A Simple Approach to Moderately Parallel SSOR and ILU0 Preconditioning for Iterative Solution of Linear Systems

Mark Kremenetsky*and Edward Rothberg†

July 12, 2001

Abstract

Preconditioned iterative methods are widely used for solving large, sparse linear systems of equations. This has led to significant interest in parallelization strategies. A wide range of preconditioners have been proposed, some of which scale extremely well on parallel machines. However, SOR (Successive Over-Relaxation) and ILU0 (Incomplete LU with 0 fill) remain two of the more effective general purpose schemes, and both are quite difficult to parallelize in their 'textbook' form. This paper describes a very simple parallelization scheme for moderately parallel machines. Our approach can be viewed as a simple extension of block multi-color schemes. The novel aspect is the method for constructing the blocks. We discuss implementation issues under OpenMP, and present results from a few important classes of linear systems, showing that convergence is only moderately degraded by the parallelization and that parallel scaling is quite good.

1 Introduction

The problem considered in this paper is the parallel iterative solution of systems of equations $Ax = b$, where A is a large, sparse, matrix. A preconditioned iterative method applies a preconditioning matrix M, where the hopefully easier system $M^{-1}Ax = M^{-1}b$ is solved instead of the original. We are interested in performing this computation effectively on moderately parallel machines, with 10 or fewer processors. Our target platform is a shared-memory system, programmed using OpenMP directives, but the approach described here could be implemented on a distributed memory system as well.

The approach described in this paper has been implemented and available in the SGI parallel iterative solver library since 1995 (documentation is included in the Appendix of this paper), and has been used in a variety of customer applications, including several commercial software products.

*Silicon Graphics, Inc., Mountain View, CA, 94043

†ILOG, Inc., Mountain View, CA 94043

1.1 Preconditioners

Given our moderately parallel target, good scaling would not compensate for poor sequential performance. Thus our interest is in methods that are competitive even on a single processor. Two very widely used preconditioners for sequential iterative solution are SOR and ILU. The preconditioning matrix for ILU is $M = LU$, where L and U are incomplete factors of A with limited fill. Probably the most commonly used version of ILU is ILU0, where the 0 indicates that no fill is allowed. The preconditioning matrix for SOR is $M = (L + D)^{-1}D(U + D)^{-1}$, where L, D, and U are the lower triangle, diagonal, and upper triangle of A, respectively. Note that for symmetric matrices, the SSOR (Symmetric SOR) variant of SOR, coupled with Eisenstat's trick [2], is particularly attractive, since preconditioning is obtained with essentially no additional computational cost over the unpreconditioned method.

The primary computational steps in the iterative solution of linear systems are a sparse matrix-vector product with matrix A, a linear solve $z = M^{-1}b$, and a set of vector additions and dot products. Of these steps, all but the solve $z = M^{-1}b$ are quite easy to parallelize. Since the M matrix for both SOR and ILU0 preconditioning is the product of two triangular matrices (and a diagonal matrix), this step reduces to a pair of triangular solves. Triangular solves on general sparse triangular matrices are well known to be difficult to parallelize.

1.2 Parallelization Strategies

The main difficulty in parallelizing a triangular solve is the highly recursive nature of the computation. Once an intermediate result is computed (the value of a single entry z_i in $z = M^{-1}b$), it is typically needed for the computation of multiple subsequent intermediate results. Such tight dependencies and such high communication costs greatly complicate parallelization.

Several strategies have been proposed to break these tight dependencies. Most involve partitioning the graph representation G of the matrix A (where G has a node i for every row/column in A, and a directed edge (i, j) for every non-zero value A_{ij}). A block ILU scheme, for example, finds a set of edges E_S in G such that removing those edges breaks G into multiple, disconnected subgraphs. The separator edges E_S are usually chosen so that the number of disconnected pieces is equal to the number of processors, and also so that the sizes of the pieces are comparable (for load balancing). The non-zero values in M corresponding to the edges in E_S are discarded, giving a block diagonal preconditioning matrix. Given block structure, each processor can compute its portion of $z = M^{-1}b$ independently.

A second common approach is a red-black, or more generally a multi-color strategy. In this approach, an independent set of nodes V_1 (a set of nodes where no two are adjacent) is chosen from G. This set of nodes is assigned to color 1 and removed from the graph (along with all incident edges). A second independent set V_2 is chosen, assigned color 2, and removed. The process is repeated until all nodes have been colored. The rows and columns of A are then permuted, such that all nodes of color i precede all those of color $i+1$. This permutation alters the dependencies in the triangular solve — intermediate results for a node of color j depend only on results from nodes with color $i < j$. All nodes of a certain color can therefore be computed in parallel.

While both of these approaches improve parallelism, they also typically degrade convergence significantly. Thus, the desirable properties of sequential SOR and ILU0 preconditioners are partially lost when the parallel versions are employed on multiple processors. In our experiments, we observed significant degradations with these approaches, even on two or four processors.

1.3 Block Multi-Color Methods

An alternative approach that reduces the degradation in convergence behavior is a hybrid of a block scheme and a multi-color scheme (see [4]). In such an approach, an edge separator E_S again disconnects the graph into multiple pieces. A quotient graph is then built to represent the remaining subgraphs, where the quotient graph has one node I per subgraph, and an edge between two nodes I and J if any edge in E_S connects a node in subgraph I with a node in subgraph J. This quotient graph is then colored. Colors are then processed sequentially. One limitation of this approach, especially for moderately parallel machines, is that if exactly one subgraph is assigned to each processor, then a subset of the processors sits idle for each color. A further refinement allows the assignment of multiple colors to each processor [4]. Empirically, convergence degrades as the number of colors is increased, so this approach introduces a delicate balance between parallel performance and convergence.

2 Method

This section describes our simple extension of block multi-color preconditioning. We describe our block construction scheme, discuss the advantages of the approach for moderately parallel machines, discuss parallelization issues in OpenMP, and present computational results.

2.1 Block Multi-Coloring

Our approach can be viewed in two ways — either as a constructive approach to choosing domains and colors for these domains within a block multi-color scheme so that each processor receives a well balanced set, or alternatively as a different way of looking at the problem of partitioning the graph G into multiple subgraphs. It is perhaps most easily described using the former view.

Our method begins by performing a one-way dissection on the graph G corresponding to A [3]. One-way dissection is a technique from the sparse factorization literature in which multiple vertex separators V_I are chosen such that (i) no two nodes in different separators are adjacent, (ii) the removal of all V_I from G disconnects the graph into multiple subgraphs G_I, and (iii) any resulting subgraph G_I is adjacent to at most two separators (V_{I-1} and V_I). The traditional approach to finding a one-way dissection is to first perform a Reverse Cuthill-McKee ordering [1] to find a dominant axis for the graph, then perform a level-set ordering on this axis, and then to choose a set of levels to remove. Each removed level becomes a vertex separator V_I.

Given this one-way dissection, we can then label the subgraphs G_I with color 1, the separators V_I with color 2, and map both G_I and V_I to processor I. If the number of vertices in G is large relative to the number of subgraphs G_I, then the sizes of the separators V_I will be small relative to the sizes of the subgraphs. This also makes it straightforward to choose V_I such that the load is well balanced among the G_I. Note there will be some load imbalance when the values in V_I are computed, since there will only be $P-1$ separators for P subgraphs. This turns out to be a minor issue, for two reasons. First, the separators are small compared to the subgraphs, so the imbalance is on a small portion of the work. Second, the relative size of the imbalance shrinks as the number of processors grows. When using two processors, one sits idle during separator processing, so half of the available compute resources are wasted. When using ten processors, only one sits idle still, so only ten percent are wasted.

2.2 Parallelization

Parallelization of this method using OpenMP is quite straightforward. We first perform the one-way dissection and reorder the matrix. Both are done sequentially. The cost is roughly equal the cost of 5 iterations of an SSOR preconditioned conjugate gradient method. Then we permute the rows and columns of A such that all nodes (rows) mapped to a processor are contiguous in the matrix, and such that all domain nodes precede all separator nodes within a processor. We use the same mapping to divide up the work of the matrix-vector multiplication, the dot products, the vector sums, and the triangular solves among processors.

The triangular solves are performed as a pair of 'parallel for' loops in OpenMP:

```
#pragma omp parallel for
  for (i = 0; i < domains; i++)
    for ( row = first_domain_row[i];
          row < last_domain_row[i];
          row++)
      process_row (row);

#pragma omp parallel for
  for (i = 0; i < domains-1; i++)
    for ( row = first_sep_row[i];
          row < last_sep_row[i];
          row++)
      process_row (row);
```

A global synchronization occurs at the end of the first for loop, to guarantee that the data needed by the loop over separator rows will be available.

While it is conceptually straightforward to effectively map this parallel computation to a NUMA machine, the details require some knowledge of low-level machine architecture. To make effective use of NUMA for this computation, we would ideally: (i) store the rows mapped to a processor in the memory local to that processor; (ii) avoid false sharing among computed results; and (iii) move as much data as possible in each cache line shared among processors.

Local mapping can be achieve by padding the data, which is perhaps most easily accomplished by adding an artificial row to the set of rows mapped to each processor (placing it after all domain and seperator rows). This row would not participate in any computations. The row length is chosen so that the data for the row following the artificial row (which is mapped to another processor) is page-aligned. This avoids the situation where a single page contains data that should be local to two different processors. Platform-specific directives are then used to map pages of data to appropriate processors.

False sharing issue is mostly circumvented by the multi-coloring and the one-way dissection. Since processors work on domains first, then separators, and since storage associated with each is contiguous after our initial permutation, it is unlikely that two processors will compute data that shares a cache line.

Regarding the issue of minimizing communication costs by maximizing spatial locality, note that the only interprocessor communication that occurs during a triangular solve is for the values computed from domain G_I and later used to compute values for separators V_{I-1} and V_I. Communication can be reduced by making all vertices i in G_I that are adjacent to vertices in V_{I-1} contiguous (and likewise for vertices adjacent to V_I). However, the penalties for remote access in a typical NUMA system are small enough that this optimization would probably not produce significant improvements in runtime. We have not tried it in our tests.

2.3 Results

Table 1 shows degradation in iteration counts for an SSOR preconditioned conjugate residual method on several large, symmetric matrices as the number of processors is increased (using a 10^{-6} reduction in residual as the convergence criteria). Note that the degradations due to parallelization are quite small.

Table 2 shows parallel performance on two 8 processor parallel systems. The first, an SGI Power Challenge, is a UMA system. The second, an SGI Origin 3000, is a NUMA system. We should make two points about the results. First, note that iterative methods place extremely heavy demands on the memory system, thus making parallelization difficult even under ideal conditions. Second, note that runtimes in all cases include the time required to compute the partitioning, permute the matrix, and unpermute the computed result vector. All these operations are performed sequentially, so reported speedups would have

Table 1: Iteration counts for parallel SSOR conjugate residual.

	Matrix Size		Iteration Counts			
Name	Rows	NZ	P=1	P=2	P=4	P=8
M1	47,360	628,676	670	667	702	723
M2	70,656	949,510	341	346	348	358
M3	123,440	1,605,669	473	475	480	489

Table 2: Parallel speedups on an SGI Power Challenge (PwrC) and an SGI Origin 3000 (O3000).

	P=2		P=4		P=8	
Name	PwrC	O3000	PwrC	O3000	PwrC	O3000
M1	1.8	1.9	4.0	3.7	6.9	5.3
M2	2.0	3.4	3.6	4.5	6.1	6.0
M3	1.8	1.6	3.1	3.0	5.0	4.3

been higher had we chosen not to include these overheads in our speedup computations. Speedups would also have been higher had we chosen a tighter convergence tolerance, since more time would have been spent in iterations.

Having said this, achieved parallel speedups are still quite high, on both the UMA and NUMA systems.

References

[1] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the 24th National Conference of the ACM, New York, 1969.

[2] S. Eisenstat, *Efficient implementation of a class of CG methods*, SIAM J. Sci. Stat. Comput., 2 (1981).

[3] A. George, *An automatic one-way dissection algorithm for irregular finite-element problem*, SIAM J. Numer. Anal., 19 (1980), pp. 740-751.

[4] Y. Saad and M. Sosonkina, *Enhanced Parallel Multicolor Preconditioning Techniques for Linear Systems*, in Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, Philadelphia, PA, 1999.

Appendix

The methods described in this paper are included in the SGI math library SCSL (version 1.4 or higher). This appendix gives documentation for the associated library routines.

Overview

This library implements four different parallel preconditioned iterative solvers: conjugate gradient and conjugate residual for symmetric systems, and conjugate gradient squared (CGS) and BiCGSTAB for unsymmetric systems. Four different types of preconditioners are available: Jacobi, symmetric successive over-relaxation (SSOR), ILDLT (incomplete LDLT) by pattern, and ILDLT by value. Note that the ILDLT preconditioners are currently only available for symmetric matrices, and ILDLT by value is currently not parallel.

Sparse matrix A must be input to the library in Compressed Sparse Column (CSC) or Compressed Sparse Row (CSR) format. The matrix is held in three arrays: pointers[], indices[], and values[]. In CSC format, the indices[] array contains the row indices of the non-zeros in A. The values[] array holds the corresponding non-zero values. The pointers[] array contains the index in indices[] for the first non-zero in each column of A.

Symmetric matrices may be input as either the lower or upper triangle of A, but not both.

To give an example, the following symmetric matrix...

```
( 1.0   symmetric )
( 0.0 3.0         )
( 2.0 0.0 5.0     )
( 0.0 4.0 0.0 6.0 )
```

would be represented in FORTRAN as follows:

```
pointers[] = (1, 3, 5, 6, 7)
indices[]  = (1, 3, 2, 4, 3, 4)
values[]   = (1.0, 2.0, 3.0, 4.0,
              5.0, 6.0)
```

Zero-based indexing is used in C, so the pointers[] and indices[] arrays would instead contain:

```
pointers[] = (0, 2, 4, 5, 6)
indices[]  = (0, 2, 1, 3, 2, 3)
```

Two additional routines control parameters for the 'ILDLT by value' preconditioner: Iterative_DropTol() allows the user to set the drop tolerance for the incomplete factorization, and Iterative_DropStorage() allows the user to control the amount storage used for the incomplete factor. The arguments are explained below.

To vary the number of processors used by the iterative solver, use the standard environment variables (MP_SET_NUMTHREADS or MPC_NUM_THREADS).

C Interface

```
void Iterative(
               int    n,
               int    *pointers,
               int    *indices,
               double *values,
               int    storage_format,
               double *x,
               double *b,
               int    method,
               int    preconditioner,
               int    max_iters,
               double convergence_tolerance,
               int    *iters,
               double *final_residual
              );

void Iterative_DropTol(
          double drop_tolerance
          );

void Iterative_DropStorage(
          double storage_multiplier
          );
```

FORTRAN Interface

```
SUBROUTINE Iterative(
          N, POINTERS, INDICES, VALUES,
          STORAGE_FORMAT, X, B,
          METHOD, PRECONDITIONER,
          MAX_ITERS,
          CONVERGENCE_TOLERANCE,
          ITERS, FINAL_RESIDUAL
          )
 INTEGER N, STORAGE_FORMAT, METHOD
 INTEGER PRECONDITIONER, MAX_ITERS
 INTEGER ITERS
 INTEGER POINTERS(*), INDICES(*)
 DOUBLE PRECISION VALUES(*), X(*), B(*)
 DOUBLE PRECISION CONVERGENCE_TOLERANCE
 DOUBLE PRECISION FINAL_RESIDUAL

SUBROUTINE Iterative_DropTol(
          DROP_TOLERANCE
          )
 DOUBLE PRECISION DROP_TOLERANCE

SUBROUTINE Iterative_DropStorage(
          STORAGE_MULTIPLIER
          )
 DOUBLE PRECISION STORAGE_MULTIPLIER
```

Arguments

n (input): The number of rows and columns in the matrix A (n ¿= 0).

pointers, indices, values (input): The pointers[] and indices[] arrays store the non-zero structure of sparse input matrix A in Compressed Sparse Column (CSC) format. The pointers[] array contains n+1 integers, where pointers[i] gives the index in the indices[] array for the first non-zero in column i of A. The indices[] array stores the row indices of the non-zeros in A. The values[] array stores the non-zero values in the matrix A.

storage_format (input): 0=stored by columns (CSC), 1=stored by rows (CSR).

x (input/output): initial/final solution vector

b (input): The right-hand-side vector.

method (input): The iterative solver to use: 0=conjugate gradient, 1=conjugate residual, 10=conjugate gradient squared, 11=BiCGSTAB. Methods 0 and 1 are used for symmetric matrices, and 10 and 11 are used for unsymmetric matrices.

preconditioner (input): The preconditioner to use: 0=Jacobi, 1=SSOR, 2=no-fill ICCG, 3=thresholded ICCG. Only preconditioners 0 and 1 are available for unsymmetric matrices.

max_iters (input): The solver terminates once it has performed max_iters iterations.

convergence_tolerance (input): The solver terminates when the norm of the residual, relative to the norm of the right-hand-side is less than convergence_tolerance.

iters (output): The actual number of iterations performed.

final_residual (output): The final 2-norm of the residual.

drop_tolerance (input): In thresholded factorization, an entry is discarded if it is smaller than drop_tolerance times the corresponding diagonal.

storage_multiplier (input): In thresholded factorization, the drop tolerance is automatically increased if the incomplete factor matrix contains more than storage_multiplier times the number of non-zero values in A.