MATRIX MULTIPLICATION ON THE INTEL TOUCHSTONE DELTA*

STEVEN HUSS-LEDERMAN§
ELAINE M. JACOBSON§
ANNA TSAO§

Supercomputing Research Center
17100 Science Drive
Bowie, MD 20715-4300

AND

GUODONG ZHANG†,§

CONVEX Computer Corporation
3000 Waterview Parkway
Richardson, TX 75083-3851

SRC-TR-93-101
(revised)

Summary

Matrix multiplication is a key primitive in block matrix algorithms such as those found in LAPACK. We present results from our study of matrix multiplication algorithms on the Intel Touchstone Delta, a distributed memory message-passing architecture with a two-dimensional mesh topology. We analyze and compare three algorithms and obtain an implementation, BiMMeR, that uses communication primitives highly suited to the Delta and exploits the single node assembly-coded matrix multiplication. Our algorithm is completely general, i.e., able to deal with various data layouts as well as arbitrary mesh aspect ratios and matrix dimensions, and has achieved parallel efficiency of 86%, with overall peak performance in excess of 8 Gflops on 256 nodes for an 8800×8800 matrix. We describe BiMMeR's design and implementation and present performance results that demonstrate scalability and robust behavior over varying mesh topologies.

---

*This paper is PRISM Working Note #14, available via anonymous ftp to `ftp.super.org` in the directory `pub/prism`.

†This research was done while author was at Department of Mathematics, University of Maryland, College Park, MD 20742.

§This research is being partially supported by ARPA's Applied and Computational Mathematics Program, under contract DM28E04120. Access to the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium is being provided by NSF.

1991 *Mathematics Subject Classification.* 65Y05, 65Y10.

Key words and phrases. matrix multiplication, scalable algorithm, parallel algorithm, distributed memory, two-dimensional mesh topology, MIMD.

1. Introduction

The multiplication of two matrices is one of the most basic operations of scientific computing. Versions for serial computers have long been based on optimized primitives embodied in the kernels of standard software packages, such as LINPACK [14]. A stable and fairly uniform set of appropriate kernels well-suited to most serial machines makes these implementations hard to beat. This is not yet the case for parallel architectures, where the set of primitives can so readily change from one machine to another, but the block algorithms of LAPACK ([1], [2], [6]) and ScaLAPACK [9] are one step in this direction. In this paper we study algorithms for matrix multiplication on distributed-memory message-passing architectures with a two-dimensional mesh topology and develop an implementation on one such machine, the i860-based Intel Touchstone Delta.

While some studies of distributed matrix multiplication have been made ([10],[18],[31]), we felt that a separate focused effort was required in our work for several reasons. First, we are developing scalable algorithms for eigenproblems as part of the Parallel Research on Invariant Subspace Methods (PRISM) project, composed of researchers from Argonne National Laboratory, Supercomputing Research Center, University of California at Berkeley, and the University of Kentucky. In particular, we are currently investigating the symmetric Invariant Subspace Decomposition Algorithm (symmetric ISDA), a new technique for finding all the eigenvalues and eigenvectors of a dense real symmetric matrix, which is based predominantly on matrix multiplication operations ([4],[23]). As such, an obvious and crucial initial goal for this effort was to develop a highly optimized code for this key primitive. Second, our present target machine represented a new architecture on which the costs of data motion were not clear. Just as reducing the cost of hierarchical memory data access is crucial to the design of efficient algorithms for vector and shared-memory machines [13], the same is true in a multicomputer environment, where we want to reduce the cost of data movement across the distributed memory. Our design philosophy is therefore to first understand the behavior of the machine and then to use this knowledge to choose and/or design high-level algorithms well-matched to its capabilities. Thus, benchmarks of the Delta architecture and comparative cost analysis across a variety of algorithms were needed in order to make valid implementation decisions. Third, we desired generality over arbitrary matrix dimensions and processor mesh configurations, and at the time our work began this functionality was not readily available in any existing Delta code. Finally, while our orientation was machine-specific and aimed at optimization rather than portability, the resulting kernel could still be useful to future portable distributed-memory software such as ScaLAPACK [9], being an optimized version of one of the core Level 3 BLAS building blocks [12]. Indeed, BiMMeR parallels PUMMA, the prototype matrix multiplication software of ScaLAPACK, but differs from it in several respects. A comparison can be found in [21].

Our analysis of matrix multiplication on the Intel Delta assumes an optimized single-processor matrix multiplication subroutine and uses the i860 assembly-coded implementation of DGEMM for this purpose [27]. Thus, we can focus our efforts on gaining performance through careful choice of distributed algorithm, distributed data layout, and message-passing primitives. We describe architectural and performance details of the Delta relevant to our purposes and, in particular, review its performance on communication primitives that could be useful in designing matrix multiplication algorithms. Based on these results we analyze three matrix multiplication algorithms and their suitability for the Delta. We describe our design objectives, decisions, and implementation, and present performance results that demonstrate scalability and robust behavior even on non-optimal mesh configurations.

The remainder of this paper is organized as follows:

- Section 2 presents our design objectives and candidate algorithms,
- Section 3 discusses relevant details on the Intel Delta architecture and its performance on required communication primitives,
- Section 4 examines costs for each algorithm, makes design decisions, and presents an overview of the BiMMeR implementation,

◦ Section 5 presents performance results for BiMMeR,
◦ and Section 6 offers a summary and conclusions.

2. Algorithms

In this section we discuss our design objectives and candidate algorithms, highlighting their computational granularity and key communication primitives. Our objectives for this initial effort were straightforward:

(1) Provide a highly efficient algorithm for use by ISDA.
(2) Strive for high performance on large square matrices.
(3) Show robust performance for mesh configurations with poor aspect ratios.
(4) Adjudicate decisions in favor of expected capabilities of future machines.

Besides our own use for the resulting software in the ISDA application, we were also aware of other researchers who would welcome the availability of such a matrix multiplication routine, especially one that could exhibit high performance on as large a matrix as possible. Thus, *generality, scalability,* and high performance on *large problems* were primary goals. As usual, the crucial issue for high efficiency is data locality; we must strive for maximum use of data in floating point computations, while minimizing costs of data transfer. To this end, we utilize the highly optimized assembly-coded DGEMM on single nodes. It is capable of sustaining 36.5 Mflops (empirically determined using Release 1.5 of Intel NX/M OS and related software) of the 40 Mflops possible for standard double precision matrix multiplication. The likelihood of attaining this rate increases sharply as the size of the matrices is increased, as shown in Figure 1. Thus, our first general rule is to *strive for large granularity in local computations.*

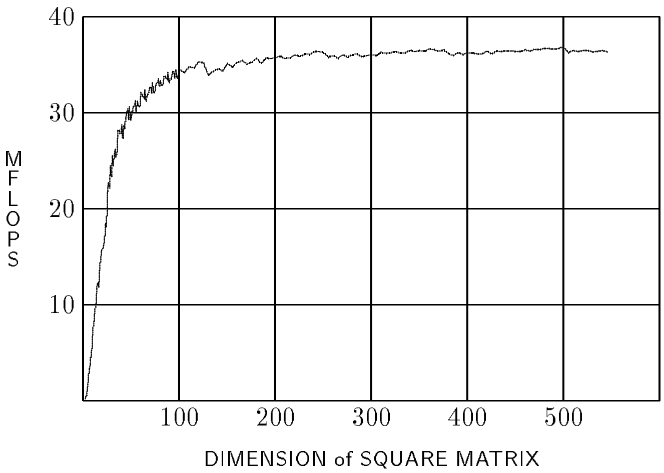


Figure 1. DGEMM on Single Nodes

We consider three standard algorithms: 1D-systolic (1D) [20], 2D-systolic or Cannon's algorithm (2D) ([8],[20]), and Broadcast-Multiply-Roll (BMR) ([17], [18], [19]). We quickly review the algorithms while examining their communication requirements. Descriptions of the algorithms could be given from the *matrix point-of-view*, i.e., seeing first the matrix layout and marking entries with the processor location where the entry is stored. We prefer the *processor point-of-view*, where we primarily see the processor layout and collect matrix entries in their actual storage locations. We found this perspective to be more illuminating in studying the actual behavior of the distributed algorithms and more helpful as an aid to optimization. Although our implementation can actually deal with various data layout schemes by means of a user-passed function providing this information, for clarity we will first review the algorithms assuming contiguous data layouts. We will return to the important issue of data distribution after the basic algorithms are explained.

Given matrices A, B, we wish to compute $C = AB$ on a $p \times q$ two-dimensional processor mesh. Again, for simplicity, we further assume that A, B, C are square, of dimension $N \times N$. For load balancing we distribute the matrix elements as evenly as possible across the processor memories and assume that after C is computed A and B end up in their original locations.

2.1. 1D Systolic.

We can embed an appropriate Hamiltonian path in the $p \times q$ mesh and view it as a ring of pq processors. For clarity, assume $N = rpq$, and let A, B, C be partitioned into 1D panels, i.e., panels of $r = N/pq$ full columns, $A = [A_0 \quad A_1 \quad \ldots \quad A_{pq-1}]$, $B = [B_0 \quad B_1 \quad \ldots \quad B_{pq-1}]$, $C = [C_0 \quad C_1 \quad \ldots \quad C_{pq-1}]$, with processor $k, k = 0, \ldots, pq-1$, holding A_k, B_k at the start of the algorithm and, in addition, C_k at the conclusion. Let $B_{ik}, i = 0, \ldots, pq-1$, represent rows ir to $(i+1)r - 1$ of panel B_k. Then C_k is calculated in node k by accumulating partial sums of the form $A_i B_{ik}$, i.e., $C_k = \sum_{i=0}^{pq-1} A_i B_{ik}$, while performing a cyclic shift, or roll, of the full panels of A around the ring. Thus, the first multiplication occurs before anything else. Then A is rolled, and another summand is added to the accumulating sum, as shown in Figure 2. This continues until $pq - 1$ rolls have been done, at which point C_k is fully computed. One more roll will put A back in its original position.

The single required communication primitive is a *roll*. Notice that the algorithm performs a total of pq rolls, each on N^2/pq data elements, and single-node multiplications are on matrices of size $N \times N/pq$ and $N/pq \times N/pq$.

node k	A_i	B_k	summand of C_k
0	A_0	B_0	$A_0 B_{00}$
1	A_1	B_1	$A_1 B_{11}$
2	A_2	B_2	$A_2 B_{22}$

↓ roll A's

node k	A_i	B_k	summand of C_k
0	A_2	B_0	$A_2 B_{20}$
1	A_0	B_1	$A_0 B_{01}$
2	A_1	B_2	$A_1 B_{12}$

FIGURE 2. THE 1D SYSTOLIC ALGORITHM

2.2. 2D Systolic.

The two-dimensional analogue of the above systolic algorithm allows us to distribute the data in two directions. We view the machine as a $p \times q$ mesh with wraparound. For clarity, assume for now that $p = q$ and $N = rp$, and partition A, B, and $C = AB$ into 2D blocks, i.e., blocks of size $r \times r$,

$$A = \begin{bmatrix} A_{00} & \ldots & A_{0,p-1} \\ \vdots & \ddots & \vdots \\ A_{p-1,0} & \ldots & A_{p-1,p-1} \end{bmatrix}, \; B = \begin{bmatrix} B_{00} & \ldots & B_{0,p-1} \\ \vdots & \ddots & \vdots \\ B_{p-1,0} & \ldots & B_{p-1,p-1} \end{bmatrix}, \; C = \begin{bmatrix} C_{00} & \ldots & C_{0,p-1} \\ \vdots & \ddots & \vdots \\ C_{p-1,0} & \ldots & C_{p-1,p-1} \end{bmatrix},$$

with processor (i, j), $i, j = 0, \ldots, p-1$, holding A_{ij}, B_{ij} at the start of the algorithm and, in addition, C_{ij} at the end. C_{ij} is calculated in node (i, j) by accumulating partial sums of the form $A_{ik} B_{kj}$, i.e., $C_{ij} = \sum_{k=0}^{p-1} A_{ik} B_{kj}$. Before starting we must align the blocks of A and B so that each node contains two appropriate blocks that can be multiplied. This is accomplished by 'skewing' blocks of A (B) left (up) until the diagonal blocks of A (B) are in the first column (row) of processors, as shown in Figure 3. Then the first multiplication can occur. Next, thanks to the fact that $p = q$, rolling all the A's (B's) left (up) one step will align them properly for the next summand to be computed and added to the accumulating sum. This continues until a total of $p - 1$ rolls each of A and B have been done, at which point C_{ij} is fully computed. One more roll each will return A and B to their skewed positions; an inverse skew returns them to their initial locations.

The communication primitives for the 2D systolic algorithm are *skew* and *roll*. Overall the algorithm performs 2 skews and p rolls on each of A and B, all on N^2/p^2 data elements. Local multiplications are of two blocks, each of size $N/p \times N/p$.

start

$i \backslash j$	0		1		2		3	
0	A_{00}	B_{00}	A_{01}	B_{01}	A_{02}	B_{02}	A_{03}	B_{03}
1	A_{10}	B_{10}	A_{11}	B_{11}	A_{12}	B_{12}	A_{13}	B_{13}
2	A_{20}	B_{20}	A_{21}	B_{21}	A_{22}	B_{22}	A_{23}	B_{23}
3	A_{30}	B_{30}	A_{31}	B_{31}	A_{32}	B_{32}	A_{33}	B_{33}

$\longleftarrow$ skew $A's$ $\quad\uparrow$ skew $B's$

$i \backslash j$	0		1		2		3	
0	A_{00}	B_{00}	A_{01}	B_{11}	A_{02}	B_{22}	A_{03}	B_{33}
1	A_{11}	B_{10}	A_{12}	B_{21}	A_{13}	B_{32}	A_{10}	B_{03}
2	A_{22}	B_{20}	A_{23}	B_{31}	A_{20}	B_{02}	A_{21}	B_{13}
3	A_{33}	B_{30}	A_{30}	B_{01}	A_{31}	B_{12}	A_{32}	B_{23}

FIGURE 3. THE 2D SYSTOLIC ALGORITHM

2.3. BMR.

BMR is similar to the 2D Systolic algorithm, but avoids skewing by replacing rolls in one direction by a 1D broadcast, a one-to-all spreading of data across a linear array of processors. Again we view the machine as a $p \times q$ mesh with wraparound. Assume again that $p = q$, $N = rp$, and partition A, B, and C as in the 2D systolic case. C_{ij} is computed in node (i, j) as follows. First, the diagonal blocks of A (A_{ii}, $i = 0, \ldots, p-1$) are broadcast across their respective rows, after which each processor can do one multiplication, and the blocks of B are rolled up. At stage k, $k = 0, \ldots, p-1$, block $A_{i,[(i+k) \bmod p]}$ is broadcast across row i, after which a multiplication and upward roll of B is done. We illustrate the first stage in Figure 4. This continues until p broadcasts and $p - 1$ rolls have been done, at which point C_{ij} is fully computed. One more roll will return B to its original position.

The communication primitives for BMR are *1D (linear) broadcast* and *roll*. The algorithm performs p rolls and p broadcasts, each on N^2/p^2 elements. Local multiplications are on square matrices of size $N/p \times N/p$.

		$j=0$		1		2		3	
i	0	A_{00}	B_{00}	A_{00}	B_{01}	A_{00}	B_{02}	A_{00}	B_{03}
	1	A_{11}	B_{10}	A_{11}	B_{11}	A_{11}	B_{12}	A_{11}	B_{13}
	2	A_{22}	B_{20}	A_{22}	B_{21}	A_{22}	B_{22}	A_{22}	B_{23}
	3	A_{33}	B_{30}	A_{33}	B_{31}	A_{33}	B_{32}	A_{33}	B_{33}

after first broadcast; ready to multiply

FIGURE 4. THE BMR ALGORITHM

2.4. Non-square meshes.

Now suppose that the mesh is not square, i.e., $p \neq q$. This has no effect on 1D systolic, while 2D systolic and BMR easily generalize by treating the mesh as a virtual $\mathrm{lcm}(p,q) \times \mathrm{lcm}(p,q)$ mesh. In the worst case, p and q are relatively prime. Denote $\mathrm{lcm}(p,q)$ by α. Without loss of generality, assume $p > q$. We use a $p = 3$, $q = 2$ example for illustration. Thus $\alpha = 6$, and we proceed as if we were on a 6×6 virtual mesh. For both 2D systolic and BMR the initial data layout is shown in Figure 5. Notice that the blocks for all matrices are of size $N/\alpha \times N/\alpha$ and each processor holds $\alpha/p \times \alpha/q$ blocks of each matrix.

		$j=0$	1
i	0	A_{00} A_{01} A_{02} B_{00} B_{01} B_{02} A_{10} A_{11} A_{12} B_{10} B_{11} B_{12}	A_{03} A_{04} A_{05} B_{03} B_{04} B_{05} A_{13} A_{14} A_{15} B_{13} B_{14} B_{15}
	1	A_{20} A_{21} A_{22} B_{20} B_{21} B_{22} A_{30} A_{31} A_{32} B_{30} B_{31} B_{32}	A_{23} A_{24} A_{25} B_{23} B_{24} B_{25} A_{33} A_{34} A_{35} B_{33} B_{34} B_{35}
	2	A_{40} A_{41} A_{42} B_{40} B_{41} B_{42} A_{50} A_{51} A_{52} B_{50} B_{51} B_{52}	A_{43} A_{44} A_{45} B_{43} B_{44} B_{45} A_{53} A_{54} A_{55} B_{53} B_{54} B_{55}

FIGURE 5. LAYOUT ON NON-SQUARE MESHES

While direct imitation of the 2D systolic algorithm on the virtual mesh leads to a large amount of skewing on multiple columns (rows) of blocks in single processors, as well as fine-grained local multiplications on matrices of size $N/\alpha \times N/\alpha$, as shown in [31], it is possible to reduce the skewing costs and increase granularity. We first skew all the A data in processor row i left by $(\alpha/p) \cdot i$ virtual steps and all the B data in processor column j up $(\alpha/q) \cdot j$ virtual steps. After skewing, the data appears as shown in Figure 6. We box the data that will then be involved in the first multiplication. Now we can perform the first multiplication and then roll α/p columns of A and α/p rows of B at a time between subsequent multiplications.

Observe that in translating the virtual skewing steps into movement across actual physical processors, some skews involve less than the full data of a given matrix (e.g., matrix A above in processor row 1), and many involve all the data but must be performed in two stages (e.g., matrix B above in processor column 1). We will estimate costs for this detailed skewing process in Sections 3 and 4; for the moment we note that, counting unskewing, we must perform two complete skewing operations on each matrix. We also have p rolls of $\alpha/p \times \alpha/p$ blocks $= N^2/p^2$ data elements of A and p rolls of

i \ j	0	1
0	A_{00} A_{01} A_{02} B_{00} B_{01} B_{02} A_{10} A_{11} A_{12} B_{10} B_{11} B_{12}	A_{03} A_{04} A_{05} B_{33} B_{34} B_{35} A_{13} A_{14} A_{15} B_{43} B_{44} B_{45}
1	A_{22} A_{23} A_{24} B_{20} B_{21} B_{22} A_{32} A_{33} A_{34} B_{30} B_{31} B_{32}	A_{25} A_{20} A_{21} B_{53} B_{54} B_{55} A_{35} A_{30} A_{31} B_{03} B_{04} B_{05}
2	A_{44} A_{45} A_{40} B_{40} B_{41} B_{42} A_{54} A_{55} A_{50} B_{50} B_{51} B_{52}	A_{41} A_{42} A_{43} B_{13} B_{14} B_{15} A_{51} A_{52} A_{53} B_{23} B_{24} B_{25}

FIGURE 6. 2D SYSTOLIC ON NON-SQUARE MESHES

i \ j	0	1
0	A_{00} A_{01} A_{02} B_{00} B_{01} B_{02} A_{10} A_{11} A_{12} B_{10} B_{11} B_{12}	A_{03} A_{04} A_{05} B_{03} B_{04} B_{05} A_{13} A_{14} A_{15} B_{13} B_{14} B_{15}
1	A_{20} A_{21} A_{22} B_{20} B_{21} B_{22} A_{30} A_{31} A_{32} B_{30} B_{31} B_{32}	A_{23} A_{24} A_{25} B_{23} B_{24} B_{25} A_{33} A_{34} A_{35} B_{33} B_{34} B_{35}
2	A_{40} A_{41} A_{42} B_{40} B_{41} B_{42} A_{50} A_{51} A_{52} B_{50} B_{51} B_{52}	A_{43} A_{44} A_{45} B_{43} B_{44} B_{45} A_{53} A_{54} A_{55} B_{53} B_{54} B_{55}

FIGURE 7. BMR ON NON-SQUARE MESHES

$\alpha/p \times \alpha/q$ blocks $= N^2/pq$ elements of B. There are p multiplications on matrices of size $N/p \times N/p$ and $N/p \times N/q$.

For BMR, we are also able to achieve single-node multiplications with large granularity. In Figure 7 we box the blocks of A to be sent on the first broadcast and the blocks of B that will be involved in the first multiplication and subsequently rolled. While two broadcasts on half the data are actually required in processor row 1, we will see in Section 3 that communication costs for large messages are primarily a function of data size. Thus, for each processor row we can count one broadcast on the full data, giving a total of p broadcasts of $\alpha/p \times \alpha/p$ blocks $= N^2/p^2$ data elements of A and p rolls of $\alpha/p \times \alpha/q$ blocks $= N^2/pq$ elements of B. There are p multiplications on matrices of size $N/p \times N/p$ and $N/p \times N/q$.

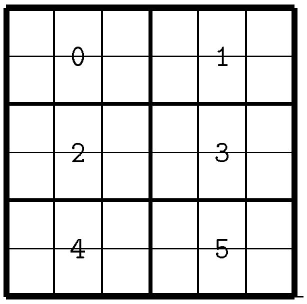
We note that there are other variants of BMR for non-square meshes that could be advantageous on other machines but are not effective on the Delta. We do not consider these here, but expect to report on them in future papers.

2.5. Data Distribution.

As mentioned earlier, the above descriptions can be interpreted assuming a contiguous distribution of data across processors, and in fact a contiguous layout is sufficient for matrix multiplication alone. However, many other algorithms that might call such a matrix multiplication routine utilize a scattered data layout. One scattered distribution scheme is the block scattered layout ([3], [9], [10]), where successive matrix blocks of size $r \times s$ are assigned to successive *physical* processors in both mesh/matrix dimensions. The r and s are application-dependent panel widths determined to be advantageous for local computation, while scattering the blocks across physical processors helps maintain good load balance in applications such as eigensolvers [23] and LU factorization [20]. Unfortunately, as noted in [10], for the important basic operation of matrix multiplication, the contiguous layout is optimal; the algorithm is already load balanced, and spreading the data to the level achieved by the block scattered scheme often incurs extra communication overhead. But in order to avoid the potentially expensive operation of transforming the layout when the matrix multiplication routine is called, it would still be advantageous to include scattered layout in the design of the matrix multiplication kernel and yet preserve the costs achievable using contiguous layout.

To this end, we opt for an alternate scattered data layout scheme – *virtual 2D torus wrap*. This is more general than block scattered layout and differs from it in two important ways. Assignment of data is to *virtual* rather than physical processors. This decouples the processor view from the mesh configuration and allows the programmer to treat a physical processor array of arbitrary aspect ratio as a virtual square mesh of processors. This in turn makes it possible to maintain symmetric placement of matrix indices onto virtual processors, thus allowing for maximal data alignment and granularity for local computations.

More specifically, as we saw earlier in describing our algorithms, we view a non-square $p \times q$ processor mesh as a square mesh of $\mathrm{lcm}(p,q) \times \mathrm{lcm}(p,q) = \alpha \times \alpha$ *virtual* processors. Figure 8(a) shows the six physical processors in our $p = 3, q = 2, \alpha = 6$ example, and in Figure 8(b) we see the resulting two-dimensional processor numbering when we view it as a 6×6 virtual mesh.



(a) 3×2 physical mesh

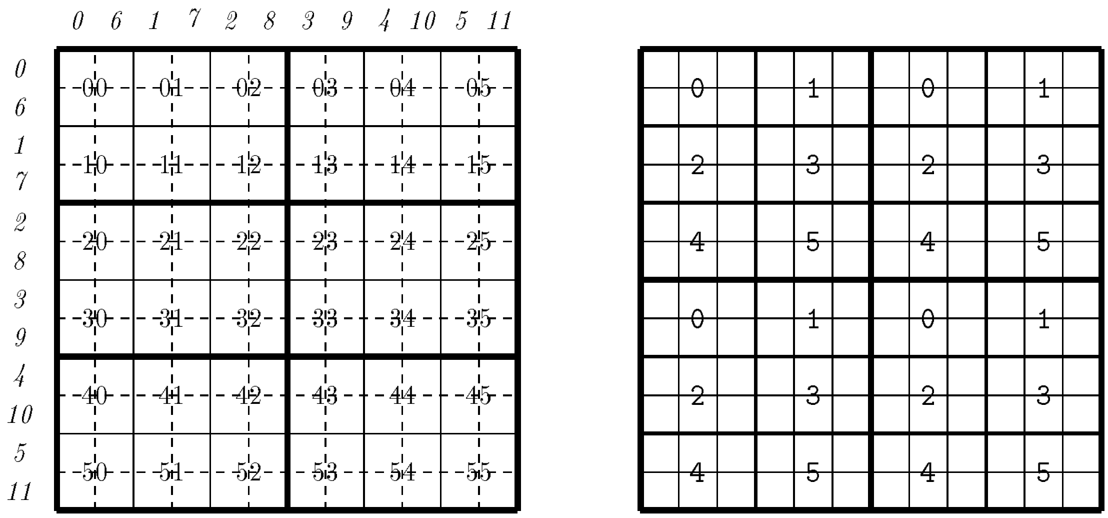
00	01	02	03	04	05
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

(b) Virtualized as 6×6 mesh

FIGURE 8. VIRTUALIZED 3×2 MESH

To distribute an $N \times N$ matrix M, panel M in row and column dimensions with panels of width r and s, respectively; this results in a decomposition of M into $r \times s$ blocks. In virtual 2D torus wrap, we assign successive blocks to *virtual* processors in both mesh dimensions. If assignment is to successive virtual processors and $r = s = N/\alpha$, we obtain a contiguous data layout. This contiguous virtual layout is used, for example, in [31]. Smaller r, s, $1 \le r, s \le N/\alpha$, results in more spreading of data across physical processors; for example, suppose $r = s = N/12$, so that $N = 12r = 12s$, and M contains 12 panels in each dimension. Figure 9 shows assignment of blocks to successive virtual processors. In the leftmost picture we see the processor point-of-view. The numbers at the left (top) edge of the processor mesh refer to the row (column) panels of the matrix. Within a processor, memory is mapped so that data corresponding to all virtual processors within a node are stored in a contiguous buffer. We also show in Figure 9 the matrix point-of-view, where we superimpose the physical processor assignments over the blocks of M.

In addition, panels can be arbitrarily assigned to virtual processors, and a very useful case giving further spreading across physical processors can be obtained by specifying spacing parameters s_r and s_c in the row and column directions, respectively. This data layout has been previously discussed in [16] and [7]. For the case where both s_r and s_c divide α, panels are then assigned consecutively so



Processor point-of-view   Matrix point-of-view

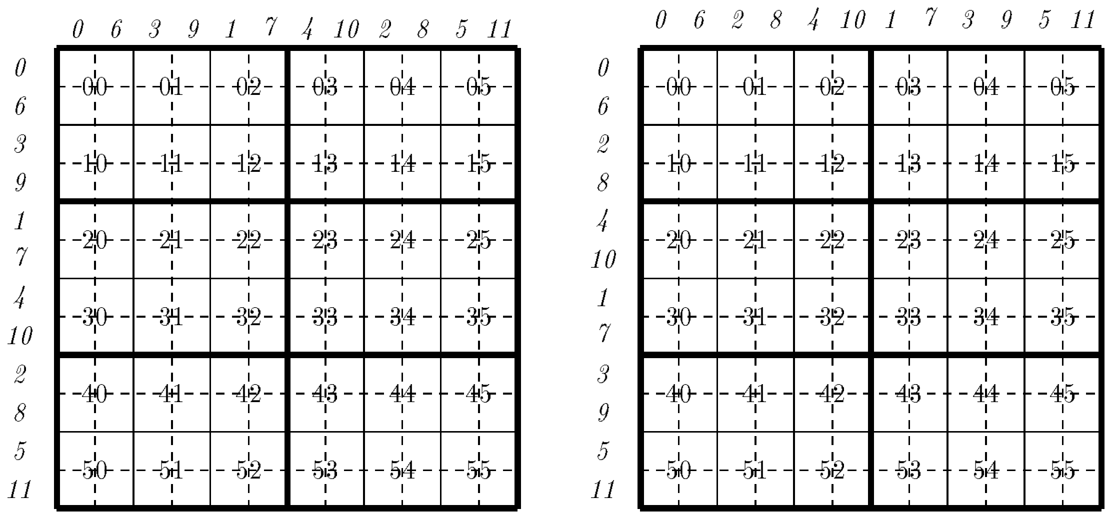
FIGURE 9. VIRTUAL TORUS WRAP

that panel i in the row dimension is assigned to row virtual processor

$$s_r \left(i \mod \frac{\alpha}{s_r} \right) + \left(\left\lfloor \frac{i}{\left(\frac{\alpha}{s_r}\right)} \right\rfloor \mod s_r \right),$$

where $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x. Panels are assigned in the column dimension analogously, substituting s_c for s_r in the formula above. Qualitatively, virtual panel spacing is the number of virtual processors to the next panel.

A useful spacing value is the number of virtual processors per node in a given dimension. This has the effect of placing consecutive panels in that dimension in successive *physical* processors. Figure 10 shows our example with $s_r = s_c = \alpha/p$ (here 2) and $s_r = s_c = \alpha/q$ (here 3). Note that choosing $s_r = \alpha/p$ and $s_c = \alpha/q$ would give us the block scattered decomposition, up to local storage differences.



Panel spacing = 2   Panel spacing = 3

FIGURE 10. PHYSICAL ROW OR COLUMN SPACING FROM PROCESSOR POINT-OF-VIEW

By applying different panel width and spacing parameters, a variety of distributions are thus easily obtained that achieve more or less spreading of data across actual physical processors. Virtualization provides a means to easily make many choices, including both the purely contiguous or block scattered layouts. Of particular note is that by using the same virtual panel spacing in both dimensions, i.e., choosing $s_r = s_c$, we preserve the symmetric placement of indices onto virtual processors. With this symmetric placement we can interpret the matrix partitions such as A_{ij} used in our algorithm descriptions as representing the data grouping within a virtual processor, whether or not it is actually contiguous matrix data, and still be assured of their *conformality*, i.e., that the data movement required in the basic algorithm alone will align data so that in each computational stage at least one operand's full data in each processor can be utilized. Thus we achieve maximal data alignment and granularity in local computations without incurring extra communication costs, while still providing a spread of data to assist in load balancing of calling applications.

Our present implementation allows for 2D virtual torus wrap layout with panel spacings having the property that $s_r = s_c$. We also assume s_c divides α. At present, the BiMMeR code uses square matrix blocks. In addition, for use in the ISDA application we make the often-used choices of small panel width and setting $s_c = \alpha/q$. This results in communication costs nearly equal to that of the optimal contiguous layout.

3. The Intel Touchstone Delta

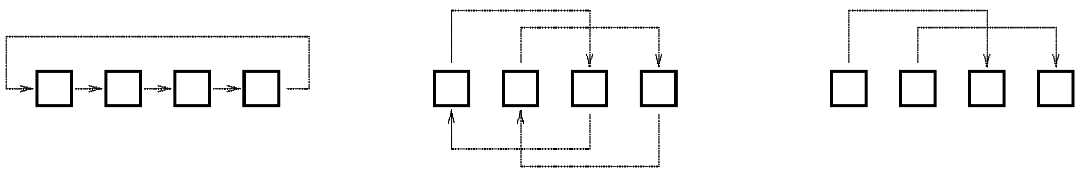
In this section we present an overview of the Intel Delta architecture, focusing primarily on its characteristics that influenced our choice of algorithm. The Delta is a descendant of the earlier Intel iPSC/1, iPSC/2, and iPSC/860 (Gamma) machines [28]. While these had a hypercube topology, the Delta is a two-dimensional mesh and represents the alternate design choice of having more wires between connected processors but less connectivity than the earlier machines ([11],[25]). The Delta is a prototype for Intel's follow-on machine, the Paragon [26].

Developed for the Concurrent Supercomputing Consortium and housed at the California Institute of Technology, the Delta consists of 528 i860 XR compute processors ([24], [30], [34]). Each has a peak double precision rate of 60 Mflops, 40 Mflops if the number of multiplications and additions is equal, as in standard matrix multiplication. One important feature of the i860 that influenced our design is that it has a write-back cache, requiring the processor to mediate all memory requests. Context-switching delays thus thwart attempts to overlap well-tuned communication with computation [29]. This explains the highly synchronous nature of all three candidate algorithms.

The compute processors are on node boards that also contain a FIFO interface to a backplane of separate routing chips (MRCs) that form the actual two-dimensional mesh [25]. Communication between MRCs occurs on two byte-wide communication channels, one in and one out, which operate at 65 MB/s aggregate bandwidth. In addition, switching is fast; MRC latency is 75 nanoseconds to move in the same direction, 150 nanoseconds to change direction. Packetized, e-cube (x-direction first) circuit-switched wormhole routing is used, together with additional protocols to avoid deadlock by, for the most part, guaranteeing that messages are 'consumed' by receiving nodes [33]. This takes the form of system buffering and system flow control (handshaking) and incurs extra message-passing overhead. As we will see later, if the user can guarantee that receiving nodes are ready to 'consume' messages, this overhead can be avoided.

One immediate consequence is that these routing techniques should limit communication costs due to internode distance, i.e., there should be little difference in delivery time between a message sent to a nearest neighbor or to a node several 'hops' away. This has been confirmed in timing studies [15]. Thus we may assume *hops are nearly free*, and our communications cost formulas will not include internode distance as a parameter. In addition, although the Delta is a true two-dimensional mesh, the above indicates that we may also assume it is *effectively a two-dimensional torus, i.e., has wraparound wires.* Thus it was natural to consider 2D and BMR, algorithms based on a 2D-toroidal topology.

Current maximum throughput between nodes and MRCs is approximately 11 MB/s of user data in each direction. Thus the MRC channels should be able to handle some number of multiple messages simultaneously, without degrading node-to-node throughput. This *contention*, i.e., multiple messages using the same mesh wires, should thus be a factor only when the channels are near saturation. Support for this conjecture is seen in performance studies. A more serious factor is *concurrency*, i.e., requiring a single node to handle two messages at once, for although a node is connected to its MRC by a channel in each direction, only one of these may use the FIFO at any given time. Thus, messages in and out of a single node, as well as multiple messages in either direction, cannot be processed simultaneously. See [15],[22],[29].



(a) Roll (b) Skew (c) Half-Skew

FIGURE 11. ROLL AND SKEW COMMUNICATION PATTERNS

These effects are crucial in computing the communications costs for our three algorithms. For simplicity, in the analysis that follows we present the case where communication occurs over an even number of processing nodes. Similar arguments giving slightly different formulas but leading to the same ultimate conclusions apply to the other cases. The *roll* primitive required by all the algorithms is shown in Figure 11(a). Its inherent concurrency explains the measured result that for moderate to large messages it is more efficient to guarantee that any given processor is only sending or only receiving at any given time ([22], [29]). Thus, for example, odd-numbered nodes can send first, and then receive, so that a roll costs two sequential communication steps. The *1D broadcast* primitive used by BMR does not exhibit problems with concurrency, but could have problems with contention. However, it can be designed to be free of both and costs $\lceil \log_2 p \rceil \approxeq \log_2 p$ communication steps over p processors, where $\lceil x \rceil$ is the smallest integer greater than or equal to x. The *skew* primitive clearly has both problems. In the 2D systolic algorithm various skew patterns, including those with high levels of wire contention as shown in Figure 11(b), are required. To cost this high-contention skew pattern we first eliminate concurrency by performing it in two stages, as with roll. Then each stage is a 'half-skew,' as shown in Figure 11(c). Over p processors its contention level is $p/2$; as such it can cost no more than a factor of $p/2$ over the time required for a contention-free communication on the slowest node, since we could instead perform $p/2$ sequential contention-free steps. It is observed [22] that costs for this pattern increase linearly with p, and cost factors approach this upper bound of $p/2$ for large data sizes. Thus, our cost estimate for this high contention skew over p processors on large matrices is $2(p/2) = p$ sequential communication steps. Further details on these primitives can be found in Section 4 and [22].

We have already seen that internode distance is not an important parameter, so the cost of sending w data elements can be modeled as $b + cw$, where b is a constant start-up cost and the remaining term is linear in the message size. Since our design is to favor large problems, startup should be small compared to the remaining costs for sending a large amount of data. This is borne out in performance studies [22]. Thus, to simplify formulas we will neglect startup and set $b = 0$. Our results would also hold for non-zero b. Also, since our objective is comparison, without loss of generality we may assume $c = 1$. Thus, the basic cost parameter for a single communication is the size of the message. The other important parameter for our primitives is their cost as a function of the number of processors p.

Table 1. Communication Costs for Large Matrices over p DELTA Processors

Basic Cost Parameter is Message size

PRIMITIVE		*COST*
Roll	...	*2*
1D Broadcast	...	$\log_2 p$
Skew with Contention $p/2$	...	p

Table 1 summarizes our simplified communication cost model.

4. Design and Implementation

In this section we present comparative cost models for the three candidate algorithms and choose the most promising one for initial implementation. As examples, we compute totals for both 2D and BMR on a non-square mesh. From Section 2, recall BMR entails p broadcasts of N^2/p^2 elements across q processors and p rolls of N^2/pq elements. Using Table 1, we obtain a total communication cost of $[(2p + q\log_2 q)/pq]\,N^2$. For 2D, requirements are p rolls of $\alpha/p \times \alpha/p$ blocks $= N^2/p^2$ data elements of A, and p rolls of $\alpha/p \times \alpha/q$ blocks $= N^2/pq$ elements of B, plus two skewing operations on each matrix. The roll costs are easily computed and total $[(2p+2q)/pq]\,N^2$. Estimates for skewing are obtained by computing the cost for the skew with contention $p/2$. This pattern will occur in the centrally located rows (columns). Recall from Section 2 that this often entails two stages of communication, each on a subset of the data. When this occurs, the contention level on each of the two stages actually differs by 1, but we may compute an estimate by counting the contention as $q/2$ ($p/2$) for the full data. Also, for large matrices we may neglect startups as indicated above and treat the two stages as one, on the full data. We have a total of 2 skews over q processors and 2 over p processors, both on N^2/pq elements. Using Table 1 we can now compute a cost estimate for skewing of $[(2p+2q)/pq]\,N^2$. Thus, communication costs for 2D range between a minimum of $[(2p+2q)/pq]\,N^2$ (neglecting skewing) to approximately $[(4p+4q)/pq]\,N^2$. Tables 2 and 3 summarize communication costs and granularity of the three algorithms.

Table 2. Algorithmic Communication Cost on $p \times q$ Mesh

	$p = q$	$p > q$
1D	$2N^2$	$2N^2$
2D	$(4/p)N^2$ *to* $\approx$ $(8/p)N^2$	$[(2p+2q)/pq]\,N^2$ *to* $\approx$ $[(4p+4q)/pq]\,N^2$
BMR	$[(2+\log_2 p)/p]\,N^2$	$[(2p + q\log_2 q)/pq]\,N^2$

Table 3. Algorithmic Granularity on $p \times q$ Mesh

	$p = q$	$p > q$
1D	$(N \times N/p^2)\cdot(N/p^2 \times N/p^2)$	$(N \times N/pq)\cdot(N/pq \times N/pq)$
2D	$(N/p \times N/p)\cdot(N/p \times N/p)$	$(N/p \times N/p)\cdot(N/p \times N/q)$
BMR	$(N/p \times N/p)\cdot(N/p \times N/p)$	$(N/p \times N/p)\cdot(N/p \times N/q)$

It is clear from Table 2 that 1D has significantly higher communication cost and can be eliminated. As we have seen, both remaining algorithms can be designed to achieve the same granularity on arbitrary meshes. They both rely on the *roll* primitive, 2D requires *skew*, and BMR uses *1D broadcast*. We might lean towards BMR since *1D broadcast* seems better suited to the Delta than *skew*. To compensate for this, we continue the analytic comparison using minimal costs for 2D, i.e., neglecting skewing. In this case, while 2D has lower communication costs than BMR, they are not that far apart, given that $\log_2 q$ can be no bigger than 4 on the Delta. In addition, the broadcast primitive is more likely to be implemented efficiently or in hardware on other architectures. This is less likely to be true for *skew*; as such, we could expect BMR to be less machine-dependent and to more easily achieve good performance when ported to other machines. In addition, *skew* must be performed in both mesh dimensions, whereas *1D broadcast* involves only one. Thus, for a mesh with extremely poor aspect ratio, we can choose to do the broadcast over the short dimension but cannot avoid the expense of the long skew. We are thus pushed towards reliance on the broadcast rather than the skew primitive and conclude that BMR is our algorithm of choice.

The BiMMeR software is thus based on the BMR algorithm and optimized implementations of its two communication primitives: *roll* and *1D broadcast*. See [22] for details on the design and optimization of these primitives on the Delta. We now summarize with a few descriptive comments. Our roll is designed for any of the four mesh directions and for arbitrary subsets of single rows (columns) of processors, as well as for all rows (columns) in an arbitrary rectangular submesh of processors. It can also handle linear arrays and rings embedded in the two-dimensional mesh. Our broadcast code also operates on arbitrary subsets of single rows (columns) of processors and on all rows (columns) in a rectangular submesh. It is a generalization of a similar algorithm given in [5], free of contention and concurrency. In our design, the number of processors in a row (column) can be any size and is not restricted to powers of two; we have also optimized the lengths of communication paths.

For both primitives we investigated various message-passing options to determine techniques for achieving high performance on the Delta. For message-passing one can choose between the usual blocking or non-blocking calls, as well as either normal or forced-type. Normal message types invoke the system buffering and system flow-control protocols to guarantee there is space for messages before they are sent. The special forced-type message circumvents these protocols. While this requires the user to guarantee that space is available and that the sending processor knows when this occurs (a zero-length message sent from receiver to sender will do), this is a small price to pay for the large gain in performance that can be seen when forced-type is used ([22],[29],[32]). As an illustration, we tested the two-step roll primitive using 10 different versions of message passing [22]. The three with best performance for moderate to large messages are all variations of forced-type communication. We thus use forced-type in our implementation.

Our matrix multiplication based on these communication kernels is general, presently applicable to arbitrary $p \times q$ rectangular meshes of Delta processors where $p \geq q$, and easily generalized to arbitrary meshes.

5. Performance

In this section we first report on performance measurements that compare an implementation of the 1D algorithm and BiMMeR (the worst and best algorithms based on our analysis) and confirm our analytic conclusions above. Then we report on timings which demonstrate that BiMMeR exhibits a high degree of scalability and robust behavior over varying mesh configurations. All testing was carried out on the Intel Delta System housed at the California Institute of Technology. Runs were done under Release 1.5 of the Intel NX/M Operating System, and timings represent measurements reported by the slowest node in the allocated mesh of processors.

As the 1D systolic algorithm also uses the roll primitive and generalizes easily to arbitrary meshes, it was also implemented and compared to BiMMeR. Figure 12 shows rates achieved on square matrices. We maintained 550×550 double precision matrix blocks per node on a square mesh over

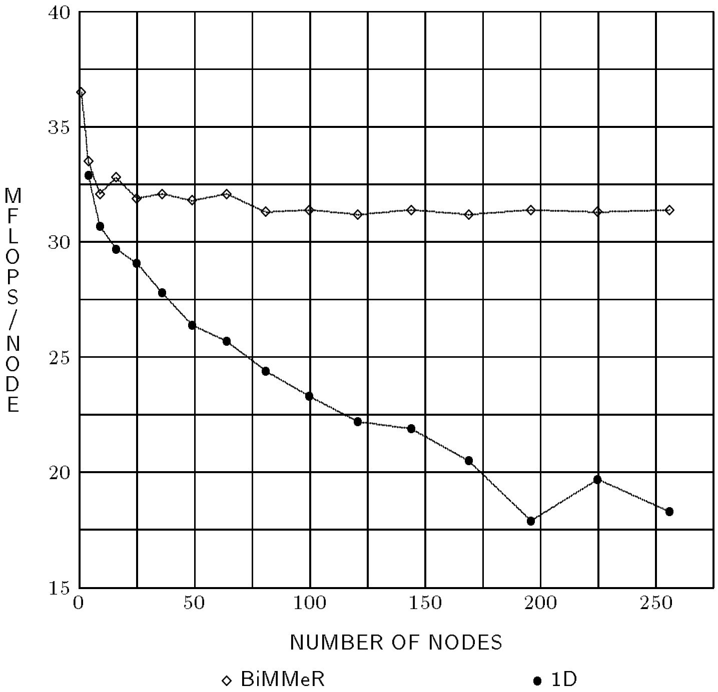


FIGURE 12. BIMMER VS. 1D SYSTOLIC

increasing numbers of processors. Observe that for a fixed number of nodes, behavior of the 1D algorithm is essentially unchanged by reconfiguration of the mesh and will thus achieve the same performance as that shown here for square meshes. In a subsequent paragraph we will see that BiMMeR maintains its high Mflop rates on these problems as the mesh aspect ratio is varied. Thus, we have confirmed the expected result that BiMMeR is superior to 1D for presently targeted matrix and mesh sizes. We note that we have also found 1D or other variants of BMR to be preferable for other (extreme) matrix shapes and will report on this in future work.

We examined performance of BiMMeR in several ways. First, we can observe its behavior over various local granularities in Table 4. To restrict our attention to just the change in granularity, for each square mesh size we ran BiMMeR on square matrices of increasing order. Table entries are the achieved Mflops per node for the slowest node. As expected, for a fixed granularity, performance decreases slightly as the mesh size increases due to increases in both computational requirements and communication cost. Also as expected, for a fixed mesh configuration, performance decreases with granularity, with a sharp drop for granularities below approximately 100^2 data elements per node (shown in the table to the right of the darker vertical line). This agrees with the results for DGEMM on single nodes shown earlier in Figure 1. However, results for such low granularities are shown only for completeness. One would not generally run small problems on large meshes, paying unneccesarily for parallelism inappropriate to the problem size. Instead, a smaller mesh or even a single processor could be used, hence increasing the local granularity on each processor. For example, the problem with local granularity of 50^2 being run on a 16×16 mesh and achieving only 10.9 Mflops per node represents a matrix of size 800×800. Using instead a 4×4 mesh would provide local granularity of 200^2 and hence speeds of 27.4 Mflops per node. Even higher efficiency would be achieved on a 2×2 mesh. We see that for problems large enough for the available parallelism, performance of approximately 20 to over 30 Mflops per node is maintained.

Next, we fixed the number of compute nodes and varied the aspect ratio. In order to measure just the effect of the changing topology, we maintained a constant number of data elements per node over

Table 4. Effects of Local Granularity on Per Node Performance

	Elements per node						
Mesh	550^2	400^2	200^2	100^2	50^2	25^2	10^2
16×16	31.4	29.3	24.7	18.3	10.9	4.5	0.69
12×12	31.3	29.6	24.5	18.5	10.3	4.9	0.70
8×8	32.1	30.5	26.0	19.7	12.1	5.2	0.77
4×4	32.9	31.4	27.4	22.3	13.6	6.1	0.88

the different mesh configurations. Table 5 shows our results on 144, 16, and 12 nodes for problems with $550^2 = 302500$, $500^2 = 250000$, $400^2 = 160000$, $200^2 = 40000$, and $100^2 = 10000$ data elements per node. To maintain k^2 data elements per node on a $p \times q$ mesh, we choose input matrices A and B to be square, of order $\sqrt{pqk^2}$. Table entries are the achieved Mflops per node for the slowest node. Observe that while there are significant differences in communications patterns as the aspect ratio degrades, we achieve good performance in almost all cases. Between configurations that are not strictly linear, performance differences are no more than 6.5%. Even the linear configuration provides good performance for granularities of 400^2 and above, where the effect is at most 9.5%.

Table 5. Effect of Mesh Aspect Ratio for Fixed Granularities

CONFIGURATION	*Elements per node*				
	550^2	500^2	400^2	200^2	100^2
12×12	31.4	31.2	29.7	24.9	18.5
16×9	31.6	31.2	29.8	24.4	17.9
16×1	29.8	30.3	28.7	22.0	14.8
8×2	32.4	32.6	31.5	27.1	20.0
4×4	32.8	32.7	31.4	27.4	21.4
12×1	30.6	30.0	29.4	23.2	16.6
6×2	33.1	32.5	32.0	27.6	21.0
4×3	32.9	32.2	31.5	27.3	21.0

Scaling behavior for fixed size matrices over increasing square mesh sizes was also considered. Figure 13 shows our results for square matrices of order 2200, 4400, and 6600 on square meshes from the smallest size on which the problem would fit up to size 16×16. Thus local granularities range from a high of 550^2 on the smallest mesh down to approximately 140^2 for the smallest problem on the largest mesh. Speedup is computed as the ratio of extrapolated single-node sequential time to parallel time. The extrapolated sequential time is determined by assuming that the given problem size can be run on a single node maintaining the measured peak rate for DGEMM. As expected, achieved rates per node decline as the fixed problem is spread over more processors for two reasons:

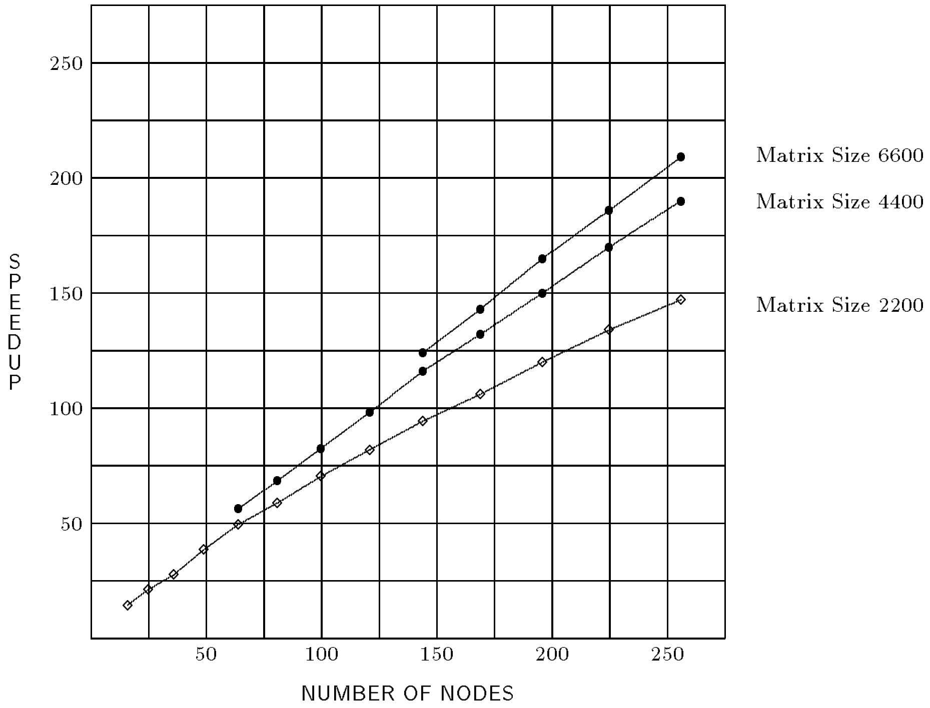


FIGURE 13. PARALLEL SPEEDUP ON SQUARE MESHES

the communication requirements increase and the computational requirements (granularities) per node decrease. Furthermore, efficiency increases with problem size, since higher granularity can be maintained. This behavior agrees with our philospy of making design decisions in favor of large problems.

We next considered blocksize effects, i.e., the effect of changing the size of the basic $r \times s$ block of contiguous matrix row and column indices that will not be further divided by the data layout scheme. Thus, the data in a matrix block will always be placed in the same physical processor. If the blocksize is such that in both dimensions blocks can be evenly divided among the processors, then work will be evenly distributed and computation will be load balanced. On the other hand, excess blocks will increase the work of their assigned processor. Smaller blocksizes are therefore preferable for load balancing, but a larger size may be useful to a calling application for other reasons. We tested BiMMeR with panel widths $r = s = 1, 4, 16$ over many mesh configurations. Matrix sizes were chosen to represent problems that would remain load balanced over the three panel widths (guaranteed in this case when $\mathrm{lcm}(p, q) \cdot r$ divides all matrix dimensions) and those that would not. In addition, the unbalanced problem sizes were chosen so as to assign full extra blocks to very few processors, thus providing a worst case for load balancing. Tables 6 and 7 show our results for balanced problems on fixed size square matrices. Tables 8 and 9 give results for fixed size matrices that are unbalanced. Tables 10 through 13 are similar, but consider problems whose sizes are chosen to maintain similar performance across all mesh sizes, hence allowing us to focus on blocksize effects alone. This is achieved by choosing problems with approximately fixed local granularities. One way to do this is by the technique used for Table 5 – for granularity k^2 on a $p \times q$ mesh choose matrices A and B to be square of order $\sqrt{pqk^2}$. Notice that this changes the shape of the local data for both the A and B matrices as p and q vary, with both being of local size $\sqrt{(q/p)\,k^2} \times \sqrt{(p/q)\,k^2}$. Although this approach would serve our purposes in this case, we use an alternate technique that also maintains the shape for the local data of the A matrix. For a local granularity of k^2 on a $p \times q$ mesh, choose A and B to be approximately $pk \times qk$ and $qk \times pk$, respectively. The local data sizes

Tables 6 and 7. Panel Width Effects: Balanced Fixed Matrices

Mesh	*Panel Width*		
	1	**4**	**16**
16 × 16	18.0	18.0	18.0
16 × 8	23.2	23.2	23.2
16 × 6	24.4	24.3	24.4
16 × 4	26.3	26.2	26.3
16 × 3	26.8	26.9	26.9
12 × 12	20.6	20.6	20.6
12 × 8	24.0	24.0	24.0
12 × 4	27.3	27.1	27.2
8 × 8	25.7	25.6	25.7
8 × 6	27.1	27.0	27.1
8 × 4	29.4	29.4	29.4
4 × 4	31.1	31.1	31.1
4 × 3	31.9	31.9	31.9

Table 6. Matrix Order **1536**

Mesh	*Panel Width*		
	1	**4**	**16**
16 × 16	24.5	24.5	24.5
16 × 8	28.6	28.7	28.7
16 × 6	29.5	29.5	29.5
16 × 4	30.9	30.9	30.9
16 × 3	31.3	31.3	31.3
12 × 12	26.6	26.7	26.6
12 × 8	29.4	29.4	29.4
12 × 4	31.7	31.7	31.7
8 × 8	30.2	30.2	30.2
8 × 6	31.2	31.1	31.2
8 × 4	32.7	32.7	32.7
Too large for other meshes shown at left			

Table 7. Matrix Order **3072**

Tables 8 and 9. Panel Width Effects: Unbalanced Fixed Matrices

Mesh	*Panel Width*		
	1	**4**	**16**
16 × 16	17.8	17.1	14.1
16 × 8	22.6	22.4	19.0
16 × 6	23.1	23.1	20.2
16 × 4	25.4	25.3	22.4
16 × 3	26.0	25.1	23.0
12 × 12	20.5	20.2	17.0
12 × 8	24.0	23.6	20.3
12 × 4	27.1	26.8	24.1
8 × 8	25.6	25.5	22.8
8 × 6	27.0	26.8	24.4
8 × 4	29.4	29.1	27.0
4 × 4	31.2	30.8	29.7
4 × 3	31.9	31.7	30.5

Table 8. Matrix Order **1559**

Mesh	*Panel Width*		
	1	**4**	**16**
16 × 16	24.2	23.9	21.4
16 × 8	28.3	28.2	25.8
16 × 6	29.1	29.1	26.8
16 × 4	30.5	30.3	28.4
16 × 3	30.8	30.2	28.8
12 × 12	26.7	26.3	24.0
12 × 8	29.3	29.0	27.0
12 × 4	31.5	31.2	29.6
8 × 8	30.1	30.0	28.3
8 × 6	31.1	31.0	29.5
8 × 4	32.6	32.5	*
* Just exceeds memory on nodes with extra panels			
Too large for other meshes shown at left			

Table 9. Matrix Order **3095**

are thus $k \times k$ for A and $(q/p)\,k \;\times\; (p/q)\,k$ for B. For example, on a 16×16 mesh, local granularity of approximately 200^2 for a balanced problem is achieved with size 3072×3072 for both input matrices A and B. On a 16×12 mesh, the sizes of A and B would be adjusted to 3072×2304 and 2304×3072, respectively. Table entries are the achieved Mflops per node for the slowest node.

Tables 10 and 11. Panel Width Effects: Balanced Fixed Granularity

Mesh	*Panel Width*		
	1	**4**	**16**
16×16	24.4	24.4	24.3
16×12	26.0	26.1	26.1
16×11	25.6	25.7	25.7
12×12	24.5	24.5	24.4
8×8	25.7	25.7	25.6
8×6	27.0	27.0	26.9
4×4	27.1	27.1	27.1

Table 10. Granularity $\approx \mathbf{200^2}$

Mesh	*Panel Width*		
	1	**4**	**16**
16×16	29.5	29.6	29.7
16×12	30.5	30.5	30.5
16×11	30.5	30.5	30.6
12×12	29.3	29.3	29.3
8×8	30.5	30.5	30.5
8×6	31.2	31.2	31.2
4×4	31.1	31.2	31.2

Table 11. Granularity $\approx \mathbf{400^2}$

Tables 12 and 13. Panel Width Effects: Unbalanced Fixed Granularity

Mesh	*Panel Width*		
	1	**4**	**16**
16×16	27.0	26.7	23.9
16×12	25.7	25.3	22.6
16×11	25.3	25.0	22.5
12×12	24.3	24.1	21.5
8×8	25.7	25.2	22.8
8×6	27.0	26.7	23.9
4×4	27.3	26.7	24.8

Table 12. Granularity $\approx \mathbf{200^2}$

Mesh	*Panel Width*		
	1	**4**	**16**
16×16	29.4	29.2	27.7
16×12	30.3	30.1	28.3
16×11	30.3	30.2	28.6
12×12	29.2	29.1	27.4
8×8	30.4	30.3	28.7
8×6	31.1	31.0	29.3
4×4	31.2	30.8	29.7

Table 13. Granularity $\approx \mathbf{400^2}$

Two general and expected trends are seen. For unbalanced problems there is a decrease in performance with increasing panel width. However, this effect lessens with increasing granularity. For example, observe that for local granularities of approximately 200^2 and 400^2 we see differences of approximately 11% and 6%, respectively. For balanced problems changes in panel width have negligible effect. In all cases, panel width 4 provides performance very close to that of panel width

1. This matches well with applications that have found panel width 5 to be optimal on the Delta ([9],[35]).

Finally, to examine scaling behavior for larger problems, we maintain 550×550 matrix blocks (double precision) per node. Figure 14 shows our total performance over square meshes of processors. We achieve rates in all cases between 31.2 and 33.5 Mflops per node, a difference of less than 7%.

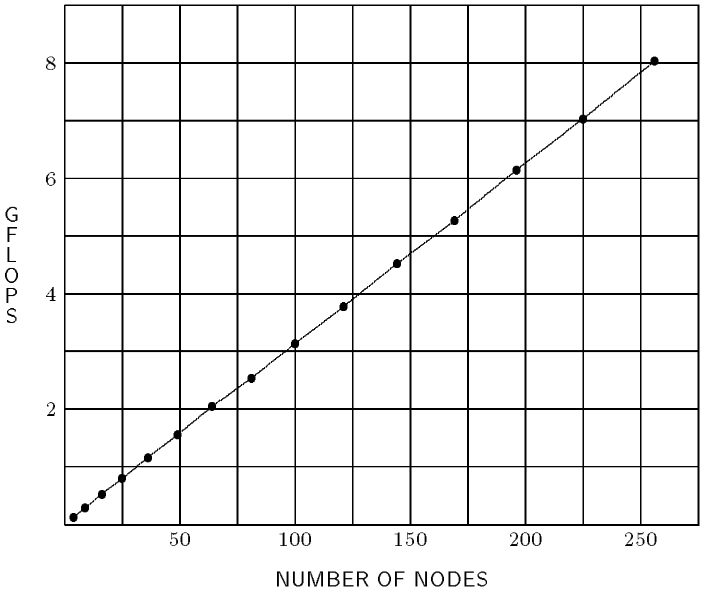


FIGURE 14. TOTAL GFLOPS ON SQUARE MESH (550×550 MATRIX/NODE)

6. CONCLUSIONS

In our effort to develop an efficient distributed matrix-matrix multiplication on the Intel Touchstone Delta, we have achieved our design objectives. Our matrix multiplication software, BiMMeR, is based on the general and highly optimized kernels of *roll* and *1D broadcast*, both of which could be useful communications primitives for general code development. BiMMeR's design is general, able to deal with arbitrary mesh aspect ratios and matrix dimensions. This is in contrast to some, whose performance depends on a restricted set of parameters, such as certain matrix sizes, or mesh dimensions that are powers of two. Changing aspect ratios affect performance on targeted matrices less than 10%, and scaling to larger meshes decreases performance less than 7% for the cases shown. On a 16×16 square mesh BiMMeR achieves 31.4 Mflops per node, for a parallel efficiency of 86%, and overall peak performance in excess of 8 Gflops for an 8800×8800 matrix.

BiMMeR is now being used in our ongoing work on ISDA and is available for general use as Beta software through `netlib` and `paralib`. (Contact PRISM group at `prism@super.org` for access information.) We also expect to expand its capabilities to include choosing alternate algorithms for other cases, such as unusual matrix shapes. Finally, our choice of algorithm and code design should transfer well to other architectures that readily support a 2D toroidal topology. We have already begun to apply our design philosophy to investigate this on the Intel Paragon and the IBM SP1.

7. ACKNOWLEDGEMENTS

We sincerely thank Rik Littlefield, of Pacific Northwest Laboratory, and Greg Regnier, David Scott, David Payne, Denise Ecklund, Ellen Deleganes, and Jerry Baugh, of Intel Supercomputing Systems Division, for valuable discussions about the Intel Delta. We also thank the staff at the Caltech Concurrent Supercomputing Facilities for their unfailing and friendly support.

References

1. Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, & D. Sorensen, *LAPACK: A portable linear algebra library for high-performance computers*, Proceedings, Supercomputing '90, IEEE Computer Society Press, Los Alamitos, California, 1990, pp. 2–11.
2. Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, & D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.
3. Ashcraft, C. C., *The distributed solution of linear systems using the torus wrap data mapping*, Engineering Computing and Analysis Technical Report ECA-TR-147, Boeing Computer Services (1990).
4. Auslander, L. & A. Tsao, *On parallelizable eigensolvers*, Advances in Applied Mathematics **13** (1992), 253–261, (also appeared as Technical Report SRC-TR-91-028, Supercomputing Research Center).
5. Barnett, M., D. G. Payne, & R. van de Geijn, *Optimal broadcasting in mesh-connected architectures*, preprint, also appears as University of Texas Computer Science Technical Report TR-91-38 (December, 1991).
6. Bischof, C. H., *LAPACK: Linear Algebra Software for Supercomputers*, Preprint MCS-P236-0491, Argonne National Laboratory (July, 1991).
7. 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*, Argonne National Lab ANL/MCS-TM-184, (available from the archives of the HPF Forum).
8. Cannon, L.E., *A cellular computer to implement the Kalman filter algorithm*, Ph.D. Thesis (1969), Montana State University.
9. 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.
10. 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.
11. Dally, W. J., *Performance analysis of k-ary n-cube interconnection networks*, IEEE Transactions on Computers **39** (1990), no. 6, 775–785.
12. 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.
13. Dongarra, J. J., I. S. Duff, D. C. Sorensen, & H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, 1991.
14. Dongarra, J., C. B. Moler, J. R. Bunch, & G. W. Stewart, *LINPACK User's Guide*, SIAM, Philadelphia, 1979.
15. Dunigan, T. H., *Communication performance of the Intel Touchstone Delta mesh*, Technical Report ORNL/TM-11983, Oak Ridge National Laboratory (January, 1992).
16. 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.
17. 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.
18. 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.

19. Fox, G., S. Otto, & A. Hey, *Matrix algorithms on a hypercube I: Matrix multiplication*, Parallel Computing **4** (1987), 17–31.
20. Golub, G. & C. F. Van Loan, *Matrix Computations, 2nd ed.*, Johns Hopkins University Press, Baltimore, 1989.
21. Huss-Lederman, S., E. M. Jacobson, & A. Tsao, *Comparison of scalable parallel matrix multiplication libraries*, Proceedings, Scalable Parallel Libraries Conference (Starksville, MS, Oct. 6–8, 1993), IEEE, 1993, pp. 142–149, (also PRISM Working Note #13, also appears as Technical Report SRC-TR-93-108, Supercomputing Research Center, 1993).
22. Huss-Lederman, S., E. M. Jacobson, A. Tsao, & G. Zhang, *Optimizing communication primitives on the Intel Touchstone Delta*, Technical Report, Supercomputing Research Center (to appear).
23. Huss-Lederman, S., A. Tsao, & G. Zhang, *A parallel implementation of the Invariant Subspace Decomposition Algorithm for dense symmetric matrices*, Proceedings, Sixth SIAM Conference on Parallel Processing for Scientific Computing (Norfolk, Virginia, March 22-24, 1993) (R. F. Sincovec, ed.), SIAM, Philadelphia, 1993, pp. 367–374, (also PRISM Working Note #9, also appears as Technical Report SRC-TR-93-091, Supercomputing Research Center, 1993).
24. Intel Corporation, *i860 64-Bit Microprocessor Programmer's Reference Manual*, 1990.
25. Intel Supercomputing Systems Division, *Intel Touchstone Delta System Description*, Advanced Information, Intel Corporation (February, 1991).
26. Intel Supercomputing Systems Division, *Paragon Supercomputers*, Order Number 203/6/92/ 10K/GA, Intel Corporation (1992).
27. Kuck & Associates, Inc., *CLASSPACK, Basic Math Library User's Guide, Release 1.2*, Document # 9202003, 1992.
28. Lillevik, S. L., *The Touchstone 30 Gigaflop DELTA Prototype*, Proceedings, Sixth Distributed Memory Computing Conference (Portland, Oregon, April 28 – May 1, 1991) (Stout, Q. & M. Wolfe, eds.), IEEE Computer Society Press, Los Alamitos, California, 1991, pp. 671–677.
29. Littlefield, R., *Characterizing and tuning communications performance for real applications*, presentation overheads, Proceedings, First Intel Delta Applications Workshop, CCSF-14-92 (February, 1992), Caltech Concurrent Supercomputing Facilities, Pasadena, California, 1992, pp. 179–190.
30. Margulis, Neal, *i860 Microprocessor Architecture*, McGraw-Hill, New York, 1990.
31. Mathur, K. K. & S. L. Johnsson, *Multiplication of matrices of arbitrary shape on a data parallel computer* (1992), Thinking Machines Corporation, preprint, also released as Technical Report TR-216.
32. Regnier, G., *Intel Touchstone Delta message passing performance*, Intel Supercomputing Systems Division (August, 1991), preprint.
33. Regnier, G., *Delta message passing protocol*, presentation overheads, Proceedings, First Intel Delta Applications Workshop, CCSF-14-92 (February, 1992), Caltech Concurrent Supercomputing Facilities, Pasadena, California, 1992, pp. 173–178.
34. Scott, D. S., & G. R. Withers, *Performance and assembly language programming of the iPSC/860 System*, Proceedings, Sixth Distributed Memory Computing Conference (Portland, Oregon, April 28 – May 1, 1991) (Stout, Q. & M. Wolfe, eds.), IEEE Computer Society Press, Los Alamitos, California, 1991, pp. 534–541.
35. Van de Geijn, R. A., *Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems: Progress Report*, Computer Science Technical Report TR-91-28, University of Texas (1991).