by $\Delta t^2 \lambda_k$ to obtain the shifted form (sForm) of the Helmholtz problem

$$(\tilde{\sigma_k} I - \mathcal{M} \nabla^2 \mathcal{M}^{-1}) \tilde{\Gamma_k} = \mathcal{M} \tilde{C_k}, \tag{7.3}$$

where $\tilde{\sigma_k} = 1/\Delta t^2 \lambda_k$ and $\tilde{C_k} = \tilde{\sigma_k} C_k$.

7.2 Numerical Impact of mCG Solver in HOMME

We next examine if a mCG-based HOMME correctly reproduces an atmospheric test problem. The Polvani-Scott-Thomas (PST) baroclinic instability test problem [85] simulates the impact of gravity waves in an atmospheric flow over 12 model days. The reproducibility of the PST test problem is useful for evaluating and debugging atmospheric dynamical cores. We examine the PST test problem using both the CG and mCG solvers. For the remainder of the chapter, the Conjugate Gradient algorithm refers to the merged inner-product Conjugate Gradient algorithm provided in Figure 3.2.

We execute the baroclinic instability test problem at a resolution of $N_e = 5$, $N_p = 8$, and $nlev = 20$ with $\Delta t = 720$ seconds using a scaled diagonal preconditioner. Each timestep involves both a linear solve for each vertical level and the associated update of the governing equations. Formal convergence of the PST test problem is achieved when the norm of the vorticity ($\| \zeta \|_2$) or angular momentum of the flow at the surface after 12 days equals $7.8E - 6$. Additionally, a converged solution also reproduces a particular structure in a plot of the surface vorticity. While the $N_e = 5$ resolution is slightly lower than the $N_e = 9$ resolution necessary for formal convergence of the PST test problem, it is sufficient to test solver characteristics while minimizing computational costs. Note that for the remainder of the section, convergence refers to the convergence of HOMME to a physically valid solution and not to the convergence of the iterative solver.

To determine the maximum solver tolerance necessary to achieve convergence, we apply a test suggested by Smolarkiewicz [97]. In particular, we calculate the norm of the divergence ($\| \nabla \|_2$) of the flow at 12 days for different solver tolerances. We expect that as the solver tolerance is decreased, $\| \nabla \|_2$ converges to a single value. A plot of $\| \nabla \|_2$ as a function of solver tolerance is provided in Figure 7.2 for the mForm (7.1) and sForm (7.3) of the Helmholtz problem using the CG solver and for the sForm of the Helmholtz problem using the mCG solver. The plot in Figure 7.2 illustrates that both the CG and mCG solvers using a scaled diagonal preconditioner converge to a single value for $\| \nabla \|_2$. We use a solver tolerance of $1E-7$ because a solver tolerance lower than $1E-7$ does not significantly improve the quality of the simulation.

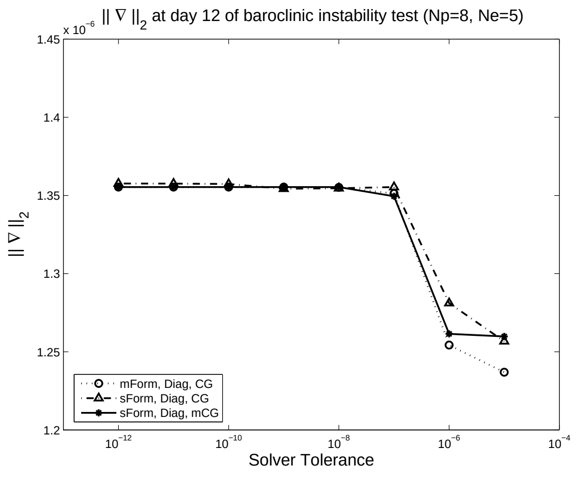


Figure 7.2: The norm of the divergence ($\| \nabla \|_2$) at the surface at day 12 for HOMME at resolution $N_e = 5$, $N_p = 8$, and $nlev = 20$ for the both CG and mCG algorithms.

We next calculate the norm of the vorticity at the surface after 12 days. For both iterative solvers $\| \zeta \|_2 = 8.0E-6$, which is sufficiently close to the converged value of $7.8E-6$ and indicates that both the CG and mCG algorithms generate comparable solutions. For further verification of the mCG algorithm, we plot surface vorticity.

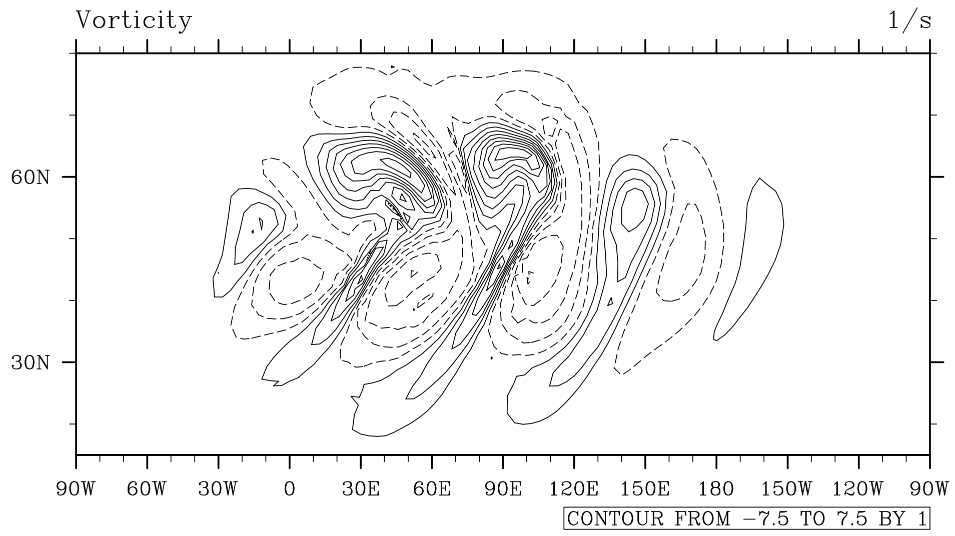

Vorticity
1/s
60N
30N
90W 60W 30W 0 30E 60E 90E 120E 150E 180 150W 120W 90W
CONTOUR FROM −7.5 TO 7.5 BY 1

Figure 7.3: The surface vorticity at day 12 for HOMME at resolution $N_e = 5$, $N_p = 8$, and $nlev = 20$ with the shifted form of the Helmholtz problem using the mCG solver.

Figure 7.3 contains a plot of surface vorticity with contours from -7.5 to $7.5 \times 10^{-5}s^{-1}$ in steps of $1 \times 10^{-5}s^{-1}$ after 12 days of simulation using the mCG algorithm. Figure 7.3 matches the equivalent plot in [85], which is sufficient confirmation that a mCG-based HOMME successfully reproduces the baroclinic instability test problem.

7.3 Alternative Krylov Solvers in HOMME

In the previous section, we verify that HOMME reproduces a standard atmospheric test problem using either the CG or mCG solver. We next determine whether the mCG algorithm allows a reduction in data movement over the CG algorithm. The mCG algorithm potentially reduces data movement because it requires only a single coefficient matrix and preconditioner.

Because the mCG algorithm does not require a linear system for each shift value or vertical level, it avoids the need for multiple coefficient matrices. This feature offers a significant reduction in data movement for the matrix-vector product when the coefficient matrices are explicitly stored in memory. However, HOMME uses a matrix-free implementation that applies functions to calculate the matrix-vector product. Further, the matrix-free implementation prevents the calculation of the new coefficient matrix $\tilde{A} = AM$ in Figure 3.3 necessary for right preconditioning at initialization. Instead, the preconditioner M in Figure 3.3 must be applied at each Krylov iteration. Consequently, the mCG matrix-vector product is more expensive than the CG matrix-vector product in HOMME.

The mCG algorithm also allows the use of a single preconditioner for all shift values if the preconditioner maintains the shifted structure. This feature may have a significant impact on data movement if the cost of applying the preconditioner is considerable. Because HOMME currently supports only a scaled diagonal preconditioner, which is inexpensive to apply, we do not expect the mCG algorithm to provide any reduction in execution time for HOMME at this time. However, recent developments by St-Cyr [100, 99] indicate that a restrictive optimized Schwarz preconditioner (o-Schwarz) significantly reduces iteration count versus the scaled diagonal preconditioner. The o-Schwarz preconditioner is stored as $6N_e^2$ small dense matrices $M \in \mathbb{R}^{N_P^2 \times N_P^2}$. The matrix M is explicitly inverted at initialization, and its inverse is subsequently applied as a matrix-vector multiply. The o-Schwarz preconditioner, unlike the scaled diagonal preconditioner, is expensive to apply. Unfortunately, the o-Schwarz preconditioner requires a nonsymmetric solver, which makes its use incompatible with the mCG solver.

While the implementation of a multishifted nonsymmetric solver in HOMME is beyond the scope of this thesis, it is possible to compare the data movement between the CG and mCG algorithms. This evaluation provides useful lessons about the potential advantage of a multishifted nonsymmetric solver and demonstrates how SLAMM is used to improve an implementation's memory efficiency. Additionally, we apply SLAMM to evaluate the impact of several nonsymmetric solvers on data movement in HOMME. In Section 7.3.1, we examine how SLAMM is used to improve the implementation of the mCG algorithm. In Section 7.3.2 we compare the execution time of HOMME using both the mCG and CG algorithms. Finally, in Section 7.3.3, we use SLAMM to predict the data movement for several nonsymmetric solvers using an o-Schwarz preconditioner versus the existing CG solver and preconditioner for a fixed number of iterations.

7.3.1 Reducing Data Movement in the mCG Algorithm

We next demonstrate how SLAMM is used to improve the implementation of the mCG algorithm in HOMME. We examine the memory efficiency of three implementations of the mCG algorithm: the initial mCG implementation (mCG-v1), an intermediate version with loop reordering (mCG-v2), and the final version (mCG-v3). We configure HOMME at a resolution $N_e = 3$, $N_p = 6$, and $nlev = 20$ and explicitly output a matrix for use by Matlab. We concentrate on two sections of the mCG algorithm. The **Precon** section (line 7) of Figure 3.3, where the preconditioner is applied, and the **Core** section (lines 5 to 7 and 8 to 15) in Figure 3.3. We ignore the matrix-free matrix-vector multiply in line 4 of Figure 3.3 for which we do not have a Matlab analog.

To simulate the impact of the o-Schwarz preconditioner on the memory hierarchy, we apply the scaled diagonal preconditioner as a dense matrix-vector multiply. In Table 7.1, we provide the predicted values WSL_P and the measured values WSL_M for the different implementations of the mCG algorithm for both the Precon and Core sections. We examine the Precon section first followed by the Core section.

Table 7.1: Predicted versus measured $Mbytes_{L1}$ for three different versions of the mCG algorithm in HOMME for 10 iterations for $N_e = 3$, $N_p = 6$, and $nlev = 20$ on Pwr4. Values for WSL_P and WSL_M are in Mbytes.

Code Section	WSL_P	WSL_M		
		mCG-v1	mCG-v2	mCG-v3
Precon	15.02	243.8	23.6	12.3
Core	50.51	60.8	60.8	39.5

To facilitate the examination of data movement in the Precon section, we provide a listing of the Precon section of HOMME in Figure 7.4. The variable k is the level index and ie is the element index. Table 7.1 indicates that there is a rather large discrepancy between the predicted (15.0) and measured (243.8) value of $Mbtyes_{L1}$ for the mCG-v1 implementation. The reason for the discrepancy is clear if we examine the Precon code for the mCG-v1 implementation in Figure 7.4. While we reuse the same preconditioner for each level, as indicated by the $cg\%state(ie)\%M$ variable, the ordering of the loops for the mCG-v1 code prevent cache reuse. The preconditioner for a particular element is flushed from the cache before it can be used again because k is not the inner loop. The simple fix, seen in the mCG-v2 version of the loop in Figure 7.4, reverses the k and ie loops. The index reversal result in cache reuse for the preconditioner, and the WSL_M for the mCG-v2 implementation drops to 23.6 Mbytes.

While significantly reduced, the mCG-v2 value of 23.6 is still higher than the predicted value of 15.0. The second fix for the Precon section involves removing the use of array syntax from the subroutine call to *ApplyPrecon*. This change removes an input argument copy that some F90 compilers generate if the form of the array declarations is not matched on each side of the subroutine interface. With the spurious copies removed, WSL_M for mCG-v3 drops to 12.3 Mbytes, which is lower than the predicted value of 15.0. The discrepancy is not a failure of SLAMM, but rather a mismatch between the implementation of the preconditioner in the Matlab test code and HOMME. In Matlab, the preconditioner is implemented as a general sparse matrix that includes the extra storage associated with indirect addressing, while the preconditioner in HOMME is implemented as an array of structures. If we calculate WSL_P based on the data structure configuration used in HOMME, the predicted value 11.4 Mbytes agrees to within 8% of the measured value. We did not encounter discrepancies between the form of the data structures in Chapter 6 because the PETSc AIJ sparse matrix storage format is similar in size to the Matlab data structures. Errors in the predictions due to differences in data structures are unavoidable. However, the discrepancy in data structures did not prevent the identification of both serious and subtle performance problems with the Precon section of the mCG algorithm.

We next examine the impact of the Core section of the mCG algorithm on the memory hierarchy. Because of the large discrepancy between the predicted and measured for the Precon in mCG-v1 of the mCG algorithm, we ignore the discrepancy for the Core section until the creation of the mCG-v3 implementation. For the mCG-v3 implementation, we reversed the order of the two inner loops for all the vector updates in the Core section. The loop reversal matches the FORTRAN array storage order. The WSL_M for the mCG-v3 implementation of the Core section drops to 39.5 Mbytes, which is lower than the predicted value of 50.5 Mbytes. This discrepancy is caused by the interaction between a particular design choice of the SLAMM language processor and the implementation characteristics of the mCG algorithm in HOMME. Note that vector updates for $\tilde{x}^{\sigma}_{j+1}$ and $\tilde{p}^{\sigma}_{j+1}$ in lines 14 and 15 of Figure 3.3 both require the vector $\tilde{p}^{\sigma}_{j}$. The mCG-v3 implementations of both updates are merged into a single loop, which allows cache reuse for the vector $\tilde{p}^{\sigma}_{j}$. However, recall from Section 5.8.2 that the SLAMM language processor only accounts for cache reuse within a single statement. Because the Matlab version of the mCG algorithm contains two separate lines, SLAMM does not identify the cache reuse and as a result overestimates $Mbytes_{L1}$. Reuse between different statements is not an issue in Chapter 6 because of the modular design of PETSc. It is possible to address cache reuse between multiple statements through an extension to the SLAMM directives. The extension would allow the user to indicate a group of statements where cache reuse is likely.

In this case, the failure of SLAMM to recognize cache reuse between statements did not prevent the successful use of SLAMM to identify performance problems in the mCG solver in HOMME. We next examine if the mCG algorithms provide a reduction in execution time for HOMME versus the CG algorithm.

```
!====================
! SCG V1
!====================
do k=1,nlev
  if(.not. cg%converged(k)) then
    do ie=1,nelem
      ...
      call ApplyPrecon(cg%state(ie)%r(:,k), &
                       cg%state(ie)%z(:,k), &
                       cg%state(ie)%M,npsq)
    enddo
  endif
enddo

!====================
! SCG V2
!====================
do ie=1,nelem
  do k=1,nlev
    if(.not. cg%converged(k)) then
      call ApplyPrecon(cg%state(ie)%r(:,k), &
                       cg%state(ie)%z(:,k), &
                       cg%state(ie)%M,npsq)
    endif
  enddo
enddo

!====================
! SCG V3
!====================
do ie=1,nelem
  do k=1,nlev
    if(.not. cg%converged(k)) then
      call ApplyPrecon(cg%state(ie)%r(1,k), &
                       cg%state(ie)%z(1,k), &
                       cg%state(ie)%M,npsq)
    endif
  enddo
enddo
```

Figure 7.4: The Precon section of the multishifted conjugate gradient in HOMME.

7.3.2 Impact of the mCG Algorithm on Execution Time

We next compare the data movement and execution time of HOMME for both the CG and mCG algorithms. To provide a fair comparison, we apply any applicable code changes identified by SLAMM during the optimization of the mCG solver to the CG solver. As in the previous section, we apply the scaled diagonal preconditioner as a dense matrix-vector multiply. In Table 7.2, we provide the measured working set load size (WSL_M), floating-point operator count ($Flops_M$), and execution time (T_M) for 10 iterations of the CG and mCG solvers in HOMME at the $N_e = 5$, $N_p = 8$, $nlev = 20$ resolution on the Pwr4 compute platform. The solver is broken into the previously discussed Precon and Core sections and the Helm and Solution sections. The **Helm** section is where the Helmholtz operator is applied and the **Solution** section is line 17 of Figure 3.3, where true solution is calculated.

Table 7.2: Measured $Mbytes_{L1}$ (WSL_M) in Mbytes and $Flops_M$ in Mflops for 10 iterations of CG and mCG algorithms at the $N_e = 5$, $N_p = 8$, and $nlev = 20$ resolution on Pwr4.

	Solver						Reduction		
	CG			mCG			mCG vs. CG		
	WSL_M	$Flops_M$	T_M	WSL_M	$Flops_M$	T_M	WSL_M	$Flops_M$	T_M
Precon	1170	250	628	561	249	200	52%	0%	68%
Core	126	35	86	196	40	82	-55%	-17%	5%
Helm	780	276	290	955	527	431	-22%	-91%	-49%
Solution				85	26	25			
Total	2077	561	1004	1798	816	738	14%	-50%	26%

As expected, column 8 of table 7.2 indicates that the mCG solver reduces data movement versus the CG solver by 52% for the Precon section. Unfortunately, the reduction in data movement for the Precon section is matched by an increase in data movement for both the Core and Helm sections of the code. As a result, the total reduction in data movement for the mCG solver versus the CG solver is 14%. Further, the addition of right preconditioning increases the required floating-point operations by 50% for the mCG algorithm. Despite the significant increase in required floating-point operations, the mCG algorithm has an execution time that is 26% lower than the CG algorithm. The values in Table 7.2 represent an upper bound on the advantage of the mCG algorithm versus the CG algorithm in HOMME. In practice, the advantage of the mCG algorithm is diluted because not all vertical levels require the same number of iterations.

In Table 7.3, we provide the average execution time per timestep for the first day of the PST test problem for HOMME on the Pwr4 compute platform. The time in the solver is broken into the previously discussed Precon, Core, Helm, and Solution sections. We also provide the time to update the governing equations **Advance** and the average iteration count for each solver. The value $iters_{max}$ is the maximum number of iterations required for any level, while the value $iters_{total}$ is the total number of iterations required for all levels. The average number of iterations per level is $iter_{avg} = iter_{total}/nlev$.

Table 7.3 indicates that in practice, the execution time for the solver component of HOMME for the mCG algorithm is 15% less than for the CG algorithm. The reason the mCG solver does not achieve the 26% reduction in execution time is due to the particular number of iterations required by each level. Because the average number of iterations required for each level is approximately three, only few levels require the maximum number of iterations to converge. Because HOMME does not apply the preconditioner to those levels that have converged, the advantage of the mCG algorithm is only significant for the first three iterations. For the small number of remaining levels, the potential reduction in data movement for the mCG algorithm in the Precon section is reduced.

Our experience evaluating the CG and mCG algorithm provides several lessons. First, while the mCG algorithm reduces the required data movement, the use of right preconditioning increases both floating-point and memory access costs of the matrix-vector multiply for matrix-free implementations. Second, multishifted algorithms are most effective when all levels converge in a similar number of iterations. The comparison

Table 7.3: Average execution time per timestep in μseconds for 120 timesteps for HOMME at $N_e = 5$, $N_p = 8$, and $nlev = 20$ on Pwr4.

		Solver		Reduction
		CG	mCG	mCG vs. CG
Solver:		419.1	355.6	15%
	Precon	220.6	89.2	59.6%
	Core	29.9	27.2	9.0%
	Helm	168.6	216.5	-28.4%
	Solution		22.7	
Advance:		571.4	559.3	2.1%
Total:		990.5	914.9	7.6%
	$iters_{max}$	11.96	11.63	
	$iters_{total}$	67.80	64.19	
	$iters_{avg}$	3.39	3.21	

of the CG and mCG algorithms provides an unambiguous demonstration of the critical importance of memory access costs in the time to solution of an iterative algorithm

7.3.3 Nonsymmetric Solvers in HOMME

We next examine how SLAMM is used to evaluate modifications to the HOMME solver. We predict data movement for a fixed number of iterations for four different Krylov and preconditioner combinations. We apply SLAMM to four Matlab codes. The code **CgHom**, written by the author, is the merged inner-product Conjugate Gradient algorithm provided in Figure 3.2. The **CgsNL** code is the Conjugate Gradient Squared [98] algorithm from the templates directory of Netlib. The **BicgstabNL** code is the Bi-Conjugate Gradient algorithm [110] from the templates directory of Netlib. The **GmresNL** code is the same GMRES algorithm used in Chapter 6 from the templates directory of Netlib. Because we are interested in data movement and not the numerical impact of the various preconditioners, we use a scaled diagonal preconditioner for each Matlab code. For the CgHom code, we apply the preconditioner as the multiplication of two vectors. For the remaining codes, the preconditioner is applied as a dense matrix-vector multiply.

We use the explicitly output matrix from HOMME described in Section 7.3.1 as input for the four Matlab codes. We provide the working set load size (WSL) for both the Precon, Helm, and Core sections in Table 7.4 for ten iterations. The **Precon** section refers to application of the preconditioner; the **Helm** section refers to the application of the Helmholtz operator; and the **Core** section refers to all other necessary calculations. Note that while the WSL values for the Precon and Core sections are predicted using the SLAMM language processor, the WSL value for Helm is an approximation based on the measured value of the existing CG implementation. Table 7.4 provides the total value of WSL along with a ratio of the nonsymmetric algorithm WSL divided by working set size for the existing CG algorithm (WSL^{CG}). The ratio of 3.19 for the CgsNL code indicates that CGS requires 3.19 times the data movement of the CG implementation per iteration. The WSL/WSL^{CG} ratio also indicates that the CGS with o-Schwarz preconditioner must reduce iteration count by a factor greater than 3.19 to have greater memory efficiency than the existing CG solver.

Table 7.4: Predicted $Mbytes_{L1}$ for the several Krylov algorithm and preconditioner combinations in HOMME for 10 iteration at resolution $N_e = 3$, $N_p = 6$, and $nlev = 20$.

	WSL			
Code	CgHom	CgsNL	BicgstabNL	GmresNL
Algorithm	CG	CGS	BiCGStab	GMRES(10)
Precon type	Diag	MxV	MxV	MxV
Precon	6.6	375.2	375.2	203
Helm	188.7	377.5	377.5	188.7
Core	54.8	46.0	58.4	114.6
Total	250.2	798.7	811.1	505.3
Ratio: WSL/WSL^{CG}	1.00	3.19	3.24	2.02

Chapter 8

Conclusions

Because of advances in computer architecture, the cost of iterative solvers is no longer dominated by floating-point costs. Thus to reduce time to solution, we must focus not only on the floating-point costs but on the memory access costs as well. We have shown that the memory efficiency of iterative algorithms can be improved by analyzing the impact on the memory hierarchy during the design process using memory analysis. An *a priori* memory analysis, which determines the amount of data that must be loaded from the memory hierarchy to the L1 cache, is performed using either manual or automated techniques. We demonstrated in Chapter 4 that an manual memory analysis improves both the design and implementation of an iterative algorithm. However, manual memory analysis is a laborious, error-prone process that is too complex to perform on a regular basis. We need a simpler memory analysis procedure.

To simplify the memory analysis procedure, we develop the Sparse Linear Algebra Memory Model (SLAMM) language processor described in Chapter 5. The SLAMM language processor combines the static analysis of an input Matlab code and dynamic execution by the Matlab interpreter to automate memory analysis. The static analysis of the SLAMM language processor determines the number of occurrences of identifiers in the input code. Because not all occurrences of identifiers indicate a need to load data from the memory hierarchy, we apply a series of corrections to the base identifier counts. The corrections represent a translation from the literal analysis of the input Matlab code to an estimate of how the code would be efficiently implemented in a compiled language.

In Chapter 6, we demonstrate that the SLAMM language processor predicts the amount of data loaded from the memory hierarchy to the L1 cache ($Mbytes_{L1}$) to within 20% error for nearly every benchmark on three different compute platforms. A prediction accuracy within 20% error is consistent with related efforts to model the impact of algorithms on the memory hierarchy using source code analysis [19, 39, 111]. The accurate prediction of $Mbytes_{L1}$ for the matrix-matrix multiply (MxM) benchmark is more difficult. The MxM benchmark is difficult to predict because the measured $Mbytes_{L1}$ is highly dependent on both compute platform and the underlying implementation of the benchmark. We also examine the ability of SLAMM to accurately predict execution time for a subset of the benchmarks. We demonstrate that under controlled conditions, we are able to accurately predict an upper bound on execution time correctly for 92% of the benchmark, input set, and compute platform combinations. Our accuracy rate drops to 61% for our lower bound on execution time.

We next provide a case study of how SLAMM is used to evaluate the memory efficiency of different solvers within the context of the Higher Order Method Model Environment (HOMME). We examine the memory efficiency of the HOMME solver based on a multishifted Conjugate Gradient (mCG) algorithm. The mCG algorithm, which is developed for this work, provides the ability to use a single preconditioner for all vertical levels. Because HOMME currently supports only an inexpensive diagonal scaled preconditioner, the mCG algorithm offers no significant reduction in data movement at this time.

However, recent developments with an optimized Schwarz preconditioner suggest that a multishifted Krylov algorithm may be useful in the future. We therefore examine the differences in memory efficiency of the CG and mCG algorithms as an example of the potential advantage of a multishifted algorithm. We demonstrate that the prediction accuracy provided by SLAMM is sufficient to easily identify excessive data movement in algorithm implementations. We discover that the use of the right preconditioning, which is necessary to maintain the shifted structure of the Helmholtz problem, significantly increases the floating-point count for the mCG algorithm versus the CG algorithm. Despite the increase in the floating-point count, the mCG algorithm reduces data movement and time to solution versus the CG algorithm. The decreased time to solution for the mCG-based HOMME is an unambiguous demonstration of the importance of memory access costs to the time to solution of iterative algorithms.

Finally, we simulate the required data movement for an existing CG solver with the scaled diagonal preconditioner with several nonsymmetric algorithms using an optimized Schwarz preconditioner. We determine that a nonsymmetric algorithm with an optimized Schwarz preconditioner must reduce iteration count by more than a factor of two to three to be a viable alternative to the existing CG algorithm with a scaled diagonal preconditioner.

8.1 Future Work

Our results indicate that SLAMM has the potential to be a useful predictive tool for the numerical analysis community. It allows rapid evaluation of the performance impact of different solver design choices. During the course of this research, we used three different techniques to evaluate the impact of design choices on data movement. The first technique, which involves implementing the algorithm in a compiled language and evaluating the memory efficiency of the implementation, requires approximately two to three months of effort. The second technique, manual memory analysis, requires as long as several days of work. The third technique, automated memory analysis with the SLAMM language processor, requires 20 minutes. Automated memory analysis therefore significantly simplifies the development of memory-efficient iterative solvers. Further, SLAMM provides the ability to improve the memory efficiency of an existing implementation.

Our results suggest several avenues for further work. The first involves the continued development and improvement of the SLAMM language processor. The second involves the application of SLAMM to improve solvers in other contexts. We discuss the possible improvements to the SLAMM language processor first.

Improvements to the SLAMM language processor are classified into two types: those that increase robustness and those that address more fundamental issues. We begin with improvements to increase robustness. During the development of SLAMM, a decision was made not to initially support Matlab structures. This decision was based on how often structures appeared in a collection of Matlab codes downloaded from the Netlib repository. The addition of structure support would increase the robustness of SLAMM and requires only a slight reworking of the form of the abstract syntax tree.

A minor restructuring of the abstract syntax tree would also allow the proper identification of cache reuse for expressions using array notation. An improved treatment of array notation has the potential to reduce the number of copies of array subsections into temporary variables in the SLAMM generated Matlab. The reduction in array subsection copies could reduce the execution time overhead for the SLAMM generated Matlab.

The SLAMM language processor also requires work to address more fundamental issues that were discovered while analyzing the accuracy of the predictions in Chapters 6 and 7. As described in Section 7.3.2, SLAMM's inability to recognize cache reuse across multiple statements leads to an overestimation of the required data movement under certain circumstances. This deficiency is easily remedied through the addition of an optional keyword for the SLAMM Start directive. This extension would alert SLAMM to cache reuse within the indicated body. An incompatability between the duplicate and special operation corrections also impairs SLAMM's ability to recognize cache reuse. The special operation correction, which performs the same code transformations necessary for profiled function call support described in Section 5.8.4, prevents the proper identification of duplicate identifiers. We believe that while it is possible to support both forms of corrections simultaneous, it requires additional syntax tree computations. Finally, we believe it is possible to use ATLAS to improve the estimate for data movement for dense matrix-matrix multiplication.

The results for the execution time prediction also require further investigation. Our ability to predict execution time under controlled conditions provides neither confidence in the results nor proof of failure. Further, it is unclear if the accurate prediction of execution time for complete applications for which the location of the operands in the memory hierarchy is unknown is possible. We suspect that prediction of execution time for a complete application requires an accurate prediction of data movement through other components of the memory hierarchy. Finally, as our results for the mCG algorithm in Section 7.3.2 indicate, the cost of floating-point calculations can not be completely ignored.

While the mCG solver offers a modest reduction in execution time versus the CG solver in HOMME, it does demonstrate the importance of memory access costs on the time to solution. Therefore, the use of multishifted algorithms to reduce time to solution should be investigated further. In particular, the potential use of a nonsymmetric solver in HOMME suggests the need to examine a multishifted nonsymmetric solver. The multishifted Bi-CGSTAB of Frommer [46] appears a likely candidate for evaluation.

Finally, we would like to use SLAMM to reduce execution time for applications other than HOMME. During the development and performance tuning of the mCG algorithm in HOMME, we observed that SLAMM allows for a rapid identification of sections of code with excessive data movement. We believe this is a property of SLAMM and not due to the familiarity of the author with the HOMME application. However, we would like to verify the ability of SLAMM to rapidly identify performance problems in other applications.

Bibliography

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In Proceedings of Supercomputing '92, November 1992.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair. Improving performance of linear algebra algorithms for dense matrices using algorithmic prefetch. IBM Journal of Research and Development, 38(3):265–275, 1994.

[3] G. Almási, C. Cascaval, and D. Padua. Calculating stack distances efficiently. In Proc. of the 2002 workshop on Memory System Performance, pages 37–43, 2003.

[4] W. K. Anderson and D. L. Bonhaus. An implicit upwind algorithm for computing turbulent flows on unstructured grids. Computers and Fluids, 23:1–21, 1994.

[5] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In Proceedings of Supercomputing '99, 1999. Also published as Mathematics and Computer Science Division, Argonne National Laboratory, Technical Report ANL/MCS-P776-0899, August 1999.

[6] Stefan Andersson, Ron Bell, John Hague, Holger Holthoff, Peter Mayes, Jun Nakano, Danny Shieh, and Jim Tuccillo. RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide. IBM International Technical Support Organization, October 1998. http://www.redbooks.ibm.com.

[7] A. Baker, 2005. Personal Communication.

[8] A. Baker, J. Dennis, and E. R. Jessup. Toward memory-efficient linear solvers. In J.M.L.M. Palma, J. Dongarra, V. Hernandez, and A. A. Sousa, editors, VECPAR '2002, Fifth International Conference on High Performance Computing for Computational Science: Selected Papers and Invited Talks, volume 2565 of Lecture Notes in Computer Science, pages 315–327. Springer, Berlin, 2003.

[9] A. H. Baker, J. M. Dennis, and E. R. Jessup. An efficient block variant of GMRES. Technical Report CU-CS-957-03, University of Colorado, Department of Computer Science, July 2003. To appear in SIAM Journal of Scientific Computing.

[10] A. H. Baker, E. R. Jessup, and T. Manteuffel. A technique for accelerating the convergence of restarted GMRES. To appear SIAMM Journal on Matrix Analysis and Applications, 2005. Also available as University of Colorado, Deparment of Computer Science, Technical Report CU-CS-945-03.

[11] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matt Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 2.1.5, Mathematics and Computer Science Division, Argonne National Laboratory, 2003. http://www.mcs.anl.gov/petsc.

[12] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Peiriyacheri, M. Walken, and D. Zaretsky. A Matlab compiler for distributed, heterogeneous, reconfigurable computing systems. In Int. Symp. on FPGA Custom Computing Machines (FCCM-2000), Napa Valley, Ca, April 2000.

[13] S. T. Barnard, A. Pothen, and H. D. Simon. A spectral algorithm for envelope reduction of sparse matrices. Numerical Linear Algebra with Applications, 2(4):317–334, 1995.

[14] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition. SIAM, Philadelphia,PA, 1994.

[15] Basic Linear Algebra Subprograms Technical (BLAST) Forum: Document for the basic linear algebra subprograms (BLAS) standard. http://www.netlib.org/blast/blast-forum, 2001.

[16] S. Behling, R. Bell, P. Farrell, H. Holthoff, F. O'Connell, and W. Weir. The POWER4 Processor Introduction and Tuning Guide. IBM Redbooks, November 2001. http://www.redbooks.ibm.com.

[17] D. A. Burgess and M. B. Giles. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. Technical Report 95/06, Numerical Analysis Group, Oxford University Computing Laboratory, May 1995.

[18] Steve Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing, pages 400–405. SIAM, 1989.

[19] C. Cascaval, L. DeRose, D. Padua, and D. Reed. Compile-time based performance prediction. In Proc. LCPC Workshop, pages 365–379, 1999.

[20] C. Cascaval and D. Padua. Estimating cache misses and locality using stack distances. In Proc. of the 17th annual International Conference on Supercomputing, pages 150–159, 2003.

[21] I. Chihaia and T. Gross. Effectiveness of simple memory models for performance prediction. In Proc. ISPASS, pages 98–105. IEEE, March 2004.

[22] A. T. Chronopoulos and C. W. Gear. S-step iterative methods for symmetric linear systems. Journal of Computational and Applied Mathematics, 25:153–168, 1989.

[23] Intel Corporation. Intel Pentium 4 and Intel Xeon Processor Optimization: Reference Manual. Technical Report 24896604, Intel Corporation, 2001.

[24] Intel Corporation. Intel Xeon Processor with 512 KB L2 cache at 1.80 Ghz to 3 Ghz DataSheet. Technical Report 29864206, Intel Corporation, 2003.

[25] E. H. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In Proceedings 24th Nat. Conf. Assoc. Comp. Mach., pages 157–172. ACM Publications, 1969.

[26] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. AIAA Journal, 32(3):489–496, 1994.

[27] T. Davis. University of Florida sparse matrix collection, http://www.cise.ufl.edu/research/sparse/matrices, 2003.

[28] E. F. D'Azevedo, V. L. Eijkhout, and C. H. Romaine. Conjugate gradient algorithms with reduced synchronization overhead on distributed memory multiprocessors. Technical Report 56, Lapack Working Note, August 2002.

[29] L. DeRose and D. Padua. A Matlab to Fortran 90 translator and its effectiveness. In Proceedings of the 10th ACM International Conference on Supercomputing - ISC'96, pages 309–316, Philadelphia, PA, May 1996.

[30] L. DeRose and D. Padua. Techniques for the translation of Matlab programs into Fortran 90. ACM Transactions on Programming Languages and Systems, 21(2):286–323, March 1999.

[31] J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff. Algorithm 679: A set of level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Software, 16:18–28, 1990.

[32] J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. Algorithm 656: An extended set of Fortran Basic Linear Algebra Subprograms: Model implementation and test programs. ACM Trans. Math. Software, 14:18–32, 1988.

[33] J. Dongarra, J. DuCroz, S. Hammarling, and R. J. Hanson. An extended set of Fortran Basic Linear Algebra Subprograms. ACM Trans. Math. Software, 14:1–17, 1988.

[34] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. LINPACK Users' Guide. SIAM Publications, 1979.

[35] A. A. Dubrulle. Retooling the method of block conjugate gradients. Electronic Trans. on Num. Anal., 12, 2001.

[36] I. S. Duff, R. Grimes, and J. Lewis. Sparse matrix test problems. ACM Trans. Math. Software, 15:1–14, 1989.

[37] D. R. Durran. Numerical Methods for Wave Equations in Geophysical Fluid Dynamics. Springer-Verlag, 1999.

[38] M. Engeli, T. Ginsburg, H. Rutishauser, and E. Stiefel. Refined Iterative Methods for Computation of the Solution and the Eigenvalues of Self-Adjoint Boundary Value Problems. Birkhäuser, Basel/Stuttgart, 1959.

[39] T. Fahringer. Estimating cache performance for sequential and data parallel programs. Technical Report TR 97-9, Institute for Software Technology and Parallel Systems, Univ. of Vienna, Vienna, Austria, October 1997.

[40] C. Fang, S. Carr, S. Önder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In Proc. of the 2002 workshop on Memory System Performance, pages 60–68, 2004.

[41] Peter Fiebach, Roland Freund, and Andreas Frommer. Variants of the block-QMR method and applications in quantum chromodynamics. In Proceedings of the IMACS World Congress, August 1997.

[42] M. Field. Optimizing a parallel conjugate gradient solver. SIAM J. Sci. Stat. Computing, 19:27–37, 1998.

[43] B. B. Fraguela, R. Doallo, J. Tourino, and E. L. Zapata. A compiler tool to predict memory hierarchy performance of scientific codes. Parallel Computing, 30:225–248, 2004.

[44] R. W. Freund. Solution of shifted linear systems by quasi-minimal residual iterations. Numerical Linear ALgebra, pages 101–121, 1993.

[45] Roland W. Freund and Manish Malhotra. The block-QMR method for the solution of multiple radiation and scattering problems in structural acoustics. http://www.bell-labs.com/project/BLQMR/, 2003.

[46] A. Frommer. BiCGStab(l) for families of shifted linear systems. Computing, 70(2):87–109, 2003. Preprint BUGHW-SC 02/04.

[47] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed H. Sameh. Impact of hierarchical memory systems on linear algebra algorithm design. International Journal of Supercomputer Applications, 2(1):12–48, 1988.

[48] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In Proc. ASPLOS, pages 228–239, Oct. 1998.

[49] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. SIAM J. Num. Anal., 13:236–249, 1976.

[50] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. Communications of the ACM, 35(2):121–131, February 1992.

[51] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In A. Ecer et al., editor, Proceedings of Parallel CFD'99, pages 233–240. Elsevier, 1999.

[52] William D. Gropp, Dinesh K. Kaushik, David E. Keyes, and Barry F. Smith. High-performance parallel implicit CFD. Parallel Computing, 27:337–362, 2001.

[53] J. Hennessey and D. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 2nd edition, 1996.

[54] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. J. Research Nat. Bur. Standards, 49:409–436, 1952.

[55] Eun-Jin Im and K. Yelick. Optimizing sparse matrix vector multiplication on SMPs. In Ninth SIAM Conference on Parallel Processing for Scientific Computing, 1999.

[56] Eun-Jin Im and Katherine Yelick. Model-based memory hierarchy optimizations for sparse matrices. In Workshop on Profile and Feedback-Directed Compilation, Paris, France, 1998.

[57] The MathWorks Inc. MCC. http://www.mathworks.com/products/compiler, 2003.

[58] Beat Jegerlehner. Krylov space solvers for shifted linear systems. Technical Report IUHET-353, Indiana University, Department of Physics, December 1996.

[59] P. G. Joisha and P. Banerjee. MAGICA: A software tool for inferring types in Matlab. Technical Report CPDC-TR-2002-10-004, Department of Electrical and Computer Engineering, Northwestern University, October 2002.

[60] P. G. Joisha and P. Banerjee. Static array storage optimization in Matlab. In ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, June 2003.

[61] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In Proceedings of the 1998 IEEE International Conference on Computer Design, pages 90–95, October 1998.

[62] V. Kotlyar, K. Pingali, and P. Stodghill. Compiling parallel sparse code for user-defined data structures. In SIAM Conference on Parallel Processing for Scientific Computing, volume 8, 1997. http://www.cs.cornell.edu/Info/Projects/Bernaulli.

[63] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991.

[64] J. R. Levine, T. Mason, and D. Brown. Lex & Yacc. O'Reilly & Associates, 2nd edition, 1992.

[65] J. W. H. Liu and A. H. Sherman. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. SIAM J. Num. Anal., 13:198–213, 1976.

[66] R. Loft, 2001. Personal Communication.

[67] Richard D. Loft, Stephen J. Thomas, and John M. Dennis. Terascale spectralelement dynamical core for atmospheric general circulation models. In Proceedings of SC2001, 2001.

[68] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. ACM SIGMETRICS Performance Evaluation Review, 32:2–13, June 2004.

[69] MATLAB: The language of technical computing. The MathWorks Inc. http://www.mathworks.com.

[70] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture Newsletter, December 1995. http://www.cs.virginia.edu/stream.

[71] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream, 1995.

[72] S. McKee and W. Wulf. Access order and memory-conscious cache utilization. In First Symposium on High Performance Computer Architecture (HPCA1), January 1995.

[73] J. Michalakes, S. Chen, J. Dudhia, L. Hart, J. Klemp, J. Middlecoff, and W. Skamarock. Development of a next generation regional weather research and forecast model. In Walter Zwieflhofer and Norbert Kreitz, editors, Developments in Teracomputing: Proceedings of the Ninth ECMWF Workshop on the Use of High Performance Computing in Meteorology, pages 269–276. World Scientific, 2001.

[74] Sun Microsystems. The Ultra2 architecture: Technical white paper, 2005. http://pennsun.essc.psu.edu/customerweb/WhitePapers/.

[75] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In Proceedings of Parallel Architectures and Compilation Techniques '99, October 1999.

[76] S. Naffziger and G. Hammond. The implementation of the next generation 64b Itanium microprocessor. In Proceedings of the IEEE International Solid-State Circuits Conference, volume 2, pages 276–504, 2002.

[77] National Institute of Standards and Technology, Mathematical and Computational Sciences Division. Matrix Market. http://math.nist.gov/MatrixMarket, 2002.

[78] Netlib Repository at UTK and ORNL. http://www.netlib.org, 2005.

[79] Dianne P. O'Leary. The block conjugate gradient algorithm and related methods. Linear Algebra and its Applications, 29:293–322, 1980.

[80] Dianne P. O'Leary. Parallel implementation of the block conjugate gradient algorithm. Parallel Computing, 5:127–139, 1987.

[81] PAPI: Performance Application Programming Interface: User's Guide. http://icl.cs.utk.edu/papi, 2005.

[82] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yellick. A case for intelligent RAM. IEEE Micro, pages 34–44, March/April 1997.

[83] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In Proceedings of SC'99, 1999.

[84] T. Pittman and J. Peters. The Art of Compiler Design: Theory and Practice. Prentice Hall, Englewood Cliffs, NJ, 1992.

[85] L. M. Polvani, R. K. Scott, and S. J. Thomas. Numerically converged solutions of the global primitive equations for testing the dynamical core of atmospheric GCMs. Monthly Weather Review, 132:2539–2552, 2004.

[86] M. J. Quinn, A. Malishevsky, N. Seelam, and Y Zhao. Preliminary results from a parallel Matlab compiler. In Proceedings of the International Parallel and Distributed Processing Symposium, Orlando, Florida, 1998.

[87] M. Rančić, R. J. Purser, and F. Messinger. A global shallow-water model using an expanded spherical cube: Gnomic versus conformal coordinates. Q. J. R. Meteorol. Sol., 122:959–982, 1996.

[88] J. K. Reid. On the method of conjugate gradients for the solution of large sparse systems of linear equations., pages 231–254. Academic Press, 1971.

[89] C. Ronchi, R. Iacono, and P. S. Paolucci. The "cubed sphere": A new method for the solution of partial differential equations in spherical geometry. Journal of Computational Physics, 124:93–114, 1996.

[90] Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. Mathematics of Computation, 37(155):105–126, 1981.

[91] Y. Saad. Iterative Methods for Sparse Linear Systems. PWS Publishing Company, 1996.

[92] Y. Saad and M. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing, 7:856–869, 1986.

[93] Robert Sadourny. Conservative finite-difference approximations of the primitive equations on quasi-uniform spherical grids. Monthly Weather Review, 100(2):136–144, 1972.

[94] H. Simon and A. Yeremin. A new approach to construction of efficient iterative schemes for massively parallel applications: variable block CG and BiCG methods and variable block Arnoldi procedure. Parallel Processing for Scientific Computing, pages 57–60, 1993.

[95] Horst D. Simon. The Lanczos algorithm for solving symmetric linear systems. PhD thesis, University of California, Berkeley, April 1982.

[96] A. M. Sloane. An evaluation of an automatically generated compiler. ACM Transactions on Programming Languages and Systems, 17:691–703, September 1995.

[97] P. K. Smolarkiewicz, V. Grubisic̆́, and L. G. Margolin. On forward-in-time differencing for fluids: Stopping criteria for iterative solutions of anelastic pressure equations. Mon. Wea. Rev., 125:647–654, 1997.

[98] P. Sonneveld. CGS, a fast Lanczos-type solverfor nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing, 10(1):36–52, 1989.

[99] A. St-Cyr, M. J. Gander, and S. J. Thomas. Optimized multiplicative, additive and restricted additive Schwarz preconditioning, 2005. Submitted for publication SIAM Journal on Scientific Computing.

[100] A. St-Cyr, M. J. Gander, and S. J. Thomas. Optimized restricted additive Schwarz methods. In Lecture Notes in Computational Science and Engineering. Sprint Verlag, 2005. http://cims.nyu.edu/dd16/proceeddings.html.

[101] Mark Taylor, Joseph Tribbia, and Mohamed Iskandarani. The spectral element method for the shallow water equations on the sphere. Journal of Computational Physics, 130:92–108, 1997.

[102] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In Proceedings of Supercomputing 1992, 1992.

[103] S. J. Thomas, J. M. Dennis, H. M. Tufo, and P. F. Fischer. A Schwarz preconditioner for the cubed-sphere. SIAM Journal of Scientific Computing, 25(2):442–453, 2003.

[104] S. J. Thomas and R. D. Loft. Parallel semi-implicit spectral element methods for atmospheric general circulation models. SIAM Journal of Scientific Computing, 16, June 2001.

[105] S. J. Thomas and R. D. Loft. The NCAR spectral element climate dynamical core: Semi-implicit eulerian formulation. SIAM Journal of Scientific Computing, 2005. To appear.

[106] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. IBM Journal of Research Development, 41(6):711–725, November 1997.

[107] Sivan A. Toledo. Quantitative Performance Modeling of Scientific Computations and Creating Locality in Numerical Algorithms. PhD thesis, Massachusetts Institute of Technology, 1995.

[108] H. M. Tufo and P. F. Fischer. Terascale spectral element algorithms and implementations. In Proceedings of SC'99, 1999.

[109] Jasper van den Eshof and Gerard L. G. Sleijpen. Accurate conjugate gradient methods for shifted systems. Technical Report 1265, Universiteit Utrecht, 2003.

[110] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems. SIAM Journal on Scientific and Statistical Computing, 12:631–644, 1992.

[111] A. J. C. van Gemund. Automatic cost estimation of data parallel programs. Technical Report 1-68340-44, Faculty of Information Technology and Systems, Delft University of Technology, Oct 2001.

[112] A. J. C. van Gemund. Symbolic performance modeling of parallel systems. IEEE Transactions on Parallel and Distributed Systems, Feb. 2003.

[113] Spiros Vellas. Scalar Code Optimization I, 2005.
http://sc.tamu.edu/help/origins/sgi_scalar_r14k_opt.pdf.

[114] Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In Proceedings of the IEEE/ACM SC2002 Conference, 2002.

[115] W. Waite, U. Kastens, and A. M. Sloane. Eli: Translator construction made easy. http://eli-project.sourceforge.net/, 2005.

[116] R. C. Whalely, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. Technical Report 141, Lapack Working Note, September 2000.