UNIVERSITÄT KARLSRUHE

Parallel Algorithms for Solving Linear Systems with Sparse Triangular Matrices

Jan Mayer

Preprint Nr. 08/07

Institut für Wissenschaftliches Rechnen
und Mathematische Modellbildung

76128 Karlsruhe

Anschrift des Verfassers:

Dr. Jan Mayer
Institut für Angewandte und Numerische Mathematik
Universität Karlsruhe (TH)
D-76128 Karlsruhe

Parallel Algorithms for Solving Linear Systems with Sparse Triangular Matrices

Jan Mayer
Institut für Angewandte und Numerische Mathematik
Universität Karlsruhe

Abstract

In this article, we present two new algorithms for solving given triangular systems in parallel on a shared memory architecture. Multilevel incomplete LU factorization based preconditioners, which have been very successful for solving linear systems iteratively, require these triangular solves. Hence, the algorithms presented here can be seen as parallelizing the application of these preconditioners.

The first algorithm solves the triangular matrix by block anti-diagonals. The drawback of this approach is that it can be difficult to choose an appropriate block structure. On the other hand, if a good block partition can be found, this algorithm can be quite effective. The second algorithm takes a hybrid approach by solving the triangular system by block columns and anti-diagonals. It is usually as effective as the first algorithm, but the block structure can be chosen in a nearly optimal manner.

Although numerical results indicate that the speed-up can be fairly good, systems with matrices having a strong diagonal structure or narrow bandwidth cannot be solved effectively in parallel. Hence, for these matrices, the results are disappointing. On the other hand, the results are better for matrices having a more uniform distribution of non-zero elements.

Although not discussed in detail, these algorithms may also be suitable for a distributed memory architecture.

Key words: preconditioning, iterative methods, sparse linear systems, parallelization.

AMS subject classification: 65F10, 65F50, 65Y05.

1 Introduction

In this article, we introduce two new parallel algorithms for solving sparse triangular systems intended for use on a desktop PC or workstation having a shared memory architecture. A large number of such triangular systems need to be solved when using incomplete LU-factorization based preconditioners as part of an iterative method for solving large, sparse linear systems of equations. In fact, this is the main application area that we have in mind when developing the algorithms.

When preconditioning in a parallel environment, one would ideally obtain parallel linear systems by splitting the large linear systems into smaller ones. Domain decomposition methods, for example, go in this direction, but they are certainly not applicable to all problems. The second best approach is to parallelize as much as possible the algorithm for solving the (single) linear system. Generally, these approaches make use of the special structure of the matrices or preprocess the matrix to obtain a structure more suitable for parallelization. In particular, when using an incomplete LU factorization based preconditioner in a parallel environment, it would be ideal to reorder rows and columns of the coefficient matrix to reduce fill-in during factorization and to allow for the best possible use of parallelism both during the factorization and solve phase. However, a significant number of difficult problems require either reordering prior to factorization and/or pivoting during factorization to increase robustness and to avoid small pivots, see e.g. [1], [6], [7], [12], [15] for reordering and [2], [10] for pivoting strategies. Thus, reordering such problems to allow for better parallelism may not be an option. Furthermore, even when efficient parallel algorithms for computing incomplete factorizations and for performing the corresponding triangular solves are conceivable and feasible, such algorithms or their implementation in software may not (yet) exist. This is certainly the case for many multilevel incomplete LU-based preconditioners designed for relatively unstructured sparse matrices, such as those implemented in [3] or [11]. However, common to either scenario is that we need to solve linear systems with a *given* sparse lower triangular matrix L and sparse upper triangular matrix U, usually both with a unit diagonal as efficiently as possible.

Hence, the goal of this article is to provide algorithms for the triangular solves that are *as parallel as possible*. In other words, given that the only option to solve some triangular linear systems is to use the usual sequential algorithm and given that most current PCs and workstations have at least a single dual core processor (and often quite more), having one processor doing much of the work and having the other processors doing anything useful is certainly an improvement over having one processor doing all the work and having the other processors doing nothing. With this approach, we cannot expect the speed-up or efficiency that we would usually want from parallel algorithms. However, any speed-up is better than none, as in most situations we have nothing to gain from having idle processors. On the other hand, the big advantage of an algorithm developed in this context is that it is relatively universal. All incomplete LU-factorization based preconditioners result in sparse factors L and U so that having a *single* parallel algorithm that can be used for all sparse triangular matrices is certainly attractive and an improvement over the current situation, where sequential algorithms are used in many cases.

The idea behind the first parallel algorithm which we present in this paper is to split L (or U) into blocks by both rows and columns. Note that the partitioning that leads to the block structure can be non-uniform and different for the rows and columns, providing for significant flexibility. In doing triangular solves $Lx = y$, different blocks on the same anti-diagonal of L access different sections of x and y, allowing a fixed block anti-diagonal to be processed in parallel. Although this algorithm can be quite effective, selecting the appropriate block structure is not straightforward and indeed for an arbitrary matrix it is probably impossible in practice to find the best block structure.

The second algorithm divides the coefficient matrix into two different types of regions, each of which can be processed in parallel. We refer to it as the hybrid algorithm. The first region type consists of a diagonal block and part of a block column, the block column beginning on the same block anti-diagonal as the diagonal block. The second region type is just a single block whose rows can be processed in parallel. Although dividing the matrix in such regions appears to be quite unnatural, selecting the actual blocks structure can be done optimally in a particular sense. Usually, the performance of this algorithm is comparable to the first algorithm, but unlike the first algorithm, the block structure of the matrix is determined automatically.

Although we also considered other options for parallel solving, such as processing L by block rows or columns, the results were worse in all cases tested. We will mention these options in the sequel mainly because they provide the motivation for considering other algorithms, but we will not go into details.

As indicated above, the speed-up and efficiency is not as good as what we would expect for algorithms parallelized for more specific problems. In particular, the algorithms presented require significant communication which becomes a bottleneck as the number of threads increases. As this is a common problem for parallel algorithms, this drawback may become less significant on future architectures. Currently, however, it seems that using more than 4 threads is often counterproductive, actually increasing computation time for some matrices. Nevertheless, for many desktop PCs, even this modest degree of parallelization is an improvement. Additionally, using a parallel algorithm for only a few threads may be attractive if this approach can be combined with a procedure which reduces a large linear system to numerous smaller ones, e.g. domain decomposition methods.

In addition to the drawbacks mentioned above, it should be pointed out that the parallelization is not efficient for triangular matrices having a (tight) band structure and/or for extremely sparse matrices. In particular, if L is a block diagonal matrix, then the blocks can be solved in parallel, but these algorithm will not find or use this structure. Given that parallelization is straightforward in this case and that such a block structure is unlikely to occur coincidentally, this drawback is not very serious as the appropriate algorithm can be used easily. The worst-case situation, however, is a lower triangular bidiagonal matrix, for which the solve cannot be parallelized at all. However, this is not the fault of any algorithm as solving such matrices is inherently sequential. Hence, dropping entries on the off-diagonal to create a diagonal block matrix may be worthwhile and acceptable if the matrix L is used as a part of a preconditioner.

Nevertheless, such a band structure above is unlikely if L stems from an incomplete factorization of a very poorly conditioned coefficient matrix for which the incomplete factorization requires pivoting and/or one of the modern ordering techniques described in [12] or [15]. In this case, any reordering taking the numerical values of the non-zero elements of the coefficient matrix and not just their location into account and any pivoting is likely to destroy any inherent structure that might otherwise be in L, leading to a more uniform distribution of elements in L. This is certainly better for the parallelism exploited by these algorithms than any band structure.

In the next section, we provide the necessary definitions and background information including the sequential algorithm for triangular solves. We continue with block anti-diagonal based triangular solve and discuss the selection of the appropriate block structure. We also introduce the appropriate matrix format, the parallel solve format (PS), for implementing a parallel triangular solve algorithm effectively as well as the parallel solve algorithm itself. The hybrid algorithm, which is introduced next, also uses the PS format. Although this article contains all the technical details needed to implement these algorithms, these sections have been clearly marked and can be skipped. We continue with numerical results for the lower triangular matrices obtained by modern multilevel preconditioners. We conclude with a few comments on the possibility and difficulty of implementing these algorithms on a distributed memory architecture.

2 Background and Notation

For this entire article, we will assume that L is a lower triangular matrix with unit diagonal. We begin by recalling the standard forward substitution for solving linear systems with L. The same algorithms can be used for general lower triangular matrices $\tilde{L}$ not having a unit diagonal by factoring $\tilde{L} = DL$, where $D = \operatorname{diag}(\tilde{L})$ is the diagonal matrix containing the diagonal of $\tilde{L}$. Alternatively, these algorithms can easily be modified to take general lower triangular matrices into account by dividing by the diagonal element at the appropriate places. Furthermore, unless indicated otherwise, all algorithms can be modified in an obvious manner so that they can be used for upper triangular matrices U with unit diagonal.

2.1 Standard Sequential Triangular Solve

Before discussing different parallel implementations, we recall the standard sequential algorithm. Let $L \in \mathbb{R}^{d\times d}$ be a unit lower triangular matrix and $y \in \mathbb{R}^d$. The following algorithm returns the solution x of the linear system $Lx = y$.

Algorithm 2.1 (Triangular Solve)

```
1.  $x = y$
2.  for $i = 2, \dots, d$
3.      for $j = 1, \dots, i-1$
4.          $x_i = x_i - l_{ij} x_j$
5.      end for $j$
6.  end for $i$
```

2.2 Block Matrices and Vectors

The class of algorithms presented in the sequel aim at dividing L into a block matrix and using the naturally occurring parallelism. Before continuing, we need to specify the notation.

Definition 2.2 (a) Let $v \in \mathbb{R}^d$. We call a set $R = \{r_1, \dots, r_{M+1}\}$ of integers satisfying $1 = r_1 < \ldots < r_{M+1} = d+1$ a *block partition* or just *partition* for v. A block partition defines a M-dimensional block vector v^R having blocks v^R_m, $m = 1, \dots, M$ such that $v^R_m = (v_i)$, $i = r_m, \dots, r_{m+1} - 1$.

(b) Let $A \in \mathbb{R}^{d\times d}$. We call a pair of two sets of integers (R, S) with $R = \{r_1, \dots, r_{M+1}\}$ and $S = \{s_1, \dots, s_{N+1}\}$ satisfying $1 = r_1 < \ldots < r_{M+1} = d+1$ and $1 = s_1 < \ldots < s_{N+1} = d+1$ a *block partition* or just *partition* for A. A block partition defines a $(M \times N)$ block matrix $A^{(R,S)}$ having blocks $A_{mn} = A^{(R,S)}_{mn}$, $m = 1, \dots, M$, $n = 1, \dots, N$ such that $A^{(R,S)}_{mn} = (a_{ij})$, $i = r_m, \dots, r_{m+1} - 1$, $j = s_n, \dots, s_{n+1} - 1$.

2.3 A First Approach to Parallel Triangular Solves

Before discussing how parallel solves can be done fairly efficiently, we need to briefly address naive approaches for two reasons. First of all, when addressing algorithmic questions, knowing what does not work well is just as important as knowing what works well.

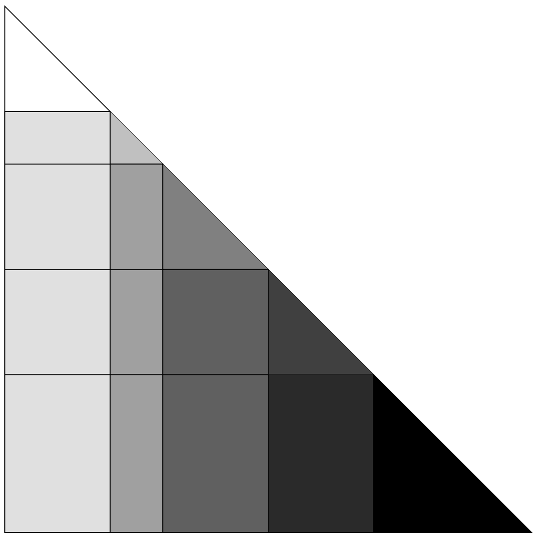
Figure 1: Illustration of triangular solve by block columns. Lower triangular blocks must be processed sequentially. Blocks shaded identically can be processed in parallel by rows. Light blocks are solved first.

Secondly, understanding one of the naive approaches is useful for improving one of the algorithms presented in the sequel.

The first naive possibility is to interpret line 4 of Algorithm 2.1 as a scalar product of the (sparse) ith row of L with the (dense) vector x and to compute this scalar product in parallel. However, as already observed in [14] section 11.6.1, the row of L usually contains very few non-zero elements and the threads have to be synchronized d times, resulting in very poor performance. Thus, this approach is useless.

Alternatively, it is possible to split L into a block matrix using the same partition for rows and columns. Then it is possible to solve all diagonal blocks sequentially and all rows occurring in the same block column in parallel, see Figure 1. Similarly, it would be possible to solve L in parallel by block rows. Generally, either approach is highly ineffective because the lower triangular blocks must be processed sequentially and often these blocks are amongst the densest in the entire matrix. Hence, we need algorithms which allow useful work to be done while the lower triangular blocks are being processed.

3 Parallel Triangular Solve by Block Anti-Diagonals

3.1 Various Versions of the Basic Algorithm

Using given partitions $R = \{r_1, \ldots, r_{M+1}\}$ and $S = \{s_1, \ldots, s_{N+1}\}$ of L, we can rearrange the operations in Algorithm 2.1 so that we process L by increasing block anti-diagonals.

Algorithm 3.1 (Triangular Solve by Block Anti-Diagonals in Explicit Form)

```
1.  $x = y$
2.  for $k = 2, \ldots, M + N$
3.      for $m, n : m, n \geq 1$ and $m + n = k$
4.          for $i = r_m, \ldots, r_{m+1} - 1$
```

```
5.           for $j = s_n, \ldots, s_{n+1} - 1$ and $j < i$
6.              $x_i = x_i - l_{ij} x_j$
7.           end for $j$
8.        end for $i$
9.     end for $m, n$
10. end for $k$
```

The previous algorithm is perhaps a bit clearer in block notation. Let $\hat{L}$ be L with the diagonal set to 0, i.e. $\hat{L} = L - I$, I being the identity matrix. Note that the triangular solve only requires matrix-vector-multiplication of blocks of $\hat{L}$ with blocks of x.

Algorithm 3.2 (Triangular Solve by Block Anti-Diagonals in Block Form)

```
1. $x = y$
2. for $k = 2, \ldots, M + N$
3.    for $m, n : m, n \geq 1$ and $m + n = k$
4.          $x_m^S = x_m^S - \hat{L}_{mn}^{(R,S)} x_n^R$
5.    end for $m, n$
6. end for $k$
```

It is clear that blocks corresponding to a fixed k, i.e. lying on the same block anti-diagonal, can be processed in parallel for a triangular solve because the reading and writing in x is disjoint for processing the various blocks. Only a block containing part of the diagonal of L requires reading from and writing to x at identical positions. Note that the terms "diagonal block" and "block containing part of the diagonal" are not synonymous. In fact, when the sizes of the row and column partitions are different, speaking of a diagonal block makes no sense. But even when the sizes of the partions are equal, but the partitions themselves are different, then a block diagonal element does not necessarily contain part of the diagonal and the diagonal must not be contained entirely in block diagonal elements, see Figure 2. On the other hand, if the same partition is used for rows and columns, then the two concepts are synonymous.

However, not just the blocks on any block anti-diagonal can be processed in parallel — all rows in all blocks, except for those in any block containing part of the diagonal, can be processed independently. In other words, a block containing part of the diagonal must be assigned entirely to a single processor, but all other blocks can be subdivided further by rows and different rows may be assigned to different processors. The parallelism just described is illustrated in Figure 2.

Algorithm 3.3 (Parallel Triangular Solve by Anti-Diagonals in Explicit Form)

```
1.  $x = y$
2.  for $k = 2, \ldots, M + N$
3.     parallel for $m, n : m, n \geq 1$ and $m + n = k$
4.        parallel if $\{r_m, \ldots, r_{m+1} - 1\}$ and $\{s_n, \ldots, s_{n+1} - 1\}$ are disjoint
5.           parallel for $i = r_m, \ldots, r_{m+1} - 1$
6.              for $j = s_n, \ldots, s_{n+1} - 1$ and $j < i$
7.                 $x_i = x_i - l_{ij} x_j$
8.              end for $j$
9.           end for $i$
10.       else (process block containing part of the diagonal sequentially)
11.          for $i = r_m, \ldots, r_{m+1} - 1$
```

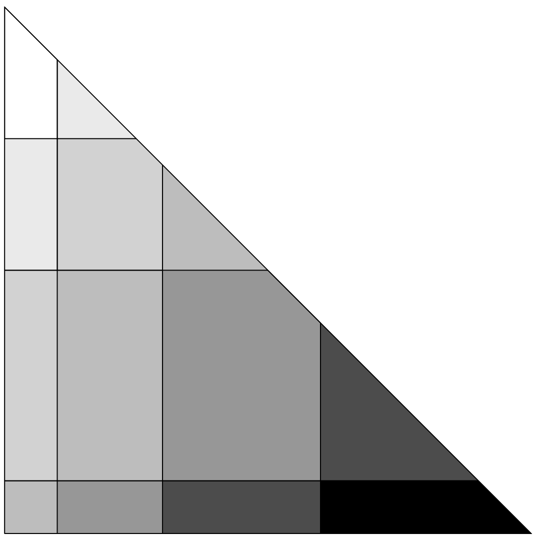

Figure 2: Illustration of triangular solve with block partition. Blocks shaded identically can be processed in parallel by rows, but all elements of a block containing part of the diagonal must be assigned to a single processor. Light blocks are processed first.

```
12.                 **for** $j = s_n, \dots, s_{n+1} - 1$ **and** $j < i$
13.                     $x_i = x_i - l_{ij} x_j$
14.                 **end for** $j$
15.             **end for** $i$
16.         **end if**
17.     **end for** $m, n$
18. **end for** $k$
```

Note that in line 4, the algorithm checks if a block contains part of the diagonal. Furthermore, **parallel** indicates that both branches of **if** or all loops of **for** can be processed in parallel. Other loops must be processed sequentially. For this algorithm, one branch of the **if**-clause is assigned to one processor and the other branch, which can be processed in parallel, is distributed amongst all processors, possibly, but not necessarily including the processor to which the block containing part of the diagonal was assigned.

For the data distribution, we use the following approach. First, we count the number of non-zero elements in any block containing the diagonal and the number of non-zero elements in each row in the remaining blocks which can be processed in parallel. We determine the ideal number of elements per processor, which is defined to be the total number of non-zero elements divided by the number of processors. We assign all rows of any block containing part of the diagonal to a single processor. If this block contains less than the ideal number of non-zero elements, then we assign further rows to this processor until we reach the ideal number of elements. Then we assign rows to the next processor, again until we reach the ideal number of non-zero elements. We proceed similarly for each processor. On the other hand, if the block containing the diagonal has more than the ideal number of non-zero elements, no further rows are assigned to this processor. In this case, we redefine the ideal number of non-zero elements to be the number of non-zero elements of the remaining blocks divided by the number of remaining processors and proceed as before. Details on the implementation of the data distribution depend on the matrix formats used, so these will be addressed in Section 3.4.

Even though a more sophisticated approach for distributing the rows is conceivable, e.g. distributing rows with the most non-zero elements first, usually the rows in each block contain a fairly small number of non-zero elements, so that we obtain a uniform distribution of the data by using this simple method. Indeed, the biggest problem occurs whenever the block containing part of the diagonal has many non-zero elements because they cannot be distributed. Hence, the prime goal in selecting a partition is to keep the number of non-zero elements in these blocks small (relative to the total number of non-zero elements to be distributed).

3.2 Partition Construction

To obtain good parallelization, we should try to select fairly coarse partitions because then we will need to synchronize the processors and restart threads less frequently. However, blocks containing part of the diagonal should not contain too many non-zero elements because these cannot be distributed to different processors. As these blocks constitute the main bottleneck for effective parallelization, we will prescribe the ideal number Δ of non-zero elements in any block containing part of the diagonal and use this restriction to select the partition. Additionally, we require the partitions for rows and columns to be the same.

Although we also experimented with different approaches allowing for different partitions, such counting elements in the rows and columns of L and determining the partitions so that the number of elements in the resulting block rows and columns would be similar, the performance was worse, so we will not consider them here. Nevertheless, more investigation is warranted because using different partitions for rows and columns is an attractive option. If we use the same partition of cardinality $N+1$ for rows and columns, then we have exactly N diagonal blocks and $2N-1$ block anti-diagonals. However, if we use different partitions for rows and columns, we may be able to distribute elements close to the diagonals more effectively so that more block anti-diagonals contain part of the diagonal. If this could be done in such a manner that each block containing part of the diagonal contains fewer non-zero elements then we might be able to obtain a more uniform data distribution to the processors, because these blocks must be assigned to a single processor. Unfortunately, it is not clear how this could be done effectively, so we continue with the simple approach.

Hence, for the simple approach, we scan the rows of L by increasing indices and insert a break whenever this will result in a diagonal block containing approximately Δ non-zero elements. In other words, the first row break is inserted so that the top triangular block has the prescribed number of non-zero elements Δ and we insert a column break with the same index. We continue and insert a second row break, if the resulting triangular block again has the prescribed number of non-zero elements and insert a column break with the same index. Continuing in this fashion, we obtain the required partitions. We assume that L is stored either in compressed sparse row (CSR) or compressed sparse column format (CSC). For details on these formats, consult [14]. If L is stored in CSR format, then implementing this approach is straightforward. If L is stored in CSC format, we can read L by increasing rows by using the auxiliary structure described in [4] consisting of two vectors `Lfirst` and `Llink` of dimension d. For an upper triangular matrix U, the approach is similar, except that the rows should be processed in descending order. This procedure for L is summarized in Algorithm 3.4.

Algorithm 3.4 (Anti-Diagonal-Based Partition Construction)

```
1.  Initialize partition: $s_1 = 1$
2.  ps $= 1$ (current size of partition)
3.  CurrentNNZ $= 0$
4.  for $i = 1, \dots, d$
5.      for $j = 1, \dots, i-1$ and $l_{ij} \neq 0$
6.          if $j \geq s_{\texttt{ps}}$
7.              CurrentNNZ $=$ CurrentNNZ $+ 1$
8.          end if
9.      end for $j$
10.     if CurrentNNZ $\geq \Delta$
11.         ps $=$ ps $+ 1$, $s_{\texttt{ps}} = i$
12.         CurrentNNZ $= 0$
13.     end if
14. end for $i$
15. if $s_{\texttt{ps}} \neq d$: ps $=$ ps $+ 1$, $s_{\texttt{ps}} = d$
```

If the partition of L is fairly coarse, i.e. if each block has a large number of rows, then we would expect most columns of the block anti-diagonal to have non-zero elements. In this case, the algorithm above is efficient. However, if the partition is quite fine and/or a large number of columns are empty, then checking every column while scanning each anti-diagonal is inefficient. In this case, a linked list may be used to keep track of those columns which have elements in the current block anti-diagonal. A similar approach is used in the implementation of ILUC, see [4]. However, parallization is only efficient if the blocks are fairly large, so that using linked lists offers no advantage in practice. Indeed, initial numerical tests supported this, so that we will not pursue using linked lists any further.

Although the partition as constructed by Algorithm 3.4 depends on the parameter Δ, the performance of the parallel solve in terms of the computation time is fairly constant for a large range of Δ. Often variations of one or two orders of magnitude in Δ do not effect the computation time by more than 10%. Nevertheless, a good choice for Δ depends on L, so we also consider algorithms for which all parameters can be fixed. However, we continue first with the implementation details of this algorithm.

3.3 The Parallel Solve Format for a Triangular Matrix (PS Format)

Here, we begin providing the technical details needed for storing L in such a manner that it can be solved effectively in parallel. For parallel solves to be efficient, all processors need to access the data sequentially. We call this the parallel solve (PS) format for triangular matrices.

Suppose we have P processors and that the matrix L consist of K regions that can be solved in parallel, e.g. for Algorithm 3.3, we have $K = M + N - 1$ and the regions that can be processed in parallel are the block anti-diagonals. The parallel solve data format requires an array of integers `pointer` of dimension $K \cdot P + 1$ and three parallel arrays of dimension equal to the number of non-zero elements of L. Two of these arrays, `rowind` and `colind` are integer arrays, and one, `values`, is a real array. `pointer` is a pointer array indicating the beginning and end of the data in step k, $k = 1, \dots, K$ which is assigned to a particular processor. The actual data of L is stored in the parallel arrays `rowind`, colind and values. rowind and colind store the row and column indices of the non-zero elements of L and values stores the corresponding values of l_{ij}. The elements of L that can be processed by processor p in step k are stored in the positions q of rowind, colind, values satisfying pointer$[P \cdot (k-1)+p] \leq q <$ pointer$[P \cdot (k-1)+p+1]$. Hence, the triangular solve can be performed in parallel by the following algorithm. Here, x and y denote arrays for storing the vectors x and y.

Algorithm 3.5 (Parallel Triangular Solve in Explicit Form for the PS Format)

```
1.  x = y
2.  for $k = 1, \ldots, K$
3.     parallel for for each processor $p = 1, \ldots, P$
4.        for $q =$ pointer$[P * (k-1)+p], \ldots,$ pointer$[P * (k-1)+p+1]-1$
5.           x[rowind[$q$]] = x[rowind[$q$]] − values[$q$] * x[colind[$q$]]
6.        end for $q$
7.     end parallel for $p$
8.  end for $k$
```

Memory may be reduced slightly by storing parallel regions in CSR or CSC format. In this case, we can replace one of the integer arrays rowind or colind by an array which will usually be smaller. Nevertheless, the savings are small, so that we deemed it preferable to use a format which will allow for a slightly simpler implementation of the parallel solve.

3.4 Conversion to PS Format for the Parallel Block Anti-Diagonal Algorithm

We continue with the technical details by describing how an arbitrary unit lower triangular matrix L can be converted efficiently into the PS format using the parallelism of Algorithm 3.3. Recall that the CSC format is the usual format when L is obtained by ILUC, an incomplete LU-factorization based on Crout's implementation, see [4] so we will assume that L is stored in this format. A few comments for conversion from other formats can be found at the end of this section. We also assume that a block partition (R, S) has been selected for L.

As L is stored in CSC format, we have an integer array cpointer of dimension $n+1$ and two parallel arrays, indices and data, of dimension equal to the number of non-zero elements of L, the former being an integer array and the latter a real array. For the j-th column, the values l_{ij} and the corresponding row indices i are stored in the arrays data and indices at the positions cpointer$[j]$ to cpointer$[j+1]-1$. For details, see [14].

Before discussing details, we summarize the general idea of the algorithm. For a given block anti-diagonal, we scan L by columns and count the number of non-zero elements in each row in that block anti-diagonal. Based on this information, each row is assigned to a particular processor. Then, in a second scan, we copy the elements of L to the arrays rowind, colind and values, grouping rows that have been assigned to the same processor. We proceed by increasing processor number. Once this has been completed, we update pointer. We continue until all block anti-diagonals of L have been copied, beginning with the anti-diagonal containing the initial block L_{11}.

For an efficient implementation, we also need an integer arrays first of dimension d. The array first is initialized to equal the first d elements of cpointer so that it points to the beginning of the columns of L. For $k = 2, \ldots, M+N$, we scan the kth block anti-diagonal whose block elements L_{mn} satisfy $m + n = k$. At the beginning of the kth step, the jth element of `first` points to the position in `data` and `indices` corresponding to the first element in the jth column contained the kth (or higher) block anti-diagonal, i.e. to the first element contained in a block L_{mn} satisfying $m + n \geq k$. For each column j, we scan the array `indices` starting at position `first`$[j]$ until we reach the first index outside of the kth block anti-diagonal. During scanning, we count the number of non-zero elements in each row contained in the kth block anti-diagonal. Based on this information, we assign each row to a processor. Then, we scan the same region of the matrix again using `first` and copy the data to PS format. We also update `first` and `pointer`. For the algorithm we also need two d-dimensional arrays `nnz_row` and `assign` to respectively store the number of non-zero elements in each row and the assignment of each row to a processor. Additionally, we have two P-dimensional arrays `nnz` and `wp` to store the number of non-zero elements assigned to each processor and to store the write position in the PS format for each processor.

Algorithm 3.6 (Anti-Diagonal Based Conversion of CSC to PS Format)

```
1.  pointer[1] = 1, $h = 1$ (write position for rowind, colind, values)
2.  initialize first: first = cpointer[1 : $d$]
3.  for $k = 2, \ldots, M + N$

      Analysis Phase:
          count number of elements in each row of block anti-diagonal

4.    nnz_row = 0
5.    for $m, n : m, n \geq 1$ and $m + n = k$
6.      for $j = s_n, \ldots, s_{n+1} - 1$
7.        $r$ = first[$j$]
8.        while $r <$ cpointer[$j + 1$] and indices[$r$] $< s_{n+1}$
9.          nnz_row[indices[$r$]] = nnz_row[indices[$r$]] + 1, $r = r + 1$
10.       end while
11.     end for $j$
12.   end for $m, n$

      Assignment Phase:
           Assign rows, determine write positions, update pointer

13.   Assign each row to a processor $p$: row $i$ is assigned to processor assign[$i$].
14.   Determine number of non-zero elements nnz[$p$] for each processor $p$
15.   wp[1] = pointer[$P * (k - 2) + 1$] (write position for data for first processor)
16.   for $p = 2, \ldots, P$
17.     wp[$p$] = wp[$p - 1$] + nnz[$p - 1$]
18.     pointer[$P * (k - 2) + p$] = wp[$p$]
19.   end for $p$
20.   pointer[$P * (k - 1)$] = pointer[$P * (k - 1) - 1$] + nnz[$P$]

      Copy Phase: copy data to parallel solve format

21.   for $m, n : m, n \geq 1$ and $m + n = k$
22.     for $j = s_n, \ldots, s_{n+1} - 1$
23.       $r$ = first[$j$]
```

```
24.             while $r <$ cpointer[$j+1$] and indices[$r$] $< s_{n+1}$
25.                 $i =$ indices[$r$] (row index of current element)
26.                 $h =$ wp[assign[$i$]] (current write position)
27.                 colind[$h$]$=j$, rowind[$h$]$=$indices[$r$], values[$h$]$=$data[$r$]
28.                 $r = r+1$, wp[assign[$i$]] $=$ wp[assign[$i$]] $+1$
29.             end while
30.             update first: first[$j$] $= h$
31.         end for $j$
32.     end for $m, n$

        End Copy Phase

33. end for $k$
```

Note that here we set $k = 2, \ldots M+N$ as in Algorithm 3.3. This is natural in this context as opposed to $k = 1, \ldots K$ with $K = M+N-1$ as one might expect for the PS format.

Perhaps a few words of explanation are needed for line 14 and line 15 of Algorithm 3.6. It is possible to assign all rows by increasing row indices, so that any block containing part of the diagonal is assigned first. Specifically, we assign all of these rows to the first processor. As the number of non-zero elements in each row is stored in `nnz_row`, we can determine the total number of non-zero elements in the entire block anti-diagonal as a sum of elements in `nnz_row`. Next, we can determine the number of non-zero elements that should be assigned to each processor. Here, we have two scenarios: if the number of non-zero elements already assigned to the first processor is more than the ideal, i.e. more than the total number of non-zero elements divided by P, then the workload for the first processor is already above the ideal, so that no further non-zero elements are assigned to this processor. Then we determine the number of elements that should be distributed to each of the remaining processors as the quotient of the total number of remaining non-zero elements divided by the number of remaining processors $P-1$. Note that we have already processed any block containing part of the diagonal and we know the total number of non-zero elements in the entire block anti-diagonal, so that we can actually determine the intended workload for each processor. On the other hand, if the first processor has a workload that is below the ideal, then we can assign it the difference between the ideal and the number of non-zero elements that it has already been assigned to process. Furthermore, we also intend for the remaining processors to process the ideal number of non-zero elements. So, we continue assigning the rows of the block anti-diagonal to the processor with lowest index p whose current workload is less than the intended workload. Whenever a processor has just reached its intended workload, we assign subsequent rows to the next processor.

Finally, we need to address the conversion of L from other formats to PS format as well as the conversion for a unit upper triangular matrix U. If L is stored in CSR format, we can just read L by rows, reading each row upto the index, where the next block anti-diagonal begins. We store this position as the starting point for reading the next block anti-diagonal. Hence, the implementation is straightforward and simpler than when L is stored in CSC format. Similarly, we can convert U from CSR format to PS format, the only difference being that the rows of U should now be read in descending order. For U stored in CSC format, we need to also read the rows in descending order. This can be accomplished by modifying the use of `Lfirst` and `Llink` appropriately.

4 The Hybrid Anti-Diagonal-Column Algorithm

4.1 Description of the Algorithm

The main drawback of the anti-diagonal oriented algorithm is the dependency of the partition on the parameter Δ. Although the partition for the algorithm which we present here depends on two parameters, a fixed choice yields optimal results for all matrices tested.

For this algorithm, we assume that we have the same partition $S = \{s_1, \ldots, s_{N+1}\}$ for both rows and columns of L. In this case, the blocks on the block diagonal are lower triangular blocks and the only blocks containing part of the diagonal of L. These blocks, but no others, must be processed sequentially in their natural order for the triangular solve of L. For $n = 2, \ldots, N-1$ denote by C_n the set of blocks $L_{m,n-1}$ with $m > n$. Note that for any diagonal block L_{nn}, the rows of C_n can be processed parallel to L_{nn}. Hence, for $n = 2, \ldots, N-1$, we can process L_{nn} and C_n in parallel, thus covering the entire matrix with the exception of the blocks on the lower block off-diagonal, i.e. the blocks $L_{n,n-1}$, $n = 2, \ldots, N$. The processing of these blocks, however, must be interlaced with the processing of the diagonal blocks and block columns. Note that the rows of each of these off-diagonal blocks can be processed in parallel as well. In this manner, we obtain the following algorithm which is illustrated in Figure 3.

Algorithm 4.1 (Hybrid Parallel Triangular Solve in Explicit Form)

```
1.  $x = y$
2.  for $k = 2, \ldots, 2N$
3.      if $k$ is even
4.          $n = \frac{k}{2}$
5.          begin parallel region
6.              parallel region 1 (do diagonal block $L_{nn}$)
7.                  for $i = s_n, \ldots, s_{n+1} - 1$
8.                      for $j = s_n, \ldots, s_{n+1} - 1$ and $j < i$
9.                          $x_i = x_i - l_{ij} x_j$
10.                     end for $j$
11.                 end for $i$
12.             parallel region 2 (do $C_n$ but only if $2 \le n \le N - 1$)
13.                 parallel for $i = s_{n+1}, \ldots, d$
14.                     for $j = s_{n-1}, \ldots, s_n - 1$
15.                         $x_i = x_i - l_{ij} x_j$
16.                 end for $j$
17.             end for $i$
18.         end parallel region
19.     else ($k$ odd) do off-diagonal block $L_{n-1,1}$
20.         $n = \frac{k+1}{2}$
21.         parallel for $i = s_n, \ldots, s_{n+1} - 1$
22.             for $j = s_{n-1}, \ldots, s_n - 1$
23.                 $x_i = x_i - l_{ij} x_j$
24.             end for $j$
25.         end for $i$
26.     end if
27. end for $k$
```

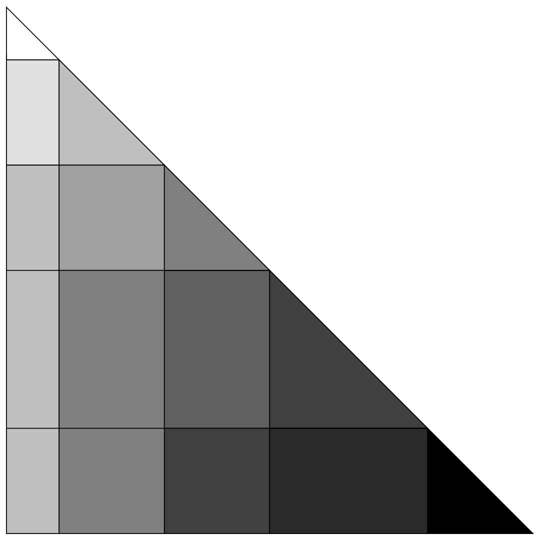
Figure 3: Illustration of the hybrid triangular solve algorithm by block anti-diagonals and block columns. Lower triangular blocks must be processed sequentially but rows of the block column shaded identically can be processed in parallel. Single blocks are also solved in parallel. Light blocks are solved first. The algorithm alternates between processing a lower triangular block and a block column and processing a off-diagonal single block.

4.2 Partition Construction

The underlying idea for finding a suitable partition for Algorithm 4.1 is that during the sequential processing of the lower triangular blocks, the remaining processors should not be idle. Hence, the partition should be selected in such a manner that the number of non-zero elements in C_n is $P-1$ times the number of non-zero elements of L_{nn}. For this approach, we specify two parameters: Λ indicates the ideal number of non-zero elements for the initial block L_{11} and Θ the minimal number of non-zero elements that we require every other diagonal block L_{nn} to have. The second parameter is needed to prevent the partition from becoming to fine. The first parameter is needed to specify a compromise between two options: If Λ is very large, processing the initial triangular block is expensive as it must be processed sequentially while the other processors are idle. If Λ is too small, then L_{11} will be small, usually resulting in C_1 also being small. However, then L_{22} will also be small, thus propagating a fine partition. Hence, if we start with small blocks, then subsequent blocks will also be small, which is not desirable. Although the algorithm depends on Λ and Θ, setting $\Lambda = 1000$ and $\Theta = 25$ was the optimal choice for *all* matrices tested. Actually, the algorithm is not very sensitive to the choice of Λ and Θ and other choices would have yielded very similar results.

For the efficient implementation, we again assume that L is stored in CSC format. We again scan L for $i = 1, \ldots, d$ by increasing rows and count the number of non-zero elements. As soon as we have obtained the required Λ non-zero elements, we insert a break and obtain s_2. We continue scanning rows, counting the number of non-zero elements of L_{21} and L_{22} separately. This is possible because s_2 also indicates the division point between L_{21} and L_{22}. After the ith row is scanned, we can determine the number of non-zero elements that L_{21} and L_{22} would contain if we inserted a row break either *before* or *after* the ith row. Because we also know the number of non-zero elements of L_{11} and because L is stored in CSC format, we can compute the number of non-zero elements that would be contained in C_2. If after inserting a break before the ith row, the number of non-zero elements of L_{22} would be at least Θ and if $P-1$ times the number of non-zero elements of L_{22} would exceed number of non-zero elements of C_2 if the break were inserted after the ith row, then we actually insert the break before the ith row. Note that for $\Theta = 0$ we insert the row break in the last position for which $P-1$ times the number of non-zero elements of L_{22} is less than the number of non-zero elements of C_2, so that we obtain the largest possible lower triangular block allowing for optimal data distribution. For larger values of Θ, we risk not having the optimal data distribution, but we avoid having lower triangular blocks that are too small. Note that we continue scanning by rows if the conditions for inserting a break are not satisfied. Continuing in this manner, we obtain S.

To obtain a more precise description of this algorithm, we define t as a counter for the number of non-zero elements in the current triangular block, t_{old} as a counter for non-zeroes in the previous triangular block, t_{row} as a counter for the number of non-zero elements in the row of the triangular block currently being scanned, b as a counter for non-zeroes in the off-diagonal block, c as a counter non-zeroes in the current C block and `ps` as a counter for current number of elements in S. This yields:

Algorithm 4.2 (Hybrid-Based Partition Construction)

```
1.   $t_{\text{old}} = 0$, $t = 0$, $b = 0$, $c = 0$, $\texttt{ps} = 1$
2.   $s_0 = 0$, $s_1 = 1$ ($s_0$ is a dummy initialization)
3.   for $i = 1, \ldots, d$
4.       $t_{\text{row}} = 0$
5.       for $j = 1, \ldots, i-1$ and $l_{ij} \neq 0$
6.           if $s_{\texttt{ps}-1} \leq j < s_{\texttt{ps}}$ then $b = b + 1$
7.           if $s_{\texttt{ps}} \leq j$ then $t_{\text{row}} = t_{\text{row}} + 1$
8.       end for $j$
9.       if $\texttt{ps} > 1$ then $c = \texttt{cpointer}[\texttt{ps}-1] - \texttt{pointer}[\texttt{ps}-2] - b - t_{\text{old}}$
10.      if ($\texttt{ps} = 1$ and $t \geq \Lambda$) or ($\texttt{ps} > 1$ and $c \geq \Theta$ and $(t + t_{\text{row}}) \cdot (P-1) \geq c$)
11.          $\texttt{ps} = \texttt{ps} + 1$, $s_{\texttt{ps}} = i$, $t_{\text{old}} = t$, $t = 0$, $b = t_{\text{row}}$
12.      else $t = t + t_{\text{row}}$
13.      end if
14.  end for $i$
15.  if$s_{\texttt{ps}} \neq d$: $\texttt{ps} = \texttt{ps} + 1$, $s_{\texttt{ps}} = d$
```

If L is stored in CSR format, we first scan L and count the number of non-zero elements in each column. Then it is possible to proceed as above. Alternatively, it is possible to convert L to a hybrid anti-diagonal and row-based structure. The corresponding illustration is obtained by reflecting Figure 3 along the anti-diagonal. In this case, L is scanned by decreasing columns. If U is stored in CSR format, then we use the appropriate hybrid format obtained by transposing Figure 3 and scanning U by columns. For U stored in CSC format, we either have the option of first counting the number of elements in each row and proceeding as before or we may use the format obtained by transposing Figure 3 along both the diagonal and anti-diagonal and scanning by decreasing rows.

4.3 Conversion to PS Format

Now that we have determined the partition S, we still need to provide the technical details to convert L from CSC to PS format. To this end, we scan L by increasing columns. We scan the first column and copy all elements of L until the first triangular block L_{11} ends and the first off-diagonal block L_{21} begins. We store this position. Next, we scan the second column and proceed just as for the first column and continue until all columns of the triangular block have been scanned and all the elements have been copied. Next, we scan the first off-diagonal block L_{21} column by column until this block ends, counting the number of elements in each row. As we know the number of elements in each row, each of these rows can be assigned to a processor to achieve a uniform distribution. We rescan the off-diagonal block copying elements appropriately. Just as before, we also store the position in the CSC format at which the various columns of the off-diagonal block L_{21} end and the columns of C_2 begin. Although we have not yet copied C_2, we scan and copy the second triangular block L_{22}, assigning it to a single processor. This is done just as for the first diagonal block L_{11}. Next, we scan C_2 to count the non-zero elements in each row. As we have already copied the second triangular block L_{22}, we know the number of non-zero elements that have been assigned to a particular processor, so that we can assign the rows of C_2 to the various processors, aiming to obtain a uniform workload. After assigning the rows, we rescan C_2 and copy the data appropriately. Continuing as before with the second off-diagonal block L_{32}, C_3 and L_{33}, we eventually obtain L in PS format. Algorithm 4.3 summarizes this approach. $N + 1$ is again the cardinality of the partition S. The array `nnz_row` of dimension d counts the number of non-zero elements in each row, the arrays `nnz_OD` and `nnz_adc` of dimension P count the number of elements per processor in the off-diagonal parallel block and in the hybrid block anti-diagonal/column parallel region respectively and `assign` is an array of dimension d indicating the assignment of each row to a particular processor. Finally, `nnz_D` counts the number of non-zero elements in a fixed diagonal block, `sum_nnz_OD` the number of elements in a fixed off-diagonal block and `nnz_C` the number of non-zeroes in a block column.

Algorithm 4.3 (Conversion of CSC to PS Format for Hybrid Algorithm)

```
1.   for $k = 1, \ldots, N$
2.       nnz_row $= 0$, nnz_OD $= 0$, nnz_adc $= 0$
3.       for $j = s_k, \ldots, s_{k+1} - 1$ (count elements in rows)
4.           for $i =$ cpointer$[j], \ldots,$ cpointer$[j+1] - 1$
5.               nnz_row[indices$[i]$] $=$ nnz_row[indices$[i]$] $+ 1$
6.           end for $i$
7.       end for $j$
8.       nnz_D $= 0$
9.       for $j = s_k, \ldots, s_{k+1} - 1$ (count elements in diagonal block, assign rows)
10.          nnz_D $=$ nnz_D $+$ nnz_row$[j]$
11.          assign$[j] = P$
12.      end for $j$
13.      nnz_adc$[P]$ $=$ nnz_D
14.      if $k < N$ (check if an off-diagonal block exists)
15.          sum_nnz_OD $= 0$
16.          for $j = s_{k+1}, \ldots, s_{k+2} - 1$ (count elements in off-diagonal block)
17.              sum_nnz_OD $=$ sum_nnz_OD $+$ nnz_row$[j]$
18.          end for $j$
```

```
19.         for $j = s_{k+1}, \ldots, s_{k+2} - 1$ (assign rows of off-diagonal block)
20.             Use nnz_row, sum_nnz_OD to assign row $j$ to a processor,
                        store assignment in assign[$j$]
21.             nnz_OD[assign[$j$]] = nnz_OD[assign[$j$]] + nnz_row[$j$]
22.         end for $j$
23.     if $k < N - 1$ (check if a block column exists)
24.         nnz_C = 0
25.         for $j = s_{k+2}, \ldots, s_{k+3} - 1$ (count elements in block column)
26.             nnz_C = nnz_C + nnz_row[$j$]
27.         end for $j$
28.         for $j = s_{k+2}, \ldots, s_{k+3} - 1$ (assign rows of block column)
29.             Use nnz_row, nnz_C to assign row $j$ to a processor,
                        store assignment in assign[$j$]
30.             nnz_adc[assign[$j$]] = nnz_adc[assign[$j$]] + nnz_row[$j$]
31.         end for $j$
32.     end if
33.     Use nnz_adc and nnz_OD to update pointer
34.     Determine write position for all $p$ and both parallel regions
35.     for $j = s_k, \ldots, s_{k+1} - 1$ (copy by columns)
36.         for $i$ = cpointer[$j$], ..., cpointer[$j + 1$] − 1
37.             Use row index indices[$i$] and assign to copy data
38.         end for $i$
39.     end for $j$
40. end for $k$
```

5 Numerical Results

To test the algorithms presented in this article on realistic problems, we selected some of the largest unsymmetric matrices in the University of Florida Sparse Matrix Collection [5]. For each of these matrices, we determined a good multilevel ILU-based preconditioner using the ideas presented in [2] and [15]. Specifically, we proceeded as follows: First, the coefficient matrix A was preprocessed (i.e. rows and columns were permuted and scaled) to make A more suitable for incomplete factorization. For an overview of these methods, see [12]. Then, the matrix was factored using ILUC, see [4], but the factorization was terminated prematurely if the pivot was less than 0.01 in absolute value. In this case, an approximate Schur complement was computed, which was again preprocessed and factored incompletely just as the original matrix. In this manner, we obtain a recursive algorithm resulting in a multilevel factorization.

For the preprocessing, we considered just normalizing columns and then rows, using PQ-reordering as described in [15] or using one of the following two step preprocessing routines. First, A was transformed into an I-matrix, a matrix whose diagonal elements have absolute value of 1 and whose off-diagonal elements have absolute value of at most 1, see [6], [7] and [13]. In a second step, we applied one of the symmetric permutations (i.e. the same permutation for rows and columns) to the I-matrix, which is then again an I-matrix, as described in [12]. Specifically, we used Algorithms 1,2 and 3. Algorithms 1 and 2 were used with weighting strategy (c) of [12], which considers both the number of non-zero elements in each row and column and the 1-norm to obtain a sparse, more diagonally dominant initial block. We also tested just transforming A into an I-matrix without any further preprocessing.

For all matrices A, we constructed the multilevel ILU preconditioner with all preprocessing techniques using as a drop parameter $\tau = 10^{-t}$, $t = 1, \ldots, 9$. For all these combinations, we determined the total computation time consisting of the setup time for the preconditioner and the solve time for the iterative method. The results, which we present in the sequel, are for the lower triangular matrices L of each level of the preconditioner requiring the shortest total computation time for a specific linear system.

The algorithms in this article were implemented in C++ and will be made available in the next release of the software package ILU++ [11]. The code was compiled with the Intel C++ compiler and the actual testing was done on an Intel Xeon platform with two 2.33 Ghz quad core processors.

We will not report in detail on the time needed to compute partitions and to convert L from CSC to PS format. Suffice it to say that when implemented as described in this article, the time needed for partition computation and conversion lies in the range of about 8 to 15 times the time required for a single sequential triangular solve. Although this computation time may be too long for a practical algorithm, it is possible to save much time by implementing the following ideas:

- Partitions and conversions for L and U are independent so that L and U can be processed in parallel.
- In a multilevel setting, the levels are independent and they can be processed in parallel.
- Computing partitions and converting L and U are now being done one after the other for ease of implementation and clarity of exposition. However, these can be combined into a single algorithm, which may save some computation time.
- At least some of the work for computing partitions and converting L and U may be done as part of the partial factorization.

Note that even if we just implement the first idea and convert L and U in parallel, then the conversion of L and U will require no more than time than 2-4 (sequential) preconditioned BiCGstab iterations. Finally, the time for partition computation and conversion may even be amortized for the current implementation, if a preconditioner is to be used for several systems.

Hence, we will restrict our attention to triangular solve times that are obtained by using the anti-diagonal based and the hybrid algorithms. Recall that for the anti-diagonal based algorithm, we need to specify Δ, the ideal number of elements in each diagonal block. We tested 16 different values for Δ in geometric progression between 100 and 100 000 and report the best result obtained for each matrix. Usually, the optimal value for Δ was in the range of 1000 to 10 000. Although the performance was fairly constant over a range of Δ, it certainly was not independent of it. Hence, the need to determine a good value for Δ is certainly a drawback of the anti-diagonal based algorithm. For the hybrid algorithm, we report on the results for $\Lambda = 1000$ and $\Theta = 25$ fixed for all matrices.

In Table 1, we record the speed-up obtained for both algorithms and for $P = 2, 4, 8$. The speed-up is computed with respect to the standard sequential triangular solve for matrices stored in CSC format. We only record a result if the total solve time for 100 triangular solves was at least 0.1 seconds for the sequential algorithm. Otherwise, the computation times were too short to be meaningful. Consequently, we do not record the

Matrix	PP	Level	Anti-Diag. Speed-Up			Hybrid Speed-Up		
			2	4	8	2	4	8
av41092	PQ	2	1.7	3.1	4.3	1.5	2.5	3.1
	PQ	3	2.1	3.4	5.3	1.9	2.9	4.9
	PQ	4	1.8	2.9	3.7	1.8	2.6	3.3
	PQ	5	1.8	2.6	3.1	1.6	2.2	2.5
FEM_3D_thermal1	0	1	1.0	0.9	0.9	0.7	0.7	0.7
g7jac180	3	1	2.0	3.4	5.3	1.8	3.1	5.3
	3	2	1.6	2.1	2.0	1.9	2.0	2.0
g7jac200	3	1	1.9	3.1	5.0	1.8	3.1	5.2
	3	2	1.6	2.1	2.1	1.9	2.1	2.0
garon2	1	1	1.6	1.9	2.1	1.6	1.8	1.5
ns3Da	PQ	1	1.8	2.9	3.8	2.0	3.3	4.0
ohne2	2	1	1.4	1.8	1.8	1.5	2.3	2.2
	2	2	1.8	2.8	3.9	1.8	2.3	3.9
	2	3	1.9	2.8	3.7	1.9	3.0	4.0
para-9	3	1	1.6	1.8	1.9	1.9	2.4	2.2
	3	2	1.7	2.5	2.3	1.9	2.3	2.2
poisson3Db	1	1	1.4	1.4	1.5	1.3	1.4	1.2
rajat31	0	1	0.9	0.9	0.9	0.8	0.8	0.8
rma10	3	1	1.7	2.9	2.9	1.9	2.9	3.0
	3	2	1.8	3.1	3.3	1.9	3.1	3.1
sme3Dc	3	1	1.9	2.5	2.4	1.9	2.5	2.5
torso1	1	1	1.4	1.6	1.6	1.8	1.9	1.5
venkat50	1	1	1.8	2.4	2.3	1.7	2.5	2.4

Table 1: Speed-up with respect to the standard sequential algorithm for matrices in CSC format. PP indicates the preprocessing used.

results for some of the levels. We also use the following abbreviations: The column marked with PP indicates the preprocessing used. 0 indicates just normalization of columns and then rows but no reordering of either. 1,2,3 refer to the corresponding algorithms in [12] combined with weighting strategy (c). PQ is used to indicate the standard PQ algorithm introduced in [15].

Clearly, the results indicate that we do not have the speed-up that we would ideally like to have for parallel algorithms. The results for 2 processors are generally quite good and for 4 processors the results are generally good enough that using 4 processors is often worthwhile. However, the results for 8 processors are quite disappointing. A more careful analysis indicates that there are two main reasons for this poor performance. Generally, the data is distributed quite effectively, so this is usually not the cause for poor speed-up. However, sometimes we obtain a large number of parallel regions which requires the processors to synchronize more frequently resulting in poor performance. For a few matrices, almost all non-zero elements are quite close to the diagonal, so that no efficient parallization is possible at all. Of course, in this case, we cannot expect good performance.

To illustrate this differences in performance, we elaborate on the results for L of the first level of the preconditioner for three specific matrices A. Pictures of L can be found in

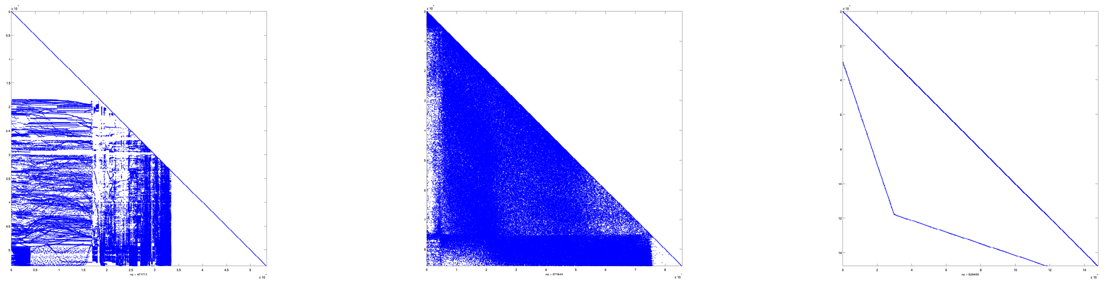

Figure 4: L for the first level of the preconditioner for the matrices g7jac180, poisson3Db and FEM_3D_thermal.

Figure 4. The best overall performance on 8 processors for the hybrid method is obtained for the matrix g7jac180. For this matrix, the initial triangular block contains exactly 1000 non-zero elements. Then, we have 3 parallel regions. In the first, each processor has 1860 or 1861 non-zero elements, in the next between 13 540 and 13 592 non-zero elements and between 36 653 and 36 800 non-zero elements in the final region. Hence, we have almost ideal data distribution. Taking a look at the picture of this matrix, we notice that the initial triangular block is quite sparse. This is in fact due to the preprocessing used and as a consequence, the initial triangular block is quite large. This is beneficial for the subsequent partitioning of L. Similarly, the final triangular block, which can also not be processed in parallel, is empty because the incomplete factorization was terminated prematurely and the approximate Schur complement was factored on the next level. On the other hand, the performance for poisson3Db is quite poor and indeed this matrix requires 22 parallel regions. Although the workload is quite uniform (except for the initial block, the difference between the number of non-zero elements assigned to the 8 processors never exceeds 6), sometimes the workload per processor is very low, often below 1000, sometimes even below 100 non-zero elements. Observe that the region closer to the diagonal of L is much denser for poisson3Db than for g7jac180. The worst performance overall is for FEM_3D_thermal1. Here, the algorithm assigns 998 non-zero elements to the initial triangular block and almost all others, 379 176 out of a total of 380 530, to a single triangular block. This is a consequence of the fact that this matrix is quite sparse and almost all elements are close to the diagonal as can also been seen in picture of the matrix. Even the thin angular structure further away from the diagonal contains only very few non-zero elements. Hence, no efficient parallization is possible. In this sense, this is no fault of the algorithm as it is clear that this matrix does not permit for an efficient parallel solve.

Slightly surprising is the fact that the performance of g7jac180, which has excellent data distribution and requires few parallel regions, is not any better for 8 processors. This is probably the result of the intense memory access that these algorithms require. Each element in L and x is used only once when read, so when 8 processors access memory simultaneously, this is likely to be a bottleneck.

Note that solving linear systems with the matrix FEM_3D_thermal1 does not require a very sophisticated preconditioner. Other than normalization, no preprocessing is used and the multilevel preconditioner consists of a single level, so that the preconditioner used is essentially ILUC, see [4], applied to FEM_3D_thermal1 without any pivoting or other more advanced strategy. Generally, one would consider such a situation fortunate, and the fact that the strong diagonal structure of FEM_3D_thermal1 is preserved in L to be beneficial. For parallelization using the algorithms in this article, however, this is not advantageous. In fact, rajat31 is the only other matrix for which parallelization is detrimental. This matrix also has a strong diagonal structure and because again no reordering is done, so does its factor L. On a more positive note, for both FEM_3D_thermal1 and rajat31 it certainly is possible to use reorderings, e.g. multilevel nested dissection [8], [9] to obtain better parallelization.

On the other hand, g7jac180 could only be solved when preprocessed with Algorithm 3 in [12], so reordering for better parallelization is highly unlikely to yield successful solves. Fortunately, Algorithm 3 also appears to yield a matrix structure for which the algorithms presented in this paper work well. This is not surprising because Algorithms 3 reorders A to obtain a particularly sparse initial block. If L inherits this sparsity, as is the case for g7jac180, and thus also has a sparse initial lower triangular block, then the algorithms presented in this article can be expected to perform well.

6 Conclusion and Outlook

This article introduces two new algorithms for solving triangular systems in parallel on a shared memory architecture. The first algorithm is based on solving the system by block anti-diagonals. The algorithm allows for great flexibility in choosing the block structure, i.e. different partitions may be used for rows and columns. Although this freedom has the potential of making this algorithm quite effective, in practice choosing appropriate partitions is difficult. We proposed selecting the same partition for rows and columns such that the diagonal blocks have a prescribed number of non-zero elements. Although this works quite well, it is probably possible to find better partitions and this should be considered for future work. The second algorithm tries to overcome this difficulty by solving the triangular matrix using a combination of processing by block columns and block anti-diagonals. In this situation, the partition can be chosen in a nearly optimal manner.

Preconditioning with multilevel incomplete LU factorizations consists of solves with triangular systems. Hence, the algorithms presented in this article provide a method for parallelizing the application of multilevel preconditioners. If a single linear system with a given coefficient matrix A is to be solved iteratively, then this can only be done efficiently in parallel if the computation of the preconditioner is also done in parallel. So in this case, parallelizing the factorization is essential and this is the subject of ongoing work. However, if many linear systems with the same matrix A are to be solved in parallel, then it may still be acceptable for the preconditioner to be computed sequentially and just using the parallization provided by the algorithms presented here may already be sufficient.

Although these algorithms were designed with shared memory systems in mind, they can be used with distributed memory systems as well, at least theoretically. For a triangular solve, each non-zero element of the triangular matrix is accessed by exactly one processor. Hence, each non-zero element of the triangular matrix should simply be stored in the local memory of the processor needing to access that element. The parallel solve format can be modified easily to accommodate this. Parts of the solution vector, however, need to be broadcast to all processors at the beginning of a parallel sweep and the updated vector needs to be collected at the end. This is a relatively expensive option, but may be acceptable if there are only a few parallel regions. On the other hand, determining effective partitions and distributing the data amongst the processors for a matrix stored on a distributed memory system is likely to be both difficult and expensive. Whether this problem can be solved satisfactorily depends both on the manner in which the matrix is stored after computation and on the underlying architecture. In any case, redistributing the triangular matrix after factorization is probably not an option so that it is necessary to distribute the triangular matrix during computation in a manner appropriate for the parallel solve. Hence, the implementation of the parallel solve algorithms presented here depends on the implementation of the factorization. Consequently, the suitability of these algorithms on a distributed memory architecture needs to be decided on a case-by-case basis and requires extensive investigations.

References

[1] Michelle Benzi, John C. Haws, and Miroslav Tůma. Preconditioning highly indefinite and nonsymmetric matrices. *SIAM J. Sci. Comp.*, 22(4):1333–1353, 2000.

[2] Matthias Bollhöfer and Yousef Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM J. Sci. Comput.*, 25(2):715–728, 2003.

[3] Matthias Bollhöfer and Yousef Saad. ILUPACK. `http://www.math.tu-berlin.de/ilupack/`, Jul. 2008.

[4] Edmond Chow, Na Li, and Yousef Saad. Crout versions of ILU for sparse matrices. *SIAM J. Sci. Comput.*, 25(2):716–728, 2003.

[5] Tim Davis. University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices`, Jul. 2008.

[6] Iain S. Duff and Jacko Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(1):889–901, 1999.

[7] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.*, 22(4):973–996, 2001.

[8] George Karypis. METIS. `http://glaros.dtc.umn.edu/gkhome/views/metis/`, Jul. 2008.

[9] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comp.*, 20(1):359–392, 1998.

[10] Jan Mayer. A multilevel Crout ILU preconditioner with pivoting and row permutation. *Numer. Linear Algebra Appl.*, 14(10):771–789, 2007.

[11] Jan Mayer. ILU++. `http://www.iluplusplus.de`, Jul. 2008.

[12] Jan Mayer. Symmetric permutations for I-matrices to delay and avoid small pivots. *SIAM J. Sci. Comput.*, 30(2):982–996, 2008.

[13] Markus Olschowska and Arnold Neumaier. A new pivoting strategy for Gaussian elimination. *Lin. Alg. Appl.*, 220:131–151, 1996.

[14] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, Philadelphia, 2003.

[15] Yousef Saad. Multilevel ILU with reorderings for diagonal dominance. *SIAM J. Sci. Comput.*, 27:1032–1057, 2006.

IWRMM-Preprints seit 2007

Nr. 07/01	Armin Lechleiter, Andreas Rieder: A Convergenze Analysis of the Newton-Type Regularization CG-Reginn with Application to Impedance Tomography
Nr. 07/02	Jan Lellmann, Jonathan Balzer, Andreas Rieder, Jürgen Beyerer: Shape from Specular Reflection Optical Flow
Nr. 07/03	Vincent Heuveline, Jan-Philipp Weiß: A Parallel Implementation of a Lattice Boltzmann Method on the Clearspeed Advance Accelerator Board
Nr. 07/04	Martin Sauter, Christian Wieners: Robust estimates for the approximation of the dynamic consolidation problem
Nr. 07/05	Jan Mayer: A Numerical Evaluation of Preprocessing and ILU-type Preconditioners for the Solution of Unsymmetric Sparse Linear Systems Using Iterative Methods
Nr. 07/06	Vincent Heuveline, Frank Strauss: Shape optimization towards stability in constrained hydrodynamic systems
Nr. 07/07	Götz Alefeld, Günter Mayer: New criteria for the feasibility of the Cholesky method with interval data
Nr. 07/08	Marco Schnurr: Computing Slope Enclosures by Exploiting a Unique Point of Inflection
Nr. 07/09	Marco Schnurr: The Automatic Computation of Second-Order Slope Tuples for Some Nonsmooth Functions
Nr. 07/10	Marco Schnurr: A Second-Order Pruning Step for Verified Global Optimization
Nr. 08/01	Patrizio Neff, Antje Sydow, Christian Wieners: Numerical approximation of incremental infinitesimal gradient plasticity
Nr. 08/02	Götz Alefeld, Zhengyu Wang: Error Estimation for Nonlinear Complementarity Problems via Linear Systems with Interval Data
Nr. 08/03	Ulrich Kulisch : Complete Interval Arithmetic and its Implementation on the Computer
Nr. 08/04	Armin Lechleiter, Andreas Rieder: Newton Regularizations for Impedance Tomography: Convergence by Local Injectivity
Nr. 08/05	Vu Hoang, Michael Plum, Christian Wieners: A computer-assisted proof for photonic band gaps
Nr. 08/06	Vincent Heuveline, Peter Wittwer: Adaptive boundary conditions for exterior stationary flows in three dimensions
Nr. 08/07	Jan Mayer: Parallel Algorithms for Solving Linear Systems with Sparse Triangular Matrices

Eine aktuelle Liste aller IWRMM-Preprints finden Sie auf:

www.mathematik.uni-karlsruhe.de/iwrmm/seite/preprints