Parallel Electromagnetic Field Analysis using ACCUFIELD

P.E. Strazdins**, K. Homma*, K. Nagase, V. Nguyen**, M. Noro*,
T. Yamagajo

Technology Development Group, Fujitsu Ltd.
{gajo,knagase}@cs.fujitsu.co.jp

* Secure Computing Laboratory, Computer System Laboratories, Fujitsu Laboratories Ltd.
{homma,noryo}@flab.fujitsu.co.jp

** Department of Computer Science, Australian National University
viet@cafe.anu.edu.au, peter@cs.anu.edu.au

December 11, 2000

Abstract

ACCUFIELD is a commercial application for electro-magnetic field analysis currently being developed by Fujitsu Ltd. It can be used to simulate the emissions for computer hardware sub-systems such as a printed circuit board, a combination of circuit boards and wire connecting boards, or even a cabinet.

The ACCUFIELD software consists of a pre-processor, a solver and a postprocessor. The pre-processor is used to input or modify a model of the sub-system to be analysed. It converts the model to a dense complex symmetric indefinite linear system, which can then be solved for various EM frequencies by the solver. These systems are often 'weakly indefinite', that is, most diagonal elements are considerably larger than the off-diagonal elements in the same column. The postprocessor displays the resultant electromagnetic fields. ACCUFIELD also has a highly developed user interface, enabling many convenient model editing functions and graphical displays of the resulting analysis.

The size of the linear system N depends on the model to be analysed. For a Note PC, $N \approx 5000$, but for more complex systems, $N \approx 30,000$ is required. The memory and computational requirements of such a large system requires parallel processing, for example on a 24 node Fujitsu AP3000 distributed memory multicomputer, with each node being a 300 MHz Ultra-SPARC II with 1.5 GB of memory.

This paper describes the design, implementation, performance and validation of the AC-CUFIELD application. The main computational challenges lie in the solver stage, where an $O(N^3)$ computation is required for the direct solution of the linear system about a central frequency ω_c. Some of this cost is amortized using a *frequency stepping method*, where the system can be solved for nearby frequencies ω by $O(N^2)$ iterative methods, using the solution at ω_c as a preconditioner. This reduced parallel solution time by a factor of 2 for moderate-sized matrices, with much larger improvements expected for large matrices.

The performance of the solver stage is of key importance. For ACCUFIELD, the first known parallelization of a dense direct symmetric indefinite solver was made, requiring many issues to be solved in order to minimize communication costs. The algorithms used are variants of the Bunch-Kaufman diagonal pivoting method, such as those used in the LAPACK routine zsysv(), which is one of the fastest publically available of such routines. However, our corresponding routine, called pzsysv(), out-performs zsysv() by $\approx 15\%$ for large matrices on the UltraSPARC family of processors. Further performance gains were achieved from developing a new variant of

the Bunch-Kaufman method that reduces symmetric interchanges while maintaining the same growth bound. This algorithm's efficiency is demonstrated by its parallel speedup of 12–13 for large matrices on a 16 node AP3000.

To give ACCUFIELD high performance on the UltraSPARC family of processors, a heavily optimized implementation of the complex precision UltraSPARC BLAS was developed. A combination of various techniques are described in the paper that improved the solver's performance by over 40% over the commercially available Sun Performance Library 1.2 BLAS.

1 Introduction

The design of electronic devices is severely limited by the country-specific Electromagnetic Compatibility (EMC) requirements. Hence, minimizing the undesired radiation and avoiding malfunctions caused by intruding electromagnetic radiation is required in the design of these devices. Recently, the MPU operating speed has become faster, resulting in higher emission frequencies. In addition, the electronic circuits become smaller and more highly integrated. Thus, the undesired electromagnetic wave radiation from these devices tends to increase. In such a situation, the estimation of EMC via simulation becomes more and more important.

ACCUFIELD is a 3D simulator. It was developed to simulate the electromagnetic wave radiation and the immunity of many kinds of electronic devices by using the moment method. ACCUFIELD has been used mainly to simulate relatively simple electronic devices, such as printed circuit boards.

ACCUFIELD calculates the electromagnetic field in the frequency domain, and so, a matrix calculation, being a dense complex symmetric linear system solve, is executed repeatedly. When a large scale complicated model of an electronic device is being analysed, the computing time becomes long, as a direct solution requires $O(N^3)$ floating point operations. So, the reduction of this execution time is desired to estimate the EMC phenomenae of such devices efficiently.

The Fujitsu AP3000 [12] is a distributed memory multicomputer, comprised of RISC UltraSPARC scalar processors [22] with a deep memory hierarchy (having a 16KB top-level data cache and a 1MB 2nd-level cache, both direct-mapped, and a TLB of $|TLB| = 64$ entries). 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. For the AP-Net, the AP3000's communication network, communication latencies (from software) has been measures at $\approx 20\mu s$, with a transfer rate of up to 80 MB/s sustained for large messages [17]. The AP3000 also has many properties of the cluster computing model; this extra flexibility contributes to its communication costs.

For solving complex linear systems of the order $N \approx 30,000$ that can be generated by ACCUFIELD, a machine such as a 24 node AP3000, with each node being a 300 MHz UltraSPARC II with 1.5 GB of memory, is required to meet the computational and storage requirements.

This paper is organized as follows. Section 2 explains the moment method used by ACCUFIELD. An overview of the ACCUFIELD application, including its user interface and validation, is given in Section 3. Section 4 outlines the *fast frequency stepping method* of the solve stage, which can reduce the number of direct system solutions for analysis with multiple frequencies, and describes its parallel implementation. Section 5 describes in detail the direct solution methods required by the solver stage, including algorithm choice, parallelization issues and new algorithmic variants suitable for use parallel ACCUFIELD. Section 6 describes how the Basic Linear Algebra Subroutines (BLAS) computational kernels used by the solver were heavily optimized for the UltraSPARC processor. Performance results are given in Section 7, with conclusions being given in Section 8.

2 The Moment Method

ACCUFIELD calculates the electromagnetic field of the wave radiated from the model electronic device by using the moment method [10]. Here, we describe this method briefly.

The material of the analysed model is the metal or dielectric with or without loss. The an-
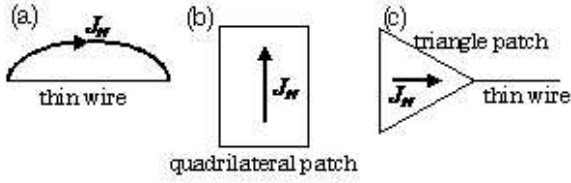
Figure 1: Expansion functions for (a) thin wire mode, (b) surface patch mode and (c) attachment mode

alyzed model is segmented to triangular or rectangular patch elements or thin wire elements [16]. The surface current on the metal J_c^s and dielectric J_d^s and the surface magnetic current M^s on the dielectric are expanded in terms of the basis expansion functions $J_{c,n}$, $J_{d,n}$ and K_n as follows:

$$J_c^s = \sum_{n=1}^{N_c} I_{c,n} J_{c,n}, J_d^s = \sum_{n=1}^{N_d} I_{d,n} J_{d,n}, M^s = \sum_{n=1}^{N_d} M_n K_n$$

(1)

Here, $I_{c,1:N_c}$, $I_{d,1:N_d}$ and $M_{1:N_d}$ are the $N = N_c + 2N_d$ unknown coefficients. The expansion function is specified in three types i.e., thin wire mode, patch mode and attachment mode applied to wire/surface junctions. These expansion functions are shown in Figure 1.

From the surface equivalence theorem and the reaction matching moment method, the following equation is obtained:

$$\begin{bmatrix} Z_{c,c}^0 & Z_{c,d}^0 & B_{c,d}^0 \\ Z_{d,c}^0 & Z_{d,d}^0 + Z_{d,d}^d & B_{d,d}^0 + B_{d,d}^d \\ B_{d,c}^0 & B_{d,d}^0 + B_{d,d}^d & Y_{d,d}^d - Y_{d,d}^0 \end{bmatrix} \begin{bmatrix} I_c \\ I_d \\ M \end{bmatrix} = \begin{bmatrix} V_i \\ 0 \\ 0 \end{bmatrix}$$

(2)

The matrix of the left side of Equation 2 is an immittance matrix and each element of it represents the mutual impedance Z, the mutual admittance Y and the reaction B between two elements. Here, superscripts 0 and d indicate the field material and corresponds to the air and the dielectric, respectively. Subscripts d and c indicate the material of the segmented elements and correspond to dielectric and conductor, respectively. Explicit equations of the immittance are shown in [16, 15]. $I_{c,1:N_c}$, $I_{d,1:N_d}$ and $M_{1:N_d}$ are the unknown expansion coefficients which appear in Equation 1. V_i is the driver voltage. From Equation 2, the current distribution of the model surface and the electromagnetic field of the radiated wave are calculated.

For convenience, we will use $Z(\omega)I = V(\omega)$ as an abbreviated form of Equation 2, where ω is the frequency of the incident electric field $V(\omega)$.

3   Overview of ACCUFIELD

The analysis procedure of ACCUFIELD is shown in Figure 2. ACCUFIELD consists of a preprocessor for inputting or editing the model of the electronic device, a solver for the calculation of the electromagnetic field and a postprocessor for the output and interpretation of the calculation results.

Many libraries are supplied for entry and retrieval of commonly used parameters of circuit components, specific constants of the materials and so on [8]. By using these libraries, it becomes easier to create the model.

The preprocessors' editing functions include the changing of routing paths (both on printed circuit boards and of cables), the addition or deletion of components such as filters and damping resistors, and changing the structure of the shielding.

The solver calculates the electromagnetic field generated by the model with the moment method described before. The transmission line approximation method is also employed to take into account the effects of components or subsystems beyond the moment method [16, 8].

The postprocessor displays the simulation results of the solver part, by using a GUI. Graphically displayed results are helpful for easy understanding of physical implementation of the EMC phenomena. The following results are available.

- 2D radiation pattern.

  For a given frequency, the electromagnetic field strength can be plotted as a function of the angle from the centre of the electronic device being analysed.

- Contour map of the electromagnetic field.

  The electromagnetic field across any plane can be plotted, enabling an easy-to-understand display of the field behavior.
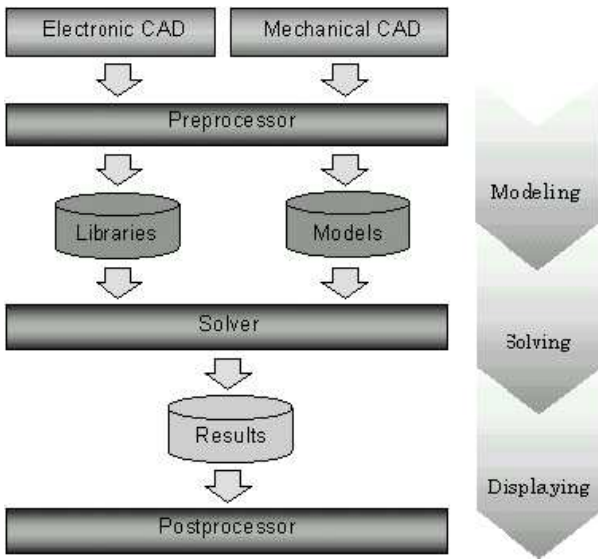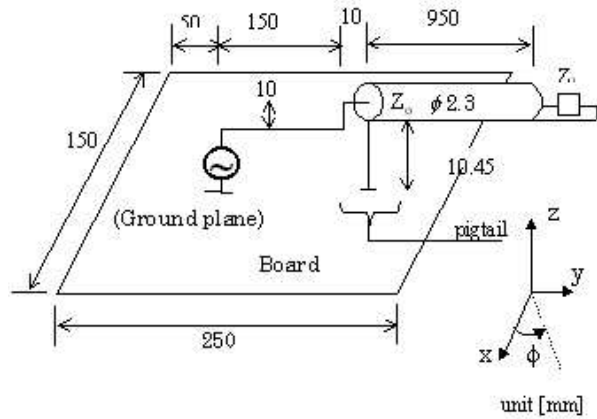
Figure 2: ACCUFIELD Analysis Procedure

Figure 3: The PCB with coaxial cable. Microstrip line: height 1.6, width 0.4, $\epsilon_r = 4.7$. Source: 1.1V sine wave, frequency 150 MHz, $Z_0 = Z_L = 50\Omega$

- Frequency spectrum of the radiated wave.

  These show the electric field strength for each frequency and compare this with the maximum strengths dictated by EMC regulations of various countries.

- Current distribution and vector plots.

  These can display the amount and direction of the current flowing across a conductor (eg. cable), aiding in the understanding of the emission mechanisms.

- Gradients of the electric field.

- Input impedance diagrams.

. . . and so on.

The analysis can demonstrate mechanisms such as the shielding effectiveness of enclosures, the influence of finite ground plates, common-mode and differential mode current emissions and the effectiveness of components such a filters in handling EM emissions.

Here is an example of an ACCUFIELD analysis [8]. The model is illustrated in Figure 3. It contains a printed circuit board (PCB) and a coaxial cable. By using the preprocessor of AC-CUFIELD, the model is created and displayed as shown in Figure 4.

The radiation pattern of the electric field calculated by ACCUFIELD is shown in Figure 5.
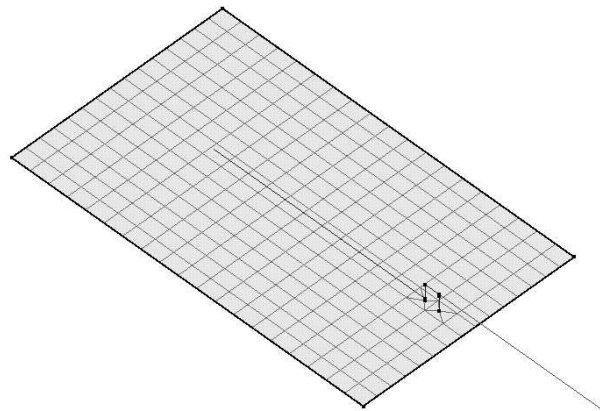

Figure 4: The PCB model displayed in the model display window. The model is created by using the preprocessor and the library. It consists of 291 patch elements and 5 wire elements

The solid curve indicates the electric field intensity of the wave radiated from the model shown in Figure 4. The measured electric field intensity is also shown with the closed circles. The numerical results agree well with the measured ones. This demonstrates the validity of the electromagnetic simulation with ACCUFIELD.

Consider the parallel implementation on PQ processors of an ACCUFIELD analysis over a range of frequencies $\Omega = \{\omega_1, \omega_2, \ldots, \omega_{n_f}\}$, where $\omega_i < \omega_{i+1}$ for $1 \le i < n_f$. The preprocessor broadcasts the (required parts of) model's information to all processors. MPI is used as the communication library, enabling wide portability of the application.

The immittance matrices $Z(\omega_{s_1}), \ldots, Z(\omega_{s_{n_s}})$ are then generated, where $\{\omega_{s_1}, \ldots, \omega_{s_{n_s}}\} \subseteq \Omega$. Here $\{\omega_{s_1}, \ldots, \omega_{s_{n_s}}\}$ (typically $s_1 = 1, s_{n_s} = n_f$, and n_s is small) is a set of sampling frequencies chosen for the analysis.

As the elements of these matrices can be generated independently, this process is done in parallel so that the matrices conform to a $r \times r$ block-cyclic matrix distribution over a logical $P \times Q$ processor grid [5], where r is the fixed storage block size.

The remaining matrices $Z(\omega_i)$ are generated by interpolation on the sampled immittance matrices $Z(\omega_{s_1}), \ldots, Z(\omega_{s_{n_s}})$. As they will have the same distribution, this step requires no communication.

At this point, the solver stage can begin on each $Z(\omega_i)$. In the following two sections, we describe in detail issues in the parallelization of this stage.

4 The Fast Frequency Stepping Method

ACCUFIELD often calculates field emissions over a range of EM wave frequencies. To avoid the cost of directly solving Equation 2 at each desired frequency in the range, the following can be performed:

- instead of obtaining a direct solution of $Z(\omega_i)I = V(\omega_i)$ for each ω_i, the *fast frequency stepping* method (FFS) chooses a central frequency ω_{i_c}, $1 \le i_c \le n_f$.

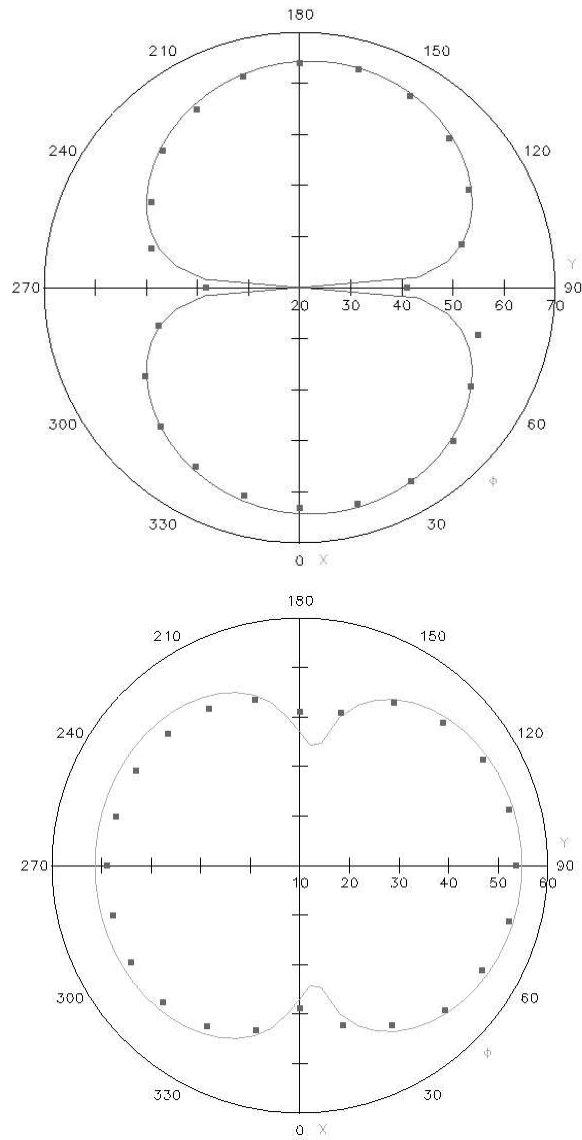 For ω_{i_c}, a direct solution is obtained.

Figure 5: Radiation pattern of the PCB's electric field, (a) horizontal components (b) vertical components. The distance between the model and the measured point is about 10m

$$r_0 = b - Ax_0$$

$$p_0 = A_0^{-1} r_0, \quad k = 0$$

while $\|r_k\| \geq \epsilon$

$$\alpha_k = (A_0^{-1} r_k, r_k)/(p_k, Ap_k)$$

$$x_{k+1} = x_k + \alpha_k p_k$$

$$r_{k+1} = r_k - \alpha_k A p_k$$

$$\beta_k = (A_0^{-1} r_{k+1}, r_{k+1})/(A_0^{-1} r_k, r_k)$$

$$p_{k+1} = A_0^{-1} r_{k+1} + \beta_k p_k$$

$$k = k + 1$$

Figure 6: Preconditioned Congugate Gradients for Complex Symmetric Equation $Ax = b$

- the factorization of $Z(\omega_{i_c})$ is then used to precondition iterative solutions for $\omega_{i_c-1}, \omega_{i_c-2}, \ldots$ and $\omega_{i_c+1}, \omega_{i_c+2}, \ldots$ until convergence rates slows down to the extent that the iterative solution time exceeds half that of a direct solution.

Here, the Preconditioned Conjugate Gradients method is used [9, 11]. For a positive definite matrix A, the preconditioning is $A \rightarrow (L_0^T)^{-1} A L_0^{-1} : A_0 = L_0^T L_0$ where L_0 is the Cholesky factorization of the preconditioner A_0. This algorithm converges in $rank(I - (L_0^T)^{-1} A L_0^{-1})$ iterations [9]; however, in practice this quantity is difficult to estimate.

We have adopted this method for complex symmetric indefinite matrices by using the LDLT factorization for the preconditioner ie. $A_0 = Z(\omega_{i_c}) = P_0 L_0 D_0 L_0^T P_0^T$. Figure 6 describes the algorithm.

Thus, the FFS method reduces the number of $O(N^3)$ direct solutions potentially required, replacing them by $O(N^2)$ iterative methods, resulting in potentially large gains in efficiency.

The iterative method for FFS is implemented using the DBLAS parallel BLAS (see Section 5) routines for symmetric matrix-vector multiply `pzsymv()`, vector inner product and norm routines, as well as the complex indefinite back-solve routine `pzsytrs()` that will be described in Section 5. The above routines have a compatible interface to ScaLAPACK's PBLAS [5].

Here, to economize on storage space, the factored preconditioner L_0 and D_0 are stored in the lower half of the matrix, and the current matrix $A = Z(\omega_i)$ is stored in the upper half (with one column shift).

When executed on a $P \times Q$ processor grid, each iteration involves a vector transpose-broadcast, one vector reduction and two scalar reduction/broadcasts, as well as the communications required in the call to `pzsytrs()`. For large matrices, eg. $\frac{N}{P} \geq 1000$, the AP3000 execution time is dominated by the level-2 computational performance (matrix-vector multiply); for smaller matrices, execution time is dominated by communication startup costs.

5 The Direct Solution of Symmetric Indefinite Systems

In electromagnetic scattering and compatibility applications, the linear systems generated by the moment method often have diagonal elements (representing the self-impedance or self-admittance of the corresponding 'moment') which are relatively large compared with the off-diagonal entries of the same column. Such a matrix we will call *weakly indefinite.*

This means that (faster) transformations appropriate to definite systems can be applied to eliminate most columns of these matrices, without sacrificing numerical stability.

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, 13, 14, 3], all but [13] consider only uniprocessor implementations, and [13] only considers parallelization on a small-scale shared memory machine.

In this section, we describe how to derive variants of the Bunch-Kaufman diagonal pivoting method to yield improved performance, especially on parallel platforms, and especially for weakly indefinite systems.

The parallel routines are coded entirely in terms of the DBLAS Distributed BLAS Library [19, 21], which is a portable version of paral-

lel BLAS. It has been used to implement very efficient parallel matrix factorization applications using various techniques [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.

5.1 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 of that era being the Bunch-Kaufman method [6]). Variants of the diagonal pivoting method have since been proposed [13, 14, 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.

Aasen's method involves exploiting properties of Hessenberg matrices to perform the decomposition $A = PLTL^TP^T$, where A is an $N \times N$ symmetric matrix, L is an $N-1 \times N-1$ lower triangular matrix with a unit diagonal, P is a permutation matrix, 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 diagonal pivoting methods perform the decomposition $A = PLDL^TP^T$, where here 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, 13].

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 $|A_{i,j}| = \max_{k=j+1}^{N-1}|A_{k,j}|$. 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 $j+1$ 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 sub-matrix (A' in Figure 7) 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 implementation of the Bunch-Kaufman algorithm to be superior by $\approx 5\%$ for large matrices.

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.

5.1.1 Parallel Symmetric Interchanges

To maintain their advantage of requiring $\frac{N^3}{3}$ floating point operations, symmetric factorization algorithms 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 7. Algorithmically, this symmetric interchange consists of a row swap, a transposition and a column swap, ie:

$$(a_{j,j},\ a_j^u) \leftrightarrow (a_{i,j},\ a_i''); \ \ a_j^l \leftrightarrow a_i^l\ ; \ \ (l_j',a_j') \leftrightarrow (l_i',a_i')$$

Here, $a_j' = A_{j,\,k:j}$, $a_i' = A_{i,\,k:j}$, $a_j^l = A_{i:N-1,\,j}$, $a_i^l = A_{i:N-1,\,i}$, and $a_j^u = A_{j+1:i-1,\,j}$, $a_i'' = A_{i,\,j+1:i-1}$. Thus, using such a combination of row and column segment exchanges, the diagonal elements are exchanged, and $A_{i,j}$ remains unchanged.
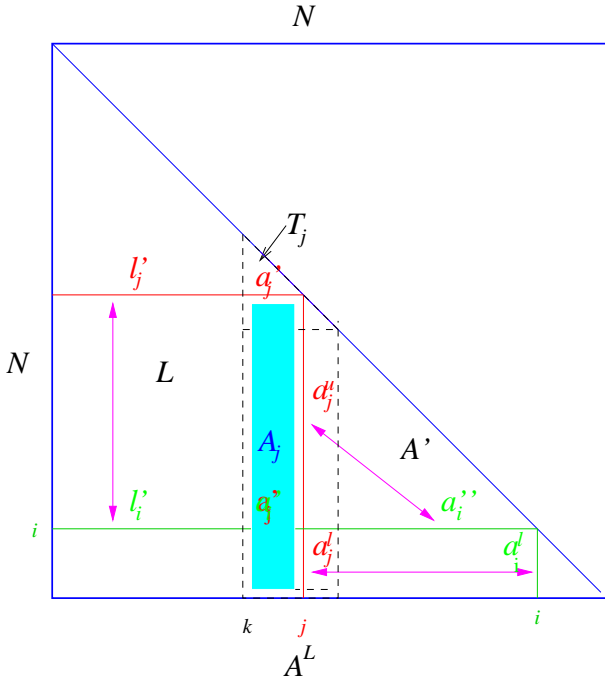

Figure 7: Symmetric interchange during an LDL^T factorization, showing the current panel to be factored A^L

The dashed trapezoid in Figure 7 represents the panel A^L of the matrix currently being factored.

From Figure 7, 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 $A_{j:N-1,j}$, 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 back-solve stage for LDLT, as opposed to only once for LU.

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

3. The transposition $A_{j:i-1,i} \leftrightarrow A_{i,j+1:i}$ further exacerbates these costs, but its cost can be minimized if a square processor grid is used.

In other words, symmetric interchange 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 40% of the time (this results 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 ≈ 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.

5.2 The LAPACK Diagonal Pivoting Algorithm

In this section, we describe an implementation of the Bunch-Kaufman algorithm, essentially a simplification of that used in LAPACK `_sysv()`, which is publically available from NetLib [7]. A formal matrix-notation description of this algorithm can be found in [18]; an informal description of the partial factorization (cf. Figure 7) is given in Figure 8.

Note that $a_j = A_{j:N-1,\,j} = (a_{j,j}, a_j^u, a_j^l)$ represents the entire jth column. Correspondingly, we define $a_{j+1} = A_{j:N-1,\,j+1} = (a_{j,j+1}, a_{j+1}^u, a_{j+1}^l)$ and $a'_{j+1} = A_{j+1,\,k:j}$.

A temporary matrix W (aligned with A) is used; the notations w_j etc. refer to the sections in W as do a_j etc. in A. Of course, in an actual implementation, explicit storage for only those elements of W in the current panel is required.

For the sake of simplicity, it is assumed that the input matrix is invertible.

Step -(14) undoes the row swaps applied to A^j, instead of applying these row swaps to $A_{k:N-1,0:k-1}$ (now over-written by L). In the LAPACK source codes, this is referred to as "[LAPACK] standard form". Provided the routines which subsequently use L (eg. the $X \leftarrow (LDL^T)^{-1}X$ back-solve routine `zsytrs()`) 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

for each column j in current panel

$\quad w_j \leftarrow (w_j')^T A_j + a_j;$ -(1)

$\quad$find the pos. i of the max. in w_j -(2)

$\quad$if case D1 does not apply:

$\qquad w_{j+1} \leftarrow (a_{i,j}, a_i'', a_i^l)$ -(3)

$\qquad w_{j+1} \leftarrow (w_{j+1}')^T A_j + w_{j+1}$ -(4)

$\qquad$find max. of w_{j+1} (excluding $w_{i,j+1}$)   -(5)

$\qquad$if case D3 applies /* complete interchange */

$\qquad\quad w_j \leftarrow w_{j+1}$ -(7)

$\qquad\quad (a_{i,j}, a_i'') \leftarrow (a_{j,j}, a_j^u); \quad a_i^l \leftarrow a_j^l$ -(8)

$\qquad\quad (w_j', a_j') \leftrightarrow (w_i', a_i')$ -(9)

$\qquad$if case D4 applies /* complete interchange */

$\qquad\quad (a_{i,j}, a_i'') \leftarrow (a_{j,j+1}, a_{j+1}^u); \quad a_i^l \leftarrow a_{j+1}^l$ -(10)

$\qquad\quad (w_{j+1}', w_{j+1,j+1}, a_{j,j+1}, a_{j+1,j+1}) \leftrightarrow$

$\qquad\qquad (w_i', w_{i,j+1}, a_i', a_{i,j+1})$ -(11)

$\quad$if case D4 applies /* apply 2×2 D_j^{-1} */

$\qquad D_j = \begin{pmatrix} A_{j,j} & A_{j+1,j} \\ A_{j+1,j} & A_{j+1,j+1} \end{pmatrix}$

$\qquad (a_j, a_{j+1}) \leftarrow (w_j, w_{j+1}) D_j^{-1}$ -(12)

$\qquad$skip column $j+1$

$\quad$else

$\qquad a_j \leftarrow w_j / A_{j,j}$ -(13)

undo row swaps applied to $a_k', a_{k+1}', \ldots$ in A^L -(14)

$A' \mathrel{-}= A^L (W^L)^T$ -(15)

Figure 8: LAPACK algorithm for partially factorizing an $N \times N$ symmetric indefinite matrix A

9

point of view of the efficiency of the total factor and back-solve computation. A formal matrix-notation description of the back-solve algorithm can also be found in [18]; the informal description below will suffice for our purposes.

Let P denote the (symmetric) permutation matrix representing the interchanges from the factor stage. First the updates associated with column j of D^{-1}, L^{-1}, P are applied to $X_{j:N-1}$, in a loop iterating from $j = 0 : N-1$. Note that here, a rank-1 update (or rank-2, in the case of D4) using $L_{j+1:N-1,j}$ is applied to $X_{j+1:N-1}$.

The pivoting on X has to occur in this single loop (rather than in a separate loop) is a direct consequence of using the LAPACK standard form. This makes blocking of the computations (and communications) of this algorithm difficult.

Second, the updates associated with column j of $P^{-1}L^{-T}$ are applied to $X_{0:j-1}$ in a loop iterating from $j = N-1 : 0$.

5.3 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 or panel width $\omega > 1$, has been shown to be efficient on memory hierarchy uniprocessors [2, 13, 14, 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 algorithm depicted by Figure 8 should have a straightforward parallelization that is also reasonably efficient. However, several modifications and optimizations can still be performed.

We will consider the $r \times s$ block-cyclic matrix distribution over a $P \times Q$ logical processor grid [5], 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 load balance advantages, at the expense of extra communication startup costs; it has been shown to yield better performance across platforms with relatively low communication costs [20]. Our implementation encompasses both techniques.

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

5.3.1 Array Element Access

In the distributed memory context it is important to minimize the communication startup costs associated in the manipulation or use of single array elements, which occurs extensively in the diagonal pivoting algorithms.

The DBLAS vector maximum finding function returns both the index and the value of the maximum element (eg. i and $W_{i,j}$ in step -(2)) of Figure 8.

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 $W_{i,i}$. By writing a modified maximum finding function which chooses the cell holding $W_{k,k}$ as the root of the column-wise reduction and broadcast, $W_{k,k}$ can be broadcast along with i and $W_{i,j}$, saving $\lg P + \lg Q$ startups per column.

This was applied similarly to step -(5). Furthermore, here, the condition 'excluding $w_{i,j+1}$' suggests two separate column searches, as is done in the LAPACK code. For the parallel implementation, it is more efficient to temporarily zero $W_{i,j+1}$ 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 -(9) or -(11)) saves 2 startups each time these cases apply.

Before step -(10), temporarily set $A_{j,j+1}$ to $W_{j,j+1}(= A_{j+1,j})$. As W and A are aligned, this requires no communication. Thus, the RHS of of the first sub-step of -(10) becomes simply $A_{j:i-1,j+1}$, saving 1 startup for each instance of case D4.

Similarly, this was applied to the RHS of step -(8), saving a startup for each instance of case D3.

In the factor stage, we record the elements of D ($W_{j,j}$ for cases D1-3, and the elements of D_j for case D4) in a column-replicated vector. This saves $N \lg Q$ startups in the back-solve stage. Note that for case D4, the values of all elements

of each D_j were broadcast to all processors in steps -(2) and -(5).

Consider the net effect of these optimizations for storage blocking ($\omega = r \gg 1$) and $P = Q$ (which will minimize startups) on the combined factor and back-solve stages. For a definite matrix (D1 applies in all cases), these have reduced the number of startups from $\approx 7N \lg P$ to $\approx 4N \lg P$. For an indefinite matrices with for example $\frac{N}{2}, \frac{N}{6}, \frac{N}{6}, \frac{N}{6}$ columns eliminated by cases D1, D2, D3 and D4 respectively, they have reduced the number of startups from $\approx (11.5 \lg P + 5.3)N$ to $\approx (6 \lg P + 4)N$.

5.3.2 Improving Performance in the Back-Solve Stage

While the back-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. The back-solve stage is also used for the FFS method of Section 4.

The LAPACK algorithm 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 back-solve stage can be implemented by the standard DBLAS triangular matrix solve routine to perform the second and fourth steps of:

$$X \leftarrow P^{-1}X \; ; \; X \leftarrow L^{-1}X \; ; \; X \leftarrow D^{-1}X \; ;$$
$$X \leftarrow L^{-T}X \; ; \; X \leftarrow P^{-T}X$$

Here P is the permutation matrix of the accumulated interchanges from the factorization stage. 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. This routine we name `pzsytrs()`.

Although this scheme results in increased communication volume costs in the factor stage, this is compensated for in its greater reduction in the back-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 [20].

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.

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 storage block size r. Such an optimization would be difficult to perform on the LAPACK algorithm, as the pivoting especially in the first loop hinders the blocking of communications.

5.4 A New Reduced Interchange Variant

As explained in Section 5.1.1, minimizing the amount of symmetric interchanges (while keeping the algorithm stable) has potentially large gains in the parallel algorithm performance.

One method of achieving this is implementing a key idea in the algorithm in [13], which is to use the accumulated growth of the previous columns in the test for case D1. This algorithm was largely motivated by the requirements of band matrices, where the minimization of interchanges helps preserve the structure of these matrices [13].

Let j be the current column of A^L to be eliminated, and let column $j - p$, $0 < p \leq j$ be the

last column not eliminated by case D1. Let λ_i be the absolute value of the maximum element of the ith subdiagonal. Then the condition for determining case D1 can be relaxed to:

$$\mu_j = \Pi^j_{i=j-p+1}(1 + \frac{|W_{i,i}|}{\lambda_i}) \le (1 + \frac{1}{\alpha})^p \qquad (3)$$

This scheme is no worse than the original Bunch-Kaufman algorithm in the sense that it attains the same growth bound in the reduced matrix A' [13]. 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 algorithm in [13] is based on a different (3-case) variant of the diagonal pivoting method [6], and furthermore uses an *a priori* growth bound instead, which requires a (worst-case) estimation of λ_i before the updates from columns $k, \ldots, i-1$ are applied. This is necessarily more conservative than using the actual λ_i; indeed for the purposes of most ACCUFIELD matrices, and a target blocking factor $\omega = 44$, it was found to be too conservative to be useful.

In the algorithm of [13], the value of p becomes reset whenever the target blocking factor ω is reached, *or the a priori* bound was exceeded. 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 [13], resulting in a potentially serious reduction in computational performance. Table 1 indicates the blocking factors p achieved by the *a priori* growth bound of [13] on simulated matrices of the form $A = A' + \beta I$, where A' has random elements from the unit square and $0 \le \beta \le 10$ (ACCUFIELD matrices show similar pivoting behavior to these simulated matrices for $1 \le \beta \le 10$). These small values of p are all the more disappointing as when applying Equation 3 *a postiori*, as described below, *no* interchanges were found to be necessary for the experiments of Table 1.

Furthermore, in a distributed memory implementation, it would compromise the advantages

β	4	6	8	10	15	20	25	30
p	2	2	2	4	6	16	32	37

Table 1: Average blocking factor p using *a priori* growth bounds for complex simulated matrices of various β for $N = 528$ and a target blocking factor $\omega = 44$

of using storage blocking, as often the panel A^L would be straddling a storage block boundary.

Our implementation of this idea is different in that Equation 3 is applied *a postiori*, resulting in a greater reduction of interchanges, and we apply it to the 4-case LAPACK diagonal pivoting algorithm. Our implementation also limits p to the range $0 \le p < p_{max} = 64$, which is necessary to avoid overflow in μ_j. 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_j can include contributions from columns in previous blocks. Furthermore, the optimal blocking factor is always met regardless of whether Equation 3 is. We call this the *reduced interchange* variant.

A potential problem with such methods is that, compared with the original Bunch-Kaufman method, they allow increased growth in L_j by a factor of $|\frac{a_{j,j}}{\lambda_j}| \approx \alpha^p$ [3]. A compromise, which we call the *guarded reduced interchange* variant, would be to disallow case D1 when $|\frac{a_{j,j}}{\lambda_j}| > \alpha^{p_0}$, where $p_0 \ll p_{max}$, ie. $p_0 = 5$, even when Equation 3 is satisfied.

Table 2 lists the normalized residual (using a random RHS vector with elements from the unit square) for the original and reduced interchange variants of the algorithm for sample AC-CUFIELD matrices. The normalized residual is calculated in the same way as in LAPACK test programs [7]; ideally, an accurate algorithm will produce normalized 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 9 extends this study to simulated matrices. These represent the averaged values of the normalized residual (and f) for 20 such matrices as functions of the diagonal bias β. In

N :	161		1601		4736	
original:	.01	(.06)	.01	(.07)	.45	(.98)
reduced:	.02	(.07)	.02	(.01)	1.2	(.03)
guarded:	.02	(.05)	.02	(.02)	1.0	(.02)

Table 2: Comparison of residual (and f) for diagonal pivoting variants for ACCUFIELD matrices

terms of the pivot distribution f, the range $5 \leq \beta \leq 7$ corresponds to Table 2. In terms of accuracy, the reduced interchange variants have a residual generally within twice that of the original variant, except at $\beta = 4$, where the average of the unguarded variant is much higher, due to greatly increased growth on a single matrix. However, as these residuals are all within their threshold value (unity), in most circumstances this is not a serious point for concern. The reduced interchange variants does significantly reduce f for $\beta > 1$; in particular $f \approx 0$ for $\beta > 5$.

The guarded reduced interchange variant, as it shows to be a suitable compromise between accuracy and performance, is chosen for the ACCUFIELD solver.

6 Fast Computational Components for the Solver

Parallelism is of course a means to an end: speeding up the application. In this section, we discuss the development of computational kernels required by the solver which also have brought substantial performance gains to the ACCUFIELD application. While the techniques presented here apply particularly to the UltraSPARC processor, most can be applied, with similar results, to other RISC processors with a deep memory hierarchy.

From Figure 8, it can be seen that the important computational operations include a (large) triangular matrix multiply (step -15), a matrix-vector multiply (`zgemv()`, steps -(1) and -(4)), vector maximum finding (`izamax()`, steps -(2) and -(5)), and vector scaling (`zscal()`, step -(13)). The triangular matrix multiply is implemented as a series of rectangular matrix multiplies (`zgemm()`); a vector copy routine (`zcopy()`)

is required implicitly in many places. Note also that `zgemv()` is the computationally dominant component of the solve stage (Section 5.3.2) and the FFS method (Section 4). Above, the corresponding BLAS routines have been named; thus an efficient implementation of the BLAS library on the UltraSPARC is required.

In this section, we will describe how these operations were optimized.

6.1 A Fast UltraSPARC BLAS Implementation

For optimal performance, the UltraSPARC requires *software pipelining*, ie. a pair of dependent instructions must be separated in software by the latency of the first. Floating point add and multiply has a latency of 3 cycles [22].

This technique is taken further in the case of *cache lookahead*. This can be used to hide the miss penalty for the top-level cache, which is too small for most computations. The cache lookahead requires each load operation to be initiated at least 8 cycles before the first instruction using that value [22].

Loop unrolling must be used to expose a sufficiently large number of floating point operations in order to perform such deep software pipelining. For nested loops (`zgemm()`, `zgemv()`), the outer loop should be unrolled as this has the effect of reducing the number of load operations and reducing startup costs. The latter is especially important for good performance of `zgemv()` when used in step -(1) of Figure 8.

Table 3 compares the performance of our BLAS (ANU) with that of the commercially available Sun Performance Library 1.2 (SPL). These can be compared with the theoretical maximum (max), which is what the UltraSPARC is capable in the absence of instruction dependencies, cache misses and startup overheads. Comparing the results 16 KB data size with that of 256 KB gives an indication of the effectiveness of cache lookahead in each implementation (noting that startup costs are less significant for the latter).

6.2 Triangular matrix multiply

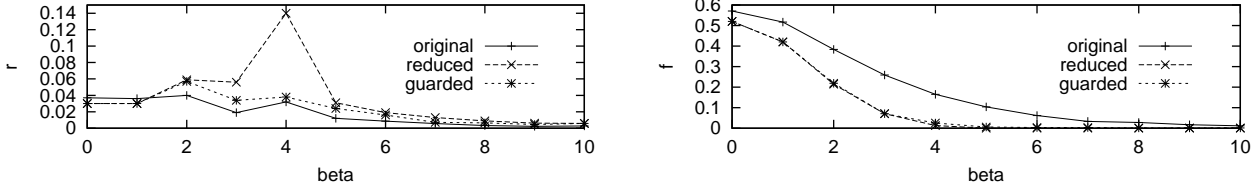An efficient triangular matrix multiply algorithm, capable of dealing with the added com-

Figure 9: Averaged residual and pivot distribution for simulated 500×500 matrices

	unroll	ANU	SPL	max
`zgemm()`	3×2×1	314 / —	270 / —	334
`zgemv()`	4×1	283 / 312	238 / 279	334
`zscal()`	4	223 / 162	239 / 177	250
`zcopy()`	4	164 / 164	164 / 165	167
`izamax()`	4	165 / 166	107 / 100	167

Table 3: BLAS performance in double MFLOPs (16 KB / 256 KB data size) on a 170 MHz UltraSPARC

plexities of block-cyclic matrices, is computationally the most important component of the solver. The corresponding serial LAPACK algorithm (in the routine `_lasyf()`) partitions A', assumed to be column-major, (see Figure 7) into vertical strips of width $\Delta N = \omega$, and applies level 2 operations to update the triangular portion of the strip.

The DBLAS routine partitions A' into *horizontal* strips of larger 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). Using horizontal strips, as is shown in Figure 10, enables $\Delta N > |TLB|$ without causing a large number of TLB misses Thus, at $\omega = 44$, $\Delta N \approx 90$ for double-complex, and the larger strip width 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 algorithm depicted in Figure 10 optimizes cache and TLB usage as follows. ΔN is chosen so that (the contiguous) A requires no more than half of the UltraSPARC 64 TLB entries and no more half of the second level cache. Thus, A remains in the cache and TLB, while different
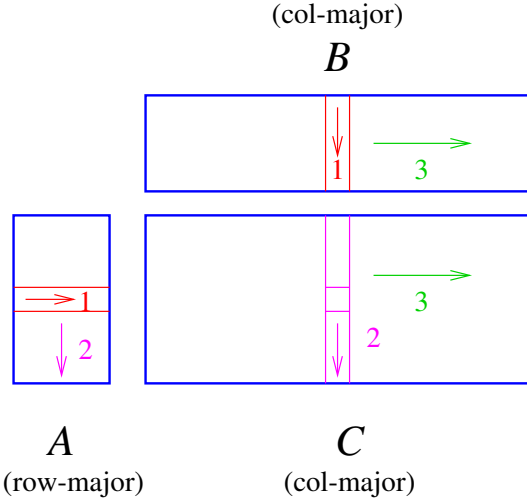

Figure 10: Strip matrix multiply $C \leftarrow C + AB$, A is $\Delta N \times \omega$, indicating optimal storage format and order of iteration

columns of B and C are applied to it.

The innermost iteration (1) has unit stride for both A and B; the next innermost iteration (2) has unit stride on C. In the worst case, there could be one TLB miss for each column segment of C. However, as there will be $2\omega\Delta N$ floating point operations applied for each segment, the effect of the TLB miss should be small for the values of ΔN and ω.

The above ideas were applied to a general rectangular matrix-matrix multiply (`zgemm()`. For large matrices (1–36 MB), the ANU `zgemm()` sustained 280–295 double MFLOPs on an 170 MHz UltraSPARC, as compared with the SPL `zgemm()` which sustained 220-245 MFLOPs.

It is thus argued that this algorithm is optimal for rectangular matrix-multiply for such a cell architecture.

A final issue is the storage format of A and B required by this algorithm. In the context of the parallel LDLT algorithm, these will come from

14

replicated A^L and $(W^L)^T$ respectively. It turns out that for considerations of the efficiency of the broadcast, these will have the opposite storage format as required by Figure 10 [21].

Thus, an efficient local matrix transpose algorithm is also required. By explicitly partitioning the matrices into $\frac{|TLB|}{2} \times \frac{|TLB|}{2}$ chunks, large local matrix transpositions were sped up by a factor of 4–8 [21] on the UltraSPARC, due to the improvement in TLB usage over a naive transpose algorithm.

7 Performance

This section evaluates the direct solver's serial and parallel performance; firstly in isolation and secondly when integrated into the ACCUFIELD application.

The DBLAS implementation of LDLT solver 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 `pzsysv()`.

In terms of performance, the reduced interchange and guarded reduced interchange variants of Section 5.4 were effectively indistinguishable; thus results in this section labelled as 'reduced interchange' apply to either of these two variants.

7.1 Serial BLAS and Direct Solver Performance

Table 4 compares the performance of the LDLT solvers on a 300 MHz UltraSPARC II (U300) for two ACCUFIELD matrices. 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

N	zsysv()		pzsysv(), ANU BLAS	
	SPL BLAS	ANU BLAS	original	reduced
161	75	80	73	74
1601	71	96	108	109

Table 4: LDLT solver performance in complex MFLOPS for ACCUFIELD matrices on a U300 ($\omega = 44$)

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 `pzsysv()`, 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 [19]).

For larger matrices, `pzsysv()` shows a clear improvement in speed, even when using the same (ANU) BLAS. This is primarily because the DBLAS has a more efficient triangular matrix multiply routine than that used in LAPACK's `zsysv()`, as previously mentioned. The overall improvement on the Sun Performance Library 1.2 `zsysv()` thus amounts to 53%. Note that the reduced interchange variant only offered a marginal gain in serial performance. $N = 1601$ is sufficiently large to represent the general asymptotic speed of the solvers.

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

The back-solve stage component of these timings (see Section 5.3.2), while not having a large impact on the overall performance, similarly showed that `zsytrs()` was slightly faster at $N = 161$, but `pzsytrs()` was 50% faster at $N = 1601$, indicating that the method outlined in Section 5.3.2 did indeed achieve some computational advantages.

7.2 Parallel Direct Solver Performance

For these results, MPI was used as the underlying communication library.

Figure 11 gives parallel factorization 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 interchange.

Comparing with Table 4, 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 interchange version 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 version, 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 $N \lg Q$ startups and an extra $\frac{N}{P} \lg Q$ communication volume to reduce horizontally a_j for the vector-matrix multiply. For moderate-large N, storage blocking has an unusual advantage on the AP3000: its larger messages in the broadcast of A^L and W^T can take better advantage of the protocol communication method [17], effectively achieving a higher bandwidth.

However, for LU, and to a lesser extent LDLT without reduced interchanges, algorithmic blocking at $\omega = 48, r = 4$ slightly outperformed storage blocking for $N > 5000$. While the highly optimized LU implementation [20] achieved somewhat higher speeds for a given N, they are never greater than by a factor of 1.3 for $N \geq 2000$. In other words, the LDLT factorization is clearly quicker than LU in this range, with the residuals for LU being only marginally smaller.

7.3 Application Performance

This section investigates the performance of the analysis stage of ACCUFIELD, where computation time is dominated by the solver stage (direct solution only is used here).

The results here are for an AP3000 comprised of 300 MHz UltraSPARC processors with 2 GB memory.

Firstly, the computing time of the current (nearly equal to the computing time of the matrix) on the wire surface is measured for the di-
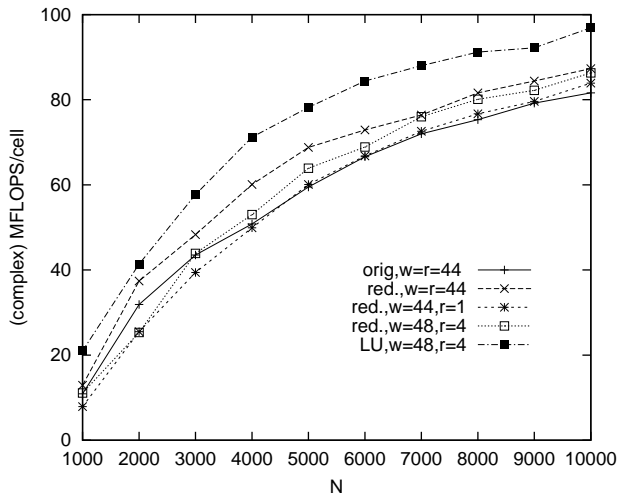

Figure 11: LDLT factorization performance for simulated complex $N \times N$ matrices with $\beta = 7$ on a 4×4 300 MHz AP3000

	1 CPU	16 CPU
pzsysv()	2570s (0.6)	203s (0.05)
SPL	4131s (1)	–

Table 5: Comparison of the current computing time for the dipole antenna on a 300 MHz AP3000

pole antenna model shown in Figure 12. A driver is inserted in gap at the center of the wire. In this analysis, a matrix equation with $N = 9500$ is solved, where N is the dimension of the immittance matrix. The computing time of the current with pzsysv() (with enhanced BLAS) or SPL (SUN Performance Library zsysv() and BLAS)) is shown in Table 5. SPL is the conventionally used library to solve the system of equations; it can only run on 1 node. When the node number is 1 and pzsysv() is utilized, the matrix calculation becomes 1.6 times faster, comparing with the case utilizing SPL. As the number of nodes increases from 1 to 16, the matrix calculation becomes 12.7 times faster.

Next, the electromagnetic wave radiated from the Note PC model is simulated as an example of the analysis of the electronic devices at the design stage. The model is shown in Figure 13. In this analysis, a matrix equation with $N = 5041$ is calculated. The dependence of the computing time on the node number of AP3000 is shown in
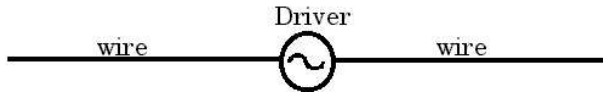
16

Figure 12: Dipole antenna model. The wire structure to be modeled consists of 9501 segments. Diameter and Length of each element are 1mm and 1cm, respectively. The frequency of the driver is 300 MHz
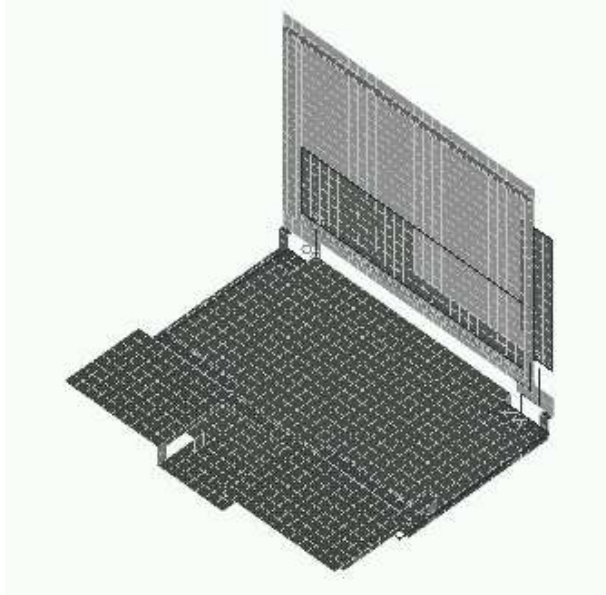

Figure 13: Note PC model The model consists of 2706 patch elements and 33 wire elements. Analyzed frequency is 66.66MHz

Figure 14. In this figure, an open circle indicates the computing time with SPL and closed circles indicate the time with DBLAS. Compared with the case using only one processor with SPL, the calculation becomes 14 / 8 times faster when the current is calculated with eight / four processors.

In the result shown in Figure 14, the current is calculated for the single frequency (66.66 MHz). In the actual design stage, the current is often calculated repeatedly for wide-range of frequencies as was described in the introduction. In case of the Note PC model shown in Figure 13, the calculation of the current requires few or more hours. If `pzsysv()` is utilized, the current computing time is found to be reduced to few or few ten minutes and the electronic device such as the Note PC can be designed more efficiently.
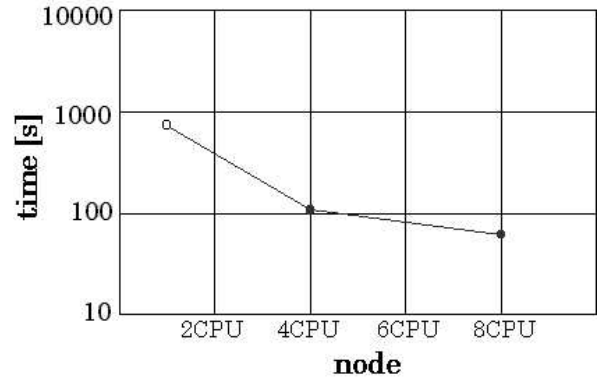

Figure 14: The dependence of the current calculation time on the node number of AP3000 for the Note PC model

7.4 Performance of the Fast Frequency Stepping Method

This section describes the application performance of the Fast Frequency Stepping method, which as explained in Section 4, enables further performance improvements for analysis with multiple frequencies.

As mentioned in Section 4, we heuristically stop the iterative solving with a LDLT preconditioner if its computing time exceeds the half of the time of a direct solve. In the examples below executed in parallel, this resulted in the number of iterative solves per central frequency being at most two, which are for ω_{i_c-1} and ω_{i_c+1}, where a direct solution is used at a central frequency ω_{i_c}.

Figure 15 shows the execution times and the number of iterations r of parallel FFS for a POS terminal, generating a linear system of size $N = 7017$. The frequencies are in the range 30 to 800 MHz. Note that the convergence is slow at lower frequencies, which is a common phenomenon of FFS. A value of $r = 0$ indicates a direct solution was obtained for this frequency. The averaged iterative solution time is 66.5s, and the average direct time is 160.3s; thus the application of FFS (over using a direct solution at each frequency) results in a speed up of 1.65.

Figure 16 shows the execution times of parallel FFS for a PenNote PC. Here the matrix size $N = 4608$. The set of frequencies is a union of two sequences $\{100n/3\}$ and $\{24n\}$,
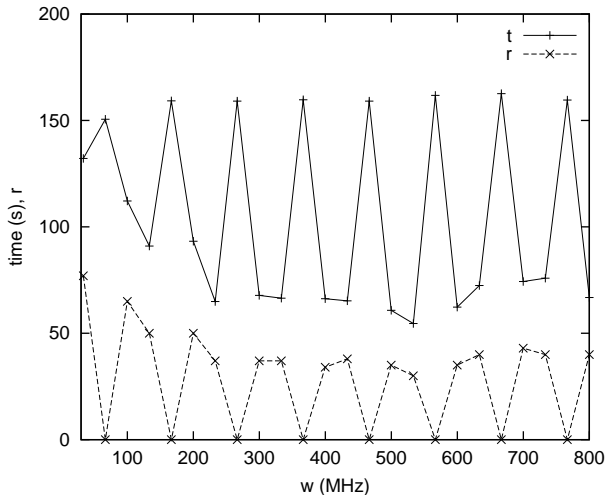
17

Figure 15: FFS solution time t and number of iterations r for each frequency for a POS terminal on an 8 node 360MHz AP3000

Figure 16: FFS solution time for each frequency for a PenNote PC on an 8 node 360MHz AP3000

$n = 1, 2, 3, \ldots$. So some adjacent frequencies are close to each other and FFS is particularly efficient for such pairs, as more rapid convergence occurs ($r \approx 10$ for such pairs, half of the average value for this model). Here, the averaged iterative solution time is 16.2s, whereas that of the direct solution is 57.7s. Hence, due to a faster average convergence rate, a higher speedup of FFS of 2.03 results.

When applying FFS to a sequential computation, we observed that a single preconditioner covers wider range of frequencies. For example, the PenNote PC analysis of Figure 16 required only four direct solvings when sequential FFS was applied. Compared with the sequential case the speedup by the parallel FFS is not as large.

The reason for this is that the parallel speedup of level 2 functions such as `pzsymv()` and `pzsytrs()` is not as high as for level 3 computations. Over 95% of the execution time was spent in `pzsymv()` and `pzsytrs()`. Of this, between 60% and 70% was spent in `pzsytrs()`, which (even in the absence of interchanges) requires at least $12N$ communication startups on an 8 node machine.

However, for larger N, this speedup of `pzsytrs()` will improve; this, together with the $O(N)$ reduction of floating point operations enabled by FFS, means that better improvements for FFS are expected for larger models.
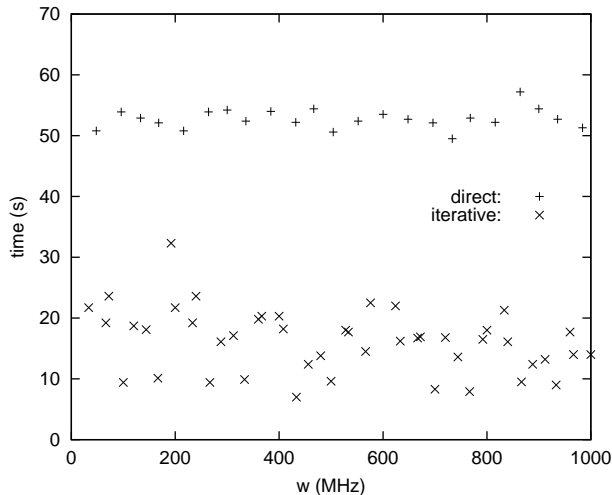
8 Conclusions

The EMC application ACCUFIELD can accurately analyse electromagnetic fields emitted by electronic devices, using a combination of the moment method and the transmission line approximation method. It also supplies extensive libraries and tools for inputting or modifying models of the devices, and for interpreting the results of the analysis, which makes the application highly usable.

However, for complex electronic devices, the resulting models can be very large (up to tens of thousands of elements), requiring huge computational and memory resources for the results to be generated in an acceptable time. Distributed memory parallel computing offers a cost-effective and scalable way of providing these resources.

In this paper, we have shown how ACCU-FIELD can be efficiently parallelized on a distribed memory multicomputer. The parallelization occurs mainly over the compute-intensive solve stage, which involves the solution of complex dense symmetric indefinite linear systems. Often, these systems are weakly indefinite, a property which we have used to speed up their solution.

The main method of solution is a direct method that was based on the Bunch-Kaufman LDLT factorization algorithm for symmetric indefinite matrices, for several reasons. Firstly, its implementation in LAPACK has been shown to

18

be highly competitive in terms of performance with other methods. Secondly, as compared with tridiagonal methods, the diagonal pivoting method offers a reduced amount of symmetric interchanges, especially for weakly indefinite systems, a property that we have argued is particularly important for efficient parallel implementation. Thirdly, they afford a highly parallelizable back-solve stage, which is important in this context.

The efficient parallelization of the LAPACK LDLT factorization and back=solve algorithms required careful design in order to minimize communication costs, potentially much higher than for LU or LLT factorizations. By merging and re-using communications over the two stages, communication startup costs were reduced almost by a factor of 2. The factor stage also had to be reorganized in order to achieve a back-solve stage with high load balance and an $O(N)$ communication volume cost. The resulting back-solve stage is not only simplified but has superior serial performance and affords further reduction in startup costs, by blocking communications.

To further speed up the parallel factorization stage, we developed a new variant of the Bunch-Kaufman algorithm which uses an accumulated growth bound, and yet always achieved the target blocking factor, thus avoiding any performance degradation. We found that only by applying this bound *a postiori* resulted in a highly effective (further) reduction of symmetric interchanges over a range of weakly indefinite matrices. After an evaluation of its accuracy, a guard condition was introduced to keep this very close to that of the original algorithm while retaining most of the performance benefits.

When used in the ACCUFIELD application, the direct solver achieved speedups of 12–13 on a 16 node 300 MHz AP3000 for matrices of the order of $N \approx 10000$, and achieved a floating point speed within 30% of a heavily optimized LU solver for $2000 \leq N \leq 10000$.

To further speedup the application on UltraSPARC-based machines, it is necessary to use highly optimized BLAS kernels. As compared with using commercially available BLAS libraries, these kernels improved solver performance by $\approx 35\%$ for large matrices. Furthermore, the efficiency of the triangular matrix-

multiply, the computationally dominant part of the solver, should not be overlooked. Our algorithm, which we have argued to be optimal for modern RISC processors with a deep memory hierarchy, achieved a further 15% improvement.

We have also explored an alternate approach to speeding up the application for the case of EMC analysis over multiple frequencies. The FFS method employed the iterative conjugate gradient method using the direct solution over a central frequency as a preconditioner. While it achieved a time reduction by a factor of only ≈ 2 for parallel solution on moderate sized systems, much greater reductions are expected for larger systems.

Future work includes investigating further methods of improving parallel performance for the direct and iterative solvers. For the former, the full potential of our reduced interchange variant could be realized by an algorithm that can predict when a (sub-) block column can be eliminated by case D1; that elimination can then be done using LLT-like transformations, which yield computational and communication advantages. Alternate diagonal pivoting algorithms, also with potential for a reduced number of interchanges, could also be investigated on the criteria of an accuracy–performance tradeoff. For the latter, applying FFS simultaneously over several (say 4) vectors could amortize the communication costs, as well as result in increased computational speed.

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. *SIMAX*, 20(2):513–561, 1998.

[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] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, J. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Linear Algebra Library for Message Passing Computers. In *SIAM Conference on Parallel Processing*, March 1997.

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

[8] S. Ohtsu et al. EMC Analysis of Radiation and Immunity by Moment Method utilizing Frequency Characteristic of Mutual Impedance. In *EMC'96: The International Conference on Electromagnetic Compatibility*, Roma, September 1996.

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

[10] R.F. Harrington. Moment Method of Field Problems. *Proceedings of the IEEE*, February 1967.

[11] G. Hoyler and R.Unbehauen. An efficient alogrithm for the treatment of multiple frequencies with the method of moments. In *EMC'96: The International Conference on Electromagnetic Compatibility*, Roma, September 1996.

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

[13] 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.

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

[15] A.A. Kishk and L. Shafai. The Effect of Various Parameters of Circular Microstrip Antennas and their Radiation Efficiency and the Mode Excitation. *IEEE Transactions on Antenna and Propagation*, August 1986.

[16] E.H. Newman. Electromagnetic Modelling of Wire and Surface Geometries. *IEEE Transactions on Antenna and Propagation*, November 1978.

[17] 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, September 1998. National University of Singapore. paper P1-E.

[18] P. E. Strazdins. A Dense Complex Symmetric Indefinite Solver for the Fujitsu AP3000. Technical Report TR-CS-99-01, Computer Science Dept, Australian National University, May 1999.

[19] 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.

[20] 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.

[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, September 1998. School of Computing, National University of Singapore. paper P1-E.

[22] M. Tremblay and J.M. O'Connor. Ultrasparc I: a four-issue processor supporting multimedia. *IEEE Micro*, pages 42–49, April 1996.