Refining an Approximate Inverse*

Robert Bridson† Wei-Pai Tang†

Appeared in J. of Comp. and Appl. Math, 2000, vol. 123 (Numerical Analysis 2000 vol. III: Linear Algebra), pp. 293–306.

Abstract

Direct methods have made remarkable progress in the computational efficiency of factorization algorithms during the last three decades. The advances in graph theoretic algorithms have not received enough attention from the iterative methods community. For example, we demonstrate how symbolic factorization algorithms from direct methods can accelerate the computation of a factored approximate inverse preconditioner. For very sparse preconditioners, however, a reformulation of the algorithm with outer products can exploit even more zeros to good advantage. We also explore the possibilities of improving cache efficiency in the application of the preconditioner through reorderings. The article finishes by proposing a block version of the algorithm for further gains in efficiency and robustness.

Key words.

factored approximate inverse, implementation, symbolic factorization, ordering, block algorithms.

1 Introduction

So far research into sparse approximate inverse preconditioners has focused on convergence, ignoring more subtle efficiency issues for the most part. This paper explores how to get the best performance out of an approximate inverse preconditioner, particularly on modern superscalar workstations.

The algorithm we turn our attention to is Benzi and Tůma's AINV[1, 2], or more specifically, a slight variation on the stabilized version SAINV[4] that is guaranteed to avoid breakdown for positive defininte problems. We previously explored the issue of

*This work was supported by the Natural Sciences and Engineering Council of Canada, and Communications and Information Technology Ontario (CITO), funded by the Province of Ontario.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada. email: rebridso@elora.uwaterloo.ca, wptang@elora.uwaterloo.ca

ordering in [7], noting that for good orderings the set-up time for the preconditioner can be reduced dramatically. Here we go into details on that and other techniques for boosting the performance of the method. We note that in [1, 2], Benzi and Tůma had already implemented the symbolic factorization enhancement and outer-product form below, though don't discuss it in depth.

Before proceeding, we introduce some notation. Column i of a matrix A is written A_i (and so row i is $(A^T)_i^T$). The j'th entry of a vector v is given by v_j (and so the (i,j)'th entry of a matrix is indeed A_{ij}). The column vector of all zeros except for the one at the i'th position is e_i: e_i is thus column i of the identity matrix I. In algorithms, $x \leftarrow y$ indicates that variable x is assigned the value of y.

2 Basic Implementation

The simplest form of SAINV is a left-looking, inner product based algorithm, given in algorithm 1. It can be viewed as the generalization of classical Gram-Schmidt[1] to constructing A-biconjugate sets of vectors from the standard basis, with small entries dropped to preserve sparsity. The results are two upper-triangular matrices W and Z containing the sets of vectors as columns and a diagonal matrix D with $W^T AZ \approx D$. (In fact, with the presented choice of dropping, the diagonal of $W^T AZ$ is exactly D—it's just the off-diagonal terms that might not be zero.) When A is symmetric, the algorithm can be simplified by eliminating the W computations, using $W = Z$.

Of course the matrices all should be stored in sparse mode. For this article, compressed column storage format is assumed: each matrix is a collection of n sparse column vectors.

However, the inner products rZ_j and $W_j^T c$ are more efficiently computed if one of the vectors is stored in full mode; while a sparse-sparse operation could theoretically be faster, a typical implementation's more complicated branching and memory accesses make it slower on today's hardware. Since each r and c is reused over many inner iterations, it is natural to keep these in full storage—though of course, there is the drawback that often W and Z will be denser than A, so the inner products would be even more efficient with W_j and Z_j in full storage.

One n-vector suffices to store both r and c. To avoid unnecessary $O(n^2)$ work it shouldn't be completely zeroed out after use: only the nonzero locations should be reset. Further unnecessary work can be eliminated by only copying nonzeros up to position $i-1$, since W and Z are upper triangular and thus locations from i onwards will not be involved in the inner products.

With compressed column storage, accessing each column c is simple, but finding each row r is more time-consuming. In the symmetric case, this is of course unnecessary. Even if A just has symmetric structure, r can be found faster since not every

[1] SAINV in [4] is actually a generalization of modified Gram-Schmidt; this variation is a slightly faster but typically equal quality algorithm.

Algorithm 1: The left-looking, inner product form of SAINV.

- Take A, an $n \times n$ matrix, and some drop tolerance $\delta \geq 0$ as input.
- For $i = 1, \ldots, n$
  - ▷ *Initialize columns i of W and Z to the i'th standard basis vector*
  - Set $W_i \leftarrow e_i$ and $Z_i \leftarrow e_i$.
  - ▷ *Make column i of W biconjugate with previous columns*
  - Get row i of A: $r \leftarrow (A^T)_i^T = e_i^T A$.
  - For $j = 1, \ldots, i-1$
    - Set $W_i \leftarrow W_i - \frac{rZ_j}{D_{jj}} W_j$
  - ▷ *Make column i of Z biconjugate with previous columns*
  - Get column i of A: $c \leftarrow A_i = Ae_i$.
  - For $j = 1, \ldots, i-1$
    - Set $Z_i \leftarrow Z_i - \frac{W_j^T c}{D_{jj}} Z_j$
  - ▷ *Drop small entries to keep W and Z sparse*
  - Zero any above-diagonal entry of W_i or Z_i with magnitude $\leq \delta$.
  - ▷ *Find the "pivot" D_{ii}*
  - Set $D_{ii} \leftarrow W_i^T A Z_i$.
- Return W, Z, and D.

column of A need be checked for a nonzero at position i: only those columns corresponding to nonzeros in column i.[2]

The updates to W_i and Z_i require some thought, as they should be done in sparse mode; if constructed as dense vectors, there will be unnecessary $O(n)$ work in every iteration to gather them up into sparse storage. If the sparse columns are not kept in sorted order, the simplest way of adding the scaled W_j to W_i (or Z_j to Z_i) is to do a linear search in W_i for each nonzero in W_j; if there is a nonzero already in that location, add it to it, and otherwise append it. If the columns are sorted, then a faster merge operation may be used instead.

However, both of these methods require time dependent on the number of nonzeros already in W_i (some fraction of the elements in W_i will be scanned to determine where to add the update), which may grow with each inner iteration as updates are applied. A better implementation is described below, adding the scaled W_j to W_i in time just proportional to the number of nonzeros in W_j, independent of how many are already in W_i (avoiding any scan of existing elements).

Maintain two n-vectors, `exists` and `location`. The former is a Boolean vector with `exists`(k) true when W_i has a nonzero in position k; then `location`(k) points to where that nonzero is stored in the sparse data structure. Now adding an entry from the scaled W_j to W_i, say at position k, takes $O(1)$ time: look up `exists`(k); if true use `location`(k) to modify the existing entry in W_i, otherwise append the new entry to W_i. If the vectors must be stored in sorted order, after the inner loop W_i can be radix or bin-sorted very efficiently.

Of course, `exists` must be reset to all false before each inner loop. A cheap to avoid this cost is to let `exists`$(k) = i$ indicate true for W_i, and $n + i$ true for Z_i, on the i'th iteration.

The calculation of the pivot $W_i^T A Z_i$ is best done with W_i in full storage, viewing it as a sum of full-sparse inner products:

$$W_i^T A Z_i = \sum_{Z_{ji} \neq 0} (W_i^T A_j) Z_{ji}$$

Thus after small entries have been dropped, W_i should be scattered into a full n-vector, and after the pivot has been calculated, only those nonzeros reset.

3 Fruitless Inner Products

Even with good handling of the sparse vs. dense issues, the algorithm as it stands must take at least $O(n^2)$ time due to the nested loops. This can be improved significantly

[2] It is also possible to store a row-oriented copy of A along side the column-oriented version, as is done in [2]; for the scalar case here we have chosen not to, trading higher complexity for more lower storage requirements. Experiments indicate that typically the row-oriented copy is only worthwhile when A is not structurally symmetric, but then essentially the same performance can be obtained by adding zeros to symmetrize the structure, as will be discussed later.

Table 1: When SAINV with drop tolerance 0.1 is applied to several standard test matrices, almost all the inner products are exactly zero. The ordering in all cases is nested dissection.

Matrix	Total number of inner products	Number that are exactly zero	Percentage of total
ADD32	24,596,640	24,580,218	99.9%
BCSSTK25	238,347,282	237,781,980	99.8%
MEMPLUS	315,328,806	315,240,589	99.97%
NASA2146	4,603,170	4,464,828	97.0%
ORSREG1	4,859,820	4,838,964	99.6%
PORES2	1,496,952	1,484,637	99.2%
SHERMAN2	1,165,320	1,118,651	96.0%
SHERMAN3	25,045,020	25,013,829	99.9%
WATSON5	3,431,756	3,421,143	99.7%

after realizing that often many of the inner j iterations are unnecessary: the inner products rZ_j and $W_j^T c$ are often zero simply because there are no nonzero locations in common.

Table 1 shows some sample statistics of how many inner products turn out to be exactly zero in the preconditioner construction for some typical test matrices, symmetrically ordered with the nested dissection routine from Metis[14].[3] This does not include the small fraction of inner products from the pivot calculation $W_i^T A Z_i$.

Fortunately, many of these inner products can be avoided. We begin by considering those inner products which are zero even without small entries dropped in the algorithm, i.e. when the true inverse factors are computed. Because the algorithm doesn't rely on cancellation anywhere, dropping can only result in more zero dot-products—thus we are always safe to avoid the ones that are zero without dropping.

First consider the case when A has symmetric structure, so the true inverse factors have the same structure as each other. Then we have the following result:

Theorem 3.1 *Assuming symmetric structure, at step i with r equal to the i'th row of A, the inner product $rZ_j \neq 0$ only if $j < i$ and j is an ancestor in the elimination tree[16] of some k with $A_{ik} \neq 0$.*

Proof: *In [7] the structure of the true inverse factors, assuming no felicitous cancellation, was shown: $Z_{kj} \neq 0$ if and only if k is a descendent of j in the elimination tree of A. The inner product rZ_j is nonzero if and only if there is some k with $A_{ik} \neq 0$ and $Z_{kj} \neq 0$. Therefore the inner product is nonzero only when there is some k with $A_{ik} \neq 0$ and k a descendent of j, i.e. j an ancestor of k. Only values $j < i$ are considered in the original loop, and so the result follows.*

[3] In [3, 7] other orderings were considered for AINV, but as nested dissection is generally close to best in convergence, often best in construction time, and most easily parallelized, this article sticks with just nested dissection. Results for other inverse factor fill reducing orderings are similar.

Another proof of this result can be made from the factorization $A = LDU$ (with L unit lower triangular, U unit upper triangular, and D diagonal), so $Z = U^{-1}$ and thus $AZ = LD$. Then the inner product rZ_j at step i is simply $L_{ij}D_{jj}$, and the nonzero structure of each row of L has been characterized precisely as above in [15]. The only difficulty with this route is determining what role cancellation plays in the structure of AZ—with inexact arithmetic and especially with dropping, it's not immediately clear that the structure of the lower triangle of AZ will be a subset of the structure of L.

In [15] a very efficient algorithm is given for finding the elimination tree of A, leading to a fast symbolic factorization. We can use this to create a symbolic factorization enhanced AINV, replacing the inner $j = 1 \ldots i-1$ loop with one just over the nonzeros in row i of L. Of course, taking note of the symmetric structure and column-oriented storage of A, the upwards-traversals of the elimination tree to find those indices should start with the nonzeros in column i of A with indices less than i.

When A doesn't have symmetric structure, things get a little more complicated. Often A is close to structurally symmetric and so ordering, symbolic factorization, and biconjugation can all be done efficiently with zeros inserted into the sparsity structure to make it symmetric. However, there may be cases when it is best to exploit the non-symmetric zeros in any or all of these steps. (For example, it may be possible to exploit unsymmetric zeros in ordering to reduce the matrix to block triangular form, in which case only smaller submatrices need be preconditioned.) Here we will consider an unsymmetric symbolic factorization enhancement.

The key again is the structure of the true inverse factors. This is most easily discussed with the language of graph theory, where the nonzero structure of an $n \times n$ matrix M corresponds to a graph G_M with vertices labelled $1, \ldots, n$ and directed edge $i \to j$ if and only if $M_{ij} \neq 0$. See [10], for example, for more discussion of graph theory and sparse matrix computations.

As proven in [12], the inverse of a matrix M has the structure of the transitive closure G_M^* of G_M, that is a graph G_M^* with a directed edge $i \to j$ whenever there is a path from i to j in G_M. The simplest characterization of the structure of the true inverse factors $W^T = L^{-1}$ and $Z = U^{-1}$ is then as the transitive closures of the graphs of L and U respectively. However, there are many unnecessary edges in G_L and G_U from this standpoint—if an edge $i \to j$ exists alongside a disjoint path from i to j, the edge $i \to j$ may be deleted without effecting the transitive closure. If all such redundant edges are deleted, the result is called the transitive reduction. If A was structurally symmetric, this turns out to be the elimination tree mentioned above[16]; otherwise G_L and G_U reduce to a pair of directed acyclic graphs called elimination dags[11].

Unfortunately the elimination dags can be fairly expensive to compute, and so somewhat denser but cheaper graphs, intermediate between the triangular factors and their transitive reductions, have been investigated in [9]. For this application, an alternative route is to use graphs whose transitive closures contain the structures of W^T and Z but may be a little denser still—for example, the elimination tree of the symmetrized A. With these cases in mind, the unsymmetric generalization of the previous theorem is:

Theorem 3.2 *Let G_L° and G_U° be directed acyclic graphs whose transitive closures contain the structures of W^T and Z respectively. Then at step i of AINV, the inner product $rZ_j \neq 0$ only if $j < i$ and there is a path in G_U° to j from some k with $A_{ik} \neq 0$; similarly the inner product $W_j^T c \neq 0$ only if $j < i$ and j is reachable in G_L° from some k with $A_{ki} \neq 0$.*

Proof: *We will only prove the rZ_j part, as the $W_j^T c$ part is essentially the same. Since the transitive closure of G_U° contains the structure of Z, $Z_{kj} \neq 0$ only if there is a path in G_U° from k to j. The inner product $rZ_j \neq 0$ if and only if there is some k with $A_{ik} \neq 0$ and $Z_{kj} \neq 0$. Therefore the inner product is nonzero only if there is some k with $A_{ik} \neq 0$ and with a path to j in G_U°. Only values $j < i$ are considered in the original loop, and so the result follows.*

Just as before, this can be interpreted as symbolic factorization, if G_L° and G_U° are chosen to be the elimination dags or other intermediate structures between the elimination dags and the triangular factors. For example, the inner product rZ_j at step i is just $L_{ij}D_{jj}$, and the above characterization is the same as that shown for the rows of L in [11, 9].

Table 2 compares the regular form of AINV with the symbolic factorization enhanced version, with a drop tolerance of 0.1 for each test matrix as before. The timing counts are from a C implementation running on an Apple Macintosh workstation with a 233MHz PowerPC 750 processor. For the matrices with non-symmetric structure an elimination dag version is tested first, followed by a symmetrized version. Even without the time required for finding the elimination dags taken into account, and even though more unnecessary zero inner products are performed, the symmetrized version is clearly much faster for these typical matrices. In all cases, the enhanced algorithm is significantly faster than the original algorithm, often by an order of magnitude or more.

For a successful ordering, the number of nonzeros in the LDU factors, hence the number of inner products in the symbolic factorization enhanced algorithm, is an order of magnitude less than $O(n^2)$ (e.g. see [13] for guarantees on two-dimensional finite element meshes). Assuming that the average cost of an inner product in the regular and the enhanced algorithms is the same—which is probably not strictly true, but still is a good rough estimate—this explains why the enhanced version is so much faster.

4 Revisiting the Outer-Product Form

The symbolic factorization enhancement may avoid all inner products that can be determined zero *a priori*. However, there are still more that result from the nonzeros that are dropped during the algorithm. Possibly these could be avoided by pruning the elimination structures as the algorithm goes, but a simpler approach is to rewrite SAINV as a right-looking outer product algorithm by switching the order of the loops. With the obvious sparsity enhancement, the result is given in algorithm 2.

In exact arithmetic without dropping, the vectors l and u at step j are the j'th colum and row of LD and DU respectively. With dropping, they naturally become sparser,

Table 2: A comparison of regular and symbolic factorization enhanced SAINV on some standard test matrices. The matrices marked as "symmetrized" had zeros inserted in their sparsity structure to make them structurally symmetric, albeit not numerically symmetric.

	Millions of inner products		Percentage of zero inner products		Seconds spent on AINV	
Matrix	Regular	Enhanced	Regular	Enhanced	Regular	Enhanced
ADD32	25	0.02	99.9%	15.5%	15.2	0.04
BCSSTK25	238	1.57	99.8%	82.0%	730	6.8
MEMPLUS	315	0.11	99.97%	18.9%	310	44.5
(symmetrized)		0.11		21.4%	260	0.42
NASA2146	4.6	0.14	97.0%	50.2%	7.7	0.31
ORSREG1	4.9	0.17	99.6%	87.6%	3.4	0.20
PORES2	1.5	0.06	99.2%	78.4%	1.2	0.28
(symmetrized)		0.09		86.0%	1.1	0.07
SHERMAN2	1.2	0.12	96.0%	60.3%	1.8	0.71
(symmetrized)		0.16		70.1%	1.4	0.50
SHERMAN3	25	0.20	99.9%	84.3%	18.1	0.34
WATSON5	3.4	0.02	99.7%	50.8%	3.7	0.46
(symmetrized)		0.09		88.8%	3.7	0.08

Algorithm 2: The outer product form of SAINV.

- Take as input A and δ.
- Set $W \leftarrow I$ and $Z \leftarrow I$.
- For $j = 1, \ldots, n$
  - Set $l \leftarrow AZ_j$
  - Set $u \leftarrow W_j^T A$
  - Set $D_{jj} \leftarrow uZ_j$ or equivalently $W_j^T l$, whichever is cheapest
  - For $i > j$, $l_i \neq 0$
    - Update $W_i \leftarrow W_i - \text{drop}\left(\frac{l_i}{D_{jj}} W_j, \delta\right)$, where entries of the update vector with magnitude $\leq \delta$ are dropped.
  - For $i > j$, $u_i \neq 0$
    - Update $Z_i \leftarrow Z_i - \text{drop}\left(\frac{u_i}{D_{jj}} Z_j, \delta\right)$.
- Return W, Z, and D.

giving the improvement over the symbolic factorization enhanced inner product algorithm.

Note that because small entries are dropped before being added in this formulation, the result will in general be different from the inner product version. Usually the same drop tolerance will produce a sparser but less accurate preconditioner than the inner product form.

The primary drawback of the outer product form is its right-looking nature: all of columns $j+1, \ldots, n$ of W and Z must be stored in dynamic data structures, since updates to them may insert entries in any row up to j. The natural implementation with each column of W and Z in a sorted linked list then can suffer from inefficient insertions, poor cache usage, and difficulties for vectorization.[4] However, the savings from exploiting the dropped zeros hopefully can make up for this.

Just as with the inner product form, there is a difficulty when A doesn't have symmetric structure and a row-oriented copy is not available: the left-multiplication $W_j^T A$ cannot be made in an efficient fully sparse mode. All entries must be computed, even though most will be zero. One possibility to speed this up is to use a similar symbolic factorization approach as before, making use of a characterization of the columns of L (rather than the rows) to *a priori* eliminate most of the zero computations. However, this would lose the motivation for the outer product form—exploiting the zeros that cannot be determined *a priori*—while still incurring the dynamic data structure penalties. Therefore we have chosen to symmetrize the sparse matrix data structure as before.

A timing comparison between the inner product and outer product algorithms is given in table 3. Since the same drop tolerance of 0.1 produces slightly different factors for the outer product form than for the inner product form, I have chosen new drop tolerances for the outer product tests to give it roughly the same number of nonzeros.

As the results show, while for some problems the extra overhead of outer product SAINV isn't worth the small gain made from exploiting the full sparsity, in several cases the benefit is considerable.

With these results in mind the choice of algorithm depends on several factors:

- How much storage is available? Enough for the overhead of the dynamic data structures in the outer product form? Enough for an additional row-oriented copy of A?
- Approximately how full will the factors be? Full enough that there will be so few zero inner products that inner product AINV is faster?
- Is A so far from structurally symmetric that it pays to exploit the unsymmetric zeros in some way? (e.g. using unsymmetric elimination structures for inner-product SAINV rather than the elimination tree of the symmetrized matrix)

[4] More sophisticated data structures such as B-trees may do better in some cases—normally though, the number of nonzeros in each column of W and Z is so small that the increased overhead is not worth it.

Table 3: Timing comparison for inner product SAINV versus outer product SAINV. Matrices with unsymmetric structure are symmetrized with additional zeros.

Matrix	Time for inner product form	Time for outer product form
ADD32	0.04	0.06
BCSSTK25	6.75	0.97
MEMPLUS	0.42	0.55
NASA2146	0.31	0.17
ORSREG1	0.20	0.06
PORES2	0.07	0.04
SHERMAN2	0.50	0.44
SHERMAN3	0.34	0.10
WATSON5	0.08	0.11

It should be noted also that for some problems, A is known only as an operator or as a product of matrices, not in explicit matrix form. In this case, finding elimination structures for A may be impossible, prompting the choice of the outer product form which doesn't require them (see [6] for example).

5 Ordering for Application

Forgetting the trivial diagonal matrix D for the time being, the basic operation in an iterative solver is applying the preconditioned operator to a dense vector: $W^T AZx$. Algorithm 3 shows the simplest algorithm for doing this with compressed column storage.

One major issue in the speed of this algorithm on modern superscalar processors comes from the memory hierarchy: efficient cache usage. For example, in the first main loop (multiplying $u = Zx$) each entry of u may be accessed several times according to the structure of Z. The more cache misses there are—the more times an entry of u has to be fetched from main memory—the slower the loop will run. Ideally, once an entry from u is fetched from cache it will stay there until done with, and won't be prematurely bumped out of the cache. The situation is complicated by how entire cache "lines" of consecutive memory locations are brought into cache at each miss—typically on the order of 64 bytes.

One of the advantages of approximate inverses is that any orderings may be used in the matrix-vector multiplies—the rows and columns of the matrices and vectors may be permuted without effecting the result, modulo finite precision arithmetic errors, with the only restriction coming from the compatibility of the orderings in multiplication (e.g. the ordering of x must be the same as the columns of Z). With detailed knowledge of the hardware hopefully this can be exploited to promote efficient cache usage in the multiplies. Such tuning is beyond the scope of this article, but some simple tests can

Algorithm 3: Multiplying $W^T AZx$.

- Take as input sparse matrices W, A, and Z (in compressed column format) and a dense vector x.
- Initialize dense vectors $u = 0$ and $v = 0$
- For $i = 1, \ldots, n$
  - For j with $Z_{ji} \neq 0$
    - Update $u_j \leftarrow u_j + Z_{ji} x_i$
- For $i = 1, \ldots, n$
  - For j with $A_{ji} \neq 0$
    - Update $v_j \leftarrow v_j + A_{ji} u_i$
- For $i = 1, \ldots, n$
  - Set $u_i \leftarrow 0$
  - For j with $W_{ji} \neq 0$
    - Update $u_i \leftarrow u_i + W_{ji} v_j$
- Return the result in u.

show the potential effect of ordering for application. We hope to raise questions here, rather than provide answers.

Table 4 compares the performance for random orderings, the nested dissection ordering used in construction of the preconditioner, and a reordering of the elimination tree for the nested dissection ordering starting at the leaves and progressing upwards level by level. This last ordering is an entirely equivalent elimination sequence to the nested dissection, but mimics the greedy choices made by minimum degree or MIP[7].

The differences in performance, at least for ADD32, BCSSTK25, MEMPLUS, and SHERMAN3, highlight how important ordering might be here. Random ordering is clearly bad—indicating for example that unstructured meshes created with no natural ordering should be appropriately reordered for iterations. The standard nested dissection is generally better than the elimination tree equivalent leaf reordering, perhaps indicating that if minimum degree or MIP is used for construction a further reordering is necessary. We believe the reason for these differences is that standard nested dissection tends to cluster most of the nonzeros in small blocks (excepting the large block separators), which intuitively will allow efficient cache usage. We note that other factors may be involved, such as how fully used are multiple instruction pipelines, but for the moment we don't see a reason they would have this effect; we refer the reader to [8] for a full discussion of all the factors involved in tuning a (dense) matrix-multiplication routine.

The question remains whether there are significantly superior orderings to standard nested dissection. The following theorem suggests that for fairly full approximate inverses, the nested dissection ordering could well be the best. In general, for symmetrically structured matrices, a post-ordering of the elimination tree[10] is a natural

Table 4: The number of milliseconds taken to compute $W^T AZx$ for various orderings of the matrices and x. W and Z are computed from inner product SAINV with a drop tolerance of 0.1 and the nested dissection ordering. The leaf reordering is an equivalent elimination sequence to the nested dissection, but begins with all the leaves of the elimination tree and progresses upwards level by level, mimicking minimum degree and MIP to some extent.

	Ordering		
Matrix	Random	Nested Dissection	Leaf Reordering
ADD32	5.6	4.7	4.8
BCSSTK25	102.3	70.6	91.6
MEMPLUS	33.6	27.4	28.6
NASA2146	13.0	12.4	12.7
ORSREG1	2.3	2.6	2.2
PORES2	1.3	1.3	1.3
SHERMAN2	4.2	4.3	4.4
SHERMAN3	7.9	6.6	6.9
WATSON5	3.7	3.4	3.8

generalization of the nested dissection ordering even when the ordering was not constructed in that manner.

Theorem 5.1 *For symmetrically structured A with a post-ordering of the elimination tree[10], the true upper triangular inverse factor has a dense skyline. In other words, its columns consist of a block of zeros followed by a single dense block of nonzeros ending at the diagonal.*

Proof: *The key characterization of a post-ordering is that any subtree is ordered in a contiguous block, with the root of the subtree coming last. The nonzeros in column i of the true upper triangular inverse factor correspond to all children of i in the etree, i.e. to the others nodes in the subtree rooted at i. Thus the nonzeros form one contiguous block, ending at the diagonal (the i'th row).*

This simple block structure is near optimal for cache use within each column, though the question of the order in which the columns should be considered is still open.

6 Block Methods

As with direct methods, the eventual goal of the algorithms should be to cut the symbolic operations to a minimum while doing the necessary numerical operations efficiently in cache. We have shown ways to eliminate unnecessary numerical operations in the preconditioner construction, and the possibility of promoting cache usage in the application. For further improvements we now turn to block methods to cut down symbolic operations and further cache efficiency.

Algorithm 4: The left-looking, inner product form of block SAINV with symbolic factorization enhancement.

- Take A, an $m \times m$ block matrix, and some drop tolerance $\delta \geq 0$ as input.
- For $i = 1, \ldots, m$
  - ▷ *Initialize block columns i of W and Z to the i'th standard basis block vector*
  - Set $W_i \leftarrow E_i$ and $Z_i \leftarrow E_i$.
  - Get block row i of A: $R \leftarrow (A^T)_i^T = E_i^T A$ (up to column $i-1$)
  - For $j < i$, $U_{ji} \neq 0$ (determined by symbolic factorization)
    - $W_i \leftarrow W_i - W_j (R Z_j D_{jj}^{-1})^T$
  - Get block column i of A: $C \leftarrow A_i = AE_i$ (up to row $i-1$)
  - For $j < i$, $L_{ij} \neq 0$ (determined by symbolic factorization)
    - $Z_i \leftarrow Z_i - Z_j (D_{jj}^{-1} W_j^T C)$
  - Zero any above-diagonal block of W_i or Z_i with norm $\leq \delta$.
  - Set $D_{ii} \leftarrow W_i^T A Z_i$ (and store D_{ii}^{-1}).
- Return W, Z, and D.

This approach, partitioning A into small dense block matrices, is used to great advantage in direct methods, where for example supernodes[17] are used to eliminate redundant symbolic operations. There are also many problems, e.g. from systems of PDE's, that naturally have a block structure and it sometimes makes sense to treat them as such: convergence may sometimes be improved, as shown below.

The generalization of SAINV to block matrices is straightforward. We redefine our notation somewhat for block structures. Throughout we assume that A and all other matrices have been partitioned with $1 = b_1 < b_2 < \cdots < b_{m+1} = n + 1$. Then A_i indicates block column i of A, consisting of the "point" columns b_i to $b_{i+1} - 1$, and A_{ij} indicates the j'th block in this block vector, the submatrix of A extending from position (b_i, b_j) to $(b_{i+1} - 1, b_{j+1} - 1)$. The i'th block column of the identity is given by E_i. Notice that diagonal blocks of a matrix are necessarily square, but off-diagonal blocks might not be if the block size isn't constant.

Block SAINV produces matrices W, Z, and D that approximately satisfy $W^T AZ = D$, where W and Z are block upper triangular and D is block diagonal. The inner product form is given in algorithm 4; for this paper we don't explore the performance of the somewhat more complicated outer product form. The generalization of the scalar outer product algorithm is straightforward nonetheless.

The symbolic factorization enhancement now must use the graph of the block form of A, where each vertex represents a diagonal block and each edge a nonzero off-diagonal block. Also notice that since the storage requirements of a sparse block matrix is strongly dominated by the numerical entries in the dense blocks, it is perfectly reasonable to store a row-oriented version of the sparsity structure (referencing the same

numerical entries) along side the column-oriented version—so finding block rows of unsymmetric A can be done with ease.

Determining when to drop "small" blocks from W and Z is an interesting issue, especially as one drop tolerance is used for blocks of potentially different sizes. One possibility, used here, is to compare the Frobenius norm of the block divided by the number of entries in the block against the drop tolerance δ.

In the scalar case it is possible that a pivot will be zero (or small enough to cause problems when dividing by the pivot later). This problem is alleviated somewhat with the block algorithm, since the inversion of the block pivots can be carried out more robustly with a partial pivoting LU decomposition, a QR decomposition, or even an SVD operation to be completely confident of numerical stability. However, it still may happen that a block pivot is singular or so close to singular that problems emerge. Our implementation currently only checks for exact zeros in the partial pivoting LU decomposition; this never happenned in the testing however. Several possibilities exist for recovery if this does happen—adding a diagonal shift to the block, for example, or using a shifted SVD instead.

Some of the test problems have a natural block structure while it isn't so clear for others. One possibility is to use the supernodes following a nested dissection ordering, hoping that since the nodes making up a supernode have essentially the same structure in the true inverse factors, they should have similar structure in the approximate inverse and thus be correctly handled with dense blocks. The problem is that usually many supernodes are singletons, so unless care is taken in coding the algorithm, the overhead of the block algorithm is wasted. It is also important to note that there are typically some supernodes of very large size, which must be broken up into more manageable sizes for storage and computational efficiency.

Perhaps a better approach is to use the aggregation algorithms of algebraic multigrid. Here we tried applying the ideas from [5] (using $|A| + |A^T|$ for the unsymmetric matrices).

Table 5 shows construction times and convergence rates for scalar inner product AINV and block inner product AINV, with the same drop tolerance of 0.1 as before. For these examples, the block form is always slower—the overhead simply isn't worth any gains in dense operations. It should be noted that the BLAS and LAPACK libraries used for these timings were not highly tuned, however, so better results are definitely anticipated in better implementations. The convergence is generally worse for the block method, presumably because the block version may drop important nonzeros in otherwise near zero blocks while retaining unimportant nonzeros that happen to occur in the same blocks as important nonzeros. The exception is SHERMAN2, where as suggested in [7] the block form succeeds but the scalar form fails.

It seems then that the block version might only be appropriate in certain cases, unless a better determination of blocks and a more sophisticated dropping strategy are adopted. For example, the improvement for SHERMAN2 over the scalar version is probably because the scalar version's simple diagonal pivoting is inappropriate—with weak diagonals and condition numbers ranging from 10^7 to 10^{11}, the diagonal blocks require partial pivoting to be inverted. (For the other matrices, the diagonal

Table 5: A comparison of preconditioner construction times and convergence rates. The drop tolerance for scalar inner product AINV is 0.1, and the drop tolerance for the block version is chosen to give approximately the same number of nonzeros. CG is used for s.p.d. problems and BiCGstab for the rest; the right-hand side is the vector of all ones, the initial guess is all zeros, and convergence is flagged when the residual 2-norm is decreased by a factor of 10^6.

	Scalar		Block		
Matrix	Time for AINV	Iterations	Average block size	Time for AINV	Iterations
ADD32	0.04	5	2	0.10	32
BCSSTK25	6.75	∞	3.0	7.88	∞
MEMPLUS	0.42	17	2	0.58	∞
NASA2146	0.31	85	3.9	0.30	135
ORSREG1	0.20	31	2.5	0.31	46
PORES2	0.07	∞	2	0.20	∞
SHERMAN2	0.50	∞	6	0.57	21
SHERMAN3	0.34	96	1.7	1.13	127
WATSON5	0.08	127	2.7	0.13	∞

blocks aren't nearly as badly conditioned.) Of course, this raises the question whether a simple block diagonal rescaling applied before scalar AINV would be enough to cure the problem.

7 Conclusions

We have presented several refinements for improving the performance of the SAINV factored approximate inverse. Ideas and algorithms from direct methods allowed significant performance enhancements for the inner product form of the algorithm; for many problems, however, even faster construction was possible with an outer product reformulation. Experimental results demonstrated how reordering the approximate inverse can greatly effect the cache efficiency during its application in an iterative solver. We finally proposed a block version of the algorithm for further gains in cache efficiency, which unfortunately are offset by increased overhead in the current implementation—we expect further tuning of the code will make block processing worthwhile. The block version can give better convergence for some badly conditioned block-structured problems thanks to its better treatment of pivots, but for other matrices appears to be less robust since the block-by-block dropping is more likely to make bad choices. More sophisticated ideas from algebraic multigrid for finding better block structures may alleviate this difficulty, as might better dropping strategies than the current *ad hoc* choice.

The underlying theme to this research is that significant gains can be made for iterative solvers by considering the techniques designed originally for direct solvers. Progress towards high performance iterative methods requires solving many of the al-

gorithmic problems that have confronted the direct methods community; the solutions developed there, tempered with knowledge of iterative approaches, are bound to be valuable.

References

[1] M. Benzi, C. Meyer, and M. Tůma, *A sparse approximate inverse preconditioner for the conjugate gradient method*, SIAM J. Sci. Comput., 17 (1996), no. 5, pp. 1135–1149.

[2] M. Benzi and M. Tůma, *A sparse approximate inverse preconditioner for nonsymmetric linear systems*, SIAM J. of Sci. Comput., 19 (1998), no. 3, pp. 968–994.

[3] M. Benzi and M. Tůma, *Orderings for factorized sparse approximate inverse preconditioners*. To appear in SIAM J. of Sci. Comput. (Revised version of Los Alamos National Laboratory Technical Report LA-UR-98-2175, May 1998)

[4] M. Benzi, J. Cullum, and M. Tůma, *Robust approximate inverse preconditioning for the conjugate gradient method*, submitted to SIAM J. Sci. Comput.

[5] M. Brezina, J. Mandel, and P. Vaněk, *Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems*, Computing, 56 (1996), pp. 179–196.

[6] R. Bridson, *Multi-resolution approximate inverses*, Masters thesis, Dept. of Computer Science, University of Waterloo, 1999.

[7] R. Bridson and W.-P. Tang, *Ordering, Anisotropy and Factored Sparse Approximate Inverses*, to be published in SIAM J. Sci. Comput. (1999)

[8] J. Dongarra and R. Whaley, *Automatically Tuned Linear Algebra Software (ATLAS)*, SC98: High Performance Networking and Computing Conference, electronic proceedings.

[9] S. Eisenstat and J. Liu, *Exploiting structural symmetry in unsymmetric sparse symbolic factorization*, SIAM J. Matrix Anal. Appl., 13 (1992), no. 1, pp. 202–211.

[10] A. George and J. Liu, *Computer solution of large sparse positive definite systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[11] J. Gilbert and J. Liu, *Elimination structures for unsymmetric sparse LU factors*, SIAM J. Matrix Anal. Appl., 14 (1993), no. 2, pp. 334–352.

[12] J. Gilbert, *Predicting structure in sparse matrix computations*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 62–79.

[13] R. Lipton, D. Rose, and R. Tarjan, *Generalized nested dissection*, SIAM J. Numer. Anal., 16 (1979), pp. 346–358.

[14] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1999), no. 1, 359–392 (electronic).

[15] J. Liu, *A compact row storage scheme for Cholesky factors*, ACM Transactions on Mathematical Software, 12 (1986), no. 2, pp. 127–148.

[16] J. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[17] J. Liu, E. Ng, and B. Peyton, *On finding supernodes for sparse matrix computations*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 242–252.