THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-99-01

A Dense Complex Symmetric Indefinite Solver for the Fujitsu AP3000

Peter E. Strazdins

May 1999

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

TR-CS-98-14 Michael Stewart. *A completely rank revealing quotient* urv *decomposition.* December 1998.

TR-CS-98-13 Michael Stewart. *Finding near rank deficiency in matrix products.* December 1998.

TR-CS-98-12 Vadim Olshevsky and Michael Stewart. *Stable factorization of Hankel and Hankel-like matrices.* December 1998.

TR-CS-98-11 Michael Stewart. *An error analysis of a unitary Hessenberg* QR *algorithm.* December 1998.

TR-CS-98-10 Peter Strazdins. *Optimal load balancing techniques for block-cyclic decompositions for matrix factorization.* September 1998.

TR-CS-98-09 Jim Grundy, Martin Schwenke, and Trevor Vickers (editors). *International Refinement Workshop & Formal Methods Pacific '98 — Work-in-progress papers of IRW/FMP'98, 29 September – 2 October 1998, Canberra, Australia.* September 1998.

A Dense Complex Symmetric Indefinite Solver for the Fujitsu AP3000

Peter E. Strazdins
peter@cs.anu.edu.au

Department of Computer Science, Australian National University

May 28, 1999

Abstract

This paper describes the design, implementation and performance of a parallel direct dense symmetric-indefinite solver routine. Such a solver is required for the large complex systems arising from electro-magnetic field analysis, such as are generated from the AccuField application. The primary target architecture for the solver is the Fujitsu AP3000, a distributed memory machine based on the UltraSPARC processor.

The routine is written entirely in terms of the DBLAS Distributed Library, recently extended for complex precision. It uses the Bunch-Kaufman diagonal pivoting method and is based on the LAPACK algorithm, with several modifications required for efficient parallel implementation and one modification to reduce the amount of symmetric pivoting. Currently the routine uses a standard BLAS computational interface and can use either the MPI, BLACS or VPPLib communication interfaces (the latter is only available under the APruntime V2.0 system for the AP3000).

The routine out-performs its equivalent LAPACK routine `zsysv()` by 14% when run on a 300 MHz UltraSPARC processor for a matrix of order 1601 and a single right hand size. Comparing from the `zsysv()` from Sun Performance Library 1.2, the overall speed gain is 55%, by the use of faster BLAS kernels recently developed at ANU. Thus a speed of 436 (double precision) MFLOPs is possible, with an execution time of 12.5s.

Using run-time settable parameters, the routine can use any logical $P \times Q$ processor grid, any (square) storage block size r and any algorithmic block size ω. This enables performance tuning via trading off load balance and communication penalties, the latter being relatively higher than for LU or LLT solvers. For a matrix of order 10000 on a 16-node AP3000, best performance was achieved with $P = Q = 4$, $r = \omega = 44$ with an execution time of 254s. This represents a sustained speed of 5.2 GFLOPs and a parallel speedup of 12. The main obstacle to a higher speedup on the AP3000 is communication volume overheads.

Future work includes improving the effective bandwidth of various communication primitives, by directly manipulating the AP3000 SDRAM and KMEM message buffering memory, which can be accessed via low-level routines of the APruntime V2.0 run-time system. Also to be investigated is an enhanced diagonal pivoting algorithm which has a 'lookahead' over a block of columns, which may yield both computational and communication advantages, enabling the solver to truly approach LLT speed. This can occur if a (very) high fraction of the diagonal elements of the matrix are large (but the matrix is not necessarily positive definite), as may be the case with the AccuField matrices.

1 Introduction

Large symmetric indefinite systems of equations arise in many applications, including incompressible flow computations and optimization of linear and non-linear programs. They also arise in electro-magnetic field analysis, in finding a stationary solution for Maxwell's Equation by

the "moment method" [13]. In this case, the systems are also complex, and if the fields arise from electrical circuit components, the imaginary part of the diagonal elements, representing the self-impedance and self-admittance of that element, may be relatively large.

The AccuField (AF) system is one such application [13]. Sample matrices from this application so far indicate the matrices could be classed as 'weakly indefinite', that is most diagonal elements are sufficiently large relative to their off-diagonal entries. This means that transformations appropriate to definite systems can be applied to eliminate most columns of these matrices, without sacrificing numerical stability. This property can yield important computational advantages, and has an important impact on this work. The order of the matrices generated by this application can be very large, eg. $N \approx 30000$ [13].

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 [9] and the references within, especially [1, 6]). While several performance evaluations of variants of these algorithms have been given [4, 2, 11, 12, 3], all but [11] consider only uniprocessor implementations, and [11] only considers parallelization on a small-scale shared memory machine.

In this paper, we describe how to efficiently parallelize Bunch-Kaufman diagonal pivoting method for solving symmetric indefinite systems. Our goal is to deliver high actual serial and parallel performance, especially for 'weakly indefinite' systems and especially for the Fujitsu AP3000. Our approach is as follows:

- select a promising serial algorithm
- parallelize this algorithm, which involves both:
  - minimizing communication overhead, while optimizing memory performance
  - selecting an optimal load balancing technique
- optimize the low-level computational components (BLAS) for the AP3000
- optimize low-level communication components for the AP3000
- devise and investigate variants of the basic algorithm likely to yield improved performance, especially for 'weakly indefinite' systems.

The parallel solver is coded entirely in terms of the DBLAS Distributed BLAS Library [5, 17, 18], which is a portable version of parallel BLAS that was originally written to be tuned for the Fujitsu AP1000 and AP+, and has been recently ported and tuned to other platforms, including the AP3000 [21]. It has been used to implement very efficient parallel matrix factorization applications using various techniques [19, 20]. The use of the DBLAS has enabled rapid development and prototyping of several variants of the Bunch-Kaufman algorithm, while enabling high reliability and performance from its highly tested and optimized components.

The Fujitsu AP3000 [10] 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.

For implementation on the AP3000, an interface to the MPI and VPPLib communication libraries has been written for the DBLAS. The VPPLib interface is useful because VPPLib is implemented, with very low additional overhead, in terms of LWSLT low-level library [14], which has yielded latencies of $26.3\mu s$ and bandwidths (for large messages) of 64.5 MB/s on a U170-based AP3000 [14].

The main original contributions of this paper are as follows: it presents a new variant of the diagonal pivoting algorithm that yields superior serial performance; it provides an analysis of the issues of symmetric indefinite solvers for distributed memory platforms; to the authors knowl-

edge, it provides the first distributed memory implementation of a dense symmetric indefinite solver, exploring the issue of blocking methods; and it outlines an enhanced algorithm (largely based on ideas from previously published work) which can yield further performance improvements, serial or parallel, for the case of 'weakly indefinite' matrices.

This paper is organized as follows. Section 2 discusses issues in the choice of known serial algorithms, describing the diagonal pivoting method and general performance issues. Parallelization of the diagonal pivoting method is described in Section 3, with its implementation and serial and parallel performance of this algorithm discussed in 4. Section 5 discusses future work proposed at the time of writing, with conclusions being given in Section 6

2 Choice of Algorithms

In the 1970's, two efficient and stable algorithms for symmetric indefinite systems were proposed and refined: the tridiagonal reduction method (Aasen's method [1]) and the diagonal pivoting method (the latest being the Bunch-Kaufman method [6]). Variants of the diagonal pivoting method have since been proposed [11, 12, 3], but the LAPACK implementation of the Bunch-Kaufman method [2] has proven to be very competitive in terms of performance with the newer methods over a range of platforms.

For the sake of brevity, the notations x' ($\tilde{x}$), for an integer expression x, is a shorthand for $x+1$ ($x-1$).

Aasen's method involves exploiting properties of Hessenberg matrices to perform the decomposition $A = LTL^T$, where A is an $N \times N$ symmetric matrix, L is an $\tilde{N} \times \tilde{N}$ lower triangular matrix with a unit diagonal, and T is a tridiagonal matrix [9]. For the same reasons as for LU decomposition, pivoting must be applied to the sub-diagonal portion of each column, with the diagonal elements always remaining in T.

The Bunch-Kaufman method performs the decomposition $A = LDL^T$, where L is an $N \times N$ lower triangular matrix with a unit diagonal, and D is a block diagonal matrix with either 1×1 or 2×2 sub-blocks [9]. A 2×2 sub-block indicates a 2×2 pivot was required for the stable elimination of the corresponding columns; the corresponding sub-diagonal element of L will be 0. In a practical implementation of this method, A can then be overwritten by L and D, with a 'pivot vector' recording any symmetric interchanges (including the position of the 2×2 pivots) [9, 2, 11].

In the elimination of column j, four cases can arise with the Bunch-Kaufman method:

D1 $|A_{jj}| \geq \alpha |A_{ij}|$, where $j < i < N$ and $|A_{i,j}| = \max_{k=j'}^{\tilde{N}} |A_{k,j}|$. Here, a 1×1 pivot from $A_{j,j}$ will be stable; no symmetric interchange is required.

D2 the conditions for D1 and D4 do not hold. Here, $A_{j,j}$ is used as a 1×1 pivot, and no symmetric interchange is required.

D3 A 1×1 pivot from A_{ii} will be stable. Here, a symmetric interchange with row / columns i and j must be performed.

D4 A 2×2 pivot using columns j and i will be stable. Here, a symmetric interchange with row / columns i and $j+1 = j'$ must be performed; however, both columns are eliminated in this step.

α is a tuning constant for the algorithm; it can be shown that $\alpha = \frac{1+\sqrt{17}}{8}$ maximizes stability of this algorithm [9, 6]. For definite systems, only case D1 is needed; case D3 is also needed for semi-definite systems, and case D4 is needed for indefinite systems.

Case D2 exists primarily to avoid a situation where case D4 might be unstable. By stability, it is meant that the growth of the trailing submatrix (A' in Figure 1) is bounded; however, due to cases D2 and D4 there is no guarantee that the growth of L is bounded [3]; recently a *bounded Bunch-Kaufman* algorithm has been presented which overcomes this problem [3].

The stability of both methods has been shown to be sufficiently high for most practical purposes [4]; choosing between them for a particular application is then essentially an issue of performance. In [4], a comparison of unblocked versions of these algorithms was given; their conclusion was there was no decisive difference under the tests performed. More recently, a comparison between blocked versions of these algorithms was given for a Cray 2 [2], showing the

LAPACK algorithm to be superior by ≈ 5% for large matrices. However, the source codes for the blocked Aasen's method used here have apparently been lost [3] and the nature of high performance computer architecture has changed considerably since that time. Taking this into account, and also that no compelling performance disadvantage of Aasen's method (for uniprocessors) has been given in the literature[1], an empirical comparison of the two methods for uniprocessors remains for future work.

However, in the case of 'weakly indefinite' systems, and especially for distributed memory architectures, a choice can be made between these methods, as we shall explain in the next section.

2.1 Parallel Symmetric Pivoting

To maintain their advantage of requiring $\frac{N^3}{3}$ floating point operations, symmetric solvers must maintain the symmetry of the matrix A being factored. Numerical stability considerations will however require pivoting to be performed; therefore the pivoting must be symmetric, ie. both rows and columns i and j must be exchanged. In the case where A is stored in its lower triangular half, this exchange is illustrated in Figure 1. Algorithmically, this symmetric interchange consists of a row swap, a transposition and a column swap, ie:

$$A_{j,0:\tilde{j}} \leftrightarrow A_{i,0:\tilde{j}} \ ; \ A_{j:\tilde{i},i} \leftrightarrow A_{i,j':i} \ ; \ A_{i':\tilde{N},j} \leftrightarrow A_{i':\tilde{N},i}$$

Note that the diagonal elements are exchanged, and $A_{i,j}$ remains unchanged.

The dashed trapezoid in Figure 1 represents the panel of the matrix currently being factored.

From Figure 1, the following observations may be applied to the symmetric interchange on a distributed memory platform:

1. the amount of data exchanged is the same as partial pivoting in LU. This implies, relative to the number floating point operations, the communication volume cost per exchange is double that of LU.

2. The interchange requires three separate operations. Furthermore, as all processors may have to potentially contribute to the new value of $A_{j:\tilde{N},j}$, which is within the current panel, the factorization cannot further proceed unless the value of i is broadcast to all processors. This is unlike LU factorization in the case where the panel is contained in a column of processors, in which only processor in that column can contribute to the updated panel.

   Furthermore, these interchanges has to be applied twice to the right hand side vector in the solve stage for LDLT (cf. Figure 3), as opposed to only once for LU.

   These imply (potentially) greater absolute startup costs than for LU.

3. The transposition $A_{j:\tilde{i},i} \leftrightarrow A_{i,j':i}$ further exacerbates these costs, but its cost can be minimized if a square processor grid is used.

[1] Despite this, very little development of Aasen's method has occurred over the last twenty years, unlike the diagonal pivoting methods.

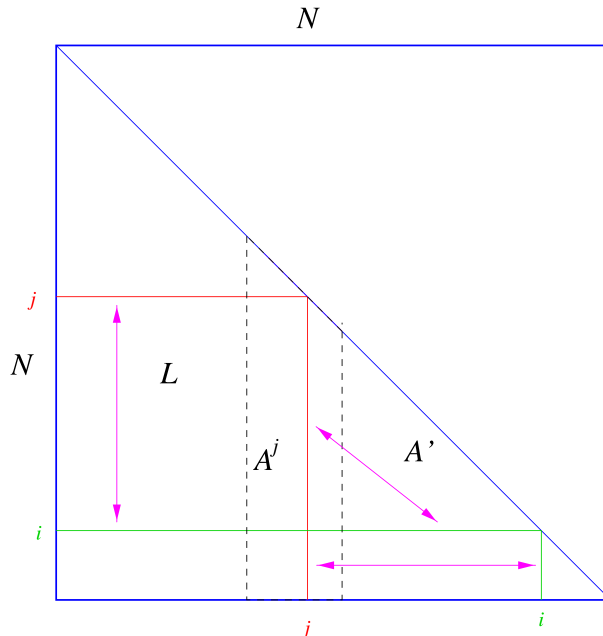

Figure 1: Symmetric Pivoting during an LDL^T factorization, showing current panel to be factored A^j

In other words, symmetric pivoting is potentially an expensive operation in a distributed memory implementation. For the Bunch-Kaufman method, case D1 can be applied for most columns of a 'weakly indefinite' matrix; even for random matrices, experiments have shown it is applied approximately 50% of the time (this re-

sults from $\alpha \approx 0.64$ being significantly less than 1).

On the other hand, tridiagonal methods are likely to require a symmetric interchange on most columns, even for 'weakly indefinite' matrices, as the values of diagonal elements play no role in determining the pivots.

Furthermore, the tridiagonal solve $T^{-1}X$ must be serialized [3], whereas the diagonal solve $D^{-1}X$ can be fully parallelized. This gives a further disadvantage to tridiagonal-based methods, such as Aasen's method.

The choice between the bounded and original Bunch-Kaufman algorithm deserves some treatment. While the bounded algorithm offers better stability, empirical and analytical studies show that it requires on average at least 2.5 column searches every time the test for case D1 fails [3]. Furthermore, empirical studies on random matrices have shown that the average number of symmetric interchanges of the bounded algorithm is $\approx$ 1.7 times greater [3]. Unless such stability is required for a particular application, this extra overhead favors the original algorithm.

Thus, for distributed memory implementation, methods that offer reduced symmetric interchanges for the matrices of interest should have a distinct performance advantage.

2.2 The LAPACK Diagonal Pivoting Algorithm

In this section, we describe a diagonal pivoting algorithm, essentially a simplification of that in LAPACK `_sytrf()`, which is publically available from NetLib [8]. A concise matrix-notation description of this algorithm is not found elsewhere; also it provides a starting point for subsequent discussions. For the sake of simplicity, it is assumed that the input matrix is invertible. The machine-dependent parameter ω is the target blocking factor (or panel width).

The level-2 portion (the first k-loop) is left-looking; the level-3 portion (step -(7)) is right-looking. The requirements of symmetric pivoting to a large degree dictates such a structure: if the level-2 portion was right-looking, a left-looking update would need to occur in any case to bring the portion of row i outside the panel up to date (see Figure 1). Similarly, if the level-3 portion was left-looking, then A^j would have to be updated by L (and W^T) before it was factored. However, at this stage, it is unknown which rows will be subsequently brought into A^j, and hence which rows of the W matrices (which now must be retained from previous stages) should be used.

The LAPACK algorithm has an elegant economy. Step -(3) is the first half of a symmetric interchange, which is completed if need be in either steps -(D3.1) and -(D3.2), or in -(D4.1) and -(D4.2).

At this point, we can begin to analyse the relative costs of each of the cases D1–D4. Case D1 is the most efficient; case D2, while having the same outcome, requires half a symmetric interchange and wastes the matrix-vector multiply of step -(4). Case D3 is less efficient still, as it similarly wastes the computation of step -(1), and requires the symmetric interchange to be completed. Case D4 is between cases D1 and D2 in terms of efficiency, for although it requires a symmetric interchange, it eliminates 2 columns. Thus, unlike cases D2 and D3, it involves no redundant matrix-vector multiplies.

The second k-loop of Figure 2 undoes the row swaps applied to A^j, instead of applying these row swaps to $A_{j:\tilde{N},0:\tilde{j}}$ (now over-written by L). In the LAPACK source codes, the former is referred to as "[LAPACK] standard form". Provided the routines which subsequently use L (eg. the $(LDL^T)^{-1}$ solve routine `_sytrs()`) take this into account, this potentially can speed up the factorization stage.

However, the decision to use the LAPACK standard form must be also considered from the point of view of the efficiency of the solve algorithm, which is given below in Figure 3.

Note that the pivoting of X having to occur during the loops performing the updates by L^{-1} and L^{-T} (rather than separately) is a direct consequence of using the LAPACK standard form. This makes blocking of this algorithm difficult.

2.3 Comparison with LU and LLT Algorithms

A LU algorithm can be used to factor an $N \times N$ symmetric matrix, but this requires $\frac{2}{3}N^3$ FLOPS, as opposed to the LLT and LDLT algo-

$j \leftarrow 0$; $P_{0:\tilde{n}} \leftarrow (0:\tilde{n})$ $\qquad \alpha = \frac{1+\sqrt{17}}{8}$

while $(j < N)$

  $k \leftarrow 0$ $\qquad A^j = A_{j:\tilde{N},\, j:j+\omega}$, $n = N - j$

  while $(k < \omega)$

    $\Delta k \leftarrow 1$

    $W_{k:\tilde{n},\,k} \leftarrow -A^j_{k:\tilde{n},\,0:\tilde{k}} W^T_{k,0:\tilde{k}} + A^j_{k:\tilde{n},\,k}$ -(1)

    find $i \in k' : \tilde{n}$ s.t. $|W_{i,k}| \geq |W_{k':\tilde{n},k}|$ -(2)

    if $(|W_{k,k}| < \alpha |W_{i,k}|)$ /* cases D2-4 */

      $W_{k:\tilde{n},k'} \leftarrow ((A_{j+i,\,j+k:j+\tilde{i}})^T \;,\; A_{j+i:\tilde{n},\,j+i})$ -(3)

      $W_{k:\tilde{n},k'}$ -= $A^j_{k:\tilde{n},0:\tilde{k}} W^T_{i,0:\tilde{k}}$ -(4)

      find $l \in k' : \tilde{n}$ s.t. $l \neq i, |W_{l,k'}| \geq |W_{k':\tilde{n},k'}|$ -(5)

      if $(|W_{k,k}| \geq \alpha |W_{i,k}|^2 / |W_{l,k'}|)$ /* case D2 - do nothing */ ;

      else if $(|W_{i,k'}| \geq \alpha |W_{l,k'}|)$ /* case D3: interchange k and i */

        $W_{k:\tilde{n},\,k} \leftarrow W_{k:\tilde{n},\,k'}$ -(D3.0)

        $A_{j+i,\,j+k:j+\tilde{i}} \leftarrow (A^j_{k,k} \;,\; (A^j_{k':\tilde{i},k})^T)$ -(D3.1)

        $A_{j+i:\tilde{N},\,j+i} \leftarrow A^j_{i:\tilde{n},k}$ -(D3.2)

        $A^j_{k,0:k} \leftrightarrow A^j_{i,0:k}$; $W_{k,0:k} \leftrightarrow W_{i,0:k}$; $P_{j+k} \leftarrow j + i$ -(D3.3)

      else /* case D4: 2 x 2 pivot, interchange k' and i */

        $A_{j+i,\,j+k:j+\tilde{i}} \leftarrow (A^j_{k',k} \;,\; (A^j_{k':\tilde{i},k'})^T)$ -(D4.1)

        $A_{j+i:\tilde{N},\,j+i} \leftarrow A^j_{i:\tilde{n},k'}$ -(D4.2)

        $A^j_{k',0:k'} \leftrightarrow A^j_{i,0:k'}$; $W_{k',0:k'} \leftrightarrow W_{i,0:k'}$; $P_{j+k} \leftarrow P_{j+k'} \leftarrow -j - i$; $\Delta k \leftarrow 2$ -(D4.3)

    if $(\Delta k = 1)$ /* cases D1-3 */

      $A^j_{k:n,k} \leftarrow (W_{k,k} \;,\; W_{k':n,k} / W_{k,k})$ -(6)

    else

      $A^j_{k:n,k:k'} \leftarrow (D_k,\;\; W_{k+2:n,\,k:k'} D_k^{-1})$ $\quad D_k = \begin{pmatrix} W_{k,k} & W_{k',k} \\ W_{k',k} & W_{k',k'} \end{pmatrix}$ -(D4.4)

    k += Δk

  $A^j_{k:\tilde{n},k:\tilde{n}}$ -= $A^j_{k:\tilde{n},0:\tilde{k}} (W_{k:\tilde{n},0:\tilde{k}})^T$ /* update lower half only */ -(7)

  while $(k > 0)$ /* undo row swaps in A^j */

    $\Delta k \leftarrow 1$; $i \leftarrow P_{j+k}$

    if $(i < 0)$ $\Delta k \leftarrow 2$; $i \leftarrow -i$

    $A^j_{k,0:\tilde{k}+\Delta k} \leftrightarrow A^j_{i,0:\tilde{k}+\Delta k}$; k -= Δk -(8)

Figure 2: LAPACK algorithm for factorizing an $N \times N$ symmetric indefinite matrix A

```
j ← 0
while (j < N)                                   /* apply D^{-1}L^{-1} */
   if (P_j > 0)
      X_j ↔ X_{P_j}                                                   -(1)
      X_{j':Ñ} -= A_{j':Ñ,j} X_j                /* rank-1 update */   -(2)
      X_j ← X_j/A_{j,j} ; j += 1                                      -(3)
   else
      X_{j'} ↔ X_{-P_j}                                               -(4)
      X_{j+2:Ñ} -= A_{j+2:Ñ,j:j'} X_{j:j'}      /* rank-2 update */   -(5)
      X_{j:j'} ← X_{j:j'} D_j^{-1} ; j+=2       D_j = ( A_{j,j}   A_{j',j}  )   -(6)
                                                      ( A_{j',j}  A_{j',j'} )
j ← Ñ
while (j ≥ 0)                                   /* apply L^{-T} */
   if (P_j > 0)
      X_j -= A^T_{j':Ñ,j} X_{j':Ñ}              /* vector-matrix multiply */   -(7)
      X_j ↔ X_{P_j} ; j -= 1                                                    -(8)
   else
      X_{j̃:j} -= A^T_{j':Ñ,j̃:j} X_{j':Ñ}         /* (2-) vector-matrix multiply */   -(9)
      X_j ↔ X_{-P_j} ; j -= 2                                                   -(10)
```

Figure 3: LAPACK algorithm for solving an $N \times K$ linear system with right hand size matrix X using a matrix A overwritten by L and D, from the LDL^T factorization of an $N \times N$ symmetric indefinite matrix

rithms which exploit symmetry to require only $\frac{1}{3}N^3$ FLOPS. LLT, of course, can only be used when the matrix is known to be positive definite. For sufficiently large N, (computation) time is dominated by matrix-matrix multiply in all 3 cases; thus in principle, all 3 algorithms should achieve similar efficiencies.

In practice, LLT and especially LDLT algorithms will achieve a lower efficiency. The first reason is that these perform the matrix-matrix multiply over a triangular matrix, which even for moderate values of N, is typically slower than the rectangular matrix-matrix multiply which LU uses. This is only slightly offset by the fact that LLT can use level-3 computations in its panel formation, whereas pivoting requirements of LU ensure the lower panel formation is a level 2 computation.

For execution on a message-passing parallel computers, both LU and LLT have to broadcast horizontally and vertically their respective panels. As this accounts for the bulk of communication volume in either case, LLT has a slightly higher ratio of communication volume costs to floating point costs. This gives it a disadvantage for moderate-large N, but if storage blocking is used [16], LLT has a lower communication startup costs ($O(\frac{N}{r})$ vs $O(N)$), giving it an advantage over small N. For their horizontal broadcasts across a $P \times Q$ grid, both LU and LLT can use pipelining, reducing communication costs by a factor of $\frac{\lg Q}{2}$ [7, 21].

Comparing now LLT with (Bunch-Kaufman) LDLT, LDLT has the following inherent performance disadvantages:

1. with the possibility of a symmetric interchange occurring in the lower panel A^j:
   (a) the formation of A^j must be done by level 2 computations.
   (b) pipelined communication cannot be used for horizontal broadcasts (steps -(2), -(5) and -(7)).
   (c) column searches for the maximum elements (steps -(2) and -(5) of Figure 2) are now required, the result of which must be immediately broadcast to *all* processors, introduce $(2\lg P + \lg Q)$ communication startups.
   (d) if a significant portion of the columns require cases D2–4, there will be

further significant computational and communication (both startup and volume) overheads.

The largest source of these is the cost of a (partial of full) symmetric interchange, as explained in Section 2.1.

(e) advanced load balancing techniques such as *lookahead* [19] (loss of pipelining) and *panel scattering* [20] (the interchange within A^j requires bringing in elements outside A^j) cannot be used to speed up the formation of A^j.

2. the vertical broadcast of the panel L^T in LLT can be performed more efficiently utilizing the fact that L has already been horizontally broadcast; in LDLT, this cannot be (easily) done in the vertical broadcast of W^T, as $W \neq A^j$.

   This results in extra communication volume costs.

For all of these reasons, a parallel implementation of LDLT may have lower efficiency than that of LLT, which is in turn lower than that of LU; in particular, there will be a value N_{LU} for which the general LU factorization will be faster if $N < N_{\mathrm{LU}}$, despite requiring twice as many floating point operations.

3 Parallelizing the Diagonal Pivoting Method

The LAPACK LDLT algorithm, based on BLAS operations with a high fraction of level-3 computations due to the blocking factor $\omega > 1$, has been shown to be efficient on memory hierarchy uniprocessors [2, 11, 12, 3]. Thus, in principle, as other processor's memory can be regarded as an extra level of the memory hierarchy in the distributed memory context, the algorithms depicted by Figures 2 and 3 should have a straightforward parallelization that is also reasonably efficient. However, several modifications and optimizations can still be performed, as will be described in this section.

We will consider the $r \times s$ block-cyclic matrix distribution over a $P \times Q$ logical processor grid [7], where, for an $N \times N$ global matrix A, block (i, j) of A will be on processor $(i \bmod P, j \bmod Q)$. For this distribution, two established techniques can be used to parallelize this algorithm: *storage blocking*, where $\omega = r = s$, and *algorithmic blocking*, where $\omega > r = s \approx 1$. The latter has been shown to yield better performance across a variety of platforms [16, 15, 19].

As algorithmic blocking provides potentially better load balance, it can use a larger value of ω than storage blocking, yielding a possible computational advantage. For LDLT, however, the redundant computations for cases D2 and D3, whose average cost is proportional to ω, may offset this advantage. Also, by inspecting Figure 2, it incurs potentially expensive horizontal communications in steps -(2), -(5) and -(D4.4); these can be avoided if A^j is within a single processor column. As the level-2 factorizations is left-looking, the communication volume costs of steps -(2) cannot be 'recycled', as they can be for algorithmically blocked LU, LLT and QR with right-looking level-2 factorizations [19, 20].

On the other hand, even if $\omega = r$, a 2×2 pivot (case D4) occurring on the $\omega - 1$th column of A^j will require the horizontal communications (and the associated complexity of their implementation) of step -(D4.4) anyway.

Thus, a sensible strategy would be to implement an algorithm encompassing both techniques, and evaluate their relative effectiveness on the target architecture.

For parallel implementation, the optimizations of the following subsections can be performed on the symmetric solver.

3.1 Array Element Access

In the distributed memory context it is important to reduce the communication startup costs associated in the manipulation or use of single array elements.

The DBLAS vector maximum finding function returns both the index and the value of the maximum element (eg. i and $W_{i,k}$ in step -(2)) of Figure 2.

This function involves a parallel reduction and broadcast operation (in this case, requiring $2 \lg P + \lg Q$ startups). To determine whether case D1 applies, all processors similarly require the value of $W_{k,k}$. By writing a modified max-

imum finding function which chooses the cell holding $W_{k,k}$ as the root of the column-wise reduction and broadcast, $\lg P + \lg Q$ startups per column are saved.

This was applied similarly to step -(5). However, here, the condition $l \neq i$ suggests tow separate column searches, as is done in the LAPACK code. For the parallel implementation, it is more efficient to temporarily zero $W_{i,k}$ so that a single search can be used, saving $2 \lg P + \lg Q$ startups (for each time cases D2-4 apply).

Merging the row swaps in A and W (steps -(D2.3) or -(D3.3)) saves 2 startups each time these cases apply.

After step -(3), temporarily set $A^j_{k,k'}$ to $W_{k,k'}(= A^j_{k',k})$. As W and A^j are aligned, this requires no communication. Thus, the RHS of step -(D4.1) becomes simply $A^j_{k:\tilde{i},k'}$, saving 1 startup for each instance of case D4.

Similarly, the RHS of step -(D3.1) can become simply $(A^j_{k:\tilde{i},k})^T$, saving a startup for each instance of case D3.

Record the elements of D ($W_{k,k}$ for cases D1-3, and the elements of D_k for case D4) in a column-replicated vector. This saves at least $\lg Q$ startups for each column in the solve stage (steps -(3) or -(6) in Figure 3). Note that for case D4, the values of all elements of D were broadcast to all processors in steps -(2) and -(5).

3.2 Improving Performance in the Solve Stage

While the solve stage has only $O(N^2)$ floating point operations, it has high associated overheads which makes its optimization particularly important in the distributed memory context. This can be achieved by improving load balance and reducing communication volume costs, as well as increasing computation speed. Of most interest is the case where X has a small number of right hand sides (for the AccuField computation, there is only one [13]).

The algorithm of Figure 3, updates X by each column of A individually. Thus, if implemented by a series of parallel BLAS calls, this would have 2 results: (1) all of A would be communicated in each of the loops, and (2) only the cells holding part of X would perform any computation (the independent parallel BLAS calls have no scope for performing any load balancing in such a situation).

However, by completing the row swaps in L during or after the factorization, in other words not using the LAPACK standard form, the solve stage can be implemented by the DBLAS triangular matrix solve routine to perform the second and fourth steps of:

$$X \leftarrow P'^{-1}X \ ; \ X \leftarrow L^{-1}X \ ;$$
$$X \leftarrow D^{-1}X \ ; \ X \leftarrow L^{-T}X \ ; \ X \leftarrow P'X$$

Here P' is the permutation matrix formed by the row swaps of the pivot vector $P_{0:\tilde{N}}$. With the scheme of L and D overwriting the original matrix A, the implicit zero sub-diagonal elements of L, which occur where D has 2×2 pivots, must be made (temporarily) explicit.

Although this scheme results in increased communication volume costs in the factor stage, this is compensated for in its reduction in the solve stage, as the DBLAS triangular matrix solve routine communicates only X. Furthermore, the completion of the row swaps in L can be done in blocks; for column-major matrices, this permits optimization of memory access patterns. Furthermore, for algorithmic blocking, this allows an effective reduction of message cost by a factor of at least 2 [19].

The DBLAS triangular solve routine achieves a very high degree of load balance. Furthermore, it has two computational advantages: it implements blocking of the computations (in the case of multiple RHS, most of the work is done in level 3, rather than level 2, BLAS), and it is matrix-vector multiply based (faster on most cell architectures, including the UltraSPARC, than rank-1 updates, cf. the first loop of Figure 3).

As a parallel BLAS triangular solve routine is a standard component, this method can take advantage also of any other optimizations already present, including blocking of the communications of X by the block size r. Such an optimization would be difficult to perform on the algorithm of Figure 2, as the pivoting in both loops hinders the blocking of any communications.

3.3 Minimizing Symmetric Interchanges

As explained in Section 2.1, minimizing the amount of symmetric interchanges (while keep-

ing the algorithm stable) has potentially large gains in algorithm performance.

One method of achieving this is implementing a key idea in the algorithm in [11]. This algorithm was largely motivated by the requirements of band matrices, where the minimization of interchanges helps preserve the structure of these matrices [11]. Let k be the current column of A^j to be eliminated, and let column $k-p$, $0 < p \leq k$ be the last column not eliminated by case D1. Let λ_i be the value of the maximum element determined at step -(2) of Figure 2. Then the condition for determining case D1 can be relaxed to:

$$\mu_k = \Pi_{i=k-p+1}^{k}(1 + \frac{W_{i,i}}{\lambda_i}) \leq (1 + \frac{1}{\alpha})^p \qquad (1)$$

This is stable since the the overall growth of A' from the block of p 1×1 pivots still remains within its bounds [11]. Intuitively, this can be thought of as the existence of large diagonal elements in preceding columns reducing the growth bounds on A' sufficiently to compensate for a smaller current diagonal element.

The implementation of this idea is somewhat different however. The algorithm in [11] is based on a different (3-case) variant of the diagonal pivoting method [6], and furthermore uses an *a priori* growth bound instead (which is necessarily more conservative, see Section 5.2). Here, the value of p becomes reset whenever the target blocking factor ω is reached, *or* a symmetric interchange is required. This has two undesirable consequences. Firstly, the behavior of algorithm can vary slightly, depending on the value of ω (this makes it hard, for example, to evaluate the performance of the algorithm as a function of ω). Secondly, and more importantly, the blocking factor p often falls short of the target blocking factor ω [11], resulting in a reduction in computational performance. Furthermore, in a distributed memory implementation, it would compromise the advantages of using storage blocking, as often the panel A^j would be straddling a storage block boundary.

Our implementation is to limit p to the range $0 \leq p < p_{\max}$, which is necessary to avoid overflow (and underflow) in μ_k. This can be efficiently achieved by storing the values of $1 + \frac{W_{i,i}}{\lambda_i}$ in a circular queue of size $p_{\max}$. Thus, μ_k can

N:	53		161		1601	
original:	1	(.04)	.01	(.06)	.01	(.07)
reduced:	2	(0)	.02	(.07)	.02	(.01)

Table 1: Comparison of residual (and f) for diagonal pivoting methods for AccuField Matrices

include contributions from columns in previous blocks. Furthermore, the optimal blocking factor is always met regardless of whether Equation 1 is.

Table 1 lists the normalized residual (using a random RHS vector with elements from the unit circle) for the original and reduced pivoting versions of the solver for sample AccuField matrices. The normalized residual is calculated the same way as in LAPACK test programs [8]; ideally, an accurate algorithm will produce residuals of less than unity, although in practice it may occasionally exceed unity (especially for small matrices) without implying a significant loss in accuracy. It also shows the fraction f of columns eliminated by cases D2–D4. With the typically large diagonal elements of these matrices, generally $f < 0.1$.

Figure 4 extends this study to simulated matrices of the form $A = A' + \beta I$, where A' has random elements from the unit circle and $0 \leq \beta \leq 10$. These represent the averaged values of the residual (and f) for 10 such matrices as functions of the diagonal bias β. In terms of the pivot distribution f, the range $5 \leq \beta \leq 7$ corresponds to Table 1. In terms of accuracy, the reduced interchange method has a residual generally within twice that of the original method, except for the range $3 \leq \beta \leq 5$, which coincides with the largest absolute reduction in f. However, as these residuals are all within their threshold, in most circumstances this is not a serious point for concern. The new scheme does significantly reduce f for $\beta > 1$; in particular $f \approx 0$ for $\beta > 5$.

In terms of stability, this scheme is no worse than the original Bunch-Kaufman algorithm in the sense that it attains the same growth bound in A'. If anything, it should offer some advantage in reducing the growth of L, the main shortcoming of the original algorithm, as it reduces the

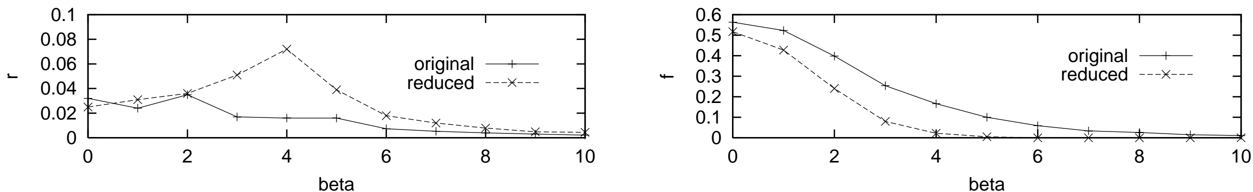

Figure 4: Solver averaged residual and pivot distribution for simulated 500×500 matrices

number of occurrences of case D4.

4 Implementation and Performance

With the DBLAS-based implementation of the LDLT solver, it is important that the DBLAS routines, as well as the lower-level libraries that it uses, in this case BLAS for computation and MPI or VPPLib libraries for communication, are implemented as efficiently as possible for the target architecture.

Within the DBLAS, efficient spread (or multicast) operations are especially important; the details for their AP3000 implementation can be found in [21].

Within the (complex precision) UltraSPARC BLAS, deep software pipelining including (top-level) *cache lookahead* is implemented in all routines. Optimization of cache and TLB usage in matrix-matrix multiply, local matrix transpose, and even matrix-vector multiply is very important for computational performance. How these can be achieved for double precision is explained in [21]; the same techniques have been applied similarly to complex precision.

An efficient block-cyclic triangular matrix multiply algorithm is an important component of the computation. The corresponding serial LAPACK algorithm (in the routine `_lasyf()`) partitions A' (see Figure 1) into strips of width $\Delta N = \omega$, and applies level 2 operations to update the triangular portion of the strip. The DBLAS routine uses a larger strip width $\Delta N = \frac{C}{2\omega}$, where the effective cache size C is the minimum of the second-level cache size and half the TLB size multiplied by the page size (256 KB for an UltraSPARC under Solaris). It has been argued that this is optimal for rectangular matrix-multiply for such a cell architecture [21]. Thus, at $\omega = 44$, $\Delta N \approx 90$ for double-complex, and the larger strip width thus enables better cache performance. The triangular portion of the strip is in turn broken down into strips of width $\sqrt{\Delta N}$; thus a very high fraction of level 3 operations is also achieved.

The DBLAS implementation of LDLT decomposition allows the grid size $P \times Q$, the storage block size r and the algorithmic blocking size ω to be run-time settable parameters. Thus, simply setting $\omega = r$ means that storage blocking will be used; the DBLAS routines then ensure all the communication savings from storage blocking then occur. Thus, given a matrix of size N and PQ processors, the optimum combination of these parameters can then be chosen. The name of the combined factor-solve routine is `DZSYSV()`.

4.1 Serial Performance

Table 2 compares the performance of the LDLT solvers on a 300 MHz UltraSPARC II (U300). The Sun Performance Library 1.2 `zsysv()` performed almost identically to the NetLib LAPACK `zsysv()` (but also using the Performance Library 1.2 BLAS) with the same blocking factor; as it is difficult to use the former with different BLAS, the results below are given for the latter. The default blocking factor was 64; performance was improved slightly by choosing a smaller blocking factor ($\omega = 44$).

For small matrices (eg. $N < 250$), `zsysv()` is slightly faster, primarily due to the software overheads in `DZSYSV()`, which is a parallel algorithm in this case run on a single processor (these overheads entail several extra layers of procedure calls, redundant conditional evaluations, and extra error checking [18]).

For larger matrices, `DZSYSV()` shows a clear improvement in speed, even when using the same (ANU) BLAS. This is primarily because the

	`zsysv()`		`DZSYSV()`, ANU BLAS	
N	Perf Lib BLAS	ANU BLAS	orig.	reduced
161	75	80	73	74
1601	71	96	108	109

Table 2: LDLT solver performance in complex MFLOPS for AccuField matrices on a U300 ($\omega = 44$)

DBLAS has a more efficient triangular matrix multiply routine than that used in LAPACK `_sytrf()`, as previously mentioned. The overall improvement on the Sun Performance Library 1.2 `zsysv()` thus amounts to 53%. $N = 1601$ is sufficiently large to represent the asymptotic speed of the solvers.

It should be noted that for these matrices, the reduced interchange scheme of Section 3.3 only increased performance by 1-2%.

Similar results were also found on a 170 MHz UltraSPARC I, and 200, 250 and 360 MHz UltraSPARC II's.

The solve stage component of these timings (see Section 3.2), while not having a large impact on the overall performance, similarly showed that `zsytrs()` was slightly faster at $N = 161$, but `DZSYTRS()` was 50% faster at $N = 1601$, indicating that the method outlined in Section 3.2 did indeed achieve some computational advantages.

4.2 Parallel Performance

Figure 6 gives parallel solver performance for simulated matrices with $\beta = 0$.$\beta = 0$ was chosen to minimize f, which should heighten the difference between the first 2 plots. As APruntime V2 is not yet installed on the 16 node U300-based AP3000 used for these result, MPI was used as the underlying communication library.

While the 'unoptimized algorithm' is proved to be reasonably efficient, the optimizations of the 'original algorithm' (Sections 3.1 and 3.2) achieves a worthwhile performance improvement of 10-20% in this range. As to be expected from Figure 4, the 'reduced interchange' algorithm (Section 3.3) yielded only a very small performance advantage for these matrices. Comparing the plots those of with Figure 6 shows the

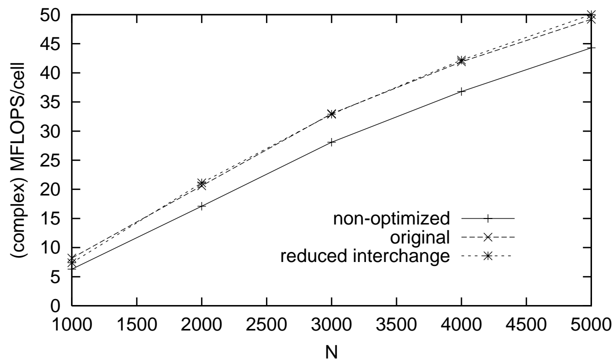

Figure 5: LDLT solve performance for simulated $N \times N$ matrices with $\beta = 0$ and $r = \omega = 44$ on a 4×4 300 MHz AP3000

reduced efficiency in the parallel diagonal pivoting algorithms when the diagonal elements are small.

Figure 6 gives parallel solver performance for simulated matrices with $\beta = 7$. It was found that the pivoting ratio f increases significantly with N. Eg. at $N = 10000$, $f = 0.20$ for the original method, and $f = 0.05$ with reduced pivoting. It is not yet known whether this also occurs for AccuField matrices of a similar order.

Comparing with Table 2, it can be seen that communication overheads prevent efficiencies that are possible in the serial case. Comparing the plots for $\omega = r = 44$, we can see that much of this overhead is from the interchanges, with the reduced method being faster by $\approx 10-15\%$ at the low-mid ranges, decreasing to $\approx 7\%$ at the upper range.

Comparing the plots for the reduced method, we see that storage blocking ($\omega = r = 44$) has a small but consistent advantage over algorithmic blocking ($\omega \approx 44$ with $r = 1, 4$). This is to be expected for small N, as algorithmic blocking incurs an extra $\lg QN$ startups at steps -(1), and also at steps -(4) and -(D4.4). For moderate-large N, storage blocking has an unusual advantage on the AP3000: its larger messages in step -(7) can take better advantage of the protocol communication method [14], effectively achieving a higher bandwidth. This inhibits load balance factors from compensating for this. A block size of $r = 4$ achieves a performance gain over $r = 1$ as some pipelining of the r

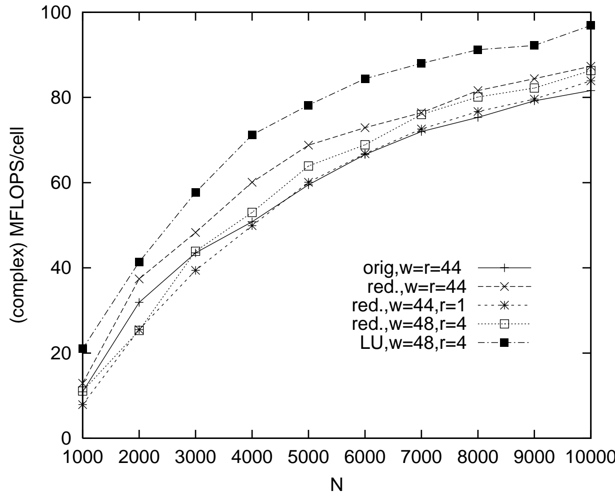

Figure 6: LDLT solve performance for simulated $N \times N$ matrices with $\beta = 7$ on a 4×4 300 MHz AP3000

(tree-based) broadcasts/reductions occurs when they are rooted in the same cell row/column in step -(1)[2]. Secondly, because the probability of extra communication in step -(D4.4) is proportional to $\frac{1}{r}$. This is in despite of $r = 4$ achieving a load balance factor in the formation of A^j of $E_d(\omega = 44, r = 4, Q = 4) = 0.77$, being somewhat lower than $E_d(\omega = 44, r = 1, Q = 4) = 0.93$ [16].

However, for LU, and to a lesser extent LDLT without reduced interchanges, algorithmic blocking at $\omega = 48, r = 4$ slightly outperformed storage blocking for $N > 5000$. This is due to algorithmic blocking introducing a relatively smaller amount of extra communications in these cases, and to the increased costs of multiple row swaps here (for which algorithmic blocking affords a degree of parallelization [20]).

While LU achieves much higher speeds for a given N, they are never greater than 2 (ie. $N_{\mathrm{LU}} < 1000$ here). In other words, the LDLT solver is quicker than the LU solver in this range, with the residuals for the LU solver being only marginally smaller.

[2]Note that the vertical reduction and horizontal broadcast of step -(2) should reduce some of this effect.

5 Future Work

In this section, two different approaches to further performance enhancements of the LDLT solver are described. The first is very specific to the AP3000 (under APruntime V2); the second is specific to 'weakly indefinite' matrices but could potentially apply to any platform, parallel or serial.

5.1 Improving Communication Pattern Performance

On the AP3000, a message transfer to and from user memory must occur in three stages, as the message must be copied to and from 'special memory', that is, (message buffering) memory that can be accessed by each node's Message Controller (MSC). The MSC is connected to the node by an SBUS. This memory includes 12 MB of SDRAM memory directly attached to the MSC, and KMEM, a 12 MB part of the node's memory, allocated for use by the MSC [14]. Such a convention helps ensure safety and security in a multi-user environment, as a message transfer to and from user memory requires the co-operation of both the source and destination nodes, and is similarly useful for fast user-level communications in the cluster computing model.

However, this mode of message transfer can potentially degrade communication performance seriously, with for example 8KB messages (a typical size for a vector) being transferred at a rate of only 30 MB/s, whereas the AP3000's hardware inter-node bandwidth is 200 MB/s [10].

A method to improve performance for larger messages is the *protocol method* [14], which involves breaking the messages into large (eg. 32 KB) chunks and pipelining the transfers of chunks over the three stages. This is possible because the SBUS can perform read and write transfers simultaneously without loss of bandwidth. In this way, 1 MB or larger messages can be transferred at the rate of 85 MB/s. This method is already incorporated into VPPLib and MPI message send and receive calls on the AP3000.

This idea can be extended to *communication patterns*, where the pipelining can be performed over several messages, or the copies to

special memory can be amortized over the messages. For the LDLT factorization algorithm of Figure 2, this can occur the tree broadcast (steps -(1) and -(4), for small messages, and step -(7), for large messages), tree reductions (steps -(1) and -(4), for moderate-sized messages), and the spread [21] or multicast (steps -(7), for moderate-large messages if $\omega > r$) for the LDLT factorization algorithm.

As previously stated, the main cost in parallel LDLT factorization for moderate-large matrices on the AP3000 is in communication bandwidth; this method's main potential in enhancing performance is then for moderate-sized messages, that is messages too small for the (full) benefit of the protocol method. For smaller messages, performance might be improved by looking at alternate memory copy routines than Solaris `memcpy()` (which we have found to be very efficient on moderate-large data sizes to and from special memory).

5.2 Towards a True 'Cholesky Speed' Solver

As discussed in Section 2.3, an LLT solver has several inherent performance advantages over an LDLT solver. However, for matrices close to (positive) definite, ie. 'weakly indefinite' matrices, it should be possible to design an LDLT solver whose performance approaches that of an LLT solver. In the following discussion, we will assume that the matrices are such that a high proportion (say 95% or more) of their columns can be eliminated using case D1.

The key idea is to 'lookahead' over the next block of ω columns in A^j and search for their (current) maximum elements λ^{p}_k, and then predict the *a priori* element growth for the 1st k diagonals of the block being used as the pivots [11].

$$\mu^{\mathrm{p}}_k = \mu^{\mathrm{p}}_{k-1}(1 + \frac{\mu^{\mathrm{p}}_{k-1}\lambda^{\mathrm{p}}_k}{|A^j_{k,k}|}) \qquad (2)$$

The algorithm of [11] terminates the search whenever this bound is exceeded, but suggests that a search over all columns would be efficient provided this occurs infrequently. Following a further suggestion in [11], even if the growth bounds are exceeded at column k, ie. $\mu^{\mathrm{p}}_k > (1+\frac{1}{\alpha})^{k+1}$ for some $k < \omega - 1$, provided the bounds are satisfied at some column $\omega^{\mathrm{p}} - 1$, $k < \omega^{\mathrm{p}} - 1 < \omega$, the diagonals for the first ω^{p} columns can be stably used as pivots. This could make up for the fact that the a priori element growth is necessarily more conservative than the actual, ie. $\mu^{\mathrm{p}}_k \geq \mu_k$, so that the method described in Section 3.3 may still achieve a similar fraction of cases D1 with this scheme.

In the distributed memory setting, the simultaneous search over ω columns could reduce the communication startup costs of the searches by a factor of ω, in other words, reducing startup costs to $O(\frac{N}{\omega})$, comparable to LLT.

Furthermore, the first ω^{p} columns, $A^j_{:,0:\tilde{\omega}^{\mathrm{p}}}$ can be eliminated by a level-2 factorization of the $\omega^{\mathrm{p}} \times \omega^{\mathrm{p}}$ triangular matrix T^j, and then applying the level-3 update of $(T^j)^{-1}$ to the remainder of the columns. As $W_{:,0:\omega^{\mathrm{p}}} = A^j_{:,0:\tilde{\omega}^{\mathrm{p}}} D^j$, where D^j is the diagonal matrix made from the diagonals of $A^j_{:,0:\tilde{\omega}^{\mathrm{p}}}$, the vertical broadcast of W^T can be produced in the same way as L^T for LLT, avoiding explicit transposition (at the cost of then having to scale it by D^j).

Thus, the disadvantages 1(a), 1(c), 1(d) and 2 of Section 2.3 may be overcome provided on average $\omega^{\mathrm{p}} \approx \omega$. An adaptive algorithm could even be devised, where pipelined communication could be introduced if, based in the previous history of the factorization, an occurrence of cases D2–4 is predicted to occur in fewer than one in rQ columns. This would overcome disadvantage 1(d). The only significant performance disadvantages over an LLT algorithm would be the memory accesses in extra level-1 operations (in the column searches, and in the scaling of W^T).

The plot for LU in Figure 6 represents an upper bound on the efficiency this solver could achieve.

6 Conclusions

Symmetric indefinite solvers are an interesting computation where there is a tradeoff in accuracy and performance. We have presented and developed variants of the diagonal pivoting method which yield improved performance, especially for 'weakly indefinite' systems where it is easier to obtain high accuracy. Indeed, to

our knowledge, it is the first time a clear performance gain over the LAPACK `_sysv()` routine for large matrices has been demonstrated, in this case over the UltraSPARC family of processors. This is partly due to introducing a reduced interchange scheme that ensures a fixed blocking factor (and hence does not sacrifice computational speed), and partly due to developing optimized algorithms for the low-level operations, such as the triangular matrix multiply.

However, on distributed memory platforms the reduction of symmetric interchanges becomes an increasingly important concern which has guided our choice of algorithm. On state of the art distributed memory platforms such as the Fujitsu AP3000, the reduction in communication costs it affords a 10% increase even for large matrices. For smaller matrices, improvements of a similar order were afforded by minimizing the potentially high communication startup costs inherent in diagonal pivoting methods.

The choice of blocking methods made less difference, with storage blocking with $\omega = r = 44$ generally being marginally faster than algorithmic blocking, with $r = 4, \omega = 48$. That algorithmic blocking did not gain an advantage was primarily due to peculiarities in both the communication performance of the AP3000 and in the structure of the LDLT algorithm.

For moderate-large matrices, communication volume costs are the main obstacles to achieving high efficiencies on the AP3000. There remains only two recourses to reduce these. The first is to maximize the bandwidth of communication patterns at the low-level platform-specific communication library level. The second is to use an algorithm which performing lookahead over pivot blocks, and apply LLT-like operations on blocks where no interchanges are required. This may offer the ultimate performance for matrices where interchanges are rare, as this enables pipelined broadcasts and more efficient transposition. It also offers large reductions in communication startup costs for smaller matrices, and computational advantages also benefiting serial performance.

Acknowledgements

The author would like to sincerely thank Vacation Scholar Mr Viet Nguyen of ANU for implementing a high performance version of complex precision UltraSPARC BLAS, which yielded large performance gains for the solver. The author would also like to thank Dr Noro from Fujitsu Laboratories providing information on the AccuField application and providing helpful feedback on the issues of solver interface and performance. The author would also like to thank Mark Jones, Linda Kaufman and John Lewis for their insights into why tridiagonal-based methods have received so little development over recent years.

Finally, the author would also like to thank the Fujitsu Parallel Research Computing Facilities for the use of a 16 node AP3000.

References

[1] Jan Ole Aasen. On the Reduction of a Matrix to Tridiagonal Form. *BIT*, 11:233–241, 1971.

[2] C. Anderson and J. Dongarra. Evaluating Block Algorithm Variants in LAPACK. In *Fourth SIAM Conference for Parallel Processing for Scientific Computing*, Chicago, December 1989. 6 pages.

[3] Cleve Ashcraft, Roger G. Grimes, and John G. Lewis. Accurate Symmetric Indefinite Linear Equation Solvers. to appear in *Simax*, 1999. 49 pages.

[4] Victor Barwell and Alan George. A Comparison of Algorithms for Solving Symmetric Indefinite Systems of Linear Equations. *ACM Transactions on Mathematical Software*, 2(3):242–251, September 1976.

[5] R. P. Brent and P. E. Strazdins. Implementation of BLAS Level 3 and LINPACK Benchmark on the AP1000. *Fujitsu Scientific and Technical Journal*, 29(1):61–70, 1993.

[6] James R. Bunch and Linda Kaufman. Some Stable Methods for Calculating Inertia and

Solving Symmetric Linear Systems. *Mathematics of Computation*, 31(137):163–79, January 1977.

[7] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D.W. Walker, and R.C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR and Cholesky Factorization Routines. *Scientific Programming*, 5:173–184, 1996.

[8] C. Anderson et al. *LAPACK User's Guide*. SIAM Press, Philadelphia, 1992.

[9] Gene Golub and Charles Van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore, second edition, 1989.

[10] H. Ishihata, M. Takahashi, and H. Sato. Hardware of the AP3000 Parallel Server. *Fujitsu Scientific and Technical Journal*, 33(1):24–29, 1997.

[11] Mark T. Jones and Merrell L. Patrick. Factoring Symmetric Indefinite Matrices on High-Performance Architectures. *SIAM Journal on Matrix Analysis and Applications*, 12(3):273–283, July 1991.

[12] Linda Kaufman. Computing the MDM^T decomposition. *ACM Transactions on Mathematical Software*, 21(4):476–489, December 1995.

[13] Dr. Noro. Private comunications, 1998–1999.

[14] David Sitsky and Paul Mackerras. A high-performance message passing Library for the Fujitsu AP3000. In *Proceedings of the Eighth Parallel Computing Workshop*, pages 245–251, Singapore, September7–8 1998. National University of Singapore. paper P1-E.

[15] K. Stanley. *Execution Time of Symmetric Eigensolvers*. PhD thesis, University of California, Berkelely, 1997. viii+184p.

[16] P.E. Strazdins. Matrix Factorization using Distributed Panels on the Fujitsu AP1000. In *IEEE First International Conference on Algorithms And Architectures for Parallel Processing (ICA3PP-95)*, pages 263–73, Brisbane, April 1995.

[17] P.E. Strazdins. A High Performance, Portable Distributed BLAS Implementation. In *Sixth Parallel Computing Workshop*, pages P2–K–1 – P2–K–10, Kawasaki, November 1996. Fujitsu Parallel Computing Research Center.

[18] P.E. Strazdins. Reducing Software Overheads in Parallel Linear Algebra Libraries. In *The 4th Annual Australasian Conference on Parallel And Real-Time Systems*, pages 73–84, Newcastle Australia, September 1997. Springer.

[19] P.E. Strazdins. Lookahead and Algorithmic Blocking Techniques Compared for Parallel Matrix Factorization. In *PDCN'98: 10th International Conference on Parallel and Distributed Computing and Systems*, pages 291–297, Las Vegas, September 1998. IASTED.

[20] P.E. Strazdins. Optimal Load Balancing Techniques for Block-Cyclic Decompositions for Matrix Factorization. In *PDCS'98: 2nd International Conference on Parallel and Distributed Computing and Networks*, pages 192–199, Brisbane, December 1998. IASTED.

[21] Peter E. Strazdins. Transporting Distributed BLAS to the Fujitsu AP3000 and VPP-300. In *Proceedings of the Eighth Parallel Computing Workshop*, pages 69–76, Singapore, September7–8 1998. School of Computing, National University of Singapore. paper P1-E.