HIGH PERFORMANCE DENSE LINEAR SYSTEMS SOLUTION ON A BEOWULF CLUSTER

P.E. Strazdins
Department of Computer Science, Australian National University,
Acton, ACT 0200, Australia.
peter@cs.anu.edu.au

Abstract

In this paper, we describe techniques which can improve the performance of dense linear system solution, based on LU, LLT and QR factorizations, on distributed memory multiprocessors, including cluster computers.

The most important of these are refinements of the algorithmic blocking technique which reduce the bulk of its introduced communication startup costs and make the technique superior to storage blocking in terms of communication volume costs. These primarily rely on pipelined communication and the choice of a small storage block size.

Two other techniques, optimizing the memory behavior in multiple row swaps, and the coalescing of vector-matrix multiplies in QR, also afford modest improvements in storage blocking and serial performance.

Performance results on a 24 node Beowulf cluster with 550 MHz dual SMP Pentium III nodes connected by a COTS switch with 10 Mb/s links, show that algorithmic blocking generally improves performance by 15–30% or more for these computations over a large range of system sizes.

KeyWords and Phrases: linear systems, dense linear algebra, parallel computing, cluster computing, block-cyclic decomposition, algorithmic blocking.

1 INTRODUCTION

Dense linear algebra computations such as LU, LLT (Cholesky) and QR factorization (see Section 2) require the technique of 'block-partitioned algorithms' for their efficient implementation on memory-hierarchy processors. Here, the rows and/or columns of a matrix are partitioned into *panels*, ie. block row/columns of width $\omega \geq 1$, which are typically formed by matrix-vector operations. The remainder of the computation typically involves 'Level 3' or matrix-matrix operations, which can run at optimal speed.

In the distributed memory parallel computer context, most, if not all, communication occurs within the panel formation stages. Once the optimal panel width $\omega = \omega_m$ is achieved for the

matrix-matrix operations, the most scope for the acceleration of the overall computation is in the panel formation stages.

In this paper, we will consider the $r \times s$ block-cyclic matrix distribution over a $P \times Q$ logical processor grid (Choi, Dongarra, Ostrouchov, Petitet, Walker & Whaley 1996). Here, an $N \times N$ global matrix A is divided into (storage) blocks which are contiguous $r \times s$ sub-matrices; block (i, j) of A is then stored on processor $(i \bmod P, j \bmod Q)$. We will now review two established techniques for parallel panel formation, known as *storage blocking*, where $\omega = r = s$, and *algorithmic blocking* (Hendrickson & Womble 1994, Strazdins 1995, Petitet 1996, Stanley 1997), where $\omega \approx \omega_m, r \approx s \approx 1$.

Storage blocking suffers from load imbalance on the panel formation stage, in that only one row or column of processors of the grid will be involved in this stage, resulting in an $O(N^2(r/Q + s/P))$ penalty. It also exacerbates an unavoidable load imbalance of the same order on the Level 3 computation, which is due to the cell owning the last and column block row having more overall work to do than the others in these stages (Greer & Henry 1997, Stanley 1997).

Algorithmic blocking can overcome much of this imbalance, and has been shown to yield significant performance gains for various matrix factorizations (Brent 1992, Hendrickson & Womble 1994, Strazdins 1995, Strazdins 1998*b*) and reductions (Stanley 1997). However, these have all been on earlier vendor-built distributed memory parallel computers such as the Fujitsu AP+ and the Intel Paragon, with relatively low communication costs compared with a cluster computer. Reflecting this, all of these studies have either assumed or shown that a storage block size of $r = s = 1$ was optimal for algorithmic blocking on those platforms.

However, storage blocking is widely believed to minimize communication costs, especially communication startup costs. There has been a strong trend of relative increase in communication costs relative to processor (Level 3) floating point performance seen in distributed memory multiprocessors with vendor-supplied communication networks over the mid 90's (Dongarra & Donigan 1997). With the advent of the cluster computing model, this trend has increased sharply, due to the use of the slower COTS communication networks typically used for this model. This is further exacerbated by the generality required by the cluster model, where safety and security requirements may impose several layers of overhead on communication, and possibly even the negotiation of a full TCP/IP stack.

Furthermore, storage blocking is relatively simple and robust to implement, an important requirement for general, portable parallel dense linear algebra libraries. To this date, ScaLAPACK (Blackford, Choi, Cleary, D'Azevedo, Demmel, Dhillon, Dongarra, Hammarling, Henry, Petitet, Stanley, Walker & Whaley 1997) is by far the most widely used such library, using algorithms based on storage blocking (Choi et al. 1996).

Hence, whether algorithmic blocking can yield worthwhile, or indeed any, performance gains in performance for cluster computers has been so far largely unresolved.

The main original contributions of this paper are firstly to show how algorithmic blocking can be implemented to have little extra communication startup overheads and, surprisingly, negative extra communication volume overheads over storage blocking (Section 2). Secondly, we also there describe new optimizations that also improve the performance of storage blocking (including serial performance) for LU and QR factorizations as well. Thirdly, to show, particularly on clusters, that for algorithmic blocking the optimal tradeoff between load imbalance and communication costs occurs at $r = s \approx 4$, due to the effects of pipelined

communication. Fourthly, to demonstrate that even on a Beowulf cluster with a relatively low-speed network, algorithmic blocking can yield significant performance gains over storage blocking (Section 3), for both small and medium numbers of cells.

We chose linear systems solutions based on LU, LLT and QR factorization because each of these involves different potential load balance and communication overhead tradeoffs.

For the comparison of the two methods, we choose a combined factor and backsolve computation, rather than just the matrix factorization. This is fairer because, on a cluster, the relatively slower communication network will increase the cost of the largely communication startup-bound backsolve computation, where a large storage block size can be used to reduce these costs.

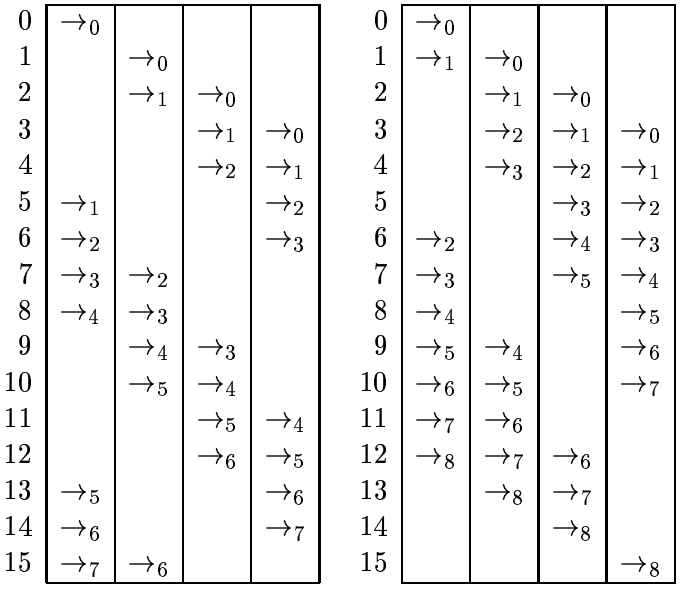
Pipelining with lookahead is an alternate technique using $\omega = r$, which can also improve the load balance in the lower panel formation in LU, LLT and QR factorizations (Greer & Henry 1997, Strazdins 1998*b*, Petitet, Whaley, Dongarra & Cleary 2000). However, we will concentrate on algorithmic blocking in this study as it is more widely applicable in dense linear algebra computations (eg. lookahead cannot be applied to symmetric tridiagonal reduction (Stanley 1997) or LDLT factorization (Strazdins 1999)), and because it does not reduce load imbalance in the upper panel formation or in the Level 3 computation. Furthermore, *lookahead* is much more complex to implement, often requires platform-dependent tuning, and requires a large asynchronous buffering capacity in the underlying communication system (Strazdins 1998*b*). For these reasons, it is less suitable for implementation in portable parallel libraries.

2 ALGORITHM DESIGN FOR FAST ALGORITHMIC BLOCKING

In this section, we describe aggressive optimizations that can be applied to parallel dense linear systems solution. The primary goal is to minimize communication startups and volume for both algorithmic and storage blocking, which is especially important for cluster computing. The secondary goal is to increase computational performance.

An important optimization is the use of the *pipelined* broadcast such computations. Figure 1(a) demonstrates a series of such broadcasts, with broadcasts 0 to 6 (originating from cells 0, 1, 2, 3, 0, 1, and 2 respectively) being completed in the first 16 timesteps. In the context of storage blocking (Choi et al. 1996), this can reduce the communication startup and volume costs by a factor of $\frac{\lg_2 Q}{2}$ over a normal binary tree-based broadcast. It can be applied to all horizontal communications in LU, LLT and QR factorization, as all information flows from left to right. The factor of 2 arises due to the 'bubble' introduced as the source column of the broadcast shifts rightwards. The size of this bubble can be reduced if consecutive broadcasts originate from the same cell, as shown in Figure 1(b).

Our parallel algorithms apply this, not only in the case of storage blocking, but in algorithmic blocking as well. They also apply all other optimizations for storage blocking currently used in ScaLAPACK (Blackford et al. 1997). Our algorithms are based on the DBLAS Distributed BLAS library, which were designed to support algorithmic blocking, and offer reduced software and communication overheads (Strazdins 1998*c*).



(a) alternating sources, cost per b/c is $1 + 1$

(b) paired sources, cost per b/c is $1 + \frac{1}{2}$

Figure 1: Multiple pipelined broadcasts (of equal size) across a 1×4 grid

For the remainder of this paper, we will assume a square block-cyclic distribution is used, ie. $r = s$, for the sake of simplicity.

2.1 LU FACTORIZATION

LU factorization is the decomposition $A \to LU$, where L is a lower triangular matrix and U is an upper triangular matrix with a unit diagonal. Figure 2 indicates the sub-matrices involved in the ith partial LU factorization with row pivoting, using a panel width of ω, for an $N \times N$ matrix A. Here $i_1 = i\omega$, $i_2 = i_1 + \omega$. This is repeated for $i = 0, 1, 2, \ldots, \frac{N}{\omega} - 1$ to give a full factorization. Matrix indices begin from 0, with the notation x' being used as a shorthand for $x - 1$.

The blocked LU factorization algorithm then can be expressed as:

$\texttt{LU2}(N - i_1, \omega, A_{i_1:N',i_1:i_2'}, P_{i_1:i_2'})$
`for` $j \leftarrow i_1 : i_2'$
  $A_{j,0:i_1'} \leftrightarrow A_{P_j,0:i_1'}$; $A_{j,i_2:N'} \leftrightarrow A_{P_j,i_2:N'}$;
$U^i \leftarrow (T^i)^{-1} U^i$ — $T^i = A_{i_1:i_2',i_1:i_2'}$ (triangular, with unit diag.); $U^i = A_{i_1:i_2',i_2:N'}$

$A^i \leftarrow A^i - L^i U^i$ — $A^i = A_{i_2:N',i_2:N'}$, $L^i = A_{i_2:N',i_1:i_2'}$

where $\texttt{LU2}(M, N, A, P)$ is:
  `for` $j \leftarrow 0 : N'$
    find $P_j \in j : M'$ s.t. $|A_{P_j,j}| \geq |A_{j:M',j}|$
    $A_{j,:} \leftrightarrow A_{P_j,:}$ — $u^j = A_{j,j+1:N'}$
    $l^j \leftarrow l^j / A_{j,j}$ — $l^j = A_{j+1:M',j}$
    $L^j \leftarrow L^j - l^j u^j$ — $L^j = A_{j+1:M',j+1:N'}$

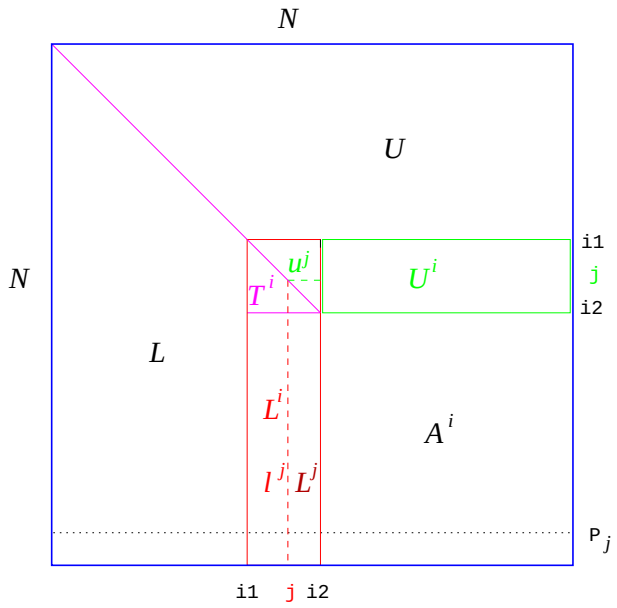


Figure 2: Partial LU factorization of an $N \times N$ matrix A

For storage blocking, the lower panel L^i is held over a column of cells. Hence, the dominant communication overheads are from the startups associated with the determination of P_j and the vertical broadcast of u^j. By combining these with the row swaps within L^i (Petitet et al. 2000) via a (contention-free) binary tree reduce-broadcast, these number $2\lg_2(P)N$.

For algorithmic blocking, l^j and P_j also has to be broadcast horizontally. However, assuming $\omega \geq rQ$, these can be pipelined with reduced cost; indeed, since we will have $2r$ consecutive broadcasts from the same cell column (cf. Figure 1(b)), the pipeline 'bubble' only applies to the first of these; thus only $(2 + \frac{1}{r})N$ startups are actually introduced (Strazdins 1998*a*).

Furthermore, the broadcast columns l^j can be 'cached', so that a re-broadcast of the factored L^i (and T^i) is obviated (Strazdins 1998*a*). Thus, algorithmic blocking need introduce no extra communication volume costs. This however introduces a problem: subsequent row swaps have to be applied to the 'cached' l^j columns as well.

For algorithmic blocking with $r > 1$, it may seem tempting to block the communications of l^j to reduce the number of introduced startups to $\frac{1}{r}N$. Apart from introducing extra complexity, there are two reasons not to do so. Firstly, unless ω is several times larger than rQ, the load balance will be lower. For example, at $\omega = 80, r = 4, Q = 8$, modelling techniques predict that the degree of load balance would decrease from 0.72 to 0.61 (Strazdins 1995). Secondly, and more importantly, the effective overall communication volume due to l_i is reduced from $2 \cdot \frac{N^2}{2}$ to $(1 + \frac{1}{r}) \cdot \frac{N^2}{2}$, again due to the hiding of the pipeline bubble mentioned earlier.

Similarly, in the formation of U^i, pipelined communication can still be applied on the individual rows of U^i, which are broadcast vertically (Strazdins 1998*a*). This optimization is encapsulated in the standard DBLAS triangular update routine (with an extended interface to return the cached rows). This similarly introduces $(1 + \frac{1}{r})N$ startups, but again these can be cached. The communication volume cost of the broadcast of U^i is *reduced* by algorithmic blocking by a factor of $\frac{\lg_2 P}{(1+\frac{1}{r})}$ also.

There is also scope for optimizations in the application of the multiple row swaps:

1. the row swaps in the 'cached' l^j columns can be combined with those in U^i, ie. the row segments of both are packed (unpacked) into a buffer before (after) communication. This means that 'caching' the l^j columns and hence algorithmic blocking introduces no extra message startup overheads here.

2. assuming column major storage (the standard for such computations), if this packing (and unpacking) stage is performed over all affected rows, the memory access behavior of this stage can be significantly improved (Strazdins 1998*a*). This is because by blocking the rows in chunks of $\approx \frac{|\mathrm{TLB}|}{2}$, where $|\mathrm{TLB}|$ is the number of TLB entries for data accesses, the number of TLB and top-level cache misses can be reduced. This optimization actually aids storage blocking to a greater extent, since the cell row holding U^i has all ω source columns. Subsequent to its introduction, this technique has been shown to improve serial LU performance on Alpha 21164, UltraSPARC 2200 and R10000 processors, and was proposed for inclusion in LAPACK `dlaswap()` (Whaley 1998).

3. algorithmic blocking will automatically parallelize a multiple swap operation (Strazdins 1995), including communications and TLB misses, due to the fact that all cell rows will hold a roughly equal number of rows in the ranges $i_1 : i_2$ and $P[i_1 : i_2]$, provided $\omega \geq rP$. The degree of this parallelization is not perfect, due to some interference between row swap operations, but simulation studies have shown that it is $\approx \frac{P}{2}$ for contention-free networks. This effect also results in a similarly reduced vertical communication volume cost over storage blocking.

In the context of a linear system solve, note that the row swap to the left of L^i need not be performed (see Section 2.4).

2.2 LLT FACTORIZATION

LLT factorization is the decomposition $A \rightarrow LL^T$, where L is a lower triangular matrix; A must be a symmetric positive definite matrix. LLT factorization differs from LU in that there is no pivoting. This means that L^i can be formed via a Level 3 computation, making the panel formation stage much less expensive than in LU or QR. The exploitation of symmetry means that only the lower triangular half of A is accessed (note that the resulting reduced number of FLOPs exacerbates the imbalance due to r in Level 3 computations also). The partial blocked LLT factorization (cf. Fig 2) can be expressed as:

`LLT2`(ω, T^i)	$T^i = A_{i_1:i'_2, i_1:i'_2}$ (tri.)
$L^i \leftarrow L^i (T^i)^{-T}$	$L^i = A_{i_2:N', i_1:i'_2}$
$A^i \leftarrow A^i - L^i (L^i)^T$	$A^i = A_{i_2:N', i_2:N'}$ (lower triangular)
where `LLT2`(N, A) is:	
`for` $j \leftarrow 0 : N'$	
$A_{j,j} \leftarrow \sqrt{A_{j,j}}$	
$l^j \leftarrow l^j / A_{j,j}$	$l^j = A_{j+1:N', j}$
$L^j \leftarrow L^j - l^j (l^j)^T$	$L^j = A_{j+1:N', j+1:N'}$ (lower triangular)

For storage blocking, there are $O(\frac{N}{r})$ communication startups, considerably smaller than the $O(N \lg_2 P)$ startups in LU and QR. It is well known that communication costs for this computation are minimized if $P = Q$ (Choi et al. 1996), which we will assume in this section.

In the formation of L^i using algorithmic blocking, pipelined communication can similarly reduce the introduced startups to $(1 + \frac{1}{r})N$, and offers a reduced communication volume cost over storage blocking by a factor of $\frac{2}{(1+1/r)}$, as for LU. However, it is in the factorization of T^i that challenges occur for reducing communication startup costs.

Firstly, pipelined broadcasts may be applied to the horizontal broadcasts of the columns l^j and to their vertical re-broadcast, at a cost of $2(1 + \frac{1}{r})N$ introduced startups. However, as the amount of work involved is small, T^i could be semi-replicated, across cell rows, before the factorization, reducing the number of horizontal communication startups (Strazdins 1998*a*). Indeed, it could be fully replicated across all cells; using multi-broadcast techniques (Mitra, Payne, Shuler, van de Geijn & Watts 1995, Strazdins 1998*c*), this would only introduce $\approx 2 \cdot \frac{NP}{\omega}$ startups.

2.3 QR FACTORIZATION

QR factorization is the decomposition $A \rightarrow QR$, where Q is an orthogonal matrix and R is an upper triangular matrix with a unit diagonal. QR factorization is by far the most complex computation of the three, but thereby offers some interesting scope for optimization. Also, unlike in LU or LLT, the blocked-partitioned algorithm introduces redundant $O(N^2\omega)$ computations; this factor favours the selection of a smaller ω (and hence favours storage blocking).

The partial blocked QR factorization (cf. Figure 2) can be expressed as:

QR2$(N - i_1, \omega, V^i, T)$	$V^i = A_{i_1:N', i_1:i'_2}$, T is $\omega \times \omega$
$W \leftarrow (V'^i)^T A'^i$	$A'^i = A_{i_1:N', i_2:N'}$
$W \leftarrow TW$	W is $\omega \times (N - i_2)$
$A'^i \leftarrow A'^i - V'^i W$	V'^i is the trapezoidal part of V^i (with a unit diag.)
where QR2(M, N, A, T) is:	
for $j \leftarrow 0 : N'$	
$a \leftarrow A_{j,j}$; $A_{j,j} \leftarrow 1$	
$(t^j, \beta', w^j) \leftarrow A^T_{j:M',j} A_{j:M',:}$	$t^j = T_{j,0:j'}$
$(A_{j,j}, \tau, \beta) \leftarrow f(a, \beta')$	$f()$ is a scalar function
$l^j \leftarrow l^j / \beta$;	$l^j = A_{j+1:M',j}$
$L^j \leftarrow L^j - \frac{\tau}{\beta} l^j w^j$	$L^j = A_{j+1:M',j+1:N'}$
$t^j \leftarrow \frac{\tau}{\beta} t^j$; $T_{j,j} = \tau$	
for $j \leftarrow 0 : N'$	
$t^j \leftarrow T^j t^j$	$T^j = T_{0:j',0:j'}$ (lower triangular)

In a parallel implementation, the temporary matrix W is aligned with U^i, and T is aligned with T^i (cf. Figure 2).

The new optimization applied here is to merge two vector-matrix multiplies and the dot product $v^j \cdot v^j$ into a single vector-matrix multiply (Strazdins 1998*a*). Note that the temporary

vector w^j (aligned with u^j, cf. Figure 2) can be stored in the upper triangular half of T, and $T_{j,j}$ can be used to store β'.

Previously, implementations such as LAPACK and ScaLAPACK (Choi et al. 1996) performed instead:

$$\beta' = l^j \cdot l^j; \;\; l^j \leftarrow l^j/\beta; \;\; w^j \leftarrow A^T_{j:M',j} A_{j:M',:}$$
$$L^j \leftarrow L^j - \tau l^j w^j; \;\; t^j \leftarrow \tau A^T_{j:M',j} A_{j:M',:}$$

Using a merged vector-matrix multiply has the following advantages:

1. better computational speed, with a wider vector-matrix multiply, improving parallel and serial performance.

2. reducing the dominant communication startups of the computation, from $6\lg_2(P)N$, as in ScaLAPACK QR, to $2\lg_2(P)N$ (for both algorithmic and storage blocking). This is possible if β and τ are broadcast in the same message as w^j.

3. perfect load balance is achieved for algorithmic blocking, as the the matrix width is always ω.

Similar to LU, pipelined horizontal broadcasts and caching can be applied to l^j, to save re-communication for the rank-1 update on L^j, as well as for the subsequent Level 3 computations. τ must then be broadcast horizontally, so that as for LU algorithmic blocking introduces $(2 + \frac{1}{r})N$ startups in the formation of the lower panel.

By coalescing the vertical broadcast of t^j with that of w^j, T becomes a row-replicated matrix without further communication startup costs. T can then be broadcast horizontally to make it fully replicated. The completion of T can then proceed without any further communication.

For the Level 3 computations, (temporarily) making the upper triangular part of V'^i zero (so that V'^i becomes a rectangular rather than a trapezoidal matrix) introduces further $O(N^2\omega)$ redundant computations, but has previously been found to be a worthwhile optimization (Choi et al. 1996), which concurs with our experience. It also greatly simplifies the efficient implementation of algorithmic blocking for the formation of the upper panel W.

For storage blocking, a vertical reduce and broadcast of W is required, resulting in a total communication volume of $2 \cdot \lg_2(P)\frac{N^2}{2}$, using tree-based communication patterns.

For algorithmic blocking, our implementation of the Level 3 parallel BLAS routine performing $W \leftarrow TW$ broadcasts W in its entirity before updating it. This means that effectively perfect load balance is achieved in the formation of W. Thus the three Level 3 computations require a multi-reduce followed by two multi-broadcasts on W; however, as these can be implemented using ring-shifts (Mitra et al. 1995, Strazdins 1998*c*)), this results in a smaller communication volume cost of $\approx 3 \cdot \frac{N^2}{2}$.

2.4 BACKSOLVE COMPUTATION

To solve the linear system, we are required to apply $X \leftarrow A^{-1}X$ to some solution vector (or set of vectors) X.

In the case of LU, a fairly standard optimization (Brent 1992) is to perform the factorization on the augmented matrix $(A|X)$; in this case, the backsolve stage need only perform $X \leftarrow U^{-1}X$. Note that standard LU factorization routines can operate on non-square matrices.

This can similarly be applied to QR, although here the gains are greater since any blocked algorithm applying $Q^{-1}X$ would have to re-construct the triangular reflectors T for each block, resulting in $N^2\omega$ extra floating point operations.

It can also be applied to LLT, by factorizing the augmented matrix $\left(\begin{array}{c|c} A & X \\ \hline X^T & x \end{array}\right)$, where x is made to be sufficiently large to ensure positive definiteness. This is possible because, for LLT, adding an extra row X^T will not affect the factorization of A.

The backsolve stage in each case thus reduces to essentially the same computation $X \leftarrow U^{-1}X$, which uses the standard parallel BLAS triangular matrix solve routine. This is implemented in an equivalent fashion to that of ScaLAPACK, in that communications (broadcasts and reductions) are blocked by a factor of r, resulting in $(\lg_2 P + \lg_2 Q)\frac{N}{r}$ startups. However, as for the symmetric rank-k update routine used in LLT, here also computational blocking occurs with a machine-dependent blocking factor that is independent of r, to better support the small block sizes used with algorithmic blocking.

2.5 OVERALL COMPARISON

Table 1 summarizes the main performance factors in an $N \times N$ linear system solve for the two blocking methods on a moderate sized grid, using the analysis given above. For LU, a degree of parallelization of $\frac{P}{2}$ is assumed in the multiple row swaps for algorithmic blocking. It is assumed that $\omega \geq 40$ (eg. so that $O(\frac{NP}{\omega})$ startup costs have a negligible effect). For the count of FLOPs in the panel formation stages of QR, redundant computations from zero-padding V^{ri} (see Section 2.3) are included. The Level 3 imbalance is the number of extra FLOPs in the matrix-matrix multiply executed by a $\frac{1}{P}$ fraction of the cells (Greer & Henry 1997, Stanley 1997). The Table will be helpful in interpreting the results of Section 3.

	LU	LLT	QR
startup cost ($\times N$)	5 : 8.5	0.0 : 2.8	6.0 : 9.8
volume cost ($\times N^2/P$)	3 : 1.5	2.5 : 2.1	4.0 : 2.1
panel load balance	0.1 : 0.7	0.1 : 0.7	0.1 : 0.9
panel FLOPs ($\times \omega N^2$)	1	0.5	3
Level 3 imbalance ($\times rN^2$)	1	1	2
overall FLOPs ($\times N^3$)	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{4}{3}$

Table 1: Performance factors for storage : algorithmic blocking ($r = 4$) for $P = Q = 8$

3 PERFORMANCE

This section describes performances results on the ANU Beowulf Cluster. Firstly serial performance is discussed, which is used to determine the optimal blocking factors. Then a

ω	20	30	40	50	60	70	80
MFLOPS	294	340	377	367	375	388	400

Table 2: Large square matrix multiply performance, with a blocking factor of ω applied to the inner dimension

comparison of the performance of algorithmic and storage blocking algorithms on a moderate-sized cluster configuration is given.

3.1 THE ANU BEOWULF CLUSTER

The ANU Beowulf cluster, named *Bunyip*, consists of 96 dual Pentium III nodes, rated at 550 MHz. Each CPU in a node has a non-shared 256 KB direct-mapped second-level cache.

The 96 nodes are arranged in four groups of 24, connected in a tetrahedral fashion by four 48-port switches. Communication speed is limited by the nodes' 100 Mb/s Network Interface Cards.

The Bunyip has been designed with price/performance in mind; indeed it won a Gordon-Bell award for this category in Supercomputing 2000 (Aberdeen, Baxter & Edwards 2000).

Communication occurs through LAM MPI, which sends messages via the TCP/IP protocol. Benchmark tests for point-to-point messages have shown a communication startup cost of of $\alpha = 44\mu$s, and a sustained communication bandwidth of 6.5 MB/s.

3.2 SERIAL PERFORMANCE

ATLAS Pentium BLAS (Whaley & Dongarra 1998) is used by our algorithms for cell computation.

For this BLAS, Table 2 indicates the performance of double precision large square matrix-matrix multiply on a Bunyip node.

Performance did not improve significantly for larger ω; the peaks then at $\omega = 40$ and $\omega = 80$ indicate two candidates for optimal blocking factors for linear system solution.

Level-2 computations speed for this BLAS was found to be ≈ 100 MFLOPS for rank-1 updates and 150–200 MFLOPS for matrix-vector multiply.

At $N = 3000$, the serial linear system solve performance using LU, LLT and QR factorizations was 320, 380 and 330 MFLOPS, respectively, using a blocking factor of $\omega = 80$. Comparing this with Table 2, we see that the Level 2 computations in the panel formation of LU and QR degrades even large computations to a significant extent.

3.3 PARALLEL PERFORMANCE

For the LU and QR tests, we use all 48 CPUs in a group using a logical 6×8 grid, which we found to be the optimal grid shape for both algorithmic and storage blocking. LAM MPI was set up here so that the CPUs on the same node would be on the same column of the grid. For

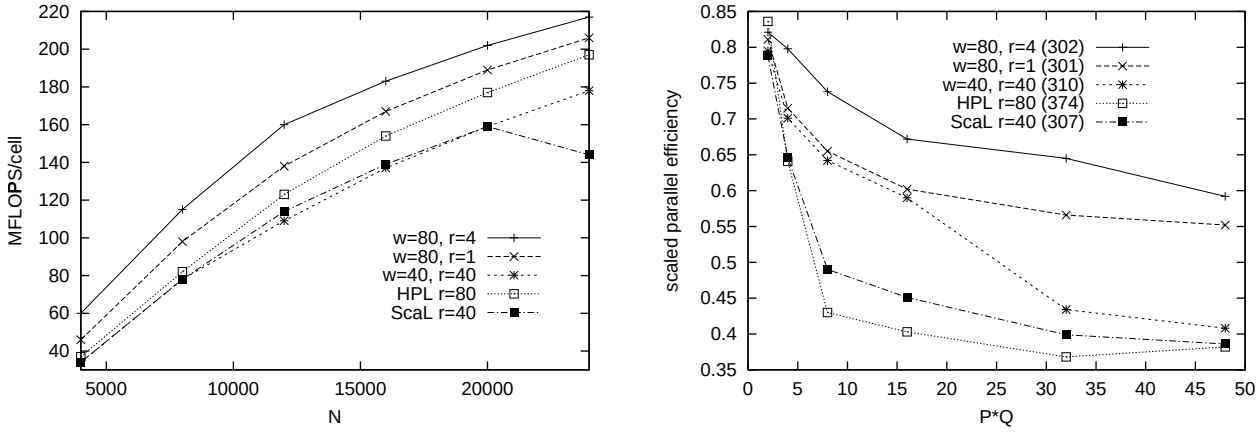
LLT, a 6×6 grid was used, as here a square grid minimizes communication costs for symmetric computations (Choi et al. 1996, Stanley 1997, Strazdins 1999).

Tests on a logical 4×4 grid indicated that running two such processes on each dual SMP node was $\approx 10\%$ slower than using twice the number of nodes, and only running one such process per node. This applied equally to algorithmic and storage blocking.

For all tests, there were some compute-bound jobs run by another user running on all nodes (such a situation often occurs on a cluster!); as these jobs were set at low process priority, they only reduced parallel performance by less than 5%. Even under these conditions, its was found that all performance results were highly reproducible.

Tests were run on blocking factors $\omega = 40, 60, 80$ for storage block sizes of 1, 4 and ω. For algorithmic blocking, $\omega = 80$ was always faster than the others (but only by at most 5%, at the upper end of the range). Except for LLT, $\omega = 40$ was similarly always faster for storage blocking, by a similar margin.

Comparisons were done with HPL, a recently released, state-of the art portable parallel LINPACK benchmark (Petitet et al. 2000). HPL is based on storage blocking[1] and uses some techniques more advanced than ours, such as using binary-exchange communication patterns. This can be argued to require only $\lg_2(P)N$ startups for the vertical communications within L^i, assuming a contention free network (which applies to the Bunyip's switches). It also uses a combined swap and broadcast of U^i, which in the best case can reduce the volume cost for LU in Table 1 from 3.5 to 2 (their implementation also incorporates memory pattern optimizations). Most importantly, it uses a recursive algorithm for factorizing L^i, effectively making this a level-3 computation, thereby reducing the impact of load imbalance. ScaLAPACK version 1.6, known to have an efficient implementation of storage blocking, was also used for comparison.



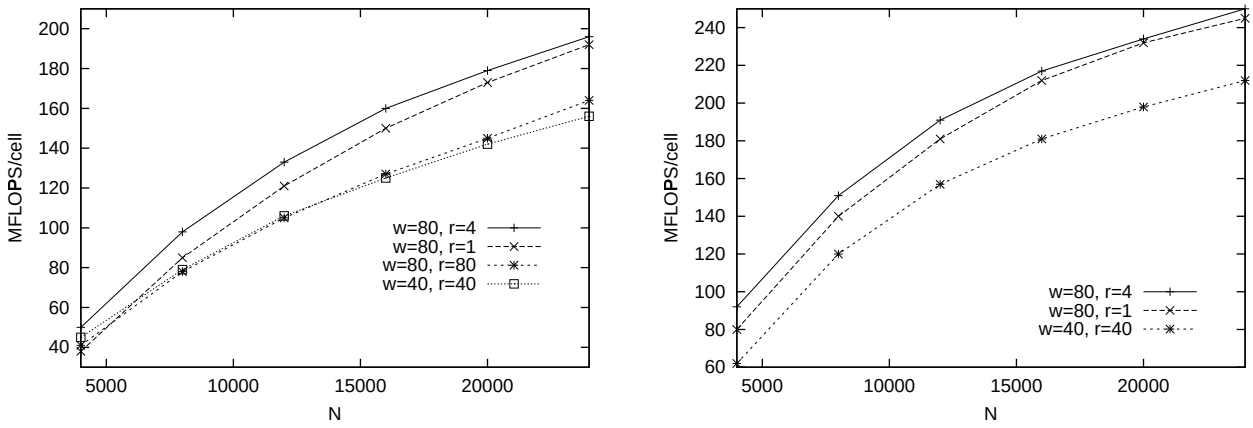
(a) Speed/cell on a 6×8 grid

(b) $||$ efficiency at $\frac{N}{\sqrt{PQ}} = 2000$

Figure 3: Performance of LU-based $N \times N$ system solve

Figure 3(a) gives the parallel speed for the LU-based solve. Here, algorithmic blocking with $r = 4$ out-performed storage blocking by 76–22%, the greatest difference being in the lower end of the range. With $r = 1$, the performance was slightly less. Our implementation of

[1] It can also use pipelining with lookahead; however, on the Bunyip, this did not yield large performance gains.



(a) LLT-based, 6×6 grid

(b) QR-based, 6×8 grid

Figure 4: Speed in MFLOPS/cell of parallel $N \times N$ system solve

storage blocking was similar to ScaLAPACK's, except that ScaLAPACK lost performance for large matrices; probably due to poor memory access patterns on the row swaps. HPL without recursive panel factorization performed only marginally better than our storage blocking implementation. However, with recursion, it gained significant improvements; this also enabled $r = 80$ to be the optimal blocking factor.

Figure 3(b) gives the scaled parallel efficiency of these LU implementations at $\frac{N}{\sqrt{PQ}} = 2000$, for $P \times Q$ grids of aspect ratio between 0.5 and 1. The scaled matrix size was chosen to give maximum performance on a 1×1 grid (absolute speed in MFLOPs for this grid is given in parentheses). It should be noted here that HPL has a much higher speed here, which somewhat prejudices its scaled efficiency rating. This graph clearly shows the full advantage of algorithmic blocking requires $PQ \geq 8$ on this platform. It also shows that for the largest performance drop-off occurs for $PQ > 2$, ie. when more than one dual SMP node is being used.

Figure 4(a) gives the results for LLT-based solve. Here, for storage blocking, $r = \omega = 80$ was slightly faster at the upper half of the range, due to the fact that the panel formation is 'Level 3' and hence faster than for LU, and so results in less of a load imbalance penalty. For algorithmic blocking, fully-replicating T^i as described in Section 2.2, was found to be faster, although only significantly for $r = 1$ and $N \leq 8000$. At $r = 4$, algorithmic blocking was only 10% faster at $N = 4000$, reflecting the impact of the extra startup costs. However, the performance differences for the rest of the range were 25–20%, somewhat surprisingly being comparable to LU or QR (cf. Table 1). ScaLAPACK's performance at $r = 40$ was essentially identical to our storage blocking implementation, except for $N \geq 20000$, where it was about 5% slower.

Figure 4(b) gives the results for QR-based solve. In contrast to LLT, the largest gain is at $N = 4000$, being 46%, due to the fact that for this computation, startup costs account for a much smaller fraction of the overall performance, and the advantage of load balance for algorithmic blocking is more significant. Elsewhere, algorithmic blocking at $r = 4$ similarly achieved a 25–18% performance gain, although here, the performance difference between $\omega = 40$ and $\omega = 80$ was slightly smaller, slightly negative on the lowest quarter of the range, increasing to $\approx 5\%$ at $N = 24000$. As ScaLAPACK only provides an efficient QR

factorization (and not a factor-solve), it is difficult to perform a direct comparison. Based on factorization times only, ScaLAPACK's QR was between 0–5% slower over this range compared with our storage blocking implementation.

It should be noted that the backsolve computation is indeed faster for large block sizes (ie. as used in storage blocking), by a factor of ≈ 2 for small to moderate N, due to the $O(\frac{N}{r})$ blocked communications. However, this had little effect on the overall performance difference between algorithmic and storage blocking.

The corresponding tests were also performed on a 4×4 grid. The trends were very similar, except that the advantage of algorithmic blocking over storage blocking was $\approx 5\%$ less in the mid-range. This is largely due to the fact that load imbalance effects are reduced on a smaller grid.

On other platforms, we have found similar results. On a 64 cell Intel Paragon, with our LU and QR implementation out-performing that of ScaLAPACK 1.6 by -1% – 5% over the range $1000 \leq N \leq 12000$ (Strazdins 1998*a*) [2]. Similarly, our implementations (including LLT) ran -3% – 10% faster on a 8 or 9 cell AP3000 (Mackerras & Strazdins 1999), over the range $1500 \leq N \leq 9000$. As for the Bunyip, the same serial BLAS and underlying communication libraries were used. It is relevant to also note that algorithmic blocking on both of these platforms showed similar performance improvements (Strazdins 1998*a*, Mackerras & Strazdins 1999).

4 CONCLUSIONS

Algorithmic blocking, due to its perceived communication overheads, was not an obvious candidate for dense linear systems solution the cluster model. However, with some refinement of the algorithms, it generally achieved gains of 15–30% on a Beowulf cluster over the standard technique, storage blocking.

This was due to several reasons. Firstly, although it has higher communication startup costs, the bulk of these can be eliminated by techniques such as replication of the triangular factor matrices (for LLT and QR), parallelization of multiple row swaps (in LU), caching of panels and pipelined communication within panel formation (in both vertical and horizontal panels). This latter effect can be magnified by choosing a small storage block size, eg. $r = 4$.

Secondly, the second, third and fourth of these techniques also serve to make the effective communication volume costs of algorithmic blocking *actually less*; on a COTS cluster network, such as on the ANU Beowulf, these costs more quickly dominate those of communication startups, as matrix size increases.

Thirdly, because load balance effects in panel formation become increasingly important with increasing CPU speed, as the difference between the speeds of panel formation speed and matrix-multiply correspondingly widens. With recursive panel factorization techniques, such as used by HPL, this is becoming less important. However, in the parallel context, these techniques, and some of the others used in HPL, require a large implementation effort justifiable for a 'prestigious' computation such as the parallel LINPACK benchmark. They

[2] Our LLT was slower, but this was due to Paragon-specific computational speed effects in the DBLAS symmetric rank-k update routine, not to any difference in communication performance.

would however be difficult to sustain across a comprehensive parallel linear algebra library. For example, a parallel recursive QR panel factorization algorithm would be much more complex than for LU.

Unlike in most previous work relating to algorithmic blocking, we have shown that the choice of a small storage block size gives the optimal tradeoff in load balance and communication costs. In this, we have also introduced some new optimizations which improve to a modest extent storage blocking and indeed serial performance of these computations.

Work which we intend to do in the near future includes adding computational blocking to the factorization of L^i for LU. This should achieve 90% of the benefits of recursive factorization, but will be simple enough to incorporate into algorithmic blocking. Also, coalescing messages in the multiple row swaps can be used to reduce startup costs.

However, the parallel efficiency on a moderate sized cluster of these computations is still low, relative to that on distributed memory machines with vendor-supplied networks, being only 60–65% for computations taking more than a half hour of elapsed time. This situation can only be rectified when faster networks become more affordable.

In general, parallelization techniques of minimum communication startup cost are not necessarily superior on a modern cluster. Alternate techniques offering better load balance, and potentially reduced communication volume costs, should be carefully considered. A particularly important issue on clusters is whether, and to what extent, can pipelined communication be utilized.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their suggestions for improving the manuscript.

REFERENCES

ABERDEEN, D., BAXTER, J. & EDWARDS, R. (2000): 98c/MFLOP Ultra-Large Scale Neural Network Training on a PIII Cluster: *in* 'Proceedings of SuperComputing 2000'.

BLACKFORD, L., CHOI, J., CLEARY, A., D'AZEVEDO, E., DEMMEL, J., DHILLON, J., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D. & WHALEY, R. (1997): *ScaLAPACK User's Guide*: SIAM Press: Philadelphia.

BRENT, R. (1992): The LINPACK Benchmark on the Fujitsu AP 1000: *in* 'Frontiers '92: Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation': Virginia.

CHOI, J., DONGARRA, J., OSTROUCHOV, S., PETITET, A., WALKER, D. & WHALEY, R. (1996): The Design and Implementation of the ScaLAPACK LU, QR and Cholesky Factorization Routines: *Scientific Programming* **5**, 173–184.

DONGARRA, J. & DONIGAN, T. (1997): Message-Passing Performance of Various Computers: *Concurrency* **9**(10), 915–926.

GREER, B. & HENRY, G. (1997): High Performance Software on Intel Pentium Pro Processors, or Micro-Ops to TeraFLOPS: *in* 'Supercomputing 97': ACM SIGARCH - IEEE Computer Society Press.

HENDRICKSON, B. & WOMBLE, D. (1994): The Torus-Wrap Mapping for Dense Matrix Calculations on Massively Parallel Computers: *SIAM J. Sci. Stat. Comput.* **15**(5), 1201–1226.

MACKERRAS, P. & STRAZDINS, P.E. (1999): 'ANU/Fujitsu CAP Research Program Report': http://cap.anu.edu.au/cap/reports.

MITRA, P., PAYNE, D., SHULER, L., van de GEIJIN, R. & WATTS, J. (1995): Fast Collective Communication Libraries, Please: *in* 'Proceedings of the Intel Supercomputing Users' Group'.

PETITET, A. (1996): Algorithmic Redistribution Methods for Block Cyclic Decompositions: PhD thesis: University of Tennessee: Knoxville. xv+193p.

PETITET, A., WHALEY, R., DONGARRA, J. & CLEARY, A. (2000): 'HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers': http://www.netlib.org/benchmark/hpl/.

STANLEY, K. (1997): Execution Time of Symmetric Eigensolvers: PhD thesis: University of California: Berkeley. viii+184p.

STRAZDINS, P.E. (1995): Matrix Factorization using Distributed Panels on the Fujitsu AP1000.: *in* 'IEEE First International Conference on Algorithms And Architectures for Parallel Processing (ICA3PP-95)': Brisbane: pp. 263–73.

STRAZDINS, P.E. (1998*a*): 'Enhanced Parallel Matrix Factorization Routines': seminar given at the Department of Computer Science, University of Tennessee Knoxville.

STRAZDINS, P.E. (1998*b*): Lookahead and Algorithmic Blocking Techniques Compared for Parallel Matrix Factorization: *in* '10th International Conference on Parallel and Distributed Computing and Systems': IASTED: Las Vegas: pp. 291–297.

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

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

WHALEY, R. (1998): 'University of Tennessee Knoxville, private communications'.

WHALEY, R. & DONGARRA, J. (1998): Automatically Tuned Linear Algebra Software (ATLAS): *in* 'Proceedings of SuperComputing 98'.