DISTRIBUTED BLAS

Description of Function

AUSTRALIAN NATIONAL UNIVERSITY

Third Edition : May 1997

Contents

a_{00}	a_{01}	a_{06}	a_{07}	a_{02}	a_{03}	a_{08}	a_{09}	a_{04}	a_{05}
a_{10}	a_{11}	a_{16}	a_{17}	a_{12}	a_{13}	a_{18}	a_{19}	a_{14}	a_{15}
a_{40}	a_{41}	a_{46}	a_{47}	a_{42}	a_{43}	a_{48}	a_{49}	a_{44}	a_{45}
a_{50}	a_{51}	a_{56}	a_{57}	a_{52}	a_{53}	a_{58}	a_{59}	a_{54}	a_{55}
a_{80}	a_{81}	a_{86}	a_{87}	a_{82}	a_{83}	a_{88}	a_{89}	a_{84}	a_{85}
a_{90}	a_{91}	a_{96}	a_{97}	a_{92}	a_{93}	a_{98}	a_{99}	a_{94}	a_{95}
a_{20}	a_{21}	a_{26}	a_{27}	a_{22}	a_{23}	a_{28}	a_{29}	a_{24}	a_{25}
a_{30}	a_{31}	a_{36}	a_{37}	a_{32}	a_{33}	a_{38}	a_{39}	a_{34}	a_{35}
a_{60}	a_{61}	a_{66}	a_{67}	a_{62}	a_{63}	a_{68}	a_{69}	a_{64}	a_{65}
a_{70}	a_{71}	a_{76}	a_{77}	a_{72}	a_{73}	a_{78}	a_{79}	a_{74}	a_{75}

Figure 1: A 10×10 block-cyclic distributed matrix A with $B_0 = B_1 = 2$ $P0_0 = P0_1 = B0_0 = B0_1 = 0$ on a 2×3 processor grid.

1 Introduction

Libraries of linear algebra routines such as LINPACK and LAPACK have been implemented using certain basic linear algebra sub-programs (BLAS) as primitives. The motivation for this is a combination of portability and efficiency – the high-level routines are written in a machine-independent manner in a widely-available language (usually Fortran 77), but the BLAS may be coded in a machine-dependent manner in a high or low-level language. When porting the linear algebra libraries to a new machine, it is easy to get them running by using a portable (but possibly inefficient) implementation of the BLAS. Higher efficiency can then be obtained by recoding the BLAS to take better advantage of the machine architecture.

The *BLAS Level 3* (BLAS-3) [4] implement matrix-matrix operations, which, for n by n, matrices, involve $O(n^3)$ arithmetic operations on $O(n^2)$ data items. This yields is a higher ratio of arithmetic operations to data access, and hence the BLAS-3 routines are potentially efficient on processors with a memory hierarchy. In particular, they are potentially efficient on distributed memory machines, which, as in the case of the AP1000 and AP+, have four levels of memory hierarchy: registers, cache, cell memory and external processor memory.

However, on distributed memory machines, which are usually programmed in Single Program Multiple Data (SPMD) mode, more information is required to specify a distributed matrix-matrix operation than is needed for the corresponding non-distributed operation. For example, the ids of the cell containing the leading element of each distributed matrix must be given. Thus, distributed BLAS requires an extended interface to the non-distributed BLAS.

Our approach to the Distributed BLAS, called the DBLAS, uses the rectangular $B_0 \times B_1$ block-cyclic matrix distribution [7, 9].

An $M \times N$ distributed matrix (A) on a $P_0 \times P_1$ processor grid can then be specified by:

$$A = (t, (\mathtt{Ax}, I_{0..1}), D_{0..1}) \quad \text{where} \;\; D_i = (B0_i, B_i, P0_i, P_i)$$

Here `Ax` is the address of the leading element of the local part of the array, I_0 (I_1) are the row (column) stride of `Ax`[1]. (B_0, B_1) is the offset of the leading block of the cell; thus the leading block is of size $(B_0 - B0_0) \times (B_1 - B0_1)$. $(P0_0, P0_1)$ is the processor offset of the current cell relative to the leading cell (ie. the cell holding the leading block of A, which has absolute co-ordinates $((P_0 + \mathtt{getcidy}() - P0_0) \bmod P_0, (P_1 + \mathtt{getcidx}() - P0_1) \bmod P_1)$).

With the block-cyclic matrix distribution, block (i_b, j_b) on cell $((P0_0 + i_B) \mathrm{mod} P_0, (P0_1 + j_b) \mathrm{mod} P_1)$, as illustrated on Figure 1.

The transpose bit $t \in \{0, 1\}$ indicates the relation between the global dimension sizes and the block-cyclic matrix dimension specifiers, ie. M (N) corresponds to D_t (D_{1-t}). If the transposition

[1] $I_0 = 1$ ($I_1 = 1$) corresponds to column- (row) major storage.

specifier $t = 1$, the matrix A is regarded as 'transposed'. The convention chosen will be that computations corresponding to the 'Left' side will be regarded as 'normal'; to express a BLAS computation corresponding to the 'Right' side, the transpose fields of all matrix operands should be reversed. The notation A^T $(= \texttt{Trans}(A))$ signifies the toggling of the t field of A.

In a DBLAS operation, the leading elements of the distributed matrices may be on different cells, and the DBLAS routines themselves are responsible for the 'on-the-fly' alignment of all data, including matrix transposition, during the computation.

The DBLAS user interface corresponds to that of the ordinary BLAS, except that it requires the matrix operands, eg. A, to be represented by the `DistMat` Abstract data Type (ADT). Matrix sizes (eg. M, N) are global. Information in the ordinary BLAS interface that is subsumed by the `DistMat` ADT no longer needs to be specified, ie. matrix strides and transposition and side specifiers. Once a distributed matrix A is set up, the interface is thus simpler than that of the ordinary BLAS. The DBLAS procedure names have a 'D' prefixed to the name of their ordinary BLAS counterparts.

2 Functionality of the DBLAS

The C language interface supports block-cyclic matrices with either row or column major storage. Only double precision matrices are currently supported. The DBLAS have also a Fortran interface, which has identical calling sequences and semantics to (version 1.2 of) the Parallel BLAS (PBLAS), which is part of the ScaLAPACK package produced by University of Tennessee, Knoxville, and is available on the public domain via netlib [2]. For details on this interface, see [3]. Note however that the alignment constraints of Section 2.2 apply to the PBLAS interface to the DBLAS.

Let {`REGD`} denote the root directory for the DBLAS software. All internal modules within the DBLAS may be called by client programs; see {`REGD`}`/src/*.h` for how to use these, where {`REGD`} is the installation directory for the DBLAS. In particular, the matrix spread and reduce operations are available, and can be used directly to avoid redundant communications in parallel matrix computations (see {`REGD`}`/src/SpreadMatrix.h`).

2.1 Manipulating Distributed Matrix structures via the `DistMat` ADT

While block-cyclic distributed matrices are by nature complex structures, judicious use of the `DistMat` ADT can make their manipulation simpler. Indeed, it is strongly inadvisable to manipulate them in any other way than by the ADT operations described below.

For AP/AP+ systems, and other architectures with relatively low communication overheads, the special case of the scattered distribution ($B_0 = B_1 = 1$) will be optimal for most algorithms, as it affords the best load balance [6, 8, 7] and is recommended for such platforms. It also simplifies many issues, particularly that of block alignment (see Section 2.2).

Below, the main ways of using the `DistMat` ADT will be described. For further details, see {`REGD`}`/src/DistMat.h`.

`DistMat` variables are defined to be *pointers* to structures, via the declaration:

$\texttt{DistMat} A, \mathrm{Aij}, B, X, \mathrm{sA};$

When new structures are created, (stack) space must also be allocated for them. Thus, the above declaration should be accompanied by the declaration:

```
DistMat_RVAlloc(5);
```

which declares an array of 5 such structures.

A can be initialized by the calls:

$$A = \texttt{InitDimMat}(t, B_0, B_1, \texttt{cidy}, P_0, \texttt{cidx}, P_1, i_0, j_0); \texttt{LM\_Init}(\texttt{DM\_Loc}(A), \texttt{Ax}, I_0, I_1);$$

which makes A point of one of the (5) structures in the above `DistMat_RVAlloc()` declaration, and initializes the structure accordingly. Here (`cidy`, `cidx`) is the (absolute) cell id of the current cell, and (i_0, j_0) represents a global offset for some super-matrix A' which has a leading cell of absolute co-ordinates $(0,0)$ and a block-offset of $(0,0)$. If A is desired to have leading cell $(\mathtt{Id}_0, \mathtt{Id}_1)$ with block-offset $(B0_0, B0_1)$, we would have $(i_0, j_0) = (\mathtt{Id}_0 B_0 + B0_0, \mathtt{Id}_1 B_1 + B0_1)$.

Allocating the correct amount of workspace, and setting up proper values for the matrix strides $I_{0..1}$ for the block-cyclic matrix distribution can be non-trivial in the general case. A global $M \times N$ matrix A with row-major storage can be more easily initialized by the calls:

$$A = \mathtt{InitDimMat}(t, B_0, B_1, \mathtt{cidy}, P_0, \mathtt{cidx}, P_1, i_0, j_0); \mathtt{NewMat\_}(A, M, N, A, 0);$$

Here the call to `NewMat_()` is similar to that to `NewMat()` described below, except that the storage at $*A$ is used for the result of the operation.

Once a `DistMat` variable is properly initialized, new distributed matrices can be created by the following constructors.

$\mathrm{Aij} = \mathtt{SubMat}(A, i, j);$

returns the sub-matrix of A at global position (i, j), with i (j) being applied to dimension D_t (D_{1-t}). (note: if A is $M \times N$, the constraint $0 \leq i < M, 0 \leq j < N$ should hold).

An $M' \times N'$ matrix B with row-major local storage starting at address W with the same dimension specifiers as A can be created by:

$B = \mathtt{NewMat}(M', N', A, W);$

If $W = 0$, a workspace of an appropriate size (eg. $g(M', D_t) \times g(N', D_{1-t})$, where g is the global-to-local index conversion function) is allocated. In this case, the call $\mathtt{DisposeMat}(B)$; should be made to free this space when it is no longer required.

The call:

$X = \mathtt{ReduceDimMat}(s, A);$

returns a distributed matrix X like A, except that the dimension specifier $D_{r \oplus t} = (0, 1, 0, 1)$ has been 'collapsed' (See Section 2.2). This is useful if X is subsequently passed to a level 1 DBLAS procedure where it is desired to restrict the scope of the cells receiving the return value (and hence the cells that must participate in the call). With $r \oplus t = 0$ $(r \oplus t = 1)$, X would be a row (column) vector and hence the return value would be restricted to the corresponding row (column) of cells; note that the macro `RowVec()` (`ColVec()`) is a shorthand for this case. With the special value $s = -1$, the return value would be restricted to the cell containing the leading element of A.

Finally, the call:

$\mathrm{sA} = \mathtt{Spread2DistMat}(t', r, D, (\mathtt{Ax}, I_{0..1}));$

creates a `DistMat` structure $(t', (\mathtt{Ax}, I_{0..1}), D_{0..1})$ where $D_r = (0, 1, 0, 1)$ and $D_{1-r} = D$. This is useful to convert a workspace $(\mathtt{Ax}, I_0, I_1)$ created by a spread or reduce operation (aligned with dimension D) into the corresponding `DistMat` representation.

The following `DistMat` operations do *not* require allocation of structures:

$\mathtt{DM\_Trans}(A)$	returns t
$\mathtt{DM\_Loc}(A)$	returns $(\mathtt{Ax}, I_0, I_1)$
$\mathtt{DM\_D0}(A)$	returns D_0
$\mathtt{DM\_D1}(A)$	returns D_1
$\mathtt{DM\_DM}(A)$	returns D_t
$\mathtt{DM\_DN}(A)$	returns D_{1-t}
$\mathtt{DM\_SetTrans}(t, A)$	sets transposition bit of A to t'
$\mathtt{Trans}(A)$	returns A^T (ie. transposition bit is toggled)
$\mathtt{DisposeMat}(A)$	free memory space at `Ax`
$\mathtt{InColScope}(A)$	returns whether this cell contains the column vector beginning at A
$\mathtt{InRowScope}(A)$	returns whether this cell contains the row vector beginning at A
$\mathtt{InEltScope}(A)$	returns whether this cell contains the leading element of A

In general, the macro call `DistMat_RVAlloc(`n`)` is used to declare a local structure array of length n within any C syntactic block that had inside it at most n `DistMat` constructor operations. This number would not include operations within a nested block that had its own `DistMat_RVAlloc` declaration. All loops are required have such a declaration (unless $n = 0$). See Figure 3 for examples. The value of n may be over-estimated; if under-estimated, and the code is compiled with the `-DASSERT` option (with `{REGD}/src/FDATAAssert.h` included), execution will abort with an appropriate error message.

2.2 General Constraints on the DBLAS Parameters

The DBLAS procedures require that all matrix sizes (eg. M, N) are non-negative. The (first character of) any triangular matrix upper/lower and diagonal specifiers must be one of `'L', 'U', 'l', 'u'` and `'D', "N, 'd', 'n'` respectively (as for the normal BLAS).

The `DistMat` operands must be internally consistent, ie.

$$t \in \{0,1\}, 0 \leq B0_i < B_i, 0 \leq P0_i < P_i, \quad \text{for } i = 0,1$$

With the exception of the BLACS-compiled versions of the DBLAS, $P_0 \in \{1, \texttt{getncely}()\}$ and $P_1 \in \{1, \texttt{getncelx}()\}$. If the DistMat ADT is used properly, these conditions will always be met. Currently, the smallest of the local matrix strides must be 1, ie. $\min(I_0, I_1) = 1$.

If A is an $N_0 \times N_1$ matrix, the largest matrix stride must also be consistent with the corresponding global matrix size, ie. $I_{i_m} \geq g(N_{1-i_m}, D_{1-i_m})$, where $\max(I_0, I_1) = I_{i_m}$. Here, g is the global-to-local length conversion function. Currently, the value of i_m should be consistent for A across all cells, but the value of I_{i_m} need not be. The t bit of each matrix operand must also be consistent across all cells.

The `Ax` field can be `NULL`, but only on cells where one of the local matrix sizes is 0, ie. $\min(g(N_0, D_0), g(N_1, D_1)) = 0$.

The Distributed BLAS procedures perform BLAS operations on the respective global matrices provided (a given task on) all cells are simultaneously executing (in SPMD mode) the corresponding DBLAS procedure.

In all DBLAS procedures, matrix size, shape, and scaling specifiers (eg. M, `*UpLo`, `alpha`) refer to the corresponding global computation and must have identical values across all cells.

The $B0$, and B and P dimension sub-fields of A must similarly be the same across all cells. The $P0$ sub-fields must be consistent in that all cells will derive from them the same absolute cell id of the leading cell.

As well as checking the internal validity of each `DistMat` operand, the DBLAS procedures also checks between `DistMat` operands in the same call. For example, consider the computation $C = AB$ where A is $M \times K$ and B is $K \times N$. Firstly, the cell dimension specifiers $P_{0..1}$ within each of A, B and C must be the same, with the exception that $P_i = 1$ is allowed on the input-only operands (in this case A or B).

Secondly, with the matrix dimensions (eg. M, N, K), the DBLAS procedures require alignment in at least all but one of the corresponding pairs of dimension specifiers. In the case of level 1 and 2 DBLAS, the non-alignment can only occur on the trivial matrix dimensions. For example, in the case of non-transposed A, B and C, D_0 of A and C correspond to the M-dimension, D_1 of A and D_0 of B correspond in the K-dimension, and D_1 of B and C correspond to the N-dimension. 'Alignment' means that the block sizes and offsets, as well as the processor sizes, are the same.

The input matrices (only), eg. A or B, may be 'pre-spread' across either of their matrix dimensions. This corresponds to a vector replicated across cell rows or columns, except that it can be a proper matrix. This means that each cell holds the complete portion of the corresponding global index for that matrix, and the corresponding dimension specifier is given by $D = (0, 1, 0, 1)$.

The DBLAS procedures check the validity of each of its parameters on each cell (checking of corresponding parameters across cells is not implemented); if one is found invalid, a message is written to the cell's standard output, indicating the procedure's name, the number of the 1st parameter from which an error was found, and the values of all parameters. Where the error is with respect to several parameters (such as an alignment restriction is not met), the numbers of the associated parameters are also given. Note that symmetric matrices must have the same block-alignment along both matrix dimensions. Program execution on this cell is then aborted.

2.3 Synopsis of the DBLAS procedures

Below, a synopsis of the double precision DBLAS is given. α denotes the value of `alpha`, β denotes the value of `beta`, The parameters are of types:

```
char *UpLo, *Diag;
int   M, N, K;
double alpha, beta;
DistMat A, B, C /*matrices*/; DistMat X, Y /*vectors*/;
```

The Level 3 subroutines do not use side or transposition specifiers. Instead, such functionality is gained by setting the transposition bits of any of the `DistMat` parameters. 'A – $M \times N$' means that A is of length M along processor dimension t, and length N across processor dimension $1 - t$.

The `UpLo` and `Diag` specifiers refer to the (single) triangular matrix operand of the call. Note that symmetric matrices are stored as triangular matrices. The `*Diag` specifier being `'U'` (Unit) means the diagonal entries of the matrices are assumed to be unity and hence are not accessed. The `*UpLo` specifier being `'U'` (Upper) means the matrix is stored as a distributed upper triangular matrix; otherwise, it is stored as a distributed lower triangular matrix.

The meaning of the `*UpLo` specifier thus interacts with the transposition bit of the corresponding matrix operand. The `UpLo` specifier needs to be reversed (the reversed value will be denoted `UpLo'`) if the transposition bit of the corresponding matrix is set, ie. whether a triangular matrix is regarded as `Upper` or `Lower` depends on its layout over the 2-dimensional processor grid[2].

The level 3 subroutines will now be described.

`DDGEMM(M, N, K, alpha, A, B, beta, C)`

performs $C \leftarrow \alpha AB + \beta C$, with A – $M \times K$, C – $M \times N$.

Note: to get the equivalent of $C \leftarrow \alpha A'^T B + \beta C$, where A' is $K \times M$, use `DDGEMM(M, N, K, alpha, Trans(A'), B, beta, C)`.

`DDSYMM(UpLo, M, N, alpha, A, B, beta, C)`

performs $C \leftarrow \alpha AB + \beta C$, with A – $M \times M$ (symmetric), C – $M \times N$.

Note: to perform the equivalent of the 'Right' version of this computation, ie. $C \leftarrow \alpha BA + \beta C$ with A – $N \times N$, use: `DDSYMM(UpLo, N, M, alpha, A, Trans(B), beta, Trans(C))`.

`DDSYRK(UpLo, N, K, alpha, A, beta, C)`

performs $C \leftarrow \alpha AA^T + \beta C$, with A – $N \times K$. C – $N \times N$ (symmetric).

Note: to perform the equivalent of the 'Transpose' version of this computation, ie. $C \leftarrow \alpha (A')^T A' + \beta C$, use: `DDSYRK(UpLo, N, K, alpha, Trans(A'), beta, C)`.

`DDSYR2K(UpLo, N, K, alpha, A, B, beta, C)`

[2] This is only the current proposal of the DBLAS interface. It is motivated by the useful property that merely by setting the t bit in A, DBLAS-based LU, LLT and QR algorithms (see Section 4.1) will perform the 'mirror image' algorithms, ie. UL, UUT and LQ factorization. Feedback from researchers and users is welcome on this or any other aspect of the DBLAS interface, and may well be taken into account in future releases.

performs $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$, with A – $K \times N$, C – $N \times N$ (symmetric).

Note: to perform the equivalent of the 'Transpose' version of this computation, ie. $C \leftarrow \alpha(A')^T B' + (B')^T A' + \beta C$, use: `DDSYR2K(UpLo, N, K, alpha, Trans(A'), Trans(B'), beta, C)`.

```
DDTRMM(UpLo, Diag, M, N, alpha, A, B)
```

performs $B \leftarrow \alpha AB$, with B – $M \times N$ and A – $M \times M$ (triangular).

Note: to perform the equivalent of the 'Right' version of this computation, ie.$B \leftarrow \alpha BA$ with A – $N \times N$, use: `DDTRMM( UpLo', Diag, N, M, alpha, A, B)`.

```
DDTRSM(UpLo, DIAG, M, N, alpha, A, B)
```

performs $B \leftarrow \alpha A^{-1}B$, with B – $M \times N$ and A – $M \times M$ (triangular).

Note: to perform the equivalent of the 'Right' version of this computation, ie.$B \leftarrow \alpha BA^{-1}$ with A – $N \times N$, use: `DDTRSM( UpLo', Diag, N, M, alpha, A, B)`.

In the DBLAS, only row and column vectors are supported; thus this is not as general a representation of vectors as possible in the ordinary BLAS with the `INCX` parameters.

In the Level 2 and 1 procedures, vector operands are taken to be column vectors, if their transposition bit is not set, and row vectors, otherwise. The Level 2 routines are defined as follows.

```
DDGEMV(M, N, alpha, A, X, beta, Y)
```

performs $Y \leftarrow \alpha AX + \beta Y$, with A – $M \times N$, X – $N \times 1$, Y – $M \times 1$.

Note: to perform the equivalent of the 'Transposition' variant of this computation, ie. $Y \leftarrow \alpha A'X + \beta Y$ with A' – $M \times N$ but X – $N \times 1$, Y – $M \times 1$, use: `DDGEMV(N, M, alpha, Trans(A'), X, beta, Y)`.

```
DDSYMV(UpLo, N, alpha, A, X, beta, Y)
```

performs $Y \leftarrow \alpha AX + \beta Y$, with A – $N \times N$ (symmetric), X – $N \times 1$, Y – $M \times 1$.

```
DDTRMV(UpLo, Diag, N, alpha, A, X)
```

performs $X \leftarrow \alpha AX$, with A – $N \times N$ (triangular), X – $N \times 1$.

```
DDTRSV(UpLo, Diag, N, alpha, A, X)
```

performs $X \leftarrow \alpha A^{-1}X$, with A – $N \times N$ (triangular), X – $N \times 1$.

```
DDGER(M, N, alpha, X, Y, A)
```

performs $A \leftarrow \alpha XY^T + A$, with X – $M \times 1$, Y – $N \times 1$, A – $M \times N$.

```
DDSYR(UpLo, N, alpha, X, A)
```

performs $A \leftarrow \alpha XX^T + A$, with X – $N \times 1$, A – $N \times N$ (symmetric).

Note: to perform the equivalent operation on a $1 \times N$ row vector X', use: `DDSYR(UpLo, N, alpha, Trans(X'), A)`.

```
DDSYR2(UpLo, N, alpha, X, Y, A)
```

performs $A \leftarrow \alpha XY^T + \alpha YX^T + A$, with X – $N \times 1$, A – $N \times N$ (symmetric). Currently, the transposition bits of X and Y must be the same.

Note: to perform the equivalent operation on a $1 \times N$ row vectors Y', X', use: `DDSYR2(UpLo, N, alpha, Trans(X'), Trans(Y'), A)`.

In the Level 1 procedures which have return values, the 'scope' of the return value is the $P' \times Q'$ sub-grid of the vector operands, ie. P' (Q') is the processor size in the dimension specifier D_0 (D_1). All cells in this scope must participate in the call[3]. The Level 1 procedures are defined as follows:

```
DDSWAP(N, X, Y)
```

performs $X \leftrightarrow Y$, with X, Y – $N \times 1$. Currently, the transposition bits of X and Y must be the same.

```
DDSCAL(N, alpha, X)
```

performs $X \leftarrow \alpha X$, with X – $N \times 1$.

```
DDCOPY(N, X, Y)
```

performs $Y \leftarrow X$, with X, Y – $N \times 1$.

```
DDAXPY(N, alpha, X, Y)
```

performs $Y \leftarrow \alpha X + Y$, with X, Y – $N \times 1$.

```
(double) DDOT(N, X, Y)
```

returns $X^T Y$, with X, Y – $N \times 1$. Currently, the transposition bits of X and Y must be the same.

```
(double) DDNRM(N, X)
```

returns $||X||_2$, with X – $N \times 1$.

```
(double) DDASUM(N, X)
```

returns $\Sigma_{i=0}^{N-1} |X_i|$, with X – $N \times 1$.

```
(int) DDAMAX(N, X, (double *) AMAX)
```

returns i such that $0 \leq i < N$ and $\forall_{j=0}^{N-1} : |X_i| \geq |X_j|$ and $*$`AMAX` $= X_i$, with X – $N \times 1$.

There are also the following miscellaneous procedures:

```
DDScaleAdd(M, N, alpha, A, beta, B)
```

performs $C \leftarrow \alpha A + \beta C$, where A, C – $M \times N$. Note: if the transpose bits of A and C differ, this will involve a (parallel) transposition operation.

```
(double) DDGetElt(A)
```

returns the leading element $A_{0,0}$ (to all processors in the 'scope' of A).

```
(double) DDSetElt(A, alpha)
```

performs $A_{0,0} \leftarrow \alpha$. Only the value of alpha on the processor containing $A_{0,0}$ will be used.

[3]This convention is slightly different to that of the PBLAS [3].

3 Implementation

The (Level 2 and 3) DBLAS procedures all partition the computation across the non-aligned matrix dimension. If all dimensions are aligned, the smallest is chosen. Where possible, the partitioning is in units large enough to achieve good cell computation speed (see {REGD}`/src/MachineParams.h`).

'Spread' ('reduce') operations [9] are then applied to the input (output) matrix operands which have a dimension specifier corresponding to this dimension. These operations avoid unnecessary communication in the case where the operation involves only a single row or column of processors. These also use, where possible, the workspace of the matrix to be operated on, where that matrix is a single row or column of blocks. The advantages of this approach are modularity, reduced code size and optimal communication performance (provided messaging has a 'stride' capability for no extra cost). A further advantage is that one matrix dimension need not be aligned. This enables for example the DBLAS to implement LU decomposition for non-square blocks.

Workspaces are dynamically allocated from heap memory. Partitioning and if needed synchronization is also performed on messaging so that in systems where incoming messages have to be stored in system buffers, the size of unread messages should be not exceed 2*`OPT_MSG_SIZE`. The AP/AP+ ring buffer size should thus be set to at least this value. If the DBLAS procedures are compiled with the `-DTUNE` option, all `MachineParams.` constants may be set dynamically.

The DBLAS has a 'deep' software architecture (see Figure 2), and uses generalized matrix procedures with long parameter lists. Both the outer-level C and Fortran interface procedures perform error checking, and select the appropriate parallel 'driver' routine. These in turn use spread/reduce operations for communication (and use generalized BLAS calls for computation). The spread/reduce operations performs buffer allocation and in turn call procedures from a high-level matrix communication library called `FDATADistMatrix`, having routines for matrix rotation, transposition, multi-broadcast, multi-sum etc. These routines in turn call procedures from the `FDATACommun` library, which finally provides an interface to either (various versions of) the `ApLib` library or the BLACS (Basic Linear Algebra Communication Subroutines [3]).

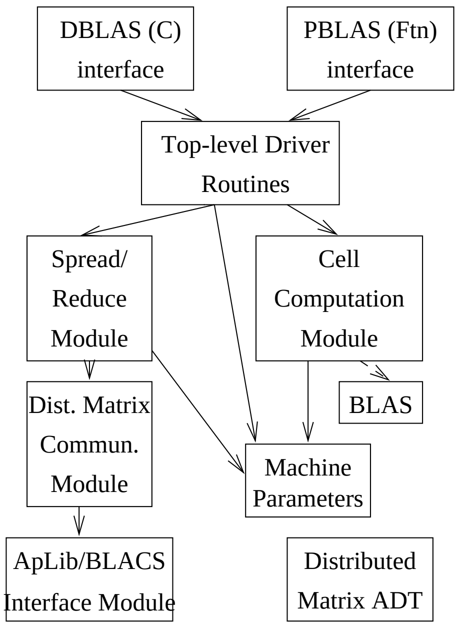

Figure 2: Overall Architecture of the DBLAS Library

Cell computation involving triangular block-cyclic distributed matrices can be complex, and indeed ordinary BLAS cannot be directly used for this purpose. A cell's local portion of a triangular matrix is partitioned into strips of width ω'; a rectangular (BLAS) matrix operation is performed on the rectangular part of this strip, and ω' vector operations are used to complete the computation on the remainder. $\omega' = 4$ on the AP1000/AP+. The implementation of `DDSY2(K)` also tries to take advantage of the rank-2 operation by 'merging' the workspaces and performing a local matrix multiply corresponding to $(A|B)\left(\frac{B^T}{A^T}\right)$.

4 Performance And Evaluation

The computational routines used by the DBLAS have been highly tuned for Sparc 1 and Sparc 10 processor architectures. The outer-level DBLAS routines are generally close to optimal with respect to the number of communication startups, the volume of communication, memory copying and software overheads. Exceptions occur with the 'standard ApLib' implementation (which uses extra communication startups to avoid memory copying) and the BLACS implementation (which performs extra memory copying to avoid extra communication startups). The performance of the DBLAS, whether in a single routine benchmark, or in an application such as a matrix factorization algorithm, should be as good or better than in previously reported results [1, 7, 8, 9].

The only case where the current DBLAS algorithms will be non-optimal is for the symmetric matrix routines, where the symmetric matrix is much smaller than the other matrix operands.

4.1 Example: LU Decomposition using the DBLAS

In [1], 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 matrices on the limit of the AP1000's cell memory (ie. $N/P_i = 8192$).

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 (having leading element in cell $(0,0)$) is overwritten $PA = LU$, where P is some row permutation matrix. The cell program can be succinctly expressed as shown in Figure 3.

The performance of the DBLAS-based LU algorithm compares well with that the hand-crafted algorithm. It is shown, together with that of Cholesky (LLT) and QR factorization algorithms, which are also based entirely on the DBLAS, in Figure 4.

5 Contact

When users have questions about the DBLAS, or when users encounter difficulties in using the DBLAS please, contact:
Peter Strazdins, Department of Computing Science, Australian National University
E-mail: Peter.Strazdins@cs.anu.edu.au

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.

```
void DDGETF2(int M, int N, DistMat A, int P[]){
   int j; double A_max;
   for (j = 0; j < min(M, N); j++){
      DistMat_RVAlloc(6);
      P[j] = j + DDAMAX(M-j, SubMat(A, j, j), &A_max);
      DDSWAP(N, Trans(SubMat(A, j, 0)), Trans(SubMat(A, P[j], 0)));
      DDSCAL(M-j-1, 1.0/A_max, SubMat(A, j+1, j));
      DDGER(M-j-1, N-j-1, -1.0, SubMat(A, j+1, j), Trans(SubMat(A, j, j+1)), SubMat(A, j+1, j+1));
   }
}

void DDGETRF(int M, int N, DistMat A, int P[]){
   int j = 0;
   while (j < min(M, N)){
      DistMat_RVAlloc(6);
      int i, ω = min(64, M - j);
      DDGETF2(M-j, ω, SubMat(A, j, j), &P[j]);
      for (i = j; i < j+ω; i++){
         DistMat_RVAlloc(4);
         P[i] += j; DDSWAP(j, Trans(SubMat(A, i, 0)), Trans(SubMat(A, P[i], 0)));
         DDSWAP(N-j+ω, Trans(SubMat(A, i, j+ω)), Trans(SubMat(A, P[i], j+ω)));
      }
      DDTRSM("L", "U", ω, N-j-ω, 1.0, SubMat(A, j, j), SubMat(A, j, j+ω));
      DDGEMM(M-j-ω, N-j-ω, ω, -1.0, SubMat(A, j+ω, j), SubMat(A, j, j+ω), 1.0, SubMat(A, j+ω, j+ω));
      j += ω;
   }
}
```

Figure 3: DBLAS expression of blocked LU factorization with partial pivoting

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

[5] A. Rendell. Porting ScaLAPACK (and PBLAS) to the Fujitsu AP1000. Technical report, ANU Supercomputing Facility, Australian National University, 1995.

[6] 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.

[7] 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.

[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 *Fifth Parallel Computing Workshop for the Fujitsu PCRF*, pages P2–K–1 – P2–K–10, Kawasaki, November 1996.

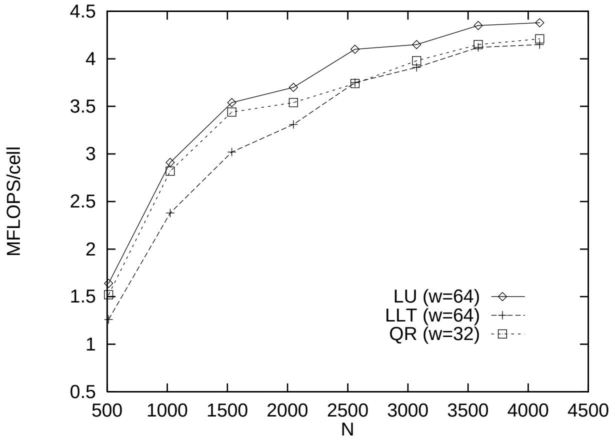

Figure 4: Performance of DBLAS-based matrix factorizations for $N \times N$ row-major matrices of block size 1 and blocking factor ω on an 8×8 AP1000