SANDIA REPORT

SAND2009-6197
Unlimited Release
Printed September 2009

Highly Scalable Linear Solvers on Thousands of Processors

Jonathan J. Hu and Christopher M. Siefert and Ian Karlin and Raymond S. Tuminaro and Stefan P. Domino and Allen C. Robinson

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2009-6197
Unlimited Release
Printed September 2009

Highly Scalable Linear Solvers on Thousands of Processors

Jonathan J. Hu
Scalable Algorithms Dept.
Sandia National Laboratories
P.O. Box 969, MS 9159
Livermore, CA 94551-0969
jhu@sandia.gov

Christopher M. Siefert
Computational Shock & Multiphysics Dept.
Sandia National Laboratories
P.O. Box 5800, MS 0378
Albuquerque, NM 87185
csiefer@sandia.gov

Ian Karlin
Department of Computer Science
University of Colorado at Boulder
Boulder, CO 80309-0430
ian.karlin@colorado.edu

Raymond S. Tuminaro
Scalable Algorithms Dept.
Sandia National Laboratories
P.O. Box 969, MS 9159
Livermore, CA 94551-0969
rstumin@sandia.gov

Stefan P. Domino
Computational Thermal & Fluid Mechanics Dept.
Sandia National Laboratories
P.O. Box 5800, MS 0836
Albuquerque, NM 87185
spdomin@sandia.gov

Allen C. Robinson
Computational Shock & Multiphysics Dept.
Sandia National Laboratories
P.O. Box 5800, MS 0378
Albuquerque, NM 87185
acrobin@sandia.gov

Abstract

In this report we summarize research into new parallel algebraic multigrid (AMG) methods. We first provide a introduction to parallel AMG. We then discuss our research in parallel AMG algorithms for very large scale platforms. We detail significant improvements in the AMG setup phase to a matrix-matrix multiplication kernel. We present a smoothed aggregation AMG algorithm with fewer communication synchronization points, and discuss its links to domain decomposition methods. Finally, we discuss a multigrid smoothing technique that utilizes two message passing layers for use on multicore processors.

Contents

List of Figures

List of Tables

Executive Summary

This is the final report for the Lab Directed Research and Development three-year project, ”Highly Scalable Linear Solvers on Thousands of Processors”. This project’s objectives were several, and all centered around algebraic multigrid (AMG) algorithms for massively parallel simulations. First, we focused on improving existing parallel algebraic multigrid methods for current and coming computer architectures. Second, we investigated novel algorithms with very different properties than standard methods.

Chapter 1 gives a background on algebraic multigrid. It also covers potential AMG shortcomings in the context of very large-scale computations. Finally, it provides a survey of related research efforts.

Chapter 2 concentrates on key improvements to computation-intensive kernels in the AMG setup phase. The dominant cost in setup is calculation of coarse grid matrix approximations to the application-supplied linear system. This is done by an explicit triple-matrix product. Our contribution was refactoring the matrix-matrix multiply in the TRILINOS package ML in order to exploit matrix block structure. This improvement demonstrated significant speedups for block-structure systems and evidence that the benefit would be much greater for systems with much larger blocks.

Chapter 3 summarizes our exploration of a AMG method with very different communication patterns from standard multigrid. This method requires communication only on the coarsest multigrid mesh. It is not unusual for latency penalties due to message passing on coarse levels to be as costly as the finest (application) level. Thus, any method that can avoid these costs has potential advantages. We begin with a geometric algorithm proposed in the literature and demonstrate how it can be adapted to AMG methods such as those in TRILINOS. We provide 3D numerical experiments that demonstrate its effectiveness. We discuss limitations to the method that we discovered, as well as its connections to two-level domain decomposition methods.

Chapter 4 discusses a new multigrid smoother for multicore architectures. This smoother utilizes two layers of MPI communicators for managing local and global communication. It is fully incorporated into the TRILINOS package IFPACK and has been tested in an outer AMG solver on up to $10,000$ cores of a Cray XT4. We have demonstrated that it outperforms other existing smoothers on challenging convection-diffusion problems.

In Chapter 5 we discuss interfaces to high-performance third-party numerical libraries (TPL) that provide capabilities not available in TRILINOS itself. The first TPL is a sparse matrix library that provides common kernels (e.g., matrix-vector multiplication). We provide numerical experiments on several interesting computing architectures for a number of different matrix types. The second TPL is a truely parallel incomplete factorization library. This is particular important in

combination with the multicore-aware smoother in Chapter 4.

Chapter 1

Introduction

1.0.1 Parallel Algebraic Multigrid

Multigrid methods (e.g., [25], [46], [7]) are among the most efficient iterative algorithms for solving the linear system, $Ax = f$, associated with elliptic partial differential equations. Under certain basic assumptions, it can be shown that the work per unknown required to reduce the residual $f - Ax$ by a specified amount is independent of the problem size. The central idea is to reduce errors by utilizing multiple resolutions in the iterative scheme. High-energy (or oscillatory) components are efficiently reduced through a simple smoothing procedure, while the low-energy (or smooth) components are tackled using an auxiliary lower resolution version of the problem (coarse grid). The idea is applied recursively on the next coarser level. An example multigrid V-cycle iteration is given in Algorithm 1 to solve

$$A_1 u_1 = f_1. \tag{1.1}$$

Algorithm 1: multilevel(A_k, b_k, u_k, k)

```
  Multigrid V-cycle consisting of $N_{levels}$ grids to solve $A_1 u_1 = f_1$.
1:  if ( $k \neq N_{levels}$ ) then
2:      $u_k = \mathscr{S}_k(A_k, b_k, u_k)$
3:      $r_k = b_k - A_k u_k$
4:      $A_{k+1} = P_{k+1}^T A_k P_{k+1}$
5:      $u_{k+1} = 0$
6:      multilevel($A_{k+1}, P_{k+1}^T r_k, u_{k+1}, k+1$)
7:      $u_k = u_k + P_{k+1} u_{k+1}$
8:      $u_k = \mathscr{S}_k(A_k, b_k, u_k)$
9:  else
10:     $u_k = A_k^{-1} f_k$
11: end
```

The two operators needed to specify the multigrid method fully are the relaxation (smoothing) procedures, $\mathscr{S}_k$, $k = 1, \ldots, N_{levels}$, and the grid transfers, P_k, $k = 2, \ldots, N_{levels}$. Note that P_k is an interpolation operator that transfers grid information from level $k+1$ to level k. The interpolation operators lead to a natural definition of the coarse grid operators A_{k+1} by the Galerkin product

$$A_{k+1} = P_k^T A_k P_k, \quad k \geq 1. \tag{1.2}$$

The key to fast convergence is the complementary nature of these two operators. That is, errors not reduced by $\mathscr{S}_k$ must be well interpolated by P_k. While constructing multigrid methods via algebraic concepts presents certain challenges, AMG can be used for several problem classes without requiring a major effort for each application. In the remainder of this report, it is assumed that A_1 and f_1 are given by the application.

We are interested in a particular type of AMG called *smoothed aggregation* (SA) multigrid, which forms the basis for the algorithms in the TRILINOS AMG software library ML. For a detailed description of SA AMG, see [51], [50], [33], [48].

1.0.2 Parallel Performance Bottlenecks

Good performance of SA AMG requires good scaling in both setup and application times. Here, we discuss the leading issues related to scalability.Multigrid methods require the solution of a series of linear systems, typically beginning with the finest (application-level) system (1.1). If some care is not taken, each system can have the same latency characteristics due to message passing as the fine grid system. This is in spite of the decreasing size of the linear systems. In effect, the computation requirements of each system decrease, but the communication penalties remain the same. Multigrid methods can also be sensitive to load-balancing of the matrix data. A poorly balanced fine-level problem can easily result in poor load balancing of *all* the coarse linear systems. In geometric multigrid methods, the problem of interest is discretized on a sequence of increasingly coarse meshes. In contrast, in smoothed aggregation AMG each coarse linear system is created via a triple-matrix Galerkin product, specified in equation 1.2. The kernel here is clearly the matrix-matrix multiplication. This operation requires both the indirect lookup and communication of matrix data. Clearly, this can be hampered by poor parallel load-balancing of the nonzero matrix entries. Poor matrix-matrix multiplication performance can be due to uncontrolled growth in the nonzero density of the individual sparse matrices.

1.0.3 Related Work

Other research efforts have explored a variety of methods to improve parallel AMG scalability. As discussed in [10], specialized parallel AMG methods can generally be categorized into four general types: concurrent iterations, multiple coarse grid corrections, full domain partitioning, and block factorizations.

In standard parallel multigrid, all processors work at the same time to solve a residual equation on a particular mesh. That is to say, the chain of multigrid levels is processed serially, but the solution method on a given level is parallel. In *concurrent iterations*, the chain of multigrid levels is processed in parallel, so that a smoother may run on one level at the same time as a residual equation is being solved on another. Concurrent iterations have been studied in [43, 57, 56, 20, 24, 44, 47, 17, 2, 5]

The second category of specialized parallel multigrid methods are those that uses *multiple coarse grid corrections*. In contrast to standard multigrid, in which there is but one coarse grid correction, this category uses additional coarse grid corrections in an attempt to accelerate convergence. Additional parallelism arises from the fact that each of the coarse grid corrections can be solved simultaneously. The key in all cases is that the corrections must not interfere with one another, or the interference must be constructive in nature. Key papers in this area are [18, 26, 27, 9, 55, 15, 36, 13, 12, 1].

Another category of specialized parallel multigrid is that of *full domain partition*. In this multigrid flavor, there are again many multigrid hierarchies, typically one per processor. The main idea is that the entire domain is meshed and discretized on each domain. However, only a subdomain of this mesh corresponds to the true fine grid mesh. On a processor, the mesh becomes much coarser as the distance from the subdomain increases. Each processor can then traverse its multigrid hierarchy independently, only synchronizing at the finest and coarsest levels. This method was proposed by Mitchell [35, 34].

The last specialized parallel multigrid method is that of parallel multilevel block factorizations. Here, the fine grid matrix written as a block matrix and approximately factored into upper and lower block matrices. The approximate lower block factor contains the well-known *Schur* complement S. If solved exactly, S requires the inverse of block of the original matrix. However, there are many techniques for solving S approximately [32, 11].

1.0.4 Project Overview

This project's objectives were several, and all centered around algebraic multigrid (AMG) algorithms for massively parallel simulations. First, we focused on improving existing parallel algebraic multigrid methods for current and coming computer architectures. Second, we investigated novel algorithms with very different properties than standard methods.

Chapter 2 concentrates on key improvements to computation-intensive kernels in the AMG setup phase. The dominant cost in setup is calculation of coarse grid matrix approximations to the application-supplied linear system. This is done by an explicit triple-matrix product. Our contribution was refactoring the matrix-matrix multiply in the TRILINOS package ML in order to exploit matrix block structure. This improvement demonstrated significant speedups for block-structure systems and evidence that the benefit would be much greater for systems with much larger blocks.

Chapter 3 summarizes our exploration of a AMG method with very different communication patterns from standard multigrid. This method requires communication only on the coarsest multigrid mesh. It is not unusual for latency penalties due to message passing on coarse levels to be as costly as the finest (application) level. Thus, any method that can avoid these costs has potential advantages. We begin with a geometric algorithm proposed in the literature and demonstrate how it can be adapted to AMG methods such as those in TRILINOS. We provide 3D numerical experiments that demonstrate its effectiveness. We discuss limitations to the method that we discovered, as well as its connections to two-level domain decomposition methods.

Chapter 4 discusses a new multigrid smoother for multicore architectures. This smoother utilizes two layers of MPI communicators for managing local and global communication. It is fully incorporated into the TRILINOS package IFPACK and has been tested in an outer AMG solver on up to $10,000$ cores of a Cray XT4. We have demonstrated that it outperforms other existing smoothers on challenging convection-diffusion problems.

In Chapter 5 we discuss interfaces to high-performance third-party numerical libraries (TPL) that provide capabilities not available in TRILINOS itself. The first TPL is a sparse matrix library that provides common kernels (e.g., matrix-vector multiplication). We provide numerical experiments on several interesting computing architectures for a number of different matrix types. The second TPL is a truely parallel incomplete factorization library. This is particular important in combination with the multicore-aware smoother in Chapter 4.

Chapter 2

Improving AMG Sparse Matrix Setup Kernels

In AMG methods, the time to create the preconditioner can be considerable compared to the time to apply the preconditioner. Within ML's AMG setup, a major computational kernel is the matrix-matrix multiply. It is used in the construction the grid transfer operators that move information to and from coarser levels and in the coarse approximations to the operator A_1 in 1.1. Additionally, it is used to form the coarse matrix $A_i, i > 1$, which is often referred to as an *RAP* calculation because A_i is the product of three matrices, R, A_{i-1}, and P. Typically, matrix matrix multiplication accounts for over 50% of the time used to create an ML AMG preconditioner.

Applications that have more than one degree of freedom (DOF) per node often lead to block structured matrices. These matrices can be stored in a special format called *variable block row*, in which the DOFs associated with a node are stored in a dense submatrix. This suggests that we may be able to capitalize on the block structure in the setup and execution of the matrix matrix multiply in order to significantly speedup the setup of the AMG preconditioner.

In this section, we report on a new implementation and initial profiling of a matrix matrix multiply method for variable block matrices. In §2.1, we motivate why a block matrix matrix multiply is important to ML. In §2.2, we give an overview of ML's existing point matrix matrix multiply. In §2.3, we discuss the design and implementation of the block matrix matrix multiply. In §2.4, we provide some initial numerical profiling results. In §2.5, we suggest future directions. Finally, in §2.6 we present the conclusions we draw from our work.

2.1 Motivation for having a block matrix-matrix multiply

Applications governed by systems of PDEs often lead to block structured matrices. Examples of such applications are linear elasticity, chemically reacting flow, and compressible flow calculations. These problems have multiple degrees of freedom (DOFs) associated with each grid point (node) in the problem mesh. The group of DOFs at a node comprise a block of coefficients in the matrix. Matrices with block structure can be stored in a variable block row (VBR) structure [8, 49]. The salient feature of this matrix structure is that individual blocks are stored as dense matrices. Hence, accessing column indices require fewer indirect references, and tuned numerical routines may be used for the dense computation.

Profiling of ML's point matrix matrix multiply has shown that the majority of time to calculate the matrix product AB is in the lookup of B's column indices. More specifically, suppose A and (more importantly) B can be stored as VBR matrices. The reduction in lookups of B's column indices is directly related to the block size in B. If B has $d \times d$ blocks, then the number of column indices is reduced by a factor of d^2, compared to storing B as a point matrix. We note that $d = 3$ is the smallest typical block size. It is not unusual for applications to have $d = 5$ or even larger block sizes. Hence, a reduction of these indirect lookups should lead directly to improvements in the overall runtime.

2.2 Overview of the current point matrix matrix multiply

We first give a high level logical overview of how ML performs a matrix-matrix multiply, $A \times B$. For simplicity, A_i denotes the subset of rows of A stored on processor i. First, rows of B are exchanged among processors so that processor i has all the information that it needs to calculate $A_i \times B$. Second, the column indices of B are stored in global numbering in a hash table for fast lookup. Third, the local product $A_i \times B$ is calculated. Fourth, the product is converted back to local numbering. Descriptions of the major ML functions used in setup and execution of matrix matrix multiplies in ML are given in Table 2.1. As mentioned in §1, the matrix-matrix multiply is an

`Function`	`Description`
`Convert`	Convert matrix from point to VBR format
`Exchange Rows`	Communicates rows of B for the product $A_i \times B$.
`Matrix Matrix Multiply`	Performs actual matrix-matrix multiply
`Back to Local`	Converts matrix column indices from global to local
`Getrow`	Access single point or block row of a block matrix

Table 2.1. Important functions in ML for calculating the matrix product $A \times B$.

important kernel in the setup of ML's multigrid preconditioners. It is used in the creation of the grid transfer operators, P_i, from preliminary transfer operators, $P_i^{(t)}$. For more details on how $P_i^{(t)}$ is constructed, see [51]. Once $P_i^{(t)}$ is available, the prolongator P_i is formed via the step

$$\begin{aligned} P_i &= P_i^{(t)} \\ P_i &\leftarrow (I - \omega_i D_i^{-1} A_i) P_i, \end{aligned} \tag{2.1}$$

where I is an identity matrix, ω_i is a damping parameter, and D_i is the diagonal of A_i. We note that in some cases it is desirable to used repeated applications of (2.1), each of which involves a matrix matrix multiply.

The matrix matrix multiply is also used heavily in the creation of the coarse grid operators A_i, $i > 1$. Once P_i and R_i are available, then A_i is formed as in (1.2). Multiplications are performed from right to left. Proceeding in this manner reduces the memory requirements and operation counts in the intermediate product matrices.

2.3 Design and Implementation of block matrix matrix multiply

In this section, we discuss the design and implementation strategy of the block matrix matrix multiply. As mentioned in §2.1, when the matrix A arises from a system of PDE's, a block matrix multiply has the potential to speedup of the entire multigrid setup, compared to the same calculation with point matrices. This is largely due to multiplication with VBR matrices requiring fewer indirect references.

There are two logical approaches to implementing a block multiplication. In the first approach, every function required to complete the multiplication is refactored to operate natively on block matrices. While this avenue should lead to the best speedups possible, it would also require a large amount of human effort. In the second approach, only certain time-intensive kernels are refactored to operate on block matrices, while the remaining functionality leverages existing point-matrix capabilities.

To keep this project within the scope of a summer, we chose the second approach. Numerical studies in §2.4 demonstrate that this decision still leads to acceptable overall speedups. In the remainder of this section, we discuss the major phases of the multiplication, our changes to key phases, and potential benefits to refactoring the remaining phases.

The first major component that we implemented is a function that converts point matrices to VBR. This function plays four important roles. First, it was very convenient for testing purposes. It allowed us to use existing point matrices to produce VBR matrices. Second, this method is essential for converting (portions of) a matrix from point to block form after exchange rows has been called. Third, this method allows us to convert an existing $P^{(t)}$ to VBR, rather than having to generate $P^{(t)}$ in VBR format initially. [1] Fourth, this method converts R back to VBR after it is created by transposing P. The convert function is sufficiently flexible to be able convert a matrix both before and after rows of that matrix have been communicated. There are two different modes for the function: first, to convert matrices prior to a call to exchange rows; second, to convert the

[1]The first phase in which the matrix matrix multiply is used in the creation of P from $P^{(t)}$. (See (2.1).) From initial performance runs it is unclear whether $AP^{(t)}$ multiplication is faster in point or in block form. This is due to the sparsity of the blocks in $P^{(t)}$, which have nonzero entries only on their main block diagonal. If $P^{(t)}$ is in VBR form, all zero entries within a block must be stored explicitly. This increases the effective number of nonzeros by $n^2 - n$ times for relatively small $n \times n$ blocks. This also increases the amount of data that needs to be exchanged in parallel by a corresponding amount in the exchange rows function. Finally, the number of arithmetic operations is increased a factor of n, which is not be an important factor in the cost as mentioned in §2.1 due to the dominant cost of indirect referencing in the matrix-matrix multiply. We estimate that the cost of converting the point matrix P to VBR is 25% of the cost of creating P initially as a VBR matrix.

data received by exchange rows. The convert function performs a deep copy of data. An important feature of the convert before exchange rows is called is to ensure that blocks are fully populated (dense) with any missing zeros. By doing so, this speeds up the convert of any exchanged rows.

The second major component that we implemented was two getrow methods. One extracts from a VBR matrix a single point row, and the other extracts a single block row. The capability to extract a point row from a VBR matrix allows us to use any existing ML matrix function that requires point row access. In particular, this allowed us to reuse the exchange row function (discussed below). The capability to extract a block row is critical for the core matrix matrix multiply function.

The third major component that we implemented was the matrix matrix multiply kernel. We began this summer project with an existing prototype block multiply. This prototype was capable of squaring a square matrix with $n \times n$ blocks. However, it had several serious limitations. It assumed a fixed block size and worked only in serial. From this prototype, we produced a fully parallel matrix matrix multiply kernel that supports variable block sizes. Tasks included defining a new VBR structure within ML, allowing for variable block sizes for the left matrix and a fixed column width for the right matrix, and establishing correct storage estimates for block matrices.

A function that we decided not to refactor is the exchange rows. As mentioned previously in §2.2, exchange rows must be invoked to communicate rows of B before the product AB can be calculated. Exchange rows accesses matrix data in point fashion (one row at a time). Refactoring this function to access VBR matrices in block fashion could easily have required the entire summer. Moreover, we would have had to ensure that the resulting function's efficiency and scalability were similar to that of the point version. However, because we implemented a VBR matrix getrow that fetches one point row at a time, we were able to reuse the point version of exchange rows.

Refactoring exchange rows may have longer term benefits, however, assuming that a block version has similar performance characteristics to the point version. The cost of data movement of the point version is over 95% of its total cost. A VBR version will still move roughly the same amount of data. However, the data produced by a block exchange row would already be in block format. In contrast, the data from the point version must be converted to block format. The percentage of total time spent in the point exchange row and subsequent convert varies with the amount of data on each processor. At 5000 DOFs per processor, the cost is approximately 66% of the total multiply. At 40000 DOFs per processor, the cost is approximately 25%. Regardless of work per processor, we have observed that the conversion from point to block format requires approximately 25% of the time of the exchange row routine. Based on this data, we expect that a block exchange row could decrease the runtime of each multiply by 5-15%. One necessary component that we have not implemented, but that must be, is back to local. In ML, the product of two matrices is a matrix with column indices that are globally numbered. In order for the product to be used in subsequent calculations, the column indices must be converted to local numbering. Because the underlying VBR data structure is quite different than that of the ML point matrix, ML requires a new method to convert VBR matrices from global to local column indices. Without this capability, the conversion to local is possible but is computationally infeasible.

Finally, we decided not to refactor the point matrix transpose. While not a core piece of the matrix matrix multiply, this function is necessary to the calculation (1.2). We expect that the difference in cost between a block and point transpose operation will be similar to that of exchange rows. This is because each is bound by data transfer, and each exchanges approximately the same information between processors. However, the result of the point transpose will be a fairly dense matrix and will therefore be costly to convert. For this reason, we believe that a native block matrix transpose will be beneficial. The effort to write a block transpose should be significantly less than writing a new exchange rows routine.[2]

2.4 Results

Testing and profiling of functions discussed in §2.3 were performed on the Sandia CSRI machine QED. QED is a 32 node, 64 processor cluster with 2GB of memory per node. Tests were run on three different size matrices, described in Table 2.2. These matrices are typical of those used in elasticity problems and contain 3×3 subblocks. Processor counts from 1 to 40 were used in tests. The larger matrices were not run on the smallest processor counts due to memory limitations. Each calculation involved squaring the matrix. This was done since it is much easier to setup and

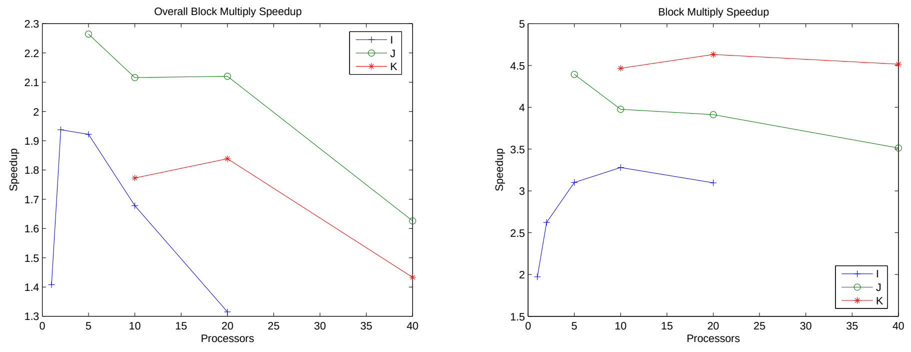
Matrix	Degrees of Freedom	Number of non-zeros
I	26460	1928958
J	201720	15494286
K	403440	31311086

Table 2.2. Test matrices

run tests in this fashion. These tests should be indicative of the potential performance gains from embedding the block multiply fully within the setup of a multigrid cycle for two reasons. First, in the ML RAP process, the intermediate matrices will be have fewer columns than A, and therefore require less time to convert and exchange data than with A itself. Second, 3×3 blocks represent the smallest block size for which the routine can be expected to be used. Other typical sizes such as 3×6, 5×5 and 6×6 will yield larger gains in performance due to less indirect addressing per calculation. As shown by Figure 2.1(a) the new block multiply results in a 1.3 to 2.3 speedup in the overall multiply calculation. This is due to the 2 to 4.5 speedup of the core multiply routine itself, as shown in Figure 2.1(b). Figures 2.2(a) and 2.2(b) show the component breakdown of the overall costs of the point and block routines for matrix J. The exchange rows function in each routine takes approximately the same time for the same processor count. The main advantage of the block routine is from the reduced cost of the multiply routine. The time spent in the two conversions, however, offsets some of this reduction. The conversions account for approximately 25% of the overall runtime, and are a potential spot for further optimization.

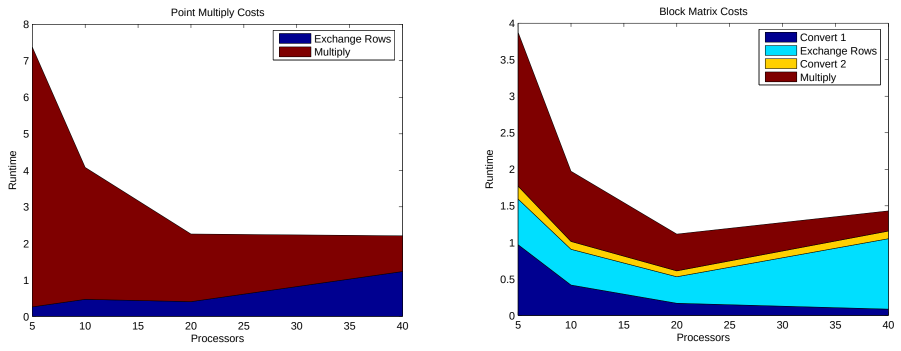
Note that the time for exchange rows in both routines increases when moving from 20 to 40 processors. As there was no attempt to load balance other than the equal distribution of rows

[2]Note that this must be written from scratch, or after exchange rows is rewritten, as the current transpose routine uses a multiplication by the identity in its operation which requires a call to exchange rows.



(a) Overall Speedup **(b)** Multiply Speedup

Figure 2.1. Performance gains from block multiply

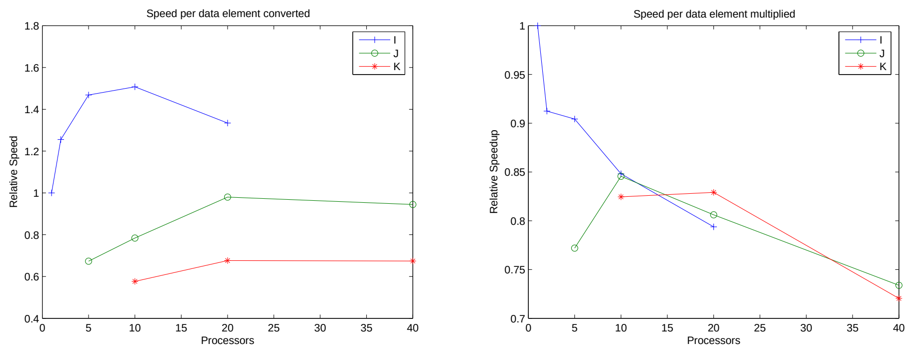


(a) Point Multiply **(b)** Block Multiply

Figure 2.2. Component costs for multiplying the J matrix

among processors this 3 fold increase could be due to a bad data exchange pattern or a bad parallel distribution of matrix rows. In a real application load balancing would likely fix this issue. Figures 2.3(a) and 2.3(b) show the scaling of the convert of the B matrix to VBR and the block multiply. The results are normalized to the speed per nonzero of the I matrix running in serial. Scaling of exchange rows is not shown as previous work has explored its scaling properties, and no work was done on this function during this project. The scalability of the second convert was not studied as its cost is approximately 25% of cost of exchange rows.

What is shown in 2.3(a) is the convert becomes more efficient per nonzero converted as the work per processor decreases up to a certain point, where the trend reverses. In addition for larger matrices the convert is less efficient than for smaller ones. For the multiply 2.3(b) shows that the scaling of the multiply is tied to the number of processors used for the problem. With the exception of the 5 processor example for the J matrix, the efficiency of the computation is nearly identical for each matrix when the number of processors is held constant.



(a) Convert Scaling **(b)** Multiply Scaling

Figure 2.3. Relative speed of convert and multiply routines

2.5 Future work

To fully integrate the block matrix-matrix multiply into the ML multigrid setup phase, a few functions need to be finished. More specifics are outlined in the ML developers documentation [22]. A VBR version of back to local should be written. The writing of a wrapper routine modeled after the current driver and `ML_2matmult()` would make the routine much more accessible for a developer to call.

Within the current design approach, if one were looking for additional efficiency, the following are the best candidates for performance gains. By changing the convert to handle matrices exchanged in point format, $P^{(t)}$ could be more efficiently exchanged. This may increase the convert time on the exchanged rows but would decrease the exchanged information to $1/n$ of its current amount, where $n \times n$ is the block size. The convert routine has not been profiled, and there is a chance it has inefficiencies that could eliminated. Also, while the multiply has no obvious inefficiencies, it may benefit from calls to BLAS [14] routines, especially for larger block sizes. Profiling of this routine might uncover other areas for improvements, though this is unlikely as it was derived from an efficient point multiply. A VBR matrix vector multiply could also lead to performance gains in the application of the multigrid preconditioner.

If full fledged VBR support were desired, we suggest the following order for the implementation. First if a VBR transpose is easy to write, or if an EPETRA function can be utilized, this would be the easiest function to write with potentially the largest performance gains. If the transpose is not easy or requires a block exchange row function to work, then the creation of a block $P^{(t)}$should be the first priority. A new exchange rows function should be lowest priority, unless a block transpose requires it. This is because the expected reduction in runtime of a new block exchange rows is small in comparison to the effort to refactor the code.

2.6 Conclusions

We have demonstrated 2-4.5 times speedups in the multiply kernel for linear systems with 3×3 blocks, and overall speedups of 1.3-2.3, although these results are likely a lower bound on actual performance. Development time was dramatically reduced through the use of a point-to-VBR converter function and existing point matrix capabilities, while still allowing for significant speedups. While we chose to refactor only portions of the multiply, we believe that the results from the initial profiling show this decision was correct.

Chapter 3

Domain-Decomposed Multigrid

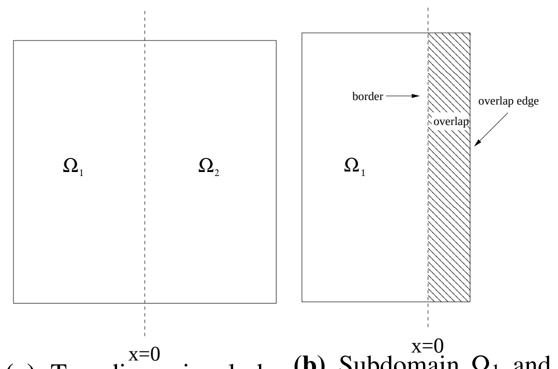
Brandt and Diskin [6] proposed a multigrid method that trades communication for computation. It can be viewed as a hybrid overlapping domain decomposition technique. The key feature is that synchronization is only necessary at the coarsest level. The authors refer to this method as *domain-decomposed multigrid*; we will call this method DDM. We first present Brandt and Diskin's DDM method, then our variant adapted to smoothed aggregation multigrid as well as analysis thereof. Finally, we present experiments illustrating the effectiveness of the proposed solver.

3.1 The Domain-Decomposed Multigrid (DDM) Algorithm

We explain the two-level geometric DDM algorithm proposed by Brandt and Diskin [6] in the context of solving,

$$L^1 u^1 = f^1, \tag{3.1}$$

on the domain in Figure 3.1(a), where level 1 is the fine level, i.e., the system of interest. The



(a) Two-dimensional domain. **(b)** Subdomain Ω_1 and overlap region.

Figure 3.1. Two-dimensional example domain used in the DDM algorithm.

example computational domain Ω is symmetric about the y-axis, e.g., $\Omega = [-1,1] \times [0,1]$. Ω is split into two subdomains, Ω_1 and Ω_2. The dividing line is the y-axis, and the subdomains share the points on the y-axis. In Figure 3.1(b), we show examples of a *border*, *overlap region*, and *overlap edge*. These terms will be used in the following discussion. Table 3.1 shows the notation used in our description of Brant and Diskin's DDM. Algorithm 2 shows the DDM algorithm with our explanatory comments in *italics*. This algorithm can also be executed in a V-cycle (rather than FMG) fashion and this is illustrated in Figure 3.2.

symbol	definition
Ω_p^k	That part of grid Ω_k that belongs to processor p
$\Omega_p^{k,j}$	extension of Ω_p^k by adding all points at a distance of jh_k or closer
$J(k)$	amount of overlap on level k
$\widehat{\Omega}_p^k$	$\Omega_p^{k,j}$ where $j = J(k)$
I_k^j	grid transfer from level k to level j using full-weighting
$\widehat{I}_k^j$	grid transfer from level k to level j using injection
$\widetilde{u}_i^{k,p}$	current approximate solution for processor p at grid point $i \in \widehat{\Omega}_p^k$
$\widetilde{u}_i^k$	"genuine" value of $\widetilde{u}^k$ at point i, given by $\widetilde{u}_i^k = \begin{cases} \widetilde{u}_i^{k,p_i} \text{for } i \text{ not on borderline and living on processor } p_i \\ \frac{1}{N}(\Sigma_{p=1}^N \widetilde{u}_i^{k,p}) \text{ for } i \text{ on borderline and found on } N \text{ processors} \end{cases}$

Table 3.1. Notation for Brant and Diskin's DDM multigrid method.

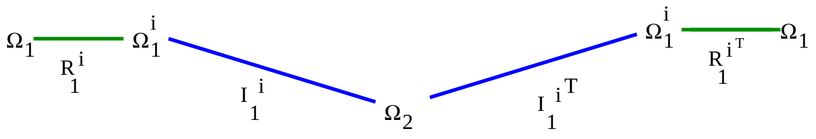


Figure 3.2. A V-cycle variant of Brant and Diskin's DDM

Brant and Diskin discuss several issues with respect to a multilevel (rather than two level) variant of Algorithm 2, though they do not provide a concrete path to implementation. However, they do note that the level K at which communication occurs should be chosen so that the number of points on level K is at least as great as the number of points in the interface [6],

$$h_K \geq \mathcal{O}(h_M^{(d-1)/d}), \tag{3.4}$$

where M is the finest grid level and d is the problem dimension. Of course, we could continue coarsening past level K using a standard multigrid technique should we desire to do so, instead of using a direct solver at level K.

3.2 A Full Algebraic Multilevel Variant of DDM

Brant and Diskin's DDM has two main limitations. First, its extension from a two level method to a multilevel method is not obvious. Second, it is a purely geometric algorithm. We now propose a fully algebraic multilevel variant of the DDM. Notation for this variant of DDM can be found in Table 3.2. Our proposed method is divided into three separate parts — treatment on the fine

Algorithm 2: Brant & Diskin Two Level DDM Method

```
   % Establish a good initial guess.
1: Solve $L^2 u^2 = f^2 = I_1^2 f^1$ however you like.
   % Interpolate this approximate solution to the fine grid as follows.
2: For each overlapping subdomain $\widehat{\Omega}_p^1$, $\widetilde{u}_i^{1,p} = (\Pi_2^1 u^2)_i$ for each $i \in \widehat{\Omega}_p^1$, where $\Pi_2^1$ is bicubic
   interpolation.
   *Higher order interpolation is standard in FMG for the initial fine grid approximate solution. In overlap region,
   solution is the same for each processor p.*
3: while not done do
       % Pre-smoothing
4:     Each processor $p \in \{1,2\}$ smooths $\nu$ times on its subdomain $\widehat{\Omega}_p^k$. (Note: in the overlap
       region, solution no longer the same.)
       *We have observed that the points at the overlap edge must* not *be smoothed (see Figure 3.1(b)). These
       points can be thought of as boundary points, but where the boundary conditions are unknown. Smoothing
       them pollutes the solution in the rest of the subdomain. To deal with them, we turn them into Dirichlet
       points during smoothing.*
       % Coarse Grid Correction
5:     Calculate the coarse right-hand side $\widetilde{f}^2 = L^1(\widehat{I}_1^2 \widetilde{u}^1) + I_1^2 r^1$, where the coarse grid vectors
       $\widehat{I}_1^2 \widetilde{u}^1$ and $I_1^2 \widetilde{r}^1$ are given as follows:

       $$(\widehat{I}_1^2(\widetilde{u}^1))_i = \begin{cases} \widetilde{u}_{1i}^{1,p} & \text{for interior points } i \\ \frac{1}{2}(\Sigma_p \widetilde{u}_{1i}^{1,p}) & \text{for boundary points } i \end{cases} \qquad (3.2)$$

       $$(I_1^2 \widetilde{r}_i^1)_i = \begin{cases} f_i^1 - L^1 \widetilde{u}_i^{1,p} & \text{for interior points} \\ f_i^1 - \frac{1}{2}(\Sigma_p(L^1 \widetilde{u}_i^{1,p})) & \text{for boundary points } i \end{cases} \qquad (3.3)$$

       *The solution is restricted via injection. The residual is restricted via full-weighting.*
6:     Solve $L^2 \widetilde{u}^2 = \widetilde{f}^2$ exactly.
7:     For each processor $p$, correct approximate solution is: $\widetilde{u}^{1,p} \leftarrow \widetilde{u}^{1,p} + I_1^2(\widetilde{u}^2 - \widehat{I}_1^2 \widetilde{u}^{1,p})$.
       *The solution, $\widetilde{u}^{1,p}$, in the overlap region must be consistent among processors before adding in the coarse
       grid correction. This is due to the fact that the coarse grid correction is calculated using a consistent
       solution.*
       % Post-smoothing
8:     Each processor $p$ smooths $\nu'$ times on its subdomain $\widehat{\Omega}_p^k$.
9: end
```

level, treatment on the coarse level, and treatment on all intermediate levels. The coarse level most closely resembles that of the V-cycle variant of Algorithm 2. From there up we eschew all communication until we reach the fine level, where we will need to synchronize solutions between domains. Algorithm 3 shows a three-level variant of the proposed method, which is also illustrated in Figure 3.3.

notation	definition
G_j	Standard AMG "grid" on level j.
A_j	Standard AMG matrix on grid level j
P_j	Standard AMG prolongator from grid level j to $j+1$ ($P_j : G_j \rightarrow G_{j+1}$).
$G_j^{(i)}$	"Grid" for overlapping subdomain i at level j.
$R_j^{(i)}$	Maps from unified (AMG) grid to overlapping subdomain i at level j ($R_j^{(i)} : G_j \rightarrow G_j^{(i)}$).
$A_j^{(i)}$	Matrix corresponding to domain i on grid-level j.
$P_j^{(i)}$	Prolongator on subdomain i from level j to $j+1$ ($P_j^{(i)} : G_j j^{(i)} \rightarrow G_{j+1}^{(i)}$).

Table 3.2. Algebraic Multilevel DDM Notation

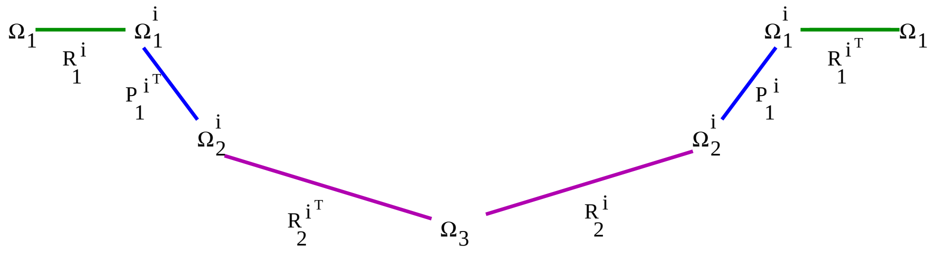


Figure 3.3. Fully Algebraic Multilevel DDM

The Fine Level (1)

We first define $R_1^{(i)}$ operators that transfer between the consistent global grid and various subdomains. For the case of two subdomains, they will look like,

$$R_1^{(1)} = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \end{bmatrix}, \tag{3.5}$$

$$R_1^{(2)} = \begin{bmatrix} 0 & I & 0 \\ 0 & 0 & I \end{bmatrix}, \tag{3.6}$$

where the columns are ordered in the order $G_1^{(1)} \setminus G_1^{(2)}$, $G_1^{(1)} \cap G_1^{(2)}$, $G_1^{(2)} \setminus G_1^{(1)}$. The multi-domain case is handled similarly.

The $R_1^{(i)}$ operators now allow us to define the operators on each domain $A_1^{(i)}$ in the Galerkin sense, namely,

$$A_1^{(i)} = R_1^{(i)} A_1 \left(R_1^{(i)}\right)^T. \tag{3.7}$$

Levels $2,\ldots,n-1$

Starting with level j, we now desire to define operators for level $j+1$. We then create prolongators, $P_j^{(i)}$, which create coarser levels unique to each domain. In order to glue the domains back together at the bottom of the hierarchy, these prolongators must be chosen such that nodes which are part of multiple subdomains (i.e. the nodes in the overlap) are aggregated in the same fashion on each subdomain.

We can now define the matrix on the next level by an ordinary Galerkin product,

$$A_{j+1}^{(i)} = \left(P_j^{(i)}\right)^T A_j^{(i)} P_j^{(i)}. \tag{3.8}$$

The Coarse Level (n)

To move towards the coarse problem, we need to be able to glue things back together at level $n-1$. Since we have assumed that the intermediate levels have coarsened the overlap regions consistently between subdomains, we can now create R operators for level $n-1$.

$$\begin{aligned} R_{n-1}^{(1)} &= \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \end{bmatrix} \\ R_{n-1}^{(2)} &= \begin{bmatrix} 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} \end{aligned} \tag{3.9}$$

where the columns are ordered in the order $G_{n-1}^{(1)} \setminus G_{n-1}^{(2)}$, $G_{n-1}^{(1)} \cap G_{n-1}^{(2)}$, $G_{n-1}^{(2)} \setminus G_{n-1}^{(1)}$. The multi-domain case is handled similarly.

Using those R operators and AMG's prolongator for that level P_{n-1} we can now form the Galerkin coarse grid operator,

$$\widetilde{A}_n = P_{n-1}^T \left(\sum_i \left(R_{n-1}^{(i)}\right)^T A_{n-1}^{(i)} R_{n-1}^{(i)} \right) P_{n-1}. \tag{3.10}$$

3.3 Analysis of Multilevel DDM

Like any DDM or multilevel method, the entire technique can be written as a matrix splitting. For simplicity's sake, assume that our smoothing operator, called $S_j^{(i)}$, is applied directly to the relevant

Algorithm 3: Sample 3-Level Algebraic DMM

```
    % Form residual
1:  $r_1 = b_1 - A_1 x_1$
2:  For $i = 1, \dots, p$
        % Grid Transfer $G_1 \to G_1^{(i)}$
3:      $b_1^{(i)} = R_1^{(i)} r_1, \;\; x_1^{(i)} = 0.$
        % Smooth on Level 1
4:      $x_1^{(i)} = \left(M_1^{(i)}\right)^{-1} b_1^{(i)}$
        % Residual
5:      $r_1^{(i)} = \left(I - A_1^{(i)} \left(M_1^{(i)}\right)^{-1}\right) b_1^{(i)}$
        % Grid Transfer $G_1^{(1)} \to G_2^{(i)}$
6:      $b_2^{(i)} = \left(P_1^{(i)}\right)^T r_1^{(i)}, \;\; x_2^{(i)} = 0.$
        % Smooth on Level 2
7:      $x_2^{(i)} = \left(M_2^{(i)}\right)^{-1} b_2^{(i)}$
        % Residual
8:      $r_2^{(i)} = \left(I - A_2^{(i)} \left(M_2^{(i)}\right)^{-1}\right) b_2^{(i)}$
9:  end
    % Form $r_2$
10: $r_2 = \sum_{i=1}^{p} R_2^{(i)} r_2^{(i)}$
    % CGC on Level 3
11: $x_2 = P_2 \widetilde{A}_3^{-1} P_2^T r_2$
12: For $i = 1, \dots, p$
        % Update Level 2
13:     $x_2^{(i)} += \left(R_2^{(i)}\right)^T x_2$
        % Smooth on Level 2
14:     $x_2^{(i)} += \left(M_2^{(i)}\right)^{-1} (b_2^{(i)} - A_2^{(i)} x_2^{(i)})$
        % Grid Transfer $G_2^{(1)} \to G_1^{(i)}$
15:     $x_1^{(i)} += P_2^{(i)} x_2^{(i)}$
        % Smooth on Level 1
16:     $x_1^{(i)} += \left(M_1^{(i)}\right)^{-1} (b_1^{(i)} - A_1^{(i)} x_1^{(i)})$
17: end
    % Create Solution
18: $x_1 = \sum_{i=1}^{p} \left(R_1^{(i)}\right)^T x_1^{(i)}$
```

residual (i.e. it has a zero initial guess). Then the three-level method shown in Algorithm 3 can be written as,

$$x = \sum_{i=1}^{p} \left(R_1^{(i)} S_1^{(i)} P_1^{(i)} S_2^{(i)} (R_2^{(i)})^T \right) P_2 \widetilde{A}_3^{-1} P_2^T \sum_{i=1}^{p} \left(R_2^{(i)} S_2^{(i)} (P_1^{(i)})^T S_1^{(i)} (R_1^{(i)})^T \right) r_1. \tag{3.11}$$

Adding additional levels just adds terms to the appropriate sums. From this matrix splitting statement, we can look at the analysis of our multilevel DDM in two ways, both of which are based on the analysis of classical Schwarz methods. The first such classical method is 1-level additive Schwarz [43, Algorithm 1.3.3], which we restate in our notation for convenience in Algorithm 4. The second algorithm is a method which applies a *multiplicative* coarse grid correction and an *additive* domain decomposition correction, which is referred to the 2-level Hybrid II Schwarz preconditioner in [43, Algorithm 2.3.5]. We reproduce this in Algorithm 5. In both of these cases, the subdomain correction $B_1^{(i)}$ is usually implemented with a direct solve on the subdomain, namely, $B_1^{(i)} = (R_1^{(i)})^T (A_2^{(i)})^{-1} R_1^{(i)}$. However this correction can be performed approximately, so long as the approximation is good enough, and the relevant results still hold.

Algorithm 4: 1-Level Additive Schwarz

```
1: $x = \sum_{i=1}^{p} B_1^{(i)} r_1$
```

Algorithm 5: 2-Level Hybrid II Schwarz

```
1: $x = \sum_{i=1}^{p} B_1^{(i)} r_1$
2: $x = x + P_1 A_2^{-1} P_1^T (r_1 - A_1 x)$
```

We now detail two ways of analyzing our DDM method. The first is based on the one-level additive Schwarz technique shown in Algorithm 4. The second is based on the Hybrid II technique, shown in Algorithm 5.

3.3.1 DDM as One-Level Additive Schwarz

We note first that our method can be considered as an additive Schwarz method on the fine-level with approximate subdomain solves (which include the rest of the V-cycle). More specifically, we can define the $B_1^{(i)}$ operators as,

$$B_1^{(i,j)} = \left(R_1^{(i)} S_1^{(i)} P_1^{(i)} S_2^{(i)} (R_2^{(i)})^T \right) P_2 \widetilde{A}_3^{-1} P_2^T \left(R_2^{(j)} S_2^{(j)} (P_1^{(j)})^T S_1^{(j)} (R_1^{(j)})^T \right), \tag{3.12}$$

which is a kind of Petrov-Galerkin “projector-like” correction. If the subdomain smoothers and grid transfer operators are good enough, then it is easy to show that the DDM method converges at least as well as classical one-level additive Schwarz with an approximate subdomain solve.

Assume that h represents the mesh diameter of a typical Laplace problem. It is well know that when additive Schwarz is used as a preconditioner for a Krylov method, it does not have h-independent convergence unless a coarse grid correction is used [45]. Thus, viewing DDM as a one-level Schwarz technique cannot give us the convergence result we desire.

3.3.2 DDM as Two-Level Additive Schwarz

From our discussions with Clark Dorhmann, we note that there are certain circumstances when our multilevel DDM is exactly equivalent to a two-level additive Schwarz method. Specifically, consider the symmetric two-level Schwarz method shown in picture form in Figure 3.4. This method uses a two-level multigrid algorithm on each subdomain and an aggressively coarsened coarse grid correction that creates a grid which is coarser than any of the subdomain grids.

For simplicity, we'll look at the three-level case, though everything extends to hierarchies of arbitrary depth *mutatis mutandis*. Consider again Algorithm 3, as shown in Figure 3.3. Let us change the way we define the $R_2^{(i)}$ operators from (3.9). Instead, let

$$R_2^{(i)} = P_{1,2} R_1^{(i)} (P_1^{(i)})^T, \tag{3.13}$$

where $P_{1,2}$ is the aggressive prolongator shown in Figure 3.4. Figure 3.5 shows graphically that these new $R_2^{(i)}$'s allow us to create a multilevel DDM which is exactly equivalent to a two-level symmetric Hybrid II algorithm that uses a two-level multigrid on each subdomain. Under a certain set of assumptions, the $R_2^{(i)}$'s given by (3.9) and by (3.13) are in fact identical. First, the overlap must be coarsened in a fashion such that overlap nodes are only aggregated with other overlap nodes. Second, the overlap nodes must be aggregated in the same fashion regardless of which domain they are in. Third, prolongator smoothing cannot be used. If all three of these conditions hold, then the two methods produce identical $R_2^{(i)}$'s, for ideal aggregates on a regular mesh.

In practice, the approach of (3.13) is preferable since it allows us more flexibility in the aggregation process than the approach of (3.9). Perhaps more importantly, the approach of (3.9) requires that the overlap on the fine grid be sufficiently large to allow it to be coarsened repeatedly, which means the overlap must grow as the number of levels needed increases. With this is mind, we have concluded that (3.13) is a superior approach in practice.

Let H be the size of the subdomain and δ measure the width of the overlap. Then the conditioned number of a Laplace problem preconditioned with a two-level Schwarz method, such as the one shown in Algorithm 5 is $O(1 + H/\delta)$. This means that if we fix the ratio of subdomain to overlap and if the preconditioner on the individual subdomains is good enough, we can expect similar convergence for our DDM.

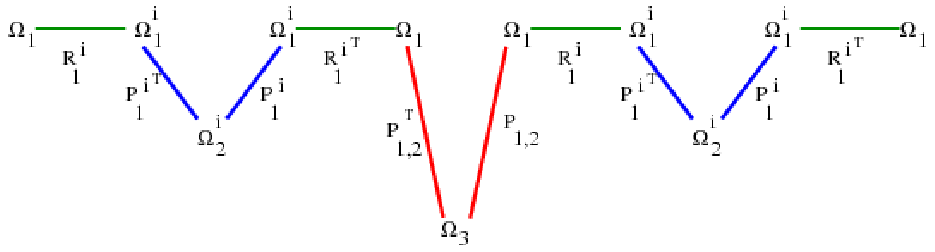


Figure 3.4. Two-level additive Schwarz method with approximate subdomain solves

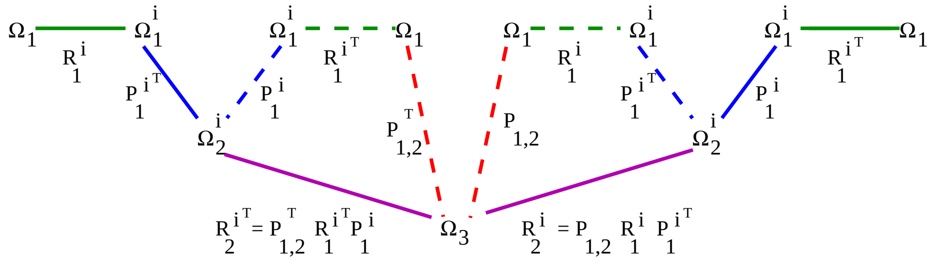


Figure 3.5. Multilevel DDM as a two-level additive Schwarz method

3.4 Experiments

Consider the model problem:

$$\begin{aligned} \Delta u &= f \text{ on } \Omega \\ u &= 0 \text{ on } \delta\Omega \end{aligned} \tag{3.14}$$

where $\Omega = [0,1]^d$ and d is the number of dimensions. We discretize this problem with finite differences and consider the performance of our DDM solver. We consider the problem in one, two and three dimensions, varying the number of levels in our hierarchy between two and five and varying the number of domains between two and 27. We fix the amount of overlap at two fine grid points, which means that both H and δ should shrink as the grid is refined, although the latter will shrink far more rapidly than the former.

Figure 3.3 shows both GMRES iterations and operator complexity for the 1D version of the test problem. We note that the operator complexity is comparable to that of a traditional smoothed aggregation multigrid algorithm (which is about 1.5 for this problem). In the 2D case, shown in Figure 3.4, we note that our methods are slightly more expensive in terms of operator complexity than traditional smoothed aggregation AMG (which is about 1.2 for this problem). The 3D case, as shown in Figure 3.5, is similar.

In Figures 3.3, 3.4 and 3.5, we can interpret the iteration counts in four different ways. First, we can read *between subtables*, which shows the effect of changing the number of levels in the multilevel method. Each additional level added the hierarchy is another level without communication and therefore a smaller (globally synchronized) coarse problem. Thus we should expect an increase in the number of iterations as the number of levels increases.

If we read *across a row*, we are decreasing H while fixing h and δ. In this regime, we should see a condition a number of $O(1+H/\delta)$. Thus, convergence should be better as we add more subdomains. We do not seem to see that in practice, but this is likely because our number of subdomains is too small for this asymptotic result to take hold.

If we read *down a column* we are refining h and shrinking δ. In this regime, our method should yield a condition number of $O(2+H_0/(c_0h))$, where $c_0 = \delta_0/h$, H_0 is the diameter of the subdomain without overlap and δ_0 is the amount of overlap on the coarsest grid. This predicts performance degradation in the asymptotic case, although we don't seem to see that in practice.

If we read *diagonally* from top left to bottom right, we are decreasing both h and H. This should yield a $O(1)$ condition number and thus a flat iteration profile. Once we get past very small numbers of subdomains we do indeed see this behavior.

	# of Domains			
Grid Refinement	2	3	9	27
Number of Levels: 2 Levels				
27	7	7	7	$*$
81	7	7	7	7
243	7	7	7	7
729	7	7	7	7
2,187	7	7	7	7
6,561	7	7	7	7
19,683	7	7	7	7
59,049	7	7	7	7
177,147	7	7	7	7
531,441	7	7	7	7
Number of Levels: 3 Levels				
27	8	9	$*$	$*$
81	8	9	13	$*$
243	8	9	10	13
729	8	9	9	10
2,187	8	9	9	9
6,561	8	9	9	9
19,683	8	9	9	9
59,049	8	9	9	9
177,147	8	9	9	9
531,441	8	9	9	9
Number of Levels: 4 Levels				
81	8	12	$*$	$*$
243	9	12	18	$*$
729	9	10	13	21
2,187	9	10	11	13
6,561	9	10	10	10
19,683	9	10	10	11
59,049	9	10	10	10
177,147	9	10	10	10
531,441	9	10	10	10
Number of Levels: 5 Levels				
243	9	14	$*$	$*$
729	9	13	23	$*$
2,187	9	12	15	35
6,561	9	11	13	15
19,683	9	11	13	13
59,049	9	11	12	13
177,147	9	11	11	13
531,441	9	11	11	12

	# of Domains			
Grid Refinement	2	3	9	27
Number of Levels: 2 Levels				
27	1.44	1.57	2.33	$*$
81	1.37	1.41	1.66	2.41
243	1.35	1.36	1.44	1.69
729	1.34	1.34	1.37	1.45
2,187	1.33	1.34	1.35	1.37
6,561	1.33	1.33	1.34	1.35
19,683	1.33	1.33	1.33	1.34
59,049	1.33	1.33	1.33	1.33
177,147	1.33	1.33	1.33	1.33
531,441	1.33	1.33	1.33	1.33
Number of Levels: 3 Levels				
27	1.54	1.61	$*$	$*$
81	1.48	1.50	1.70	$*$
243	1.46	1.46	1.53	1.73
729	1.45	1.45	1.47	1.54
2187	1.45	1.45	1.45	1.48
6,561	1.44	1.45	1.45	1.45
19,683	1.44	1.44	1.45	1.45
59,049	1.44	1.44	1.44	1.45
177,147	1.44	1.44	1.44	1.44
531,441	1.44	1.44	1.44	1.44
Number of Levels: 4 Levels				
81	1.51	1.51	$*$	$*$
243	1.49	1.49	1.54	$*$
729	1.48	1.48	1.50	1.55
2,187	1.48	1.48	1.49	1.50
6,561	1.48	1.48	1.48	1.49
19,683	1.48	1.48	1.48	1.48
59,049	1.48	1.48	1.48	1.48
177,147	1.48	1.48	1.48	1.48
531,441	1.48	1.48	1.48	1.48
Number of Levels: 5 Levels				
243	1.50	1.50	$*$	$*$
729	1.50	1.49	1.51	$*$
2187	1.49	1.49	1.50	1.51
6,561	1.49	1.49	1.50	1.50
19,683	1.49	1.49	1.49	1.50
59,049	1.49	1.49	1.49	1.49
177,147	1.49	1.49	1.49	1.49
531,441	1.49	1.49	1.49	1.49

Table 3.3. Preconditioned GMRES iterations and multilevel operator complexities for a 1D Laplace problem (3.14) preconditioned with multilevel DDM. Asterisks($*$) indicate invalid combinatations of number of domains and grid refinement.

Grid Refinement	# of Domains 2	3	9	27
Number of Levels: 2 Levels				
9^2	8	8	$*$	$*$
27^2	9	10	10	$*$
81^2	10	10	10	10
243^2	10	10	10	10
729^2	10	10	10	10
Number of Levels: 3 Levels				
27^2	10	12	$*$	$*$
81^2	11	13	15	$*$
243^2	11	12	12	15
729^2	11	12	12	12
Number of Levels: 4 Levels				
81^2	15	19	$*$	$*$
243^2	17	18	24	$*$
729^2	17	18	19	25
Number of Levels: 5 Levels				
243^2	24	28	$*$	$*$
729^2	27	30	38	$*$

Grid Refinement	# of Domains 2	3	9	27
Number of Levels: 2 Levels				
9^2	1.98	2.85	$*$	$*$
27^2	1.45	1.72	3.41	$*$
81^2	1.28	1.37	1.91	3.60
243^2	1.23	1.26	1.44	1.98
729^2	1.21	1.22	1.28	1.46
Number of Levels: 3 Levels				
27^2	1.57	1.86	$*$	$*$
81^2	1.40	1.51	2.03	$*$
243^2	1.35	1.38	1.56	2.09
729^2	1.33	1.34	1.40	1.58
Number of Levels: 4 Levels				
81^2	1.42	1.53	$*$	$*$
243^2	1.37	1.40	1.58	$*$
729^2	1.35	1.36	1.42	1.60
Number of Levels: 5 Levels				
243^2	1.37	1.40	$*$	$*$
729^2	1.35	1.36	1.42	$*$

Table 3.4. Preconditioned GMRES iterations and multilevel operator complexities for a 2D Laplace problem (3.14) preconditioned with multilevel DDM. Asterisks($*$) indicate invalid combinatations of number of domains and grid refinement.

Grid Refinement	# of Domains 2	3
Number of Levels: 2 Levels		
9^3	9	9
27^3	11	11
81^3	12	12
Number of Levels: 3 Levels		
27^3	12	14
81^3	13	14

Grid Refinement	# of Domains 2	3
Number of Levels: 2 Levels		
9^3	2.41	3.84
27^3	1.54	1.98
81^3	1.27	1.41
Number of Levels: 3 Levels		
27^3	1.86	2.25
81^3	1.65	1.77

Table 3.5. Preconditioned GMRES iterations and multilevel operator complexities for a 3D Laplace problem (3.14) preconditioned with multilevel DDM. Asterisks($*$) indicate invalid combinatations of number of domains and grid refinement.

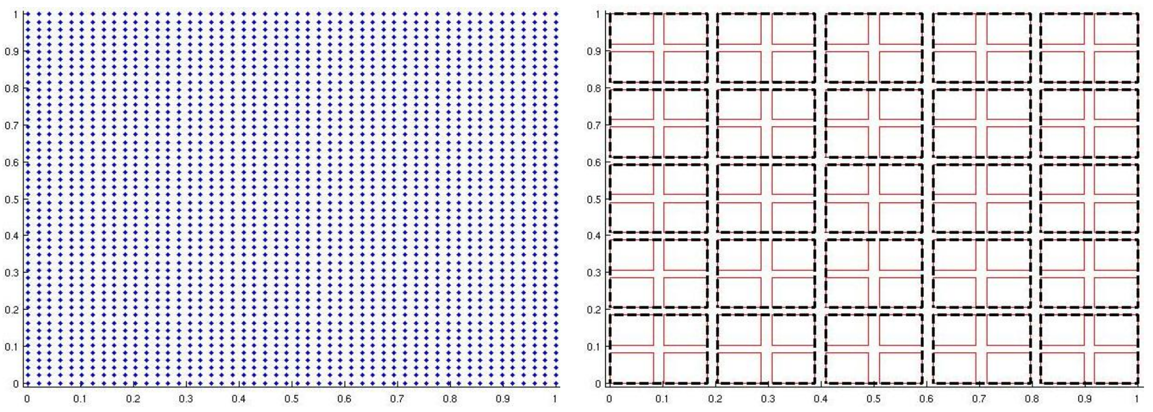
Chapter 4

Multicore-aware Smoothing

In this chapter we describe a novel smoother intended for distributed memory machines with many cores per compute node. The motivation for this smoother is to address current and future computer architectures that are characterized by increasing on-chip core counts. In essence, this smoother allows for much more expensive local computation, e.g., incomplete factorizations or local multigrid methods, while minimizing off-node computation in a domain-decomposition like manner.

4.0.1 Algorithm Description

The smoother can be viewed as an overlapping domain decomposition method. In this case, the subdomains are groups of matrix rows. Subdomain identification is determined at runtime and is controlled by the calling application. A natural definition is to assign the processes on a given compute node to the same subdomain. This is not strictly required, however. A node could be divided into multiple subdomains, or subdomains could span nodes. In any case, the processor groups must be disjoint. Figure 4.1 shows the layout of compute nodes and individual MPI tasks for a simple two-dimensional mesh on the unit square. Once processor groups are identified, each subdomain



(a) Mesh nodes **(b)** Subdomain and processor distribution.

Figure 4.1. (**a**) Nodes of two-dimensional Cartesian mesh. (**b**) Mesh node ownership. The heavy dashed squares denote subdomain ownership. The thin solid squares denote individual MPI task ownership.

is assigned its own message-passing communicator. These processor-group communicators handle all intra-group processor communication and thus allow for independent computation on each subdomain that can proceed in parallel. A global communicator associated with the linear operator then provides for intergroup communication. The subdomains may either be nonoverlapping or overlapped.

The overall AMG algorithm can then be applied as in Algorithm 1, with smoothers on one or more levels replaced by the multicore-aware smoother. Within the multicore-aware smoother, we have complete freedom in the choice of subdomain solve. For example, one can use a very lightweight solve, e.g., Chebyshev polynomials, or more expensive solves, such as an incomplete factorization. In practice, we have used a multigrid method on the local domain as a smoother.

4.0.2 Software Description

All algorithms are implemented within the TRILINOS package IFPACK [39]. Ifpack provides a suite of algebraic preconditioners, such as successive overrelaxation methods and incomplete factorizations. These methods are available as smoothers, via factory interfaces, to ML [23], the TRILINOS AMG solver package. The two main classes in Ifpack that implement the multicore-aware smoother are Ifpack_OverlappingRowMatrix and Ifpack_NodeFilter . The first class has been heavily modified and extended from an existing implementation, and the second class is new.

Class Ifpack_OverlappingRowMatrix

Class Ifpack_OverlappingRowMatrix inherits from the pure virtual class Epetra_RowMatrix. The constructor for class Ifpack_OverlappingRowMatrix is given here:

```
Ifpack_OverlappingRowMatrix(const Teuchos::RCP<const
    Epetra_RowMatrix> &Matrix_in, int OverlapLevel_in, int
    myNodeID )
```

The constructor accepts an Epetra_RowMatrix. The amount of overlap is specified by the second parameter. Subdomain assignment is based upon the value of myNodeID, the node identifier. All processes that have the same identifier are assigned to the same subdomain and MPI communicator. The processes in each subdomain work in concert to identify ghost information that must be imported from other subdomains. Each process sends the ghost rows that it requires to local process 0. Process 0 acts as the arbiter to break ties between processes that depend on the same off-node information and need to import it. It is important to note that this class does not duplicate information on the interior of a subdomain. Only ghost information is duplicated on the local subdomain. The amount of duplication is proportional to the subdomain boundary size, and thus a low order effect. Class OverlappingRowMatrix establishes the proper EPETRA row and column maps such that an additive Schwarz domain decomposition method can be applied. We note that the overlapped matrix's row map is *not* one-to-one, as is usually the case. This is due to the fact that ghost row information is replicated on each subdomain.

Class Ifpack_NodeFilter

The class Ifpack_NodeFilter is a lightweight class that allows independent computation on each subdomain. This class also inherits from the pure virtual class Epetra_RowMatrix. The constructor's signature is given by

```
Ifpack_NodeFilter(const Teuchos::RCP<const Epetra_RowMatrix>
    &Matrix, int nodeID)
```

The constructors accepts an Epetra_Rowmatrix and a node identifier, nodeID, that is the same as that supplied to Ifpack_OverlappingRowMatrix. Class Ifpack_NodeFilter does not copy data. However, it recalculates Epetra maps, importers, and exporters based upon a local communicator. This is so that local solves be applied independently of any other subdomain solves. We note that this recalculation is done only when the smoother is set up and is not compute intensive. The local remapping does require an additional layer of indirection when Ifpack_NodeFilter −>Apply() is called, i.e., during a subdomain matrix/vector multiply. Profiling indicates that this overhead is quite low.

4.0.3 Invocation

An application typically will not interact with either class directly. Instead, multicore-aware smoothing is requested as an option while setting up the AMG preconditioner. This is done via a TEUCHOS parameter list option. Internally, ML calls the Ifpack additive Schwarz solver factory, which in turn calls both the Ifpack_OverlappingRowMatrix and Ifpack_LocalFilter constructors. Figure 4.3 shows a fragment of an XML input deck.

4.0.4 Numerical Results

We now examine the effectiveness of an AMG preconditioner that uses the multicore-aware smoother. The problem we will consider is a two-dimensional recirculating flow on the unit square with Dirichlet boundary conditions:

$$-\varepsilon \Delta u + \mathbf{v} \cdot \nabla u = f, \qquad (4.1)$$

where the components of **v** are given by

$$\begin{aligned} v_x &= 4x(x-1)(1-2y), \\ v_y &= -4y(y-1)(1-2x). \end{aligned}$$

The diffusive term is discretized using a five-point stencil and the convective term is discretized using a standard upwind stencil.

The subsequent experiments use $\varepsilon = 10^{-8}$. We considered weak-scaling on the Oak Ridge National Laboratory machine "Jaguar", a Cray XT4. Each node has a single quad-core AMD Opteron 1354 (Budapest) 2.1 GHz processor and 8 gigabytes of RAM.

Figure 4.2 illustrates scaling results on the Cray XT4 for problem (4.1). It compares the best existing AMG method to an AMG method using the multicore aware smoother. Note that the two methods are competitive until 8100 cores. From 8464 to 10000 cores, the AMG with multicore-aware smoother demonstrates modestly better solve times. Details of the right portion of the plot

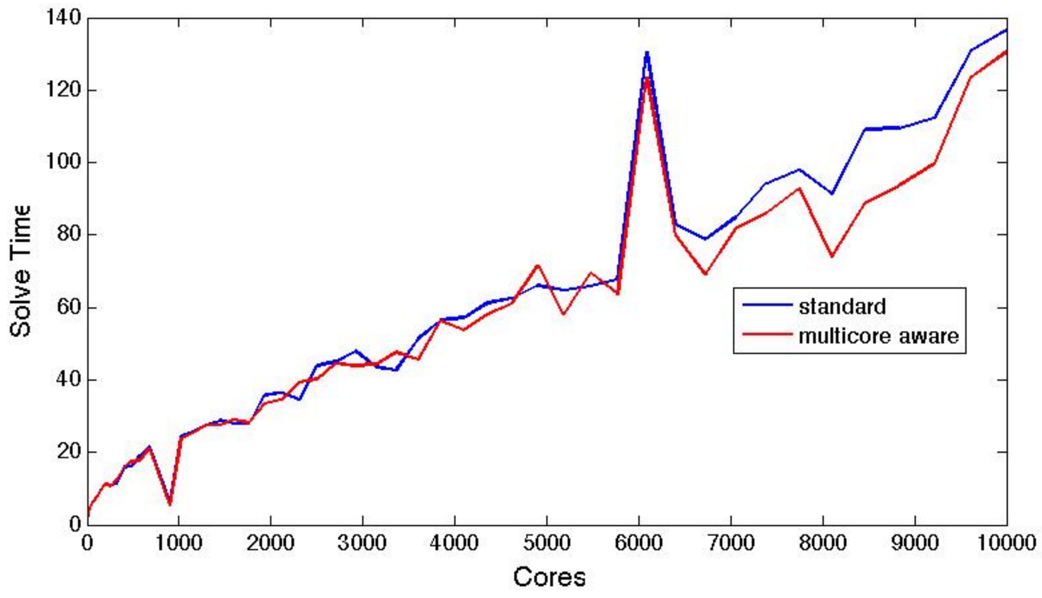


Figure 4.2. Weak scaling on Cray XT4.

(from 8100 cores upwards) are given in Table 4.1. This table compares AMG methods that use multicore-aware and standard ("single core") smoothers. Note that the W-cycle improves iteration counts, as expected. However, it also is the most expensive in solution time. The AMG with multicore smoother is slightly more expensive to setup, but this cost is offset by its faster solve times.

#cores	Method	Iteration	Solution Time	Timer per iteration	Setup Time
6724	multicore	278	69.0050	0.2482	3.0385
	single core	300	78.8532	0.2628	2.1076
	multicore W cycle	119	104.8935	0.8815	3.1808
	single core W cycle	119	103.4593	0.8694	2.2932
7056	multicore	289	81.6527	0.2825	3.1513
	single core	293	84.8143	0.2895	2.1667
	multicore W cycle	125	111.0956	0.8888	3.4737
	single core W cycle	120	108.3071	0.9026	2.2156
7396	multicore	344	86.1292	0.2504	3.5490
	single core	381	94.3887	0.2477	2.2432
	multicore W cycle	135	128.3267	0.9506	3.2964
	single core W cycle	135	136.5535	1.0115	2.2897
7744	multicore	356	92.8327	0.2608	3.7389
	single core	363	97.8513	0.2696	2.2723
	multicore W cycle	128	127.4574	0.9958	4.0151
	single core W cycle	120	115.1721	0.9598	2.4379
8100	multicore	296	73.9988	0.2500	3.8733
	single core	320	91.2657	0.2852	2.3032
	multicore W cycle	121	117.8392	0.9739	4.4177
	single core W cycle	140	142.1480	1.0153	2.6778
8464	multicore	359	88.9224	0.2477	3.9305
	single core	383	109.2368	0.2852	2.6478
	multicore W cycle	130	134.5475	1.0350	3.5371
	single core W cycle	136	148.2833	1.0903	2.3605
8836	multicore	341	93.6434	0.2746	3.7616
	single core	400	109.4229	0.2736	2.4052
	multicore W cycle	150	158.5187	1.0568	3.8069
	single core W cycle	149	159.8648	1.0729	2.4483
9216	multicore	386	99.6191	0.2581	4.5088
	single core	439	112.2955	0.2558	2.6458
	multicore W cycle	125	136.9810	1.0958	4.7513
	single core W cycle	136	145.2709	1.0682	2.5478
9604	multicore	458	123.4286	0.2695	3.9548
	single core	454	130.6036	0.2877	2.8888
	multicore W cycle	147	177.0784	1.2046	4.2746
	single core W cycle	150	173.8975	1.1593	2.8072
10000	multicore	464	130.5256	0.2813	4.0989
	single core	494	136.6465	0.2766	2.7809
	multicore W cycle	150	175.3394	1.1689	4.2566
	single core W cycle	136	160.5415	1.1805	2.6837

Table 4.1. Details for Figure 4.2, scaling comparison on Cray XT4. Iterations, solve time, time per iteration, and setup time.

```
<ParameterList name="MultiLevelPreconditioner">

  <!-- General options -->
  <Parameter name="SetDefaults" type="string" value="SA"/>
  <Parameter name="ML output" type="int" value="10"/>
  <Parameter name="max levels" type="int" value="10"/>

  <!-- Smoother options -->

     <!-- all levels unless otherwise specified (polynomial) -->
     <Parameter name="smoother: type" type="string" value="
        Chebyshev"/>

     <!-- level 1 only (multicore aware) -->
     <Parameter name="smoother: type (level 1)" type="string"
        value="self"/>
     <Parameter name="smoother: sweeps (level 1)" type="int"
        value="1"/>
     <Parameter name="smoother: pre or post (level 1)"
        type="string" value="pre"/>
     <Parameter name="smoother: self overlap (level 1)"
        type="int" value="2"/>

     <!-- subdomain solve options (use AMG) -->
     <ParameterList name="smoother: self list">
       <Parameter name="SetDefaults" type="string" value="SA"/>
       <Parameter name="smoother: type" type="string"
          value= "Chebyshev"/>
       <Parameter name="ML output" type="int" value="10"/>
     </ParameterList>

</ParameterList>
```

Figure 4.3. XML input for invoking multicore-aware smoothing

Chapter 5

Interfaces to HPC Libraries

5.1 Introduction

In the course of improving parallel AMG performance, we investigated some third-party high-performance libraries whose capabilities complement those in TRILINOS. In this chapter we discuss our experiences with these third-party high performance libraries and in some cases present numerical results to demonstrate their performance within TRILINOS. We developed interfaces to the Optimized Sparse Kernel Interface (OSKI) and Hierarchical Iterative Parallel Solver (HIPS). Our interest in OSKI is to improve single-processor sparse matrix kernel performance, and in HIPS to improve AMG multicore performance via true parallel incomplete factorization smoothers.

5.2 OSKI

In this section, we discuss a new interface within the TRILINOS package EPETRA [40] to the Optimized Sparse Kernel Interface (OSKI) [53, 3, 53], and assess OSKI's impact on TRILINOS computations. EPETRA is a foundational package within TRILINOS that provides fundamental classes and methods for serial and parallel linear algebra, e.g., point and block matrices, multivectors, and graphs. All solver packages within Trilinos can use EPETRA kernels as building blocks for both serial and parallel algorithms. Therefore, making improving EPETRA's single processor speed will improve the performance and efficiency of other packages that depend on it. The new EPETRA/OSKI interface enables Trilinos and application developers to leverage the highly tuned kernels provided by OSKI in a standardized manner.

In Section 5.2.1, we give an overview of the design and features of the OSKI package itself. In Section 5.2.2, we discuss the design of the EPETRA interface to OSKI. In Section 5.2.3, we discuss the results of performance tests run on the OSKI kernels within EPETRA. Tests were run on individual OSKI kernels, and include small scaling studies. In Section 5.2.6, conclusions of the work and results described in this section are presented. In Section 5.2.7, ways to add more functionality to our implementation, and suggestions of things to test in new OSKI releases are presented.

5.2.1 OSKI High Level Overview

OSKI is a package used to perform optimized sparse matrix-vector operations. It provides both a statically tuned library created upon installation and dynamically tuned routines created at runtime. OSKI provides support for single and double precision values of both real and complex types, along with indexing using both integer and long types. When possible it follows the sparse BLAS standard [16] as closely as possible in defining operations and functions.

Before a matrix can use OSKI functionality, it first must be converted to the matrix type `oski_matrix_t`. To store a matrix as an `oski_matrix_t` object, a create function must be called on a CSR or CSC matrix. An `oski_matrix_t` object can either be created using a deep or shallow copy of the matrix. When a shallow copy is created, the user must only make changes to the matrix's structure through the OSKI interface. When a deep copy is created, the matrix that was passed in can be edited by the user as desired. OSKI automatically makes a deep copy when any matrix is tuned in a manner that changes its structure.

Routine	Calculation
Matrix-Vector Multiply	$y = \alpha Ax + \beta y$ or $y = \alpha A^T x + \beta y$
Triangular Solve	$x = \alpha A^{-1}x$ or $x = \alpha A^{T^{-1}}x$
Matrix Transpose Matrix-Vector Multiply	$y = \alpha A^T Ax + \beta y$ or $y = \alpha AA^T x + \beta y$
Matrix Power Vector Multiply	$y = \alpha A^p x + \beta y$ or $y = \alpha A^{T^p} x + \beta y$
Matrix-Vector Multiply and Matrix Transpose Vector Multiply	$y = \alpha Ax + \beta y$ and $z = \omega Aw + \zeta z$ or $z = \omega A^T w + \zeta z$

Table 5.1. Computational kernels from OSKI available in EPETRA.

OSKI provides five matrix-vector operations to the user. The operations are shown in Table 5.1. Hermitian operations are available in OSKI, but are not shown in the table since EPETRA does not include Hermitian functionality. The last three kernels are composed operations using loop fusion [21] to increase data reuse. To further improve performance, OSKI can link to a highly tuned BLAS library.

OSKI creates optimized routines for the target machine's hardware based on empirical search, in the same manner as ATLAS [54] and PHiPAC [4]. The goal of the search is create efficient static kernels to perform the operations listed in Table 5.1. The static kernels then become the defaults that are called by OSKI when runtime tuning is not used. Static tuning can create efficient kernels for a given data structure. To use the most efficient kernel, the matrix data structure may need to be reorganized.

When an operation is called enough times to amortize the cost of rearranging the data structure, runtime tuning can be more profitable than using statically tuned functions. OSKI provides multiple ways to invoke runtime tuning, along with multiple levels of tuning. A user can explicitly ask for a matrix to always be tuned for a specific kernel by selecting either the moderate or aggressive tuning option. If the user wishes for OSKI to decide whether enough calls to a function occur to justify tuning, hints can be used. Possible hints include telling OSKI the number of calls expected to the routine and information about the matrix, such as block structure or symmetry. In either case, OSKI tunes the matrix either according to the user's requested tuning level, or whether it expects to be able to amortize the cost of tuning if hints are provided. Instead of providing hints the user may, periodically call the tune function. In this case, the tune function predicts the number of future kernel calls based on past history, and tunes the routine only if it expects the tuning cost to be recovered via future routine calls.

OSKI can also save tuning transformations for later reuse. Thus, the cost of tuning searches can be amortized over future runs. Specifically, a search for the best tuning options does not need to be run again, and only the prescribed transformations need to be applied.

OSKI is under active development. As of this writing, the current version is 1.0.1h, with a multi-core version under development [52]. While OSKI provides many optimized sparse matrix kernels, some features have yet to be implemented, and certain optimizations are missing. OSKI is lacking multi-vector kernels and stock versions of the composed kernels. These would greatly add to both OSKI's usability and performance. The Matrix Power Vector Multiply is not functional. Finally, OSKI cannot transform (nearly) symmetric matrices to reduce storage or convert from a CSR to a CSC matrix (or vice versa). Both could provide significant memory savings. Thus, performance gains from runtime tuning should not be expected for point matrices. An exception is pseudo-random matrices, which may benefit from cache blocking.

5.2.2 Design and Implementation

In the design and implementation of the EPETRA OSKI interface the EPETRA coding guidelines [29] were followed as closely as possible. In doing so, we ensured the consistency of our code with the existing EPETRA code base, as well as its readability and maintainability. Finally, the EPETRA interface to OSKI will likely be ported to Kokkos [41], and the interface's design will make this process easier.

In the design phase we focused on allowing the greatest amount of flexibility to the user, and exposing as much of the functionality of OSKI as possible. In some places, however, OSKI functionality is not exposed because there is not a corresponding EPETRA function. For example, OSKI has a function that allows changing a single value in a matrix, but EPETRA does not. When two copies of a matrix exist, as when the OSKI constructor makes a deep copy of the underlying data, the corresponding EPETRA copy is guaranteed to contain the same data. Since EPETRA can only change data values one row at a time, a point set function is not included in the OSKI interface. Instead, we include a function to change a row of data within OSKI by overloading the EPETRA function to change row data. When a single copy of the data exists, the EPETRA function is called on the matrix. When both an OSKI and EPETRA matrix exist, both the matrix copies are modified to keep the data consistent. The EPETRA function is called once for the EPETRA version of the matrix, and the OSKI matrix has its point function called once for each entry in the row.

When there are clear equivalent functions in OSKI and EPETRA, the OSKI function is designed to overload the EPETRA function. In the cases where OSKI provides more functionality than EPETRA, the interface is designed with two functions to perform the operation. The first function mimics EPETRA's functionality and passes values that eliminate the extra functionality from OSKI. The second function exposes the full functionality OSKI provides. Also, as appropriate new functions are added that are specific to OSKI, such as the tuning functions. Conversely, EPETRA functions without any analogue in the OSKI context are not overloaded in the `Epetra_Oski` namespace.

The interface is also designed to maintain robustness and ease of use. All `Epetra_OskiMatrix` functions that take in vectors or multi-vectors allow for the input of both `Epetra_Vector` or `Epetra_MultiVector` objects, and `Epetra_OskiVector` or `Epetra_OskiMultiVector` objects. The objects are converted to the proper types as necessary through the use of the lightest weight wrapper or converter possible.

The implementation follows the idea of wrapping and converting data structures in as lightweight a fashion as possible, to maximize speed and minimize space used. In addition, the implementation provides the user with as much flexibility as possible. For example, the user can specify as many tuning hints as they like. Alternatively, the user can ask EPETRAto figure out as much as it can about the matrix and pass along those hints to OSKI. Both options can be combined, with user-specified hints taking precedence over automatically generated hints. Options are passed by the user via Teuchos parameter lists [42].

Class	Function
`Epetra_OskiMatrix`	Derived from `Epetra_CrsMatrix`. Provides all OSKI matrix operations.
`Epetra_OskiMultiVector`	Derived from `Epetra_MultiVector`. Provides all OSKI multi-vector operations.
`Epetra_OskiVector`	Derived from `Epetra_OskiMultiVector`. Provides all OSKI vector operations.
`Epetra_OskiPermutation`	Stores permutations and provides Permutation functions not performed on a `Epetra_OskiMatrix`.
`Epetra_OskiError`	Provides access to OSKI error handling functions and the ability to change the default OSKI error handler.
`Epetra_OskiUtils`	Provides the initialize and finalize routines for OSKI.

Table 5.2. OSKI classes within EPETRA.

Finally, the design is broken into six separate classes. Table 5.2 shows the classes and provides information about which classes each derives from, and what functions each contains. The design is as modular as possible to allow for the easy addition of new functions, and to logically group related functions together.

5.2.3 Results

To assess the potential benefit of using OSKI in Sandia applications, we ran tests on representative data and a variety of advanced architectures. For these tests OSKI version 1.0.1h was used. OSKI runtimes were compared to the runtimes of the currently used EPETRA algorithms, in both serial and parallel. In this section, we first present our test environment and methodology, and then present the results of performance tests run comparing EPETRA to OSKI.

5.2.4 Test Environment and Methodology

Performance tests were run on two different machine architectures in serial and parallel. The first test machine has two Intel Clovertown processors. The second test machine has one Sun Niagara-2 processor. Machine specifications and compilers are shown in Table 5.3. On each machine, Trilinos was compiled with widely used optimizations levels, and OSKI was allowed to pick the best optimization flags itself.

processor	#chips	cores	threads	frequency	L2 cache	compiler
Clovertown	2	8	8	1.87 Ghz	4 M per 2 cores	Intel
Niagara-2	1	8	64	1.4 Ghz	4 M per core	Sun

Table 5.3. Test machines used for performance testing.

These machines were chosen for their diversity and potential for use at Sandia. The Clovertown is one of Intel's latest processors, and the Niagara is an example of an extremely parallel chip.

On each machine, tests were run on three matrices arising from Sandia applications. The first matrix is from a finite element discretization within a magnetics simulation. The second is a block-structured Poisson matrix. The third matrix is unstructured and represents term-document connectivity. The data is from the Citeseer application. Table 5.4 gives some matrix properties. Each matrix was able to fit within the main memory of each test machine. These matrices were

matrix	rows	columns	nnz	structure
point	556356	556356	17185984	nearly symmetric point
block	174246	174246	13300445	symmetric 3 by 3 blocks
Citeseer	607159	716770	57260599	unstructured point

Table 5.4. Test machines for EPETRA OSKI performance testing.

also used in a scaling study. Tests were run up to the total number of available threads that can be executed simultaneously, on each machine.

5.2.5 Performance Test Results

The serial results for each machine are shown in Figures 5.1 and 5.2 for four OSKI kernels: Ax, A^Tx, A^TAx, and the two-vector multiplication $y = Ax$; $z = Aw$. The last operation is henceforth referred to as "2Mult". In addition, Table 5.5 shows the speeds of EPETRA calculations as a baseline. Since OSKI has no atomic versions of the composed kernels, the OSKI stock numbers represent two separate matrix-vector multiply calls to OSKI. There is potential that the tuned composed kernels are not performing optimally due to tuning to a non-ideal data structure, as is seen in the tuning cost data later. Results for the matrix power kernel are unavailable due to a bug in the kernel. Also results for the AA^T kernel were excluded because EPETRA only stores matrices in CSR. OSKI cannot convert CSR to CSC, which is needed to take advantage of these kernels in serial. Finally, the direct solve kernel was not profiled, as it is not critical to many Sandia applications.

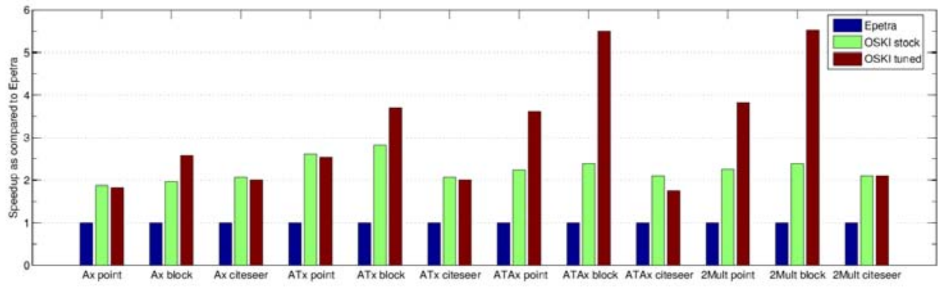


Figure 5.1. Relative performance of EPETRA and OSKI in serial on Clovertown.

Machine	Ax	A^Tx	A^TA	2Mult
Clovertown	220/227/55	150/154/43	178/183/48	178/184/48
Niagara	58.3/69.9/20.7	56/66.4/20.3	57.1/68.1/20.5	57.1/68.1/20.5

Table 5.5. EPETRA serial routine speeds in Mflops. Results are in the form point/block/Citeseer.

On the Clovertown, OSKI produced large speedups over EPETRA for all matrices in serial, as shown in Figure 5.1. The stock kernels demonstrated speedups of 1.8 to 2.8. Tuning improved the block matrices by about one third when compared to the stock kernels. The composed algorithms demonstrated even more significant speedups of up to 5.5, when composing and blocking were combined. Tuning did not improve the runtime of point matrices, except when a composed kernel was used. In the case of the Citeseer matrix, a composed kernel resulted in either no performance gain or performance degradation.

Figure 5.2 shows that on the Niagara, the stock OSKI and EPETRA kernels had roughly the same performance Tuning for point matrices once again resulted in either no gains or slight losses. Tuning for block matrices resulted in a one third to one half gain in speed. Again, composing increased the speed of all kernels significantly, except for the Citeseer matrix, for which the OSKI kernels where actually slower.

As expected, the serial tests show that the tuning of point matrices is counterproductive, except when needed to use composed kernels. However, tuning of block matrices results in significant

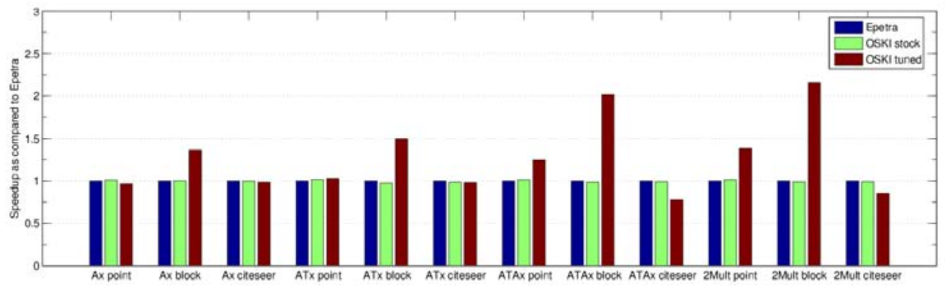


Figure 5.2. Relative performance of EPETRA and OSKI in serial on Niagara.

speedups through the reduction of indirect addressing. For the pseudo random Citeseer matrix, tuning is never beneficial. This is probably due to either lack of cache-blocking in the composed kernels and/or more random access, which create a greater number of cache misses. For structured matrices, composing results in a 25% to 60% gain over the faster of the stock and tuned kernels.

Even if the tuning gains shown above are large, the amount of time it takes to tune a matrix at runtime is important in determining whether tuning will result in performance gains. Tables 5.6, 5.7 and 5.8 show the cost of tuning and the number of matrix-vector calls needed to amortize that cost for the point, block, and Citeseer matrices, respectively. The tuning and retuning costs are expressed in terms of the number of matrix-vector multiplies that could be performed in the time it takes to tune. *Tuning cost* is the amount of time it takes to tune a matrix the first time, and includes time to analyze the matrix to determine what optimizations are beneficial. *Retuning cost* is the amount of time it takes to tune the matrix if the optimizations to be performed are already known. All comparisons are to the faster of the EPETRA and OSKI matrix-vector multiplies. The amortize columns show the number of calls to the tuned kernel needed to realize tuning gains. When N/A is listed in an amortize column, it is never better to tune because the tuned kernels are no faster than the untuned kernels. We note that the tuning cost depends only on the matrix structure, not on the matrix kernel to be performed.

Machine	Tune/Retune	Amortize Ax/Retune	Amortize A^TA/Retune	Amortize 2Mult/Retune
Clovertown	37.6 / 20.1	N/A	48 / 26	45 / 24
Niagara	22.1 / 12.7	N/A	56 / 33	40 / 24

Table 5.6. OSKI tuning costs for point matrix. Cost is equivalent number of matrix-vector multiplications.

Machine	Tune/Retune	Amortize Ax/Retune	Amortize A^TA/Retune	Amortize 2Mult/Retune
Clovertown	31.1 / 17.7	131 / 75	27 / 16	28 / 16
Niagara	22.5 / 14.1	86 / 54	22 / 14	21 / 13

Table 5.7. OSKI tuning costs for block matrix. Cost is equivalent number of matrix-vector multiplications.

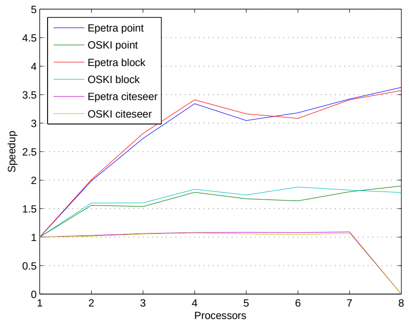
In many cases, the tuned OSKI kernels are much more efficient than the EPETRA and OSKI stock kernels. However, the data structure rearrangement required to create an OSKI kernel is non-

Machine	Tune/Retune	Amortize Ax/Retune	Amortize $A^T A$/Retune	Amortize 2Mult/Retune
Clovertown	14.5 / 6.7	N/A	N/A	N/A
Niagara	11.5 / 5.2	N/A	N/A	N/A

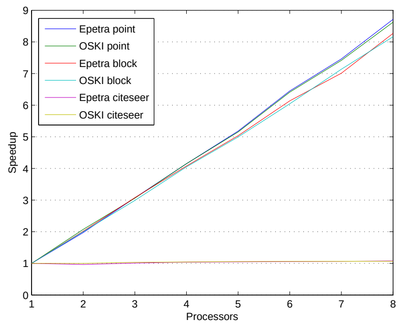
Table 5.8. OSKI tuning costs for Citeseer matrix. Cost is equivalent number of matrix-vector multiplications.

trivial. The cost of tunings ranges from 11.5 to 37.6 equivalent matrix-vector multiplies. It can require as many as 131 subsequent kernel applications to recoup the cost of initial tuning. However, re-tuning costs are usually slightly over half the cost of the initial tuning, so saving transformations for later use could be profitable. Block matrices require the smallest number of calls to recover tuning costs, and when combined with composed kernels, this number drops even more. For point matrices tuning the matrix-vector multiply is never profitable, but the tuning of composed kernels can be profitable for structured matrices.

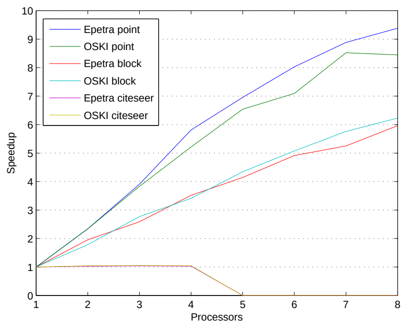
While serial performance is important to application performance, most scientific simulations are run on parallel machines. The first level of parallelism is within a single node, which typically contains one or two multicore processors. To test the scalability of our implementation of OSKI, within EPETRA, we ran tests on each matrix on 1 to 8 cores of each machine and also on 1 to 8 threads per core on the Niagara.



(a) Clovertown



(b) Niagara



(c) Niagara multi-threaded

Figure 5.3. OSKI matrix-vector multiply strong scaling results.

Figures 5.3(a)-5.3(c) show the strong scaling of the matrix-vector kernel for each matrix. Figure 5.3(a) shows that on the Clovertown that EPETRA has better scaling than OSKI. Table 5.9 shows, however, that the overall performance of OSKI is either comparable or better to that of EPETRA. The better scaling for EPETRA comes from its slower performance in the single processor case, which allows for more improvement within a limited memory bandwidth situation. For the point matrix, both EPETRA and OSKI improve significantly until each is running at about 735 Mflops on 4 cores. At this point, the calculations likely become memory bandwidth limited. With added processing power, the speeds then improve to slightly under 800 Mflops. The block matrix results show a similar pattern, with the OSKI block matrix remaining more efficient throughout. The Citeseer matrix does not scale most likely due to the large amounts of data it needs to exchange, because its unstructured. Also it could not be run on 8 processors due to an increasing memory footprint, perhaps due to exchanged data.

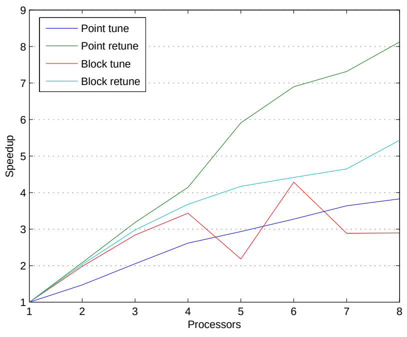
machine	point EPETRA/OSKI	block EPETRA/OSKI	Citeseer EPETRA/OSKI
Clovertown	798/782	810/1099	59.6/122
Niagara 1 thread/core	508/507	578/778	22.3/22.0
Niagara multiple threads/core	4767/4321	3447/4847	23.2/23.2

Table 5.9. EPETRA and OSKI maximum parallel matrix vector multiply speeds in Mflops.

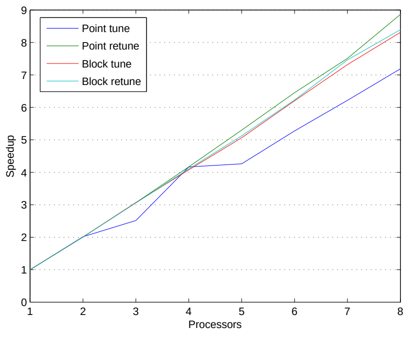
Figure 5.3(b) shows that on the Niagara both the point and block matrix algorithms scale linearly with the number of cores. Essentially, there is enough memory bandwidth to feed each core. As seen in Figure 5.3(c), adding more threads per core to the calculating power leads to approximately linear speedup for all matrices. This begins to tail off at 5 threads for block matrices, and 7 threads for point matrices. The Citeseer matrix once again does not scale and becomes too large to run above 32 threads.

Scalability also matters when a matrix is being tuned. Figures 5.4(a)-5.4(c) show how well each matrix scales on each machine in terms of tuning cost. Scaling is usually linear or slightly better with the number of processors. This result is expected as tuning is a local computation with no communication between processors. As seen in Figure 5.4(c), increasing the number of threads per Niagara processor initially leads to improved performance, before dropping off at 6 or more threads per processor. The dropoff is most likely due to threads competing for processor resources. Results for the Citeseer matrix were not shown, as OSKI does not tune its matrix-vector multiply kernel for the Citeseer matrix. Finally, note that the retune function demonstrates better scaling than the same tune function in all cases.

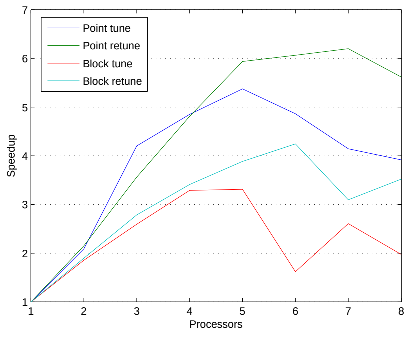
In addition to strong scaling tests, we also ran a weak scaling test on the Niagara. We used the block matrix from the 8 thread test case in Table 5.4. Tests were run on 1, 8, 27 and 64 threads. Results are shown in Figures 5.5(a)-5.5(c). As seen in Figure 5.5(a), the OSKI tuned and untuned matrix-vector multiplies both scale similarly to EPETRA's matrix-vector multiply. Figure 5.5(b), shows that the tuned composed kernels do not scale well. The same result was seen for the untuned composed kernels. For these operations to be possible there is extra data copying in the wrapping of the serial kernels, which could be the problem. There could also be inefficiencies in the code



(a) Clovertown



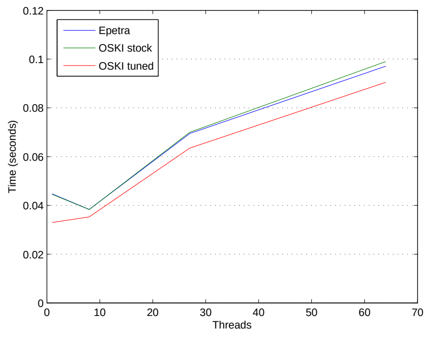
(b) Niagara single-threaded



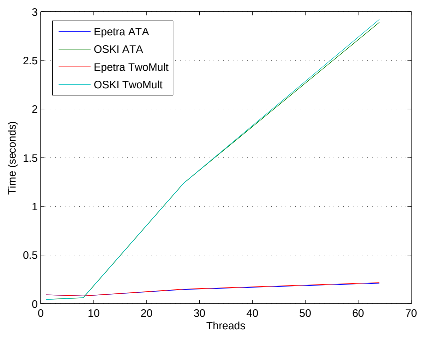
(c) Niagara multi-threaded

Figure 5.4. Scalability of OSKI tuning.

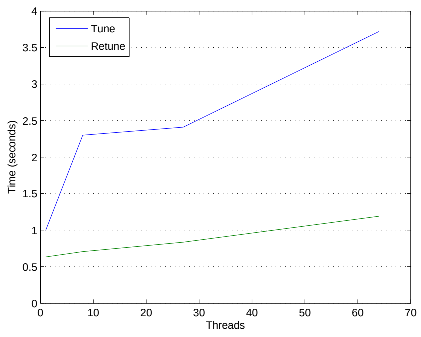
in other places or resource contention on the processor. Figure 5.5(c) shows that re-tuning scales better than tuning as the problem size grows.



(a) MatVec



(b) Composed



(c) Tuning

Figure 5.5. Weak scalability of OSKI on Niagara

5.2.6 Conclusions

Overall, OSKI can produce large speedups in sparse matrix computational kernels. This is especially true when the matrix is block structured or multiple multiplications are performed using the same matrix. In some cases it can also produce large gains for matrix-vector multiplies involving only a single matrix. However, OSKI is still missing some features, such as a multi-vector kernel and the ability to tune matrices to make them symmetric. Both could produce large runtime gains. Our EPETRA/OSKI interface has stubs to allow the use of these missing features as soon as they become available in OSKI. Our experiments show that Sandia applications that make heavy use certain sparse matrix kernels can benefit from the current version of OSKI. As new OSKI features become available, its potential impact on other Sandia applications should increase.

5.2.7 Future Work

For the current (1.0.1h) version of OSKI, a developer may want to implement the solve function and run more weak scalability or other parallel tests to determine why the composed kernels do not scale well. For a newer version of OSKI, a developer may want to test any new tuning features, the matrix power kernel, as well as any other new functions. Finally, we recommend any new version of OSKI be tested on the Barcelona and Xeon chips, as we were never able to successfully install OSKI on these architectures. The Barcelona is of particular interest, as it is the processor found in the center section of Red Storm.

5.3 HIPS

In this section we discuss a new interface in TRILINOS/IFPACK to a subset of the functionality of the HIPS library (formerly PHIDAL) developed at INRIA Bordeaux - Sud Ouest and made available through the LGPL-compatible CeCILL-C license. [19]. The HIPS functionality we have made available through IFPACK is focused predominantly on their multistage parallel incomplete factorization routines. Prior to the incorporation of an interface to HIPS, IFPACK's incomplete factorization methods would disregard off-processor connections. With HIPS accessible through IFPACK, such dropping is no longer required.

5.3.1 HIPS Overview

The core HIPS functionality made available through IFPACK is a multistage ILUT algorithm [28]. The core insight behind HIPS is to exploit a "hierarchical" graph decomposition in a fashion inspired by cross points in domain decomposition. The algorithm begins with a graph partitioning (with overlap 1) by either METIS or SCOTCH [30, 38]. Interfaces to parallel graph partitioners like ParMETIS or PT-SCOTCH [31, 37] have not been integrated into HIPS as of the writing of this document, so the initial partitioning occurs in serial. The overlap of this initial partitioning serves as a separator for the lower levels and the algorithm continues recursively. Computational experiments with PHIDAL, HIPS' predecessor, have yielded reasonable scalability up to 256 processors [28].

5.3.2 IFPACK-HIPS Interface

Like many other preconditioners in IFPACK, `Ifpack_HIPS` inherits from the `Ifpack_Preconditioner` class. It takes a `Epetra_RowMatrix` in the constructor and is driven by a TEUCHOS `ParameterList` which exposes a select subset of the HIPS functionality to IFPACK users. Specifically, users can control drop tolerances, optimizations for symmetry, output levels and special treatment for matrices with multiple dofs per node. The only complication is that the user must call the proper initialization (and cleanup) routines for HIPS outside of the `Ifpack_HIPS` class, since HIPS uses it's own pre-allocated static storage to store it's internal data, and assigns each instance of HIPS an id handle to access the HIPS storage. Since users may wish to have multiple copies of HIPS set up at any point in time, such allocations and cleanup cannot be handled in the `Ifpack_HIPS` class.

5.3.3 Experiments

We consider the same model convection-diffusion problem discussed in Section 4.0.4. Again we consider an $\varepsilon = 10^{-8}$ convection parameter and test on the 16 core per node "glory" Linux cluster. For testing purposes we using non-symmetric smoothed aggregation (NSSA) multigrid with IFPACK's ILU(0) as a smoother on all levels except for level 1 and the coarsest level. On level 1 we use a domain-decomposition smoother and the on the coarsest level we use LU. As previously, we study the problem in a weak scaling sense.

Figure 5.10 shows results where we do a single pre-smoothing sweep on each level of the hierarchy. We compare using a single smoothing sweep of domain-decomposed ILU(0) from IFPACK, with a single smoothing sweep of domain-decomposed HIPS (with a drop tolerance of $5e-3$) on level 1. Since HIPS does not drop off-core entries, we have each *node* serve as a additive Schwarz domain. As for the ILU(0), we have each *code* serve as an additive Schwarz domain. From these results we can see quite clearly that while on a single node these methods are comparable, the node-level HIPS smoothing converges on larger problems, while the ILU(0) does not.

	Number of Glory Nodes							
Method	1	4	9	16	25	36	49	64
HIPS(node)	53	80	112	150	181	229	268	301
ILU	52	$*$	$*$	$*$	$*$	$*$	$*$	$*$

Table 5.10. GMRES iterations for NSSA using one smoothing sweep with additive Schwarz (either node-level HIPS or IFPACK's ILU(0)) on level 1 for a convection-diffusion problem on an orthogonal grid with diffusion coefficient $\varepsilon = 1e-8$. Asterisks ($*$) indicate inability to converge in less than 500 iterations.

Figure 5.11 shows similar results to Figure 5.10, only we do two smoothing sweep at each level. With respect to level 1, we do one sweep of additive Schwarz with two HIPS/ILU(0) sweeps inside the subdomain. Again we note that the HIPS performance is better than IFPACK's ILU(0) in terms of iterations, but as Figure 5.12 shows, HIPS is usually slower than IFPACK on smaller problems. The greater scalability of HIPS shows itself after about 16 cores, where the ILU(0) method starts to degrade in performance rapidly.

5.3.4 Conclusions

In summary, the implementation of a HIPS interface in IFPACK has provided an important new feature to the TRILINOS, namely the ability to do parallel incomplete factorizations without dropping

	Number of Glory Nodes							
Method	1	4	9	16	25	36	49	64
HIPS(node)	28	53	76	94	125	142	160	179
ILU	32	57	84	117	189	228	$*$	$*$

Table 5.11. GMRES iterations for NSSA using two smoothing sweeps with additive Schwarz (either node-level HIPS or IFPACK's ILU(0)) on level 1 for a convection-diffusion problem on an orthogonal grid with diffusion coefficient $\varepsilon = 1e-8$. Asterisks ($*$) indicate inability to converge in less than 500 iterations.

	Number of Glory Nodes							
Method	1	4	9	16	25	36	49	64
HIPS(node)	2.0	3.9	7.8	9.8	15.8	21.3	24.2	32.2
ILU	1.3	2.4	5.8	8.3	18.1	26.4	$*$	$*$

Table 5.12. Solve times in seconds for NSSA using two smoothing sweeps with additive Schwarz (either node-level HIPS or IFPACK's ILU(0)) on level 1 for a convection-diffusion problem on an orthogonal grid with diffusion coefficient $\varepsilon = 1e-8$. Asterisks ($*$) indicate inability to converge in less than 500 iterations.

off-processor entries. HIPS is also a natural match for the domain decomposition-based node-level parallelism discussed in Chapter 4, as incomplete factorizations tend to perform well in modestly parallel conditions. We have also demonstrated that such a preconditioner can be more effective than a similar technique based on IFPACK's ILU(0) routine for particularly challenging convection-diffusion problems.

Chapter 6

Conclusion

In this report we have summarized the results of our LDRD, ”Highly Scalable Linear Solvers on Thousands of Processors.” We have improved existing multigrid methods as well as developed new algorithmic capabilities. We have made significant improvements in compute intensive kernels used during the AMG setup. Numerical results show that for block structured matrices these kernels can result in significant speedups over the existing production kernels. We have extended a geometric two-dimensional multigrid algorithm to a two- and three-dimensional algebraic method. We have shown how this algorithm can be interpreted in a domain-decomposition context. Finally, we have developed a smoother for multicore architectures, deployed this smoother in the TRILINOS framework, and shown that it can be effective on large-scale problems.

References

[1] V. Bandy, J. Dendy, and W. Spangenberg. Some multigrid algorithms for elliptic problems on data parallel machines. *SIAM J. Sci. Stat. Comp.*, 19(1):74–86, 1998.

[2] P. Bastian, W. Hackbusch, and G. Wittum. Additive and multiplicative multi-grid – a comparison. *Computing*, 60:345–364, 1998.

[3] Berkeley Benchmarking and Optimization Group. OSKI: Optimized Sparse Kernel Interface. http://bebop.cs.berkeley.edu/oski/about.html, May 2008.

[4] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM.

[5] J. Bramble, J. Pasciak, and J. Xu. Parallel multilevel preconditioners. *Math. Comp.*, 55:1–22, 1990.

[6] A. Brandt and B. Diskin. Multigrid solvers on decomposed domains. In *Domain Decomposition Methods in Science and Engineering: The Sixth International Conference on Domain Decomposition*, volume 157 of *Contemporary Mathematics*, pages 135–155, Providence, Rhode Island, 1994. American Mathematical Society.

[7] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial.* SIAM Books, Philadelphia, 2000. Second edition.

[8] S. Carney, M. Heroux, and G. Li. A proposal for a sparse BLAS toolkit. Technical report, Technical report, Cray Research Inc., Eagen, MN, 1993.

[9] T. Chan and R. Tuminaro. Analysis of a parallel multigrid algorithm. In S. McCormick, editor, *Proceedings of the Fourth Copper Mountain Conference on Multigrid Methods*, NY, 1987. Marcel Dekker.

[10] E. Chow, R.D. Falgout, J.J. Hu, R.S. Tuminaro, and U.M. Yang. A survey of parallelization techniques for multigrid solvers. *Parallel Processing for Scientific Computing*, pages 179–195.

[11] E. Chow and P. S. Vassilevski. Multilevel block factorizations in generalized hierarachical bases. 10:105–127, 2003.

[12] J. Dendy. Revenge of the semicoarsening frequency decomposition method. *SIAM J. Sci. Stat. Comp.*, 18:430–440, 1997.

[13] J. Dendy and C. Tazartes. Grandchild of the frequency decomposition method. *SIAM J. Sci. Stat. Comp.*, 16:307–319, 1995.

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990. Available from: `http://doi.acm.org/10.1145/77626.79170`.

[15] C. Douglas and W. Miranker. Constructive interference in parallel algorithms. *SIAM Journal on Numerical Analysis*, 25:376–398, 1988.

[16] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2), June 2002.

[17] L. Fournier and S. Lanteri. Multiplicative and additive parallel multigrid algorithms for the acceleration of compressible flow computations on unstructured meshes. *Applied Numerical Mathematics*, 36(4):401–426, 2001.

[18] P. Frederickson and O. McBryan. Parallel superconvergent multigrid. In S. McCormick, editor, *Proceedings of the Third Copper Mountain Conference on Multigrid Methods*, pages 195–210, NY, 1987. Marcel Dekker.

[19] J. Gaidamour, P. Hénon, and Y. Saad. Hips user's guide. Available from: `http://hips.gforge.inria.fr`.

[20] D. B. Gannon and J. R. van Rosendale. On the structure of parallelism in a highly concurrent PDE solver. *Journal of Parallel and Distributed Computing*, 3(1):106–135, 1986.

[21] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *1992 Workshop on Languages and Compilers for Parallel Computing*, number 757, pages 281–295, New Haven, Conn., 1992. Berlin: Springer Verlag.

[22] M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, M. G. Sala, and I. Karlin. ML developer's guide. 2007.

[23] M.W. Gee, C.M. Siefert, J.J. Hu, R.S. Tuminaro, and M.G. Sala. ML 5.0 smoothed aggregation user's guide. (SAND2006-2649), 2006.

[24] A. Greenbaum. A multigrid method for multiprocessors. In S. McCormick, editor, *Proceedings of the Second Copper Mountain Conference on Multigrid Methods*, volume 19 of *Appl. Math and Computation*, pages 75–88, 1986.

[25] W. Hackbusch. *Multigrid Methods and Applications*, volume 4 of *Computational Mathematics*. Springer–Verlag, Berlin, 1985.

[26] W. Hackbusch. A new approach to robust multi-grid methods. In *First International Conference on Industrial and Applied Mathematics*, Paris, 1987.

[27] W. Hackbusch. The frequency decomposition multigrid method, part I: Application to anisotropic equaitons. *Numer. Math.*, 56:229–245, 1989.

[28] P. Hénon and Y. Saad. A parallel multilevel ILU factorization based on hioerarchical graph decomposition. *SIAM J. Sci. Comput.*, 28:2266–2293, 2006.

[29] M. A. Heroux and P. M. Sexton. Epetra developers coding guidelines. Technical Report SAND2003-4169, Sandia National Laboratories, Albuquerque, NM, December 2003.

[30] G. Karypis and V. Kumar. MeTiS: A softwar epackage for paritioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices, version 4.0, 1998. Available from: `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`.

[31] G. Karypis, K. Schlogel, and V. Kumar. ParMeTiS: Paralle graph partitioning and sparse matrix ordering library: Version 3.1, 2003. Available from: `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

[32] Z. Li, Y. Saad, and M. Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. *Numerical Linear Algebra with Applications*, 10:485–509, 2003.

[33] J. Mandel, M. Brezina, and P. Vaněk. Energy optimization of algebraic multigrid bases. *Computing*, 62:205–228, 1999.

[34] W. F. Mitchell. A parallel multigrid method using the full domain partition. *Elect. Trans. Numer. Anal.*, 6:224–233, 1997.

[35] W. F. Mitchell. Parallel adaptive multilevel methods with full domain partitions. *App. Num. Anal. and Comp. Math.*, 1:36–48, 2004.

[36] W. Mulder. A new multigrid approach to convection problems. *J. Comput. Phys.*, 83:303–329, 1989.

[37] F. Pellegrini. PT-SCOTCH 5.1 user's guide. Technical report, LaBRI, September 2008.

[38] F. Pellegrini. SCOTCH 5.1 user's guide. Technical report, LaBRI, September 2008.

[39] M. Sala and M. Heroux. Robust algebraic preconditioners with IFPACK 3.0. (SAND-0662), 2005.

[40] Sandia National Laboratories. Epetra - Home. http://trilinos.sandia.gov/packages/epetra/index.html, May 2008.

[41] Sandia National Laboratories. Kokkos - Home. http://trilinos.sandia.gov/packages/kokkos/index.html, May 2008.

[42] Sandia National Laboratories. Teuchos - Home. http://trilinos.sandia.gov/packages/teuchos, May 2008.

[43] B. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[44] J. Swisshelm, G. Johnson, and S. Kumar. Parallel computation of Euler and Navier-Stokes flows. In S. McCormick, editor, *Proceedings of the Second Copper Mountain Conference on Multigrid Methods*, volume 19 of *Appl. Math. and Computation*, pages 321–331, 1986.

[45] A. Toselli and O. Widlund. *Domain Decomposition Methods — Algorithms and Theory*. Springer, New York, 2005.

[46] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid.* Academic Press, London, 2001.

[47] R. Tuminaro. A highly parallel multigrid-like algorithm for the Euler equations. *SIAM J. Sci. Comput.*, 13(1), 1992.

[48] R. Tuminaro and C. Tong. Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines. In J. Donnelley, editor, *SuperComputing 2000 Proceedings*, 2000.

[49] R. S. Tuminaro, M. A. Heroux, S. A. Hutchinson, and J. N. Shadid. Official aztec user's guide: Version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, Albuquerque, NM, 1999.

[50] P. Vaněk, M. Brezina, and J. Mandel. Convergence of algebraic multigrid based on smoothed aggregation. *Numer. Math.*, 88(3):559–579, 2001.

[51] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid based on smoothed aggregation for second and fourth order problems. *Computing*, 56:179–196, 1996.

[52] R. Vuduc. Personal Communication, July 2008.

[53] R. Vuduc, J. W. Demmel, and K. A. Yelick. The Optimized Sparse Kernel Interface (OSKI) library user's guide for version 1.0.1h. Technical report, University of California at Berkeley, Berkeley, CA, June 2007.

[54] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. *Procceding of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Nov 1998.

[55] S. Xiao and D. Young. Multiple coarse grid multigrid methods for solving elliptic problems. In N. Melson, T. Manteuffel, and S. McCormick C. Douglas, editors, *Proceedings of the Seventh Copper Mountain Conference on Multigrid Methods*, volume 3339 of *NASA Conference Publication*, pages 771–791, 1996.

[56] J. Xu. *Theory of Multilevel Methods*. PhD thesis, Cornell University, 1987.

[57] H. Yserentant. On the multi-level splitting of finite element spaces. *Numer. Math.*, 49:379–412, 1986.

DISTRIBUTION:

1	MS	0123	Donna Chavez, 01011
1	MS	9159	Jonathan Hu, 01416
1	MS	0378	Christopher Siefert, 01431
1	MS	0378	Allen Robinson, 01431
1	MS	0836	Stefan Domino, 01541
1	MS	1320	Scott Collis, 01416
1	MS	1320	Michael Wolf, 01416
1	MS	0899	Technical Library, 9536 (electronic)

Sandia National Laboratories