A Flexible Class of Parallel Matrix Multiplication Algorithms*

John Gunnels Calvin Lin Greg Morrow Robert van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

This paper explains why parallel implementation of matrix multiplication—a seemingly simple algorithm that can be expressed as one statement and three nested loops—is complex: Practical algorithms that use matrix multiplication tend to use matrices of disparate shapes, and the shape of the matrices can significantly impact the performance of matrix multiplication. We provide a class of algorithms that covers the spectrum of shapes encountered and demonstrate that good performance can be attained if the right algorithm is chosen. These observations set the stage for hybrid algorithms which choose between the algorithms based on the shapes of the matrices involved. While the paper resolves a number of issues, it concludes with discussion of a number of directions yet to be pursued.

1 Introduction

Over the last three decades, a number of different approaches have been proposed for implementing matrix-matrix multiplication on distributed memory architectures These include Cannon's algorithm [4], broadcast-multiply-roll [12, 11], and generalizations of broadcast-multiply-roll [8, 13, 2]. The approach now considered the most practical, known as broadcast-broadcast, was first proposed by Agarwal et al. [1], who showed that a sequence of parallel rank-k updates is a highly effective way to parallelize $C = AB$. This same observation was independently made by van de Geijn and Watts [17], who introduced the Scalable Universal Matrix Multiplication Algorithm (SUMMA). In addition to computing $C = AB$, SUMMA implements $C = AB^T$ and $C = A^T B$ as a sequence of matrix-panel-of-vectors multiplications, and $C = A^T B^T$ as a sequence of rank-k updates. Later work [5] showed how these techniques can be extended to a large class of commonly used matrix-matrix operations that are part of the Level-3 Basic Linear Algebra Subprograms (BLAS) [9].

These previous efforts have focused primarily on the special case where the input matrices are approximately square. Recent work by Li, et al. [15] creates a "poly-algorithm" that chooses between algorithms (Cannon's, broadcast-multiply-roll, and broadcast-broadcast). Empirical data show some advantage for different shapes of *meshes* (e.g. square vs rectangular). However, since the three algorithms are not inherently suited to specific shapes of *matrices*, limited benefit is observed.

We previously observed [16] the need for algorithms to be sensitive to the shape of the matrices and hinted that a class of algorithms naturally supported by the Parallel Linear Algebra Package (PLAPACK) provides a good basis for such shape-adaptive algorithms. Thus, a number of hybrid algorithms are now part of PLAPACK. This observation was also made by the ScaLAPACK [6] project, and indeed ScaLAPACK includes a number of matrix-matrix operations that choose algorithms based on the shape of the matrix. We contrast our approach with ScaLAPACK's later.

This paper describes and analyzes a class of parallel matrix multiplication algorithms that naturally lends itself to hybridization. The analysis combines theoretical results with empirical results to provide a complete picture of the benefits of the different algorithms and how they can be combined into hybrid algorithms. A number of simplifications are made to focus on the key issues. Having done so, a number of extensions are identified for further study.

2 Data Distribution

For all algorithms, we will assume that the processors are *logically* viewed as a $r \times c$ mesh of computational nodes which are indexed $\mathbf{P}_{i,j}$, $0 \leq i < r$ and $0 \leq j < c$, so that the total number of nodes is $p = rc$. Physically, these nodes are connected through some communication network, which could be a hypercube (Intel iPSC/860), a

*This research was supported primarily by the PRISM project (ARPA grant P-95006) and the Environmental Molecular Sciences construction project at Pacific Northwest National Laboratory. Additional support came from the NASA HPCC Program's Earth and Spaces Sciences Project (NRA Grants NAG5-2497 and NAG5-2511), and the Intel Research Council. Equipment was provided by the National Partnership for Advanced Computational Infrastructure, the University of Texas Compuation Center's High Performance Computing Facility, and the Texas Institute for Computational and Applied Mathematics.

higher dimensional mesh (Intel Paragon, Cray T3D/E) or a multistage network (IBM SP2).

Physically Based Matrix Distribution (PBMD). We have previously observed [10] that data distributions should focus on the decomposition of the physical problem to be solved, rather than on the decomposition of matrices. Typically, it is the *elements of vectors* that are associated with data of physical significance, and so it is their distribution to nodes that is significant. A matrix, which is a discretized operator, merely represents the relation between two vectors, which are discretized spaces: $y = Ax$. Since it is more natural to start by distributing the problem to nodes, we first distribute x and y to nodes. The matrix A is then distributed so as to be consistent with the distribution of x and y.

To describe the distribution of the vectors, assume that x and y are of length n and m, respectively. We will distribute these vectors using a *distribution block size* of b_{distr}. For simplicity assume $n = N b_{\text{distr}}$ and $m = M b_{\text{distr}}$. Partition x and y so that

$$x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{N-1} \end{array} \right) \text{ and } y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_{M-1} \end{array} \right)$$

where $N >> p$ and $M >> p$ and each x_i and y_i is of length b_{distr}. Partitioning A conformally yields the blocked matrix

$$A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,N-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{M-1,0} & A_{M-1,1} & & A_{M-1,N-1} \end{array} \right) \quad (1)$$

where each subblock is of size $b_{\text{distr}} \times b_{\text{distr}}$. Blocks of x and y are assigned to a 2D mesh of nodes in column-major order. The matrix distribution is induced by assigning blocks of columns $A_{*,j}$ to the same column of nodes as subvector x_j, and blocks of rows $A_{i,*}$ to the same row of nodes as subvector y_i [16]. This distribution wraps blocks of rows and columns onto rows and columns of processors, respectively. The wrapping is necessary to improve load balance.

Data movement. In the PBMD representation, vectors are fully distributed (not duplicated), while a single row or column of a matrix will reside in a single row or column of processors. One benefit of the PBMD representation is the clean manner in which vectors, matrix rows, and matrix columns interact. Converting a column of a matrix to a vector requires a scatter communication within rows. Similarly, a *panel* of columns can be converted into a *multi*vector (a group of vectors) by simultaneously scattering the columns within rows. For details, see [16].

3 A Class of Algorithms

The target operation will be

$$C = \alpha AB + \beta C$$

where for simplicity we will often treat only the case where $\alpha = 1$ and $\beta = 0$. There are three dimensions involved: m, n, and k, where C, A, and B are $m \times n$, $m \times k$, and $k \times n$, respectively.

It is always interesting to investigate extremal cases which happen when one or more of the dimensions equal unity:

One dimension equals unity:	
m, n large $k = 1$	A and B become column and row vectors, respectively, and the operation becomes a *rank-1* update.
m,k large $n = 1$	C and B are column vectors, and the operation becomes a matrix-vector multiply.
n, k large $m = 1$	C and A are row vectors, and the operation becomes a row vector-matrix multiply.
Two dimensions equal unity:	
m large $n = k = 1$	C and A become column vectors, and B a scalar. The operation becomes a scaled vector addition (`axpy`).
n large $m = k = 1$	C and B become row vectors, and A a scalar. Again, the operation becomes a scaled vector addition.
k large $m = n = 1$	A and B become row and column vectors, respectively, and C a scalar. The operation becomes an inner-product (`dot`).

Implementing the matrix-matrix multiplication for these degenerate cases is relatively straight-forward: The rank-1 update is implemented by duplicating vectors and updating local parts of C on each node. The matrix-vector multiplications are implemented by duplicating the vector to be multiplied, performing local matrix-vector multiplications on each node, and reducing the result to C. In the cases where two dimensions are small, the column or row vectors are redistributed like vectors (as described in the previous section) and local operations are performed on these vectors, after which the results are redistributed as needed.

In the next section, we will discuss how the case where none of the dimensions are unity can be implemented as a sequence of the above operations, by partitioning the operands appropriately. We present this along with one minor extension where rather than dealing with one row or column vector at a time, a number of these vectors are treated as a block (panel or multivector). This reduces the number of messages communicated and improves the performance on each node by allowing matrix-matrix multiplication kernels to be called locally.

3.1 Notation

For simplicity, we will assume that the dimensions m, n, and k are integer multiples of the algorithmic block size b_{alg}, we will use the following partitionings of A, B, and C:

$$X = \left(\begin{array}{c|c|c} X_1 & \cdots & X_{n_X/b_{\text{alg}}} \end{array} \right) = \left(\begin{array}{c} \hat{X}_1 \\ \hline \vdots \\ \hline \hat{X}_{m_X/b_{\text{alg}}} \end{array} \right)$$

where $X \in \{A, B, C\}$, and m_X and n_X are the row and column dimension of the indicated matrix. Here X_i and $\hat{X}_i$ indicate *panels* of columns and rows, respectively. Also,

$$X = \left(\begin{array}{c|c|c} X_{1,1} & \cdots & X_{1,n_X/b_{\text{alg}}} \\ \hline \vdots & & \vdots \\ \hline X_{m_X/b_{\text{alg}},1} & \cdots & X_{m_X/b_{\text{alg}},n_X/b_{\text{alg}}} \end{array} \right)$$

In these partitionings, b_{alg} is chosen to maximize the performance of the local matrix-matrix multiplication operation. In the below discussion, we will use $\hat{M} = m/b_{\text{alg}}$, $\hat{N} = n/b_{\text{alg}}$, and $\hat{K} = k/b_{\text{alg}}$.

3.2 Two dimensions are "large"

First we present algorithms that use kernels that correspond to the extremal cases discussed above where one dimension equals unity.

Panel-panel update based algorithm: m and n large. Notice that if m and n are both large compared to k, then C contains the most elements, and an algorithm that moves data in A and B rather than C may be a reasonable choice.

Observe that

$$\begin{aligned} C = AB &= \left(\begin{array}{c|c|c} A_1 & \cdots & A_{\hat{K}} \end{array} \right) \left(\begin{array}{c} \hat{B}_1 \\ \hline \vdots \\ \hline \hat{B}_{\hat{K}} \end{array} \right) \\ &= A_1 \hat{B}_1 + \cdots + A_{\hat{K}} \hat{B}_{\hat{K}} \end{aligned}$$

Thus, if we know how to perform one update $C \leftarrow C + A_i \hat{B}_i$ in parallel, the complete matrix-matrix multiplication can be implemented as $\hat{K}$ calls to this "panel-panel multiplication" kernel. In [17, 16] it is shown that one panel-panel update can be implemented by the following sequence of operations: Duplicate (broadcast) A_i within rows; Duplicate (broadcast) $\hat{B}_i$ within columns; On each node perform an update to the local portion of C. Computing $C = AB$ is then implemented by repeated calls to this kernel, once for each $A_i, \hat{B}_i$ pair.

Matrix-panel multiply based algorithm: m and k large. If m and k are both large compared to n, then A contains the most elements, and an algorithm that moves data in C and B rather than A may be a reasonable choice.

Observe that

$$C = \left(\begin{array}{c|c|c} C_1 & \cdots & C_{\hat{N}} \end{array} \right) = A \left(\begin{array}{c|c|c} B_1 & \cdots & B_{\hat{N}} \end{array} \right)$$

so that $C_j = AB_j$. Thus, if we know how to perform $C_j \leftarrow AB_j$ in parallel, the complete matrix-matrix multiplication can be implemented as $\hat{N}$ calls to this "matrix-panel multiplication" kernel. In [16] it is shown that one matrix-panel update can be implemented by the following sequence of operations: Scatter B_j within rows followed by a collect (all-gather) within columns to duplicate it; Perform a matrix-multivector multiplication local on each node, yielding contributions to C_j; Reduce (sum) the result to C_j. By calling this kernel for each B_j, C_j pair, $C = AB$ is computed.

Panel-matrix multiply based algorithm: n and k large. Finally, if n and k are both large compared to m, then B contains the most elements, and an algorithm that moves data in C and A rather than B may be a reasonable choice. The algorithm for this is similar to the matrix-panel based algorithm, except that it depends on a panel-matrix kernel instead.

3.3 Two dimensions are "small"

Next we present algorithms that use kernels that correspond to the extremal cases discussed above where two dimensions equal unity. Notice that implementation of these kernels is unique to PLAPACK, since other libraries like ScaLAPACK do not provide a facility for distributing vectors other than as columns or rows of matrices.

Column-axpy based algorithm: m large. Notice that if $n = k = 1$, the matrix-matrix multiplication becomes a scaled vector addition: $y \leftarrow \alpha x + y$, sometimes known as an "axpy" operation. If n and k are both small (e.g., equal one), then C and A will exist within only one or a few columns of nodes, while B is a small matrix that exists within one node, or at most a few nodes. To get good parallelism, it becomes beneficial to redistribute the columns of C and A like vectors.

Observe that

$$\begin{aligned} C &= \left(\begin{array}{c|c|c} C_1 & \cdots & C_{\hat{N}} \end{array} \right) \\ &= \left(\begin{array}{c|c|c} A_1 & \cdots & A_{\hat{K}} \end{array} \right) \left(\begin{array}{c|c|c} B_{1,1} & \cdots & B_{1,\hat{N}} \\ \hline \vdots & & \vdots \\ \hline B_{\hat{K},1} & \cdots & B_{\hat{K}\hat{N}} \end{array} \right) \end{aligned}$$

So that $C_j = A_1 B_{1,j} + \cdots + A_{\hat{K}} B_{\hat{K}j}$. Thus, if we know how to perform one update $C_j \leftarrow C_j + A_p \hat{B}_{pj}$ in parallel, the complete computation of C_j can be implemented as $\hat{K}$ calls to this "axpy" kernel, and the complete matrix-matrix multiply is accomplished by computation of all C_js. The following operations implement the computation of C_j:

- Redistribute (scatter) the columns of C_j like vectors
- for $q = 1, \ldots, \hat{K}$
  - Redistribute (scatter) the columns of A_q like vectors; Duplicate (broadcast) B_{qj} to all nodes; Update local part of $C_j \leftarrow C_j + A_q \hat{B}_{qj}$.
- Redistribute (gather) the columns of C_j to where they started

Repeated calls to this kernel compute the final C.

It may appear that the cost of redistributing C_j and A_q is prohibitive, making this algorithm inefficient. Notice however that the redistribution of C_j is potentially amortized over many updates. Also, the ratio of the computation with A_q and the cost of redistribution of this data is typically $b_{\text{alg}} : 1$. Thus, when k is small but comparable to b_{alg}, which itself is reasonably large, this approach is a viable alternative.

Row-axpy based algorithm: n large. Notice that if $m = k = 1$, the matrix-matrix multiplication again becomes a scaled vector addition, except that now rows of C and B are redistributed like vectors. The algorithm is similar to the last case discussed.

Dot based algorithm: k large. Notice that if $m = n = 1$, the matrix-matrix multiplication becomes an inner-product between one row A and one column B. For m and n small, we observe that

$$\begin{aligned} C &= \left(\begin{array}{c|c|c} C_{1,1} & \cdots & C_{1,\hat{N}} \\ \hline \vdots & & \vdots \\ \hline C_{\hat{M},1} & \cdots & C_{\hat{M}\hat{N}} \end{array} \right) \\ &= \left(\begin{array}{c} \hat{A}_1 \\ \hline \vdots \\ \hline \hat{A}_{\hat{M}} \end{array} \right) \left(\begin{array}{c|c|c} B_1 & \cdots & B_{\hat{N}} \end{array} \right) \end{aligned}$$

so that $C_{ij} = \hat{A}_i B_j$. Computation of each C_{ij} can be accomplished by redistributing columns of $\hat{A}_i$ and rows of B_j like vectors, performing local operations that are like local inner-products which are then reduced to C_{ij}.

3.4 Cost analysis

Notice that the above algorithms are implemented as an iteration over a sequence of communications and computations. By modeling the cost of each of the components, one can derive models for the cost of the different algorithms. Thus, we obtain a performance model based on the estimated cost of local computation as well as models of the communication required. Parameters for the model include the cost of local computation, message latency and bandwidth, and cost of packing and unpacking data before and after communications. Due to space limitations, we do not develop this cost analysis in this paper. Rather, we present graphs for the estimated cost of the different algorithms in the next section.

4 Results

This section provides performance results for the described algorithms on the Cray T3E and IBM SP-2. Our implementations are coded using PLAPACK which naturally supports these kinds of algorithms. In addition, we report results from our performance model, using parameters measured for the different architectures.

Ultimately, it is the performance of the local 64-bit matrix-matrix multiplication implementation that determines the asymptotic performance rate that can be achieved. Since the algorithms make repeated calls to parallel implementations of kernels that assume that one or more of the dimensions are small, the operands to the matrix-matrix multiplication performed on each individual processor also have different shapes that depend on the algorithm chosen. Even on an individual processor the shape of the operands can greatly affect the performance of the local computation, as is demonstrated in Table 1. In that table, the small dimensions were set equal to the blocking size used for the parallel algorithm, specifically 128 for the Cray T3E. It is this performance that becomes an asymptote for the per-node performance of the parallel implementation. It should be noted that since these timings were first performed, Cray has updated its BLAS library, and performance for the sequential matrix-matrix multiply kernels is much less shape dependent. Thus, the presented data is of qualitative interest, but performance is now considerably better than presented in the performance graphs.

It is interesting to note that the local matrix-matrix multiply performs best for panel-panel update operations. To

m	n	k	Rate (MFLOPS/sec)
4096	4096	128	439
4096	128	4096	172
128	4096	4096	237
4096	128	128	230
128	4096	128	184
128	128	4096	184

Table 1: Performance of local 64-bit matrix-matrix multiplication kernel on one processor of the Cray T3E.

those of us experienced with high-performance architectures this is not a surprise: This is the case that is heavily used by the LINPACK benchmark, and thus the first to be fully optimized. There is really no technical reason why for a block size of 128 the other shapes do not perform equally well. Indeed, as an architecture matures, those cases also get optimized. For example, on the Cray T3D the performance of the local matrix-matrix BLAS kernel is not nearly as dependent on the shape.

In Fig. 1, we report performance on the Cray T3E the different algorithms. By fixing one or more dimensions to be large, we demonstrate that, indeed, the appropriateness of the presented algorithms is dependent upon the shape of the operands. Of particular interest are the asymptotes reached by the different algorithms and how quickly the asymptotes are approached.

Let us start by concentrating on the top two graphs of Fig. 1. Notice that when m and n are large, it is the panel-panel based algorithm that attains high performance quickly. This is not surprising since the implementation makes repeated calls to an algorithm that is specifically designed for small k. The matrix-panel based algorithms suffer from the fact that for small k they incur severe load imbalance. The dot and axpy based algorithms incur considerable overhead, which is particularly obvious when k is small, and also hampers the asymptotic behavior of these algorithms.

In the middle two graphs of Fig. 1 we show that when m and k are large, a matrix-panel algorithm is most appropriate, at least when n is small. However, since the *local* matrix-matrix multiply does not perform as well in the matrix-panel case, eventually the panel-panel based algorithm outperforms the others.

In the bottom two graphs of Fig. 1, we show that when only one dimension is large it becomes advantageous to consider the dot or axpy based algorithms.

5 Conclusions and Future Work

We have presented a flexible class of parallel matrix multiplication algorithms. Theoretical and experimental results show that by choosing the appropriate algorithm from this class, high performance can be attained across a range of matrix shapes.

The analysis provides a means for choosing between the algorithms based on the shape of the operants. Furthermore, the developed techniques can easily used to parallelize other cases of matrix-matrix multiplication ($C = AB^T$, $C = A^T B$, and $C = A^T B^T$) and other matrix-matrix operations, as are included in the level-3 BLAS [9]. We intend to pursue this line of research in the near future.

Acknowledgments. We are indebted to the rest of the PLAPACK team (Philip Alpatov, Greg Baker, Carter Edwards, James Overfelt, and Yuan-Jye Jason Wu) for providing the infrastructure that made this research possible.

References

[1] Agarwal, R. C., F. Gustavson, and M. Zubair, "A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication," IBM J. of Research and Development, 38 (6), 1994.

[2] Agarwal, R. C., S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A 3-Dimensional Approach to Parallel Matrix Multiplication," IBM J. of Research and Development, 39(5), pp. 1–8, 1995.

[3] Barnett, M., S. Gupta, D. Payne, L. Shuler, R. van de Geijn, and J. Watts, "Interprocessor Collective Communication Library (InterCom)," *Scalable High Performance Computing Conf. 1994*.

[4] Cannon, L.E., *A Cellular Computer to Implement the Kalman Filter Algorithm*, Ph.D. Thesis (1969), Montana State University.

[5] Chtchelkanova, A., J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, "Parallel Implementation of BLAS: General Techniques for Level 3 BLAS," TR-95-40, Dept. of Computer Sciences, UT-Austin, Oct. 1995.

[6] Choi J., J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers," *Fourth Symposium on the Frontiers of Massively Parallel Computation.* IEEE Comput. Soc. Press, 1992, pp. 120-127.

[7] Choi, J., J. J. Dongarra, and D. W. Walker, "Level 3 BLAS for distributed memory concurrent computers", *CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing*, Sept. 7-8, 1992. Elsevier Science Publishers, 1992.

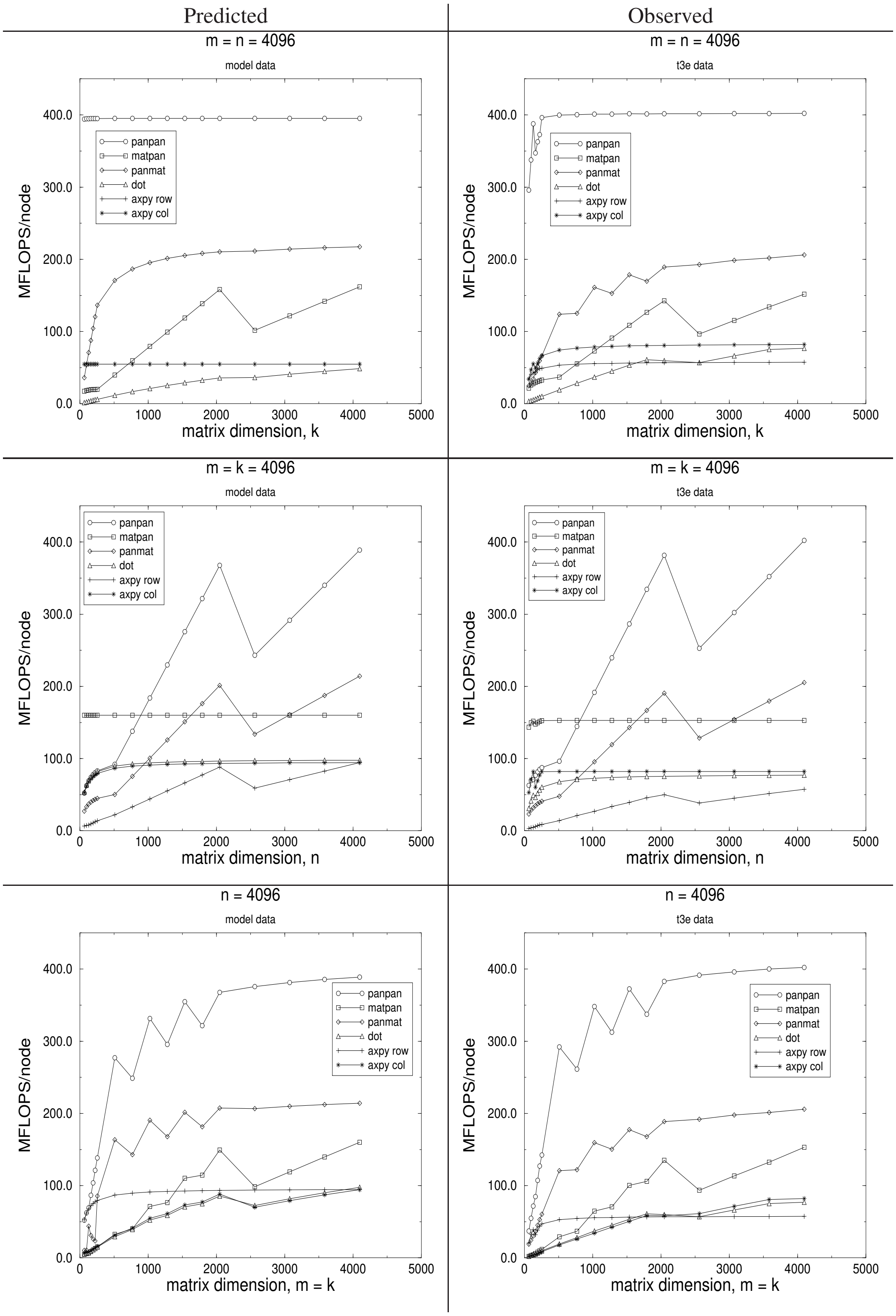

Figure 1: Performance of the various algorithms on a 16 processor configuration of the Cray T3E. Within each row of graphs, we compare predicted performance (*left*) to observed performance (*right*). In the top row of graphs, m and n are fixed to be large, and k is varied. In the middle row, m and k are fixed to be large, and n is varied. Finally, in the bottom row, n is fixed to be large, and m and k are simultaneously varied.

[8] Choi, J., J. J. Dongarra, and D. W. Walker, "PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, 6(7), 543-570, 1994.

[9] Dongarra, J. J., J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *TOMS*, 16 (1) pp. 1–16, 1990.

[10] Edwards, C., P. Geng, A. Patra, and R. van de Geijn, *Parallel matrix distributions: have we been doing it all wrong?*, Tech. Report TR-95-40, Dept of Computer Sciences, UT-Austin, 1995.

[11] Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, Vol. 1, Prentice Hall, Englewood Cliffs, N.J., 1988.

[12] Fox, G., S. Otto, and A. Hey, "Matrix algorithms on a hypercube I: matrix multiplication," Parallel Computing **3** (1987), pp 17-31.

[13] Huss-Lederman, S., E. Jacobson, A. Tsao, G. Zhang, "Matrix Multiplication on the Intel Touchstone DELTA," *Concurrency: Practice and Experience*, 6 (7) Oct. 1994, pp. 571-594.

[14] Lin, C., and L. Snyder, "A Matrix Product Algorithm and its Comparative Performance on Hypercubes," in *Scalable High Performance Computing Conference*, IEEE Press, 1992, pp. 190–3.

[15] Li, J., A. Skjellum, and R. D. Falgout, "A Poly-Algorithm for Parallel Dense Matrix Multiplication on Two-Dimensional Process Grid Topologies," *Concurrency, Practice and Experience,* 9(5) 345–389, May 1997.

[16] van de Geijn, R., *Using PLAPACK: Parallel Linear Algebra Package*, The MIT Press, 1997.

[17] van de Geijn, R. and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," *Concurrency: Practice and Experience,* 9(4) 255–274, April 1997.