

# Applying Automated Memory Analysis to Improve Iterative Algorithms

John M. Dennis  
Elizabeth R. Jessup

University of Colorado at Boulder  
Technical Report CU-CS-1012-06  
Department of Computer Science  
Campus Box 430  
University of Colorado  
Boulder, Colorado 80309

# Applying Automated Memory Analysis to Improve Iterative Algorithms

J. M. Dennis<sup>1</sup> E. R. Jessup<sup>2</sup>

## Abstract

Historically, iterative solvers have been designed so as to minimize the number of floating-point operations. We propose instead that iterative solvers should be designed to minimize the amount of data that must be loaded from the memory hierarchy to the CPU. In this paper, we describe automated memory analysis, a technique to improve the memory efficiency of a sparse linear iterative solver. Our automated memory analysis uses a language processor to predict the data movement required for an iterative algorithm based upon a Matlab implementation. We demonstrate how automated memory analysis is used to reduce the execution time of a component of a global parallel ocean model. In particular, code modifications identified or evaluated through automated memory analysis enables a 46% reduction in execution time for the conjugate gradient solver on a small serial problem. Further, we achieve a 9% reduction in total execution time for the full model on 64 processors. The predictive capabilities of our automated memory analysis can be used to simplify the development of memory efficient numerical algorithms or software.

## 1 Introduction and Motivation

Many important scientific and engineering computations involve the solution of a set of partial differential equations formulated as 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 [?, ?]. Historically, iterative solvers have been designed to achieve the best numerical accuracy for a given number of floating-point operations [?, ?, ?]. This approach is based on the assumption that the time to perform those operations dominates the cost of the 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 [?]. We must therefore take a memory-centric approach to the design and implementation of efficient algorithms. To that end, we propose to use memory analysis as an integral component of the design process. Memory analysis evaluates the interaction of the algorithm with the memory hierarchy by calculating the amount of data that must be loaded from the memory hierarchy for each component of the iterative algorithm. We have developed the Sparse Linear Algebra Memory Model (SLAMM) language processor to automate memory analysis. The SLAMM language processor analyzes an algorithm written in Matlab code. The SLAMM analysis gives a

---

<sup>1</sup>Scientific Computing Division, National Center for Atmospheric Research, Boulder, CO 80307-3000 (dennis@ucar.edu). The work of this author was supported by the National Science Foundation.

<sup>2</sup>Department of Computer Science, University of Colorado, Boulder, CO 80309-0430 (jessup@cs.colorado.edu). The work of this author was supported by the National Science Foundation under grants no. ACI-0072119 and CCF-0430646.

prediction of the minimum required data movement for an efficient implementation of the algorithm in a compiled language.

Automated memory analysis provides both the ability to rapidly evaluate the memory efficiency of a particular design choice during the design phase and the ability to improve the memory efficiency of a pre-existing solver. We illustrate both applications by using SLAMM to reduce the execution time of the Parallel Ocean Program (POP) [?]. In Section ??, we provide the background for memory efficient programming and review related work. In Section ??, we describe POP, a parallel ocean model developed at Los Alamos National Laboratory. We describe the two-dimensional (2D) data structure it currently uses and an alternative one-dimensional (1D) data structure in Section ?. In Section ??, we describe memory analysis, a technique to calculate the amount of data that must be moved through the memory hierarchy for each component of an iterative algorithm. We describe manual memory analysis in Section ?? and automated memory analysis using SLAMM in Section ?. Finally, in Section ??, we demonstrate how SLAMM is used to increase the memory efficiency of the POP iterative solver and reduce execution time.

## 2 Background and Related Work

The creation of software that is both numerically and memory efficient [?, ?, ?, ?] is a non-trivial task that has not been done extensively or in a systematic fashion. We propose a technique to simplify the design and implementation of memory efficient software through automated memory analysis. Our automated memory analysis uses compiler techniques to evaluate an algorithm implemented in Matlab. We chose Matlab because it is a ubiquitous programming language in numerical analysis research. Matlab is popular because it allows for the rapid prototyping of numerical algorithms without the complexities of a compiled language like Fortran or C. The SLAMM language processor inputs Matlab code marked with SLAMM directives and outputs the same code with additional code blocks to perform memory analysis. Our work differs from the compilers described in [?, ?, ?, ?] because we do not convert Matlab to another language but rather analyze the code to determine its potential memory efficiency before implementation in a compiled language.

We focus our efforts on improving the execution rate of linear algebra algorithms that are typically memory bandwidth-limited. Memory bandwidth is the rate that data is moved through the memory hierarchy in units of cache-lines. We quantify the impact of memory bandwidth on a particular algorithm by comparing the total storage requirement of an algorithm as well as 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 accessed during a particular section or phase of the algorithm. Different implementations of the same algorithm may have the same total storage requirement but may have significantly different working set sizes. The results of matching an algorithm's working set size to cache size can result in significant performance gains. For example, loop blocking [?, ?] of dense linear algebra, which breaks up larger code blocks or loops into smaller blocks in order to match working set size and cache size, is used in the

optimized level 2 and 3 BLAS routines [?].

Several researchers have used compiler techniques to predict serial and parallel performance. Cascaval et al. [?] use the Polaris compiler to predict the performance of serial Fortran and C codes. Polaris-based predictions for execution time and cache miss-rates, which are fed back into the compiler to improve code optimization, are accurate to within an average of 20% error. C. van Gemund [?] use the PAMELA compiler to add symbolic cost expressions for data parallel problems, which, in combination with a simple machine model, allow it to predict execution time to within 10% error. Fahringer [?] developed the parameter-based performance prediction Tool ( $P^3T$ ) which is integrated with Vienna Fortran compiler [?].

Unlike [?, ?, ?], 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 in a compiled language. Analyzing an algorithm written in an interpreted high level language like Matlab provides an analysis that is not biased by the implementation details present in a Fortran or C version. This allows SLAMM to accurately predict the minimum possible data movement needed by an algorithm and compares it to the existing data movement. We do not develop a sophisticated analytical model of memory performance because the consumer of SLAMM’s analysis is the developer of the algorithm. Specifically, we find that a simple model that predicts the amount of data loaded from L2 to L1 cache ( $Mbytes_{L1}$ ) is sufficient to provide valuable insight to the algorithm developer.

The performance of POP has been analyzed by several groups. Kerbyson [?] developed an analytical model of expected parallel performance. Snavely [?] employs a convolution-based framework that uses hardware performance counts and MPI library traces to predict parallel performance. Our work does not specifically address parallel performance but rather concentrates on reducing data movement for a single component of the POP model.

### 3 Parallel Ocean Program

The Parallel Ocean Program (POP) [?] is a global ocean model developed at Los Alamos National Laboratory that is used extensively as the ocean component of the Community Climate System Model [?]. POP uses finite-difference for the baroclinic component of the timestep and a preconditioned conjugate gradient solver to solve for surface pressure in the barotropic component. Parallelism on distributed memory computers is supported through the Message Passing Interface (MPI) [?] standard.

We use POP version 2.0.1 [?] to examine data movement in the barotropic solver using the *test* and *gx1v3* grids. The test grid is a coarse grid provided with the source code to facilitate the porting of POP to other compute platforms. The *gx1v3* grid is higher resolution than the test grid with one degree resolution at the equator. POP using the *gx1v3* grid represents 20% of the total compute cycles at the National Center for Atmospheric Research (NCAR) [?].

### 3.1 Existing data structure

POP uses a three-dimensional computational mesh. The horizontal dimensions are decomposed into logically rectangular two-dimensional (2D) blocks [?]. The computational mesh is distributed across multiple processors by placing one or more 2D blocks on each processor. There are  $km$  vertical levels associated with each 2D horizontal block. Blocks that do not contain any ocean points, or land blocks, are eliminated. Table ?? contains the dimensions of the test and gx1v3 grids. The variables  $nx$  and  $ny$  are the global numbers of points in the x and y directions of the grid respectively. Table ?? also contains the block sizes  $bsize\_x$  and  $bsize\_y$ , the number of blocks  $nblocks$ , and the numbers of ocean, land, and total points in the barotropic solver for several block size configurations. The block size parameters  $bsize\_x$  and  $bsize\_y$  correspond to the number of points in the x and y directions for each 2D block. Note that land block elimination reduces the number of  $20 \times 24$  blocks to 246 resulting in a reduction in the number of land points versus the  $40 \times 48$  configuration.

name	grid $nx \times ny \times km$	$bsize\_x \times bsize\_y$	nblocks	# points		
				ocean	land	total
test	192x128x20	$16 \times 16$	92	15378	8174	23552
gx1v3	320x384x40	$40 \times 48$	64	86354	36526	122880
		$20 \times 24$	246	86354	31726	118080

Table 1: Description of POP grids.

Figure ?? illustrates the gx1v3 grid decomposed into 2D blocks of size  $20 \times 24$ . The white area in Figure ?? represents land, while ocean is indicated by gray. The black lines in Figure ?? indicate the boundaries of the  $20 \times 24$  blocks. We next describe the form of the 2D data structure in greater detail.

Each 2D block contains  $bsize\_x \times bsize\_y$  internal grid points and a halo region of width  $nghost$ . The halo region is used to store grid points that are located in neighboring 2D blocks. Figure ?? illustrates the 2D data structure within the source code for the block from Figure ?? containing the Iberian peninsula. Note that the large white structure on the upper right side of the block in Figure ?? is the Iberian peninsula while the white structure in the lower right-side is the northern coast of Africa. For this particular block, only about 60% of the grid points represent ocean.

The matrix-vector multiply within the conjugate gradient solver is applied via a 9-point stencil. The code for the 9-point stencil is illustrated in Figure ?. To calculate point  $Y(i, j)$ , the nine points  $X(i, j)$ ,  $X(i, j+1)$ ,  $X(i, j-1)$ ,  $X(i+1, j)$ ,  $X(i-1, j)$ ,  $X(i+1, j+1)$ ,  $X(i+1, j-1)$ ,  $X(i-1, j+1)$ , and  $X(i-1, j-1)$  are required along with the associated coefficient arrays  $A0$ ,  $AN$ ,  $AE$ , and  $ANE$ . The need for a halo region is apparent if we consider the calculation of the corner point  $Y(1, 1)$ . We need among other points  $X(0, 0)$  which is contained within the halo region. The primary advantage of the 2D data structure is that it provides a regular stride-one access for the matrix-vector multiply. The disadvantage of the 2D data structure is that it includes a large number of grid points that represent land. In effect, a number of explicitly stored zeros are added to the matrix. While it is possible to reduce the number of

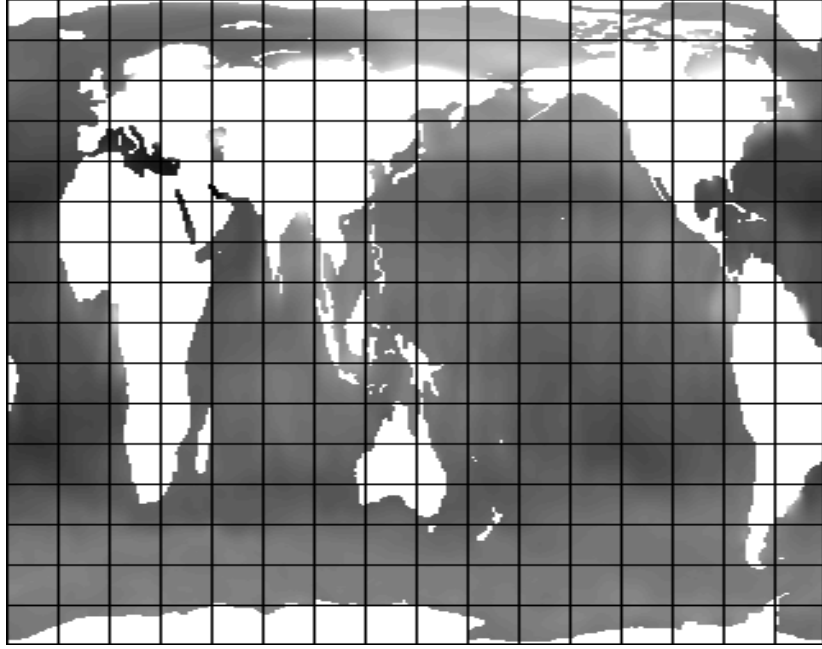


Figure 1: The POP gx1v3 grid, where white corresponds to land points, gray to ocean points, and superimposed lines indicate  $20 \times 24$  blocks.

excessive land points by reducing the block size, smaller blocks have a larger percentage of their total size dedicated to halo points. We present an alternative to the 2D data structure in the next section.

### 3.2 An alternate data structure

We next describe a one-dimensional data structure that enables the elimination of a potentially large number of land points present in the 2D data structure. The 1D data structure consists of a 1D array of extent  $n$ . The first  $n_{Active}$  elements in the 1D array correspond to active ocean points, while the remaining  $n - n_{Active}$  points correspond to the off-processor halo needed by the 9-point stencil. In removing the excessive land points, the 1D data structure changes the form of the matrix-vector multiply. Indirect-addressing is now necessary to apply the operator matrix which is stored in a compressed sparse row format. The code for the matrix-vector multiply for the 1D data structure is provide in Figure ?? . The need for the  $Mat\%Ia$  and  $Mat\%Ja$  index arrays has the potential to negate any reduction in data movement achieved by the elimination of land points. We demonstrate in the next section

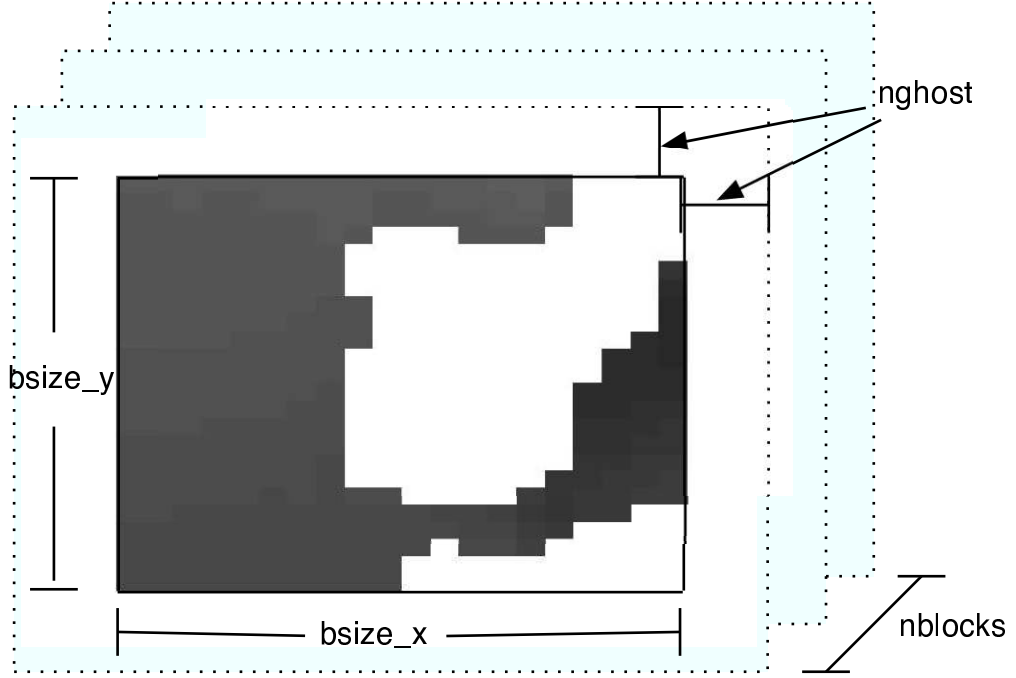


Figure 2: A  $20 \times 24$  block of the gx1v3 grid over Iberian peninsula. Note the presence of land and ocean points within the block.

how memory analysis is used to evaluate the impact of alternative data structures on data movement.

## 4 Memory Analysis

We next examine the process of memory analysis. Memory analysis examines the amount of data that must be loaded from the memory hierarchy to the CPU to perform an operation. Consider two fundamental linear algebra operations, a dot product and an AXPY. The dot product of two vectors  $\alpha = u'v$  where  $v, u \in \mathbb{R}^n$  and  $\alpha \in \mathbb{R}$  requires two vectors of length  $n$ . The amount of data that must be loaded from the memory hierarchy, or working set load size (WSL), is

$$WSL^{DOT} = 2nL_d, \quad (1)$$

where  $L_d$  is the size of a double precision floating-point value in bytes. The AXPY operation  $u = u + \alpha v$ , where  $v, u \in \mathbb{R}^n$  and  $\alpha \in \mathbb{R}$ , requires two vectors of length  $n$  and a scalar. The working set size for an AXPY operation is

$$WSL^{AXPY} = (2n + 1)L_d, \quad (2)$$

```

do j=this_block%jb,this_block%je
do i=this_block%ib,this_block%ie
  Y(i,j) = AO (i ,j )*X(i ,j ) + &
            AN (i ,j )*X(i ,j+1) + &
            AN (i ,j-1)*X(i ,j-1) + &
            AE (i ,j )*X(i+1,j ) + &
            AE (i-1,j )*X(i-1,j ) + &
            ANE(i ,j )*X(i+1,j+1) + &
            ANE(i ,j-1)*X(i+1,j-1) + &
            ANE(i-1,j )*X(i-1,j+1) + &
            ANE(i-1,j-1)*X(i-1,j-1)
end do
end do

```

Figure 3: The barotropic operator in POP, implemented as a 9-point stencil using the two-dimensional data structure.

```

is = Mat%Ia(1)
do i=1,n2
  ie = Mat%Ia(i+1)
  tmp = 0.0
  do j=is,ie-1
    tmp = tmp + Mat%A(j)*X(Mat%Ja(j))
  enddo
  Y(i) = tmp
  is = ie
enddo

```

Figure 4: The barotropic operator in POP, implemented as compressed sparse row matrix-vector multiply using the one-dimensional data structure.



```

1.  $r_0 = b - Ax_0, p_0 = 0, \phi_0 = (r_0, r_0)$ 
2. for  $j = 0, 1, \dots$  until convergence
3.      $Mz = r_j$                                 /* Apply Precon */
4.      $\phi_{j+1} = (r_j, z)$                         /* dot product */
5.      $\beta = \phi_{j+1}/\phi_j$ 
6.      $p_{j+1} = z + \beta p_j$                         /* AXPY */
7.      $q = Ap_{j+1}$                                 /* MxV */
8.      $\delta = (p_{j+1}, q)$                         /* dot product */
9.      $\alpha = \phi_{j+1}/\delta$ 
10.     $x_{j+1} = x_j + \alpha p_{j+1}$                   /* APXY */
11.     $r_{j+1} = r_j - \alpha p_{j+1}$                   /* AXPY */
12. end

```

Figure 5: Preconditioned Conjugate Gradient (PCG) algorithm.

The derivation of equations (??) and (??) is an example of manual memory analysis. While manual memory analysis is trivial for simple linear algebra operations, its complexity grows if we extend the technique to an entire iterative algorithm. We describe manual memory analysis of a complete iterative algorithm in Section ??, followed by our automated memory analysis procedure in Section ??.

#### 4.1 Manual Memory Analysis

We focus on the preconditioned conjugate gradient (PCG) iterative solver used by POP [?] to update the ocean surface pressure at each timestep. In Figure ??, we provide a common form of the PCG solver. Iterative solvers are composed of basic linear algebra operations. Thus, we compute the working set load size of the solver from the working set load size of its constituent operations. In Table ??, we provide the working set sizes for each line of the PCG algorithm in Figure ?. Note that the PCG algorithm requires several dot products (DOT) and axpy operations (AXPY), a single sparse matrix-vector product (MxV), and the application of the preconditioner (Precon) for each iteration.

We first determine the working set load size for the sparse matrix-vector multiply for the 2D and 1D data structures, followed by the application of the preconditioner. First, consider the matrix-vector multiply for the 2D data structure. The working set load size (WSL) of a matrix-vector multiply  $y = Ax$ , where  $A$  is a sparse coefficient matrix, is

$$WSL = \text{sizeof}(y) + \text{sizeof}(x) + \text{sizeof}(A), \quad (3)$$

where  $\text{sizeof}()$  is a function that returns the size of its argument in bytes. An examination of Figure ??, which contains the code for the matrix operator for the 2D data structure, indicates that arrays  $A0$ ,  $AN$ ,  $AE$ , and  $ANE$  form the coefficient matrix  $A$ . Because  $\text{sizeof}(A) = 4 \cdot \text{sizeof}(x)$ , we use (??) to calculate working set load size for the 2D matrix-vector product

$$WSL_{2D}^{MxV} \equiv (2n_{2D} + 4n_{2D})L_d = 6n_{2D}L_d, \quad (4)$$

Line	Operation	WSL (bytes)
3	Precon	varies
4	$DOT()$	$2nL_d$
6	$AXPY()$	$(2n+1)L_d$
7	$MxV()$	varies
8	$DOT()$	$2nL_d$
10	$AXPY()$	$(2n+1)L_d$
11	$AXPY()$	$(2n+1)L_d$
2-12	TOTAL	$(10n+3)L_d$ $+WSL^{MxV}$
	for 1 iteration	$+WSL^{Precon}$

Table 2: Working set sizes (WSL) for each line of the PCG algorithm in Figure ??

where

$$n_{2D} = nblocks \cdot (bsize\_x + 2 \cdot nghost) (bsize\_y + 2 \cdot nghost).$$

The WSL for the matrix-vector multiply for the 1D data structure is different than for the 2D data structure because  $sizeof(A)$  is different. In the case of the 1D data structure, we must account for both the number of non-zero entries and the size of the indirect addressing arrays. The  $WSL_{1D}^{MxV}$  is therefore

$$WSL_{1D}^{MxV} \equiv (nnz_{1D} + 2n_{1D})L_d + (nnz_{1D} + n_{1D})L_i \quad (5)$$

where  $n_{1D}$  is the number of ocean points from column 5 of Table ??,  $nnz_{1D}$  is the number of nonzero values in the matrix  $Mat\%A$  in Figure ??, and  $L_i$  is the size of an integer in bytes.

We next calculate the working set load size for the application of the preconditioner. Because POP uses a scaled diagonal preconditioner,

$$WSL^{Precon} = 2nL_d. \quad (6)$$

Using equations (??), (??), and (??) and the equations provided in Table ?? we can calculate the working set load sizes for both the 2D and 1D based solvers. Deriving and evaluating the analytical expression for data movement in this way is a time consuming, laborious, and error prone process that can be automated using language processing techniques. In the next section we describe the automated memory analysis procedure which greatly simplifies memory analysis.

## 4.2 Automated Memory Analysis

We next describe how the sparse linear algebra memory model (SLAMM) language processor is used to automate the memory analysis procedure. We use the dot product described in Section ?? as an example. Figure ?? contains a Matlab code that calculates the dot product of two vectors  $u, v \in \mathbb{R}^n$ . Note that lines 2 and 4 in Figure ?? are the Matlab code to initialize

```

%SLM start caseStudy;           % line 1
n=200; u=rand(n,1); v=rand(n,1); % line 2
%SLM start dotprod;            % line 3
alpha=u'*v                     % line 4
%SLM end dotprod;              % line 5
%SLM print dotprod;            % line 6
%SLM end caseStudy;            % line 7

```

Figure 6: Matlab code that computes the dot product of two vectors with SLAMM directives.

```

SLAMM Memory Analysis for Body: dotprod
TOTAL: Storage Requirement  Kbytes (SR) : 3.13
TOTAL: Loaded from L2 -> L1 Kbytes (WSL): 3.12 +- 0.00
      DGEMM      Kbytes      : 0.00 +- 0.00
      Sparse Ops  Kbytes      : 0.00 +- 0.00

```

Figure 7: The SLAMM generated predictions for dot product code in Figure ??.

and calculate the dot product, respectively. The remaining lines in Figure ?? with the prefix *%SLM* are directives that control the SLAMM language processor. Lines 1 and 7 delineate the beginning and end of a body of Matlab code. Lines 3 and 5 also delineate a body of code and identify it with the symbolic name *dotprod*. Line 6 is a SLAMM directive requesting the printing of the memory analysis of the *dotprod* code block.

Based on the input Matlab code in Figure ??, the SLAMM language processor analyzes and generates modified output code which contains both the original code and new blocks of code that calculate memory usage properties. Figure ?? shows the resulting automated memory analysis of code body *dotprod* from Figure ?. Note that the total storage requirement (SR) and the working set load size (WSL) are printed in Kbytes. SLAMM separately indicates the component of data movement due to dense matrix-matrix multiplication, indicated by DGEMM, as well as sparse matrix operations. We differentiate between the different components of WSL to indicate the fraction of the algorithm that might benefit from specific optimization techniques like the use of an optimized DGEMM subroutine. Fundamentally, the SLAMM language processor derives and evaluates equation (??) automatically. While automated memory analysis is certainly not necessary for a simple operation like a dot product, its advantages become apparent when it is applied to a complete iterative algorithm that may contain a large number of linear algebra operations. We demonstrate the application of automated memory analysis to evaluate the memory efficiency of the POP preconditioned conjugate gradient solver in the next section.

## 5 Results

Automated memory analysis can be used to evaluate a design decision before implementation in a compiled language and to evaluate the quality of a particular implementation. We first demonstrate how automated memory analysis is used to evaluate the choice of data structure for POP. We implement the PCG algorithm of Figure ?? in Matlab using both the 2D and 1D data structures described in Sections ?? and ??, respectively. Using the SLAMM directives described in the previous section, we delineate one iteration of the PCG algorithms. The predicted data movement ( $WSL_P$ ) for the 2D data structure (PCG2+2D) and the 1D data structure version (PCG2+1D) for the test grid are provided in the second column of Table ???. SLAMM predicts that the use of the 1D data structure would reduce the amount of data loaded from the L2 to the L1 cache ( $Mbytes_{L1}$ ) by 34% versus the existing 2D data structure.

Solver Implementation	$WSL_P$	version	Ultra II		POWER4		R14K	
			$WSL_M$	error	$WSL_M$	error	$WSL_M$	error
PCG2+2D	4902	v1	5163	5.3%	5068	3.4%	5728	16.9%
		v2	4905	0.1%	4865	-0.7%	4854	-1.0%
PCG2+1D	3218		3164	-1.7%	3335	3.7%	3473	7.9%
Reduction in $Mbytes_{L1}$	34%		39%		34%		39%	

Table 3:  $Mbytes_{L1}$  for a single iteration of preconditioned conjugate gradient solver in POP using the test grid. The values of  $WSL_P$  and  $WSL_M$  are in Kbytes.

To compare SLAMM-predicted to measured data movement, we instrumented each version of the solvers with a locally developed performance profiling library (Htrace), which is based on the PAPI [?] hardware performance counter API. 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 [?] (Ultra II), IBM POWER 4 [?] (POWER4), and MIPS R14K [?] (R14K). A description of the cache configurations for each compute platform is provided in Table ??.

CPU	Ultra II	POWER4	R14K
Company	SUN	IBM	SGI
Mhz	400	1300	500
L1 Data-cache	32KB	32KB	32KB
L2 cache	4 MB	1440 KB	8 MB
L3 cache	–	32 MB	–

Table 4: Description of the microprocessor compute platforms and their cache configurations.

The measured data movement ( $WSL_M$ ) or  $Mbytes_{L1}$  the amount of data loaded from the L2 to the L1 cache for the existing 2D data structure implementation (PCG2+2D v1), an optimized 2D version (PCG2+2D v2), and the 1D version are provided for each of the compute platforms in Table ???. We measure 10 iterations and report the average measured

```

! =====
! code block: solver v1
! =====
do iblock=1,nblocks
    P(:, :, iblock) = Z(:, :, iblock) + P(:, :, iblock)*beta
    Q = operator(P, iblock)
    WORK0(:, :, iblock) = Q(:, :, iblock)*P(:, :, iblock)
enddo
delta=global_sum(WORK0, LMASK)

! =====
! code block: solver v2
! =====
delta_local=0.d0
do iblock=1,nblocks
    P(:, :, iblock) = Z(:, :, iblock) + P(:, :, iblock)*beta
    Q = operator(P, iblock)
    WORK0 = Q(:, :, iblock)*P(:, :, iblock)
    delta_local = delta_local + local_sum(WORK0, LMASK(:, :, iblock))
enddo
delta=gsum(delta_local)

```

Figure 8: A code block that implements lines 6 to 8 of the PCG algorithm in Figure ?? for the v1 and v2 solvers.

data movement. While the discrepancies between the measured and predicted WSL for the PCS2+2D v1 solver are minimal for both the Ultra II and POWER4 platforms, the measured value of 5728 Kbytes for the R14K is 17% greater than the predicted value of 4902 Kbytes. The difference in data movement between the three compute platforms may be due to additional code transformations performed by the Ultra II and POWER4 compilers. That the PCG2+2D v1 solver is loading 17% more data from the memory hierarchy than necessary on the R14K is an indication that it is possible to improve the quality of the implementation.

An examination of the source code for the PCG2+2D v1 solver indicates that a minor change to the dot product calculation reduces data movement. Code blocks that correspond to lines 6 to 8 of the PCG algorithm in Figure ?? for the v1 and v2 versions of the solver are provided in Figure ?. The function *operator* applies the 9-point stencil from Figure ?, and the array *LMASK* is an array that masks out points that correspond to land points. In version v1 of the do loop, a temporary array *WORK0* is created that contains the point-wise product of two vectors *Q* and *P*. Outside the do loop, the product of *LMASK* and the *WORK0* array is calculated by *global\_sum* to complete the dot product. If the size of data accessed in the do loop is larger than the L1 cache, then a piece of the *WORK0* array at the end of the do loop is no longer located in the L1 cache and must be reloaded to complete the calculation.

In version v2 of the do loop, in Figure ?, a scalar temporary *delta\_local* is added to accumulate each block's contribution to the dot product of *Q* and *P*. An additional function

`local_sum` is used that applies the land mask to complete the dot product. Finally, we replace the function `global_sum` with a call to `gsum`. The subroutine `gsum`, when executed on a single processor, is an assignment of `delta_local` to `delta`. Because version v2 of the code block does not access `WORK0` outside the do loop, it potentially reduces data movement.

Both dot product calculations (lines 4 and 8 of Figure ??) in the PCG algorithm were rewritten to create the v2 solver. The  $WSL_M$  values in Table ?? for the v1 and v2 versions of the PCG2+2D solver on the R14K indicate that the rearranged dot product calculations reduce data movement by 18%. Note, that data movement is also reduced on the Ultra II and POWER4 platforms but to a lesser extent.

The previous description is a demonstration of how SLAMM is used to guide performance tuning. Table ?? also provides the relative error between predicted and measured data movement for the solvers. SLAMM predicts data movement to within an average error of 0.6% for the PCG2+2D v2 solver and within 4.4% for the PCG2+1D solver. We provide the actual percentage reductions in data movement for the 1D versus 2D solver for each compute platform. Table ?? indicates that the actual percentage reductions in data movement are very similar to the predicted reductions. Table ?? clearly demonstrates that it is possible for automated memory analysis to accurately predict the amount of data movement required for an algorithm. Such a capability provides *a priori* knowledge of the memory efficiency of a particular design choice before implementation in a compiled language.

We next examine the impact of reducing data movement on execution time. The timestep of POP includes a baroclinic and a barotropic component. The barotropic component is composed of a single linear solver for surface pressure. We execute POP using the test grid on a single processor of each compute platform for a total of 20 timesteps. This configuration requires an average of 69 iterations of the PCG algorithm per timestep. In Table ??, we provide the barotropic execution time in seconds using the three implementations of the solver. We include the execution time for the initial implementation of the solver using 2D data structures to accurately reflect the overall impact automated memory analysis has on execution time. Note that the PCG1+1D solver consistently has a lower execution time than either of the 2D data structure based solvers. The last row in Table ?? contains the percentage reduction in barotropic execution time for the PCG2+2D v1 versus the PCG2+1D solver. Table ?? indicates that code modifications either evaluated or identified by automated memory analysis reduce execution time by an average of 46%. Curiously, a comparison of Tables ?? and ?? indicates that the percentage reduction in execution time is even larger than the percentage reduction in data movement. This discrepancy may be due to improved compiler optimization for the greatly simplified PCG2+1D solver.

We next examine the impact of the 1D data structures on parallel execution time using the gx1v3 grid. We use the generalized gather-scatter routines of Tufo-Fischer [?] to provide parallel execution under MPI for the 1D version of the solver. In addition to the two dot product PCG gradient algorithm (PCG2) in Figure ??, POP also provides a single dot product PCG algorithm (PCG1) [?]. The PCG1 algorithm provides a performance advantage for parallel execution because it eliminates one of the distributed dot production calculations. Because we want to realistically estimate the impact the 1D data structure has on execution time, we provide timing results for both PCG algorithms using the 2D and 1D data structures

Solver implementation	Ultra II	POWER4	R14K
PCG2+2D v1	21.17	4.57	8.58
PCG2+2D v2	20.49	4.01	7.97
PCG2+1D	12.74	2.11	4.61
Reduction	39%	54%	46%

Table 5: Barotropic execution time for 20 timesteps of POP in seconds using the test grid on a single processor.

in Table ??.

We execute POP on 64 IBM POWER4 processors for a total of 200 timesteps, with an average 151 iterations per timestep. The total execution time in seconds for the four solver implementations is provided in Table ?. These results indicate that use of the PCG1+1D solver versus the PCG1+2D solver reduces total POP execution time by 9%. A 9% reduction in total execution time of POP is significant because it has been extensively studied and optimized [?, ?]. Further, POP consumes approximately 2.4 million CPU hours every year at NCAR. A 9% reduction in execution time eliminates the need for 216,000 CPU hours per year.

	Solver Implementation			
	PCG2+2D v1	PCG1+2D v1	PCG2+1D	PCG1+1D
total time (sec)	86.2	81.5	78.8	73.9

Table 6: Total execution time for 200 timesteps with gx1v3 grid on 64 POWER4 processors.

## 6 Conclusion

We demonstrate how automated memory analysis is used to performance tune the Parallel Ocean Program. We automate memory analysis using the SLAMM language processor which accurately predicts the amount of data loaded from the L2 to L1 cache to within a relative error of 10%. The accurate predictive capability of SLAMM enables the identification of excessive data movement in the existing 2D data structure based solver, and it allows us to evaluate the use of an alternative 1D data structure. We observe that the SLAMM predicted reduction in data movement for the 1D versus the 2D solver is 34% and compares favorably with a measured 36% reduction. The reduction in data movement results in an average 46% reduction in single processor execution time. Finally we demonstrate how use of the 1D data structure reduces total execution time of POP on 64 processors using a 1 degree resolution grid by 9% which represents a substantial savings in compute time.

## References

- [1] A. H. Baker, J. M. Dennis, and E. R. Jessup. On improving linear solver performance: A block variant of GMRES. *SIAM J. of Sci. Comput.*, 27(5):1608–1628, 2006.
- [2] 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)*, page 39, Napa Valley, Ca, April 2000.
- [3] 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.
- [4] S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming*, 7(1):67–81, 1999.
- [5] T. Bettge, 2006. Personal Communication.
- [6] 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.
- [7] C. Cascaval, L. DeRose, D. Padua, and D. Reed. Compile-time based performance prediction. In *Proc. LCPC Workshop*, pages 365–379, 1999.
- [8] 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.
- [9] W. D. Collins, C. M. Bitz, M. L. Blackmon, G. B. Bonan, C. S. Bretherton, J. A. Carton, P. Chang, S. C. Doney, J. J. Hack, T. B. Henderson, J. T. Kiehl, W. G. Large, D. S. McKenna, B. D. Santer, and R. D. Smith. The community climate system model: CCSM3. *Journal of Climate: CCSM Special Issue*, 11(6), 2005.
- [10] E. F. D’Azevedo, V. L. Eijkhout, and C. H. Romine. Conjugate gradient algorithms with reduced synchronization overhead on distributed memory multiprocessors. Technical Report 56, LAPACK Working Note, August 1993.
- [11] 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.
- [12] 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.
- [13] 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.



- [14] M. Field. Optimizing a parallel conjugate gradient solver. *SIAM J. Sci. Stat. Computing*, 19:27–37, 1998.
- [15] P. G. Joisha and P. Banerjee. Static array storage optimization in Matlab. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, San Diego, June 2003.
- [16] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. III White, and J. Levesque. Practical performance portability in the Parallel Ocean Program (POP). *Concurrency Comput. Prac. Exper*, 17:1317–1327, 2005.
- [17] D. J. Kerbyson and P. W. Jones. A performance model of the parallel ocean program. *International Journal of High Performance Computing Applications*, 19(3):261–276, 2005.
- [18] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):63–74, 1991.
- [19] Sun Microsystems. The Ultra2 architecture: Technical white paper, 2005. <http://pennsun.essc.psu.edu/customerweb/WhitePapers/>.
- [20] Dianne P. O’Leary. The block conjugate gradient algorithm and related methods. *Linear Algebra and its Applications*, 29:293–322, 1980.
- [21] PAPI: Performance Application Programming Interface: User’s Guide. <http://icl.cs.utk.edu/papi>, 2005.
- [22] 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.
- [23] The Parallel Ocean Program (POP). <http://climate.lanl.gov/Models/POP>, 2006.
- [24] 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*, pages 81–87, 1998.
- [25] Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. *Mathematics of Computation*, 37(155):105–126, 1981.
- [26] H. D. Simon and A. Yeregin. A new approach to construction of efficient iterative schemes for massively parallel applications: variable block CG and BiCG methods and variable block Arnoldi procedure. In *Parallel Processing for Scientific Computing*, pages 57–60, 1993.
- [27] Horst D. Simon. *The Lanczos algorithm for solving symmetric linear systems*. PhD thesis, University of California, Berkeley, April 1982.
- [28] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Parallel ocean general circulation modeling. *Physica D*, 60:38–61, 1992.

- [29] A. Snively, X. Gao, C. Lee, L. Carrington, N. Wolter, J. Labarta, J. Gimenez, and P. Jones. Performance modeling of HPC applications. In *Parallel Computing (ParCo2003)*, 2003.
- [30] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference: Volume 1, The MPI Core*. The MIT Press, 2000.
- [31] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing 1992*, pages 578–587, 1992.
- [32] H. M. Tufo. *Algorithms for Large-Scale Parallel Simulation of Unsteady Incompressible Flows in Three-Dimensional Complex Geometries*. PhD thesis, Brown University, May 1998.
- [33] A. J. C. van Gemund. Symbolic performance modeling of parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):154–165, Feb. 2003.
- [34] Spiros Vellas. Scalar Code Optimization I, 2005.  
[http://sc.tamu.edu/help/origins/sgi\\_scalar\\_r14k\\_opt.pdf](http://sc.tamu.edu/help/origins/sgi_scalar_r14k_opt.pdf).