Comparison of Scalable Parallel Matrix Multiplication Libraries*

Steven Huss-Lederman Elaine M. Jacobson Anna Tsao

Supercomputing Research Center, 17100 Science Drive,
Bowie, MD 20715

Abstract

*This paper compares two general library routines for performing parallel distributed matrix multiplication. The PUMMA algorithm utilizes block scattered data layout, whereas Bi*MM*eR utilizes virtual 2-D torus wrap. The algorithmic differences resulting from these different layouts are discussed as well as the general issues associated with different data layouts for library routines. Results on the Intel Delta for the two matrix multiplication algorithms are presented.*

1. Introduction

Matrix multiplication is a standard algorithm that is an important computational kernel in many applications including eigensolvers [3] and LU factorization [15]. Utilizing matrix multiplication is one of the principal ways of achieving high efficiency block algorithms in packages such as LAPACK [2]. The BLAS 3 routines were added to achieve this block performance on computers, and optimized versions are available on most serial machines [10]. For matrix multiplication, the BLAS 3 routine XGEMM is available to compute $C = \beta C + \alpha \cdot op(A) \cdot op(B)$, where $op(X)$ is either the matrix or its transpose. For analogous reasons, utilization of matrix multiplication-based algorithms on parallel computers also yields efficient implementations [6]. Thus, there is a desire to have a widely available library-quality implementation of XGEMM on parallel machines.

Several algorithms have been proposed for parallel, distributed memory matrix multiplication including 1D-systolic [15], 2D-systolic or Cannon's algorithm [5, 15], Broadcast-Multiply-Roll [12, 13, 14] and the Transpose algorithm [18]. These algorithms have been implemented on a variety of machines. However, a single, standard, library-quality parallel version of XGEMM is not yet available. To satisfy the diverse needs of users, a library version must be able to deal with arbitrary matrix sizes and numbers of processors. Furthermore, the routines must run "out of the box" with reasonably high efficiency on the wide variety of distributed memory machines that are available. Unfortunately, the routines implemented to date are often not general, and even the general routines have only achieved high performance by using machine-specific implementations.

This paper will focus on two efforts, PUMMA [7, 8] and BiMMeR [17], both aimed at providing a library-quality implementation of distributed matrix multiplication that retains the functionality of XGEMM on serial computers. Since these implementations are for distributed memory computers, the data distribution (decomposition) is critical to the implementation. The regular patterns in matrix multiplication in fact do not require complicated data distributions. However, as a library routine, the matrix multiplication code must be able to deal with the desired data distribution of the calling application code. It is possible to transform between data distributions when entering the matrix multiplication routine, but this can be expensive. Thus, it is advantageous for the matrix multiplication routine to utilize the data distribution that is desirable in the calling applications. PUMMA uses the block scattered data layout, whereas BiMMeR uses virtual 2-D torus wrap data layout. Choosing different data layouts leads to different algorithmic choices that affect the design of a matrix multiplication library routine.

The remainder of this paper is organized as follows. Section 2 reviews the broadcast-multiply-roll algorithm for square processor meshes. Section 3 extends the algorithm to nonsquare meshes and discusses the algorithmic consequences of the two different data

---

*This paper is PRISM Working Note #13, available via anonymous ftp to `ftp.super.org` in the directory pub/prism.

This work was partially supported by the Applied and Computational Mathematics Program, Advanced Research Projects Agency, under Contract DM28E04120. Access to the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium was provided by NSF.

Typeset by $\mathcal{AMS}$-TEX

layout choices in PUMMA and BiMMeR. Section 4 elaborates on the general data distribution question for libraries and application codes. Finally, Section 5 presents results and Section 6 gives a summary and conclusions.

2. Square Meshes

Both PUMMA and BiMMeR use variants of the broadcast-multiply-roll algorithm [14]. This paper will present the algorithms on a 2D toroidal mesh, since this is the logical topology desired as noted in [14]. If the physical machine topology differs from the algorithmic topology, then one needs to embed the 2D toroidal mesh into the physical machine topology (for example, a hypercube).

For initial presentation, the algorithm will be shown on a square $P \times P$ mesh with square matrices of order N. For simplicity of discussion, we assume that $P|N$, so that each processor contains a contiguous $N/P \times N/P$ subblock of the matrix. We present the algorithm from the processor point-of-view, which emphasizes the location of the data on each processor and how it moves between processors. Figure 1 gives an example where $N = 12$ and $P = 3$. The numbers down (across) the rows (columns) represent the row (column) entry of the matrix. The values contained in each box represent the (x, y) processor location in the 2D mesh. For example, processor $(1, 2)$ has matrix elements $[4, 8], [5, 8], \ldots, [4, 9], \ldots$; each processor starts with the same elements for matrices A, B, and C.

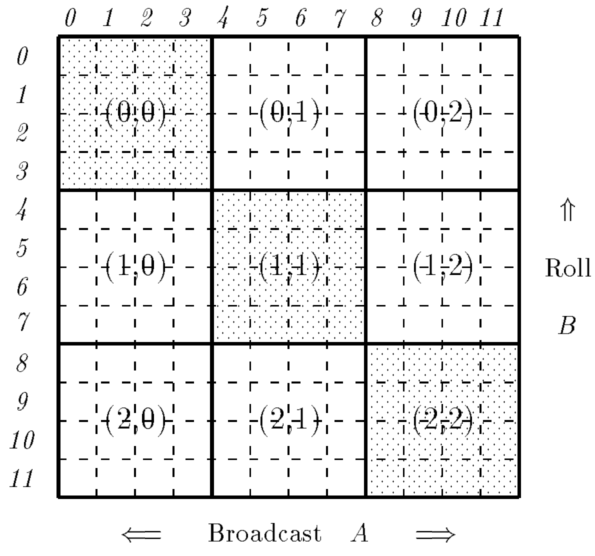

Figure 1. 3×3 block layout

Of the four matrix multiplication variants performed by XGEMM, this paper will focus on the most common variant, $C = \beta C + \alpha AB$. For ease of presentation, the scalar multipliers β and α are assumed to equal 1 so the problem reduces to the straightforward case of $C = C + AB$. As usual, the column indices of A must align with the row indices of B for a multiplication to occur. Since the data is physically distributed across the machine, computation of each element of C requires interprocessor communication. The algorithm will proceed in 3 $[P]$ stages. In the first stage, a subblock of A is broadcast across each row of the mesh and is represented by the shaded data in Figure 1. For example, in row 1 of the mesh the subblock of A on processor $(1, 1)$ is broadcast to processors $(1, 0)$, $(1, 1)$, and $(1, 2)$. Notice that each processor in row 1 of the mesh has elements of rows 4 through 7 of matrix B, and the block of A broadcast contains elements in columns 4 through 7. Thus, a local matrix multiplication can now occur on each node. The next step is to circularly shift (roll) the subblocks of B up the columns of the mesh. For example, the data for B on processor (0,2) will be shifted onto processor (2,2). This completes one stage of the algorithm. Each processor row now contains new columns of B. The algorithm continues with the broadcast of a different block of A followed by a matrix multiplication, and thus the algorithm continues. A more detailed presentation of broadcast-multiply-roll can be found in [14].

The broadcast-multiply-roll algorithm has several desirable properties, some of which are given below.

- A local matrix multiplication is performed on each node. This means that an optimized (often assembly-coded) sequential version of XGEMM can be used on each node.
- Each local matrix multiplication uses all the data available on the node, which maximizes the granularity of the computation. As a general rule, the speed of the local matrix multiplication increases as the number of elements multiplied increases.
- Both the broadcast and roll communication steps send the entire subblock of A and B, respectively. Since these operations maximize the amount of data sent, the number of startups is minimized and the data transfer rate should be maximized.

Generalizations of the algorithm to nonsquare meshes should strive to maintain as many of these desirable properties as possible.

The algorithm has so far been presented for a contiguous data layout that is sufficient for matrix multiplication alone. However, many other algorithms utilize a scattered data layout to get better load balancing properties by placing nearby rows/columns of the matrix on different processors [9]. Figure 2 illustrates the block scattered data layout for the same set of parameters as in Figure 1. As shown, successive rows (columns) of the matrix are placed in successive rows (columns) of the physical processor mesh.

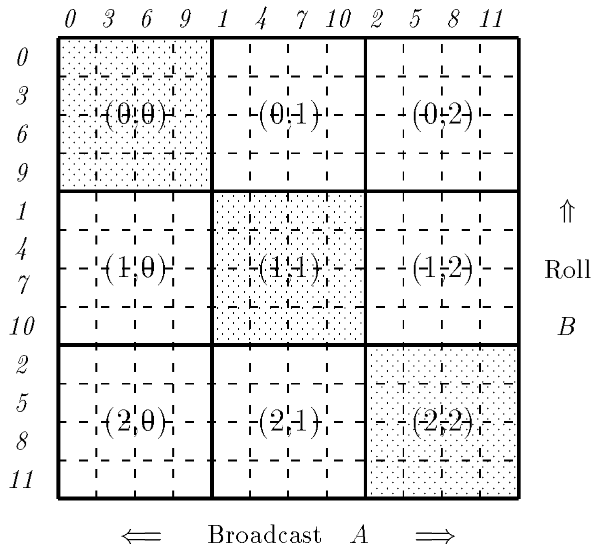

Figure 2. 3×3 block scattered layout

In addition to scattering, many algorithms also want several contiguous rows/columns of the matrix on the same node so local block operations can be utilized. The indices that represented a single row/column in Figure 2 can now represent blocks of rows/columns. Thus, [row, column] index $[i, j]$ does not represent matrix element $[i, j]$ but rather block $[I, J]$ of the matrix. Each block contains b_r rows and b_c columns of the matrix, and this is often the block size of the local computation on each node. Note that the contiguous block case can still be achieved when $b_r = b_c = N/P$. More details can be found in [9].

It can be seen that the indices of the blocks and the indices within each block align in the same manner as for the single row/column case [8]. Thus, the matrix multiplication algorithm remains the same for the block scattered data layout on square meshes; only the specific elements multiplied on each processor change. Since a scattered data layout is desirable for many algorithms, it is used in both the PUMMA and BiMMeR matrix multiplication routines. This avoids the potentially expensive operation of transforming the layout when the matrix multiplication routine is called. The rest of this paper will present matrix multiplication with a scattered data layout.

3. Nonsquare Meshes

A general library routine has to be able to deal with an arbitrary number of processors, so the algorithm must work on a nonsquare $P \times Q$ processor mesh. This section presents the generalizations made by PUMMA and BiMMeR from the square to the nonsquare processor mesh case. Both algorithms were designed to retain the positive attributes of the square mesh algorithm, that is, large granularity in the local matrix multiplication and minimal costs for communications.

PUMMA utilizes the block scattered data layout as shown in Figure 3 for a 4×3 mesh and order 24 matrices. Each successive row (column) block of data is placed in the next row (column) physical processor location. As shown in Figure 3, the block row indices given as the right-hand column of numbers (unbracketed) are not in the same order as the block column indices, as was the case for square meshes (c.f., Figure 2). The algorithm presented for square meshes must be modified to get the column indices of A to align with the row indices of B so that a local matrix multiplication can be performed.

For ease of presentation, the algorithms will be presented for the case where $P > Q$ and $GCD(P, Q) = 1$, where GCD is the greatest common divisor. Let LCM represent the least common multiple of P and Q. PUMMA begins by shuffling the rows of B on each node so adjacent rows are the same as the columns of A that will be multiplied and communicated at each stage. In Figure 3, the shuffled rows of B are given as the bracketed values to the left of the normal values for block scattered data layout. This shuffling is a local node operation that is performed once.

PUMMA now performs 4 [P] stages. At the start of each stage the columns of A to be communicated are gathered. Each processor gathers into contiguous memory the columns of A corresponding to the rows of B it has in the given row of the processor mesh. These are represented by the shaded columns in Figure 3. Since each is in contiguous memory, the gathered columns of A that align with the gathered rows of B can now be multiplied on each node. For example, on node $(0, 2)$, columns 8 and 20 of A are multiplied by the elements in rows 8 and 20 of B. Since all the processors in the same row of the mesh have the same rows of B, the gathered columns of A in each processor can now be rolled to the left and another matrix multiplication performed. Again, on node $(0, 2)$, the rows of A/columns of B involved would now be 0 and 12. One more roll and multiply completes the cycle. This series of 3 [Q] multiplications and rolls replaces the broadcast and multiply in the square mesh case. As in the square mesh case, new rows of B are now required so all the data for B on each node is rolled upward. (Since $P > Q$ and the matrices are square, the number of columns of A is greater than the number

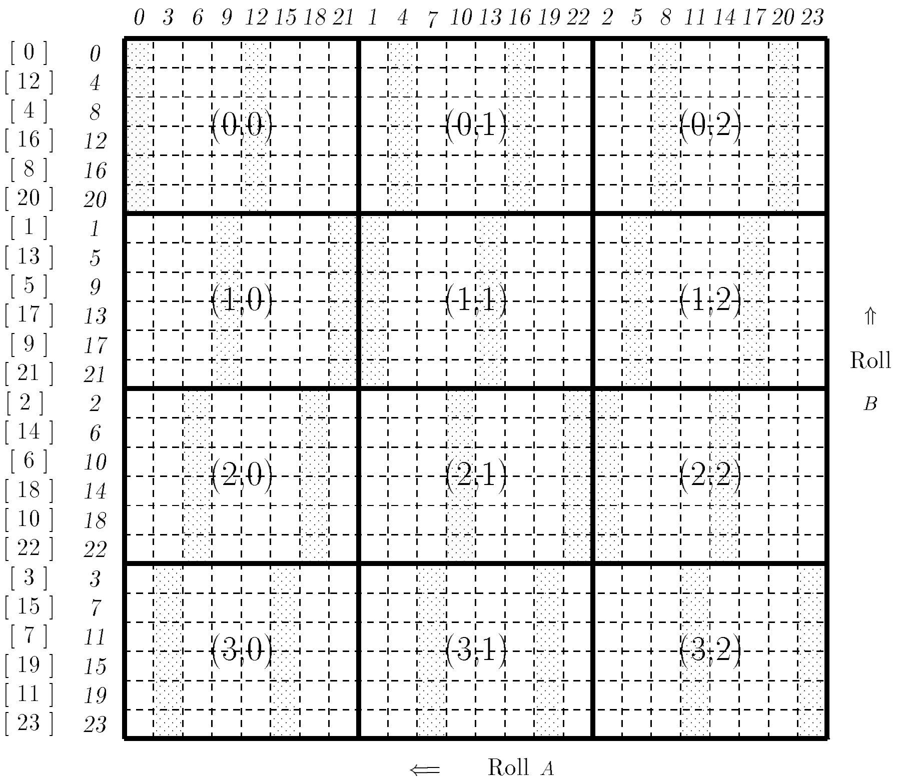

Figure 3. 4×3 **block scattered layout used by PUMMA**

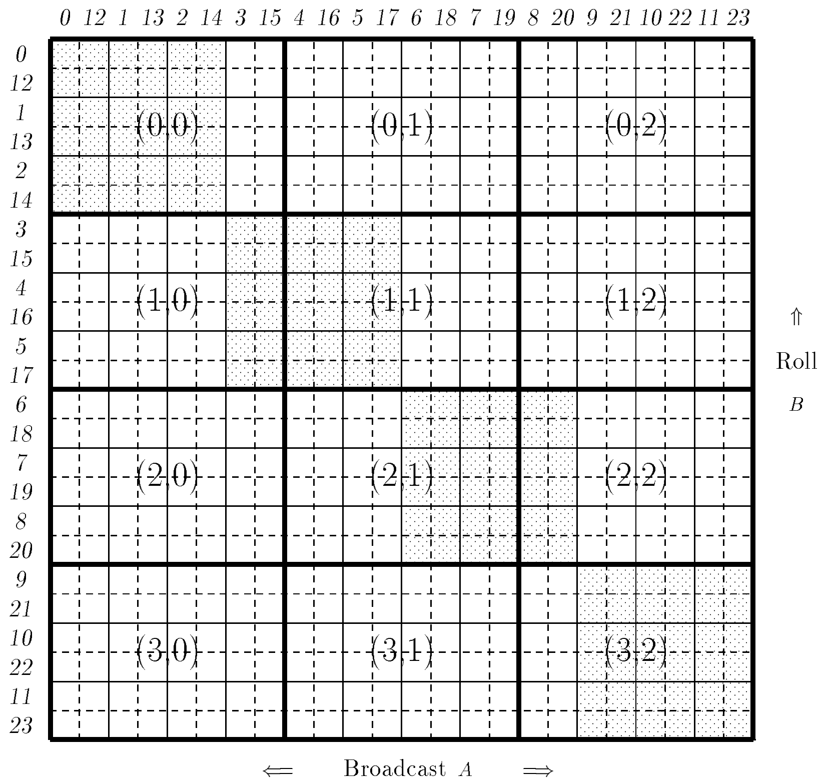

Figure 4. 4×3 **virtual 2-D torus wrap layout used by BiMMeR**

of rows of B on a node, so the number of rows of B limits the size of the multiplications that can be done before B must be rolled.) This completes one stage of the algorithm. Repetition for all 4 $[P]$ stages completes the matrix multiplication. Details of the PUMMA algorithm and other cases are given in [8].

A summary of the algorithmic changes made in PUMMA to deal with this nonsquare mesh is given below.

- The rows of B on each node must be shuffled at the start.
- Two $[N/LCM]$ columns of A must be gathered on each node at the start of each of the 4 $[P]$ stages.
- The local matrix multiplication only multiplies 2 columns of A and rows of B.
- The 2 gathered columns of A are rolled instead of broadcast.

BiMMeR utilizes the virtual 2-D torus wrap data layout shown in Figure 4 [4]. In this data layout the 4×3 mesh is virtualized to a 12×12 $[LCM \times LCM]$ mesh by placing 3×4 virtual processors on each physical node. The virtual processors are separated by thin solid lines in Figure 4. Each successive row (column) of the matrix is placed in the next virtual processor. (In general, any symmetric permutation of the row and column indices can be used.) Notice that the row and column indices still align as they did in the square mesh case.

The BiMMeR algorithm for nonsquare meshes is the same basic algorithm as it was for square meshes. For this example, it proceeds in 4 $[P]$ stages. First the columns of A that align with the rows of B in each row of the mesh are broadcast. These blocks are shaded in Figure 4. Since $P > Q$ and the matrices are square, the number of columns of A broadcast is the number of rows of B in that row of the mesh. Notice that the columns of A to be broadcast in rows 2 and 3 of the mesh lie in two processors, so two smaller broadcasts need to be done. Each node can now do a matrix multiplication of the broadcast A with B. Finally, B is rolled up. This completes one stage of the BiMMeR algorithm. Details of the BiMMeR algorithm can be found in [17].

A summary of the algorithmic changes made in BiMMeR to deal with this nonsquare mesh is as follows.

- The columns of A to be broadcast can lie in at most two processors, sometimes necessitating two smaller broadcasts. The number of columns of A broadcast (and multiplied) is limited by the number of rows of B in each row of the mesh.

The different algorithmic choices made for PUMMA and BiMMeR can now be compared for this nonsquare mesh case. First, PUMMA must do a gather of the rows of B on each node. There is no need for such an operation in BiMMeR. Second, the columns of A to be multiplied and rolled in PUMMA must be gathered on each node. Since A is stored by column, there is no need for this operation in BiMMeR for the example given. However, if $P < Q$, then the rows of B would be broadcast, and this would require a local gather in BiMMeR before the broadcast could be performed. Third, the matrix multiplications in PUMMA are for 2 $[N/LCM]$ rows of B. In BiMMeR the matrix multiplications are for all the rows of B. Thus, the granularity of the matrix multiplications is larger in BiMMeR, and the local multiplications are expected to be more efficient. Finally, PUMMA does 2 $[Q-1]$ rolls of 2 $[N/LCM]$ columns of A on each node. The total data sent over all rolls is 4 $[(Q-1)N/LCM]$ columns of A. BiMMeR, on the other hand, does a broadcast of 6 $[N/P]$ columns of A. Thus, in communicating A, BiMMeR transmits more data in its broadcast than PUMMA does in its rolls. In addition, the broadcasting algorithm used on the Intel Delta takes $\lceil log_2(Q) \rceil$ stages, whereas the roll can be done in 2 stages [16]. (On some other systems, it is possible to do both operations in a single stage of comparable time.) Except for very small meshes, $\lceil log_2(Q) \rceil$ is larger than 2. On the other hand, the fact the PUMMA does multiple rolls instead of the single broadcast means it incurs more communications startup, which might have an effect on small problem sizes.

For this example, there is a tradeoff between the higher granularity in the matrix multiplications in BiMMeR versus the lower amount of data communicated in PUMMA. PUMMA must also do two gathers of local node data, which BiMMeR does not have to do. The example given for a 4×3 mesh and square matrices is meant to be representative of some of the algorithmic differences that resulted from the choice of different data layouts in PUMMA and BiMMeR. Showing different nonsquare meshes would show other algorithmic choices made between PUMMA and BiMMeR. The detailed comparison between these two algorithms can be derived from the information contained in the papers on PUMMA [8] and BiMMeR [17] and is beyond the scope of this paper.

4. Data Layout Issues and Libraries

The example given comparing the block scattered data layout of PUMMA with the virtual 2-D torus wrap data layout of BiMMeR shows that the choice of data layout has an impact on which algorithmic alternatives are taken. The properties of the possible data layouts lead to different tradeoffs and advantages when it comes to algorithm design. These two data layouts represent only some of the many possible permutations of the rows and columns that could be utilized [11]. Each of these possibilities has potential advantages and disadvantages associated with them.

BiMMeR uses a data layout whose properties differ from block scattered in a number of important ways. For example, the virtual square mesh presents a consistent algorithmic topology for coding even when the physical mesh that the code runs on is changed. As a result, the programmer can always code for the simpler case of a square mesh. This allowed the BiMMeR algorithm to be virtually unchanged when going from square to nonsquare meshes. Additionally, virtual 2-D torus wrap allows conformal (symmetric permutation) row and column indices, as was done in the BiMMeR algorithm. This can greatly simplify the coding of algorithms that work on symmetric matrices or that rely on symmetric placement of data onto processors. Some of the ease of programming can be seen by the fact that diagonal elements of the matrix lie along diagonal virtual processors in the 2D mesh. The example of the previous section and this discussion only touch on some of the properties of virtual 2-D torus wrap; a more detailed discussion can be found in [4].

It is clear that there are several potential data layouts for distributed memory algorithms. Each layout will have its own advantages and disadvantages, and more investigation is necessary to determine which alternatives have enough merit to be worthy of support. Early determination of the viable choices is important for several reasons. First, it is a potentially expensive operation to have to transform between data layouts as different library routines are used. If libraries are written for a diverse set of layouts, then the user may be forced into accepting such overheads to utilize library routines. Second, from a library writer's standpoint, it is still a major effort to produce a library for a given data layout. Thus, recoding of libraries to support a varied set of layouts will reduce resources for developing new libraries, or simply will not occur. Finally, data layouts are starting to be incorporated into high level languages. At the current time the High Performance Fortran (HPF) standard only fully and directly supports the block scattered data layout [1]. Many applications users may only utilize the data layouts supported in high level languages to achieve ease of usage. Choosing which data layouts to support before the alternatives are investigated may discourage such investigation and limit the number of potential users of worthy alternative data layouts.

5. Results

On square meshes, both PUMMA and BiMMeR reduce to the standard broadcast-multiply-roll algorithm presented in Section 2. Thus, except for implementation differences, the two codes should yield the same results on square meshes. Table 1 gives the MFLOPS per node where the amount of data per node is held fixed while varying the square mesh size. These and all results given are for a “dedicated” machine and are reproducible when run at different times. It is seen that BiMMeR is faster on larger meshes, whereas PUMMA is faster on smaller meshes. It was unclear why this was the case, so the times for each of the three computational kernels were measured. The results for the 16×16 mesh with local matrix size of 200×200 are given in Table 2. The BiMMeR results show that there is not much variation in times between nodes, whereas the PUMMA algorithm shows a large variation. Note that the total times for PUMMA show less deviation, indicating that individual nodes are fast for some kernels and slow for others.

		Elements per node		
Mesh	Alg.	200^2	50^2	10^2
16×16	BiMMeR	24.7	10.9	0.69
	PUMMA	22.3	10.4	0.65
12×12	BiMMeR	24.5	10.3	0.70
	PUMMA	22.0	11.3	0.67
8×8	BiMMeR	26.0	12.1	0.77
	PUMMA	26.4	12.2	0.74
4×4	BiMMeR	27.4	13.6	0.88
	PUMMA	28.1	14.2	0.92

Table 1. Comparison of PUMMA and BiMMeR on square meshes for fixed local data size. Values are given as MFLOPS per node for the slowest node. (Example: the PUMMA algorithm sustained 22.0 MFLOPS per node for local matrices of size 200×200 on a 12×12 mesh, which translates into a total performance of 3.2 GFLOPS for the matrix of global size 2400×2400.)

The analysis of Section 3 indicates that PUMMA and BiMMeR have different algorithmic tradeoffs for nonsquare meshes where the $GCD(P, Q) = 1$, and additional differences would be seen for other cases. However, given the results seen for square meshes, it is difficult to draw conclusions for nonsquare meshes. Table 3 contains results for some representative non-

Part	Alg.	Max	Min	Ave
"Broadcast"	BiMMeR	2.19	2.14	2.16
	PUMMA	3.10	0.71	2.32
Multiply	BiMMeR	7.16	7.11	7.12
	PUMMA	9.74	7.11	7.13
Roll	BiMMeR	1.07	1.02	1.04
	PUMMA	3.48	0.95	1.72
Total	BiMMeR	10.3	10.3	10.3
	PUMMA	11.5	11.1	11.2

Table 2. Time in seconds spent in each of the three main computational kernels for a 16×16 mesh on a local problem size of 200×200. The maximum, minimum, and average time for all nodes is given.

		Elements per node		
Mesh	Alg.	$\approx 200^2$	$\approx 50^2$	$\approx 10^2$
16×15	BiMMeR	26.1	8.7	0.24
	PUMMA	21.5	6.6	0.13
16×12	BiMMeR	26.7	9.8	0.39
	PUMMA	21.8	10.2	0.49
16×8	BiMMeR	28.7	14.4	0.80
	PUMMA	23.4	14.1	0.77
8×7	BiMMeR	25.6	15.3	0.54
	PUMMA	27.1	15.3	0.30
8×6	BiMMeR	27.0	12.5	0.64
	PUMMA	21.8	11.9	0.51
8×4	BiMMeR	29.6	15.5	0.93
	PUMMA	30.1	15.8	0.90

Table 3. Comparison of PUMMA and BiMMeR for nearly fixed data size per node. Values are given as MFLOPS per node.

square mesh cases. Over the various mesh sizes, we adjust the matrices being multiplied to maintain a nearly fixed amount of data per node. More specifically, we strove to maintain optimal load balance, a choice beneficial to both algorithms. This was sometimes incompatible with maintaining the same local dimensions over all mesh sizes. When a choice was necessary we opted in favor of load balancing, and adjusted the matrices accordingly. Sizes are still fixed on a particular mesh configuration and nearly fixed across all mesh sizes. These results are included for completeness, but all of these timings are recent results and still under investigation.

6. Conclusions

This paper contains a comparison of two general library routines for performing distributed memory parallel matrix multiplication. PUMMA uses a block scattered data layout, whereas BiMMeR uses virtual 2-D torus wrap. The differences in these data layouts lead to different algorithmic tradeoffs in the design of a matrix multiplication routine.

The issues associated with data layout alternatives are relevant to both library writers and users of distributed memory parallel computers. There are many data layouts possible, and research needs to be performed to determine which are the most advantageous. Once this is determined, library writers and high level language developers will know where to invest their efforts.

Acknowledgment

The authors would like to thank Jaeyoung Choi, Jack Dongarra, and David Walker for helpful discussion about the PUMMA software.

References

[1] High Performance Fortran Forum, *High Performance Fortran Language Specification Version 1.0*, 1993.

[2] Anderson, E., C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, S. Hammarling, & W. Kahan, *Prospectus for an extension to LAPACK: A portable linear algebra library for high-performance computers*, CS-90-118, U of Tennessee (1990).

[3] Auslander, L. & A. Tsao, *A divide and conquer algorithm for the eigenproblem via complementary invariant subspace decomposition*, Technical Report SRC-TR-89-003, Supercomputing Research Center (1989).

[4] Bischof, C. H., S. Huss-Lederman, E. M. Jacobson, X. Sun, & A. Tsao, *On the impact of HPF data layout on the design of efficient and maintainable parallel linear algebra libraries*, (private communication).

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

[6] Choi, J., J. J. Dongarra, R. Pozo, & D. W. Walker, *ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers*, Proceedings, Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia, October 19–21, 1992), IEEE Computer Society Press, Los Alamitos, California, 1992, pp. 120–127.

[7] Choi, J., J. J. Dongarra, & D. W. Walker, *Level 3 BLAS for distributed memory concurrent computers*, CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing (Saint Hilaire du Touvet, France, September 7–8, 1992), Elsevier Science Publishers, 1992.

[8] Choi, J., J. J. Dongarra, & D. W. Walker, *PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers*, ORNL/TM-12252, Oak Ridge National Laboratory (April 1993).

[9] Choi, J., J. J. Dongarra, & D. W. Walker, *The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers*, CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing (Saint Hilaire du Touvet, France, September 7–8, 1992), Elsevier Science Publishers, 1992.

[10] Dongarra, J. J., J. Du Croz, S. Hammarling, & I. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Transactions on Mathematical Software **16** (1990), 1–17.

[11] Falgout, R. D., A. Skjellum, S. G. Smith, & C. H. Still, *The Multicomputer Toolbox approach to concurrent BLAS and LACS*, Proceedings, Scalable High Performance Computing Conference (Williamsburg, VA, April 26–29, 1992), IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 121–28.

[12] Fox, G., *Domain decomposition in distributed and shared memory environments*, Lecture Notes in Comput. Sci. 297: Proc., 1987 Intl. Conf. Supercomputing, Springer-Verlag, Berlin, 1987, pp. 1042–1073.

[13] Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, & D. Walker, *Solving Problems on Concurrent Processors, Vol. I*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[14] Fox, G., S. Otto, & A. Hey, *Matrix algorithms on a hypercube I: matrix multiplication*, Parallel Computing **4** (1987), 17–31.

[15] Golub, G. & C. F. Van Loan, *Matrix Computations, 2nd ed.*, Johns Hopkins University Press, Baltimore, 1989.

[16] Huss-Lederman, S., E. M. Jacobson, A. Tsao, & G. Zhang, *Optimizing Communication Primitives on the Intel Touchstone Delta*, Technical Report, Supercomputing Research Center (1993) (to appear).

[17] Huss-Lederman, S., E. M. Jacobson, A. Tsao, & G. Zhang, *Matrix Multiplication on the Intel Touchstone Delta*, Proceedings, Sixth SIAM Conference on Parallel Processing for Scientific Computing (Norfolk, Virginia, March 22-24, 1993) (R. F. Sincovec, eds.), SIAM, Philadelphia, 1993, (also PRISM Working Note #7).

[18] Lin, C. & L. Snyder, *A Matrix Product Algorithm and its Comparative Performance on Hypercubes*, Proceedings, Scalable High Performance Computing Conference (Williamsburg, VA, April 26–29, 1992) (Stout, Q. & M. Wolfe, eds.), IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 190-3.