A High Performance, Portable Distributed BLAS Implementation

Peter Strazdins,
E-mail: Peter.Strazdins@anu.edu.au
Australian National University.

Department of Computer Science,
Australian National University,
Acton, ACT 2600. AUSTRALIA

Phone: +61 6 249 5140
Fax: +61 6 249 0010

Abstract

In this paper, we give a report on recent developments for the Distributed BLAS (DBLAS) project. These include a powerful distributed matrix representation which yields a simple interface to the DBLAS, and the redesign the DBLAS algorithms terms of powerful 'spread' and 'reduce' matrix communication operations for reasons of programmability.

The DBLAS codes achieve portability by supporting BLACS and various forms of ApLib, including a locally developed 'stride' ApLib (for the Fujitsu AP1000/AP+), which is optimal for the 'spread' and 'reduce' operations. Also, portability of ensuring high speed cell computation across various cell architectures involved designing a cell BLAS interface and the expression of the DBLAS algorithms in terms of a set of tunable architecture-dependent parameters. Cell BLAS algorithms have also been extended for SPARC 10 platforms, and required deeper software pipelining for optimal performance. More extreme techniques were required for the UltraSparc, with a preliminary matrix multiply kernel achieved 250-300 MFLOPs

The DBLAS have been used to produce elegant parallel LU, Cholesky and QR factorization algorithms, using the 'distributed panels' technique, and we report results for the QR factorization on the AP1000, and LU factorization on the AP+. Here, this technique has yielded good performance results, being up to 15% faster than traditional algorithms for moderate matrix sizes. Performance problems include $O(N)$ software startup overheads which have a significant impact on dense linear algebra algorithms on $N \times N$ matrices.

1 Introduction

The portable and efficient implementation of extensive dense linear algebra libraries on distributed memory processors requires a parallel implementation of the Basic Linear Algebra Subroutines (BLAS). This has been the approach for the ScaLAPACK library [2], which is expressed in term of a version of distributed BLAS, called PBLAS, which supports the block-cyclic matrix distribution [3]. Efficiency and portability are achieved by implementing the distributed BLAS algorithms in terms of two machine dependent libraries: the BLAS itself and Basic Linear Algebra Communication Subprograms (BLACS), which can be optimized on each machine to achieve high performance in cell computation and communication respectively.

The Level 3 PBLAS algorithms suffers from a restriction which results in sub-optimal performance on memory hierarchy distributed memory processors. This occurs when all of the matrix operands of a computation have more than a single row or column of blocks in each matrix dimension. It has been shown that this restriction has a significant impact on parallel matrix factorization algorithms on the Fujitsu AP1000 [6].

An alternative approach to distributed BLAS, that does not suffer from this performance restriction, called the DBLAS, has been developed at ANU, firstly for the scattered or 1×1 block-cyclic matrix distribution [1], and then generalized for the general $r \times s$ block-cyclic matrix distribution [5, 6]. However, for this performance advantage, the DBLAS required more complex algorithms and data structures. Furthermore, several performance assumptions of AP1000-like architectures were implicitly built in to the algorithms.

In this paper, we outline the software engineering issues required to make the DBLAS portable not only in the normal sense, but to achieve portability of performance also. The latter is most important for the Fujitsu AP+ and AP3000 platforms, where high cell computation speed must be maintained.

Section 2 describes the overall architecture of the DBLAS library, together with the data and procedural abstractions required to elegantly and powerfully express parallel matrix operations. In Section 3, issues in portability with respect to communication are discussed, whereas Section 4 discusses those with respect to cell computation. Section 5 gives some results of matrix factorization algorithms implemented using the DBLAS on the Fujitsu AP1000 and AP+, and discusses further performance issues exposed by these results. Conclusions are given in Section 6

2 Overall Architecture of the DBLAS Library

The objectives of the design of the DBLAS are to support equally both Fortran and C interfaces, to be able to support clean, high-level parallel matrix computations (with details of data distribution and parallelism hidden), to be fully portable in terms of being able to run on various platforms, to have performance portability by means of a tunable set of machine-dependent parameters, and to be engineered to be as maintainable and extensible as possible. Meeting these required a high degree of re-organization of the previously developed software, resulting in the overall architecture depicted in Figure 1.

Currently, distributed versions of Level 1, 2 and 3 BLAS are supported, for the single and double precision data types.

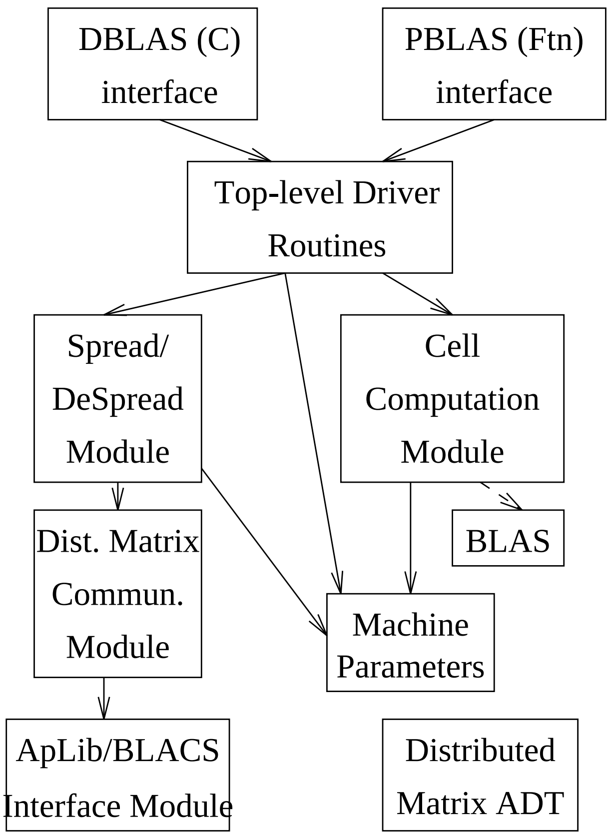

Figure 1: Overall Architecture of the DBLAS Library

2.1 Issues in Distributed Matrix Representation

The appropriate representation of local and distributed matrices in the form of Abstract Data Types (ADTs) are central to meeting the objectives outlined above.

In order to support equally row-major and column-major local matrix storage, a local cell matrix is represented by the `LocMat` structure a =`(double *ax, int Inc[0..1])` where `ax` is the address the first element and `Inc[0]` (`Inc[1]`) represents the row (column) strides of the matrix. Thus the address of `a[i,j]` is given by $\mathtt{ax} + \mathtt{i} * \mathtt{Inc}[0] + \mathtt{j} * \mathtt{Inc}[1]$.

Generally, local matrices will be either row-major or column major, eg. $\min(\mathtt{Inc}[0..1]) = 1$, but the matrix copy routines used to set up the 'spread' / 'reduce' workspaces (see Section 2.2) require non-unit row and column strides.

A distributed matrix can be represented by the `DistMat` structure A =(`int t, LocMat a, MatDim D[0..1]`). This is a slight extension of the representation described in Section 2 of [5], with the boolean variable `t` representing whether the matrix is transposed, `a` is the local matrix descriptor and `D[0]` (`D[1]`) is the (block-cyclic) distribution dimension descriptors across the vertical (horizontal) processor dimensions. These contain block and processor grid sizes and offsets across that dimension. If A is regarded as an $M \times N$ matrix, M (N) corresponds to `D[t]` (`D[!t]`).

In the case of distributed vectors (eg. in the Level 1 procedures), the value of the transpose bit determines whether the vector in question is a column vector ($\mathtt{t} = 0$) or a row vector ($\mathtt{t} = 1$).

The *data abstraction* afforded by the `DistMat` representation enables an elegant and powerful expression of distributed matrix operations (see Section 5), and is used for the C interface of the DBLAS and for the internal modules also. However, since a structure cannot (easily) be implemented in Fortran, the Fortran interface uses the PBLAS matrix descriptors, where a distributed matrix A is represented by the the 4 quantities `ax`, i, j and `DESCA`, where (i, j) is the global matrix offset and `DESCA` is a 9-element integer array [3].

2.2 Spread and Reduce Operations

The implementation of a distributed BLAS library can be greatly simplified by recognizing that all matrix communication operations can be efficiently expressed in terms of two powerful complementary operations, called 'spread' and 'reduce'. In the context of hypercube networks, these operations have been called 'recursive doubling' and 'recursive halving' respectively [7]; we do not use this terminology as it implies a sub-optimal implementation on torus-connected networks of relatively low communication startup times, such as the Fujitsu AP1000 and AP+. These operations form an appropriate *procedural abstraction* for the distributed BLAS.

Consider the distributed computation $\mathtt{C} \leftarrow \alpha \mathtt{A}^T \mathtt{B} + \beta \mathtt{C}$, where `C` is $M \times N$ and $\mathtt{A}^T$ is $M \times K$.

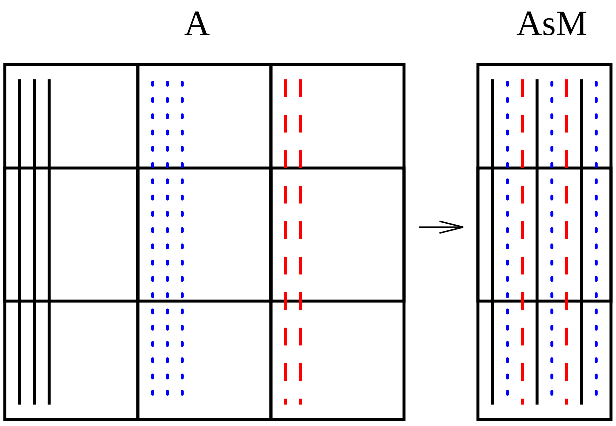

Figure 2: Row 'spread' of the distributed matrix A (having 8 block columns), across the rows of a 3×3 processor grid. The result is stored in `AsM`, which is replicated across all processor columns.

If $M \ll K, N$ then it is efficient to perform this computation by the following steps:

(i) replicate (row 'spread') the M columns of A across the rows of processors, scaling them by α, and storing them a local workspace `AsM` (having local size $M \times k$).

    Note that ith row of `AsM` contains that processor row's portion of the global ith column of A, see Figure 2.

(ii) creating a local workspace `CsM` of size $M \times n$

(iii) performing a local cell matrix multiply on `AsM` and the local portion of `B`, and storing the result in `CsM`.

(iv) sum (column 'reduce') each of the M rows of `CsM` across the columns of processors, and adding the sum of the ith row to the ith row of the global `C` matrix.

    In some sense, the 'reduce' operation is the reverse of the 'spread' operation.

The call `DistVMMult`(M, N, K, α, `A`, `B`, β, `C`) of Figure 3 performs this computation. In the above case, the transpose bit of `A` is set. However, the 'mirror-image' of the above computation, the parallel matrix-vector multiply $\mathtt{C} \leftarrow \alpha \mathtt{A}\mathtt{B}^T + \beta \mathtt{C}$, where $N \ll K, M$, can be implemented using the call `DistVMMult`(N, M, K, α, `B`, `A`, β, `C`), with the transpose bits of `A` and `C` set. This illustrates the utility of these procedural and data abstractions.

```
void DistVMMult( int M, int N, int K,
  double alpha, DistMat A, DistMat B,
  double beta, DistMat C )
{ int Row = ! TransBit(B);
  MatDim BDK=Dim(!Row, B), BDN=Dim(Row, B);
  int n = G2Ld(N, BDN), k = G2Ld(K, BDK);
  LocMat AsM, CsM;
  SpreadMat(Row, M, K, A, alpha, BDK, &AsM);
  InitReduceWS(!Row, dM, N, C, BDN, &CsM);
  MatMult(M, n, k, AsM, LocMatrix(B), CsM);
  ReduceMat(!Row, M, N, C, beta, BDN, CsM);
}
```

Figure 3: Simplified C code for parallel vector-matrix multiply

For level 3 DBLAS computations, the algorithms based on the 'spread' and 'reduce' operations were found to be no slower than that reported for the previously used algorithms [5]. They were found to have an important further advantage: the block sizes of one of the common matrix dimensions (eg. in Figure 3, the dimensions of `A` and `C` corresponding to M) need not be compatible. This is because the block structure of these dimensions has been collapsed when the 'spread' workspaces (eg. `AsM` and `CsM`) are formed.

The 'spread' workspaces of `AsM` of Figure 2 across a processor column can be made into a `DistMat` structure `sA`, which is similar to `A` but having the number of processors across the horizontal dimension descriptor (`D[1]`) set to 1. `sA` can thus be considered as a column-replicated distributed matrix, and the outer-level DBLAS procedures can accept distributed matrix operands of this type. Consider the update of the symmetric matrix $\mathtt{C} \leftarrow -\mathtt{A}\mathtt{A}^T + \mathtt{C}$ (used extensively in Cholesky factorization); the 'spread' of $\mathtt{A}^T$ across the columns of processors can be efficiently performed, without explicit transposition, using a (modified) column 'spread' of `sA`.

Any distributed matrix transposition and/or alignment that is required in a distributed BLAS operation is performed within the 'spread' or 'reduce' operations, which also perform any communication optimizations afforded by the matrix shape and size. These communication sub-operations are performed inside the Distributed Matrix Communication module of Figure 1. This module assumes the underlying communication primitives effectively buffer the message in a system area upon receipt, eg. in the 'ring buffers' of the AP1000 or AP+ [4]. Where the communication requires large messages, partitioning of the messages occur so that the maximum size of unreceived messages can be kept within a constant defined in the Machine Parameters module (see Figure 1).

3 Portability Issues in Communication

The ApLib/BLACS Interface Module of Figure 1 is the main feature for portability. This provides low-level communication routines that may be expressed in terms of the Basic Linear Algebra Communication Subroutines (BLACS) [2] or ApLib operations. Note that there are MPI versions of the BLACS available.

The most important communication sub-operation of the 'spread' operation of Section 2.2 is a stride vector row/column broadcast. Stride communication is required since, in the case of the 'spread' of Figures and 2 and 3, each cell copies its own portion of the rows of the `AsM` workspace (which occur in a striped pattern) before broadcasting directly from this workspace. In this case, the size of the 'vector' would be the local column size (eg. `k`) multiplied by the column block size s. Similarly, the most important sub-operation of the 'reduce' operation is adding a received message into a striped part of a workspace (eg. `CsM`).

While the BLACS has sufficient functionality to express such a vector stride, the 'stride' ApLib version has the fundamental advantage that in the 'reduce' operation, the message can be added directly from the ring buffer, thus avoiding a memory copy operation.

It should be noted here that it would difficult to use the AP1000/AP+ remote memory (PUT/GET) model to advantage here, as for example using a 'multiple PUT' operation to implement a row or column broadcast would require the spread workspaces to be statically allocated. Furthermore, this model would also require extensive modifications to the Distributed Matrix Communication module as well.

3.1 ApLib Interface

The ApLib/BLACS Interface Module can be compiled to use a standard AP1000 communication library (here referred to as ApLib) interface. The `xy` communication mode is chiefly used, where messages are received into the AP1000/AP+ ring buffers. However, these are not optimal for the purpose of supporting efficient 'spread'/'reduce' operations because it is not possible to send or receive a stride vector without first either performing message packing or unpacking, or sending the elements of the stride vector as individual messages.

What is required is (i) a vector stride sending capability (for point-to-point operations as well as row/column broadcast), and (ii) receive routines that return ring buffer pointer(s) so that the (Cell Computation Module) routines can either copy or add the message into a striped 'spread' or 'reduce' workspace.

For the AP1000 and AP+, a modified subset of the standard ApLib's `xy` routines, called 'stride' ApLib, has been implemented to meet these requirements. The Interface Module can also be compiled to use 'stride ApLib'.

For the AP1000, the line sending mode and ring-buffer wrap-around present special challenges. For the former, because the cache-line size L exceeds the data element size (eg. 8 bytes for double precision), extra words must be sent, and then subsequently filtered out when copying out from the ring buffer. The pattern of these extra words is semi-regular, but can be efficiently determined using the $\mathrm{GCD}(s, L)$, where s is the stride between vector elements.

Provided that the fractional volume of the extra words sent is less than 1/8, this method is more efficient than packing the message before sending. The ideas in this algorithm could also be applied to implementing stride-vector remote memory operations using DMA.

For the AP+, the implementation is simplified, since stride PUT operations (into the destination cell's ring buffers) can be used to provide these requirements. However, the communication startup time of the AP+ is so low that sending the vector elements by individual messages, as is done in the standard ApLib interface, was found to be $< 3\%$ slower than using 'stride' ApLib interface in most situations.

4 Portability Issues in Cell Performance

The Cell Computation Module of Figure 1 performs all computations required for Levels 1, 2 and 3 of the DBLAS for updating local cell matrices and vectors. For double precision data, all algorithms have been written entirely in C, including the Matrix Multiply Kernels.

In this module, there are higher-level routines for performing partitioned matrix multiply for rectangular and triangular matrices and block matrix packing and unpacking. There are also low-level routines, performing functions such as (`LocMat`) matrix copy, addition and scaling, as well vector sum and maximum operations. Finally, there are the Matrix Multiply Kernels (see Section 4.1).

These low-level routines use 2 or 4-way loop unrolling and deep software pipelining, and can achieve good performance across a broad range of modern scalar processor architectures. The Matrix Multiply Kernels however are less portable in terms of performance, since they are optimal only for processor architectures with 16 double precision registers available (eg. the SPARC 1 and SPARC 10), and which do not support a combined multiply-add instruction (which is available on the DEC Alpha and MIPS R8000 line of processors).

Thus, in terms of optimal cell computation performance, the portability of these routines is inadequate. For this reason, this module can be compiled to bypass the low-level routines and the Matrix Multiply Kernels and use a standard BLAS interface instead, where a highly optimized vendor-supplied (cell) BLAS library is generally available.

However, this in itself is still not sufficient for performance portability. This is because 5 of the 6 Level 3 BLAS routines involve triangular matrices; these are inadequate for the triangular matrix cell computations for the distributed

BLAS, since a triangular matrix distributed distributed over a $P \times Q$ processor mesh will have a diagonal of slope $\approx P/Q$ in local cell memory. For this reason, triangular matrix operations must be partitioned into rectangles; the optimum units for this partitioning reflect the size of the register files and caches on the processor architecture, and whether the optimal cell matrix multiply is inner-product based (as for most scalar processors) or saxpy-based (as for vector processors). These units are encapsulated in about a dozen constants in the Machine Parameters module.

The determination of the best values of these constants may be difficult, since in the last analysis they will depend on the implementation of the local cell BLAS, whose internal details might not be available. For this reasons, the constants of the Machine Parameters can be compiled to be tunable at run-time, so that the optimum values can be determined by experiment, if need be. This has the added benefit of helping to debug the partitioning algorithms, where matrix sizes and the partitioning parameters can be scaled down together to manageable sizes.

4.1 Matrix Multiply Kernels for SPARC Processors

The basic techniques required to implement these Kernels for the SPARC 1 architecture are described in Section 3 of [1]. To gain optimal performance on the SPARC 5 and 10 architectures, deeper software pipelining techniques were required, and, in the case of 'reduction' computations like inner product computations, to use at least 2 or 4 accumulator registers. This is because floating point addition has a 3-stage pipeline on these architectures.

On SPARC 10 workstations, these routines generally compared favourably with the equivalent BLAS routines from Sun Microsystems' Performance Library[1]. A large matrix multiply computations had a speed of 40 MFLOPs; however, on an AP+ cell, the speed reduced to 33 MFLOPs, due to a second level of cache not being available.

For the UltraSparc, these Matrix Multiply Kernels do not port efficiently (achieving at best 200 MFLOPs for matrix multiply). They need to be re-written, as 32 double precision registers are available. Generally, the same degree of outer loop unrolling is used as for the SPARC 1-10 codes, but with a deeper software pipelining between the floating point multiply and dependent floating point add instructions. The main new technique required is (top-level) 'cache lookahead', where the innermost loop is unrolled twice, and the data values get loaded at least 8 cycles, eg. a full iteration, in advance. This technique effectively bypasses the UltraSparc's top-level cache, so that the partitioning methods for large matrices need only consider the large second-level cache.

Preliminary results using gcc-compiled codes for matrix multiply are given in Table 1; these appeared to be the fastest results published for the 170 MHz UltraSparc at the time (August 1996). A matrix copy routine using the 'cache lookahead' technique achieved 625 MBs^{-1}, 520 MBs^{-1} and 120 MBs^{-1}, for matrices of sizes 8 KB, 32 KB, and 2 MB respectively.

N	32	64	128	192	256	384	512
C+=A^TA	316	299	311	283	260	238	52
C+=A^TB	312	295	269	256	71	52	37
C+=A^TB			218	257	263	240	236

Table 1: Matrix Multiply speed (MFLOPs) on the UltraSparc I; A^T, B are $K \times N$, with $K = 128$ (for the third row, partitioning methods have been applied).

The difference between C+=A^TA and C+=A^TB is simply that the data size is smaller for the former. From these results, the following observations may be drawn for matrix computations on UltraSparc-based parallel processors, such as the AP3000.

- a large value of K (eg. $K \approx 128$) is required for good performance. This reflects the size of the second-level cache, and to a lesser extent, relatively high loop startup costs. This implies a large 'panel width' is required for matrix factorization algorithms, for example.

[1] This was released mid 1995, and is available for Solaris 2 platforms

- the effect of second-level cache misses can be dramatic and is largely unpredictable. Ensuring consistently high cell performance on the UltraSparc on a varying range of matrix sizes may be difficult.

5 Applications of the DBLAS

In this section, we describe some matrix factorization algorithms expressed using the DBLAS, and discuss some outstanding performance issues that these applications have been exposed. In general, the matrix factorization algorithms are a severe test of the performance of a distributed BLAS implementation. See [6] for results of DBLAS-based LU and Cholesky factorization algorithms on the AP1000.

5.1 QR Factorization on the AP1000

Figure 4 gives the DBLAS-based pseudo-code for a parallel QR factorization algorithm; it only differs from the actual DBLAS code for minor syntactic details, for the sake of brevity. One of these is $\sigma(A, i, j)$ to represent the sub-matrix at global position (i, j) of the distributed matrix A. The predicate D`DOwnsElt(`σ`(A, j, j))` returns true only if global element `A[j,j]` is on the current processor; the call D`DGetElt(`σ_1`(A, j, j))` returns the value of of `A[j,j]` (but only on the cells that owns it). The call D`DGetElt(`σ`(A, j, j),` e`)` sets global element `A[j,j]` to e.

Apart from these, all details of parallelism and matrix distribution are hidden within the `DistMat` structures and the DBLAS calls. A reader familiar with BLAS and the LAPACK would recognize that this algorithm corresponds to the serial LAPACK QR algorithm, but its expression is in fact cleaner and shorter. A difference from the LAPACK algorithm is that the triangular factor matrix `T` is in lower, rather than upper, triangular form. This is to avoid explicit transposition which would be required if `T` was upper triangular. For the sake of simplicity, τ is a vector replicated across all processors. A row broadcast of τ is required within D`DLARFT()`, and this is the only explicitly parallel code required in the whole algorithm.

The routine D`DLARFB()` (and D`DLARFT()`) is only implemented for the 'forward' direction

```
DDLARFB(int M, int N, int K, DistMat V,
       DistMat T, DistMat C)
{ DistMat W = NewMat(K, N, C, NULL);
  DistMat C' = σ(C, K, 0), V' = σ(V, K, 0);
  DDScaleAdd(K, N, 1., C, 0., W); /* W ← C */
  DDTRMM("L", "L", "T", "U", K, N, 1., V, W);
  DDGEMM("T", "N", K, N, M-K, 1., V', C', 1., W);
  DDTRMM("L", "L", "N", "N", K, N, 1., T, W);
  DDGEMM("N", "N", M-K, N, K, -1., V', W, 1., C');
  DDTRMM("L", "L", "N", "U", K, N, 1., V, W);
  DDScaleAdd(K, N, -1., W, 1., C); /* C-=W */
  DisposeMat(W);
}
DDGEQF2(int M, int N, DistMat A, double τ[])
{ int j; double Ajj;
  for (j = 0; j < MIN(M,N); j++) {
    DDLARFG(M-j, σ(A, j, j), &τ[j]);
    if (DDOwnsElt(σ(A, j, j))
      Ajj = DDGetElt(σ1(A, j, j));
    DDSetElt(σ(A, j, j), 1.);
    DDLARF(M-j, N-j-1, σ(A, j, j), &τ[j],
           σ(A, j, j+1));
    DDSetElt(σ(A, j, j), Ajj);
  }
}
DDGEQRF(int M, int N, DistMat A, double τ[])
{ int j = 0 ;
  while (j < MIN(M,N)) {
    int w = f(N, j);
    DistMat T = NewMat(w, w, σ(A, j, j), NULL);
    DDGEQF2(M-j, w, σ(A, j, j), &τ[j]);
    if (j+w < N) {
      DDLARFT(M-j, w, σ(A, j, j), &τ[j], T);
      DDLARFB(M-j, N-j-w, w, σ(A, j, j),
                T, σ(A, j, j+w));
    }
    DisposeMat(T); j += w;
  }
}
```

Figure 4: Pseudo code for Parallel QR Decomposition using DBLAS

here. However, 4 variants of the corresponding LAPACK DLARFB(), which are simply the combinations of whether the V and C matrices are transposed, are all encapsulated implicitly in the DDLARFB() code here by the ability to set the transpose bits of the corresponding DistMat operands instead. These combinations are then transparently handled by the underlying DBLAS procedures.

By comparison, the ScaLAPACK code [2] of equivalent functionality to that expressed in Figure 4 is $\approx$ 500 lines of uncommented Fortran code, and is less general in that it can only implement the case where w $= r$ (the block size), although in the authors of ScaLAPACK here have consciously traded off the simplicity of their code to avoid redundant communications of the columns of V, which the current implementation of Figure 4 does not.

Figure 5 gives the performance of the QR algorithm on the AP1000. The curve for $\omega = r = 16$ represents an algorithm which corresponds closely to the Householder QR algorithm reported previously for the AP1000 [8]; it however represents considerably better speed, due to the highly optimized computation and communication components of the DBLAS. The plots for $r = 1$ give the QR algorithm for the 'distributed panels technique, and are 10-15% faster.

Comparing these results with the performance of the LU and Cholesky decompositions [6], it can be noted that a much smaller value of the panel width $\omega = 32$ is optimal for QR factorization, due to the $O(N\omega^2)$ floating point operations introduced by the Level 3 QR algorithm. This results in a smaller relative performance gain of the distributed panels technique.

Also the effect of $O(N)$ overheads is far more evident in the graphs for QR factorizations than for LU factorization. This is highlighted by the $r = 1$ Level 2 performance (using DDGEQF2()). This should in principle always be slower, but is in fact considerably faster for $N \leq 1024$, since DDGEQF2() has about half of the software overheads that DDGEQRF() has.

5.2 LU Decomposition the AP+

The DBLAS LU factorization algorithm developed for the Fujitsu AP1000 [6] ported easily to

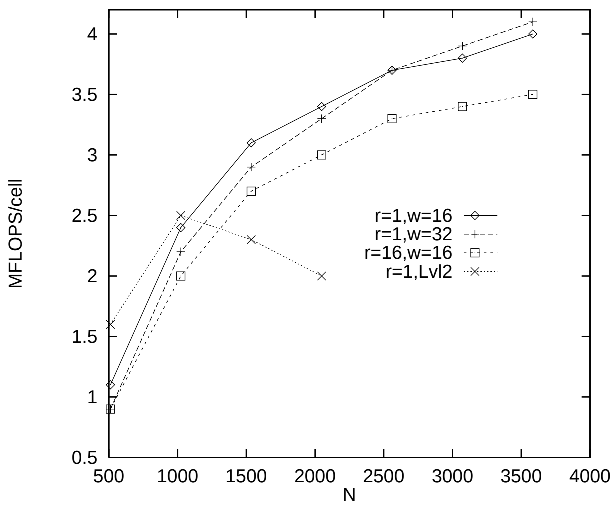

Figure 5: DBLAS-based QR factorization of an $N \times N$ matrix with $r \times r$ blocks and panel width ω, on an 8×8 AP1000

the Fujitsu AP+. The DBLAS based algorithm was essentially that described in Section 4 of [5]: the only significant difference is that the pivoting of the row segments outside the lower panel (L) is deferred until L is completely formed: this affords a small amount of extra parallelism in the pivoting operation.

The results are given in Figure 6. Firstly, the plots for $r = 1, \omega = 32$ and $P = 1, 2, 4$ indicate good scalability of the AP+ for LU factorization, at least for large matrices ($N/P \geq 500$). However, it should be remembered that for these algorithms, it is easily possible for performance to improve slightly with increasing P for fixed N/P, since the fraction of level 3 computation is $O(N^3/P^2)$. On the other hand, for $P = 4$, communication overhead is artificially increased due to the fact that the fast xy row/column broadcast cannot be used on a logical 4×4 mesh when the physical AP+ configuration is 2×8.

However, low speeds observed for small N/P indicate again significant $O(N)$ overheads. Since this trend exists for $P = 1$, these results indicate that the overheads are due to the software rather than communication startups, and these effects are more significant than on the AP1000. Secondly, even for large N, only half the theoretical peak speed obtained, whereas on the AP1000,

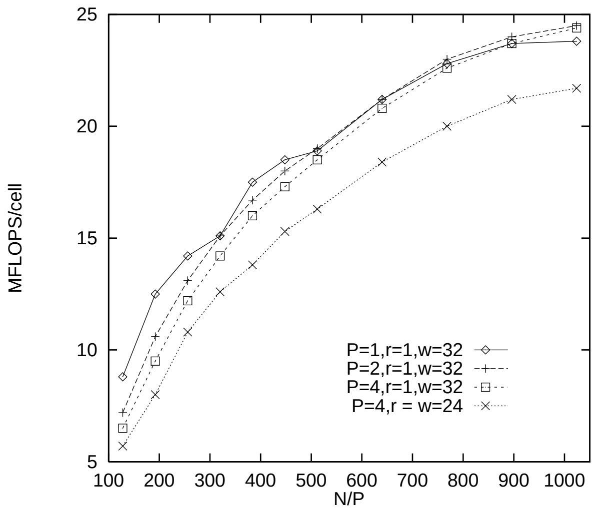

Figure 6: DBLAS-based LU factorization of an $N \times N$ matrix with $r \times r$ blocks and panel width ω, on a $P \times P$ configuration of a 16-node AP+

85% of the theoretical peak is attained for the same matrix sizes.

Comparing the plots for $P = 4$ indicates the distributed panels technique still yields a 10-15% improvement in performance, which while still significant, is not as great as that for the AP1000. This is due to the smaller cache size (16KB) of the AP+, which results in an optimal panel width ω of only 32.

5.3 Outstanding Performance Issues

In the previous two sub-sections, a major problem in performance for small N (and noticeable even for large N) are $O(N)$ overheads which are due to software, ie. the DBLAS procedure calls. This is because in an $N \times N$ LU factorization, there are $6N$ calls to the DBLAS procedures, and there are many more still in the QR factorization. Each of these calls in turn call several lower-level procedures (see Figure 1), and also involve `DistMat` and `LocMat` structure setup, parameter passing and manipulation, as well as indexing calculations. It should be noted on a SPARC 1, an integer multiply or divide operation can take up to 50 cycles; the computation of $\sigma(A, i, j)$ alone can involve about 20 such operations, amounting to overhead that is comparable to the AP1000 communication startup time [4].

These overheads are not parallelizable in any way, and thus impact on algorithm scalability. Their extent can be exposed by a careful performance modelling of the matrix factorization computation, which does *not* take into account any $O(N)$ overheads except communication startup: the discrepancy between the model and the measured performance[2] for $N = 512$ was almost 100%! Furthermore, this gap increased further when the DBLAS were re-organized for increased modularity and used the `LocMat` and `DistMat` structures more extensively, as described in Section 2.

This problem represents a dilemma between software engineering advantages outlined in Section 2 and the performance degradation for low to moderate N. While these software overheads can be reduced by various methods, their reduction to more acceptable levels will require a large number of small optimizations to be applied.

6 Conclusions

In this paper, we have identified several key elements required for a high performance, portable distributed BLAS implementation which also allows powerful and almost transparent expression of distributed matrix computations. We have shown how these have been successfully encapsulated in the DBLAS, and that the DBLAS can be used to produce parallel LAPACK algorithms that are simpler than the serial LAPACK.

A key element of this is the data abstraction afforded by the distributed matrix structures which obviates the need to explicitly code multiple variants of parallel LAPACK algorithms due to combinations of transpositions of the matrix operands. This is supported also by a powerful 'spread / reduce' procedural abstraction, which not only simplified the internal DBLAS algorithms, but also avoids the former restriction of having $r = s$ for computations such as parallel LU decomposition.

To support optimal communication performance for the 'spread / reduce' operations, a 'stride' version of ApLib was developed for the Fujitsu AP1000 and AP+ (although for the lat-

[2] However, this discrepancy reduced as N increased.

ter, standard ApLib is almost as fast). For portability to other platforms, a BLACS interface is provided; however limitations of the BLACS imply slightly lower performance than the 'stride ApLib'.

Obtaining portability of cell computation performance firstly required all codes to be written in C (rather than SPARC assembler), and then providing a cell BLAS interface for processors with cell architectures differing from the SPARC 1 and SPARC 10. However, inadequacies of the cell BLAS for handling distributed triangular matrices required careful organization of the high-level DBLAS cell computation procedures in terms of a tunable set of machine-dependent parameters.

A preliminary investigation of performance portability to UltraSparc-based multiprocessors indicates high performance is indeed achievable, but that that it may be difficult to maintain this consistently, due to large and unpredictable effects of second-level cache misses. It also indicates that a large panel width ω will be required for the AP3000 for matrix factorization algorithms, indicating that the distributed panels technique supported by the DBLAS will afford an important optimization here.

This technique has been applied to QR factorization on the AP1000 and LU factorization on the AP+. Compared to LU factorization on the AP1000, the performance gains were only about half, ie. 10-15%, due to the fact that in both cases the optimal value of $\omega = 32$ instead of 64. These examples also revealed performance problems due to software overheads in the DBLAS, and show that it is not sufficient in a distributed BLAS to have good performance only on large (level 3) computations only. Investigating how to reduce these software overheads, which can dominate the performance of small (Level 1 and 2) computations, we leave for future work.

References

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

[2] J. Choi, J. Demmel, J. Dhillon, J. Dongarra, S. Oustrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. Technical Report CS-95-283, Computer Science Dept, University of Tennessee, Knoxville, 1995.

[3] J. Choi, J. Dongarra, S. Oustrouchov, A. Petitet, and D. Walker amd R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra SubPrograms. Technical Report CS-95-292, Computer Science Dept, University of Tennessee, Knoxville, 1995.

[4] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato. An Architecture of Highly Parallel Computer AP1000. In *IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, pages 13–16, May 1991.

[5] P.E. Strazdins. Prototyping Parallel LAPACK using Block-Cyclic Distributed BLAS. In *Third Parallel Computing Workshop for the Fujitsu PCRF*, pages P1–R–1 – P1–R–7, Kawasaki, November 1994.

[6] 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.

[7] R. A. van de Geigin. Efficient Global Combine Operations. In *Sixth Doistributed Memory Computing Conference*, pages 291–4. IEEE Computer Society Press, 1991.

[8] B.B. Zhou and R.P. Brent. Parallel Implementation of QRD Algorithms on the Fujitsu AP1000. In *Proceedings of 17th Australian Computer Science Conference*, pages 733–743, Christchurch, January 1994.