The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

A CHARACTERIZATION OF SPARSE LINEAR SOLVER PERFORMANCE IN SUBSURFACE FLOW SIMULATIONS

A Thesis in

Computer Science and Engineering

by

Kelly J. Fermoyle

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2011

The thesis of Kelly J. Fermoyle was reviewed and approved* by the following:

Padma Raghavan
Professor of Computer Science and Engineering
Thesis Adviser

Suzanne Shontz
Assistant Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

Jing-Ru Cheng
U.S. Army Corps of Engineers
Special Signatory

*Signatures are on file in the Graduate School.

Abstract

We study the performance of the primary kernel used in the computational simulation of subsurface flow by varying the solver and preconditioner parameters for sparse linear system solution. The subsurface flow problem requires time integration, each step of which uses a nonlinear solver. There are a number of ways to solve the nonlinear equation, one of which is Picard method which forms a symmetric positive-definite (SPD) linear system. Since the majority of computation involves the solution of large sparse linear systems, we evaluate solving systems with iterative preconditioned methods. SPD systems are often solved efficiently by a conjugate gradient (CG) solver using a preconditioner. There are a large number of available preconditioners to use. Two of the most promising preconditioners are incomplete Cholesky factorization (IC) and one formed from an algebraic multigrid (AMG) method. For the IC solver, we try reverse Cuthill-McKee (RCM), minimum degree, and dissection reordering algorithms to improve performance. We also propose the use of thresholding small values from the matrix during the construction of the preconditioner to improve performance. Another avenue to improve performance is the use of mixed-precision algorithms, which we evaluate and propose as future work. Our results indicate that preconditioned CG with IC using RCM reordering can beat the best known performance of basic solvers, but the AMG preconditioner with application knowledge for parameter tuning showed the best overall performance.

Table of Contents

List of Tables

List of Figures

Acknowledgement

Our research was supported in part by NSF Grants CCF-0830679, OCI-0821527, and by the U.S. Army Engineering Research and Development Center through Contract No. W912HZ-09-C0120.

Chapter 1

Introduction

The computational simulation of subsurface flow consists of a difficult set of subproblems that must be solved during the execution of the simulation. The execution time is hence dominated by total solution time of this set [1, 48]. The subproblems are formulated as sparse linear systems that can perhaps be most easily solved using preconditioned methods [23]. The linear systems are iterations of a nonlinear problem that describes the flow of fluids in subsurface environment. There is a large body of work indicating that preconditioned iterative solvers perform well for large sparse linear systems [6, 27, 35, 40, 52, 56, 59]. We examine the performance of preconditioners and reordering methods used throughout the application's execution.

The subsurface flow simulation is a nonlinear problem because of dependencies in the fluids and the space in which it flows [36]. Richards' equation is used in the computation of the finite element modeling. The outer-most loop enables physical time integration using implicit time discretization. At a time step instance, a nonlinear solver is implemented for the resulting nonlinear coefficients. Each of the iterations of the nonlinear system produces a large sparse system of linear equations. The choice of nonlinear solver determines the characteristics of the linear system. Depending on these characteristics, a linear solver is used for this system.

Consider the solution of a linear-system $Ax = b$, where A is a large, sparse, symmetric positive-definite (SPD) matrix. Many solvers are available for the solution of such a system, and are generally categorized as direct [21], iterative [27], and multigrid [9]. Earlier research has shown that no method consistently performs best or is even guaranteed to complete [20, 24, 42, 49]. Direct methods have shown high reliability, but can have much higher memory requirements for sparse systems due to fill-in [19, 21, 57]. Iterative methods can have low memory usage, but may have slow convergence or fail to converge altogether [37]. Preconditioners are frequently used with iterative methods to improve convergence but only expands the choice of solvers [6]. Multigrid solvers can work very well for some problems, but it is difficult to know a priori if the problem is suitable [9]. Additionally, the tuning of certain parameters in algebraic multigrid (AMG) can play a large role in overall performance.

We present strategies to improve performance of subsurface flow simulation. The rest of the paper is organized as follows. In Chapter 2 we present some background information regarding the subsurface flow problem and methods to solve the linear systems. Chapter 3 gives initial experiments using subsurface flow. We present techniques to improve the performance of the entire application in Chapter 4 and analyze results in Chapter 5. Finally, Chapter 6 contains some concluding remarks and discussion.

Chapter 2

Background

This chapter gives a description of the subsurface flow application and how it is solved. We describe the linear system formed from the application. Section 2.2 gives a detailed description of solvers available and the reason for our choice.

2.1 Subsurface Flow

The application we are examining is subsurface flow over a three-dimensional space in simulated time. Richards' equation is the partial differential equation describing flow through a porous medium [34]. It can be derived using Darcy's law with the addition of the continuity equation. Equation 2.1 is Richards' equation as formulated by Paniconi and Putti [46], with pressure head ψ as the dependent variable, time t, and vertical dimension z.

$$\eta(\psi)\frac{\delta\psi}{\delta t} = \nabla \cdot (K_s K_r(\psi)\nabla(\psi + z)) \tag{2.1}$$

The storage coefficient is $\eta(\psi)$ and saturated conductivity K_s is multiplied with relative conductivity K_r for overall hydraulic conductivity.

There are two popular problem formulations that can be used to solve this non-linear system. The first, Picard method, forms a symmetric system and is simpler and faster on a per-iteration basis. It has been shown to fail under certain initial conditions or have poor convergence [36]. The other, Newton method is more complicated and

forms nonsymmetric systems but may converge in cases in which Picard method does not. We will be using the Picard method which was shown by Paniconi and Putti [48] to converge in most cases that Newton does with small back steps and in less simulation time.

2.2 Linear Solver Selection

As stated in Chapter 1, there are a wide variety of techniques to solve a sparse linear system. Because we are using Picard iterations, we will be solving a symmetric system and can perhaps decrease our memory and computational requirements. Common direct methods for solving a large symmetric positive-definite (SPD) are dense triangular factorization (LU) decomposition or Cholesky decomposition. We cannot use geometric multigrid for these systems because geometric information is not available. Therefore the most applicable multigrid method is algebraic multigrid (AMG), which uses only algebraic connections for coarsening. Iterative methods available are conjugate gradient (CG) for symmetric systems and generalized minimum residual (GMRES) for nonsymmetric systems, both can use various preconditioners.

Direct methods have high memory requirements for both LU and Cholesky factorization. Factoring a sparse matrix will frequently result in dense triangular matrices. The memory required to store the factorization scales as $O(n^2)$ for n unknowns. This is because of fill-in that is almost certain to occur during the factorization of a sparse matrix. We have chosen not to solve the linear systems by direct methods because the size of the systems will result in too much memory pressure.

AMG uses a technique of coarsening the problem to a smaller representation, solving this problem, and then refining the solution to get back to the initial problem. This is similar to geometric multigrid, except that no information of grid or geometry are assumed or used. Instead, AMG relies on the strength of the connection between a coarse variable and the fine variable. The connection between the fine and coarse levels are determined by the algebraic connections as described in the matrix. It has been shown that the success of the AMG depends heavily on the geometry from which the system was created [9]. Our application does come from a geometric domain, so such a solver may work well and we will evaluate it. AMG can also be used in conjunction with iterative methods as a preconditioner, which is the approach this paper takes.

CG and GMRES are commonly used iterative solvers for large symmetric systems. CG requires that the system be symmetric, while GMRES will likely converge for general sparse systems. We will use CG rather than GMRES because the systems we are solving are symmetric and CG is likely to have faster convergence on our SPD systems. The problem of which preconditioner to use with our solver will be evaluated in Chapter 4.

Chapter 3

Baseline Subsurface Flow Performance

The entire application of subsurface flow creates a very large set of linear systems. The application we run simulates 336 hours of physical time in half-hour timesteps. This simulation solves the nonlinear system at each step in six nonlinear iterations. For each of these steps, there is a large sparse symmetric linear system that must be solved.

3.1 Baseline Results for Entire Application

We begin by experimenting with solving linear systems with four-cores and sixteen-cores. The sixteen-core setup has greater resolution of the problem space, and thus is a larger linear system at each step. We also present work done on a single core for ease of programming, execution and variety of available solvers, but the results can easily be extrapolated to multicore systems. Surprisingly it was found that the structure of the matrix stays remarkably consistent through execution of the simulations. Neither the nonlinear iteration nor the timestep makes a significant difference to the matrix structure. Figure 3.1 shows the structure of the matrices at timestep 0.5, nonlinear iteration 0 and timestep 167.5, nonlinear iteration 5. Note that there is no visible difference in the matrix structure despite the very different time and iteration. The number of nonzeros in each matrix is listed with the figure, $1,073,329$ in the first matrix and $1,070,711$ in the second – a difference of only about 0.2%.

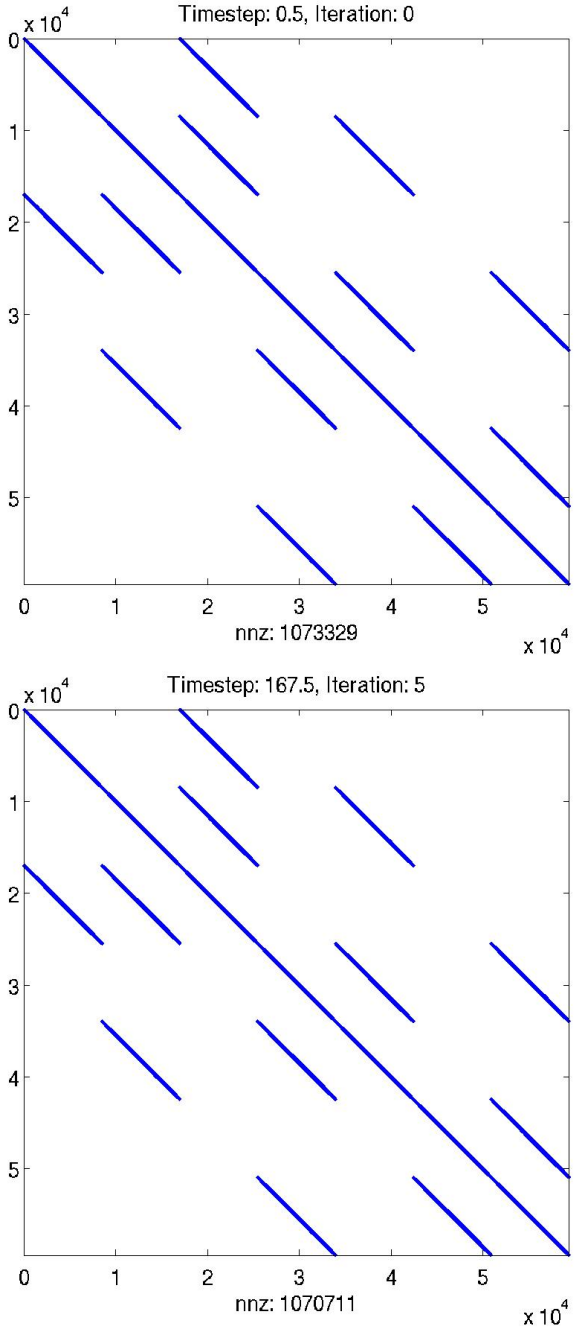

Fig. 3.1. The structure of the matrices is nearly identical at two very different instances of the application.

The entire set of linear systems formulated from the subsurface flow simulation number in the thousands. With 336 hours in half-hour timesteps and six nonlinear iterations, there are $4,032$ linear systems to solve and each one can be solved with a number of different methods. Because of the overwhelming size of this space, we look for stages that are likely to show the greatest difference in behavior instead of using every system.

3.2 Application Variance

To get the biggest possible range from the linear systems space we use the first ten, middle ten, and final ten timesteps only. We initially only evaluate the first and last nonlinear iterations to further decrease the overhead of running tests on the simulation. Intuitively, the first and last nonlinear iterations are the most likely to vary. Choosing timesteps and nonlinear iterations in this order should create a continuous subspace of the original problem such that the problems get more difficult with higher timestep and higher nonlinear iteration number. As seen in Figure 3.1, the structure of the matrices even for this large disparate sample of the application were remarkably similar. If this sampling of the space is effective in representing the entire spectrum of solver behavior, we can extrapolate the results found here to the entire space for complete application behavior and we can evaluate any single linear system to develop methods for improving performance.

We first describe preliminary results for standard characterizing of basic solvers. To get a baseline for the performance of the solver, we first use default parameters to some common methods for solving large sparse systems. This will give us information to know

what types of solver are likely to work well for the entire simulation. The baseline solvers we use are algebraic multigrid (AMG), Jacobi, symmetric successive over-relaxation (SSOR), sparse approximate inverse (SPAI), and incomplete triangular factorization (ILU). PETSc does not support incomplete Cholesky factorization (IC) for multicore systems, so these experiments will only be run in single core in Chapter 5. These solvers are described in Section 4.1, and are shown here to see the trend of execution of the simulation.

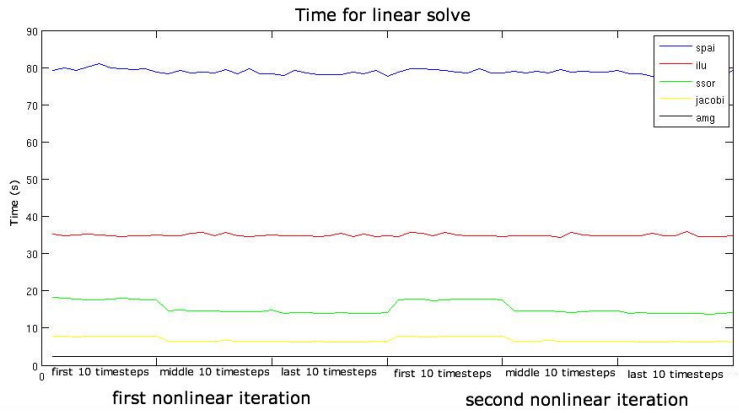

Fig. 3.2. The performance of the solver varies little over execution of the whole program. We may look at a single instance and extrapolate results.

Figure 3.2 shows the consistent performance throughout the execution. The figure is organized with the first half as the first nonlinear iteration, and the second half of the plot is the last nonlinear iteration. Each half is divided into thirds, the first, middle, and last set of ten timesteps. In this first brief examination, the AMG preconditioner was always the best solver for every timestep and nonlinear iteration examined. The next best solvers were Jacobi and SSOR. Finally, the worst performers were SPAI and ILU, which both failed to converge. This can also be seen in Figure 3.2, which shows this

order of solution time. Both the four-core and sixteen-core setups had the same order of method performance. The relative differences in performance were also approximately the same. This is expected based on the examination of the structure of the two matrices and it shows that the solvers are parallelizing well and similarly, so the performance is not degrading over the bigger size system. Since we see the same results regardless of the timestep or nonlinear iteration, the performance does not vary much. Again, this was true for both the four-core and sixteen-core setups. We thus hypothesize that the relative performance of each preconditioning method is likely to remain fairly constant across the spectrum.

Chapter 4

Characterization of Solver Performance

We want to evaluate the preconditioners and reordering presented in Chapter 2 in a controlled setup. This will give information about how to best improve performance of the entire application more easily and in an isolated way. In addition to reordering the preconditioner, there are other techniques that may be able to increase performance for this application. The first idea we explore for improved performance is using a 'sparsified' matrix to compute a more lightweight preconditioner. Additionally, we try the use of multiprecision algorithms for storing the matrix and preconditioner. Traditionally, double-precision scalars are used for scientific computations, but with the increasing use of graphics processing units (GPUs) and Cell processors, single-precision may need to be used to get optimal performance.

4.1 Preconditioners Evaluated

There is a large number of preconditioned iterative methods used to solve large sparse linear systems and each method shows different characteristics. The structure and eigenspectrum of the 'A' matrix of the linear system contributes significantly in determining which preconditioner will be most effective in solving the system. There have been countless papers showing that there is no black-box preconditioner capable of guaranteeing convergence for every linear system in optimal time. Therefore, we

examine a variety of preconditioners that cover a wide spectrum of the preconditioning possibilities. At this point we do not know a priori which type of preconditioner is likely to perform best for our given system.

We know that the linear systems formed in the application are symmetric because the nonlinear solver used is Picard method which is known to form symmetric systems [1, 48]. For solving symmetric positive-definite (SPD) linear systems, it is known that conjugate gradient (CG) shows excellent convergence on some domains when combined with preconditioning [14, 33, 41, 54]. Therefore all of the methods we use will be preconditioned CG.

The preconditioners we examine are Jacobi [15], symmetric successive over-relaxation (SSOR) [3], incomplete triangular factorization (ILU) [4, 11, 17, 51], sparse approximate inverse (SPAI) [7, 13, 29], incomplete Cholesky factorization (IC) [37, 43, 44], and algebraic multigrid (AMG) [9, 30]. Table 4.1 describes each of the preconditioners that will be evaluated.

Solvers		
Name	Description	Name in PETSc
Jacobi	Uses diagonal	jacobi
Symmetric successive over-relaxation	Uses upper triangular	ssor
Sparse approximate inverse	Approximates the inverse	spai
Incomplete factorization	Uses LU factorization	ilu
Incomplete Cholesky	Uses SPD properties	icc
Algebraic multigrid	Heirarchy of operators	hypre-boomeramg

Table 4.1. A brief description of the preconditioners used as well as their name in PETSc.

The simplest preconditioner is the Jacobi method which only uses information from the diagonal entries of the matrix. Inverting a diagonal matrix is a very lightweight operation so computing this preconditioner is fast. However, it is known that this preconditioner does not converge for all SPD systems.

An extension to the Jacobi preconditioner is SSOR. It uses the triangular part of the symmetric matrix as the preconditioner, but also uses a relaxation constant (ω) to smooth out error from the newest iteration. The choice of the ω parameter is the relaxation constant and is important to the rate of convergence [3].

The dense triangular factorization (LU) direct solver uses the inverse of the triangular portion of the matrix, however ILU is a preconditioner that uses an incomplete factorization of the matrix to reduce fill-in of the preconditioner. There are two primary criteria used to reduce fill-in. The first is by dropping values below a specified threshold. The alternative is to use level-of-fill, where lower levels usually have less fill.

The IC preconditioner is commonly used for SPD matrices because it shows good convergence in many cases. The concept behind IC is similar to ILU, but specific to SPD matrices and uses half the operations as a result. We use the same two options to reduce fill-in for the IC preconditioner.

SPAI uses a preconditioner that approximates the inverse to the matrix. The sparsity can be varied, between using just the diagonal or a complete inverse of the matrix. A common choice for the sparsity is to use the same structure as the original matrix [29].

AMG can be used as a stand-alone solver or as a preconditioner. We will be using AMG as a preconditioner to CG, in the same way as the other preconditioners. The AMG preconditioner is computed in Hypre [25], a third party library for PETSc.

4.2 Reordering

To expand the variety of solvers, we vary the parameters and also examine the effect of reordering the 'A' matrix with different factorization levels. We will use a few commonly used reordering schemes, quotient minimum degree (QMD) [28, 38], reverse Cuthill-McKee (RCM) [16], one-way dissection (1WD), and nested dissection (ND) to evaluate the effect of reordering in order to lessen the band around the diagonal [2, 18, 22]. It is a known problem that finding the reordering of a sparse matrix with least fill-in is NP-Complete, so we rely on these heuristics.

QMD is a simple reordering method that chooses the order of vertex elimination based on the degree. Since the elimination of a vertex causes its neighbors to become adjacent, it simply removes the remaining vertex with the minimum degree. QMD is a greedy algorithm and thus is only guaranteed to find the local optimum and not the global solution. This is quite easy to describe, but running time of the algorithm can be quite slow in practice [50]. The slow running time of the algorithm can be compensated for if the fill-in of the preconditioner is very low because the ordering is good.

RCM is a reordering that is most easily described as a breadth-first search (BFS) on the graph, but with the order reversed [39]. The initial vertex of the BFS, is often chosen arbitrarily. The reordering produced has been shown to have a small bandwidth

which should cause less fill-in. Reversing the ordering does not change the bandwidth properties, but can decrease fill-in of the factoring.

1WD finds a separator in the graph that best splits the graph into two equal partitions with the minimum separator size. The two partitions reorder the matrix such that the only adjacency between them is the separator, which should be of minimum size. ND is an extension to 1WD that recursively finds the separator on the smaller dissections until it can no longer be split.

4.3 Scope of Parameters

The parameters examined are reordering the 'A' matrix, and varying the factorization levels. Table 4.2 explains the exact options that will be examined below. There are two setups selected using AMG. The first (*) is a basic setup without a priori knowledge of the problem space. The only multigrid parameter is the strong threshold set to 0.6. The second setup (**) requires a lot of knowledge about the linear system that will be formed. These parameters were determined analytically by the Army Corps of Engineers, and are as follows. The number of levels of aggressive coarsening is set to 25, it uses the classical type of interpolation for hyperbolic PDEs. The AMG preconditioner is applied once for each conjugate gradient application, and the coarsening type is set to use a hybrid modified independent sets scheme, with one Ruge-Stuben pass followed by coarsening using parallel independent sets.

Setup			
Preconditioner	Ordering	Factorization	PETSc
IC	1WD	L0	cg icc 1wd l0
IC	1WD	L1	cg icc 1wd l1
IC	NAT	L0	cg icc natural l0
IC	NAT	L1	cg icc natural l1
IC	ND	L0	cg icc nd l0
IC	ND	L1	cg icc nd l1
IC	QMD	L0	cg icc qmd l0
IC	QMD	L1	cg icc qmd l1
IC	RCM	L0	cg icc rcm l0
IC	RCM	L1	cg icc rcm l1
AMG	N/A	*	cg hypre-boomeramg
AMG	N/A	**	cg hypre-boomeramg

Table 4.2. Parameters for the linear solver setup discussed in Section 4.3.

4.4 Thresholding

We have previously seen that in some applications, dropping small values from the matrix when computing the preconditioner can lead to better performance [12]. This modified 'A' matrix will be an approximation to the real matrix so each computation will be less effective but faster. As expected, it will lead to a higher iteration count, however the hope is that each iteration will gain enough performance to overcome the increased number of iterations. We will study the effect of ignoring values less than 1×10^{-03} and 1×10^{-04} when creating the preconditioner.

4.5 Multiprecision

Recent research has shown that some scientific computation codes can be made to run faster with the use of mixed-precision [5]. Baboulin, et al. showed that using single-precision computations with refinement in double-precision can be faster than standard

computation in double-precision throughout. We will try to experiment with this concept on a few linear systems generated from this application.

Trilinos [31, 32] provides new functionality allowing a user to easily declare the use of single- or double-precision linear algebra computation in a package called Tpetra [26, 53]. There currently are not capabilities to switch precision during the linear solve, so we only compare single- versus double-precision. Additionally, because it is so new, there are only a limited set of solvers and preconditioners available now, so we only use a CG solver with ILU preconditioner. Future tests will experiment with the use of graphics processor units (GPUs) or the Cell processor, both of which execute much faster in single-precision compared to double-precision. Results have shown that the Nvidia Tesla C1060 GPU can have peak performance twelve times faster with single-precision compared with double-precision [55]. This result can be explained by the design of a GPU, which was built for graphics running in 24 or 32 bits per pixel.

Chapter 5

Experimental Results

Section 5.1 explains the computing platform used to test results as well as a description of the linear solver package used. We present results from testing preconditioners on a single linear system. These results are used to find what setups show promise for speeding up the subsurface flow simulation. We do not expect that this single instance will carry over directly, so we also test the full results. Next, using this information, we show how the entire application performs better using our methodology.

5.1 Experimental Setup

Experiments are run on the Cray XT4 system 'Jade' in the DoD High Performance Computing Modernization Program (HPCMP). This system contains 2,228 compute nodes, each is a single AMD 2.1 GHz quad-code Opteron processor with Linux microkernel. Each node has 8GB of dedicated memory, and we reserve the entire node to avoid interference with other jobs running.

The linear system solver used is PETSc 3.0.0.7 with optimization flags enabled and Cray auto-tuning also enabled. PETSc currently does not support a parallel implementation of incomplete Cholesky factorization (IC), so these experiments must be run in serial. Initial results use four- and sixteen-core results of other preconditioners that support parallel processing. Though our full application results are not parallel, a solver

that supports parallel preconditioning of IC would perform similarly. Future research would implement the solvers in the Trilinos framework which supports parallel IC for Epetra objects [31].

5.2 Results From Isolated Linear System

We choose an arbitrary timestep and nonlinear iteration of 163.5 and 5 respectively to run experiments. From Section 3.2, we know that there is little variation in solver performance based on the timestep or iteration number. To get results on just this system, the 'A' matrix and the right-hand side vector are saved to disk to be evaluated individually. The linear solver program is simple and only loads the matrix and vector to be solved with selected PETSc solver options.

Before solving the linear system, we first find some characteristics of the matrix and right hand-side vector. Using Matlab, we found that the condition number estimate of the matrix is 9.70×10^8, which, while large, implies that the system should be solvable to machine precision. The maximum value in the vector is 1.10×10^9 and the minimum is approximately the negation. For this test, the initial guess for the solution vector will be filled with random values. In the actual program, the initial guess is taken from previous iterations, and thus is likely to perform better. Since, we are only looking at this system, we do not have a better initial guess available.

The solver we use applied iterations until the residual reached a tolerance less than 10^{-5}, however, stopping criteria was not exact and sometimes reached a much better tolerance. Using a reordered matrix had a profound effect on the performance as shown

in Table 5.1. For the IC results, the best to worst ordering results were reverse Cuthill-McKee (RCM), one-way dissection (1WD), natural, quotient minimum degree (QMD), and nested dissection (ND). This clearly shows that the reduced bandwidth of the reordered matrix contributed to more efficient computation and less fill-in.

Reordering					
Preconditioner	Reordering	Fact-Level	Time	Iterations	Residual
IC	1WD	L0	0.82	53	1.25e-06
IC	1WD	L1	40.55	1914	1.38e-07
IC	NAT	L0	3.30	311	1.45e-06
IC	NAT	L1	85.61	4619	4.58e-08
IC	ND	L0	25.97	1931	2.09e-07
IC	ND	L1	161.62	6464	2.21e-08
IC	QMD	L0	23.70	1587	4.98e-07
IC	QMD	L1	160.97	6270	2.42e-08
IC	RCM	L0	0.77	49	9.35e-07
IC	RCM	L1	39.35	1909	1.38e-07
AMG	N/A	*	2.47	29	1.32e-06
AMG	N/A	**	1.38	21	4.83e-07

Table 5.1. A comparison of IC with specified reordering against AMG.

Another parameter that we get results for is factorization level. For every reordering method, the linear solve time was significantly better using level zero fill (L0) compared to level one fill (L1). This is not entirely unexpected, however it is surprising that L0 requires fewer linear iterations than L1. We would expect that L1 would construct a preconditioner that better approximates the inverse and therefore use fewer iterations. To explain this, we compare the condition number of the preconditioned operators. For the RCM reordering, the L0 has a condition number of approximately 29, compared to a condition number of approximately 1.60×10^7 for L1 factorization. In both of these cases, this is an improved condition number compared to the matrix,

however the L0 is very low and can be solved much more easily. With this information, it is no longer surprising that L0 factorization is solving the solution much better. This condition number may be an artifact of the implementation in PETSc and the way it handles pivot failures in IC, but the results were verified in Trilinos.

We have results comparing the IC preconditioners to two AMG setups. Section 4.3 describes (*) as the setup without application knowledge, and (**) as a setup in which very specific multigrid parameters are chosen with knowledge of the system formed. We see that having application knowledge led to a much better preconditioner with AMG, the better solver was almost twice as fast as a basic AMG setup. When comparing AMG to the IC preconditioners performance, the times are faster than most of the setups except for RCM and 1WD for L0 factorization. Despite being slower that these two solvers, AMG used fewer linear iterations, implying that each application of the preconditioner was slower than the IC applications. Figure 5.1 shows a comparison of the AMG methods against the various orderings, each with L0 and L1 factorizations. In the first figure, the dotted lines represent the time AMG spent in the solver, and we see that RCM and 1WD with L0 are the only setups faster than either AMG* or AMG** solve time. The second figure shows the iterations of the methods. AMG, as stated earlier, used fewer iterations, displayed by the dotted lines being lower than any of the bars.

5.3 Results of Entire Application Run

We now run the entire application with solver parameters described in Section 4.3. Table 5.2 shows the solve times for each of the setups we have chosen to test. From

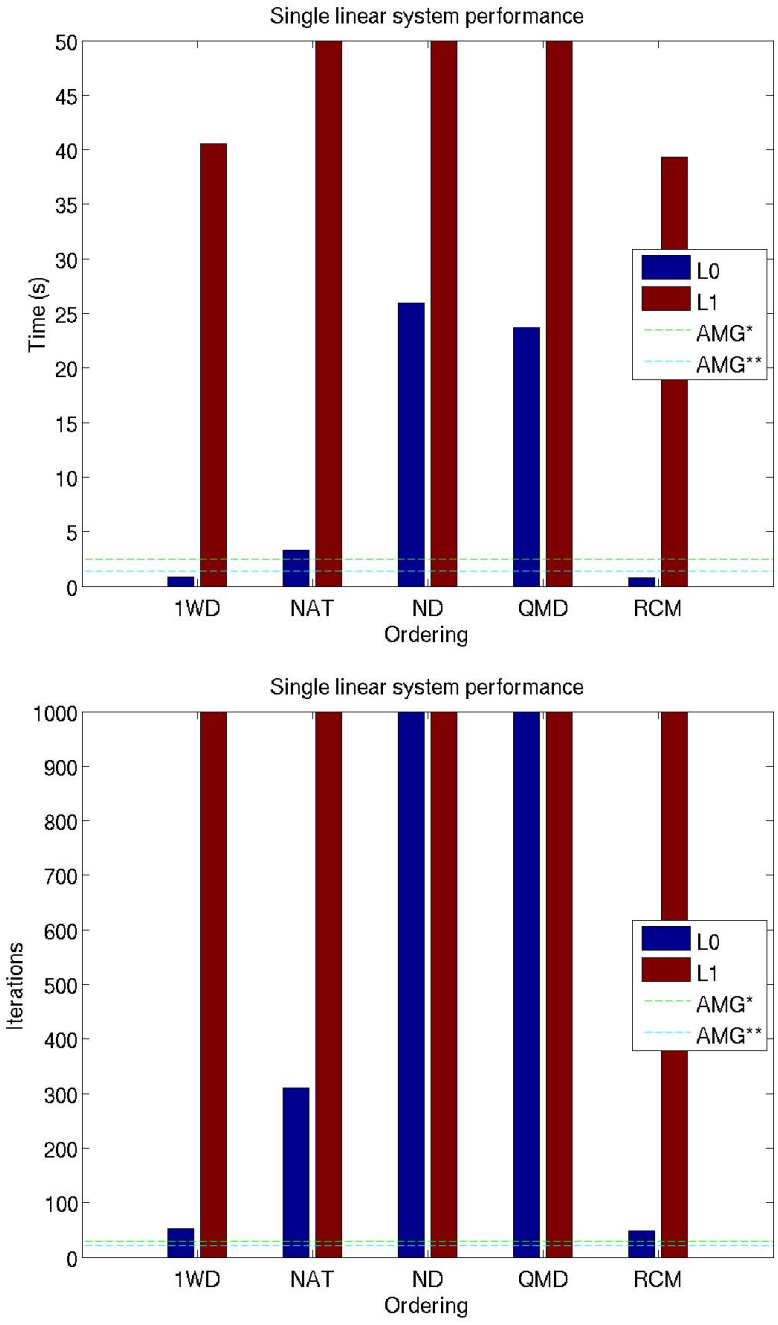

Fig. 5.1. Solve times and iterations for IC and AMG with all orderings and L0 and L1 factorizations for a specified instance of the application.

the results of Section 5.2, it comes as no surprise that L1 does not do well. We again hypothesize that L1 creates a worse preconditioner than L0 because the condition number of the preconditioned operator was much higher and converged slowly.

Application			
Preconditioner	Reordering	Fact-Level	Time
IC	1WD	L0	5243.79
IC	1WD	L1	99928.35
IC	NAT	L0	5235.15
IC	NAT	L1	99993.74
IC	ND	L0	5239.71
IC	ND	L1	99944.46
IC	QMD	L0	5238.81
IC	QMD	L1	99837.33
IC	RCM	L0	5231.15
IC	RCM	L1	99902.62
AMG	N/A	*	7490.28
AMG	N/A	**	4571.77

Table 5.2. Linear solve time for entire application comparing reordering and factorization level of IC to AMG.

Figure 5.2 shows how each method fared against AMG. Over the entire application, it was surprising to find that reordering played a much smaller role in total solve time than was expected from tests on a single system. This result holds true for both L0 and L1, though the times between the two were drastically different. For reordering, RCM was best followed by natural, QMD, ND, then 1WD, ranging from 5231 to 5244 seconds for L0 factorization. This can in part be explained by the initial guess vector, as discussed in Section 5.2, which initially was random, and now is taken from a previous iteration.

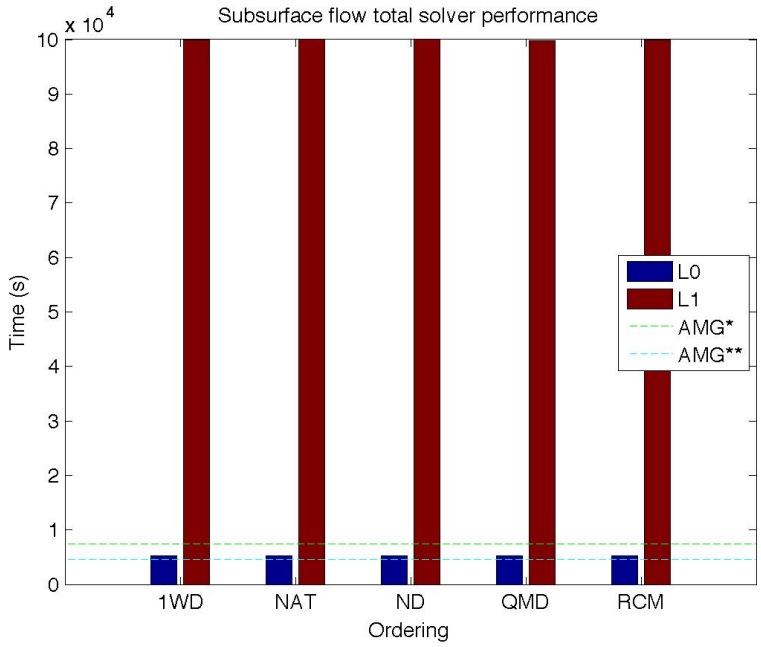

Fig. 5.2. A comparison of total linear solve time for AMG against ordering and factorizations for IC.

There was a large difference in solve time between AMG with the best parameters and a basic AMG. The solve with basic parameters for AMG took 65% longer than those found with analytical knowledge of the problem. The second setup coarsened much better because the coarsening levels were suited well for the geometric mesh that the problem comes from in the real subsurface flow.

These results show that AMG cannot be used as a black-box solver for this application because the choice of parameters greatly influences the convergence rate. On the other hand, the space of parameters is relatively small for IC – we only choose between five orderings and two level-of-fill factorizations compared with many continuous parameters for AMG.

5.4 Thresholding Results

As discussed in Section 4.4, thresholding refers to constructing the preconditioner from an A', which is modified to drop values below a specified threshold. We experimented with thresholding on a single instance of the application; the same instance as Section 5.2. The drop-level values we tried were 10^{-3} and 10^{-4}. Table 5.3 shows that thresholding the matrix to create a preconditioner does not appear to work well for this application. For both drop-levels, the linear solver was not able to converge after 10,000 iterations.

Thresholding				
Solver	Drop-Level	Time	Iterations	Residual
RCM-ICC-L0	10^{-3}	201.62	10000	9.12e-03
RCM-ICC-L0	10^{-4}	210.43	10000	9.11e-03

Table 5.3. A preconditioner constructed from the modified A matrix was not able to converge.

This shows that the idea of dropping values in forming the preconditioner is very application dependent. We showed in [12] that dropping small values in a support vector machine (SVM) can improve performance by 25%, and 100% in the best case. Clearly, the linear systems formed in subsurface flow are different than those from an SVM formulation, and are more sensitive to dropping small values. Based on these initial results, it does not make sense to try this idea with the whole application because it will not work well.

5.5 Mixed-Precision Results

The results for a few mixed precision-experiments are shown in Figure 5.3. The comparison shows the time taken to reach convergence criteria for a solve using single-precision versus double-precision. Convergence criteria for single-precision was set to 10^{-11} and 10^{-14} for double-precision. Data points above the dotted line represent the single-precision solve running faster.

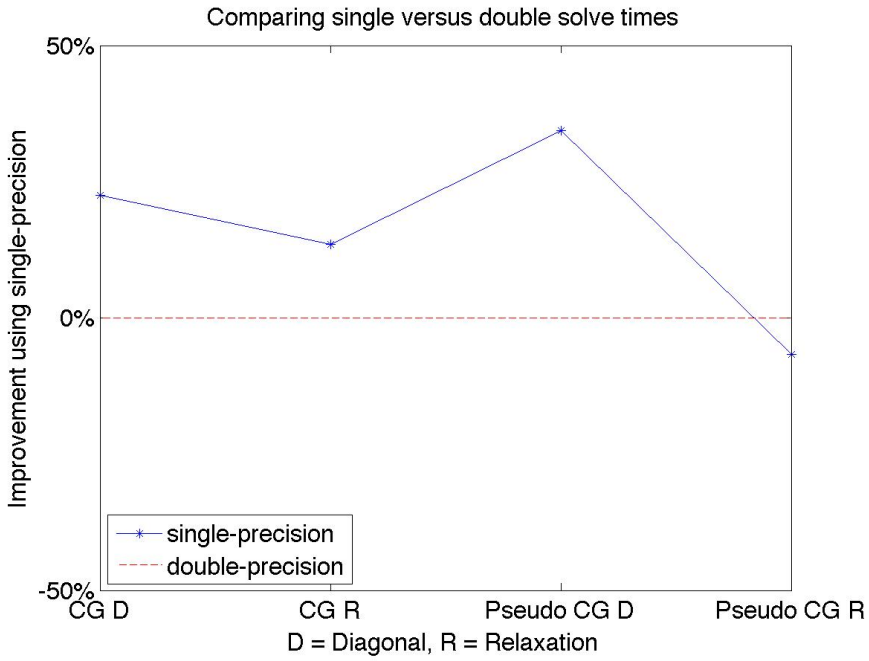

Fig. 5.3. Performance of single versus double-precision solvers for CG with a diagonal and relaxation preconditioners.

Pseudo CG with a diagonal preconditioner showed the most gain for using single instead of double-precision. In this case, the single-precision solve took 1077.67 seconds using the Tpetra and Ifpack2 setup in Trilinos. This compares with 1447.81 (34% more time) for the same setup in double-precision. The single-precision solve took 14072 linear

iterations to converge, whereas the other took 12853, implying that more iterations are needed, but each is faster because there is less data moved. In only one of the four solves double-precision converged faster. Pseudo CG with a relaxation preconditioner took 1093.15 seconds with single and 1019.77 with double-precision. In this case, the speed difference is relatively small compared to the difference that single-precision is faster in the three other cases.

These times are extremely slow because the solver and preconditioner parameters were limited, and poor choices for this application. We saw in Section 3.2 that these preconditioners do not compare well to AMG and should not expect them to in Trilinos either. Also, because the software stack used in Tpetra and Ifpack2, needed for using templates in high-level software programming, creates overhead not seen in PETSc, the times are slower.

The differences in these times would be much greater on a graphics processing unit (GPU) [45, 10] or Cell processor [47]. These results were reached using the Cyberstar machine on the Penn State campus using one core of an AMD node. The programming was done in C++ with data structures and solves in Trilinos with optimization flags on. Because we were not using a GPU, the enhancements were much more modest. We propose using linear solves in GPUs for the entire application as future work.

Chapter 6

Conclusions

The computational simulation of subsurface flow is dominated by the solving of large symmetric positive-definite (SPD) linear systems. The linear systems are formulated using Picard iteration, which creates symmetric systems at each nonlinear iteration. We found that the basic structure of these linear systems is consistent throughout execution of our simulation. Thus, we are able to experiment with individual linear systems to determine techniques to improve solve time over the whole application. We use incomplete Cholesky factorization (IC) and algebraic multigrid (AMG) for an isolated system and then after learning parameters, run the full simulation. Additionally, we experimented with reordering schemes to reduce fill-in of the incomplete factorizations. Our experiments indicated that reverse Cuthill-McKee (RCM) was the best reordering followed by one-way dissection (1WD), natural, quotient minimum degree (QMD), and nested dissection (ND). The level-of-fill factorization was found to work best with level zero fill (L0) because of the large condition number of level one fill (L1). In the single linear system test, we found that IC with RCM and L0 factorization was superior to either version of AMG tested. For the entire application, the AMG with a priori knowledge of the linear systems performed best, despite our single matrix results.

We have shown methodology for speeding up subsurface flow on a single core. The solvers and preconditioners are limited because of software available and the setup

of the overall application. As a result, our work is in a preliminary stage. We propose the use of Trilinos to enable IC in a multicore environment. New tools in Trilinos will also allow for the use of mixed-precision solvers with or without graphics processors.

Work done by Brice Toth and Sanjukta Bhowmick showed that targeted combinations of preconditioners and solvers can have unexpected benefits [8, 58]. We also saw that smart use of iterative methods and preconditioners can lead to large performance gains in the support vector machine (SVM) application [12]. This procedure can be applied to the subsurface flow as well with an appropriate software setup.

Bibliography

[1] A. Aldama and C. Paniconi, *An Analysis of the Convergence of Picard Iterations for Implicit Approximations of Richard's Equation*, (1992).

[2] C. Ashcraft and J. Liu, *Robust ordering of sparse matrices using multisection*, SIAM Journal on Matrix Analysis and Applications, 19 (1998), p. 816.

[3] O. Axelsson, *A generalized SSOR method*, BIT Numerical Mathematics, 12 (1972), pp. 443–467.

[4] O. Axelsson and N. Munksgaard, *Analysis of incomplete factorizations with fixed storage allocation*, Preconditioning Methods–Theory and Applications, D. Evans, ed., Gordon and Breach, New York, (1983), pp. 265–293.

[5] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, *Accelerating scientific computations with mixed precision algorithms*, Computer Physics Communications, 180 (2009), pp. 2526–2533.

[6] R. Barrett, *Templates for the solution of linear systems: building blocks for iterative methods*, Society for Industrial Mathematics, 1994.

[7] M. Benzi, C. Meyer, M. Tuma, et al., *A sparse approximate inverse preconditioner for the conjugate gradient method*, SIAM Journal on Scientific Computing, 17 (1996), pp. 1135–1149.

[8] S. Bhowmick, B. Toth, and P. Raghavan, *Towards Low-Cost, High-Accuracy Classifiers for Linear Solver Selection*, Computational Science–ICCS 2009, (2009), pp. 463–472.

[9] A. Brandt, *Algebraic multigrid theory: The symmetric case*, Applied Mathematics and Computation, 19 (1986), pp. 23–56.

[10] I. Buck, *Gpu computing with nvidia cuda*, in ACM SIGGRAPH 2007 courses, ACM, 2007, p. 6.

[11] T. F. Chan, Henk, A. Van, and H. A. van der Vorst, *Approximate and incomplete factorizations*, in ICASE/LaRC Interdisciplinary Series in Science and Engineering, 1994, pp. 167–202.

[12] A. Chatterjee, K. Fermoyle, and P. Raghavan, *Characterizing sparse preconditioner performance for the support vector machine kernel*, Procedia Computer Science, 1 (2010), pp. 367–375.

[13] E. Chow, *A priori sparsity patterns for parallel sparse approximate inverse preconditioners*, SIAM J. Sci. Comput., 21 (2000), pp. 1804–1822.

[14] P. Concus, G. Golub, and P. Dianne, *A Generalized Conjugate Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations*, Computer Science Department, Stanford University, 1976.

[15] P. Concus, G. Golub, and G. Meurant, *Block preconditioning for the conjugate gradient method*, SIAM J. Scl. STAT. COMPUT, 6 (1985).

[16] H. Crane Jr, N. Gibbs, W. Poole Jr, P. Stockmeyer, C. O. WILLIAM, and M. W. V. D. O. MATHEMATICS, *Matrix bandwidth and profile reduction*, ACM Trans. Math. Softw, 2 (1976), pp. 375–377.

[17] E. D'Azevedo, P. Forsyth, and W. Tang, *Towards a cost-effective ILU preconditioner with high level fill*, BIT Numerical Mathematics, 32 (1992), pp. 442–463.

[18] E. D'azevedo, P. Forsyth, W. Tang, U. of Waterloo. Dept. of Computer Science, and U. of Waterloo. Faculty of Mathematics, *Ordering methods for preconditioned conjugate gradient methods applied to unstructured grid problems*, University of Waterloo, Faculty of Mathematics, 1989.

[19] J. Demmel and I. Books24x7, *Applied numerical linear algebra*, vol. 150, Society for Industrial and Applied Mathematics Philadelphia, 1997.

[20] J. Dongarra, *Numerical linear algebra for high-performance computers*, Society for Industrial Mathematics, 1998.

[21] I. Duff, A. Erisman, and J. Reid, *Direct methods for sparse matrices*, Oxford University Press, USA, 1989.

[22] I. Duff and G. Meurant, *The effect of ordering on preconditioned conjugate gradients*, BIT Numerical Mathematics, 29 (1989), pp. 635–657.

[23] T. Dupont, R. P. Kendall, and J. Rachford, H. H., *An approximate factorization procedure for solving self-adjoint elliptic difference equations*, SIAM Journal on Numerical Analysis, 5 (1968), pp. 559–573.

[24] A. Ern, V. Giovangigli, D. Keyes, and M. Smooke, *Towards polyalgorithmic linear system solvers for nonlinear elliptic problems*, SIAM Journal on Scientific Computing, 15 (1994), pp. 681–703.

[25] R. Falgout and U. Yang, *hypre: A library of high performance preconditioners*, Computational ScienceICCS 2002, (2009), pp. 632–641.

[26] K. Fermoyle, M. Heroux, E. Cyr, and S. S. Collis, *Epetra/AztecOO and Related to Tpetra/Stratimikos and Related: A Conversion Guide*, tech. rep., Sandia National Laboratories, 2011.

[27] A. George and J. Liu, *Computer solution of large sparse positive definite systems*, PRENTICE-HALL, INC., ENGLEWOOD CLIFFS, NJ 07632, (1981).

[28] A. George and J. Liu, *The evolution of the minimum degree ordering algorithm*, Siam review, 31 (1989), pp. 1–19.

[29] M. Grote and T. Huckle, *Effective parallel preconditioning with sparse approximate inverses*, in Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, Society for Industrial & Applied, 1995, p. 466.

[30] V. E. Henson and U. M. Yang, *Boomeramg: A parallel algebraic multigrid solver and preconditioner*, Applied Numerical Mathematics, 41 (2002), pp. 155 – 177.

[31] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams, *An Overview of Trilinos*, Tech. Rep. SAND2003-2927, Sandia National Laboratories, 2003.

[32] M. A. Heroux and J. M. Willenbring, *Trilinos Users Guide*, Tech. Rep. SAND2003-2952, Sandia National Laboratories, 2003.

[33] M. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J, pp. 49–409.

[34] P. Huyakorn and G. Pinder, *Computational methods in subsurface flow*, (1986).

[35] M. Jones and P. Plassmann, *The efficient parallel iterative solution of large sparse linear systems*, tech. rep., Argonne National Lab., IL (United States), 1992.

[36] L. Kuiper, *A comparison of iterative methods as applied to the solution of the nonlinear three-dimensional groundwater flow equation*, SIAM Journal on Scientific and Statistical Computing, 8 (1987), p. 521.

[37] C. Lin and J. Moré, *Incomplete Cholesky factorizations with limited memory*, SIAM Journal on Scientific Computing, 21 (2000), pp. 24–45.

[38] J. Liu, *Modification of the minimum-degree algorithm by multiple elimination*, ACM Transactions on Mathematical Software (TOMS), 11 (1985), pp. 141–153.

[39] W. Liu and A. Sherman, *Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices*, SIAM Journal on Numerical Analysis, 13 (1976), pp. 198–213.

[40] J. Meijerink and H. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Mathematics of Computation, (1977), pp. 148–162.

[41] N. Munksgaard, *Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients*, ACM Transactions on Mathematical Software (TOMS), 6 (1980), pp. 206–219.

[42] N. Nachtigal, S. Reddy, and L. Trefethen, *How fast are nonsymmetric matrix iterations?*, SIAM Journal on Matrix Analysis and Applications, 13 (1992), p. 778.

[43] E. Ng, B. Peyton, and P. Raghavan, *A blocked incomplete Cholesky preconditioner for hierarchical-memory computers*, in Proceedings of the Fourth IMACS International Symposium on Iterative Methods in Scientific Computation. IMACS Series in Computational and Applied Mathematics: Iterative Methods in Scientific Computation IV, DR Kincaid and AC Elster, eds, Citeseer, 1999, pp. 211–222.

[44] E. Ng and P. Raghavan, *Performance of greedy ordering heuristics for sparse Cholesky factorization*, SIAM Journal on Matrix Analysis and Applications, 20 (1999), pp. 902–914.

[45] C. Nvidia, *Programming Guide*, Version 0.8, 1, p. 53.

[46] C. Paniconi, A. Aldama, and E. Wood, *Numerical evaluation of iterative and noniterative methods for the solution of the nonlinear Richards equation*, Water Resources Research, 27 (1991), pp. 1147–1163.

[47] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, et al., *The design and implementation of a first-generation CELL processor*, in 2005 IEEE International Solid-State

Circuits Conference, 2005. Digest of Technical Papers. ISSCC, 2005, pp. 184–592.

[48] R. Rach, *On the Adomian (decomposition) method and comparisons with Picard's method*, Journal of Mathematical Analysis and Applications, 128 (1987), pp. 480–483.

[49] P. Raghavan and K. Teranishi, *Parallel Hybrid Preconditioning: Incomplete Factorization with Selective Sparse Approximate Inversion*, SIAM Journal on Scientific Computing, 32 (2010), p. 1323.

[50] E. Rothberg and B. Hendrickson, *Sparse matrix ordering methods for interior point linear programming*, INFORMS Journal on Computing, 10 (1998), pp. 107–113.

[51] Y. Saad, *ILUT: A dual threshold incomplete LU factorization*, Numerical linear algebra with applications, 1 (1994), pp. 387–402.

[52] ——, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[53] M. Sala, M. A. Heroux, and D. M. Day, *Trilinos Tutorial*, Tech. Rep. SAND2004-2189, Sandia National Laboratories, 2004.

[54] J. R. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*, tech. rep., Pittsburgh, PA, USA, 1994.

[55] R. Suda, T. Aoki, S. Hirasawa, A. Nukada, H. Honda, and S. Matsuoka, *Aspects of GPU for general purpose high performance computing*, in Proceedings of

the 2009 Asia and South Pacific Design Automation Conference, IEEE Press, 2009, pp. 216–223.

[56] K. Teranishi, *Scalable hybrid sparse linear solvers*, Pennsylvania State University, 2004.

[57] W. Tinney and J. Walker, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, Proceedings of the IEEE, 55 (2005), pp. 1801–1809.

[58] B. Toth, *Machine learning techniques for sparse solver selection*, Master's thesis, The Pennsylvania State University, July 2009.

[59] Z. Zlatev, *Use of iterative refinement in the solution of sparse linear systems*, SIAM Journal on Numerical Analysis, 19 (1982), pp. 381–399.