The *Multicomputer Toolbox* Approach to Concurrent BLAS*

Robert D. Falgout
Numerical Mathematics Group
Lawrence Livermore National Laboratory
Livermore, CA 94550

Anthony Skjellum†
Mississippi State University
NSF Engineering Research Center & Computer Science
Mississippi State, MS 39762

Steven G. Smith
Charles H. Still

Numerical Mathematics Group
Lawrence Livermore National Laboratory
Livermore, CA 94550

September 1, 1993

Abstract

Concurrent Basic Linear Algebra Subprograms (CBLAS) are a sensible approach to extending the successful Basic Linear Algebra Subprograms (BLAS) to multicomputers. We describe many of the issues involved in general-purpose CBLAS. Algorithms for dense matrix-vector and matrix-matrix multiplication on general $P \times Q$ logical process grids are presented, and experiments run demonstrating their performance characteristics.

*This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy. Work performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48. Submitted to the *Concurrency: Practice & Experience.*

†Address correspondence to: Mississippi State University, Engineering Research Center, PO Box 6176, Mississippi State, MS 39762. 601-325-8435. *tony@cs.msstate.edu.*

The Multicomputer Toolbox Approach to Concurrent BLAS

1 Overview

We begin here by providing background and motivation, and indicate our approach to the work. We mention that we use the *Toolbox* and *Zipcode* systems, and discuss the notion of three classes of Concurrent Basic Linear Algebra Subprograms (CBLAS). We compare and contrast CBLAS to the three levels of the Basic Linear Algebra Subprograms (BLAS). We also recapitulate essentials of the programming methodology that we have described at length before.

In recent years, the BLAS [7, 5, 4] have become the key kernels for dense matrix computations. Traditionally divided into three levels based on operation complexity, data reuse characteristics, and chronological development, these tools provide a sound, basic functionality for sequential numerical linear algebra. These routines have become widely available, with many vendors supplying them. Thus, there is motivation to develop a CBLAS library that will be both efficient and flexible on distributed memory machines (multicomputers).

Though dense and sparse BLAS have been defined, we develop concurrent generalizations only of dense operations. Dense BLAS are widely accepted with a large quantity of software based on them, whereas both sequential and concurrent sparse operations are a hotly debated subject. Finally, we discuss dense CBLAS in the double precision case, which generalizes immediately to other precisions. We leave sparse CBLAS for future work.

1.1 Approach

There are several steps to creating high performance linear algebra on multicomputers supposed by the CBLAS. The first step is to develop a CBLAS library that will be highly portable and have efficiency based on the quality of available compiler technology, but less optimized than kernels that use specialized vendor features and/or assembly language in critical loops. The second step is for vendors to implement highly optimized routines based on a CBLAS specification. A portable library (reference implementation) would also serve as such a definitive specification for the CBLAS, and as a correctness metric for optimized versions of the library.

Our CBLAS library is based on the *Multicomputer Toolbox* with its *Zipcode* message passing system. *Zipcode* was designed to be a general-purpose message passing system that is easy to implement on any high performance multicomputer (see [10, 8, 11, 9]). Current implementations layer on the *Reactive Kernel* calls. Planned *Zipcode* ports to programming

models like CMMD, PVM, PICL, and Express offer the potential for application-level message passing in both homogeneous and heterogeneous environments, extending *Zipcode* to most multicomputers. Like other message passing systems, *Zipcode* provides the message passing functions needed to perform communication effectively on several node topologies. Unlike other systems, it also interfaces to a grid-oriented description of message-passing as well as a *Toolbox* library of data distributions [8, 11]. By building the CBLAS routines on this platform, we provide data distribution independence, portability, and hardware abstractions. By marrying the CBLAS to general data distributions from the outset, we significantly increase the number of real parallel applications, with varied data requirements, that will fit effectively with these kernels. For these reasons, the *Zipcode*/*Toolbox* combination is a suitable platform for developing the CBLAS. (Later, we expect an easy port to the MPI message-passing standard, replacing *Zipcode*.)

1.2 CBLAS Classes vs. BLAS levels

BLAS are specified in three levels. Level-1 BLAS perform vector-scalar and vector-vector operations, Level-2 BLAS provide matrix-vector and related operations, whereas Level-3 BLAS provide matrix-matrix operations. By analogy, in building the CBLAS, we need to emphasize two types of algebraic objects: sequential and concurrent. We define `Cmatrix` objects as distributed matrices partitioned across logical process grids $\mathcal{G}_{P\times Q}$, and define `Cvector` objects as either row-distributed/column-replicated or column-distributed/row-replicated vectors on such grids. We consider only operations on a single grid at present. We denote vectors and matrices that reside in single processes (with replication across processes) as `Svectors` and `Smatrices`, respectively. With these four objects, we can define three classes of CBLAS. Class-1 CBLAS involve only sequential objects, Class-2 CBLAS involve process-grid-replicated sequential objects and concurrent objects, and Class-3 CBLAS involve concurrent objects only. Classifications are not impacted by the presence or absence of scalars in a function call.

In each CBLAS class, there are routines corresponding to all three levels of BLAS, and we use the BLAS where possible for the sequential tasks within the CBLAS rather than re-implementing loops and depending on compiler optimization. There are several additional functions that should be included in a CBLAS specification, notably data reorganizations such as transpositions of distributed objects. However, in this paper we focus on the Class-3 `GEMV` and `GEMM` operations.

1.3 Scalable Programming

Two key ideas underlying scalable representations of dense linear algebra operations in the *Multicomputer Toolbox* system are as follows: data distribution independence, and the definition of logical process grids. A logical process grid, denoted here by $\mathcal{G}_{P\times Q}$, is a collection of processes (*e.g.*, one process per node of a mesh or hypercube), logically assigned a shape

$P \times Q$. The processes are named (p, q) on this logical grid, where p (q) ranges from $0 \ldots P-1$ (resp, $0 \ldots Q-1$). Such logical grids can be readily mapped to physical node topologies (*cf*, [8]).

Data distribution independence describes the generality of our linear-algebraic algorithms within the *Toolbox*. For instance, we lay out coefficients of matrices on a logical grid according to row and column data distributions. Such distributions include linear and scatter (wrap-mapped) strategies and other strategies that we can write in closed-form (strong data distributions). Algorithms are written to adapt to the data distributions selected by the application programmer, rather than working correctly only for a specific data distribution. Our efforts thus center on providing high performance through tuning of data distributions, whenever possible, between application needs and kernel needs. The logical grid shapes ($P \times Q$) and data distributions that generate peak performance for kernels most often are not the best data layouts for applications meant to utilize such kernels.

2 Data Distributions

In this section, we present the definition of *data distribution*. We also define the notion of *permutation compatibility*, which will be important for defining the data compatibility requirements of many of the CBLAS algorithms to follow.

Definition 1 (Data-Distribution Function) *A data-distribution function μ maps three integers $\mu(I, P, M) \mapsto (p, i)$ where I, $0 \leq I < M$, is the global name of a coefficient, P is the number of processes among which all coefficients are to be partitioned, and M is the total number of coefficients. The pair (p, i) represents the process p ($0 \leq p < P$) and local (process-p) name i of the coefficient ($0 \leq i < \mu^\sharp(p, P, M)$). The inverse distribution function $\mu^{-1}(p, i, P, M) \mapsto I$ transforms the local name i back to the global coefficient name I.*

The "formal" definition of data-distribution may be found in either [8] or [9].

Definition 2 (Permutation Compatibility) *Two distributions $\mu(\bullet, P, M)$ and $\nu(\bullet, Q, N)$ are permutation compatible if and only if $M = N$ and*

$$\begin{aligned} \Lambda_\mu^{-1}(\mu(I, P, M), P, M) &= \Lambda_\nu^{-1}(\nu(I, Q, M), Q, M) \\ &\quad \forall I = 0, \ldots, M-1 \end{aligned} \tag{1}$$

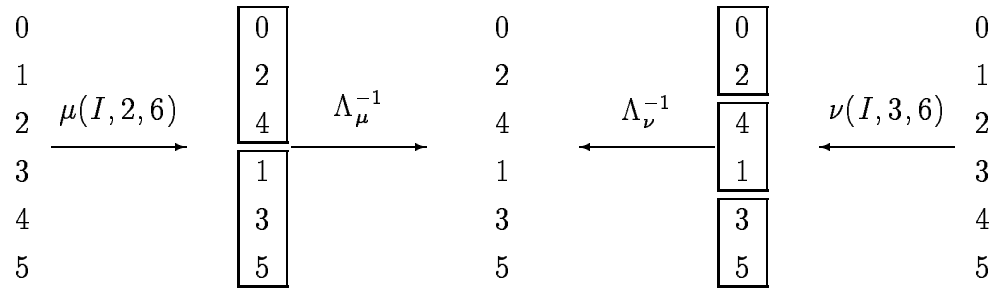
where the associated linear distributions Λ_μ and Λ_ν have matching cardinalities:

$$\Lambda_\mu^\sharp(p, P, M) = \mu^\sharp(p, P, M) \tag{2}$$

$$\Lambda_\nu^\sharp(p, P, M) = \nu^\sharp(p, P, M) \tag{3}$$

$$\forall p = 0, \ldots, P-1.$$

Figure 1. Example of Permutation Compatibility



Two different data-distributions, $\mu(\bullet, 2, 6)$ and $\nu(\bullet, 3, 6)$, are permutation compatible.

An example of two permutation compatible distributions is given in Figure 1.

3 Class-1 and Class-2 CBLAS

The Class-1 CBLAS functions are similar in purpose and implementation to the Level-1, -2, and -3 BLAS. The major difference is that CBLAS take objects as arguments rather than pointers to allocated data as in the Fortran-66 style. The main implication of this is that the interface must differ. For example, there must be a way to pass the starting location for submatrix computations other than the standard BLAS method of offsetting the pointer location (*e.g.*, `A(1,j)` for the jth column of a matrix A). One approach is to pass the upper left index coordinates for the submatrix. Another interesting issue is that of the `LDA` value (leading dimension of a canonical A matrix) used in the BLAS. The true `LDA` value is already a part of the matrix object. However, because this value is explicitly passed into the BLAS routines, one can "trick" the routines into operating on various "strided" submatrices. This same functionality can be achieved by carefully modifying the matrix object (*i.e.,* changing the dimensions) before calling the CBLAS routines. Another option is to extend this functionality by passing to the CBLAS routines both a row increment (`inci`) and a column increment (`incj`).

The Class-2 CBLAS functions operate on combinations of sequential and concurrent objects. An example of a Class-2 CBLAS routine is a `Cmatrix-Svector` multiply. Since some of the objects are maximally replicated across the logical grid $\mathcal{G}$, the Class-2 CBLAS simplify some of the issues involved in the Class-3 CBLAS. In particular, replication eliminates the need for some communication. For example, in a `GEMV` operation between a `Cmatrix` A and an `Svector` x, there is no need to communicate x since x is already contained in each process. However, if the result is an `Svector`, then we must *combine* the contributions of each process across the entire logical grid $\mathcal{G}$, rather than just across rows or along columns (as is done in

the Class-3 `GEMV` case).

4 Class-3 CBLAS

Three representative Class-3 CBLAS functions are the dot product of two `Cvector` objects, the outer product of two `Cvector` objects, and `Cmatrix`–`Cmatrix` multiplication, depending respectively on Level-1, Level-2 and Level-3 BLAS arithmetic. We have introduced such *Toolbox* functions before (*e.g.*, [9]), though without this CBLAS nomenclature. In this section, we discuss the implementation of `Cmatrix`–`Cvector` and `Cmatrix`–`Cmatrix` multiplication. It is assumed throughout that the data are distributed on the same process grid, $\mathcal{G}_{P\times Q}$.

The Class-3 CBLAS operations introduce the complications of *algorithm compatibility*, which we define as follows:

Definition 3 (Algorithm Compatibility) *That which is required of the data to ensure algorithm correctness.*

This definition, simple as it may be, is the main source of difficulty when designing a CBLAS library. If the data is not algorithm compatible, it will need to be redistributed before correct results can be obtained, and this can be costly. For example, consider an `AXPY` ($y = ax + y$) operation, and suppose we allow offset values to be passed into the routine. This causes no problems in the BLAS since data is always available. In the CBLAS, the elements of x and y needed for the addition may lie in different processes. Hence, in the most general setting the subvector that is being extracted from the `Cvector` x might have to be redistributed in order to perform the addition. The situation is even more complex if increment values are to be passed in. We do not consider the use of offsets and increments in this paper.

The global communication primitive, $shift_{(i,j)}$, will be used extensively in our Class-3 algorithms, and is defined as follows:

Definition 4 ($shift_{(i,j)}$) *Let D represent data distributed on a grid $\mathcal{G}_{P\times Q}$, and let $D_{p,q}$ be that part of D contained in process (p,q). Then, the global operation $shift_{(i,j)}$ is defined by:*

$$\begin{aligned} & shift_{(i,j)}\ D_{p,q}\ :\ D_{p,q}\ \mapsto\ D_{p',q'}; \\ & p' \ = \ (p+i) \bmod P, \\ & q' \ = \ (q+j) \bmod Q. \end{aligned} \tag{4}$$

The notation $shift_{(i,j)}\ D$ means $shift_{(i,j)}\ D_{p,q}$ for all p, q.

Notation and Pseudocode

Let A be a `Cmatrix` object, and let x_r and y_c be `Cvector` objects, where the subscript 'r' indicates row-distribution/column-replication and the subscript 'c' indicates column-distribution/row-replication. Then, the notation $A_{p,q}$ is meant to represent the submatrix of A that is stored in process (p,q), and the notation $(x_r)_p$ and $(y_c)_q$ are meant to represent the subvector of x_r replicated in grid row p and the subvector of y_c replicated in grid column q, respectively.

The notational conventions are different within our "process-(p,q)" pseudocode. Since all of the algorithms that follow are presented as being logically local to some process, (p,q), the objects within them do not need subscripts. Hence, the notation A, x_r, and y_c within our pseudocode will mean $A_{p,q}$, $(x_r)_p$, and $(y_c)_q$, respectively. Also within the pseudocode, the notation $A(i_1 : i_2, j_1 : j_2)$ represents the submatrix of $A_{p,q}$ consisting of rows i_1 through i_2 and columns j_1 through j_2 (the indices here are local indices).

4.1 Matrix-Vector Multiplication

Since `Cvector` objects are either row- or column-distributed, there are several possible orientations of vectors in the matrix-vector part of a `GEMV` operation. We consider the following operations: $y_r \leftarrow \alpha A x_c + \beta y_r$, $y_r \leftarrow \alpha A x_r + \beta y_r$, $y_c \leftarrow \alpha A x_r + \beta y_c$, and $y_c \leftarrow \alpha A x_c + \beta y_c$. The algorithm compatibility requirements of the algorithms in this section are similar to those defined in section 4.2 for the Matrix-Matrix multiplication algorithms. We therefore exclude them from the text.

4.1.1 Algorithms

For the operation, $y_r \leftarrow \alpha A x_c + \beta y_r$, the `Cvector` is correctly oriented with the `Cmatrix` for an efficient matrix-vector multiply. The local contributions can be computed in each process followed by a *row combine* to gather the total. This case is clearly understood and has been discussed in [8, 9].

For the remaining operations, the algorithms are more complicated since the elements needed for the local contributions are not present in some of the processes. We have the following process-(p,q) pseudocode for operations $y_r \leftarrow \alpha A x_r + \beta y_r$ and $y_c \leftarrow \alpha A x_r + \beta y_c$:

algorithm *MV–2* $[y_r \leftarrow \alpha A x_r + \beta y_r]$
    if $(q = 0)$ **then**
        $y_r = \beta y_r$;
    else
        $y_r = 0$;
    end if
    $x_c \leftarrow$ *vector transpose* x_r;
    $y_r = y_r + \alpha A x_c$;
    $y_r \leftarrow$ *row combine* y_r;
end algorithm

algorithm *MV–3* $[y_c \leftarrow \alpha A x_r + \beta y_c]$
    $x_c \leftarrow$ *vector transpose* x_r;
    $v_r = \alpha A x_c$;
    $v_r \leftarrow$ *row combine* v_r;
    $v_c \leftarrow$ *vector transpose* v_r;
    $y_c = \beta y_c + v_c$;
end algorithm

The algorithm for operation $y_c \leftarrow \alpha A x_c + \beta y_c$ is the same as *MV–3*, without the initial *vector transpose*.

The BLAS allow the matrix to be transposed "virtually" in a `GEMV` operation to calculate $y \leftarrow \alpha A^T x + \beta y$. Examining these cases in the CBLAS, reveals that they are variations of the untransposed cases. For example, $y_c \leftarrow \alpha A^T x_c + \beta y_c$ can be solved using the same methods as $y_r \leftarrow \alpha A x_r + \beta y_r$.

4.2 Matrix-Matrix Multiplication

Many algorithms have been proposed and studied for multiplying dense, concurrent matrices (see e.g. [6, 1, 3, 2]). These algorithms are usually defined to operate on square matrices uniformly distributed on square process grids. Also, the distribution of the data is often driven by the algorithms. In this section, we extend these methods to the general case of rectangular matrix multiplication on rectangular process grids, under the assumption that matrices are distributed as `Cmatrix` objects; hence, the algorithms are driven by the distributions.

We consider operations of the form: $C = \alpha AB + \beta C$, $C = \alpha A^T B + \beta C$, $C = \alpha AB^T + \beta C$, and $C = \alpha A^T B^T + \beta C$. The last operation is implemented as $C^T = \alpha BA + \beta C^T$, then C^T is explicitly transposed to recover the correct distribution.

The algorithms in this section are given names of the form $MM^{(a,b)}$*–tag*, where a and b indicate transposition (or not) of A and B, respectively (thus, indicating the operation), and *tag* identifies the algorithm. We will use the notation $MM^{(a,b)}$ to refer to the class of

Figure 2. Example of Multiplication Compatibility

$$\left(\begin{array}{ccc|ccc} a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ \hline a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ \hline a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \\ a_{*,0} & a_{*,2} & a_{*,4} & a_{*,1} & a_{*,3} & a_{*,5} \end{array}\right) \times \left(\begin{array}{cc|cc} b_{0,*} & b_{0,*} & b_{0,*} & b_{0,*} \\ b_{2,*} & b_{2,*} & b_{2,*} & b_{2,*} \\ \hline b_{4,*} & b_{4,*} & b_{4,*} & b_{4,*} \\ b_{1,*} & b_{1,*} & b_{1,*} & b_{1,*} \\ \hline b_{3,*} & b_{3,*} & b_{3,*} & b_{3,*} \\ b_{5,*} & b_{5,*} & b_{5,*} & b_{5,*} \end{array}\right)$$

Multiplication of matrices $A_{8\times 6}$ and $B_{6\times 4}$, distributed compatibly on a 3×2 process grid. Coefficient subscripts (*i.e.*, $a_{I,J}$) are the global (I, J) indices.

algorithms $MM^{(a,b)}$–*tag*, for all *tag*.

Since we are targeting many computer architectures in the *Toolbox*, an exhaustive performance analysis of our algorithms would be tedious. Also, because the algorithms are in general loosely synchronous, it is virtually impossible to write closed-form expressions representing their cost. However, we will discuss and compare the general aspects of each of the algorithms presented. We will also provide some experimental results at the end of the section.

4.2.1 Algorithms for $C = \alpha AB + \beta C$

The algorithms we discuss in this section have "row versions" and "column versions." We state the row versions here, and refer the column versions to the appendix. All of the algorithms require the following compatibility definition:

Definition 5 (Algorithm $MM^{(,)}$ Compatibility) *Matrices A, B, and C are compatible for algorithms $MM^{(,)}$ if and only if the column distribution of A is permutation compatible with the row distribution of B, and the row and column distributions of C are equal to the the row distribution of A and the column distribution of B, respectively.*

Figure 2 illustrates this definition for a 3×2 logical process grid and matrices A and B with shapes 8×6 and 6×4, respectively. Note, that if we view the multiplication, AB, as $(\mathcal{P}_A A \mathcal{Q}_A)(\mathcal{P}_B B \mathcal{Q}_B)$, where $\mathcal{P}_A$ (resp, $\mathcal{P}_B$) is a global permutation view of A's (resp, B's) row distribution, and $\mathcal{Q}_A$ (resp, $\mathcal{Q}_B$) is a global permutation view of A's (resp, B's) column distribution, then the compatibility definition above requires that $\mathcal{Q}_A = \mathcal{P}_B^T$. Mathematically, this means that

$$(\mathcal{P}_A A \mathcal{Q}_A)(\mathcal{P}_B B \mathcal{Q}_B) = \mathcal{P}_A (AB) \mathcal{Q}_B = \mathcal{P}_A C \mathcal{Q}_B,$$

so that C has A's row distribution and B's column distribution.

Our first algorithm is given by the following process-(p,q) pseudocode:

```
algorithm MM^(,)-1 [row version]
        C = βC;

        for (r_B = Q; r_B > 0; r_B = r_B − 1)
                (k_1, j_1) ← μ_c(μ_r^{-1}(p, 0, P, M), Q, N);
                (k_2, j_2) ← μ_c(μ_r^{-1}(p, m_B − 1, P, M), Q, N);
                if (k_2 < k_1) then
                        k_2 = k_2 + Q;
                end if

                if (k_1 ≤ q) and (k_2 ≥ q) then
                        if (k_1 < q) then
                                j_1 = 0;
                        end if
                        if (k_2 > q) then
                                j_2 = n_A − 1;
                        end if

                        S ← row combine A_(:, j_1:j_2);

                        C = C + αSB;
                end if

                shift_(−1,0) B;
        end for
end algorithm
```

In Algorithm $MM^{(,)}$–*1* we seek to mimic the canonical systolic algorithm [6]. However, more than one processor will, in general, contain fragmented data that is needed to accomplish each step of the algorithm. We are thus forced to use a *combine*, which is more expensive than a *broadcast*,

Our second algorithm is given by the following process-(p,q) pseudocode, and is illustrated in Figures 3a–f.

```
algorithm MM^(,)-2 [row version]
        C = βC;

        (k, j) ← μ_c(μ_r^{-1}(p, 0, P, M), Q, N);

        S ← row broadcast_(k) A;
        T ← S_(:,:j-1);

        a = j;    r_A = P;
        b = 0;    r_B = Q;
        while (r_B > 0)
                r = min{(n_A - a), (m_B - b)};
                C = C + αS_(:,a:a+r) B_(b:b+r,:);
                a = a + r;    b = b + r;

                if (a = n_A) then
                        if (r_A > 1) then
                                k = (k + 1) mod Q;
                                S ← row broadcast_(k) A;
                        else if (r_A = 1) then
                                S ← T;
                        end if
                        a = 0;    r_A = r_A - 1;
                end if

                if (b = m_B) then
                        shift_(-1,0) B;
                        b = 0;    r_B = r_B - 1;
                end if
        end while
end algorithm
```

Algorithm $MM^{(,)}$–*2* is able to use *broadcast* instead of *combine*, but requires more storage than Algorithm $MM^{(,)}$–*1*; in particular, T must be stored.

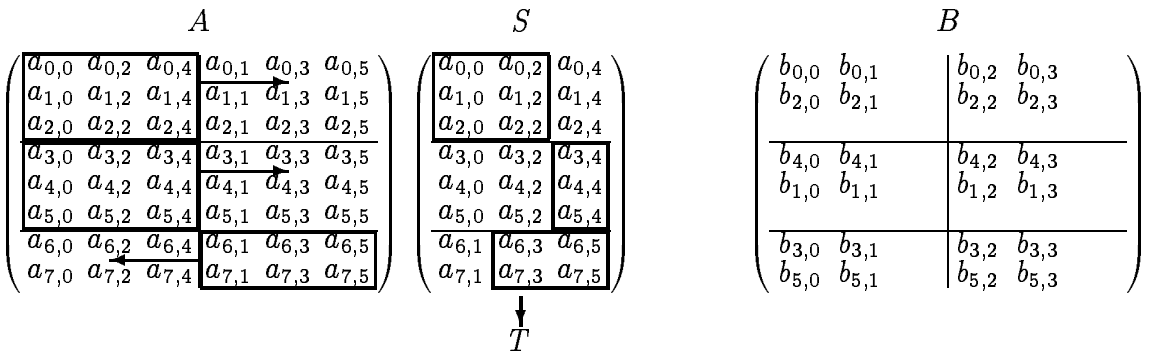


Figure 3a ($MM^{(,)}$–*2* example). Processes $(0,0)$, $(1,0)$, and $(2,1)$ *row broadcast* their data. Each process copies some of its data into temporary storage, T, then computes as much of the matrix-matrix multiply as can be done locally (data in S used in the multiplies are marked with solid boxes).

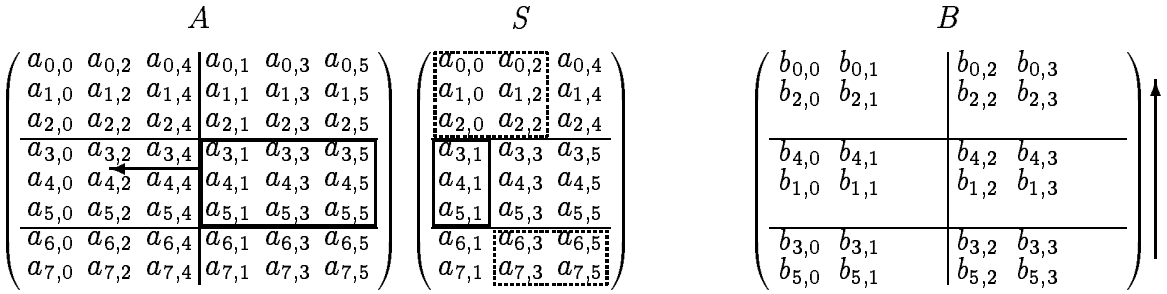


Figure 3b ($MM^{(,)}$–*2* example). Process $(1,1)$ *row broadcasts* its data, then processes (1,q) compute as much of the multiply as can be done locally (data in S already used are marked with dashed boxes). B is $shift_{(-1,0)}$ (this is actually a loosely synchronous process).

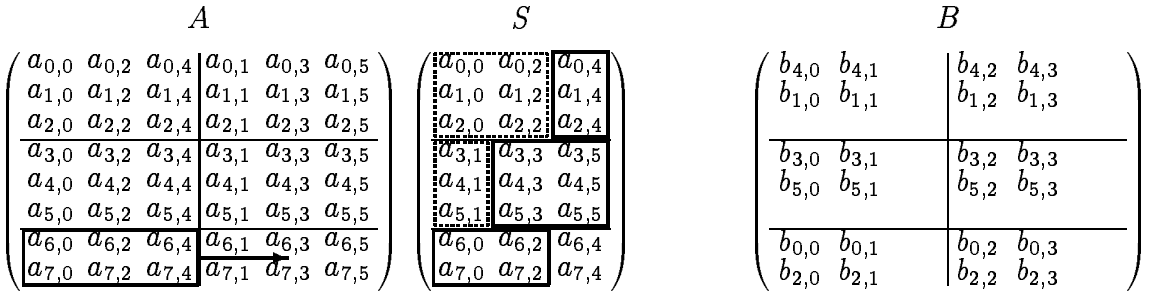


Figure 3c ($MM^{(,)}$–*2* example). Process $(2,0)$ *row broadcasts* its data, then each process computes as much of the multiply as can be done locally.

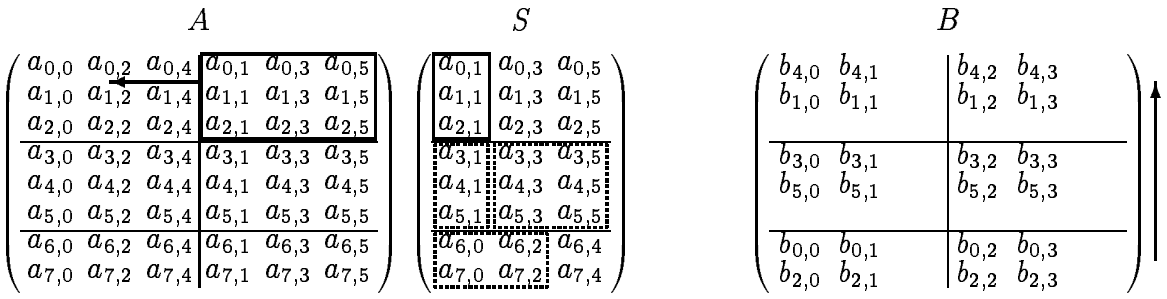


Figure 3d ($MM^{(,)}$–*2* example). Process $(0,1)$ *row broadcasts* its data, then processes (0,q) compute as much of the multiply as can be done locally. B is $shift_{(-1,0)}$.

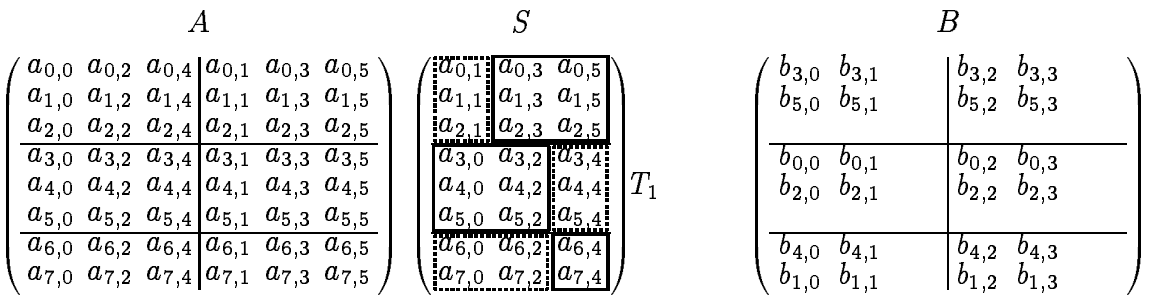


Figure 3e ($MM^{(,)}$–*2* example). Each process computes as much of the multiply as can be done locally; processes $(1,q)$ use T.

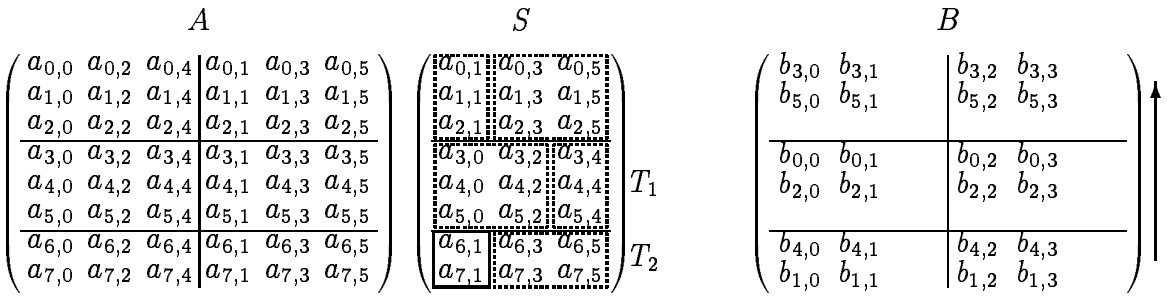


Figure 3f ($MM^{(,)}$–*2* example). Processes $(2,q)$ use T to compute the remainder of the multiplies. Then, B is $shift_{(-1,0)}$, restoring its original distribution.

algorithm *$MM^{(,)}$–3 [row version]*

$C = \beta C$;

$(k, j) \leftarrow \mu_c(\mu_r^{-1}(p, 0, P, M), Q, N)$;

$S \leftarrow$ *row broadcast*$_{(k)}$ $A_{(:,j:)}$;

$a = 0$; $r_A = P$;
$b = 0$; $r_B = Q$;
while $(r_B > 0)$
  $r = \min\{(n_A - a), (m_B - b)\}$;
  $C = C + \alpha S_{(:,a:a+r)} B_{(b:b+r,:)}$;
  $a = a + r$; $b = b + r$;

  if $(a = n_A)$ **then**
    $k = (k + 1) \bmod Q$;
    if $(r_A > 1)$ **then**
      $S \leftarrow$ *row broadcast*$_{(k)}$ A;
    else if $(r_A = 1)$ **then**
      $S \leftarrow$ *row broadcast*$_{(k)}$ $A_{(:,:j-1)}$;
    end if
    $a = 0$; $r_A = r_A - 1$;
  end if

  if $(b = m_B)$ **then**
    shift$_{(-1,0)}$ B;
    $b = 0$; $r_B = r_B - 1$;
  end if
end while
end algorithm

Algorithm *$MM^{(,)}$–3* removes the need for T, but requires one more *broadcast* than Algorithm *$MM^{(,)}$–2* to complete.

Load Balancing and Synchronization

To achieve optimal performance of a parallel algorithm, one must balance the work load of each process. In order to balance both computation and communication in the algorithms above, it is easy to see that we must have that $A_{p,q}$ is $m_A \times n_A$ for all p, q, and that $B_{p,q}$ is $m_B \times n_B$ for all p, q.

Under these conditions, algorithm *$MM^{(,)}$–1* is essentially synchronous (small asynchronous

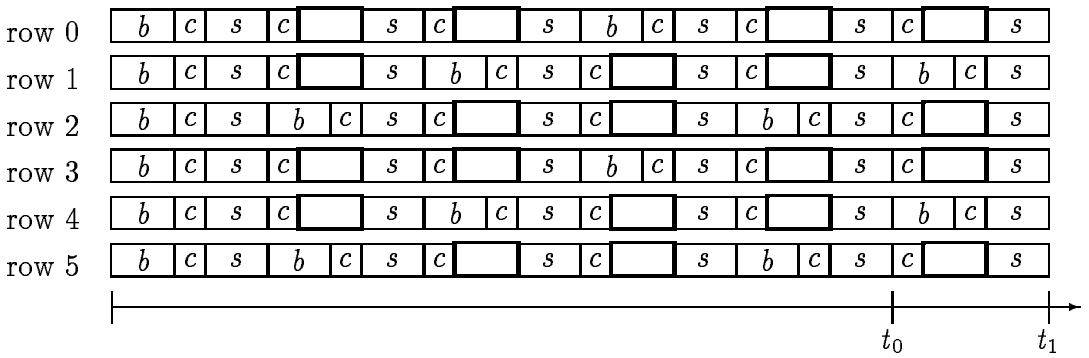


Figure 4. The effect of loose syncronization for algorithm $MM^{(,)}$–*3* multiplying matrices $A_{12\times 12}$ and $B_{12\times 2}$, linearly distributed on a 6×2 process grid. The letters b, c, and s represent *broadcast*, computation, and *shift*, respectively. Time t_1 is required to complete the multiplication, whereas time t_0 would be required were the algorithm synchronous.

effects may result if the average message size of each row *combine* differs from process row to process row).

However, algorithms $MM^{(,)}$–*2* and $MM^{(,)}$–*3* are generally loosely synchronous. As a result, the algorithms may display sub-optimal performance because of idle time accumulated by processes waiting for "not-yet-used" rows of B (note: this may offset the cost saved using *broadcast* instead of *combine*). For example, let $A_{12\times 12}$ and $B_{12\times 2}$ be linearly distributed on a 6×2 process grid, so that each $A_{p,q}$ is 2×6 and each $B_{p,q}$ is 2×1. Now, consider using using $MM^{(,)}$–*3* to multiply A and B. Figure 4 illustrates the effect of loose synchronization. We see that process rows are sometimes forced to wait for new components of B, and each wait takes roughly the time required to *broadcast* a component of A.

In algorithm $MM^{(,)}$–*4*, below, we attempt to "fix" this synchronization problem. First we must introduce a new global data redistribution operation.

Definition 6 (*slide*$_{(i,j)}$) *Let $D = \{d_{I,J}\}$ represent data distributed on a grid $\mathcal{G}_{P\times Q}$ with row and column distributions: $\mu_r(I, P, M)$ and $\mu_c(J, Q, N)$. Then, the global operation slide$_{(i,j)}$ D redistributes $d_{I,J}$ with row and column distributions:*

$$\Lambda_{\mu_r}((I_{\mu_r} + i) \bmod M, P, M),$$
$$\Lambda_{\mu_c}((J_{\mu_c} + j) \bmod N, Q, N),$$

where Λ_{μ_r} and Λ_{μ_c} are associated linear functions with matching cardinality, and

$$\begin{aligned} I_{\mu_r} &= \Lambda^{-1}_{\mu_r}(\mu_r(I, P, M), P, M), \\ J_{\mu_c} &= \Lambda^{-1}_{\mu_c}(\mu_c(J, Q, N), Q, N). \end{aligned}$$

Algorithm $MM^{(,)}$–*4* is given by the following process-(p, q) pseudocode, and is illustrated in

Figures 5a–f.

algorithm $MM^{(,)}$*–4 [row version]*
$C = \beta C;$

$(k, j) \leftarrow \mu_c(\mu_r^{-1}(p, 0, P, M), Q, N);$

$slide_{(0,-j)}\ A;$

$S \leftarrow$ *row broadcast*$_{(k)}\ A;$

$a = 0; \quad r_A = P;$
$b = 0; \quad r_B = Q;$
while $(r_B > 0)$
$r = \min\{(n_A - a), (m_B - b)\};$
$C = C + \alpha S_{(:,a:a+r)} B_{(b:b+r,:)};$
$a = a + r; \quad b = b + r;$

if $(a = n_A)$ **then**
if $(r_A > 1)$ **then**
$k = (k + 1) \bmod Q;$
$S \leftarrow$ *row broadcast*$_{(k)}\ A;$
end if
$a = 0; \quad r_A = r_A - 1;$
end if

if $(b = m_B)$ **then**
$shift_{(-1,0)}\ B;$
$b = 0; \quad r_B = r_B - 1;$
end if
end while

$slide_{(0,j)}\ A;$
end algorithm

Given the matrix multiply at hand, it is possible that one version of an algorithm above (*e.g.,* the column version of $MM^{(,)}$*–2*) may be faster than its orthogonal version (*e.g.,* the row version of $MM^{(,)}$*–2*). In general, it is difficult to predict which is best. However, it is usually the case that the row version is faster than the column version when $Q < P$, and vice-versa.

Figure 5a ($MM^{(,)}$–*4* example). Processes $slide_{(0,-t)}$ A, where $t = 0$, 2, and 1 for process rows 0, 1, and 2, respectively.

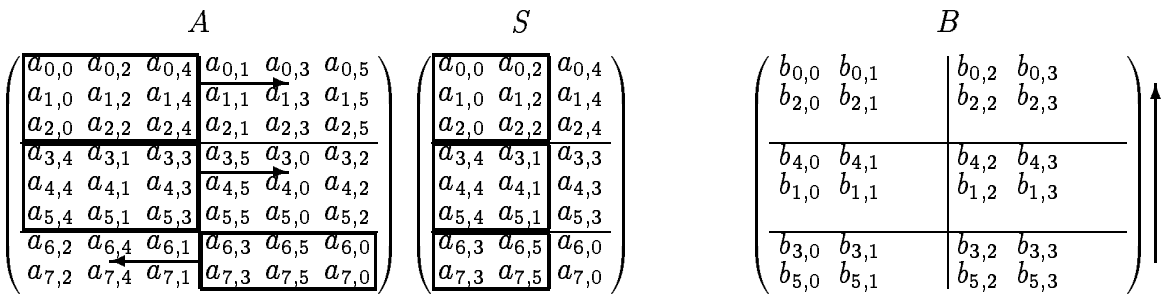


Figure 5b ($MM^{(,)}$–*4* example). Processes $(0,0)$, $(1,0)$, and $(2,1)$ *row broadcast* their data. Each process computes as much of the matrix-matrix multiply as can be done locally. B is $shift_{(-1,0)}$.

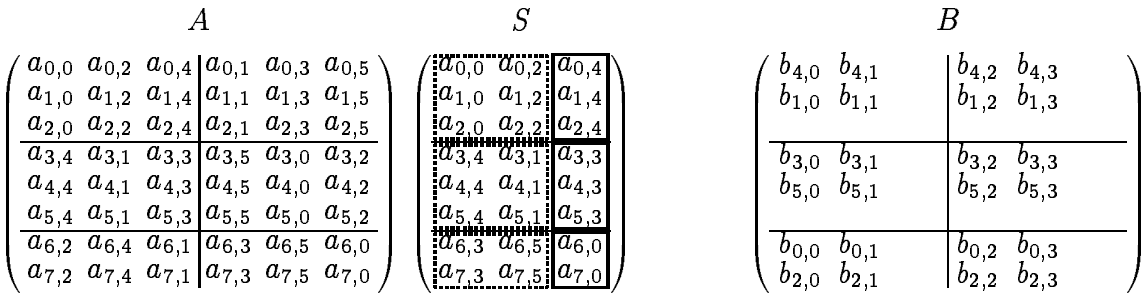


Figure 5c ($MM^{(,)}$–*4* example). Processes compute as much of the multiply as can be done locally.

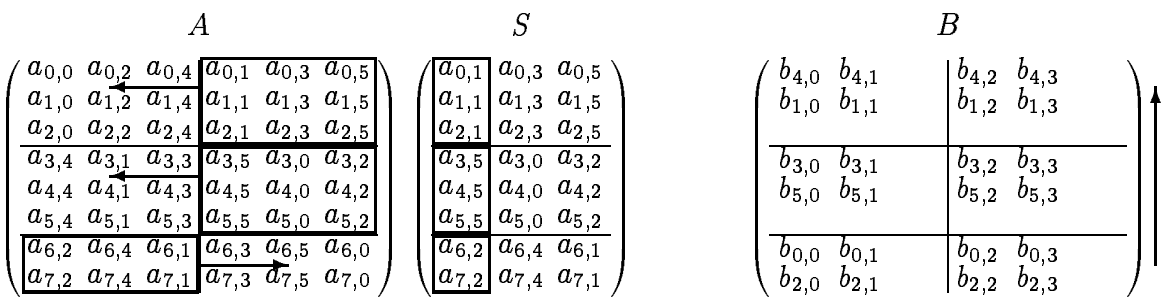


Figure 5d ($MM^{(,)}$–*4* example). Processes $(0,1)$, $(1,1)$, and $(2,0)$ *row broadcast* their data, then each process computes as much of the multiply as can be done locally. B is $shift_{(-1,0)}$.

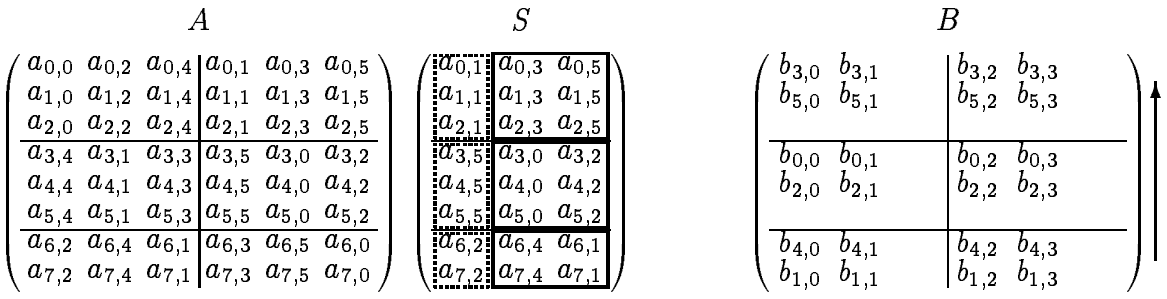


Figure 5e ($MM^{(,)}$–*4* example). Processes compute as much of the multiply as can be done locally. B is $shift_{(-1,0)}$.

Figure 5f ($MM^{(,)}$–*4* example). Processes $slide_{(0,t)}$ A to restore its originial distribution.

4.2.2 Algorithms for $C = \alpha A^T B + \beta C$

The algorithm in this section requires the following compatibility definition:

Definition 7 (Algorithm $MM^{(T,)}$ Compatibility) *Matrices A, B, and C are compatible for algorithm $MM^{(T,)}$ if and only if the row distribution of A is equal to the row distribution of B, and the row and column distributions of C are equal to the the column distribution of A and the column distribution of B, respectively.*

From this definition, we see that it is necessary that the row distribution of A be permutation compatible with the row distribution of B. Hence, if, as before, we view the multiplication, $A^T B$, as $(\mathcal{P}_A A \mathcal{Q}_A)^T (\mathcal{P}_B B \mathcal{Q}_B)$, this means that $\mathcal{P}_A = \mathcal{P}_B$ so that

$$(\mathcal{P}_A A \mathcal{Q}_A)^T (\mathcal{P}_B B \mathcal{Q}_B) = \mathcal{Q}_A^T (A^T \mathcal{P}_A^T \mathcal{P}_B B) \mathcal{Q}_B = \mathcal{Q}_A^T C \mathcal{Q}_B.$$

Note, here that on non-square grids, the distribution of C creates a problem. We will discuss this later in the section. Also, notice that here we require equality of distributions.

Our algorithm is given by the following process-(p, q) pseudocode:

```
algorithm MM^(T,)
        C = βC;

        k = q mod P;

        for (r_B = Q; r_B > 0; r_B = r_B − 1)
                if (k = p) then
                        S = C + αA^T B;
                else
                        S = αA^T B;
                end if
                C ← col sum fanin_(k) S;
                k = (k + 1) mod P;

                shift_(0,−1) B;
        end for
end algorithm
```

Implementation Details

As mentioned earlier in this section, the result C has B's column distribution, but its row distribution is A's column distribution. Consequently, C is distributed on a process grid, $\mathcal{G}_{Q \times Q}$, linearly mapped onto $\mathcal{G}_{P \times Q}$ (*i.e.*, process (p, q) of $\mathcal{G}_{Q \times Q}$ is process $(p \bmod P, q)$ of

$\mathcal{G}_{P\times Q}$). To provide generality, our `Cmatrix` must be able to represent this resultant data. This either requires new data distribution technology or modifications to the `Cmatrix` structure.

Another aspect of this algorithm mentioned earlier concerns algorithm compatibility. Unlike the $MM^{(,)}$ algorithms, equality of distributions is needed instead of compatibility of distributions. To compute A^TB in general, complete redistribution is needed when the row distributions of A and B are not equal. However, when the row distributions are not equal, but are permutation compatible, we can save much time during the redistribution because we know that the global ordering of the data is correct. This redistribution may require multiple sends and receives to both north and south neighbors, but most likely will require only one *send* and/or *receive*. Hence, it is clear that permutation compatibility also serves a role in the implementation of $MM^{(T,)}$.

4.2.3 Algorithms for $C = \alpha AB^T + \beta C$

The algorithm in this section is very similar to that in the preceding section. We have the following compatibility definition:

Definition 8 (Algorithm $MM^{(,T)}$ Compatibility) *Matrices A, B, and C are compatible for algorithm $MM^{(,T)}$ if and only if the column distribution of A is equal to the column distribution of B, and the row and column distributions of C are equal to the the row distribution of A and the row distribution of B, respectively.*

Similarly to the previous section, we require that $\mathcal{Q}_A = \mathcal{Q}_B$ so that

$$(\mathcal{P}_A A \mathcal{Q}_A)(\mathcal{P}_B B \mathcal{Q}_B)^T = \mathcal{P}_A (A \mathcal{Q}_A \mathcal{Q}_B^T B^T) \mathcal{P}_B^T = \mathcal{P}_A C \mathcal{P}_B^T.$$

The algorithm is given by the following process-(p,q) pseudocode:

```
algorithm MM^(,T)
        C = βC;

        k = p mod Q;

        for (r_B = Q; r_B > 0; r_B = r_B − 1)
                if (k = q) then
                        S = C + αAB^T;
                else
                        S = αAB^T;
                end if
                C ← row sum fanin_(k) S;
                k = (k + 1) mod Q;

                shift_(−1,0) B;
        end for
end algorithm
```

4.2.4 Results

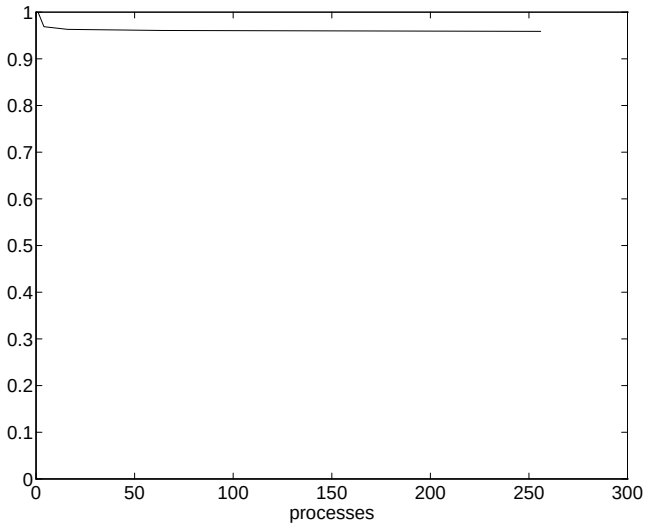
The following results were obtained on a 1,024-node nCUBE/2 at Sandia National Laboratories.

Figure 6 is a scaled speedup graph for $MM^{(,)}$–*4*. Here, we keep the problem size per process fixed at 128×128, and use grid sizes: 1×1, 2×2, 4×4, 8×8, and 16×16. We define scaled speedup as $M_p/(pM_1)$, where M_p is MFLOPS on p processes. Because of the *broadcast*, none of the algorithms in this paper are truly scalable. However, since the cost of a *broadcast* grows like $\log_2 p$, for all practical purposes the algorithms are scalable. This is evident in the figure.

In Figures 7 and 8, we graph MFLOPS for $MM^{(,)}$–*4* (row version), $MM^{(,)}$–*2* (row version), and $MM^{(,)}$–*2* (column version), for problem sizes 256×256 and 1024×1024 on grids of varying shapes (for each of these grid shapes, the total number of processes is 256). In both figures, we see that $MM^{(,)}$–*4* is faster than $MM^{(,)}$–*2* (row), even though the former algorithm requires more communication. This is because the extra cost incurred by $MM^{(,)}$–*4* to redistribute data is cheaper than the cost of the synchronization problems in $MM^{(,)}$–*2*, as discussed earlier. We also see that $MM^{(,)}$–*2* (column) is slower than both $MM^{(,)}$–*4* (row) and $MM^{(,)}$–*2* (row). This is because $MM^{(,)}$–*2* (column) is doing *col broadcasts*, and the column sizes are all at least as large as the row sizes.

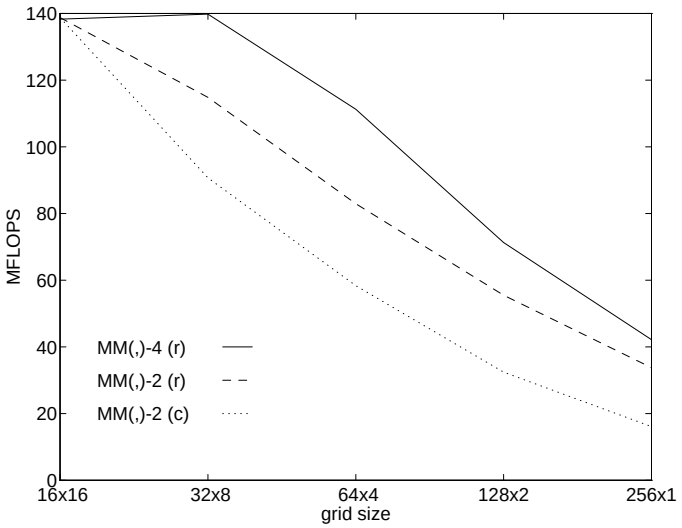
It is interesting to note that for these particular experiments, $MM^{(,)}$–*2* (column) does not suffer from synchronization problems. As a result, $MM^{(,)}$–*4* (column) is equivalent. This is only because m_B evenly divides n_A in these tests, and is not the case in general.

Figure 6. Scaled Speedup



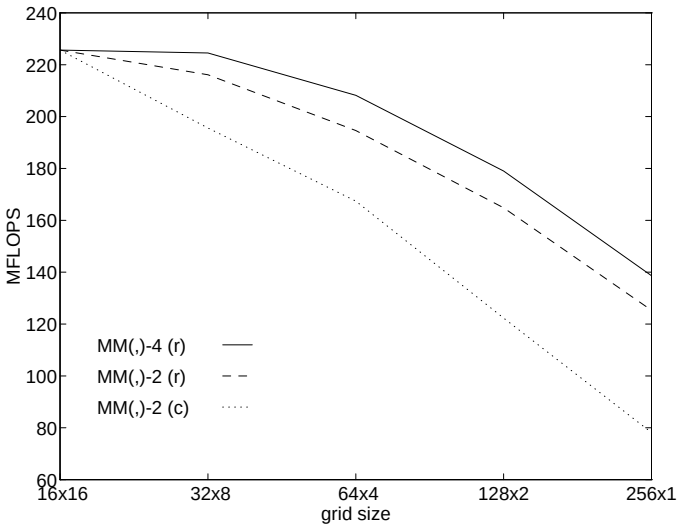
Results for algorithm $MM^{(,)}$–*4* on an nCUBE/2 keeping the $A_{p,q}$ and $B_{p,q}$ fixed at 128×128.

Figure 7. Performance Comparison of Matrix-Matrix Algorithms



Results for algorithms $MM^{(,)}$ on an nCUBE/2 where A and B are 256×256.

Figure 8. Performance Comparison of Matrix-Matrix Algorithms



Results for algorithms $MM^{(,)}$ on an nCUBE/2 where A and B are 1024×1024.

In both figures, as the grid shapes move from square to rectangular, the communication costs of the algorithms increase, causing their performance to drop. Note, however that in Figure 7, $MM^{(,)}$*–4* actually runs faster on the 32×8 grid than on the 16×16 grid. This is because the startup cost of a *send/receive* is smaller than the startup cost of a *broadcast*, so that the penalty paid for by the 16 extra *send/receives* in the 32×8 case is less than the savings made by the 8 fewer *broadcasts*. We do not see this effect as much in Figure 8 because the computation costs per process are large enough to dominate the communication startup costs. (*Zipcode* implementations with access to vendor-accelerated collective operations, such as MPI, could skew these results significantly.)

5 Summary / Future Work

We described our approach to developing a CBLAS library. We presented several matrix-vector and matrix-matrix algorithms, discussed various implementation issues, and demonstrated their effectiveness. We plan to continue developing a CBLAS library.

Acknowledgements

The authors wish to acknowledge the MPCRL at Sandia National Laboratories for access to their 1,024-node nCUBE/2 6400.

A Matrix-Matrix Algorithms (column versions)

algorithm *$MM^{(,)}$–1 [column version]*
$C = \beta C$;

for $(r_A = P;\ r_A > 0;\ r_A = r_A - 1)$
$(k_1, i_1) \leftarrow \mu_r(\mu_c^{-1}(q, 0, Q, N), P, M)$;
$(k_2, i_2) \leftarrow \mu_r(\mu_c^{-1}(q, n_A - 1, Q, N), P, M)$;
if $(k_2 < k_1)$ **then**
$k_2 = k_2 + P$;
end if

if $(k_1 \leq p)$ **and** $(k_2 \geq p)$ **then**
if $(k_1 < p)$ **then**
$i_1 = 0$;
end if
if $(k_2 > p)$ **then**
$i_2 = m_B - 1$;
end if

$S \leftarrow$ *col combine* $B_{(i_1:i_2,:)}$;

$C = C + \alpha SB$;
end if

$shift_{(0,-1)}\ A$;
end for
end algorithm

```
algorithm MM^(,)-2 [column version]
        C = βC;

        (k, i) ← μ_r(μ_c^{-1}(q, 0, Q, N), P, M);

        S ← col broadcast_(k) B;
        T ← S_(:,i-1,:);

        a = 0;    r_A = P;
        b = i;    r_B = Q;
        while (r_A > 0)
                r = min{(n_A - a), (m_B - b)};
                C = C + αA_(:,a:a+r) S_(b:b+r,:);
                a = a + r;    b = b + r;

                if (b = m_B) then
                        if (r_B > 1) then
                                k = (k + 1) mod P;
                                S ← col broadcast_(k) B;
                        else if (r_B = 1) then
                                S ← T;
                        end if
                        b = 0;    r_B = r_B - 1;
                end if

                if (a = n_A) then
                        shift_(0,-1) A;
                        a = 0;    r_A = r_A - 1;
                end if
        end while
end algorithm
```

algorithm *MM$^{(,)}$–3 [column version]*
$C = \beta C$;

$(k, i) \leftarrow \mu_r(\mu_c^{-1}(q, 0, Q, N), P, M)$;

$S \leftarrow$ *col broadcast*$_{(k)}$ $B_{(i:,:)}$;

$a = 0$; $r_A = P$;
$b = 0$; $r_B = Q$;
while $(r_A > 0)$
$r = \min\{(n_A - a), (m_B - b)\}$;
$C = C + \alpha A_{(:,a:a+r)} S_{(b:b+r,:)}$;
$a = a + r$; $b = b + r$;

if $(b = m_B)$ **then**
$k = (k + 1) \bmod P$;
if $(r_B > 1)$ **then**
$S \leftarrow$ *col broadcast*$_{(k)}$ B;
else if $(r_B = 1)$ **then**
$S \leftarrow$ *col broadcast*$_{(k)}$ $B_{(:i-1,:)}$;
end if
$b = 0$; $r_B = r_B - 1$;
end if

if $(a = n_A)$ **then**
shift$_{(0,-1)}$ A;
$a = 0$; $r_A = r_A - 1$;
end if
end while
end algorithm

algorithm *MM$^{(,)}$–4 [column version]*
$C = \beta C$;

$(k, i) \leftarrow \mu_r(\mu_c^{-1}(q, 0, Q, N), P, M)$;

$slide_{(-i,0)}\ B$;

$S \leftarrow$ *col broadcast*$_{(k)}\ B$;

$a = 0$; $r_A = P$;
$b = 0$; $r_B = Q$;
while $(r_A > 0)$
$r = \min\{(n_A - a), (m_B - b)\}$;
$C = C + \alpha A_{(:,a:a+r)} S_{(b:b+r,:)}$;
$a = a + r$; $b = b + r$;

if $(b = m_B)$ **then**
if $(r_B > 1)$ **then**
$k = (k + 1) \bmod P$;
$S \leftarrow$ *col broadcast*$_{(k)}\ B$;
end if
$b = 0$; $r_B = r_B - 1$;
end if

if $(a = n_A)$ **then**
$shift_{(0,-1)}\ A$;
$a = 0$; $r_A = r_A - 1$;
end if
end while

$slide_{(i,0)}\ B$;
end algorithm

References

[1] M. Aboelaze, N. P. Chrisochoides, E. N. Houstis, and C. E. Houstis. The parallelization of level 2 and 3 BLAS operations on distributed memory machines. Technical Report CSD-TR-91-007, Purdue University, West Lafayette, IN, 1991.

[2] L. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, Bozeman, MN, 1969.

[3] James W. Demmel, Michael T. Heath, and Henk A. van der Vorst. Parallel numerical linear algebra. Technical Report CSD-92-703, U. C. Berkeley, Berkeley, CA, 1992. Computer Science Division.

[4] J. J. Dongarra, J. DuCroz, I. Duff, and R. Hanson. A Set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. on Math. Soft., 5, December 1989.

[5] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. ACM Trans. on Math. Soft., 5, 1988.

[6] Geoffrey C. Fox, Mark A. Johnson Gregory A. Lyzenga, Steve W. Otto John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, 1988.

[7] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. ACM Trans. on Math. Soft., 14:308–325, 1989.

[8] Anthony Skjellum. *Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Ordinary Differential-Algebraic Process Systems in Chemical Engineering.* PhD thesis, Chemical Engineering, California Institute of Technology, May 1990. (You can request a copy from the author).

[9] Anthony Skjellum and Chuck H. Baldwin. *The Multicomputer Toolbox:* Scalable Parallel Libraries for Large-Scale Concurrent Applications. Technical Report UCRL-JC-109251, Lawrence Livermore National Laboratory, December 1991.

[10] Anthony Skjellum and Alvin P. Leung. *Zipcode:* A Portable Multicomputer Communication Library atop the *reactive kernel.* In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pages 767–776. IEEE, April 1990.

[11] Anthony Skjellum and Manfred Morari. Zipcode: A Portable Communication Layer for High Performance Multicomputing – Practice and Experience. Technical Report UCRL-JC-106725, Lawrence Livermore National Laboratory, March 1991. Accepted by *Concurrency: Practice & Experience.* In minor revision.