Basic Matrix Subprograms for Distributed Memory Systems

Anne C. Elster
Cornell University
School of Electrical Engineering
Ithaca, New York 14853

Abstract

Parallel systems are in general complicated to utilize efficiently. As they evolve in complexity, it hence becomes increasingly more important to provide libraries and language features that can spare the users from the knowledge of low-level system details. Our effort in this direction is to develop a set of basic matrix algorithms for distributed memory systems such as the hypercube.

The goal is to be able to provide for distributed memory systems an environment similar to that which the Level-3 Basic Linear Algebra Subprograms (BLAS3) provide for the sequential and shared memory environments. These subprograms facilitate the development of efficient and portable algorithms that are rich in matrix-matrix multiplication, on which major software efforts such as LAPACK have been built.

To demonstrate the concept, some of these Level-3 algorithms are being developed on the Intel iPSC/2 hypercube. Central to this effort is the General Matrix-Matrix Multiplication routine PGEMM. The symmetric and triangular multiplications as well as, rank-2k updates (symmetric case), and the solution of triangular systems with multiple right hand sides, are also discussed.

1 Introduction

The goal of this work is to provide a set of basic "universal" matrix subprograms for the distributed memory environment that would allow programmers to implement algorithms rich in matrix-matrix operations in terms of these basic subprograms. Local communication primitives could hence be hidden in the low-level routines and the new high-level routines not only become easier to implement, but also become portable. This has previously been done with success for serial and vector machines through the Basic Linear Algebra Subprograms (BLAS) [4,3], which among others [6] is based upon.

The high-level algorithms may not provide optimum performance measures, but our goal is to trade, say, 5-10% performance for ease of implementation and *portability*. Previous efforts in the same spirit include the hypercube library developed at Chr. Michelsen in Norway [2] and SCHEDULE, a parallel programming environment for FORTRAN developed at Argonne [7].

To adhere to a familiar standard, we will attempt to follow the Level-3 BLAS (BLAS3) [3] calling sequences as closely as feasible for our distributed memory case. Section 2 describes the BLAS in more detail, whereas the additional parameters needed in the distributed memory setting, follow in Section 3. The core routine, general matrix-matrix multiplication, is described in Section 4. Section 5 discusses the other BLAS routines, rank-2k updates (symmetric case), triangular multiplication, and the solution of triangular systems with multiple right hand sides, respectively. Future work and some of the issues related to the iPSC/2 implementation are mentioned in Section 6. Finally, a summary is given in Section 7.

2 The BLAS

The advantages of defining a set of basic linear algebra routines that higher-level linear algebra algorithms can be built on top of, were originally discussed by Hanson et al. back in 1979 [12]. The subprograms have later evolved through joint efforts by Dongarra et al. [5] The original routines (now dubbed Level-1 routines) limited themselves to vector-vector

0-8186-2113-3/90/0000/0311$01.00 © 1990 IEEE

operations, whereas the Level-2 routines [4] handle vector-matrix operations, and the Level-3 routines [3] explore matrix-matrix operations.

With their low number of data touches (and hence less communication needed) compared with number of arithmetic operations, the problems the BLAS3 cover, prove very suitable for distributed memory computers To follow up on this familiar standard from the sequential and shared-memory world, we have decided to follow the BLAS3 conventions for calling parameters wherever possible.

For example, the **GE**neral **M**atrix-**M**atrix multiplication routine in BLAS3 has the following calling format:

GEMM(TRANSA, TRANSB, M, N, K, α, A, LDA, B, LDB, β, C, LDC),

where TRANSA, TRANSB describes whether **A** or **B** transposed or not; M, N, K, the matrix dimensions; α, β, scalars; LDA, LDB, LDC, leading dimensions of **A, B, C**, respectively. The additional parameters needed in the distributed setting, are appended to the BLAS3 calling sequences.

3 Data Distribution and Other Calling Parameters

In the distributed memory case, extra parameters beyond the ones provided in the BLAS are needed for specifying items such as the topology of the network assumed, the data distribution desired, and possibly also parameters for indexing subclusters of processors. These parameters open up endless choices. We will, however, restrict ourselves to some of the most fundamental and useful ones. Many more options may be desirable, but too many choices defeat the purpose of having a few "core" routines that manufacturers may be willing to supply. It is the hope that sometime in the future the ideas behind these routines not only provide a standard for parallel library builders, but that optimized routines also become standard parts of future languages or operating system kernels.

Our data distribution choices are: block-submatrix, block-vector, and wrap-block-vector. Block-submatrix distributions facilitates *orthogonal tree structures* [9,8] which may be introduced to minimize communications costs compared to the more conventional distributed hypercube algorithms [16, 13]. The orthogonal structures also makes virtual transposes feasible.

The block-vector structure also maps well to hypercubes and meshes (through ring structures) and is the most common distribution of data in numerical problems. Since the individual vectors remain undistributed, it is easier to keep track of the data when doing vector oriented operations.

Finally, the wrap-block-vector mapping is considered since it provides superior load balancing in for several numerical algorithms [10,14]. As the standard block-vector approach, it is implemented using a ring structure. The extra parallel parameters (*input-distr*, *output-distr*, and *network*), will be added to the end of the parameter list, and the routines renamed with a **P** for **P**arallel in front of the BLAS3 name (e.g. PGEMM, for standard general matrix-matrix multiplication).

The most common and useful network topologies include hypercubes, grids (including torus), rings, and trees. This list may, however, be extended as novel architectures take on other topologies. This parameter is, perhaps, the only one that *has* to be modified when porting code between different architectures. Efficiency of the code will, however, be somewhat linked to the data structures (though the communication bandwidth of the system is probably of more importance). For instance, true ring topologies do not emulate grid structures, broadcast, and gather, as efficiently as, say, hypercubes.

Useful communication structures include rings, trees, and meshes. Rings are commonly used in numerical algorithms where operations are performed on block-vectors. They may be embedded on a hypercube network using all nodes by numbering the processors according to 1-D Gray codes. [17,1,9]. The Gray codes ensure that processors that are next to each other in the ring structure also achieves near-neighbor communication between physical hypercube nodes. This embedding also includes a spanning tree (Figure 1).

Meshes (including toroidal connections) may similarly be embedded on hypercubes using 2-D Gray codes. This embedding includes a set of orthogonal tree structures [9]. Whereas tree structures provide efficient structures for broadcast and gather

(both processor-row/column-wise and network-wide), grids – perhaps the most common parallel network – are well-suited for block-submatrix data distributions which are common in applications such as image ans seismic processing.

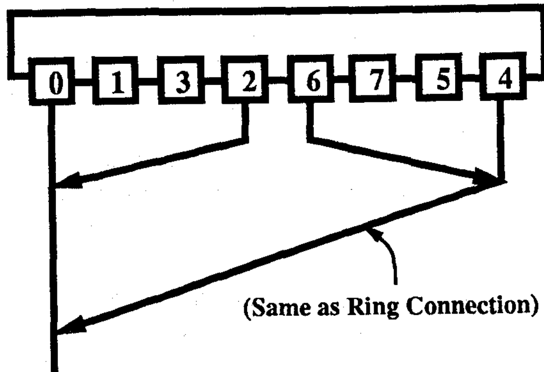

Figure 1. Ring embedding hypercube using binary reflective Gray code. Tree structure for broadcast/gather also shown.

4 Parallel Matrix-Matrix Multiplication

The general matrix-matrix multiplication routines (GEMMs) are the core of the BLAS-3 library. For real matrices **A**, **B**, and **C** (α and β are scalars), the operations can be described as follows:

$\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$, where where **A** and/or **B** may be transposed.

The scalar multiplication (α, β) may simply be performed by broadcasting the scalar value(s) to each node and then perform the scaling locally. We shall, however, assume that the scalars α and $\beta = 1$ in our discussion for simplicity. Their computation will also not be affected by the data distribution (block-column or submatrix-block). A discussion of the matrix product **AB** follows. Note, that the matrix addition included in the matrix operations above, gets performed along with the multiplication as the result gets added in during the accumulation of the summation of $\sum a_{ik}b_{kj}$.

Of the four BLAS permutations allowed through the TRANSA and TRANSB options (see last section), we will first take a closer look at the AB case. On a torus, computing the the products $a_{ik}b_{kj}$ using block-submatrix distributions, can be achieved by rotating the distributed B matrix east-west through the processor plane as the appropriate data reaches the processors. Orthogonal structures may then be used to gather the summations. These structures will also be used for the A^TB^T, A^TB and AB^T cases.

Similarly, in a ring setting, whole block-vectors are rotated left-right on a ring instead of the submatrices for a mesh in the summation phase. Notice that the ring structure mapping also provides a binary tree structure when implemented using the binary reflective Gray code. This allows for efficient broadcasts and gatherings of data.

For A^TB and AB^T, which matrix (A or B) to rotate through the processors in order to avoid stride problems, depends on the storage convention of the matrices. Finally, in the A^TB^T case, the data needs to be "transposed" in order to compute the products $a_{ik}b_{kj}$. The data may also need to be reordered locally to avoid stride problems.

If one of the matrices **A** or **B** is *symmetric,* then, either $\mathbf{A} = \mathbf{A}^T$ or $\mathbf{B} = \mathbf{B}^T$. These cases can hence be viewed as the $\mathbf{A}^T\mathbf{B}$ and $\mathbf{AB}^T$ cases described in the previous section. We are here assuming that compressing the storage of symmetric matrices is not worth while a the distributed memory setting. Although more costly in storage, the cost in increased algorithmic complexity seems to outweight the benefits. Also, in the block-submatrix case one would not be able to take full advantage of the orthogonality of the hypercube structure if storage compression is used.

5 Other BLAS Routines

Following a brief discussion of how the distributed memory setting will affect the rest of the BLAS routines.

5.1 Rank-$2k$ Updates of a Symmetric Matrix

We here consider the following updates of a symmetric matrix **C** covered by the BLAS3:

$$\mathbf{C} \leftarrow \alpha\mathbf{A}\mathbf{B}^{\mathbf{T}} + \alpha\mathbf{B}\mathbf{A}^{\mathbf{T}} + \beta\mathbf{C}$$
$$\mathbf{C} \leftarrow \alpha\mathbf{A}^{\mathbf{T}}\mathbf{B} + \alpha\mathbf{B}^{\mathbf{T}}\mathbf{A} + \beta\mathbf{C}$$

The Rank-1k cases are covered by the general matrix-matrix multiplication routines. In the case of the rank-2k updates of symmetric matrices, all matrix-matrix products are of the form A^TB or AB^T, which, as mentioned, does not require transpositions. Notice that since $AB^T = (BA^T)^T$ and $A^TB = (B^TA)^T$, only one of the products needs to be computed and the remainder of the computation reduces to matrix additions (with scaling with α and β).

5.2 Triangular Matrix Multiplication

We here consider permutations of multiplying the dense matrix $\mathbf{B}$ with a triangular matrix $\mathbf{T}$:

$\mathbf{B} \leftarrow \alpha\mathbf{T}\mathbf{B}$, where T and/or B may be transposed.

Notice that if one considers $\mathbf{T}$ upper triangular, then the $\mathbf{T}^{\mathbf{T}}$ case would represent the lower triangular case and vice versa.

Triangular matrix multiplication may easily be performed redistributing only data from $\mathbf{T}$. In the ring/block-vector case, the two first multiplications ($\mathbf{TB}$ and $\mathbf{T}^{\mathbf{T}}\mathbf{B}$) may be computed rather straightforwardly (with respect to communication) since the $\mathbf{B}$ matrix already is distributed in the same block-column fashion as used for the general multiplication case. For $\mathbf{BT}$ and $\mathbf{BT}^{\mathbf{T}}$, however, the matrix $\mathbf{B}$ is distributed in a block-column fashion whereas the general method assumes a row-wise access. In this case, redistributing $\mathbf{T}$ would hence not be sufficient.

5.3 Triangular Systems with Multiple Right-Hand Sides

In this section, orthogonal data structures are introduced in the context of solving some basic linear systems. First, triangular systems with multiple right-hand sides will be considered:

$$\mathbf{B} \leftarrow \alpha\mathbf{T}^{-\mathbf{1}}\mathbf{B}$$
$$\mathbf{B} \leftarrow \alpha\mathbf{T}^{-\mathbf{T}}\mathbf{B}$$
$$\mathbf{B} \leftarrow \alpha\mathbf{B}\mathbf{T}^{-\mathbf{1}}$$
$$\mathbf{B} \leftarrow \alpha\mathbf{B}\mathbf{T}^{-\mathbf{T}}$$

Here α is a scalar, $\mathbf{B} \in \Re^{m\times n}$, and $\mathbf{T}$ a non-singular triangular matrix. Notice how both the inverse (T^{-1}) and inverse *transpose* (T^{-T}) cases are considered for $\mathbf{T}$ providing both the upper and the lower triangular cases.

Since triangular solves involve either forward or backward substitution (both inherently sequential operation), parallelization is not as straight-forward as in the multiplication cases. However, decent parallelization can be achieved by using a "pipelined" approach, as described by [14], where the data is mapped to a ring structure. In this case, a wrap-block-vector data distribution since it provides a better processor utilization in the factorization stage [10].

It was recently shown that these other BLAS-3 subprograms can actually be implemented in terms of GEMM, at least in the sequential setting [15] with reasonable efficiency. This would be desirable in the parallel setting as well, since it would reduce the machine-depended encodings to that of PGEMM. Futher investigations of this idea are currently under considerations.

6 Some Implementation Issues and Future Work

The Intel iPSC/2 hypercube is currently being used as a test-bed for implementing the PBLAS routines. The ideas behind the routines are not ment to limit themselves to the Intel cube or its topology , but the Intel machine is rather used as test environment for how the PBLAS may be developed for common distributed memory systems. As mentioned, It is our hope that the PBLAS can become a standard for how core matrix algorithms are developed.

We chose to implement our routines in C. Fortan may seem, to many, the most natural language to implement matrix algorithms in. However, C, with its powerful pointer constructs for dynamic memory allocation and strong link to UNIX, is rapidly becoming more popular. Since we wanted to use some of the C pointer features in the implementation, it became a natural choice. C also interfaces well with Fortran and is along with Fortran 77 available in the Intel hypercube. (Fortran 90 may provide similar features, but is yet not available for the Intel cube.)

At the present, PGEMM has been partially imple-

mented. Current work includes completing most of the PGEMM cases (transpose of A and B, various data distributions, etc – see Section 3), benchmarking it, and developing some examples of application. Future work includes the implementation of some of the other PBLAS routines cases (quite probably with call to PGEMM). Since out goal is to demonstrate the concepts rather than provide production code, we will, for now limit ourselves to a couple of core cases rather than provide a full PBLAS implementation for the Intel hypercube (left as future work for people providing production code).

6.1 PGEMM on the hypercube

Taking a closer look at the case $C \leftarrow C + A \cdot B$ in a ring setting, we decided to store the matrices on the nodes in one-dimensional arrays. These are then used directly as message-buffers during the communication phases saving valuable storage space and copy-time. To emulate 2-dimensional array index, index functions where defined: including the leading dimension of the respective matrix (LDX):

Indexing function for A, B and C :
This indexing function assumes matrices stored-by-column starting at array location A[1] (FORTRAN-type).

```
#define AINDEX(X,LDX,i,j) \
  X[(j-1)*LDX + i]
```

The block-column version of $C \leftarrow C + A \cdot B$ can be described by the following equation for block-vector C_i:

$C_i = A_1 B_{1i} + A_2 B_{21} + \ldots + A_p B_{pi}$, for $i, j = 1 : p$,

where p is the number of block-vectors ($n = r \cdot p$, where r is block-width).

Assuming a ring embedding using the Binary Reflective Gray Code [17], the above equation leads to the algorithm:

```
PGEMM

Let me = position in ring
```

(As in the equation above where node i holds C_i and A_i locally. Also reflects which part of the matrices are stored locally.)

Compute on local data:

```
C_me = C_me + A_me * B_me,me
```

Following the MATLAB notation as described in Golub-Van Loan [11]) we here have (all local sub-blocks):

```
C_me = C[1:n,(me-1)r:me * r]
A_me = A[1:n,(me-1)r:me * r]
B_me,me = B[(me-1)r:me * r,1:r]
```

(numnode = no. of nodes in cube (ring))

```
A_tmp = A_me

FOR p = 1 to numnodes
  SEND A_tmp to  left-neighb
  RCV A_tmp from right-neighb
  C_me = C_me + A_tmp*B_x,me
```

(x is a function of *numnodes* and p corresponding to the above equation)

```
Leave result on nodes or SEND to host.

END{PGEMM}
```

Other cases will be described in future work along with a discussion on how to access submatrices (leaving some processor idle rather than redistributing data).

7 Conclusions

In this paper, a basic set of linear algebra algorithms in the spirit of BLAS [4,3] were proposed to form a basis for algorithmic development also in the distributed memory environment.

Extra parameters needed for the parallel environment were identified and added to the BLAS3 calling sequences. These parameters included a parameter to describe the network topology and parameters for specifying the input and output distribution of the data. Powerful communication structures (such as the orthogonal data structures involving trees and meshes) could then be hidden within the routines sparing the users from hardware details.

It is the hope that sometime in the future the ideas behind these routines not only will provide a standard for parallel library builders, but that similar optimized routines also become standard parts of future languages or operating system kernels.

Acknowledgments

The author would like to thank Charles F. Van Loan, her thesis advisor, for his helpful suggestions and support. Thanks are also due the Cornell Theory Center for providing access to their 32-node Intel IPSC/2 hypercube system.

This research is supported in part by the U.S. Army Research Office through the Mathematical Sciences Institute, Cornell University.

References

[1] R.M. Chamberlain. Gray codes, Fast Fourier Transforms and Hypercubes. Technical Report 864502-1, Chr. Michelsen Institute, Bergen, Norway, May 1986.

[2] R.M. Chamberlain, P.O. Frederickson, J. Lindheim, and J. Petersen. A High-Level Library for Hypercubes. *Hypercube Multiprocessors 1987*, pages 651–655, 1987.

[3] J.J. Dongarra, J. Du Croz, I.S. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. Technical Report AERE R 13297, Harwell Laboratory, Oxfordshire, England, October 1988. Also ANL:TM 88 and NaG:TR 14/88.

[4] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. on Mathematical Software*, 14:1–17, 1988.

[5] J.J. Dongarra and S.J. Hammarling. Evolution of Numerical Software for Dense Linear Algebra. In M.G. Cox and S.J. Hammarling, editors, *Advances in Reliable Numerical Computation*, Norfolk, VA. – to be published.

[6] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, 1979.

[7] J.J. Dongarra and D.C. Sorensen. A Portable Environment for Developing Parallel FORTRAN programs. *Parallel Computing*, 5(1&2):175–186, July 1987.

[8] A.C. Elster and H. Li. Hypercube Algorithms on the Polymorphic Torus. In G. Fox, editor, *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications*. AMC, March 1989. – to appear.

[9] A.C. Elster and A.P. Reeves. Block-matrix Operations Using Orthogonal Trees. In G. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 1554–1561, Pasadena, CA, January 1988. ACM. Vol. II.

[10] G.A. Geist and M.T. Heath. Matrix Factorization on a Hypercube Multiprocessor. *Hypercube Multiprocessors 1986*, pages 161–180, 1986.

[11] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins, Baltimore, MD, second edition, 1989.

[12] R. Hanson, D. Kincaid, F. Krogh, and C. Lawson. Basic Linear Algebra Subprograms for Fortan Usage. *ACM Transaction on Math. Software*, 5:153–165, 1979.

[13] S.L. Johnsson. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *Journal of Parallel and Distributed Computing*, 4:133–172, 1987.

[14] G. Li and T.F. Coleman. A New Method for Solving Triangular Systems on a Distributed Memory Message-Passing Multiprocessor. *SIAM Journal of Sci. Stat. Comp.*, 10(2):382–396, March 1989.

[15] C.F. Van Loan and B. Kågström. Poor-Man's BLAS for Shared Memory Systems. preliminary version through personal communications.

[16] O.A. McBryan and E.F. Van de Velde. Matrix and Vector Operations on Hypercube Parallel Processors. *Parallel Computing*, 5(1 & 2):117–125, July 1987.

[17] J. Salmon. Binary Gray Codes and the Mapping of Physical Lattice into a Hypercube. Caltech Concurrent Processor Report Hm-51, Caltech, 1983.