Linear Algebra Research on the AP1000*

R. P. Brent†, A. Czezowski, M. Hegland,
P. E. Strazdins and B. B. Zhou
Australian National University
Canberra, ACT 0200

29 October 1993
(rev. 15 Dec. 1993)

Abstract

This paper gives a report on various results of the Linear Algebra Project on the Fujitsu AP1000 in 1993. These include the general implementation of Distributed BLAS Level 3 subroutines (for the scattered storage scheme). The performance and user interface issues of the implementation are discussed. Implementations of Distributed BLAS-based LU Decomposition, Cholesky Factorization and Star Product algorithms are described.

The porting of the Basic Fourier Functions, written for the Fujitsu-ANU Area-4 Project, to the AP1000, is discussed. While the parallelization of the main FFT algorithm only involves communication on a single 'transposition' step, several optimizations, including fast roots of unity calculation, are required for its efficient implementation.

Some optimizations of the Hestenes Singular Value Decomposition algorithm have been investigated, including a "BLAS Level 3"-like kernel for the main computation, and partitioning strategies. A study is made of how the optimizations affect convergence.

Finally, work on implementing QR Factorization on the AP1000 is discussed. The Householder QR method was found to be more efficient than the Givens QR method.

1 Introduction

This paper summarizes research in Linear Algebra on the Fujitsu AP1000 at the Australian National University during 1993. Section 2 reports on progress on the implementation of Distributed BLAS Level 3 on the AP1000, and Section 4 reports on progress on further optimizations of Hestenes Singular Value Decomposition on the AP1000: both of these are sub-projects that have already begun and the reader is referred to previous papers [5, 4] for background. There are also new sub-projects: the Fast Fourier Transforms (Section 3) and Orthogonal QR Factorization (Section 5). Conclusions are given in Section 6.

2 The Distributed BLAS Level 3 Library

To date there have been several proposed definitions of Distributed BLAS. Concurrent BLAS [3] involves splitting the 3 levels of the BLAS [9] further into 3 concurrency 'classes', and uses a generalization of the block-cyclic matrix distribution. A distributed matrix operation is only considered valid if the common indices of all matrices are 'compatible'. For example, the global

 Corrected version of a paper which appeared in *PCW '93: Proceedings of the Second Parallel Computing Workshop*, held at Kawasaki, Japan, Nov. 11, 1993, pp. P1-L-1 – P1-L-13.

†E-mail address: rpb@cslab.anu.edu.au rpb146 typeset using LaTeX

matrix operation $C = AB$ is valid, if $C_{i.}$ is on the same (row of) processors as $A_{i.}$ (and similarly for $C_{.j}$ and $B_{.j}$; $A_{.k}$ and $B_{k.}$). Concurrent BLAS uses column strides and well as row strides for matrices.

ScaLAPACK [8] takes this last point further, although it only allows square block-cyclic matrix distribution. A distributed (sub-) matrix is expressed as an object of many components, including the matrix dimensions and row/column strides, block strides and offsets, as well as information on matrix distribution. This requires an Object-Oriented Programming approach to be manageable.

Our approach to Distributed BLAS, called the DBLAS, is to use the scattered matrix distribution (corresponding to a block size of 1 in the square block-cyclic distribution) [13, 1]. This distribution is generally optimal on architectures such as the AP1000 with relatively low communication overheads, as it is best for load balancing. A global matrix (A) can then be simply specified by its dimensions (eg. `M, N`), its row stride (`LdA`) and the address (`A`) of the leading element of the local sub-matrix in each processor, and the id (`AI0, AJ0`) of the processor containing the leading element of the global matrix.

In a global matrix operation, compatibility in the sense of [3] is not required: the leading elements of the global matrices may be on different cells, and indeed on a non-square processor array, this strong definition of compatibility is impossible for the scattered distribution. Instead, the DBLAS routines themselves are responsible for the 'on-the-fly' alignment of all data, including matrix transposition, during the computation.

The philosophy of DBLAS is to provide a similar as possible interface to the BLAS, so that algorithms written in BLAS Level 3 for serial processors may be easily transformed to the equivalent distributed algorithms. Generalizations of matrix storage and representation to the current BLAS should then be avoided. As shown in Section 2.3, this transformation simply involves prefixing the BLAS procedure names with a '`D`' and appending the ids of the processors containing the leading elements of each matrix to the parameter lists. If some arguments are sub-matrices of larger matrices, the offsets of the sub-matrix may also have to be rescaled.

Thus, when all AP1000 cells simultaneously call a DBLAS procedure (in SPMD mode), the effect is the same as the corresponding BLAS Level 3 call if the global matrices were on a serial processor. Currently the DBLAS procedures assume scattered storage for all matrices (row-major across and within cells), with indices starting from 0, and both single and double precision are supported. In DBLAS procedures, matrix size, shape, side, transposition and scaling parameters (eg. `M`, `N`, `UPPER`, `LEFT`, `TRANS` and `ALPHA`) and the cells containing the leading element (eg. `AI0, AJ0`) refer to the global matrix (A) and must be consistent across all AP1000 cells. The matrix parameters (eg. `A`) and the row strides refer to the local sub-matrix and need not be consistent across cells.

2.1 Implementation

The matrix multiply routines `D_GEMM`, `D_SYMM`, `D_SYRK` and `D_SYR2K` are all implemented by calling a matrix multiply 'super-procedure' called `ParallelBrdMult`. Here the result matrix (C) remains on the cells, and sections of the input matrices (A, B) are broadcast. This is an optimized version of 'Method A' in Section 3.3.2 of [13], except on an $N_y \times N_x$ AP1000, only LCM(N_y, N_x) broadcasts are required, where LCM is the lowest common multiple function.

If matrix alignment is required, eg. `AI0` $\neq$ `CI0`, this is done by explicit rotation of each section of A before it is broadcast. However, if `AJ0` $\neq$ `BI0`, explicit rotation is not required.

If matrix transposition is also required, it is performed before each broadcast, with no cell sending or receiving more than LCM(N_y, N_x) messages. Thus communication startup is minimized. Formation of the messages involves striding the cell matrices with strides of GCD(N_y, N_x), where GCD is the greatest common divisor function. At present, to give a standard implementation, communication is done using the AP1000's `xy` communication library, with explicit packing and unpacking of data. Communication overhead could be reduced significantly by using the direct message receive and stride message transfer routines, recently developed at ANU [1] for the AP1000. The performance of transposition could also be improved by doing the transpositions for N_y or N_x broadcasts at a time, as this will ensure that the AP1000's T-net is fully utilized.

In `D_SYMM`, the input matrix A is symmetric and is stored in triangular form. `ParallelBrdMult` performs a 'half-transpose' operation to form one half of the section of A it is currently broadcasting. In `D_SYRK` and `D_SYR2K`, the output matrix (C) is also triangular; this requires the cells to perform local matrix multiply with the result matrix being triangular of slope N_y/N_x (the cell-level BLAS does not provide such functionality).

The triangular matrix update routines `D_TRMM` and `D_TRSM` are implemented by the 'super-procedure' `TR_M`; this is possible since the computation for one procedure is essentially that of the other, but in reverse order. Here matrices have to be broadcasted one row at a time. Transposition is handled in a similar way as that of `ParallelBrdMult`. Blocking has been implemented for improved performance [13].

Both `ParallelBrdMult` and `TR_M` use workspace of the order of one AP1000 cache size to store the broadcasted rows/columns of A and B. This reduces cache conflicts with the AP1000's direct-mapped cache while allowing a good 'level-3' factor for moderate sized matrices. The workspace is allocated from dynamic memory storage upon procedure entry and freed before procedure exit. For reasons of modest workspace requirements and reliability, the 'explicit partitioning' method described in Section 3.2.3 of [13] was not adopted, even though it yields better performance for large matrices.

The use of two 'super-procedures' to implement the six DBLAS procedures was to facilitate prototyping, and to reduce code size and testing requirements. However, the drawback for such generalized code is its complexity, since all of the variants available in the DBLAS are not strongly orthogonal to each other.

2.2 Performance

The performance of `D_GEMM` is very close[1] to that of the preliminary results obtained in Tables 3 and 7 with [13], peaking at 7.1 MFLOPs/cell (single precision) and 4.5 MFLOPs/cell (double precision) where the size of the matrix per cell is about a half a cache size. The exception is where the matrix shapes approach that of a matrix-vector or vector-matrix multiply, since `ParallelBrdMult` does not currently optimize communication for these cases. Processor configuration does not greatly affect performance, except linear or near-linear AP1000 configurations tend to be slower. As matrix size per cell increases well beyond one cache size, performance degrades slowly.

`D_SYMM`, `D_SYRK` and `D_SYR2K` are about 10% slower for the corresponding matrix sizes than `D_GEMM`. This is because of extra overhead in transpositions and handling triangular matrices.

The performance of `D_TRSM` and `TRMM` correspond to the preliminary results in Table 9 of [13]. Thus, the fully implemented DBLAS routines can still achieve 80% of the peak performance of the AP1000 for moderate matrix sizes. The first release of these routines was made in May, 1993.

[1]It is generally a few percent less, due to extra matrix copying that was avoided in the less general preliminary algorithms.

```
void D_LU(N, A, LdA)
{ iw = 1;
  for (j = 0; j < N-1; j = j+1) {
    if (iw == 1) w = f(N, j);
    MaxAj = <global maximum of A[.,j]>;
    <exchange A[j,.] with row of A containing MaxAj>;
    DDGEMM("NoTrans", "NoTrans", N-(j+1), 1, 0, 0.0, A, 1, A, 1, 1.0/MaxAj,
           &A[sy(j+1), sx(j)],   LdA, 0, 0, 0, 0, (j+1)%Ny, j%Nx);
    if (iw < w) {
      DDGEMM("NoTrans", "NoTrans", N-(j+1), w-iw, 1, -1.0,
           &A[sy(j+1), sx(j)],   LdA, &A[s(j),  s(j+1)], LdA, 1.0,
           &A[sy(j+1), sx(j+1],  LdA,
           (j+1)%Ny, j%Nx, j%Ny, (j+1)%Nx, (j+1)%Ny, (j+1)%Nx);
      iw = iw+1;
    } else { jw = j+w-1; iw = 1;
      D_TRSM("Left", "Lower", "NoTrans", "Unit", w, N-(j+1), 1.0,
           &A[sy(jw), sx(jw)],   LdA, &A[sy(jw), sx(j+1)], LdA,
           jw%Ny, jw%Nx, jw%Ny, (j+1)%Nx);
      D_GEMM("NoTrans", "NoTrans", N-(j+1), N-(j+1), w, -1.0,
           &A[sy(j+1), sx(jw)],  LdA, &A[sy(jw),  sx(j+1)], LdA, 1.0,
           &A[sy(j+1), sx(j+1)], LdA,
           (j+1)%Ny, jw%Nx, jw%Ny, (j+1)%Nx, (j+1)%Ny, (j+1)%Nx);
    }
  } /* nb. sy(i) = i/Ny + (cidy() < i%Ny), Ny = ncely() */
}   /*     sx(i) = i/Nx + (cidx() < i%Nx), Nx = ncelx() */
```

Figure 1: Code for Parallel LU Decomposition using DBLAS

2.3 Applications of the DBLAS

This section outlines how the DBLAS may be used in the implementation of various distributed Linear Algebra algorithms.

2.3.1 LU Decomposition with partial pivoting

In [6], a hand-crafted and highly optimized implementation of LU Decomposition with partial pivoting and blocking on the AP1000 is described. This implementation is capable of achieving 4.5 MFLOPs (double precision) for very large matrices.

This section describes an implementation of the same algorithm, but derived from transforming a serial algorithm using BLAS Level 3 operations. An $N \times N$ global matrix A is overwritten $PA = LU$, where P is some row permutation matrix. The cell program can be succinctly expressed as shown[2] in Figure 1.

Here the macros `sy` and `sx`translates a global matrix index into a local index on the current cell. The first call to `D_GEMM` scales the jth column of A by the pivot; the second performs a rank-1 update on the $(N-j-1) \times (w - iw)$ global sub-matrix starting at $A_{j+1,j+1}$; the third performs a rank-w update on the $(N-j-1) \times (N-j-1)$ global sub-matrix starting at $A_{j+1,j+1}$.

Thus the parallel algorithm was derived from the serial one simply by translating global matrix offset to local ones and adding the ids of the cells where each global sub-matrix starts (this is a simple function of the global indices of the sub-matrix).

[2] We have corrected some bugs in the version of the program appearing in the *PCW'93 Proceedings*.

The D_LU procedure was tested by checking $PA = LUI$, using calls to D_TRMM:

```
D_TRMM("Left", "Upper", "NoTrans", "NonUnit", N, N, 1.0,
       A /*U*/, LdA, I, LdA, 0, 0, 0, 0);
D_TRMM("Left", "Lower", "NoTrans", "Unit", N, N, 1.0,
       A /*L*/, LdA, I, LdA, 0, 0, 0, 0);
```

Note that cell (0,0) contains the leading elements of global matrices A and I.

The code for the LU decomposition itself was developed in a few hours, the most difficult being the code determining and exchanging the pivot rows, which could not be done using the DBLAS. For moderate matrix sizes, the performance is comparable that of the hand-crafted implementation of [6], with peak speed of 3.7 MFLOPs for $N/N_x = 512$. For $256 \leq N/N_x \leq 512$, the performance is within 10% of that of the hand-crafted implementation[3]. For larger N/N_x, the performance does not improve, due to the non-optimal partitioning methods currently used in D_GEMM (see Section 2.1). For smaller N/N_x, the performance worsens relative to the hand-crafted implementation, since using the DBLAS results in data being broadcasted twice.

Cholesky Factorization was be implemented in a very similar way to LU Decomposition, except that pivoting is not required and D_SYRK is used instead of D_GEMM, since symmetric matrices are used. Here a peak speed of 3.2 MFLOPs (double precision) for $N/N_x = 512$ was achieved.

2.3.2 Matrix Star Product

In Quantum Mechanics, the scattering of particles having N quantum states by a barrier can be represented by a $2N \times 2N$ unitary *scattering matrix* S of the form:

$$S = \begin{pmatrix} t & \rho \\ r & \tau \end{pmatrix}$$

where t, r, ρ and τ are $N \times N$ matrices.

The effect of 2 barriers in series is expressed by the *Matrix Star Product* of their respective scatting matrices S_1 and S_2:

$$S_1 * S_2 = \begin{pmatrix} t_2(I - \rho_1 r_2)^{-1} t_1 & \rho_2 + t_2\rho_1(I - r_2\rho_1)^{-1}\tau_2 \\ r_1 + \tau_1 r_2(I - \rho_1 r_2)^{-1} t_1 & \tau_1(I - r_2\rho_1)^{-1}\tau_2 \end{pmatrix}$$

With the component matrices of S_1 and S_2 distributed in scattered storage across the AP1000, the Star Product can be simply implemented in terms of D_GEMM and a multiply by matrix inverse routine. Since $A^{-1}t = U^{-1}(L^{-1}(Pt))$ for $PA = LU$, then the latter involves applying D_LU(A, N, ...), permuting t by P and using the DBLAS calls:

```
D_TRSM("Left", "Lower", "NoTrans", "Unit", N, N, 1.0,
       A /*L*/, LdA, t, LdA, 0, 0, 0, 0);
D_TRSM("Left", "Upper", "NoTrans", "NonUnit", N, N, 1.0,
       A /*U*/, LdA, t, LdA, 0, 0, 0, 0);
```

Thus, most of the Star Product computation can be expressed in terms of D_GEMM and D_TRSM. For $128 \leq N/N_x \leq 512$, a performance of 4.0 – 4.3 MFLOPs (double precision) was obtained.

[3]A direct comparison cannot be made as this implementation does not do the 'backsolve step' required by the LINPACK Benchmark.

2.4 Discussions on the DBLAS and possible extensions

As shown by the examples in Section 2.3, the DBLAS make it relatively easy to parallelize serial algorithms expressed in mainly terms of BLAS Level 3 operations, provided a scattered matrix distribution is satisfactory. Indeed, much of the parallelization could be easily automated. For the scattered matrix distribution, the interface to the DBLAS is still relatively simple, with the addition of parameters specifying which cell each matrix starts in. The performance, while it can never be as good as handcrafted codes, will still be near enough for most purposes. In particular, DBLAS versions of algorithms like LU Decomposition and Cholesky Factorization may involve redundant communications. However, the program development time can be greatly reduced, especially if the algorithms need to work on non-square AP1000 configurations.

Distributed versions of High Performance Fortran (HPF) allows matrices to be implemented in block and block-cyclic distributions. It provides functions for global rectangular non-transposed matrix multiply and matrix inversion. The DBLAS can support the implementation of these functions provided it efficiently supports the most useful matrix distributions: the square block-cyclic and block distributions. Matrices having different distributions to these would have to be explicitly redistributed first.

For this purpose, `D_GEMM` and `D_TRSM` would at least have to be generalized. This however would complicate the DBLAS interface further, requiring two extra parameters per matrix to specify the offsets into a block where each matrix begins.

The question of whether the level of functionality of BLAS Level 3 is appropriate for distributed memory implementation can perhaps be answered on the current experience with the DBLAS. On one hand, it is difficult to provide a full implementation of all of the variants of distributed matrix multiply and update routines, especially for several matrix distributions, while giving the necessary guarantees of reliability and performance. This argues that the functionality should be reduced, at least as far as triangular matrices are concerned. It should be noted here that the DBLAS, even for only the scattered matrix distribution, requires substantial generalizations of cell-level BLAS to perform the local cell computations (see Section 2.1). On the other hand, our implementation of DBLAS using two 'super-procedures' (which provide automatically some extra functionality), plus a recognition of the usefulness of more general matrix storage schemes [3, 8], argues for yet more functionality and a perhaps more flexible interface.

3 Fast Fourier Transforms

The Basic Fourier Function (BAFF) Library is a high-performance Fast Fourier Transform Library analogous to the BLAS. It has been developed under the Fujitsu-ANU Area 4 Project [7], primarily for use on the Fujitsu VP2200 and VPP500 vector processors. The section gives an overview on the BAFF library and its porting to the Fujitsu AP1000.

The BAFF routines process vectors for Fourier transforms by regarding them as matrices of up to 5 dimensions; this provides a convenient way if specifying all ways of 'striding' though the vector necessary for the FFT. The BAFF library provides subroutines for 'elementary' Fourier Transforms (ie. of order ≤ 16), multiplication of a vector by roots of unity ('twiddle factors'), and transposition (ie. between the 2nd and 4th dimensions of the vector). With the BAFF, it is possible to succinctly express most of the mixed-radix FFT algorithms, as well as new FFT algorithms developed at ANU [7]. This is because of the BAFF library has high modularity; this modularity however has a drawback in terms of performance for RISC architectures, since the balance of floating point additions to multiplication (in each routine) is weakened. The implementation on the VP2200 demonstrated the effectiveness of this approach and speedups

N_x	128	64	32	16	8	4	2	1
n	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}
time (s)	11	10	10	9.4	7.5	7.3	6.8	6.1
MFLOPs	89	46	22	11	7	3	1.6	0.9

Table 1: Performance of BAFF-based FFT on a $1 \times N_x$ AP1000

between 2 and 20 were obtained compared with the resident library subroutines for large problem sizes.

The BAFFs can be used for a parallel FFT algorithm by storing the vector of length n to be transformed in scattered storage across a $1 \times N_x$ AP1000 configuration. Assume that $n = r^2$ and that r is a multiple of N_x. Then the data array x is treated as an $r \times r$ matrix X. The rows are equally distributed to the processors.

```
1. Each cell does an order r FFT on its rows
2. X is transposed across the processors
3. Each cell multiplies all the elements by the appropriate
   n-th roots of unity (twiddle factor multiply)
4. Each cell does an order r FFT on its rows (same as 1.)
```

This algorithm can be easily generalized for arbitrary n. An all-to-all communication is required is step 2; no other communication is required. The local cell FFT algorithms have a similar recursive structure to the above algorithm. Table 1 gives the results for the VP2200 BAFF code ported to the AP1000. Note that the FLOPs count is $5n\log_2(n)$, not including root of unity evaluations.

The performance is not as high as was hoped; this was partly due to the fact that the SPARC has no integer multiply instruction (needed for computing array indices) and has fewer integer registers. Thus the Fortran compiler for the SPARC was unable to optimize the code as well as that of the VP2200. The second reason is due to the parallelization of the algorithm: while the actual communication (step 2, above) itself was efficient, the associated 'twiddle factor' multiplication (step 3) required each cell to compute a large number of roots of unity. Thus, each cell spends a greater fraction of its time computing these roots of unity as N_x increases.

To overcome these problems, the BAFF code has to be rewritten so that the number of local (index) variables is reduced and as much index computation code as possible is taken out of the critical innermost loops.

Secondly, the 'roots of unity' computation must be speeded up. While many of the same roots of unity are computed on several cells, the lack of regularity in these redundant computations argue against trying to compute each different root on only one cell, and then communicating the root to all other cells that need it. Instead we have investigated methods to compute roots of unity faster than using a call to both `sin()` and `cos()` (each call is equivalent to about 120 FLOPs (at peak speed)) per root. These methods are based on compound angle formulae. The difficulty is in avoiding numerical errors introduced from many repeated applications of the formulae, especially if the terms are of opposite sign. The most promising method computes each root of unity from at most 2 applications of the compound angle formulae (always with terms of the same sign). It yields a maximum absolute error within $5x10^{-16}$ in computing the n roots of unity, where $n \leq 2048$, and requires only the equivalent of 15 FLOPs (at peak speed) per root. Note that the standard `sin()` and `cos()` routines themselves have an absolute error within $5x10^{-18}$. The results for the performance of these optimizations are not yet available.

N_x	time for 1st sweep (s)	
	no partitioning	partitioning
32	18.3 (97.1%)	18.2 (97.2%)
16	41.1 (97.3%)	34.7 (98.8%)
8	79.2 (97.7%)	67.4 (99.1%)
4	153.9 (97.0%)	131.8 (99.2%)

Table 2: Effect of partitioning on the SVD of an 512×512 matrix A on a $1 \times N_x$ AP1000 (cache hit ratio is in parentheses).

4 Singular Value Decomposition

In [2], various optimizations on the Hestenes method for Singular Value Decomposition (SVD) of a matrix A are described. These include scaling each column of A by the accumulated cosines of the angles of all rotations applied to it, which both balances and reduces floating point operations. A new optimization involves fast methods of recalculating the Euclidean norm of each column of A after it has been rotated: this improved performance by a further 30% for large matrices (eg. a 512×512 matrix on a 64 cell AP1000). However, both of these optimizations affect the numerical properties of the SVD algorithm, and while the correct results are still produced, they may adversely affect convergence (ie. increase the number of 'sweeps' required by the algorithm).

4.1 Optimizations for the memory hierarchy

The 'natural' parallelization of Hestenes algorithm uses a $1 \times N_x$ AP1000 configuration, with each processor having, for an $M \times N$ matrix A, N/N_x complete columns of A. For large M and P, this leads to poor cache utilization, since the potential cache reuse is proportional to N/N_x. One optimization is to decrease N_x while keeping the total number of processors constant, ie. to use an $N_y \times N_x$ AP1000 configuration, $N_y > 1$. However, the more N_y increases, the more vertical sumnation of inner products of segments of columns of A is required. While the AP1000's `xy` communication library provides efficient support for global sumnation across AP1000 columns, one sumnation must occur every rotation, so the overhead is significant. We have found the aspect ratios of $N_y : N_x = 1 : 2, 1 : 1$ to be optimal for $512 \leq N \leq 1024$ on a 16 processor AP1000. This resulted in an improvement of over 10% in execution time, and of 2% in cache hit ratio.

The benefit of increasing the AP1000's aspect ratio is however limited. In the parallel implementation of Hestenes algorithm, most of the computation involves each cell holding a 'left' and 'right' block of A, each containing $N/(2N_x)$ columns, and rotating every column in the left block with every column in the right. Cache utilization can then be improved by employing 'implicit partitioning methods' [13], ie. finding an optimal order for these rotations. The optimal order was found to be to split the right block into cache-sized sub-blocks, and perform all required rotations on this sub-block at once. The results obtained are given in Table 2.

The last two optimizations affect the cache level of the memory hierarchy. Rotating two columns together, which contributes the bulk of the FLOPs in the SVD computation, is poor on data reuse at the register level of the memory hierarchy. This can be improved by a factor of 2 by 'simultaneously' rotating four columns together, two from the left block and two from

N	execution time (s)	
	2-column	4-column
64	6.2	5.1
128	75.8	64.3

Table 3: Effect of using 2- and 4-column rotations on the SVD of an $N \times N$ matrix A on a 1×1 AP1000.

the right block (in BLAS terminology, this would be called a 'level 3' operation). This involves six rotations, as columns in the same block are not rotated against each other. However, to preserve correctness, the rotation angles between columns in the same block must be calculated as well, to make adjustments to the other 6 rotation angles. Even though this actually increases the FLOP count, it still affords significant speedup, as shown in Table 3.

4.2 Effect of the optimizations on convergence

Our preliminary studies on the optimizations that affect numerical performance (scaling of columns of A, fast recalculation of norms of columns of A and, to a lesser extent, using 4-column rotations) is that on average they do increase the number of sweeps by a small margin, but not enough to offset the speedup per sweep that they enable. This is largely because the time per sweep decreases during the SVD computation, as rotations become progressively more sparse. A compromise method which 'rescales' each column of A after ω rotations have been applied to it appears to be optimal, with a value of $\omega \approx 32$.

The last two optimizations given in Section 4.1 also modify the order of rotation significantly. However, our preliminary studies have shown that while on some matrices, the order of rotations affects convergence, averaged over many different random matrices, changing the order of rotations does not significantly affect convergence.

5 Orthogonal QR Decomposition

Orthogonal QR decomposition has many useful applications, e.g., to the solution of linear least squares problems, or as a preliminary step in singular value decomposition.

There are two widely used methods for QR decomposition of a matrix A – Householder transformations and Givens rotations. The QR algorithms based on Givens rotations have roughly twice the number of arithmetic operations as Householder algorithms, but the latter have roughly twice as many memory references as the former. On a single processor Householder transformations are usually cheaper than Givens rotations because the cost of a floating point operation dominates the cost of a memory reference. However, which of the two methods is superior in parallel computation is very much dependent upon the machine configuration. For example, on the Denelcor HEP, a shared memory multiprocessor, memory access times dominated floating point operation times. In [11] Givens algorithms were twice as fast as Householder algorithms. Different results obtained on distributed memory machines can also be found in [12].

Before computing a QR decomposition of a given matrix on the AP1000, one may ask questions such as:

- Among various QR decomposition algorithms, which one is most suitable for implementation on this machine?
- When the total number of cells is given, what is the best aspect ratio of the array to achieve optimal performance?
- How efficient is this machine in solving this particular problem?

To answer these questions, we have implemented various orthogonal decomposition algorithms on the AP1000. This section presents some interesting results obtained through extensive tests.

5.1 Effect of array configuration

Suppose that the total number of cells is $P = ncelx \times ncely$. The aspect ratio is the ratio $ncely : ncelx$. We estimate the effect of array configuration by fixing the problem size and varying the aspect ratio of the array. In this experiment the unblocked column Householder algorithm is applied to factorize a 1000×1000 matrix. It is shown in our experiment that the best aspect ratio is either one or two, that is, the array configuration should be rather square in order to obtain the best results for solving QR decomposition problems using Householder transformations on the 128 cell AP1000. This is mainly due to the following two contradictory factors. Since the communication required in the column Householder algorithm is column oriented, that is, the horizontal communication cost is higher than the cost for vertical communication, the number of columns in the computing array should be decreased so that the total communication cost introduced by the algorithm may be reduced. By further increasing the aspect ratio (even to P, that is, by using a one dimensional array to eliminate all the heavy horizontal communication), however, the performance can only get worse. The reason for this is that when the aspect ratio is increased, the height of the array is also increased which results in a longer communication distance for data broadcast in the vertical direction. The longer the distance between the sending cell and the receiving cell, the more expensive is the communication.

5.2 Householder method vs. hybrid method

It is known that on a distributed memory machine both the pipelined Givens and the greedy Givens algorithms are not as efficient as the hybrid algorithm, which is a combination of Givens and Householder algorithms [12]. We only implemented the hybrid algorithm. A (theoretical) complexity analysis for implementation of various QR decomposition algorithms on distributed memory machines reported in [12] states that the hybrid algorithm is more efficient than the Householder algorithm. However, the experimental results on the AP1000 do not agree with this claim. The two algorithms take about the same time to solve a 1000×1000 problem on the AP1000.

The recursive elimination procedure in the hybrid algorithm and the global summation in the Householder algorithm both act as synchronization points during the computation, which is one of the key factors in determining the active/idle ratio of cells in the system. However, the computation involved in a recursive elimination procedure is much more expensive than that in a simple global vector summation. The former will take a longer time to complete and thus decreases the active/idle ratio of cells. To reduce the cost for the recursive elimination procedure the number of rows in the array should be decreased. It is thus expected that as the total number of cells in the system increases, the optimal array configuration for implementation of the hybrid algorithm will not be as square as that for implementation of a Householder algorithm. The longer distance in data broadcasting leads to a higher communication cost, so this will certainly degrade the entire performance. Therefore, there is no obvious reason to choose the hybrid algorithm instead of the Householder algorithm as the preferred algorithm for solving the QR decomposition problem on machines like the AP1000.

Another reason for choosing Householder algorithms is that they are transportable across a wide variety of architectures and yet can obtain a reasonable performance on a given machine.

5.3 Effect of the block width

The standard (unblocked) Householder algorithm is rich in matrix-vector multiplications. On many serial machines, including the AP1000 cells, it is impossible to achieve peak performance because matrix-vector products suffer from the need for one memory reference per multiply-add. The performance is thus limited by memory accesses rather than by floating-point arithmetic. Near peak performance can be obtained for matrix-matrix multiplication [10].

To measure the effect of block width on the performance, we have implemented the block WY Householder algorithm. In the experiment the number of cells involved in the computation is fixed at 64, i.e., an 8×8 array is used. To solve a problem of a given size, different block widths are applied. Two points should be stressed. First the block Householder algorithm becomes more efficient only after the size of the matrix to be decomposed on an 8×8 array is greater than 1024×1024. Since the matrix is distributed across the whole array, each cell only holds a relatively small sub-matrix. We know that each cell on the AP1000 has a 128 KByte cache. The cache effect is severe only when the size of the stored sub-matrix is greater than the cache size. When a 1024×1024 matrix is distributed over an 8×8 array, each individual cell only stores a 128×128 sub-matrix. Suppose that double precision is applied. The sub-matrix will only occupy about 128 of KByte memory, which is the same size as the cache. It is known that the block Householder algorithm requires extra computations for generating matrix W, therefore, the block Householder algorithm will not be as efficient as the unblocked one if the size of the matrix to be decomposed is less than 1024×1024.

The second point is that the optimal block width is about 8 (or 16 for larger matrices). If the same algorithm is executed on a single cell, however, the optimal block width may be greater than 60. During the generation of the matrices W and Y in each stage of the block column Householder algorithm, only the cells in the leading row of the array are active, but all other cells remain idle, requesting inputs from the corresponding cells in the leading row. The greater the block width, the longer the time for those cells waiting for inputs. This obviously decreases the active/idle ratio of cells. Thus any restriction to a further increase in the block width is mainly a result of a lower active/idle ratio of cells.

5.4 Speedup and efficiency

Let t_p denote the time to execute a given job on P processors. The speedup S_p and the efficiency E_p for a system of P processors are defined as $S_p = t_1/t_p$ and $E_p = S_p/P$. Since Householder algorithms are preferred for the AP1000, the column Householder algorithm is applied to decompose a 1000×1000 matrix in measuring the speedup and the efficiency. The results are given in Table 4. In comparison with the LINPACK Benchmark results presented in [6], the results shown in Table 4 are somewhat lower in both speedup and efficiency. This is because the communication involved in parallel implementation of Householder algorithms is more complicated and so is more expensive.

time for one cell	cells	time (sec)	speedup (S_p)	efficiency (E_p)
467	128	7.58	61.6	0.48
467	64	11.2	41.7	0.65
467	32	20.9	22.3	0.70
467	16	36.9	12.7	0.79
467	8	69.3	6.74	0.84
467	4	130	3.60	0.90
467	2	254	1.84	0.92

Table 4: Speedup and Efficiency of the Householder algorithm for a 1000×1000 matrix.

Based on our extensive experiments and theoretical analyses several interesting results are summarized in the following:

- Among various orthogonal factorization methods, Householder transformation techniques are the most effective for QR decomposition of a matrix on the AP1000.
- The aspect ratio of the array significantly affects the performance, especially when the total number of cells involved in the computation is large. Although communication in the column (or row) Householder algorithm is predominantly column (or row) oriented, the best aspect ratio of the array is either one or two from our experiments. The main reason is that long distance data broadcasting is still relatively expensive on the AP1000.
- Because of extra computations and communications required in forming matrix W (or T) in the block Householder algorithm, the block width needs to be very small. Moreover, the block algorithm is more efficient than its unblocked counterpart only if the size of a matrix to be decomposed is larger than 1024×1024 on the 128 cell AP1000.
- QR decomposition of a matrix is much more complicated than LU decomposition in terms of both arithmetic operations and communications required for parallel implementation. The efficiency achieved for QR decomposition on the AP1000 is lower than the LINPACK Benchmark result [6]. However, the degradation in efficiency, relative to that of the LINPACK Benchmark, is less than 10%. This further confirms claims that the AP1000 is a good machine for numerical linear algebra.

6 Conclusions

The Linear Algebra project has investigated in detail the implementation of a variety of numeric algorithms on the AP1000. We have found that many of optimization techniques applied to one algorithm can be applied to the others. While most of these techniques can be to applied any SPARC-based parallel architecture, some of them are quite general in applicability, and indeed, some (eg. in Section 4) can be applied even to serial processors. With the availability of the DBLAS, we will be able to implement several new algorithms having acceptable performance, with relative ease. We have found the AP1000 gives good support, in both hardware and software, for developing efficient parallel numerical algorithms.

References

[1] *CAP Research Program, 1993 Report*, The Australian National University and Fujitsu Laboratories Ltd., Canberra, May 1993.

[2] A. Czezowski and P. E. Strazdins, *Singular Value Computation on the AP1000*, in [4].

[3] R. D. Falgout, A. Skellum, S. G. Smith and C. H. Still, "The Multicomputer Toolbox Approach to Concurrent BLAS and LACS", *The 1992 MPCI Yearly Report*, Lawrence Livermore National Laboratory, August 1992.

[4] R. P. Brent (editor), *Proceedings of the CAP Workshop '91*, Australian National University, Canberra, Australia, November 1991.

[5] R. P. Brent and M. Ishii (editors), *Proceedings of the First CAP Workshop*, Kawasaki, Japan, November 1990.

[6] R. P. Brent, "The LINPACK Benchmark on the Fujitsu AP 1000", *Proc. Frontiers '92*, Virginia, USA, October 1992, 128–135.

[7] R. P. Brent, A. J. Cleary, M. Hegland, M. R. Osborne, P. J. Price, and S. Roberts, *Intermediate Report on the Fujitsu-ANU Parallel Mathematical Subroutine Library Project*, Australian National University, Canberra, 1992.

[8] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, *ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers*, Technical Report CS-92-181, University of Texas, November 1992.

[9] J. J. Dongarra, J. J. Du Croz, S. J. Hammarling and I. S. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms", *ACM Transactions on Mathematical Software* 16 (1990), 1–17.

[10] J. J. Dongarra, I. S. Duff, D. C. Sorensen and H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, 1990.

[11] J. J. Dongarra, A. H. Sameh and Y. Robert, "Implementation of some Concurrent Algorithms for Matrix Factorization", *Parallel Computing* 3, 1986.

[12] A. Pothen and P. Raghavan, "Distributed Orthogonal Factorization: Givens and Householder Algorithms", *SIAM J. Sci. Statist. Computing* 10, 1989.

[13] P. E. Strazdins and R. P. Brent, *Implementing BLAS level 3 on the AP1000*, in [4].