# AN EFFICIENT AND STABLE METHOD FOR PARALLEL FACTORIZATION OF DENSE SYMMETRIC INDEFINITE MATRICES

P.E. Strazdins,
Department of Computer Science, Australian National University,
Acton, ACT 0200, Australia.
peter@cs.anu.edu.au
*and*
J.G. Lewis,
Boeing Company, Mail Code 7L-22, P.O. Box 3707, Seattle WA, USA.
jglewis@rt.cs.boeing.com

**Abstract**

This paper investigates the efficient parallelization of algorithms with strong stability guarantees to factor dense symmetric indefinite matrices. It shows how the *bounded Bunch-Kaufman* algorithm may be efficiently parallelized, and then how its performance can be enhanced by using exhaustive block searching techniques, which is effective in keeping most symmetric interchanges within the current elimination block. This can avoid wasted computation and the communication normally involved in parallel symmetric interchanges, but requires considerable effort to reduce its introduced overheads. It has also great potential for out-of-core algorithms.

Results on a 16 node Fujitsu AP3000 multicomputer showed the block search increased performance over the (plain) *bounded Bunch-Kaufman* algorithm by 5–14% on strongly indefinite matrices, and in some cases out-performing the well-known Bunch-Kaufman algorithm (which is without strong stability guarantees).

**KeyWords and Phrases:** symmetric indefinite matrices, LDLT decomposition, dense linear algebra, parallel computing, block-cyclic decomposition.

## 1 INTRODUCTION

Large symmetric indefinite systems of equations arise in many applications, including incompressible flow computations, optimization of linear and non-linear programs, electro-magnetic field analysis and data mining. An important special case, occurring in the latter two applications, are semi-definite and *weakly-indefinite* (ie. close to definite) systems.

Stable algorithms for solving $N \times N$ symmetric indefinite systems and yet exploit symmetry to have only $\frac{N^3}{3} + O(N^2)$ floating point operations are well known (see (Golub & Loan 1989) and the references within). The diagonal pivoting methods are dominant in the literature, and several high-performance implementations of algorithms based on this method have been given (Anderson & Dongarra 1989, Jones & Patrick 1991, Kaufman 1995, Ashcraft, Grimes & Lewis 1998, Strazdins 2000).

There have only been a few studies so far on parallel implementation of these algorithms, especially on distributed memory architectures. In (Jones & Patrick 1991), there are some results on a small-scale shared memory machine, and in (Strazdins 1999, Strazdins 2000), distributed memory implementations are described in some detail. However, all of these refer to variants of the Bunch-Kaufman algorithm, which has been recently shown to have no strong stability guarantees (Ashcraft et al. 1998).

In this paper, we describe how to derive stable variants of diagonal pivoting methods to yield high performance on parallel platforms. Note that it has already been argued that these methods should have considerable advantages in parallel implementation over the alternatives, eg. the tridiagonal methods (Strazdins 2000).

The main original contributions of this paper is firstly to show how a symmetric indefinite factorization algorithm with strong stability guarantees may be efficiently parallelized, and secondly to show how parallel and serial performance for dense matrices can then be improved by adapting techniques used previously used only in the sparse serial case. An analysis of these methods and comparison with established algorithms are also given.

This paper is organized as follows. Section 2 describes the main diagonal pivoting method, with the development of the parallelization of one stable algorithm this method, being given in Section 3. Section 4 describes the technique of searching for pivots in the current elimination block, which offers potential for improved parallel and serial performance. An analysis of the pivoting behavior of these algorithms is given in Section 5, with performance results then being given in Section 6. Conclusions are given in Section 7.

# 2 DIAGONAL PIVOTING ALGORITHMS

In the diagonal pivoting method, the decomposition $A = PLDL^TP^T$ is performed, where $P$ is a permutation matrix, $L$ is an $N \times N$ lower triangular matrix with a unit diagonal, and $D$ is a block diagonal matrix with either $1 \times 1$ or $2 \times 2$ sub-blocks (Golub & Loan 1989).

Figure 1 indicates the partial factorization of a blocked algorithm. The dashed trapezoid in Figure 1 represents the panel $A^L$ of the matrix currently being factored.

The factorisation of $A$ proceeds column by column; in the elimination of column $j$, 3 cases arise:

1. Eliminate using a $1 \times 1$ pivot from $A_{j,j}$. This corresponds to the definite case, and will be used when $A_{j,j}$ is sufficiently large (compared with $\max(A_{j+1:m,j})$).

2. Eliminate using a $1 \times 1$ pivot from $A_{i,i}$, where $i > j$. This corresponds to the semi-definite case; a symmetric interchange with row/columns $i$ and $j$ must be performed.

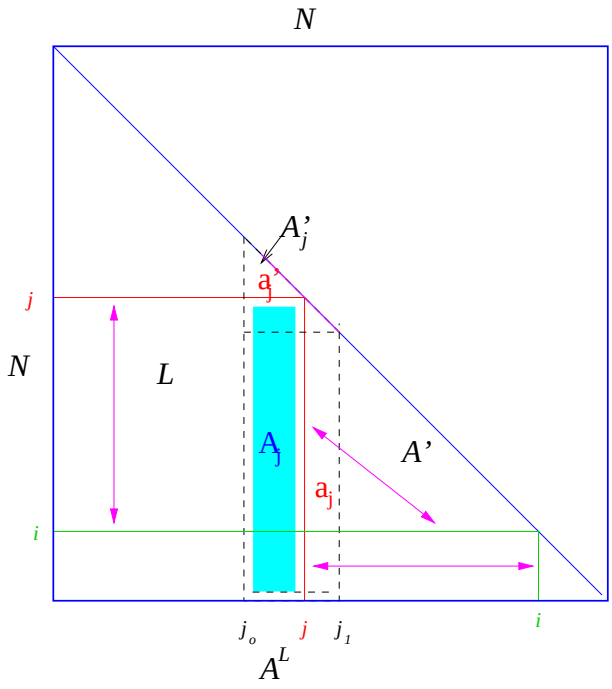


Figure 1: Partial LDLT factorization showing a symmetric interchange between columns $j$ and $i$

3. Eliminate using a $2 \times 2$ pivot using columns $i'$, $i' \geq j$, and $i$, $i \geq j+1$ (this case produces a $2 \times 2$ sub-block at column $j$ of $D$) This corresponds to the indefinite case; a symmetric interchange with rows/columns $i'$,$i$ and and $j, j+1$ is performed.

The tests used to decide between these cases, and the searches used to select column $i$ (and $i'$), yield several algorithms based on the method, the most well-known being the several variants of the Bunch-Kaufman algorithm.

The Bunch-Kaufman algorithm restricts $i' = j$ and always finds a column $i$ for cases 2 or 3 within a single search; for this reason, it can be implemented very efficiently (Anderson & Dongarra 1989, Jones & Patrick 1991, Strazdins 2000).

However, it has been recently shown for this algorithm there is no guarantee that the growth of $L$ is bounded, and hence no overall strong stability guarantee (Ashcraft et al. 1998). Variants such as the *bounded Bunch-Kaufman* and *fast Bunch-Parlett* algorithms have been devised which overcomes this problem (Ashcraft et al. 1998). The extra accuracy of these methods results from more extensive searching for stable pivot columns $i$ (and $i'$) for cases 2 and 3, with a correspondingly more frequent use of these cases.

For linear systems that are close to definite, the diagonal pivoting methods permit most columns to be eliminated by case 1, requiring no symmetric interchanges. For parallel implementation, this is a highly useful property, as even for large matrices, the communication startup and volume overheads of symmetric interchange can be large (Strazdins 1999).

Instead of suppressing interchanges, which even if done judiciously may result in the loss of some accuracy (Strazdins 2000), high parallel performance can also be achieved with a *block-search* algorithm that searches for suitable pivot columns $i$ and $i'$ from the current storage block. If this search was successful, the symmetric interchanges would require no

communication, resulting in no parallel overhead. Such a strategy could be based on the exhaustive pivot search strategy used for sparse matrices (Ashcraft et al. 1998), which also has strong guarantees of accuracy.

However, if the search was not successful, an equally stable means of eliminating column $j$ must then be used. We chose the *bounded Bunch-Kaufman* algorithm over the *fast Bunch-Parlett* algorithm, as the latter requires sorting of the columns by the size of the diagonal, which would give it higher parallel overheads.

# 3 THE PARALLEL BOUNDED BUNCH-KAUFMAN ALGORITHM

This section describes how to efficiently parallelize the bounded Bunch-Kaufman algorithm of (Ashcraft et al. 1998). Figure 2 indicates the computations corresponding to the partial factorization in Figure 1. It gives a more complete description than in (Ashcraft et al. 1998), indicating how it is used in conjunction with a left-looking level 2 factorization incorporated into a block algorithm. Note that $w_j$, $w'_j$ and $w'_i$ refer to parts of $W$ corresponding to $a_j$, $a'_j$ and $a'_i$. For the sake of simplicity, it is assumed that the input matrix is invertible.

A new feature in our implementation is the use of 'half interchanges' when searching for suitable candidate columns. The formation of $v$ corresponds to a 'half interchange', with $a_i$ denoting the remaining part of row/column $i$, ie. $a_i = (A_{i,j:i}, A_{i+1:N-1,i})$. The completion of the interchange $j \leftrightarrow i$ would then involve $a_i \leftarrow a_j$, plus the double row swap $a'_j, w'_j \leftrightarrow a'_i, w'_i$.

For case 3, the condition $\gamma_i = \gamma_r$ indicates that the new value of $A_{j+1,j}$ is the maximum of the new columns $j$ and $j+1$: this ensures that the growth from scaling these columns by $D_j^{-1}$ is bounded above by a factor of 2, when the growth parameter $\alpha = 0.5$ (Ashcraft et al. 1998). The `repeat` loop must terminate in at most $N - j$ steps as the value of $\gamma_i$ must be monotonically increasing each iteration; previous empirical studies have shown that on average it terminates in only 2.5 iterations (Ashcraft et al. 1998), which concurs with our experience also.

The 'half-interchanges' in the above implementation thus avoid some of the overhead of a full interchange occurred on each time step, particularly useful for parallel implementation. Note that this also saves twices as many interchanges when using the factored matrix to perform a linear system solve. It also ensures that at most one interchange occurs per index position, enabling $P$ to be represented by a standard pivot vector.

The matrix operations of Figure 2 are coded entirely in terms of the DBLAS Distributed BLAS Library (Strazdins 1996, Strazdins 1997, Strazdins 1998), which is a portable version of parallel BLAS. The DBLAS routines already optimize much of the communication behavior of these operations; a description of how they are parallelized can be found elsewhere (Strazdins 1996, Strazdins 1998).

Other optimizations used to reduce communication startups in the ordinary Bunch-Kaufman factorization (Strazdins 1999) have been similarly performed on this implementation; an optimization applying particularly in this case occurs on case 3 when $j \neq i'$ and $j + 1 \neq i$. Here, two 'half-interchanges' are used to form $v$ on the last two iterations; upon loop exit, they are simultaneously completed. To compensate for the slight difference from two consecutive

create a new matrix $W$ aligned with $A^L$
for each $j$ in current panel
  $w_j \leftarrow (w'_j)^T A_j + a_j$; $i' = k$
  find the pos. $i$ of the max. in $w_j$; $\gamma_i = |(w_j)_i|$
  `if` $|A_{j,j}| < \alpha\gamma_i$, (not **case 1**)
    `repeat`
      create new vector $v$ aligned with $w_j$
      $v \leftarrow (w'_i)^T A_j + a_i$
      find the pos. $r$ of the max. in $v$; $\gamma_r = |v_r|$
      `if` $|v_i| < \alpha\gamma_r$, (**case 2**)
        $w_j \leftarrow v$; complete interchange $j \leftrightarrow i$
      `else if` $\gamma_i = \gamma_r$ (**case 3**)
        $w_{j+1} \leftarrow v$;
        complete interchanges $j, j+1 \leftrightarrow i', i$
      `else`
        $w_j \leftarrow v; i' \leftarrow i; \gamma_i = \gamma_r; i \leftarrow r$
    `until` **case 2** or **3** applies;
  `if` **case 3** applies
    $D_j = \begin{pmatrix} A_{j,j} & A_{j+1,j} \\ A_{j+1,j} & A_{j+1,j+1} \end{pmatrix}$
    $(a_j, a_{j+1}) \leftarrow (w_j, w_{j+1}) D_j^{-1}$
    skip column $j+1$
  `else`
    $a_j \leftarrow w_j / A_{j,j}$
  $A'$`-=`$A^L W^T$

Figure 2: Partial Bounded Bunch-Kaufman factorization of an $N \times N$ symmetric indefinite matrix $A$

full interchanges, it is sufficient to simply extend the row swap $a_i \leftrightarrow a_j$ to include elements in column $j+1$, thus requiring no extra communications.

It is possible to adopt the algorithm of Figure 2 to perform a 'right-looking' level 2 factorization, which means that each of the the $\omega$ columns in $W$ are maintained at each iteration using rank-1 updates. Interchanges must then be performed on $W$ as they are in $A$, considerably complicating the code (and adding some overhead). For this reason, and the fact that generally rank-1 updates are considerably slower than matrix-vector multiplies (see Figure 2), right-looking factorizations are generally not used in implementing the diagonal pivoting method. However, in Section 4, we will show that they can have an advantage under some circumstances.

# 4 BLOCK SEARCH METHODS

Consider a parallel implementation of the bounded Bunch-Kaufman algorithm on a $P \times Q$ processor grid, with $A$ being distributed using and $r \times r$ block-cyclic matrix distribution, ie. block $(i,j)$ of $A$ will be on node $(i \bmod P, j \bmod Q)$. We choose an algorithmic blocking factor $\omega = r$; thus $W$ and $A^L$ would be contained in a single node column (cf. Figure 1).

In cases 2 and 3, the interchange $j \leftrightarrow i$ may require parallel communication if $i$ is outside the current storage block of column $j$. Furthermore, there will be computational overheads implied in the computation of $v$ in the left-looking algorithm (on average requiring $\frac{N\omega}{2}$ floating point operations each time). These two form the major overheads of the algorithm.

However, if we were to form $W$ using a right-looking variant, and we could find suitable pivot columns within the current storage block, all interchanges would then involve only local memory movements, and both of these forms of overheads could be removed.

We will now look at suitable methods already existing for sparse matrices, and then explore how they may be efficiently adapted for the parallel dense case.

## 4.1 *Sparse Methods*

A method performing a block search in the sparse case is the *exhaustive pivoting strategy* (Ashcraft et al. 1998), which originates from the earlier MA27 codes by Duff and Reid (Duff & Reid 1983). The block search technique is used in multifrontal methods, where the ability to select a pivot from the current block of columns has a large payoff in preserving the sparsity of the matrices (Ashcraft et al. 1998).

Here, a block of $\omega$ columns currently under consideration for elimination are arranged in a queue. Each iteration of the search selects column $i$ (initially from the head of the queue, working inwards). After computing $\gamma_i$, it is then matched with each column $i'$ preceding it in the queue. Note that at this point, it must be ensured that all updates from previous eliminations have been applied to columns $i$ and $i'$. The condition:

$$\begin{aligned} \max\{\gamma_i|W_{i'i'}| + \gamma_j|W_{ii'}|, \gamma_j|W_{ii}| + \gamma_i|W_{ii'}|\} \\ \leq |W_{i'i'}W_{ii} - W_{ii'}^2|/\alpha \end{aligned} \tag{1}$$

determines that columns $i$ and $i'$ can form a stable $2 \times 2$ pivot (Ashcraft et al. 1998). After this, column $i$ may be considered for a $1 \times 1$ pivot; the same test as for case 1 in Figure 2 is used.

If a stable pivot is found, then the associated columns are removed from the queue and the search is terminated; otherwise column $i$ is put at the end of the queue (the search will also be terminated when this occurs for each of the $\omega$ columns). Thus, this search can potentially look at all $\frac{\omega(\omega+1)}{2}$ potential pivots, the dominant cost in this case will be the computation of $\omega$ column maximums. It has been found that this exhaustive search was more effective for sparse matrices than using a limited search (over $2\omega - 1$ potential pivots, as is done in the MA27 codes), where search for the $2 \times 2$ pivot for column $i$ is restricted to a single column $i'$ where $W_{i'i} = \max\{W_{i+1:\omega-1,\,i}\}$ (Ashcraft et al. 1998).

Priority is given to $2 \times 2$ pivots, where a $1 \times 1$ pivot will not be checked until $m_{2\times2}$ $2 \times 2$ pivots for column $i$ have been searched, where a recommended value for the heuristic parameter $m_{2\times2} = 5$ (Ashcraft et al. 1998). The reason given there for this bias is that the operations for $2 \times 2$ pivots are faster.

One remaining point is what to do if no stable pivots are found in the current block. A number of fresh columns (from the trailing sub-matrix $A'$) will be then brought to the head of the queue, and the whole process will then be repeated. While the queue size $\omega$ may thus grow to be $O(N)$, eventually the whole factorization will complete (Ashcraft et al. 1998).

It should be noted that in the sparse serial case, the data structures used permit interchanges (including queue insertions and deletions) with little overhead.

## 4.2 *Adaption for the Dense Parallel Case*

The exhaustive search strategy mentioned in the previous section could be adopted for this purpose. However, to realize an actual performance improvement over our bounded Bunch-Kaufman implementation, which apart from the overheads mentioned in Section 4, uses fast computational components, the following principles must also be observed:

1. the new algorithm should not use significantly slower computational components. In particular, it should use existing optimized BLAS and Distributed BLAS kernels (requiring packed storage).

2. the optimal target blocking factor $\omega$ for the matrix multiply $A'$-=$A^L W^T$ should be guaranteed to be reached. This guarantees that optimal speed for the dominant part of the overall computation is achieved.

3. any new overheads introduced should be minimized.

Thus, from the first principle, the use of queues for managing the current block should be avoided, as it will lead in this case to a complex and/or inefficient implementation, as we require the packed matrix storage for computational speed. Instead, when eliminating column $j$, if we find a suitable $2 \times 2$ pivot $i', i$ in the block, the interchanges $j, j+1 \leftrightarrow i', i$ are performed instead (and similarly for a $1 \times 1$ pivot).

Secondly, if a suitable pivot cannot be found at column $j$ in the current block, the bounded Bunch-Kaufman algorithm will then be used to eliminate column $j$. Note that in the dense

parallel case, our blocksizes are fixed by the block-cyclic matrix distribution; using the bounded Bunch-Kaufman as a 'fall-though' ensures a fixed blocking factor can always be reached. Provided the block search is normally successful, this will only slow down the overall computation marginally. In this hybrid algorithm, the growth bounds are then the maximum of those in either algorithm, that the elements of $L$ are bounded by $\max\{\frac{1}{\alpha}, \frac{1}{1-\alpha}\}$ and the growth of the trailing sub-matrix at each step is bounded by $\frac{1}{\alpha}$ (Ashcraft et al. 1998).

Thirdly, in order to compute $\gamma_{j:i}$ and $W_{j:i,j:i}$ efficiently at each search, a right-looking factorization should be used (it turns out that this need apply only to $W$, not $A^L$ as well). This is a potential problem, as it seems to violate the first principle (see the end of Section 3). The computation of the $\gamma$'s, which in the worst case could occur $\frac{\omega(\omega+1)}{2}$ times per column, is the main source of new overhead. In the parallel context, to compute a new $\gamma_i$, a reduction operation is required, needed $\lg_2 P$ communication startups to bring the value of $\gamma_i$ to the node holding $W_{j:i,j:i}$, which then has all information required to evaluate the search. The value of $\gamma_i$ plus the result of the search at column $i$ is then broadcast to all nodes in the column, requiring a further $\lg_2 P$ startups. These startups can be amortized by a factor of $b_s$ by finding $\gamma_{i:i+b_s-1}$ at the same time; the tradeoff is in performing some potentially unnecessary column scans in case a pivot is found before column $i + b_s - 1$. Empirical studies have shown that $b_s = 4$ is a good compromise. Finally, the overall result of the search is then broadcast to all nodes.

We can limit the worst-case overhead of the $\gamma$ calculations to $\approx \omega\omega_s$ searches per block, where $1 \leq \omega_s \leq \frac{\omega}{2}$, by limiting the search from column $j$ to column $j + \omega_s - 1$. This is due to the observation that most successful searches generally terminate relatively early.

While in itself this was found to have made little difference to the overall performance, this idea permits a hybrid left-and-right looking implementation: if upon the current search, this was the first time column $i$ was accessed, the (left-looking) updates from the elimination of previous columns in the block are applied before the search was performed. From then on, column $i$ of $W$ is maintained in a right-looking fashion. This maximizes the use of left-looking updates (especially for weakly-indefinite matrices where case 1 predominates), while maintaining all advantages of the block search.

The same bias towards $2 \times 2$ pivots as described in Section 4 was implemented here, as we also found it optimal in the dense case. We found that biasing the search towards $1 \times 1$ pivots caused 'cascades' of occurrences of case 2 (see Section 5). The second main advantage of the $2 \times 2$ bias is that it will reduce the overhead of calculation of $\gamma_i$, as two columns rather than one get eliminated on a successful search.

# 5 PIVOTING BEHAVIOR

In order to understand the performance potential of the Bunch-Kaufman, bounded Bunch-Kaufman and block search algorithms, it will be useful to first investigate statistics indicating their pivoting behavior. For this study, we choose the block size $\omega = 64$, the matrix size $N = 1000$, and use simulated matrices of the form $A = A' + \beta I$, where $A'$ has double precision random elements from $[-1, 1]$ and $0 \leq \beta$.

It is useful to define three quantities related to pivoting choices and overheads. $f$ represents the number of interchanges and half-interchanges (scaled by 0.5) divided by $N$. The number of

interchanges are determined from the number of cases 2 and 3 (an instance of case 3 where $j+1 \neq i$ would add 1 to this count if $j = i'$, and 2 if $j \neq i'$).

$f'$ is like $f$ but only those interchanges where $i$ (or $i'$) are outside the current storage block are counted. Regarding the cost of an interchange within a storage block (requiring no communication) as negligible, $f'$ then indicates the parallel overheads of interchanges. A low value of $f'$ indicates a high ratio of successful block searches.

The quantity $s$ represents the number of column maximum finding operations performed by the algorithm, divided by $N$. This represents source of overhead in the computation, particularly for the block search.

Figure 3 gives a comparison of these quantities for the Bunch-Kaufman (BK), bounded Bunch-Kaufman, (BBK) and block search-augmented bounded Bunch-Kaufman (BS) algorithms; each point represents the averaged value from 20 such randomly generated matrices. For larger $N$, all quantities tend to increase slightly with increasing $\beta$, but the relative differences are the same.

A useful concept to introduce at this point is that of short-range and long-range interchanges in the diagonal pivoting methods. By the 'range', we mean the difference between the two indices involved; interchanges within the current storage block are thus short-range. As mentioned earlier, short-range interchanges may thus be more desirable in terms of performance than long-range. However, consider case 2 with the interchange $j \leftrightarrow i$. This means that column $j$ was unsuitable as a $1 \times 1$ pivot, because at that time, its diagonal element was relatively weak compared with its off-diagonal maximum. This interchange would then result in the original column being again considered as a potential $1 \times 1$ pivot after $i - j$ updates. If $i$ is close to $j$, the probability is high that the column will still prove unsuitable as a $1 \times 1$ pivot, and case 2 could again be selected. This could result in 'cascades' of case 2 pivots. On the other hand, if $i$ is far away from $j$, this is much less likely to occur as there will be more updates applied to the original column, amplified by the fact that the column is now considerably shorter.

The value for $f$ for BBK is higher than BK for low $\beta$; this indicates the effect of the BBK search taking several iterations before a suitable pivot is found, whereas BK always finds a pivot effectively in a single iteration (and loses its stability guarantees thereby). As mentioned in Section 4.2, our block search implementation is limited to $\omega_s = 16$ columns ahead, rather than in the full block. This reduced the value of $f'$ by $\leq 0.015$, indicating that the limited search is at least as effective, at least under the current circumstances.

The value of $f'$ being much lower for BS indicates the block search has a high success rate, even for low $\beta$. However, its value of $f$ being significantly greater than BK is a point for concern: while it does not in this case (because of the low $f'$) imply computational (in the form of extra matrix multiplies) or communication overheads, it does indicate some extra overhead at least in terms of local memory movements.

The reason why $f$ is high for BS is unclear. One hypothesis is that as the block search generates a larger number of cases 2 (and possibly cases 3), as unsuitable columns sometimes get repeatedly shuffled towards the end of the block, due to the short-range nature of the interchanges. An experiment was performed to cause the block search to fail if the current column was already moved twice within the same block by occurrences of case 2. This in fact resulted in a slightly increased value of $f$ (and indeed $f'$), not supporting the hypothesis (and also showing that this strategy would not improve the block search).

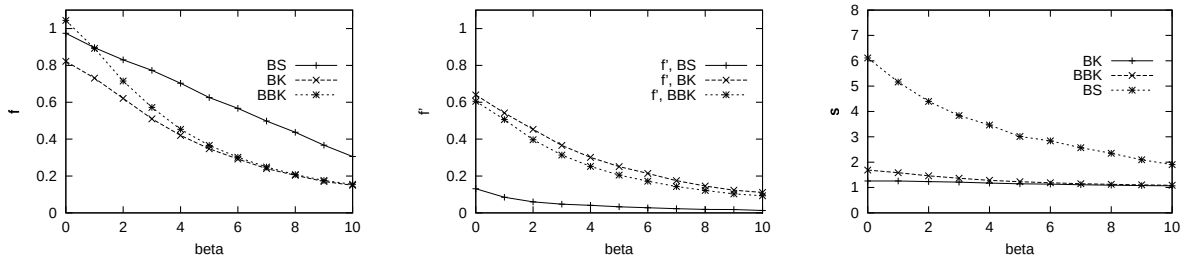


Figure 3: Averaged pivoting and search overhead parameters for simulated $1000 \times 1000$ matrices

The value of $s$ is, as expected, significantly higher for BS, from 6 times at $\beta = 0$, decreasing to twice, at $\beta = 10$. Depending on the relative slowness of the vector maximum finding function (for the UltraSPARC BLAS, it is only about twice as slow as a vector copy (Strazdins 1998)) this may or may not represent a significant overhead. Note that the average number of vector maximum calls per search can be approximated by $s/f$, as the block search is not invoked at column $j$ when case 1 occurs.

The computational overhead of the extra vector-matrix multiplies which BK (and BBK) perform over BS can be approximated by:

$$m_{\mathrm{BK}} \approx (f_{\mathrm{BK}} - f'_{\mathrm{BS}}) \frac{\omega s_{\mathrm{mx}}}{2 s_{\mathrm{mv}}}$$

where $\frac{s_{\mathrm{mx}}}{s_{\mathrm{mv}}}$ represents the speed per element of matrix-vector multiply over vector maximum finding. For the UltraSPARC BLAS, $\frac{s_{\mathrm{mx}}}{s_{\mathrm{mv}}} \approx 0.25$. Under these conditions, we see that $m_{\mathrm{BK}} \approx 6$ and $m_{\mathrm{BBK}} \approx 8$ at $\beta = 0$, indicating indeed a comparable time penalty as given by the extra search overheads of BS, $s_{\mathrm{BK}} - s_{\mathrm{BS}}$ and $s_{\mathrm{BBK}} - s_{\mathrm{BS}}$, respectively.

# 6 PERFORMANCE

The Fujitsu AP3000 (Ishihata, Takahashi & Sato 1997) is a distributed memory multicomputer, comprised of RISC scalar processors (UltraSPARC) with a deep memory hierarchy (having a 16KB top-level data cache and a 1MB 2nd-level cache, both direct-mapped, and a 64-entry TLB). It has communication networks with characteristics shared by most other state-of-the-art distributed memory computers, that is, high communication costs relative to floating point speed, and row or column broadcasts having to be simulated by point-to-point messages. The AP3000 also has many properties of the cluster computing model; this extra flexibility contributes to its communication costs.

Figure 4 gives performance results on an AP3000 based on 200MHz UltraSPARC II nodes for double precision data. These are for a combined factorization and solve computation, as pivoting advantages gained in the factorization will also benefit the solve stage. A storage block size $\omega = r = 64$ was fond to be optimal, with the block search parameter $\omega_s = 16$. Highly tuned UltraSPARC BLAS, capable of sustaining 300 MFLOPs on large matrix-matrix multiply, was used (Strazdins 1998).

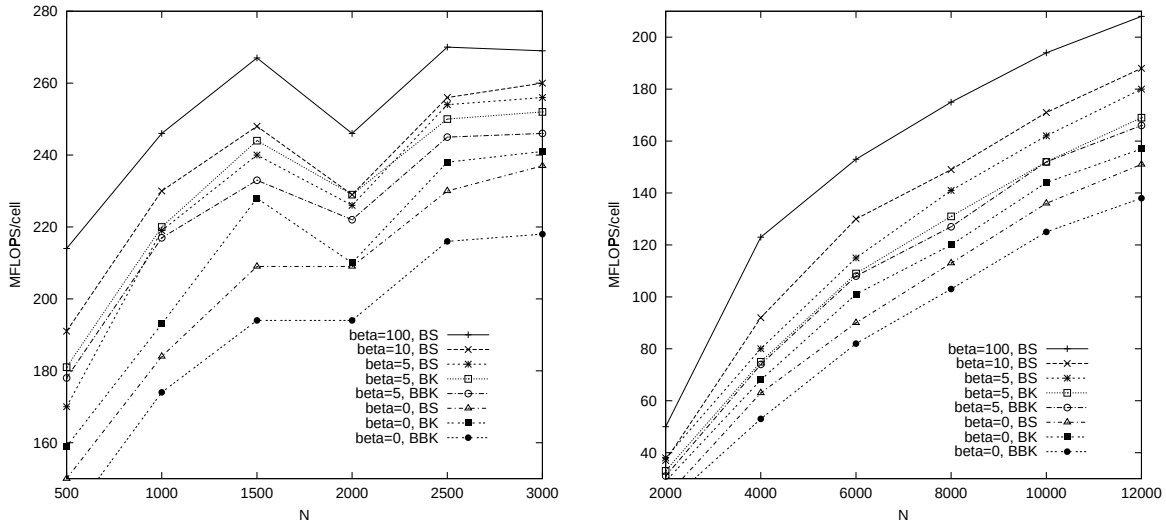


Figure 4: Performance of LDLT algorithms on $1 \times 1$ (left) and $4 \times 4$ (right) AP3000 for simulated $N \times N$ matrices

On the $4 \times 4$ processor grid, at $\beta = 0$, BK out-performs BS by $\approx 10\%$, and BS in turn out-performs BBK by the same margin. At $\beta = 5$, BS out-performs BK by $\approx 6\%$, and BBK by $\approx 8\%$. All three algorithms have essentially the same performance at $\beta \geq 10$. This is because their pivoting behavior is almost the same for these matrices.

The differences between the serial ($1 \times 1$ grid) performance is similar, except the margins are a little smaller between BK and BS at $\beta = 0$, and, at $\beta = 5$, BS is only faster by a modest margin for $N \geq 2500$.

Without a hybrid right-and-left looking level 2 factorization in BS, BS generally performed 5–10% worse than BBK for $\beta \geq 5$, showing how important the second principle of Section 4.2 is in practice. At $\beta = 0$, the high frequency of block searches means that there is a higher fraction of right-looking updates in BS than for larger $\beta$; this plus the overhead of $s \approx 8$ for these matrix sizes, is countering its inherent pivoting advantage over BK here.

# 7 CONCLUSIONS

In this paper, we have demonstrated how the bounded Bunch-Kaufman algorithm may be efficiently parallelized. Despite its overhead, due to its performing extra (long-range) searches for pivots, over the Bunch-Kaufman algorithm, it is still reasonably competitive in terms of its performance, especially for matrices close to definite. This we deem an acceptable cost for the extra stability guarantees of the former.

By augmenting the bounded Bunch-Kaufman with a block search procedures, we retain the stability guarantees but get significant improvement in serial and parallel performance. This is achieved by exploiting short-range searches to avoid wasted matrix-vector multiplies and reduce communication costs in the symmetric interchanges.

However, the block search increased the overall number of interchanges, resulting in some extra memory movement overheads. The reason for this increase is yet unknown.

The block search technique was derived from the exhaustive block search strategy for serial sparse matrices; however because the tradeoffs (potential gains and the nature of overheads) are very different, considerable effort was required for its efficient adaption to the dense parallel case. It relies on using a blocked algorithm where the blocking factor equals the block-cyclic distribution's storage block size. The most serious overhead remaining is the computational and communication overheads in the extra column maximum finding operation, as our current implementation can increase the number of these by up to 8 times.

Future work would be to try to reduce the number of searches. For example, if the test of Equation 4.1 fails when $\gamma_i$ is substituted by 0, there is no need to calculate $\gamma_i$. Alternatively, if the value of $\gamma_i$ was calculated in a previous block search, that old value could be substituted instead for a sharper test. Another strategy is to search for most likely $2 \times 2$ pivots first; a useful heuristic from sparse techniques is to find the maximum element in column $i$ from the diagonal block.

Another possible improvement, which could increase $f'$ for strongly indefinite matrices, would be to imitate the sparse strategy more closely by moving columns for which a block search failed down to the end of the storage block. These could then be exchanged with 'fresh' columns from the $(k * \mathrm{LCM}(\mathrm{P}, \mathrm{Q}))$th next global storage block, where $k > 0$. While this would involve very little communication, it would not avoid wasted computation (matrix-vector multiplies).

Finally, provided a high value of $f'$ can be sustained, the exhaustive block-search technique may be extremely useful in implementing an efficient and stable (parallel) out-of-core LDLT factorization algorithm, where only a section of columns may be in memory at once. This is because finding a stable pivot in the current elimination block also means it is in the section, and thus avoiding the large penalty in bringing in an out-of-core pivot into memory for consideration. For this reason, it may also be useful to extend the search to the remainder of the section, if the search within the elimination block fails.

# ACKNOWLEDGEMENTS

The authors would like to thank the Fujitsu Parallel Research Computing Facilities for the use of a 16 node AP3000. The authors would also like to thank the anonymous referees for their suggestions for improving the manuscript.

# REFERENCES

ANDERSON, C. & DONGARRA, J. (1989): Evaluating Block Algorithm Variants in LAPACK: *in* 'Fourth SIAM Conference for Parallel Processing for Scientific Computing': Chicago. 6 pages.

ASHCRAFT, C., GRIMES, R. G. & LEWIS, J. G. (1998): Accurate Symmetric Indefinite Linear Equation Solvers: *SIMAX* **20**(2), 513–561.

DUFF, I. & REID, J. (1983): MA27: A Set of Fortran Subroutines for Solving Sparse Symmetric Sets of Linear Equations: Technical Report AERE R 10533: Harwell.

GOLUB, G. & Van LOAN, C. (1989): *Matrix Computations*: second edn: John Hopkins University Press: Baltimore.

ISHIHATA, H., TAKAHASHI, M. & SATO, H. (1997): Hardware of the AP3000 Parallel Server: *Fujitsu Scientific and Technical Journal* **33**(1), 24–29.

JONES, M. T. & PATRICK, M. L. (1991): Factoring Symmetric Indefinite Matrices on High-Performance Architectures: *SIAM Journal on Matrix Analysis and Applications* **12**(3), 273–283.

KAUFMAN, L. (1995): Computing the $MDM^T$ decomposition: *ACM Transactions on Mathematical Software* **21**(4), 476–489.

STRAZDINS, P. (1996): A High Performance, Portable Distributed BLAS Implementation: *in* 'Sixth Parallel Computing Workshop': Fujitsu Parallel Computing Research Center: Kawasaki: pp. P2–K–1 – P2–K–10.

STRAZDINS, P. (1997): Reducing Software Overheads in Parallel Linear Algebra Libraries: *in* 'The 4th Annual Australasian Conference on Parallel And Real-Time Systems': Springer: Newcastle Australia: pp. 73–84.

STRAZDINS, P. E. (1998): Transporting Distributed BLAS to the Fujitsu AP3000 and VPP-300: *in* 'Proceedings of the Eighth Parallel Computing Workshop': School of Computing, National University of Singapore: Singapore: pp. 69–76. paper P1-E.

STRAZDINS, P. E. (1999): Parallelizing Dense Symmetric Indefinite Solvers: *in* 'PART'99: The 6th Annual Australasian Conference on Parallel And Real-Time Systems': Springer-Verlag: Melbourne: pp. 398–410.

STRAZDINS, P. E. (2000): Accelerated methods for performing the LDLT decomposition: *ANZIAM* **42**(C), C1328–C1355.