SUMMA: Scalable Universal Matrix Multiplication Algorithm*

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
rvdg@cs.utexas.edu

Jerrell Watts
Scalable Concurrent Programming Laboratory
California Institute of Technology
Pasadena, California 91125
jwatts@scp.caltech.edu

Abstract

In this paper, we give a straight forward, highly efficient, scalable implementation of common matrix multiplication operations. The algorithms are much simpler than previously published methods, yield better performance, and require less work space. MPI implementations are given, as are performance results on the Intel Paragon system.

1 Introduction

It seems somewhat strange to be writing a paper on parallel matrix multiplication almost two decades after commercial parallel systems first became available. One would think that by now we would be able to manage such an apparently straight forward task with simple, highly efficient implementations. Nonetheless, we appear to have gained a new insight into this problem.

Different approaches proposed for matrix-matrix multiplication include 1D-systolic [14], 2D-systolic [14], Cannon's algorithm [5, 14], Broadcast-Multiply-Roll [12, 13], and the Transpose algorithm [19]. Two recent efforts extend the work by Fox et. al. to general meshes of nodes: the paper by Choi et. al. [8] uses a two-dimensional block-wrapped (block-cyclic) data decomposition, while the papers by Huss-Lederman et. al. [17, 18] use a "virtual" 2-D torus wrap data layout. Both these efforts report very good performance attained on the Intel Touchstone Delta, achieving a sizeable percentage of peak performance.

The method presented our paper has the benefit of being more general, simpler and more efficient. We explain our algorithms for the case where the matrices to be multiplied, as well as the result, are block-mapped identically to nodes. However, we show how this restriction can be easily relaxed to achieve the wrapped decompositions mentioned above, as well as more general decompositions.

This paper makes a number of contributions: We present a new approach and its scalability analysis. In addition, we give *complete* Message Passing Interface (MPI) [15] implementations, demonstrating the power of this standard for coding concurrent algorithms. We show how our simpler approach outperforms more complex implementations, and, finally, we show how it is more general than alternative approaches to the problem.

*This work is partially supported by the NASA High Performance Computing and Communications Program's Earth and Space Sciences Project under NRA Grant NAG5-2497. Additional support came from the Intel Research Council. Jerrell Watts is being supported by an NSF Graduate Research Fellowship.

2 Notation

We will consider the formation of the matrix products

$$
\begin{aligned}
C &= \alpha AB + \beta C & (1)\\
C &= \alpha AB^T + \beta C & (2)\\
C &= \alpha A^T B + \beta C & (3)\\
C &= \alpha A^T B^T + \beta C & (4)
\end{aligned}
$$

These are the special cases implemented as part of the widely used sequential Basic Linear Algebra Subprograms [10].

We will assume that each matrix X is of dimension $m^X \times n^X$, $X \in \{A, B, C\}$. Naturally, there are constraints on these dimensions for the multiplications to be well defined: We will assume that the dimensions of C are $m \times n$, while the "other" dimension is k.

3 Model of Computation

We assume that the nodes of the parallel computer form a $r \times c$ mesh. While for the analysis, this is a physical mesh, the developed codes require only that the nodes can be logically configured as a $r \times c$ mesh. The $p = rc$ nodes are indexed by their row and column index and the (i, j) node will be denoted by $\mathbf{P}_{ij}$.

In the absense of network conflicts, communicating a message between two nodes requires time $\alpha + n\beta$, which is reasonable on machines like the Intel Paragon system [3] . Parameters α and β represent the startup and cost per item transfer time, respectively. Performing a floating point computation requires time γ.

4 Data Decomposition

We will consider two dimensional data decompositions. The analysis will automatically include the one dimensional cases by letting either row or column dimension of the mesh equal one. For all the algorithms, we will assume the following assignment of data to nodes: Given $m^X \times n^X$ matrix X, $X \in \{A, B, C\}$, and an $r \times c$ logical mesh of nodes, we partition as follows:

$$
X = \left(\begin{array}{c|c|c}
X_{00} & \cdots & X_{0(c-1)} \\ \hline
\vdots & & \vdots \\ \hline
X_{(r-1)0} & \cdots & X_{(r-1)(c-1)}
\end{array} \right)
$$

and assign X_{ij} to node $\mathbf{P}_{ij}$. Submatrix X_{ij} has dimensions $m_i^X \times n_j^X$, with $\sum m_i^X = m$ and $\sum n_i^X = n$. Further restrictions on the dimensions are given for each of the four variants, to ensure that the appropriate row and column dimensions match.

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

For this operation to be well-defined, we require $m^A = m$, $n^A = m^B = k$, and $n^B = n$. For simplicity, we will take $\alpha = 1$ and $\beta = 0$ in our discription. If a_{ij}, b_{ij} and c_{ij} denote the (i, j) element of the matrices, respectively, then the elements of C are given by

$$
c_{ij} = \sum_{l=1}^{k} a_{ik} b_{kj}
$$

Notice that rows of C are computed from rows of A, and columns of C are computed from columns of B. We hence restrict our data decomposition so that rows of A and C are assigned to the same row of nodes and columns of B and C are assigned to the same column of nodes. Hence, $m_i^C = m_i^A$ and $n_j^C = n_j^B$.

5.1 Basic Parallel Algorithm

Let us consider what computation is required to form C_{ij}:

$$C_{ij} = \overbrace{\left(\begin{array}{c|c|c|c} A_{i0} & A_{i1} & \cdots & A_{i(c-1)} \end{array} \right)}^{\tilde{A}_i} \left.\left(\begin{array}{c} B_{0j} \\ \hline B_{1j} \\ \hline \vdots \\ \hline B_{(r-1)j} \end{array} \right)\right\} \tilde{B}^j$$

Notice that $\tilde{A}_i$ is entirely assigned to node row i, while $\tilde{B}^j$ is entirely assigned to node column j. Letting

$$\tilde{A}_i = \left(\begin{array}{c|c|c|c} \tilde{a}_i^0 & \tilde{a}_i^1 & \cdots & \tilde{a}_i^{k-1} \end{array} \right) \text{ and } \tilde{B}^j = \left(\begin{array}{c} \tilde{b}_0^{j\,T} \\ \hline \tilde{b}_1^{j\,T} \\ \hline \vdots \\ \hline \tilde{b}_{k-1}^{j\,T} \end{array} \right)$$

we see that

$$C_{ij} = \sum_{l=0}^{k-1} \tilde{a}_i^l \tilde{b}_l^{j\,T}$$

Hence the matrix-matrix multiply can be formulated as a sequence of rank-one updates.

It now suffices to parallelize each rank-one update. Pseudo-code for this, executed simultaneously on all nodes $\mathbf{P}_{ij}$ is given in Fig. 1. The process is illustrated in Fig. 2.

To analyze the cost of this basic algorithm, we will make some simplifying assumptions: $m_i^C = m_i^A = m/r$, $n_i^C = n_i^B = n/r$, $n_i^A = k/c$, and $m_i^B = k/r$. Since relatively little data is involved during each broadcast, we will assume a minimum spanning tree broadcast is used and the cost of our algorithm is given by

$$k \left[\frac{2mn}{p}\gamma + \lceil \log(c) \rceil \left(\alpha + \frac{m}{r}\beta \right) + \lceil \log(r) \rceil \left(\alpha + \frac{n}{c}\beta \right) \right]$$

The terms within the square brackets are due to the rank-one update, broadcast within row, and broadcast within column, respectively. We ignore the packing required before sending rows of $\tilde{B}^i$. The total time is thus

$$T(m,n,k,p) = \frac{2mnk}{p}\gamma + k(\lceil \log(c) \rceil + \lceil log(r) \rceil)\alpha + \lceil \log(c) \rceil \frac{mk}{r}\beta + \lceil \log(r) \rceil \frac{nk}{c}\beta \qquad (5)$$

This compares to a sequential time of $2mnk\gamma$.

To establish the scalability of this approach, we will analyse the case where $m = n = k$, $r = c = \sqrt{p}$, and p is a power of two. Given the complexity in (5), the estimated speedup is

$$S(n,p) = \frac{2n^3\gamma}{\frac{2n^3}{p}\gamma + n\log(p)\alpha + \log(p)\frac{n^2}{\sqrt{p}}\beta} = \frac{p}{1 + \frac{p\log(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}\log(p)}{2n}\frac{\beta}{\gamma}}$$

The corresponding efficiency is

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{1 + \frac{p\log(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}\log(p)}{2n}\frac{\beta}{\gamma}} = \frac{1}{1 + O\left(\frac{p\log(p)}{n^2}\right) + O\left(\frac{\sqrt{p}\log(p)}{n}\right)}$$

Ignoring the $\log(p)$ term, which grows *very* slowly when p is reasonably large, we notice the following: If we increase p and we wish to maintain efficiency, we must increase n with $\sqrt{p}$. Since memory requirements grow with n^2, and physical memory grows linearly with p as nodes are added, we conclude that the method is scalable in the following sense: If we maintain memory use *per node*, this algorithm will maintain efficiency, if $\log(p)$ is treated as a constant.

Alternative broadcast algorithms, e.g. pipelined or scatter-collect broadcasts [4, 20] can be used to eliminate the $\log(p)$ factor, at the expense of a larger number of startups. We instead will present the benefits of pipelining *computation and communication.*

$C_{ij} = 0$
for $l = 0, k-1$
  broadcast $\tilde{a}_i^l$ within my row
  broadcast $\tilde{b}_l^j$ within my column
  $C_{ij} = C_{ij} + \tilde{a}_i^l \tilde{b}_l^{j\,T}$
endfor

Figure 1: Pseudo-code for $C = AB$.

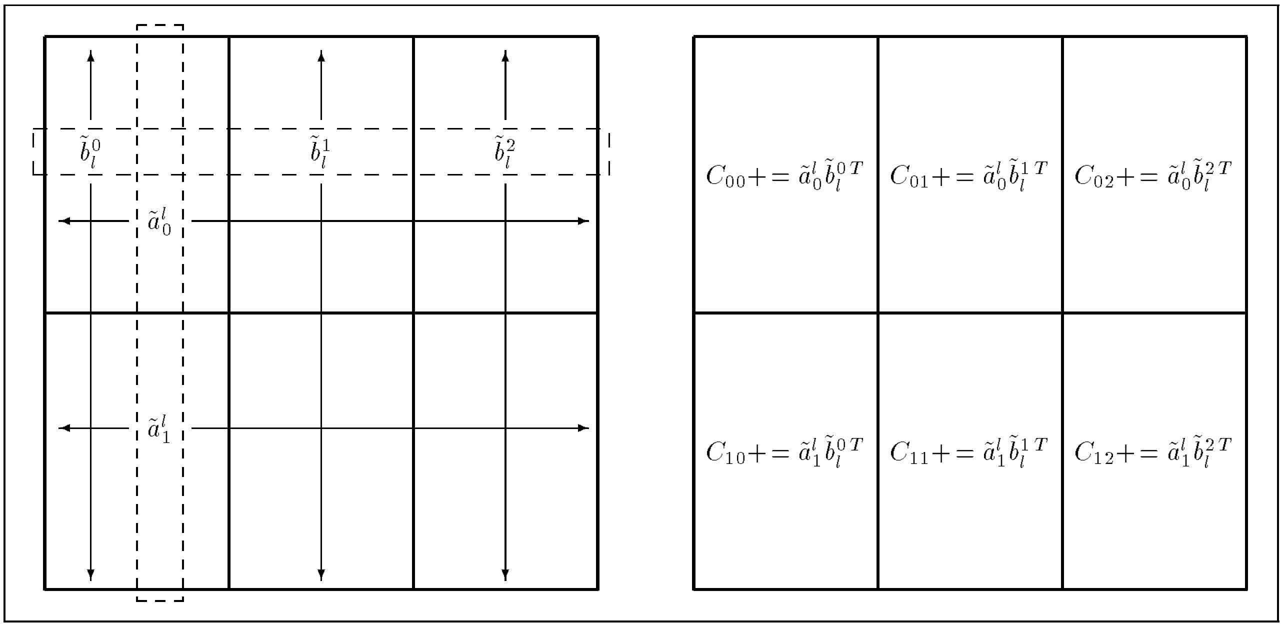

Figure 2: Operations implementing the inner loop of Fig. 1 of a 2×3 mesh of nodes.

5.2 Pipelined Algorithm

Let us consider implementing the broadcast as passing of a message around the logical ring that forms the row or column. In this case, the time complexity becomes:

$$
\begin{aligned}
& (c-1)\left(\alpha+\frac{m}{r}\beta\right)+(r-1)\left(\alpha+\frac{n}{c}\beta\right) && (6)\\
& \quad +k\left(\frac{2mn}{p}\gamma+\alpha+\frac{m}{r}\beta+\alpha+\frac{n}{c}\beta\right) && (7)\\
& \quad +(c-2)\left(\alpha+\frac{m}{r}\beta\right)+(r-2)\left(\alpha+\frac{n}{c}\beta\right) && (8)\\
& \quad +\frac{2mn}{p}\gamma && (9)\\
= & \quad \frac{2mn(k+1)}{p}\gamma+(k+2c-3)\left(\alpha+\frac{m}{r}\beta\right)+(k+2r-3)\left(\alpha+\frac{n}{c}\beta\right) && (10)
\end{aligned}
$$

Contribution (6) equals the the time required for both the first column of $\tilde{A}_{r-1}$ and the first row of $\tilde{B}^{c-1}$ to reach $\mathbf{P}_{(r-1)(c-1)}$ (filling the pipe); (7) equals the time for performing the local update and passing the messages; (8) equals the time for the final messages (initiated at $\mathbf{P}_{(r-1)(c-1)}$) to reach the end of the pipe; and (9) equals the time for the final update at the node at the end of the pipe ($\mathbf{P}_{(r-1)(c-2)}$ or $\mathbf{P}_{(r-2)(c-1)}$). Notice that for large k, the "log" factors in Eqn. (5) are essentially removed.

To establish the scalability of the pipelined approach, we will again analyse the case where $m = n = k$ and $r = c = \sqrt{p}$. This changes the complexity in (10) to approximately

$$\frac{2n^3}{p}\gamma + 2(n+2\sqrt{p}-3)\left(\alpha+\frac{n}{\sqrt{p}}\beta\right)$$

and the estimated speedup is

$$S(n,p) = \frac{2n^3\gamma}{\frac{2n^3}{p}\gamma + 2(n+2\sqrt{p}-3)\left(\alpha+\frac{n}{\sqrt{p}}\beta\right)} \approx \frac{p}{1+\frac{p}{n^2}\frac{\alpha}{\gamma}+\frac{\sqrt{p}}{n}\frac{\beta}{\gamma}}$$

The corresponding efficiency is

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{1+O\left(\frac{p}{n^2}\right)+O\left(\frac{\sqrt{p}}{n}\right)}$$

The $\log(p)$ term has disappeared and the method is again scalable in the sense that if we maintain memory use *per node*, this algorithm will maintain efficiency.

5.3 Blocking

Further improvements can be obtained by observing that reformulating the method in terms of matrix-matrix multiplications instead of rank-one updates can greatly improve the performance of an individual node. Matrix-matrix operations perform $O(n^3)$ computation on $O(n^2)$ data, thereby overcoming the memory access bandwidth bottleneck present on most modern microprocessors. Highly optimized versions of an important set of such operations (the level-3 BLAS [10]) are typically provided by major vendors of high performance microprocessors.

We can stage the computation using matrix-matrix multiplication by accumulating several columns of $\tilde{A}_i$ and rows of $\tilde{B}_j$ before updating the local matrix: In our explanation, each column $\tilde{a}_i^l$ becomes a *panel* of columns, and row $\tilde{b}_j^l$ a corresponding *panel* of rows.

An additional advantage of blocking is that it reduces the number of messages incurred, thereby reducing communication overhead.

5.4 Code

MPI code for the pipelined, blocked algorithm is given in Figs. 3 and 4. In the algorithm, arrays `a`, `b`, and `c` hold the local matrices in column-major order. Parameters `lda`, `ldb`, and `ldc` indicate the local leading dimension for arrays `a`, `b`, and `c`, respectively. Variables `m`, `n`, and `k` hold m, n, and k, respectively. Entries `m_a[i]`, `n_a[i]`, `m_b[i]`, `n_b[i]`, `m_c[i]`, and `n_c[i]` hold m_i^A, n_i^A, m_i^B, n_i^B, m_i^C, and n_i^C, respectively. Variable `nb` indicates the number of columns of $\tilde{A}_i$ and rows of $\tilde{B}^j$ that are accumulated before updated the local block of C. There are two work arrays: `work1` and `work2`, in which `nb` columns of $\tilde{A}_i$ and `nb` rows of $\tilde{B}^j$, respectively, can be accumulated. MPI communicators indicating the nodes that constitute a row and column in the logical node mesh are provided in `comm_row` and `com_col`.

Whenever possible, BLAS calls are used, and we use the LAPACK utility routine `dlacpy` to copy matrices to matrices.

6 Forming $C = \alpha A^T B^T + \beta C$

Forming $C = \alpha A^T B^T + \beta C$ can be easily derived by noting that $C^T = BA$ and reversing the roles of rows and columns in the algorithm given in Section 5.

7 Forming $C = \alpha A B^T + \beta C$

One approach to implementing this algorithm is to transpose matrix B followed by the algorithm presented in the previous section. We will show how to avoid this initial communication.

For this operation to be well-defined, we require $n^A = n$, $m^A = m^B = k$, and $n^B = m$. Again, we take $\alpha = 1$ and $\beta = 0$ in our description. If a_{ij}, b_{ij} and c_{ij} denote the (i, j) element of the matrices, respectively, then the elements of C are given by

$$c_{ij} = \sum_{l=1}^{k} a_{ik} b_{jk}$$

For our algorithm, we make the restriction that columns of A and columns of B are assigned to the same column of nodes, and rows of A and rows of C are assigned to the same row of nodes.

7.1 Basic Algorithm

Let

$$\tilde{C}_i = \left(\begin{array}{c|c|c|c} C_{i0} & C_{i1} & \cdots & C_{i(c-1)} \end{array} \right) = \left(\begin{array}{c|c|c|c} \tilde{c}_i^0 & \tilde{c}_i^1 & \cdots & \tilde{c}_i^{k-1} \end{array} \right)$$

and

$$\tilde{B}^j = \left(\begin{array}{c} \tilde{b}_0^{j\,T} \\ \hline \tilde{b}_1^{j\,T} \\ \hline \vdots \\ \hline \tilde{b}_{k-1}^{j\,T} \end{array} \right)$$

Then matrix algebra tells us that

$$c_i^l = \sum_{j=0}^{c-1} A_{ij} \tilde{b}_l^{j\,T}$$

Since rows of C_{ij} and A_{ij} are assigned to the same row of nodes, and elements of $\tilde{b}_l^j$ and columns of A_{ij} are assigned to the same column of nodes, we derive the pseudo-code given in Fig. 5 for node $\mathbf{P}_{ij}$. A picture describing the mechanism is given in Fig. 6.

```
#include "mpi.h"
                /* macro for column major indexing                 */
#define A( i,j ) (a[ j*lda + i ])
#define B( i,j ) (b[ j*ldb + i ])
#define C( i,j ) (c[ j*ldc + i ])

#define min( x, y ) ( (x) < (y) ? (x) : (y) )

int    i_one=1;         /* used for constant passed to blas call   */
double d_one=1.0,
       d_zero=0.0;      /* used for constant passed to blas call   */

void pdgemm( m, n, k, nb, alpha, a, lda, b, ldb,
             beta, c, ldc, m_a, n_a, m_b, n_b, m_c, n_c,
             comm_row, comm_col, work1, work2 )
int    m, n, k,         /* global matrix dimensions                */
       nb,              /* panel width                             */
       m_a[], n_a[],    /* dimensions of blocks of A               */
       m_b[], n_b[],    /* dimensions of blocks of A               */
       m_c[], n_c[],    /* dimensions of blocks of A               */
       lda, ldb, ldc;   /* leading dimension of local arrays that
                           hold local portions of matrices A, B, C */
double *a, *b, *c,      /* arrays that hold local parts of A, B, C */
       alpha, beta,     /* multiplication constants                */
       *work1, *work2;  /* work arrays                             */
MPI_Comm comm_row,      /* Communicator for this row of nodes      */
       comm_col;        /* Communicator for this column of nodes   */
{
  int myrow, mycol,     /* my  row and column index                */
      nprow, npcol,     /* number of node rows and columns         */
      i, j, kk, iwrk,   /* misc. index variables                   */
      icurrow, icurcol, /* index of row and column that hold current
                           row and column, resp., for rank-1 update*/
      ii, jj;           /* local index (on icurrow and icurcol, resp.)
                           of row and column for rank-1 update     */
  double *temp;         /* temporary pointer used in pdgemm_abt    */
  double *p;

                /* get myrow, mycol                                */
  MPI_Comm_rank( comm_row, &mycol );  MPI_Comm_rank( comm_col, &myrow );
                /* scale local block of C                          */
  for ( j=0; j<n_c[ mycol ]; j++ )
    for ( i=0; i<m_c[ myrow ]; i++ )
      C( i,j ) = beta * C( i,j );
```

Figure 3: MPI code for $C = \alpha AB + \beta C$ and $C = \alpha AB^T + \beta C$.

```
  icurrow = 0;          icurcol = 0;
  ii = jj = 0;
                /* malloc temp space for summation                 */
  temp = (double *) malloc( m_c[myrow]*nb*sizeof(double) );

  for ( kk=0; kk<k; kk+=iwrk) {
    iwrk = min( nb, m_b[ icurrow ]-ii );
    iwrk = min( iwrk, n_a[ icurcol ]-jj );
                /* pack current iwrk columns of A into work1       */
    if ( mycol == icurcol )
       dlacpy_( "General", &m_a[ myrow ], &iwrk, &A( 0, jj ), &lda, work1,
                &m_a[ myrow ] );
                /* pack current iwrk rows of B into work2          */
    if ( myrow == icurrow )
       dlacpy_( "General", &iwrk, &n_b[ mycol ], &B( ii, 0 ), &ldb, work2,
                &iwrk );
                /* broadcast work1 and work2                       */
    RING_Bcast( work1 , m_a[ myrow ]*iwrk, MPI_DOUBLE, icurcol, comm_row );
    RING_Bcast( work2 , n_b[ mycol ]*iwrk, MPI_DOUBLE, icurrow, comm_col );
                /* update local block                              */
    dgemm_( "No transpose", "No transpose", &m_c[ myrow ], &n_c[ mycol ],
            &iwrk, &alpha, work1, &m_b[ myrow ], work2, &iwrk, &d_one,
            c, &ldc );
                 /* update icurcol, icurrow, ii, jj                 */
    ii += iwrk;           jj += iwrk;
    if ( jj>=n_a[ icurcol ] ) { icurcol++; jj = 0; };
    if ( ii>=m_b[ icurrow ] ) { icurrow++; ii = 0; };
  }
  free( temp );
}


RING_Bcast( double *buf, int count, MPI_Datatype type, int root,
            MPI_Comm comm )
{
  int me, np;
  MPI_Status status;

  MPI_Comm_rank( comm, me );    MPI_Comm_size( comm, np );
  if ( me != root)
    MPI_Recv( buf, count, type, (me-1+np)%np, MPI_ANY_TAG, comm );
  if ( ( me+1 )%np != root )
    MPI_Send( buf, count, type, (me+1)%np, 0, comm );
}
```

Figure 4: MPI code for $C = \alpha AB + \beta C$ continued. See Fig. 3 for first part of code.

$C_{ij} = 0$
for $l = 0, k-1$
broadcast $\tilde{b}_l^j$ within my column
form $\tilde{c}_i^{l,j} = A_{ij} \tilde{b}_l^{j\,T}$
sum all $\tilde{c}_i^{l,j}$ within my row to the
node that holds $\tilde{c}_i^l$
endfor

Figure 5: Pseudo-code for $C = AB^T$.

Assuming minimum spanning tree broadcast and sum-to-one are used, the cost becomes

$$k\left[\lceil \log(c) \rceil (\alpha + m/r\beta) + \frac{2kn}{p}\gamma + \lceil \log(r) \rceil (\alpha + k/c\beta + n/c\gamma)\right] \tag{11}$$

Scalability properties are much like those of the algorithm for $C = AB$.

7.2 Pipelining

As with forming $C = AB$, the above algorithm can be improved by introducing pipelining. Let us consider implementing the broadcast as a passing of the message around the logical ring that forms the column. Similarly, let the summation within rows be implemented as a passing of a "bucket" that collects all local contributions to the node on which the result is required. The effect on the time complexity is much like that obtained for the formation of $C = AB$.

7.3 Blocking

As for the computation of $C = AB$, further improvements can be obtained by observing that reformulating the method in terms of matrix-matrix multiplications instead of matrix-vector multiplications can greatly improve the performance of an individual node. This can be accomplished by taking both $\tilde{c}_i^l$ and $\tilde{b}_l^j$ to be a small number of columns and rows, respectively.

Again, communication overhead is reduced as well.

7.4 Code

MPI code for the pipelined, blocked algorithm is given in Figs. 3 and 7. The parameters are essentially the same as those used for forming $C = \alpha AB + \beta C$.

8 Forming $C = \alpha A^T B + \beta C$

Forming $C = \alpha A^T B + \beta C$ can be easily derived by noting that $C^T = \alpha B A^T + \beta C^T$ and reversing the roles of rows and columns in the algorithm given in Section 7.

9 Performance Results

We do not compare the achieved performance with predicted performance. The reason for this is that there are too many parameters that cannot be easily controlled. For example, the performance of the BLAS kernel

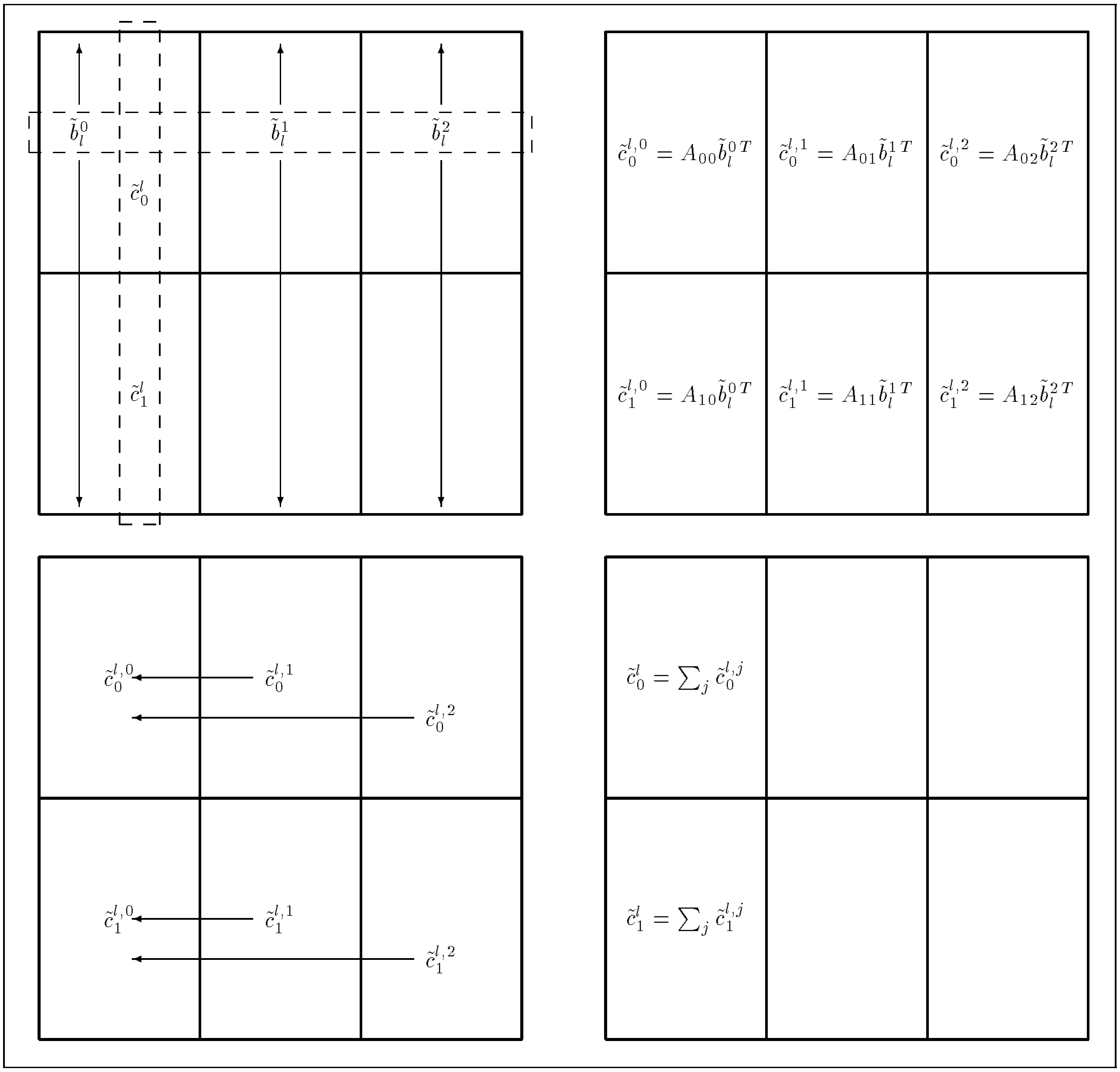

Figure 6: Operations implementing the inner loop of Fig. 5 of a 2×3 mesh of nodes.

```
  icurrow = 0;          icurcol = 0;
  ii = jj = 0;
                /* malloc temp space for summation                 */
  temp = (double *) malloc( m_c[myrow]*nb*sizeof(double) );
                /* loop over all column panels of C                */
  for ( kk=0; kk<k; kk+=iwrk) {
    iwrk = min( nb, m_b[ icurrow ]-ii );
    iwrk = min( iwrk, n_c[ icurcol ]-jj );
                /* pack current iwrk rows of B into work2          */
    if ( myrow == icurrow )
       dlacpy_( "General", &iwrk, &n_b[ mycol ], &B( ii, 0 ), &ldb, work2,
                &iwrk );
                /* broadcast work2                                 */
    RING_Bcast( work2 , n_b[ mycol ]*iwrk, MPI_DOUBLE, icurrow, comm_col );
                /* Multiply local block of A times incoming rows of B */
    dgemm_( "No transpose", "Transpose", &m_c[ myrow ], &iwrk,
            &n_a[ mycol ], &alpha, a, &lda, work2, &iwrk, &d_zero,
            work1, &m_c[ myrow ] );
                /* Sum to node that holds current columns of C     */
    RING_SUM( work1, m_c[ myrow ]*iwrk, MPI_DOUBLE, icurcol, comm_row, temp );
                /* Add to current columns of C                     */
    if ( mycol == icurcol ) {
      p = work1;
      for (j=jj; j<jj+iwrk; j++) {
        daxpy_( &m_c[ myrow ], &d_one, p, &i_one, &C( 0,j ), &i_one );
        p += m_c[ myrow ];
      }
    }
                /* update icurcol, icurrow, ii, jj                 */
    ii += iwrk;          jj += iwrk;
    if ( jj>=n_c[ icurcol ] ) { icurcol++; jj = 0; };
    if ( ii>=m_b[ icurrow ] ) { icurrow++; ii = 0; };
  }
  free( temp );
}


RING_SUM( double *buf, int count, MPI_Datatype type, int root,
          MPI_Comm comm, double *work )
{
  int me, np;
  MPI_Status status;

  MPI_Comm_rank( comm, &me );    MPI_Comm_size( comm, &np );
  if ( me != (root+1)%np ) {
    MPI_Recv( work, count, type, (me-1+np)%np, MPI_ANY_TAG, comm, &status );
    daxpy_( &count, &d_one, work, &i_one, buf, &i_one );
  }
  if ( me != root )
    MPI_Send( buf, count, type, (me+1)%np, 0, comm );
}
```

Figure 7: MPI code for $C = \alpha A B^T + \beta C$ continued. See Fig. 3 for first part of code

`dgemm`, which is used for the matrix-matrix multiplication on a single node is highly dependent on the matrix size and other circumstances. Instead, we compare the performance of our basic matrix-multiplication algorithms for $C = \alpha AB + \beta C$ and $C = \alpha AB^T + \beta C$ with that achieved by the PUMMA implementation in [8], which we obtained from `netlib`. We modified the PUMMA code to call the most efficient forms of NX communication primitives (including forced messages).

The implementation of SUMMA is essentially the one given in this paper. We implemented it using the MPI send and receive primitives used in the algorithms in this paper. However, the current MPI implementation on the Paragon incurs high latency and achieves lower bandwidth that the equivalent native NX calls. For that reason, we also implemented a highly optimized version that uses "forced" (ready-receive) messages and NX calls.

In Figs. 8 and 9, we report the performance *per node* for $C = AB$ as a function of problem size for the two SUMMA versions and PUMMA. Peak performance observed for `dgemm` on a single node is in the 45 MFLOPS range. The blocking size (`nb`) chosen for both PUMMA and SUMMA was 20, which appears to give the best performance. Much more dramatic is the difference between PUMMA and SUMMA on a non-power-two mesh, results of which are reported in Fig. 10. The primary reason is that the broadcast-multiply-roll algorithm generalizes more readily to nonsquare meshes when either the row or column dimension is an integer multiple of the other dimension. When this is not the case, performance suffers dramatically. SUMMA doesn't have the same kind of dependency, and it performs well.

In Figs. 11 and 12, we report the performance *per node* as a function of the number of nodes (p) *when memory use per node is held constant* for $C = AB$. Notice that performance can essentially be maintained, as predicted by our analyses. Also, the cases we present represent relatively small problems: a local 500×500 problem requires only 2 Mbytes of memory. We need space for three of these matrices, plus a small amount of workspace. The Paragon typically comes with at least 32 Mbytes of memory per node.

In Fig. 13, we report the performance per node for $C = AB^T$ as a function of problem size for SUMMA and PUMMA. Peak performance on a single node is again in the 45 MFLOPS range. The observed performance is very similar to that of $C = AB$. Again, SUMMA is competitive with PUMMA. In Fig. 14, we show the scalability of the this algorithm.

It should be noted that PUMMA requires considerably more workspace per node than the amount of memory used to store the local matrices. SUMMA by comparison uses a lower order amount of memory per node, which means that SUMMA has the added advantage of allowing larger problems to be run.

10 Generalization of the Code

The codes presented in this paper are slight simplifications, allowing us to include them in the body of the paper. Notice that the following generalizations are easily obtained:

10.1 Relaxing the dependence on multiples of `nb` .

We will limit our discussion in this section to the forming of $C = AB$ only. The same techniques can be easily applied to the other cases.

Notice that the code requires nodes to have a multiple of `nb` number of rows and columns of the matrix, except for the last row and column of nodes. There are two alternatives for overcoming this restriction:

- The width of the current row and column panels can be reduced when a border between nodes is encountered.
- Partial panels can be passed to the next row or column of nodes, to be filled fully before broadcasting.

The latter is particularly easy to incorporate into the pipelined version of the algorithm.

10.2 Relaxing the alignment of the matrices

Notice that it suffices that the same rows of A and C are assigned to the same row of nodes, and the same columns of B and C to the same column of nodes. If A and B are not aligned with C in the other dimension, it is a matter of passing in `icurrow` and `icurcol` as parameters to the routine.

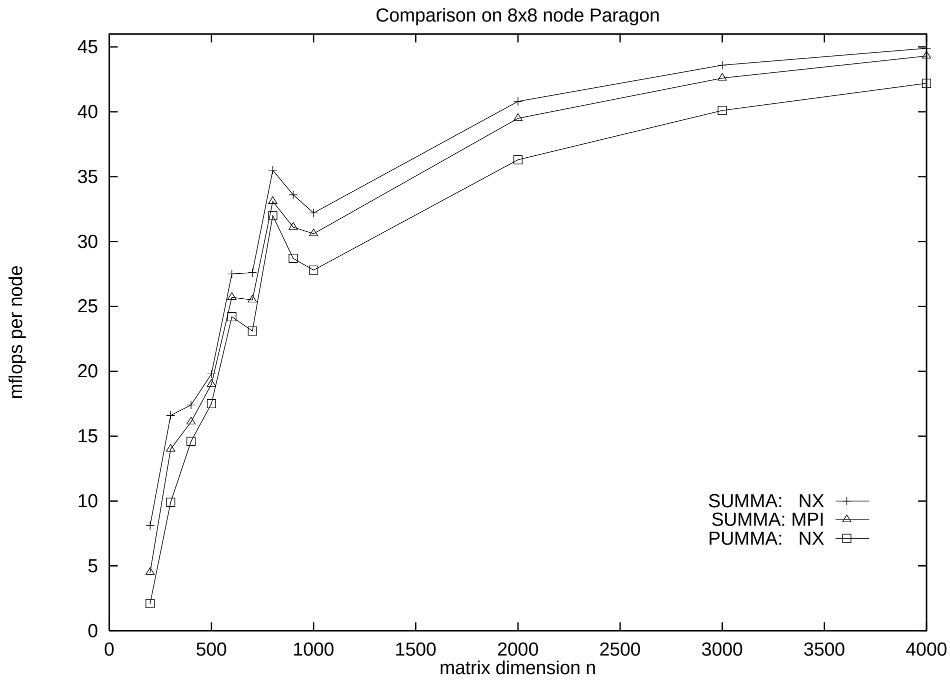

Figure 8: Performance of SUMMA vs. PUMMA for $C = AB$ on 64 nodes.

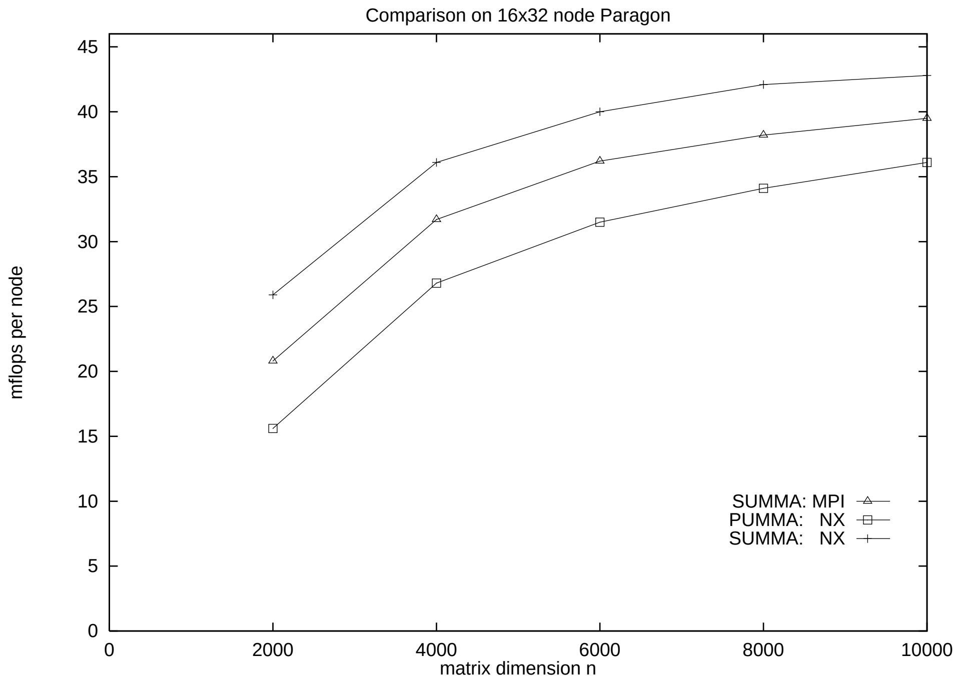

Figure 9: Performance of SUMMA vs. PUMMA for $C = AB$ on 512 nodes.

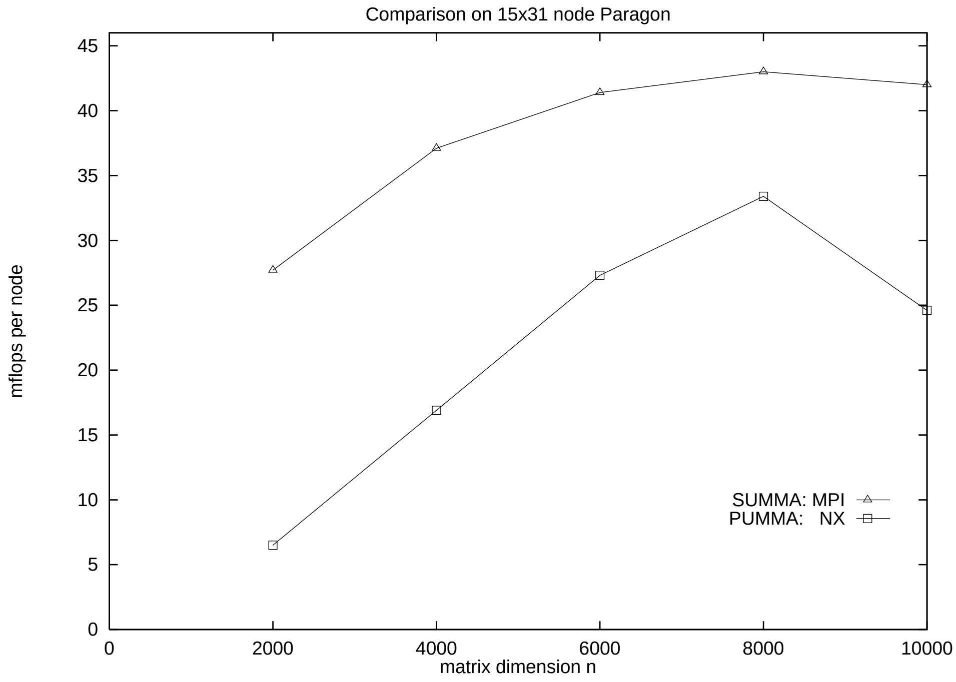

Figure 10: Performance of SUMMA vs. PUMMA for $C = AB$ on 465 nodes.

10.3 Generalizing the matrix decomposition

We will further illustrate the flexibility of our approach by showing how the algorithm in Fig. 4 can be easily changed to handle block-wrapped data decompositions like those used for ScaLAPACK. Indeed, it is a matter of changing the code segments

```
iwrk = min( nb, m_b[ icurrow ]-ii );
iwrk = min( iwrk, n_a[ icurcol ]-jj );
```

to

```
iwrk = min( nb, k-kk );
```

and

```
          /* update icurcol, icurrow, ii, jj                 */
ii += iwrk;          jj += iwrk;
if ( jj>n_a[ icurcol ] ) { icurcol++; jj = 0; };
if ( ii>m_b[ icurrow ] ) { icurrow++; ii = 0; };
```

to

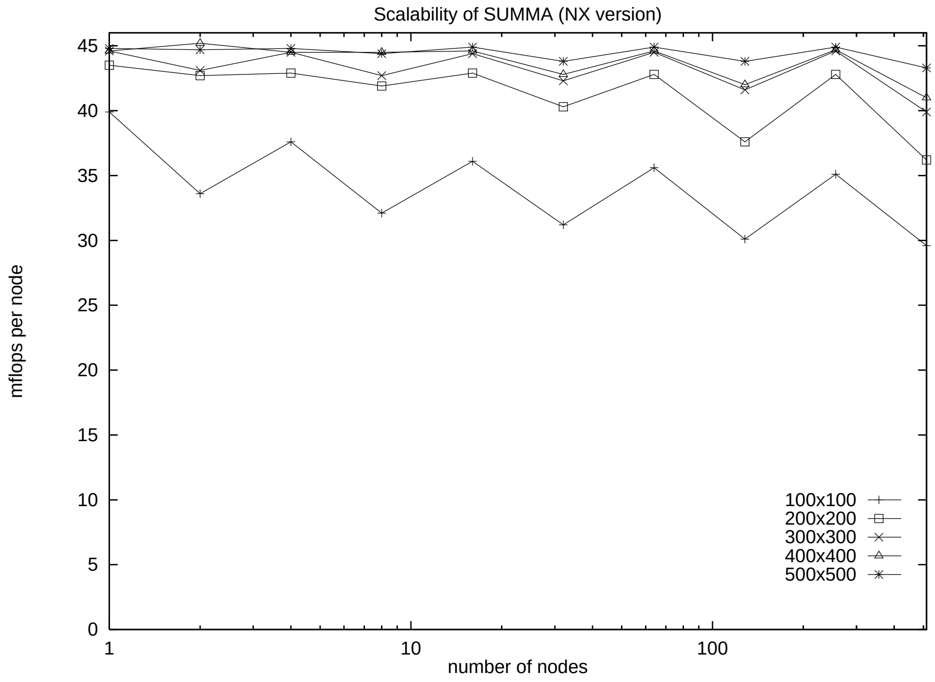

Figure 11: Performance of SUMMA (nx version) for $C = AB$ as a function of the number of nodes, when memory use per node is held constant. The curves for 100×100, 200×200, etc., indicate the use of local memory equivalent to that necessary to store a matrix of the indicated size. E.g., on 64 and 128 nodes, 100×100 is equivalent to a 800×800 and 1128×1128 global matrix, respectively. The "zigzagging" of the curves is due to the effects of square vs. nonsquare meshes.

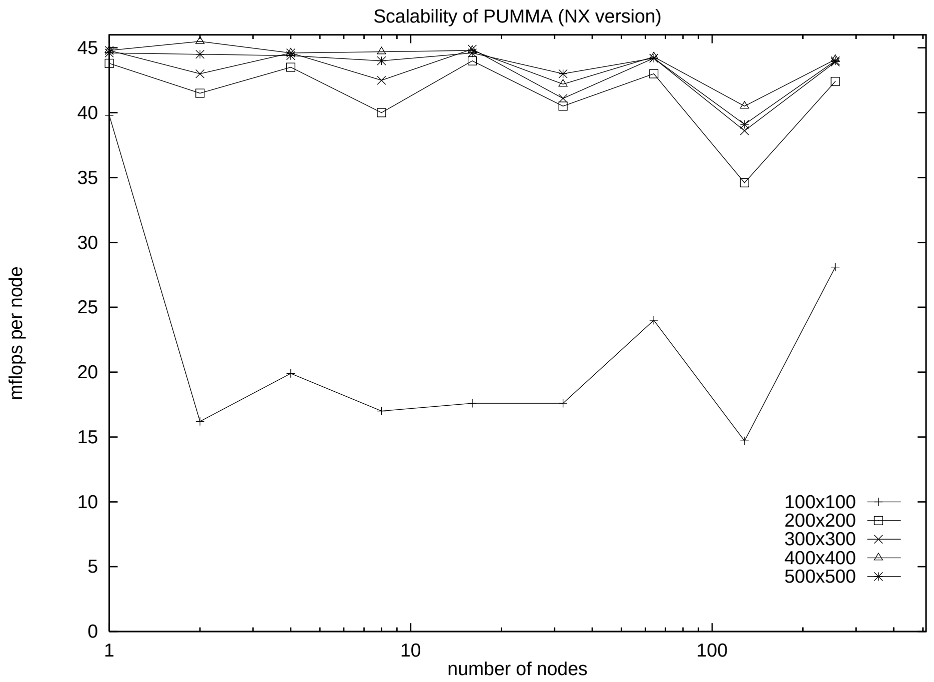

Figure 12: Performance of PUMMA for $C = AB$ as a function of the number of nodes, when memory use per node is held constant.

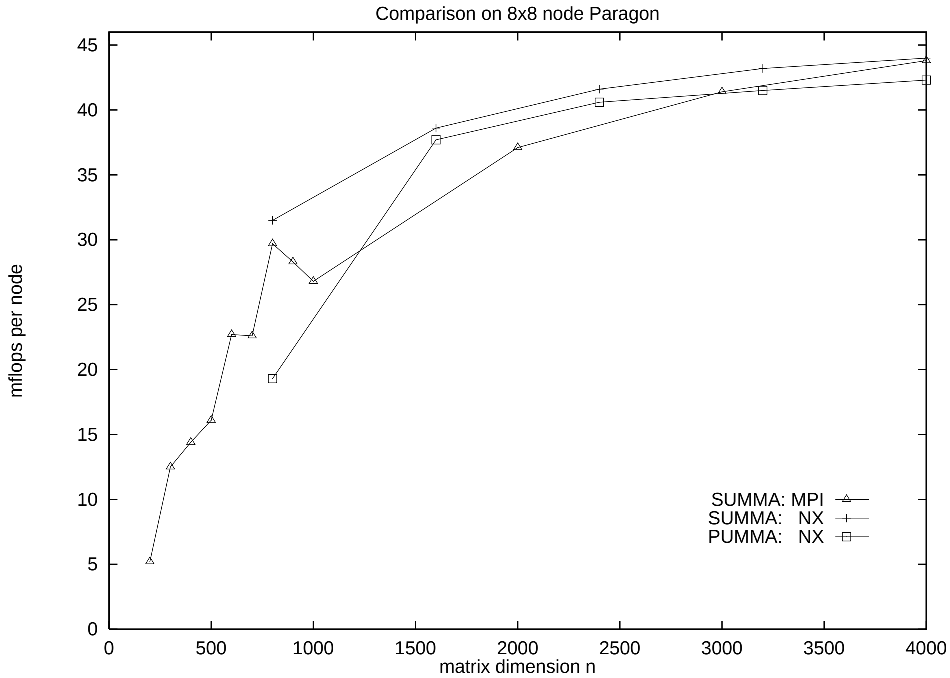

Figure 13: Performance of SUMMA vs. PUMMA for $C = AB^T$ on 64 nodes.

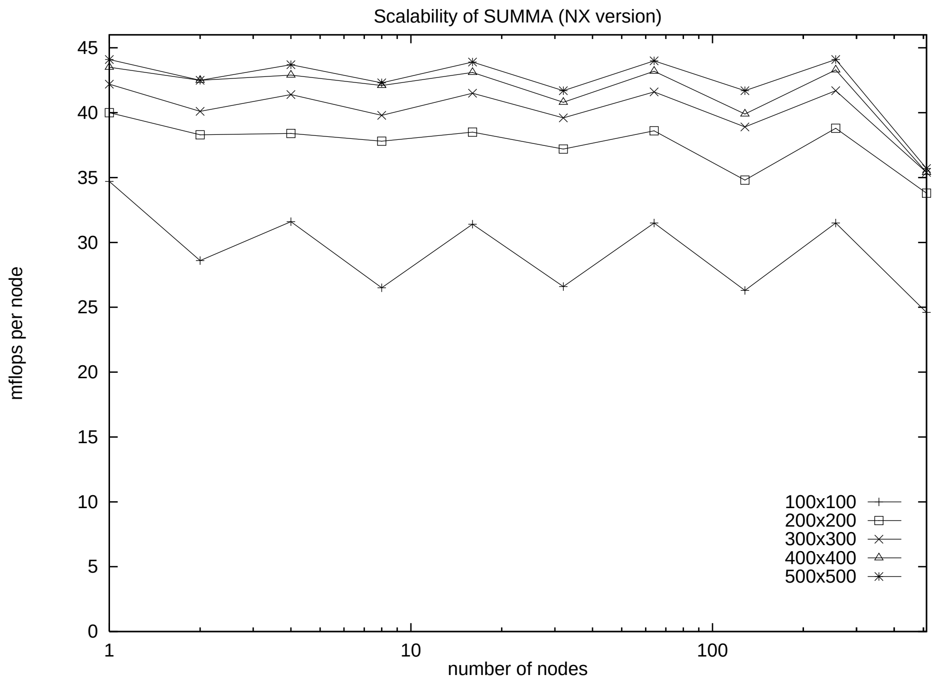

Figure 14: Performance of SUMMA (nx version) for $C = AB^T$ as a function of the number of nodes, when memory use per node is held constant. The curves for 100×100, 200×200, etc., indicate the use of local memory equivalent to that necessary to store a matrix of the indicated size. E.g., on 64 and 128 nodes, 100×100 is equivalent to a 800×800 and 1128×1128 global matrix, respectively.

```
            /* update icurcol, icurrow, ii, jj                */
    if ( myrow == icurrow ) ii += iwrk;
    if ( mycol == icurcol ) jj += iwrk;
    icurrow = ( icurrow+1 )%nprow;
    icurcol = ( icurcol+1 )%npcol;
```

Other alternative matrix decompositions can be handled similarly.

10.4 Odd-shaped matrices

A frequent use of matrix-matrix multiplication in applications is the case where k is much smaller than m and n. Examples of this occur in ScaLAPACK routines like those for the LU and QR factorization [11]. In such cases, our approach continues to be useful. However it may be necessary to substitute a minimum spanning tree broadcast or other broadcast that does not rely on pipelining of communication and computations.

10.5 Pipelining multiplications

Another interesting observation is that if a number of matrix-matrix multiplications need to be performed, the communication and computation can be pipelined *between individual multiplications.*

11 SUMMA, ScaLAPACK, and Distributed BLAS

We believe that the SUMMA approach is particularly appropriate for implementation of distributed BLAS implementations of the matrix-matrix multiplication. We summarize those in this section.

It is very interesting to note that we started pursuing the presented algorithm by making the following simple observation: The blocked right-looking LU factorization, as implemented in LAPACK [1, 2], is much like a matrix-matrix multiplication, $C = AB$, implemented as a series of rank `nb` updates, except that they require pivoting, matrices A, B, and C are all the same matrix, and the updates progressively affect less of the matrix being updated. Similarly, the approach used to derive the algorithm for $C = AB^T$ was actually inspired by the implementation of a left-looking Cholesky factorization.

We hence suspect that a ScaLAPACK implementation based on a distributed BLAS matrix-matrix multiplication would naturally benefit from SUMMA.

12 Conclusion

The presented algorithms for matrix-matrix multiplication are considerably simpler than those previously presented, all of which have been based on generalizations of the broadcast-multiply-roll algorithm. Nonetheless, their performance is competitive or better, and they are considerably more flexible. Finally, their memory use for work arrays is much lower than those of the broadcast-multiply-roll algorithm. As a result, we believe the SUMMA approach to be the natural choice for a general-purpose implementation.

We should note that on some systems PUMMA may very well outperform SUMMA under some circumstances. In particular, SUMMA is slightly more sensitive to communication overhead. However, it is competetive, or faster, and given its simplicity and flexibility, warrants consideration. Moreover, the implementations by Huss-Lederman et. al. are competitive with PUMMA, and would thus compare similarly with SUMMA.

Acknowledgements

This research was performed in part using the Intel Paragon System and the Intel Touchstone Delta System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium.

Access to this facility was provided by Intel Supercomputer Systems Division and the California Institute of Technology.

References

[1] Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "Lapack: A Portable Linear Algebra Library for High Performance Computers," *Proceedings of Supercomputing '90,* IEEE Press, 1990, pp. 1–10.

[2] Anderson, E., Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide,* SIAM, Philadelphia, 1992.

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

[4] M. Barnett, D.G. Payne, R. van de Geijn, and J. Watts, Broadcasting on Meshes with Worm-Hole Routing, *Journal of Parallel and Distributed Computing*, submitted.

[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, and D. W. Walker, "Scalapack: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers, *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation.* IEEE Comput. Soc. Press, 1992, pp. 120-127.

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

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

[9] Dongarra, J. J., I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers,* SIAM, Philadelphia, 1991.

[10] Dongarra, J. J., J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *TOMS*, Vol. 16, No. 1, pp. 1–16, 1990.

[11] Dongarra, J. J., R. A. van de Geijn, and D. W. Walker, "Scalability Issues Affecting the Design of a Dense Linear Algebra Library," *Journal of Parallel and Distributed Computing,* Vol. 22, No. 3, Sept. 1994, pp. 523–537.

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

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

[14] Golub, G. H. , and C. F. Van Loan, *Matrix Computations,* Johns Hopkins University Press, 2nd ed., 1989.

[15] Gropp, W., E. Lusk, A. Skjellum, *Using MPI: Portable Programming with the Message-Passing Interface*, The MIT Press, 1994.

[16] C.-T. Ho and S. L. Johnsson, Distributed Routing Algorithms for Broadcasting and Personalized Communication in Hypercubes, In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 640–648, IEEE, 1986.

[17] Huss-Lederman, S., E. Jacobson, A. Tsao, "Comparison of Scalable Parallel Matrix Multiplication Libraries," in *Proceedings of the Scalable Parallel Libraries Conference*, Starksville, MS, Oct. 1993.

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

[19] Lin, C., and L. Snyder, "A Matrix Product Algorithm and its Comparative Performance on Hypercubes," in *Proceedings of Scalable High Performance Computing Conference*, (Stout, Q, and M. Wolfe, eds.), IEEE Press, Los Alamitos, CA, 1992, pp. 190–3.

[20] Watts, J. and R. van de Geijn, "A Pipelined Broadcast for Multidimensional Meshes," *Parallel Processing Letters,* to appear.