Transporting Distributed BLAS to the Fujitsu AP3000 and VPP-300

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

Department of Computer Science, Australian National University

Abstract

The DBLAS Distributed BLAS Library is a portable version of parallel BLAS that has been highly tuned for the Fujitsu AP1000 and AP+. In this paper, we describe performance enhancements made for two very different high performance distributed memory platforms, the Fujitsu AP3000 and the Fujitsu VPP-300.

Even with the provision of highly tuned (vendor-supplied) serial BLAS implementations, attention must be given to cell computation speed issues, since serial BLAS does not supply a local matrix transpose routine (which is needed in many places), nor does it supply routines to adequately handle the triangular matrices which arise in the parallel context. We will describe the differing principles used on the UltraSPARC and VPP-300 nodes to optimise memory access patterns for the local matrix transpose operation and the large matrix multiply. The former uses partitioning methods which can yield a factor of 3-4 improvement of naive methods. The latter simultaneously optimizes usage of two levels of cache and the TLB, and out-performs the BLAS from the Sun Performance Library 1.2 by at least 15% on an 170 MHz UltraSPARC I. We will also give comparisons between the other functions in our version of UltraSPARC-tuned BLAS and the Performance Library 1.2.

Unlike the AP1000 and AP+, the AP3000 and VPP-300 must simulate row or column broadcasts using point-to-point messages. Also, their communication latencies to floating point speeds are much higher, and stride communication is not available. We describe and evaluate methods used in the DBLAS spread and reduce communication primitives to minimize communication costs for these machines. These include the use of pipelined broadcasts, and ring-shift methods for spread and reduce operations with multiple sources, both of which are significantly superior to binary tree based methods.

1 Introduction

The DBLAS Distributed BLAS Library [7, 9, 11] is a portable version of parallel BLAS that has been highly tuned for the Fujitsu AP1000 and AP+. It has been used to implement very efficient parallel matrix factorization applications using various techniques [12]. The AP1000 and AP+ [2] have special cell and communication network characteristics that the DBLAS implementation exploits. In particular, the AP machines can perform an asynchronous row or column broadcast for the cost of a normal message send, which is extremely useful in dense linear algebra computations [12]. These also have stride communication capabilities [9].

While BLAS computation and BLACS [1] communication interfaces have been developed for the DBLAS, and the DBLAS has also a PBLAS Parallel BLAS interface to make it callable from ScaLAPACK [1], it can in principle be ported to other distributed memory platforms without excessive effort. However, in order to *transport* it to diverse architectures, that is to ensure it also yields good performance on them, the design and implementation of the DBLAS had to be extended and generalized in a non-trivial fashion.

The Fujitsu AP3000 [3] and VPP-300 [13] are two distributed memory multicomputers of very different cell architectures, the AP3000 being comprised of RISC scalar processors (Ultra-

SPARC) with a deep memory hierarchy (having a 16KB top-level data cache and a 1MB 2nd level cache, both direct-mapped), and the VPP being comprised of state-of-the-art vector processors having high bandwidth interleaved memory (ie. a 'flat' memory hierarchy). These both have 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.

To make a parallel library such as the DBLAS transportable to a diverse range of architectures, capable of exploiting the features of each, requires a set of 'tuning parameters' [9] that can encompass all these situations. Such parameters include things such as the cache size and various optimal blocking sizes for local and global computations [9]. This idea will be extended further in this paper.

In this paper, we describe the performance enhancements made to the DBLAS in order to transport it to the AP3000 and VPP-300. These enhancements should make it easy to transport to most other contemporary distributed memory platforms. This paper is organized as follows. Section 2 discusses cell computation issues, including that for memory access patterns and the development of UltraSPARC BLAS (Section 2.3). Communication issues are discussed in Section 3, including pipelined communication and efficient spread or multicast implementation. Conclusions are given in Section 4.

2 Cell Computation Concerns

Even if an efficient vendor-supplied BLAS is available, cell computation concerns cannot be ignored in a parallel BLAS implementation. This is because serial BLAS cannot directly handle the local portions of triangular matrices for block cyclic distributions over a general $P \times Q$ grid (in which case the diagonals of these matrices will have an average slope of $\frac{Q}{P}$). This situation is most critical in the parallel symmetric rank-k and rank-2k updates, which are the dominant sub-computations in LLT factorization and symmetric tridiagonal reduction. In this case a parallel BLAS implementation will have to perform partitioning of the local triangular matrices into (rectangular) strips of optimum width.

Furthermore, in many situations, the need for local matrix transpose operations arise, which are not provided for in the serial BLAS. In the DBLAS, local transpose may be required in the following situations:

- for the spread and reduce operation (Section 3.2), where it is optimal to have unit stride along the length of the panel. In the case of reduce, adding a transposed matrix to a matrix is required.
- for the matrix multiply, to get the optimal storage for the matrix input operands (see Section 2.3.1).
- for packing the panels of a matrix, in the context of matrix factorizations. Here, having unit stride along the panel length will be optimal. Thus, in the case of column major matrices, it is worthwhile to pack the upper panel of the matrix, apply the updates on the packed panel, and unpack the panel afterwards. Even on the AP+, which has a relatively low TLB miss penalty, this technique yields 5–10% increase in performance for moderate to large matrices, especially for LU factorization where this saves strided memory copying in the communications for the row swaps.

In the following section, the issue of memory access patterns, and efficient local matrix transpose will be explored in more detail.

2.1 Memory Access Pattern Issues

Achieving good memory access patterns is an increasingly important concern for high performance processors, as the cost of a memory access, in terms of machine cycles, has been steadily increasing with improving technology. In general, achieving unit memory stride as much as possible (especially on the innermost loops) is a goal common in most situations.

Consider matrix multiplication. On scalar-based processors with relatively low memory bandwidth, such as the UltraSPARC, it is optimal have an inner-product based matrix multiply. On vector processors, with a higher memory bandwidth, the situation is the reverse, since a gaxpy-based matrix multiply is optimal as the gaxpy is a proper vector operation.

As mentioned above, transporting the DBLAS library to cover both situations presents some interesting challenges. A boolean 'tuning parameter' can be used to indicate whether inner-product based matrix multiply are optimal. For a large matrix multiplies, including those required in the DBLAS the symmetric rank-k and rank-2k update routine, the input operands are packed to have the preferred storage scheme. When partitioning is applied to form 'strips' along the result matrix, the computation is transposed to ensure unit stride applies along the length of these strips.

An architecture using interleaved memory of 2^b memory banks can be identified by having a tuning parameter of value b. Hence $b > 0$ indicates interleaved memory is present. Here, when iterating along a vector of stride s, the cost of the case $s = 1$ can be deemed no greater than that of the case s is odd, which is in turn less than the case where s is even. If $b = 0$, an adequate heuristic is to consider the cost to increase monotonically with s. These heuristics can be used to decide which is the optimal matrix dimension for iteration, as is required in local matrix transposition. In which case, a third tuning parameter being the number of entries in the TLB, $|\mathrm{TLB}|$, will also be required.

2.1.1 Local Matrix Transposition

In this section, we will give a simple algorithm for the local matrix transposition $B \leftarrow A^T$, where B is $M \times N$ and MN is larger than $|\mathrm{TLB}|$ $|\mathrm{Page}|$, where $|\mathrm{Page}|$ is the page size. This algorithm minimizes TLB misses, and can easily be extended to the addition of a transposed matrix $B \leftarrow B + A^T$. Its performance on the UltraSPARC U170 will then be evaluated.

The algorithm first chooses the matrix dimension of minimum cost for the innermost iteration (provided this is sufficiently large, ie. exceeds $|\mathrm{TLB}|/2$), as explained in the previous section. Assuming this dimension to be N, it is then partitioned into blocks of size $N' = |\mathrm{TLB}|/2$. $M \times N'$ slices of B are then formed, with the innermost iteration being across the rows of B. It can be easily seen that will be at least M (N') accesses of elements of B (A) per TLB miss.

For $M = N$ and the stride of B being N, the algorithm sustained 85–100 MB/s for $2048 \leq N \leq 4000$ on an UltraSPARC U170 (cf. the contiguous memory copy speed for such data sizes is 120 MB/s [9]). Without performing the partitioning, the speed was 30 MB/s.

For $M \gg N = 128$ and the row stride of A being M and B being packed (this corresponds to packing a panel), the above algorithm sustained 85–95 MB/s for $2048 \leq N \leq 4000$, as compared with an unoptimized transpose, which had a speed of 7–23 MB/s.

2.2 Interfaces for External BLAS

If a parallel BLAS implementation is relying on an external (eg. vendor supplied) BLAS library for computation performance, it ideally needs to be able to call that library so as to get the best performance possible. This can include being insensitive to any performance ideosyncrasies that are likely to occur with such libraries.

In practice, this is difficult if not impossible to fully achieve. However, there are a few general situations that should be considered:

- avoid extra software overheads by not using level 3 routines for level 2 computations.

  While in the DBLAS, it was convenient to regard level 2 computations as degenerate cases of level 3 computations, in order to get simplified 'high level driver' routines (eg. the parallel rank-1 update and rank-k update share the same code) [11], significant software overheads can be saved if level 2 BLAS can be called wherever it is likely to occur (eg. `dger()` is called instead of `dgemm()` when $K = 1$). This is because the level 3 routines are intended to be general, 'heavy-weight' routines.

  Such optimizations are now incorporated into the DBLAS.

- decide whether it is optimal to 'prescale' matrix input operands.

  In most situations, the matrix multiply is of the form $C \leftarrow C + \alpha AB$, where α is a scaling parameter, eg. $\alpha = -1$. Before calling the

BLAS matrix multiply routine, the DBLAS implementation may have the opportunity to copy A or B (eg. one of these might be the result of a spread operation, see Section 3.2) into a workspace, in which case the scaling factor α can be applied at the copy stage.

The question is under what situation is it efficient to scale when copying, and under what situations is it not. Note that BLAS does not supply a scaled copy primitive: the effect can only be achieved sub-optimally by a copy (`dcopy()`) followed by a scale (`dscal()`).

The heuristic used in the DBLAS is to scale when forming a workspace for a spread only if more than one processor is contributing to the spread (and hence the scaled copy can be parallelized) and a tuning parameter signifies that the BLAS matrix multiply can benefit from prescaling. This is only likely to be the case for processors having inner-product based matrix multiplies.

- try to use the 'preferred' matrix storage scheme of the BLAS.

  As will be mentioned in Section 2.3.1, for the Sun Performance Library 1.2, only `dgemm("N", "N", ...)` is well optimized: transposed matrix operands do not run nearly as fast, even for large sizes, where it would be reasonable to expect the implementation to copy the matrices if the 'preferred' storage was not offered.

  This presents a very awkward situation to a DBLAS implementation to deal with in a general way.

2.3 Ultra BLAS: Design Issues and Performance Comparisons

In general, as compared with the SPARC 1 and the SuperSparc, an optimal BLAS for the UltraSPARC requires deeper and more aggressive software pipelining. In particular, (top-level) *cache lookahead* need be used to hide the miss penalty for the top-level cache, which is too small for most computations. The cache lookahead requires the load operations to be initiated at least 8 cycles before the first instruction using that value.

In the case of matrix copy, this is relatively easy to achieve with an 8-way unrolled inner loop consisting of 8 loads followed by 8 stores. In this case, matrix copy within the level 2 cache was 540 MB/s, only marginally slower than that within the level 1 cache of 625 MB/s [9]. Where computation is also required, the unrollings become more complex, and for multiply-add operations, most of the 32 UltraSPARC registers are required. We will now look at in detail the implementation of crucial functions of the UltraSPARC BLAS.

2.3.1 Large Matrix Multiplication

Consider the matrix multiply $C += AB$, where C is $N \times N$ and A is $N \times K$. For applications of interest, eg. matrix factorizations, $128 \leq N \leq 8192$ and K can be chosen for some optimal value, eg. $K = 44, 64, 88 \ldots$ In this context, the storage conventions of A and B can be chosen (eg. row-major or column-major).

To minimize cache and TLB misses on the UltraSPARC, it is important to have unit stride memory access as much as possible, in both the input matrices, (inside calls to the inner loop kernel), and the result matrix (between calls to the inner loop kernel). Note that TLB misses have been measured to have $\approx 1\mu$s overhead on an UltraSPARC U170 under Solaris 5.5. This is equivalent to ≈ 300 floating point operations at peak speed.

The UltraSPARC matrix multiply inner loop kernel is based on codes developed previously by Fujitsu Laboratories [5], which updates a 4×4 sub-matrix of C. It however has been modified to ensure unit stride access of *both* A and B. This means that A must be row-major and B must be column-major. The inner loop is unrolled two times and *software pipelined* to achieve a 9–10 cycle load-multiply separation, and a 3 cycle multiply-add separation. It can achieve 300–320 MFLOPs for small matrix multiplies with $N = 32, 64 \geq K \leq 128$, ie fitting within the UltraSPARC's 16KB level 1 cache. Note that the theoretical peak speed of an 170 MHz UltraSPARC is 330 MFLOPs.

For large matrices, the partitioning method is to divide B into block columns of width $N' = 256$. This value was chosen so that for $K \leq 256$,

$N \backslash K$	44	64	88	128	176
512	257	267	279	255	283
1024	257	258	278	256	279
1536	251	259	274	255	279
2048	251	252	273	249	275
2560	252	245	268	250	281
3072	247	254	275	248	276
3584	245	244	260	258	278
4096	246	248	267	225	243
5120	249	254	278	256	262
6144	242	240	268	225	244

Table 1: Performance of the matrix multiply $C+ = AB$ with A being $N \times K$, on an UltraSPARC U170.

no more than half of the UltraSPARC 64 TLB entries would be required for this block column (with a page size of 8 KB). It is rather curious that on the UltraSPARC, the TLB, rather than the second level cache, dictates the partitioning sizes. Thus a (contiguous) $K \times N'$ portion of B remains in cache, while different rows of A and C are applied to it [10].

The inner loop kernel is applied across 4 rows of C at a time; assuming C is row-major, this ensures access patterns for each matrix are contiguous. In the worst case, there could be a TLB miss for the update of each segment (of width N') of each row of C for the above algorithm. However, as there will be $2KN'$ floating point operations applied for each segment, the effect of the TLB miss should be small for appropriate values of N' and K.

The performance of this algorithm is given in Table 1. Performance peaks at $K = 88$ and $K = 176$, making these appropriate blocking factors for applications such as matrix factorization. As a comparison, the Sun Performance Library 1.2 `dgemm()` sustains between 210–235 MFLOPs for comparable sizes, and good performance is only obtained for column-major A, B and C.

On the AP1000 and AP+ cells, the above algorithm is generally slightly slower (eg. 3–5%) than the partitioning methods described in [7]. This is because those methods optimize cache misses slightly better; however as these do not take TLB misses into account, their performance is relatively poor on the UltraSPARC.

2.3.2 Level 1 and 2 BLAS Performance

Matrix-vector multiply was implemented with cache lookahead, using an 8-way outer loop unrolling and a 2-way inner loop unrolling. Computations fitting within the top-level cache sustained nearly 250 MFLOPs, declining to 150–190 MFLOPs when exceeding this cache size. This indicates lookahead was not as effective here as for matrix-matrix multiply.

For vector norm, lookahead was effective, sustaining 250 MFLOPs even for vectors of length $N = 16384$.

Our implementation for these performed as well or better than the corresponding BLAS functions in the Sun Performance Library 1.2. A comparison of the other level 1 and 2 BLAS functions will be given at the Workshop.

3 Communication Enhancements for the DBLAS

These enhancements fall into two categories: firstly in order to reduce communication startup costs and to secondly to find efficient algorithms for broadcast and spread (multicast) operations. The latter are used where fast row and column broadcasts implemented by wormhole routing are not available.

3.1 Pipelined Communication

Pipelined communication can be used in some circumstances to enhance the speed of collective operations, for example broadcast and global summation operations [4]. It is used extensively in ScaLAPACK [1] to perform the horizontal communications in matrix factorizations.

Across a grid of width Q, a series of $O(Q)$ broadcasts (of approximately equal volume) can be pipelined if *all* horizontal communication can be made to flow rightwards. In this case, the origin cell of the broadcast sends to to its neighbor to its immediate right, which in turns passes it to its right and so on until all cells have received the message. This is illustrated in Figure 1. As the size of the 'pipeline bubble' is of the same size as the messages, the cost of each broadcast, relative to a point-to-point message, is $\kappa_{bc} = 2$ (this applies to both latency and volume costs).

0	$\rightarrow_0$			
1		$\rightarrow_0$		
2		$\rightarrow_1$	$\rightarrow_0$	
3			$\rightarrow_1$	$\rightarrow_0$
4			$\rightarrow_2$	$\rightarrow_1$
5	$\rightarrow_1$			$\rightarrow_2$
6	$\rightarrow_2$			$\rightarrow_3$
7	$\rightarrow_3$	$\rightarrow_2$		
8	$\rightarrow_4$	$\rightarrow_3$		
9		$\rightarrow_4$	$\rightarrow_3$	
10		$\rightarrow_5$	$\rightarrow_4$	
11			$\rightarrow_5$	$\rightarrow_4$
12			$\rightarrow_6$	$\rightarrow_5$
13	$\rightarrow_5$			$\rightarrow_6$
14	$\rightarrow_6$			$\rightarrow_7$
15	$\rightarrow_7$	$\rightarrow_6$		

Figure 1: Execution of pipelined broadcasts across a grid of 4 processors

This is superior to the tree broadcast, which has $\kappa_{bc} = \lg_2 Q$.

Pipelined communication require send operations complete regardless of the state of the receiving cell, ie. the so-called *fire-and-forget* semantics must be supported by the underlying messaging system. This means that the AP3000 *ring buffers* must be used for these messages, and this make it difficult to use MPI implementations, which typically do not support the fire-and-forget semantics.

When combined with the technique of *lookahead* for matrix factorizations, the 'pipeline bubble' can in many circumstances be hidden to give $\kappa_{bc} = 1$, enhancing the communication speed still further [12]. For machines with higher communication startup to computation ratios, such as the AP3000 and VPP-300, hybrid lookahead and algorithmic blocking methods, which can be supported by the DBLAS, are likely to offer the optimum combination of high load balance and reduced communication overheads [12].

3.2 Efficient Implementation of Spread Operations

As described in [9], all DBLAS communications are expressed in terms of two powerful complementary operations, called *spread* (or multicast) and *reduce*. Figure 2 shows a spread of the K columns of a global block-cyclic $M \times K$ matrix A across a row of a $P \times Q$ processor grid, storing the result in a row-replicated workspace matrix `AsM`. Local cell portions of `AsM` are $M/P \times K$, with the local column order corresponding to the global.

The *reduce* operation can be thought of as the reverse of the *spread* operation, with the corresponding portions of `AsM` being summed back into A.

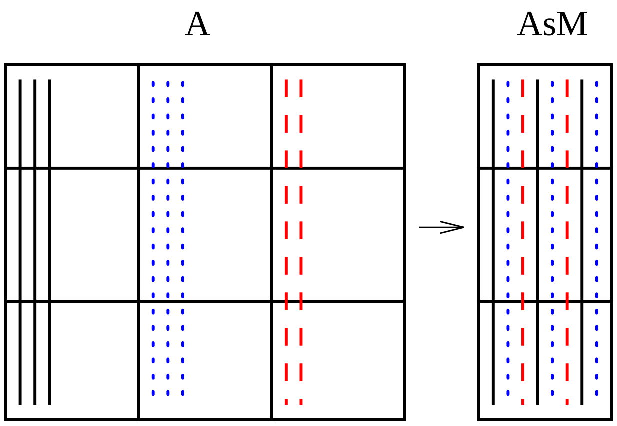

Figure 2: Row 'spread' of the distributed matrix A (having 8 block columns), across the rows of a $P \times Q$ processor grid with $P = Q = 3$. The result is stored in `AsM`.

A spread is made up of a series of logical broadcasts of roughly equal size; if there are approximately Q broadcasts, as there will typically be with algorithmically blocked matrix factorizations (with a storage block size of 1), communication costs can be reduced by a factor of $(\lg_2 Q)/2$ by using pipelined communication, as explained in Section 3.1.

However, in this situation, it is possible to reduce it by a further factor of 2 by using a series of $Q-1$ ring-shift operations on messages of width $\lceil \frac{K}{Q} \rceil$ [4]. Provided $\lceil \frac{K}{Q} \rceil \approx \frac{K}{Q}$, this will yield $\kappa_{bc} \approx \frac{Q-1}{Q}$. The ring shift method will thus be even be slightly faster on the AP1000 and AP+, as compared to implementations which use Q hardware row broadcasts with $\kappa_{bc} = 1$.

The same idea has long been applied to the DBLAS reduce operation, using ring shift-and-add operations instead of ring shifts [9]. This was because there is no comparably efficient alternative possible on the AP1000 and AP+.

Finally, on the AP1000 and AP+, *stride communications* are used to form the columns of the 'spread matrices', eg. `AsM` of Figure 2, maintaining the global index order without either communication startup or memory copy overhead [9]. The AP3000 and VPP-300 do not have such a capability; however, by sending only packed messages, which are then unpacked into `AsM` after message receipt, memory copy overheads are reduced by a factor of a half.

3.3 Reducing Startups in Triangular Matrix Updates

The parallel BLAS triangular matrix updates $B \leftarrow A^{-1}B$ and $B \leftarrow AB$, where A is triangular and B is $K \times N$ provide several opportunities for reducing communication startup overheads in broadcast operations. This was never really necessary on the AP1000 or AP+, as not only were communication startup of normal messages relatively small, row broadcasts did not amplify this startup cost.

Firstly, for small K, A can be spread in a single operation without significant workspace requirements. This can also be done for B in the case of $B \leftarrow AB$. These optimizations made a 5–10% performance improvement for QR factorizations for $512 \leq N \leq 1024$ on a 64 node AP+, but would have a larger impact on machines with higher startup penalty.

Secondly, for $B \leftarrow A^{-1}B$, the rows of B must be broadcast vertically, as soon as each has been completely updated .The simplest way of implementing this, which also yields the best load balance, is to broadcast these individually (as is done in the current DBLAS implementation). Provided all processors own (part of) a row of B, these broadcasts can be pipelined to reduce communication latencies, as explained in Section 3.1. If a storage blocksize of r is used, the size of the pipeline bubble is reduced, resulting in an effective $\kappa_{bc} = \frac{r+1}{r}$, that is a total startup cost of $K\frac{r+1}{r}\alpha$, where α the the startup cost of a single message.

Latencies can be reduced further by *blocking* r rows of B, at the loss of some load balance, although such an optimization yielded no benefits on the AP1000 [8]. Blocking however can be valuable on machines with higher communication startup penalties, effectively reducing the total startup cost to $\frac{2K}{r}\alpha$. This has been implemented in the PBLAS Parallel BLAS implementation for ScaLAPACK [1].

Load balance can potentially be improved by applying *lookahead* to this computation [6], and in ideal circumstances, the total startup cost could be reduced to just $\frac{K}{r}\alpha$, as explained in Section 3.1 (see also [12]). However, in the context of matrix factorizations on the AP3000 and VPP-300, the width of B is in the range $64 \leq K \leq 128$. With such small K, and for a processor grid of height $P \geq 4$, the conditions where lookahead with $r \geq 4$ becoming fully effective are rather narrow, as explained in [12]. We leave to future work to discover whether blocked $B \leftarrow A^{-1}B$ with lookahead will yield significant performance gains on the AP3000 and VPP-300.

4 Conclusion

Efficient and transportable parallel BLAS implementations must consider cell computation issues, and most importantly optimizing memory access patterns. The DBLAS has been extended for the AP3000 and VPP-300 by the addition of a few extra tuning parameters (primarily relating to memory access), a local matrix transpose algorithm, and techniques for achieving unit memory stride for partitioned matrix multiplications.

For a portable parallel BLAS relying on a range of vendor-supplied BLAS for computation speed, it is in general more difficult to ensure good performance, especially if the vendor BLAS exhibits ideosyncratic performance behaviors. However, we are developing a version of UltraSPARC-tuned BLAS exploiting the UltraSPARC's cache lookahead and simultaneously optimizing cache and TLB performance. This BLAS is relatively compact, has consistent performance behavior and compares favorably with the Sun Performance Libary 1.2

For ensuring good communication performance on these machines, it is required to reduce communication startups by various techniques, the most important and significant being pipelined communication. Support for pipelined communication is thus very important, yielding many advantages in reduced communication costs, and if lookahead is used, can endow greater benefits still in improved load balance.

References

[1] 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.

[2] T. Horie, K. Hayashi, T. Shimizu, and H. Ishihata. Improving AP1000 Parallel Computer Performance with Message Communication. In *International Symposium of Computer Architecture (ISCA '93)*, 1993.

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

[4] P. Mitra, D. Payne, R. van de Geijn L. Shuler, and J. Watts. Fast Collective Communication Libraries, Please. In *Proceedings of the Intel Supercomputing Users' Group*, 1995.

[5] K. Nagatsuka. Matrix Multiply codes for the UltraSPARC. Source codes, Fujitsu Laboratories, Kawasaki, Japan, April 1996.

[6] A. Petitet. PBLAS (version 2.0 ALPHA) – alignment restriction free PBLAS using logical algorithmic blocking techniques. source codes, available from http://www.netlib.org/scalapack/prototype /pblasV2alpha.tar.gz, November 1997.

[7] P. E. Strazdins and R.P. Brent. Implementing BLAS Level 3 on the AP1000. In *Proceedings of the CAP Workshop '91*, pages H–1 – H–17, Australian National University, Canberra, November 1991.

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

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

[10] P.E. Strazdins. Matrix Multiply on the UltraSPARC. Report, http://cs.anu.edu.au/P̃eter.Strazdins /projects/SparcBLAS/matmult_report.txt, December 1996.

[11] 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.

[12] 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*, Las Vegas, September 1998. IASTED.

[13] N. Uchida. Hardware of the VX/VPP300/VPP700 Series of Vector-Parallel Computer Systems. *Fujitsu Scientific and Technical Journal*, 33(1):24–29, 1997.