Parallel Implementation of BLAS: General Techniques for Level 3 BLAS*

Almadena Chtchelkanova
John Gunnels
Greg Morrow
James Overfelt
Robert A. van de Geijn

The University of Texas at Austin
Austin, Texas 78712

October 11, 1995

Abstract

In this paper, we present straight forward techniques for a highly efficient, scalable implementation of common matrix-matrix operations generally known as the Level 3 Basic Linear Algebra Subprograms (BLAS). This work builds on our recent discovery of a parallel matrix-matrix multiplication implementation, which has yielded superior performance, and requires little work space. We show that the techniques used for the matrix-matrix multiplication naturally extend to all important level 3 BLAS and thus this approach becomes an enabling technology for efficient parallel implementation of these routines and libraries that use BLAS. Representative performance results on the Intel Paragon system are given.

1 Introduction

Over the last five years, we have learned a lot about how to parallelize dense linear algebra libraries [5, 12, 15, 20, 21, 23, 36, 39]. Since much effort went into implementation of individual algorithms, it became clear that in order to parallelize entire sequential libraries, better approaches had to be found. Perhaps the most successful sequential library, LAPACK [3, 4], is built upon a few compute kernels, known as the Basic Linear Algebra Subprograms (BLAS). These include the level 1 BLAS (vector-vector operations) [32], level 2 BLAS (matrix-vector operations) [17], and level 3 BLAS (matrix-matrix operations) [18]. It is thus natural to try to push most of the effort into parallelizing these kernels, in the hope that good parallel implementations of codes that utilize these kernels will follow automatically. The above observations lead to two related problems: first, how to specify the parallel BLAS and second, how to implement them. The former is left to those working to standardize parallel BLAS interfaces [11]. This paper concentrates on the latter issue.

Notice that the dense linear algebra community has always depended on fast implementations of the BLAS provided by the vendors. Unless simple implementation techniques are developed, construction and maintenance of the parallel BLAS becomes too cumbersome, making it much more difficult to convince the vendors to provide fast implementations. We therefore must develop fast reference implementations, so that the vendors need only perform minor optimizations.

*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 and the PRISM project, which is sponsored by ARPA. Additional support came from the Intel Research Council.

In a recent paper [41], we describe a highly efficient implementation of matrix-matrix multiplication, Scalable Universal Matrix Multiplication Algorithm (SUMMA), that has many benefits over alternative implementations [14, 29, 30]. These benefits include better performance, simpler and more flexible implementation, and a lower workspace requirement. In this paper, we show how the simple techniques developed in that paper can be extended to all the level 3 BLAS.

2 Notation and Assumptions

2.1 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 [38] . Parameters α and β represent the startup and cost per item transfer time, respectively. Performing a floating point computation requires time γ.

2.2 Data decomposition

We will assume the classical data decomposition, generally referred to as a two dimensional block-wrapped, or block-cyclic, matrix decomposition [21].

In general, we would need to consider the case of matrices as being distributed using a *template matrix* as follows: Let template matrix X be partitioned as

$$X = \left(\begin{array}{c|c|c|c} X_{00} & X_{01} & X_{02} & \cdots \\ \hline X_{10} & X_{11} & X_{12} & \cdots \\ \hline X_{20} & X_{21} & X_{22} & \cdots \\ \hline \vdots & \vdots & \vdots & \end{array} \right)$$

where X_{ij} is a $n_r \times n_c$ submatrix. We will assume that the template is distributed to nodes so that X_{ij} is assigned to node $\mathbf{P}_{i \bmod r, j \bmod c}$. In other words, the template is block-wrapped onto the (logical) two dimensional mesh of nodes. The template matrix can be thought of as having infinite dimension. Since it doesn't really exist, it requires no space. For our implementations, the submatrices are square, i.e. $n_r = n_c = n_b$, which we will call the blocking factor. However, the techniques discussed in this paper encompass the more general case.

Matrices A, B, and C, involved in the matrix-matrix operation, are aligned to this template by specifying the element of template X with which the upper-left-hand element of the matrix is aligned, after which the elements of the matrices are distributed like corresponding elements of the template.

2.3 Calling sequences

In this section, we show how sequential calling sequences can be easily generalized to parallel calling sequences using the above observations. *This is not because we are proposing a standard. We are merely trying to illustrate what kind of additional information must be specified.* We will illustrate this for the simplest case of the matrix-matrix multiplication algorithm only, under the assumption that it will be clear how this generalizes.

The sequential calling sequence for the level 3 BLAS Double precision GEneral Matrix-matrix Multiplication is

Calling sequence	operation
`DGEMM ( "N", "N", M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC )`	$C = \tilde{\alpha} AB + \tilde{\beta} C$

Here `"N"`, `"N"` indicate that neither matrix A nor B are transposed, respectively. `M` and `N` are the row and column dimensions of matrix C, and `K` is the "other" dimension, in this case the column dimension of A and row dimension of B. Matrices A, B, and C are stored in arrays `A`, `B`, and `C`, respectively, which have leading dimensions `LDA`, `LDB`, and `LDC`.

In distributing the matrices, it suffices to specify the following for each:

- The *local* array in which it is stored on this node. We will assume that we simply use `A`, `B`, and `C`,
- The *local* leading dimensions `LLDA`, `LLDB`, and `LLDC`,
- The blocking factor n_b in parameter `NB`,
- MPI communicators indicating the row and column of the logical mesh: `comm_row` and `comm_col`, and
- The element of matrix template X with which the upper-right-hand element of the matrices are aligned: the row, column pairs (`IA, JA`), (`IB, JB`), and (`IC, JC`), respectively.

Thus the calling sequence becomes

Calling sequence	operation

`sB_DGEMM ( "N", "N", M, N, K, ALPHA, A, LLDA,`
`IA, JA, B, LLDB, IB, JB, BETA, C,`
`LLDC, IC, JC )`
	$C = \tilde{\alpha} AB + \tilde{\beta} C$

It will be assumed that `NB` is initialized through some initialization routine, as are the parameters and assignment of the logical $r \times c$ mesh of nodes as part of the MPI communicators.

2.4 Simplifying assumptions

In our explanations, we assume for simplicity that

- $\tilde{\alpha} = 1$ and $\tilde{\beta} = 0$,
- $n_r = n_c = n_b$,
- $m = n = k = N n_b$, and
- $IA = JA = IB = JB = IC = JC = 1$, i.e., the matrices are aligned with the $(1, 1)$ element of the template.

Thus, we will use the partitioning of the matrices

$$Y = \left(\begin{array}{c|c|c|c} Y_{00} & Y_{01} & \cdots & Y_{0(N-1)} \\ \hline Y_{10} & Y_{11} & \cdots & Y_{1(N-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline Y_{(N-1)0} & Y_{(N-1)1} & \cdots & Y_{(N-1)(N-1)} \end{array} \right) \tag{1}$$

with $Y \in \{A, B, C\}$ and Y_{ij} of size $n_b \times n_b$ assigned to $\mathbf{P}_{i \bmod r, j \bmod c}$. (We realize that slight confusion will result of our mixing of the FORTRAN convension of starting the indexing of arrays with 1 and the indexing of the blocks starting at 0, which leads to simpler explanations.)

3 Parallel Algorithms for the Level 3 BLAS

In this section, we give high level descriptions of the parallel implementations of all the level 3 BLAS.

3.1 Matrix-Matrix Multiplication (`xGEMM`)

3.1.1 Forming $C = AB$

In [41], we note the following:

$$
\begin{aligned}
C \;=\; AB &= \left(\begin{array}{c} A_{00} \\ \hline A_{10} \\ \hline \vdots \\ \hline A_{(N-1)0} \end{array}\right) \left(\begin{array}{c|c|c|c} B_{00} & B_{01} & \cdots & B_{0(N-1)} \end{array} \right) & (2) \\
&+ \left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline \vdots \\ \hline A_{(N-1)1} \end{array}\right) \left(\begin{array}{c|c|c|c} B_{10} & B_{11} & \cdots & B_{1(N-1)} \end{array} \right) + \cdots & (3) \\
&+ \left(\begin{array}{c} A_{0(N-1)} \\ \hline A_{1(N-1)} \\ \hline \vdots \\ \hline A_{(N-1)(N-1)} \end{array}\right) \left(\begin{array}{c|c|c|c} B_{(N-1)0} & B_{(N-1)1} & \cdots & B_{(N-1)(N-1)} \end{array} \right) & (4)
\end{aligned}
$$

where A_{*j} (B_{i*}) consists of n_b columns (rows) and will be referred to as a column (row) panel.

In other words, the matrix-matrix multiplication can be implemented as a sequence of N rank-n_b updates.

The parallel implementation can now proceed as given in Fig. 1.

> **in parallel** $C = 0$
> **for** $i = 0, N-1$
>   **broadcast** A_{*i} **within rows**
>   **broadcast** B_{i*} **within columns**
>   **in parallel**
> $$C = C + \left(\begin{array}{c} A_{0i} \\ \hline A_{1i)} \\ \hline \vdots \\ \hline A_{(N-1)i} \end{array}\right) \left(\begin{array}{c|c|c|c} B_{i0} & B_{i1} & \cdots & B_{i(N-1)} \end{array} \right)$$
> **endfor**

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

It has been pointed out to us that this implementation was already reported in [2]. In that paper the benefits of overlapping communication and computation are also studied.

3.1.2 Forming $C = AB^T$

One approach to implementing this case is to transpose matrix B followed by the algorithm presented in the previous section. In [41], we show how to avoid this initial communication:

$$
\left(\begin{array}{c} C_{0i} \\ \hline C_{1i} \\ \hline \vdots \\ \hline C_{(N-1)i} \end{array}\right) = A \left(\begin{array}{c|c|c|c} B_{i0} & B_{i1} & \cdots & B_{i(N-1)} \end{array} \right)
$$

C can thus be computed column panel at a time.

The parallel implementation can now proceed as given in Fig. 2

```
in parallel C = 0
for i = 0, N − 1
    broadcast B^T_{i*} within columns
    compute C_{*i} = A B^T_{i*} by
        computing local contribution
        summing local contributions to column of nodes
        that own C_{*i}
    endfor
```

Figure 2: Pseudo-code for $C = AB^T$.

3.1.3 Forming $C = A^T B$

This case can be implemented much like $C = AB^T$.

3.1.4 Forming $C = A^T B^T$

On the surface, it appears that this case can be implemented much like $C = AB$ by communicating row panels of A and column panels of B. However, this would compute C^T, requiring a transpose operation to bring the data to the final destination. We inherently dislike the transpose operation, and will therefore suggest an alternative approach. *We will limit ourselves considerably by only describing the algorithm for the case where the mesh of nodes is square.* How to generalize this approach will be briefly indicated in the conclusion.

Note that

$$C = A^T B^T = \left(A_{00} \mid A_{01} \mid \cdots \mid A_{0(N-1)} \right)^T \left(\begin{array}{c} B_{00} \\ \hline B_{10} \\ \hline \vdots \\ \hline B_{(N-1)0} \end{array} \right)^T \tag{5}$$

$$+ \left(A_{10} \mid A_{11} \mid \cdots \mid A_{1(N-1)} \right)^T \left(\begin{array}{c} B_{01} \\ \hline B_{11} \\ \hline \vdots \\ \hline B_{(N-1)1} \end{array} \right)^T + \cdots \tag{6}$$

$$+ \left(A_{(N-1)0} \mid A_{(N-1)1} \mid \cdots \mid A_{(N-1)(N-1)} \right)^T \left(\begin{array}{c} B_{0(N-1)} \\ \hline B_{1(N-1)} \\ \hline \vdots \\ \hline B_{(N-1)(N-1)} \end{array} \right)^T \tag{7}$$

In other words, the matrix-matrix multiplication can again be implemented as a sequence of N rank-n_b updates.

The parallel implementation is complicated by the fact that *this time the panels must be transposed first.* In the special case where the data is aligned as indicated and we have a square mesh of nodes ($r = c = \sqrt{p}$), the transpose can be accomplished by sending the appropriate subblocks of matrix A from the nodes in the *row* of nodes that holds the current row panel of the matrix (nodes $\mathbf{P}_{(i \bmod \sqrt{p})*}$) to the corresponding column of nodes (nodes $\mathbf{P}_{*(i \bmod \sqrt{p})}$). Similarly, the appropriate subblocks of matrix B must be sent from the nodes in the *column* of nodes that holds the current column panel of the matrix (nodes $\mathbf{P}_{*(i \bmod \sqrt{p})}$) to the corresponding row of nodes (nodes $\mathbf{P}_{(i \bmod \sqrt{p})*}$).

The parallel implementation can now proceed as given in Fig. 3.

in parallel $C = 0$
for $i = 0, N-1$
    transpose A_{i*}
    broadcast A_{i*} **within rows**
    transpose B_{*i}
    broadcast B_{*i} **within columns**
    in parallel
$$C = C + \left(\begin{array}{c} A_{0i}^T \\ \hline A_{1i}^T \\ \hline \vdots \\ \hline A_{(N-1)i}^T \end{array} \right) \left(\begin{array}{c|c|c|c} B_{i0}^T & B_{i1}^T & \cdots & B_{i(N-1)}^T \end{array} \right)$$
endfor

Figure 3: Pseudo-code for $C = A^T B^T$.

3.2 Triangular Matrix-Matrix Multiplication (xTRMM)

3.2.1 Forming $B = AB$ (A lower triangular)

Notice that if we wish to form $C = AB$, this operation can be easily derived from the case where A is a full matrix:

$$C = AB = \left(\begin{array}{c} A_{00} \\ \hline A_{10} \\ \hline \vdots \\ \hline A_{(N-1)0} \end{array} \right) \left(\begin{array}{c|c|c|c} B_{00} & B_{01} & \cdots & B_{0(N-1)} \end{array} \right) \tag{8}$$

$$+ \left(\begin{array}{c} 0 \\ \hline A_{11} \\ \hline \vdots \\ \hline A_{(N-1)1} \end{array} \right) \left(\begin{array}{c|c|c|c} B_{10} & B_{11} & \cdots & B_{1(N-1)} \end{array} \right) + \cdots \tag{9}$$

$$+ \left(\begin{array}{c} 0 \\ \hline \vdots \\ \hline 0 \\ \hline A_{(N-1)(N-1)} \end{array} \right) \left(\begin{array}{c|c|c|c} B_{(N-1)0} & B_{(N-1)1} & \cdots & B_{(N-1)(N-1)} \end{array} \right) \tag{10}$$

In other words, the matrix-matrix multiplication can be implemented as a sequence of N rank-n_b updates, where we can take advantage of the fact that some of matrix A is zero.

If B is to be overwritten, we need to be somewhat more careful. Notice that we can rearrange the computation so that

$$
\begin{aligned}
B &= AB = \left(\begin{array}{c} 0 \\ \hline \vdots \\ \hline 0 \\ \hline A_{(N-1)(N-1)} \end{array}\right) \left(\begin{array}{c|c|c|c} B_{(N-1)0} & B_{(N-1)1} & \cdots & B_{(N-1)(N-1)} \end{array} \right) && (11) \\
&+ \cdots + \left(\begin{array}{c} 0 \\ \hline A_{11} \\ \hline \vdots \\ \hline A_{(N-1)1} \end{array}\right) \left(\begin{array}{c|c|c|c} B_{10} & B_{11} & \cdots & B_{1(N-1)} \end{array} \right) && (12) \\
&+ \left(\begin{array}{c} A_{00} \\ \hline A_{10} \\ \hline \vdots \\ \hline A_{(N-1)0} \end{array}\right) \left(\begin{array}{c|c|c|c} B_{00} & B_{01} & \cdots & B_{0(N-1)} \end{array} \right) && (13)
\end{aligned}
$$

If the row panels of B are overwritten in reverse order, all entries of the original matrix will be available at the appropriate point in the algorithm.

3.2.2 Forming $B = AB$ (A upper triangular)

This operation can be easily derived from the general matrix-multiply and the lower triangular case.

$$
\begin{aligned}
B &= AB = \left(\begin{array}{c} A_{00} \\ \hline 0 \\ \hline \vdots \\ \hline 0 \end{array}\right) \left(\begin{array}{c|c|c|c} B_{00} & B_{01} & \cdots & B_{0(N-1)} \end{array} \right) && (14) \\
&+ \cdots + \left(\begin{array}{c} A_{0(N-2)} \\ \hline \vdots \\ \hline A_{(N-2)(N-2)} \\ \hline 0 \end{array}\right) \left(\begin{array}{c|c|c|c} B_{(N-2)0} & B_{(N-2)1} & \cdots & B_{(N-2)(N-1)} \end{array} \right) && (15) \\
&+ \left(\begin{array}{c} A_{0(N-1)} \\ \hline A_{1(N-1)} \\ \hline \vdots \\ \hline A_{(N-1)(N-1)} \end{array}\right) \left(\begin{array}{c|c|c|c} B_{(N-1)0} & B_{(N-1)1} & \cdots & B_{(N-1)(N-1)} \end{array} \right) && (16)
\end{aligned}
$$

This time, the order indicate guarantees that the data in B is overwritten correctly, yielding the correct result.

3.2.3 Forming $B = BA$, A upper or lower triangular

The case where the triangular matrix A multiplies B from the right can be handled similar to the case where it multiplies from the left. We leave this as an exercise for the reader.

3.2.4 Forming $B = A^T B$, A lower triangular

The operation $C = A^T B$, with A lower triangular can be implemented much like $C = A^T B$, where A is a general matrix:

$$\left(\begin{array}{c|c|c|c} C_{i0} & C_{i1} & \cdots & C_{i(N-1)} \end{array} \right) = \left(\begin{array}{c|c|c|c|c|c} A_{i0} & \cdots & A_{ii} & 0 & \cdots & 0 \end{array} \right)^T B$$

C can thus be computed a row panel at a time, where the computation does not perform calculations on the upper triangular part of A.

Since C must overwrite B, we perform this operation $i = N-1, \ldots, 0$, in that order. This will ensure that the row panels of B are overwritten in an order that does not change the final result.

3.2.5 Forming $B = A^T B$, A upper triangular

This case can be treated similarly.

3.2.6 Forming $B = BA^T$, A upper or lower triangular

Algorithms for these cases can be derived similarly.

3.3 Symmetric Matrix-Matrix Multiply (`xSYMM`)

The basic symmetric matrix-matrix multiply is defined by the operation $C \leftarrow AB$. where matrix A is symmetric, and only the upper or lower triangular part of A contains the necessary data. A can multiply B from the left or right.

Consider that case where A is stored in the lower triangular part of A only. Let $\tilde{A}$ and $\hat{A}$ equal the lower and strictly lower triangular parts of A, respectively. Since $A = \tilde{A} + \hat{A}^T$, we see that $C = \leftarrow \tilde{A}B + \hat{A}^T B$, and the parallel implementations can be easily derived from the implementations of the triangular matrix-matrix multiply. Indeed, the implementation is simplified by the fact that B is not being overwritten.

All cases of this operation can be derived similarly.

3.4 Symmetric Rank-K Update (`xSYRK`)

The basic symmetric matrix-matrix multiply is defined by the operation $C \leftarrow AA^T$, where matrix C is symmetric, and only the upper or lower triangular part of C needs to be computed.

If C is a general matrix, i.e. all of the matrix is to be updated, then this operation could be implemented imply, with two applications of the general matrix-matrix multiply, with matrix B equal to A and the "Transpose" parameter set for matrix B. Since, due to symmetry, only a triangular portion of C is to be updated, the implementation is exactly like this, except that unnecessary computation is not performed, and only the appropriate portion of matrix C is updated.

All cases of this operation can be derived similarly.

3.5 Symmetric Rank-2K Update (`xSYR2K`)

The basic symmetric matrix-matrix multiply is defined by the operation $C \leftarrow AB^T + BA^T$, where matrix C is symmetric, and only the upper or lower triangular part of C contains the necessary data.

As with the rank-k update, if C is a general matrix, i.e. all of the matrix is to be updated, then this operation could be implemented by simply two general matrix-matrix multiply, with the appropriate "Transpose" parameters set. Since, due to symmetry, only a triangular portion of C is to be updated, the implementation is exactly like this, except that unnecessary computation is not performed, and only the appropriate portion of matrix C is updated.

All cases of this operation can be derived similarly.

3.6 Triangular Solve with Multiple Right-Hand-Sides (`xTRSM`)

The final level-3 BLAS operation we discuss is the triangular solve with multiple right-hand-sides.

3.6.1 Forming $B = A^{-1}B$, A lower triangular

This operation can be reformulated as

$$AX = B$$

where A is lower triangular, and X overwrites B. Letting A, X, and B be partitioned as before, note that

$$\left(\begin{array}{c||c|c|c} A_{00} & 0 & \cdots & 0 \\ \hline\hline A_{10} & A_{11} & \cdots & 0 \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{(N-1)0} & A_{(N-1)1} & \cdots & A_{(N-1)(N-1)} \end{array}\right) \left(\begin{array}{c} X_{0*} \\ \hline\hline X_{1*} \\ \hline \vdots \\ \hline X_{(N-1)*} \end{array}\right) = \left(\begin{array}{c} B_{0*} \\ \hline\hline B_{1*} \\ \hline \vdots \\ \hline B_{(N-1)*} \end{array}\right)$$

where A_{ii} are lower triangular. From this, we derive the following equations:

$$A_{00} \left(\begin{array}{c|c|c|c} X_{00} & X_{01} & \cdots & X_{0(N-1)} \end{array}\right) = \left(\begin{array}{c|c|c|c} B_{00} & B_{01} & \cdots & B_{0(N-1)} \end{array}\right) \tag{17}$$

$$\left(\begin{array}{c} A_{10} \\ \hline \vdots \\ \hline A_{(N-1)0} \end{array}\right)^{(X_{0*})} + \left(\begin{array}{c|c|c} A_{11} & \cdots & 0 \\ \hline \vdots & & \vdots \\ \hline A_{(N-1)1} & \cdots & A_{(N-1)(N-1)} \end{array}\right) \left(\begin{array}{c} X_{1*} \\ \hline \vdots \\ \hline X_{(N-1)*} \end{array}\right) = \left(\begin{array}{c} B_{1*} \\ \hline \vdots \\ \hline B_{(N-1)*} \end{array}\right) \tag{18}$$

From (17) we note that X_{0*} can be computed by solving $A_{00}X_{0j} = B_{0j}$, $j = 0, \ldots, N-1$, overwriting B_{0j} with the results. After this, the right-hand-side can be overwritten by

$$\left(\begin{array}{c|c|c} B_{10} & \cdots & B_{1(N-1)} \\ \hline \vdots & & \vdots \\ \hline B_{(N-1)0} & \cdots & B_{(N-1)(N-1)} \end{array}\right) -= \left(\begin{array}{c} A_{10} \\ \hline \vdots \\ \hline A_{(N-1)0} \end{array}\right)^{\left(\begin{array}{c|c|c} X_{00} & \cdots & X_{0(N-1)} \end{array}\right)}$$

Here $-=$ indicates the right-hand-side of the equation is subtracted from the left-hand-side. Finally, the equation

$$\left(\begin{array}{c|c|c} A_{11} & \cdots & 0 \\ \hline \vdots & & \vdots \\ \hline A_{(N-1)1} & \cdots & A_{(N-1)(N-1)} \end{array}\right) \left(\begin{array}{c} X_{1*} \\ \hline \vdots \\ \hline X_{(N-1)*} \end{array}\right) = \left(\begin{array}{c} B_{1*} \\ \hline \vdots \\ \hline B_{(N-1)*} \end{array}\right)$$

is recursively solved.

It is important to note that this algorithm is very similar to the one derived for implementing the triangular matrix-matrix multiply in Section 3.2. Indeed the algorithm can be viewed as a sequence of rank-n_b updates, with the row panel first updated by the triangular solve with multiple right-hand-sides in Eqn. 17. Also the order of the updates is the reverse of the corresponding updates for the lower triangular matrix multiply.

The parallel implementation can now be derived in a straight forward manner: For each iteration, j, the current $n_b \times n_b$ triangular block, A_{jj} is broadcast within the row of nodes that owns it, and also owns B_{j*}. These nodes can then perform independent triangular solves with multiple right-hand-sides, overwriting B_{j*}. Finally, A_{ij} and B_{ji}, $i = j+1, \ldots, N-1$, are broadcast within rows and columns of nodes, respectively, after which local rank-n_b updates are used to update the appropriate portions of B.

3.6.2 Forming $B = A^{-1}B$, A upper triangular

This case can be treated similarly.

3.6.3 Forming $B = BA^{-1}$, A upper and lower triangular

These cases follow similarly.

3.6.4 Forming $B = A^{-T}B$, A lower triangular

This operation can be reformulated as

$$A^T X = B$$

where A is lower triangular, and X overwrites B. Letting A, X, and B be partitioned as before, note that

$$\left(\begin{array}{c|c||c||c|c|c} A_{00} & \cdots & 0 & 0 & \cdots & 0 \\ \hline \vdots & \ddots & \vdots & \vdots & & \vdots \\ \hline A_{i0} & \cdots & A_{ii} & 0 & \cdots & 0 \\ \hline \vdots & & \vdots & \vdots & & \vdots \\ \hline A_{(N-1)0} & \cdots & \cdots & \cdots & \cdots & A_{(N-1)(N-1)} \end{array}\right)^T \left(\begin{array}{c} X_{0*} \\ \hline \vdots \\ \hline\hline X_{i*} \\ \hline\hline \vdots \\ \hline X_{(N-1)*} \end{array}\right) = \left(\begin{array}{c} B_{0*} \\ \hline \vdots \\ \hline\hline B_{i*} \\ \hline\hline \vdots \\ \hline B_{(N-1)*} \end{array}\right)$$

is equivalent to

$$\left(\begin{array}{c|c|c|c|c|c} A_{00}^T & \cdots & \cdots & A_{i0}^T & \cdots & A_{(N-1)0}^T \\ \hline 0 & & & \vdots & & \vdots \\ \hline\hline 0 & \cdots & 0 & A_{ii}^T & \cdots & A_{(N-1)i}^T \\ \hline\hline \vdots & & \vdots & \vdots & & \vdots \\ \hline 0 & \cdots & 0 & 0 & \cdots & A_{(N-1)(N-1)}^T \end{array}\right) \left(\begin{array}{c} X_{0*} \\ \hline \vdots \\ \hline\hline X_{i*} \\ \hline\hline \vdots \\ \hline X_{(N-1)*} \end{array}\right) = \left(\begin{array}{c} B_{0*} \\ \hline\hline \vdots \\ \hline\hline B_{i*} \\ \hline\hline \vdots \\ \hline B_{(N-1)*} \end{array}\right)$$

From this, we derive the following equation:

$$A_{ii}^T X_{i0} + \left(\begin{array}{c} 0 \\ \hline \vdots \\ \hline 0 \\ \hline A_{(i+1)i} \\ \hline \vdots \\ \hline A_{(N-1)i} \end{array}\right)^T \left(\begin{array}{c} X_{0*} \\ \hline \vdots \\ \hline X_{i*} \\ \hline X_{(i+1)*} \\ \hline \vdots \\ \hline X_{(N-1)*} \end{array}\right) = B_{i*}$$

Taking advantage of zeroes, the following two steps will compute X_{i*}:

$$B_{i*} = B_{i*} - \left(\begin{array}{c} A_{(i+1)i} \\ \hline \vdots \\ \hline A_{(N-1)i} \end{array}\right)^T \left(\begin{array}{c} X_{(i+1)*} \\ \hline \vdots \\ \hline X_{(N-1)*} \end{array}\right) \tag{19}$$

followed by solving

$$A_{ii}^T X_{i*} = B_{i*} \tag{20}$$

Notice that X_{i*} must be computed in the order $i = N-1, \ldots, 0$, which also allows B to be overwritten by X as the computation proceeds.

Again, observe the similarity with the operation $B = A^T B$, where A is lower triangular. Exploiting this similarity, we obtain the following parallel algorithm: At step i, nonzero blocks A_{ji} are broadcast within rows. After computing local contributions of the update in Eqn. 19, the results are summed within columns to the row of nodes that hold the ith panel of B. Finally, local triangular solves with multiple right-hand-sides to update according to Eqn. 20 are performed, and i is decreased.

3.6.5 Forming $B = A^{-T} B$, A upper triangular

This case can be treated similarly.

3.6.6 Forming $B = BA^{-T}$, A upper and lower triangular

These cases follow similarly.

4 Analysis

In [41], we give an analysis of the matrix-matrix multiplication algorithm. In that paper, we assumed a slightly different matrix distribution, and hence the results were slightly different from those we will derive next. In addition, we show the results of similar analysis of representative other parallel level 3 BLAS.

For our analysis, we will make the following assumptions:

1. The nodes for a physical $r \times c$ mesh, with P_{ij} assigned to the (i, j) physical node in that mesh.
2. All alignment parameters equal one.
3. $n_r = n_c = n_b$.
4. A minimum spanning tree broadcast and summation-to-one are used, so that the cost for broadcasting a vector of length n among p nodes is given by
$$\lceil \log(p) \rceil (\alpha + n\beta)$$
and the cost for summing vectors of length n among p nodes is given by
$$\lceil \log(p) \rceil (\alpha + n\beta + n\gamma)$$
There are a number of papers on how to improve upon this method of broadcasting and summation [28, 38, 9, 40, 42], but will analyze only this simple approach, since it is typically the current MPI default implementation.
5. Dimensions m, n, and k are integer multiples of n_b: $m = Mn_b$, $n = Nn_b$, and $k = Kn_b$.

4.1 Forming $C = AB$

The estimated cost for forming $C = AB$ is given by

$$\begin{array}{l} T_{C=AB}(m, n, k, r, c) = \\ \quad K \left[\lceil \log(c) \rceil \left(\alpha + \left\lceil \frac{M}{r} \right\rceil n_b^2 \beta \right) + \lceil \log(r) \rceil \left(\alpha + \left\lceil \frac{N}{c} \right\rceil n_b^2 \beta \right) + 2 \left\lceil \frac{M}{r} \right\rceil \left\lceil \frac{N}{c} \right\rceil n_b^3 \gamma \right] \end{array}$$

The factor K comes from the number of iterations performed. Within the square brackets, the first and second term are due to the column and row panel broadcasts, respectively. The last term is due to the local matrix-matrix multiply. If we ignore all the ceilings, we get an estimate of

$$\begin{array}{cl} & T_{C=AB}(m, n, k, r, c) \\ \approx & \frac{k}{n_b} \left[\log(c) \left(\alpha + \frac{m}{r} n_b \beta \right) + \log(r) \left(\alpha + \frac{n}{c} n_b \beta \right) + 2 \frac{m}{r} \frac{n}{c} n_b \gamma \right] \\ = & \frac{k}{n_b} \log(p) \alpha + \left(\log(c) \frac{mk}{r} + \log(r) \frac{nk}{c} \right) \beta + 2 \frac{mnk}{p} \gamma \end{array}$$

If we define the *speedup* as

$$S_{C=AB}(m, n, k, r, c) = \frac{T^{\text{seq}}_{C=AB}(m, n, k)}{T_{C=AB}(m, n, k, r, c)}$$

and the *efficiency* as

$$E_{C=AB}(m, n, k, r, c) = \frac{S_{C=AB}(m, n, k, r, c)}{p}$$

then we see that

$$\begin{array}{rcl} E_{C=AB}(m, n, k, r, c) & \approx & \dfrac{1}{1 + \frac{pk}{n_b 2mnk} \log(p) \frac{\alpha}{\gamma} + \left(\log(c) \frac{pmk}{2mnkr} + \log(r) \frac{pnk}{2mnkc} \right) \frac{\beta}{\gamma}} \\ & = & \dfrac{1}{1 + \frac{p \log(p)}{n_b 2mn} \frac{\alpha}{\gamma} + \left(\log(c) \frac{c}{2n} + \log(r) \frac{r}{2m} \right) \frac{\beta}{\gamma}} \end{array}$$

We will now study the special case where $m = n = k$ and $r = c = \sqrt{p}$, in which case the efficiency becomes

$$E_{C=AB}(n, n, n, \sqrt{p}, \sqrt{p}) \;=\; \frac{1}{1 + \frac{p\log(p)}{n_b 2n^2}\frac{\alpha}{\gamma} + \log(p)\frac{\sqrt{p}}{2n}\frac{\beta}{\gamma}}$$

Given this estimate of the efficiency, we will ask ourselves the question what will happen to the efficiency if we allow the problem to grow as nodes are added, with the constraint that the memory utilization *per node* must remain constant. Since memory usage is proportional to n^2, this means we require $n^2 = Cp$ for some constant C, or $n = \sqrt{C}\sqrt{p}$. Notice that

$$\begin{aligned} E_{C=AB}(\sqrt{C}\sqrt{p}, \sqrt{C}\sqrt{p}, \sqrt{C}\sqrt{p}, \sqrt{p}, \sqrt{p}) &= \frac{1}{1 + \frac{p\log(p)}{n_b 2Cp}\frac{\alpha}{\gamma} + \log(p)\frac{\sqrt{p}}{2\sqrt{C}\sqrt{p}}\frac{\beta}{\gamma}} \\ &= \frac{1}{1 + \frac{\log(p)}{n_b 2C}\frac{\alpha}{\gamma} + \log(p)\frac{1}{2\sqrt{C}}\frac{\beta}{\gamma}} \end{aligned}$$

We thus conclude that *if* $\log p$ is treated as a constant, then the efficiency should remain constant if the problem size and number of nodes grow as indicated.

Notice that in our analysis, we have ignored the cost of copying between buffers, as may be necessary to pack or unpack noncontiguous arrays.

4.2 Other operations

Rather than performing the above analysis for each of the indicated operations, we now give a table of some of the time estimates derived similarly. We will make the same assumptions as made for $C = AB$, and again will ignore ceiling functions in this table. Also, we only report the most significant term for each of the α, β, and γ components.

Algorithm	Cost
xGEMM ($m \times n$ matrix C, $n \times k$ matrix A)	
$T^{\mathtt{GEMM}}_{C=AB} \approx T_{C=AB^T} \approx T_{C=A^TB} \approx$	$\log(p)\frac{k}{n_b}\alpha + \left(\log(c)\frac{mk}{r} + \log(r)\frac{nk}{c}\right)\beta + 2\frac{mnk}{p}\gamma$
xTRMM ($m \times n$ matrix B)	
$T^{\mathtt{TRMM}}_{B=AB} \approx T^{\mathtt{TRMM}}_{B=A^TB} \approx$	$\log(p)\frac{m}{n_b}\alpha + \left(\log(c)\frac{m^2}{2r} + \log(r)\frac{mn}{c}\right)\beta + \frac{m^2n}{p}\gamma$
xSYMM($m \times n$ matrix C)	
$T^{\mathtt{SYMM}}_{C=AB} \approx$	$T^{\mathtt{TRMM}}_{C=AB} + TR^{\mathtt{TRMM}}_{C=A^TB}$
xSYRK ($m \times n$ matrix A)	
$T^{\mathtt{SYRK}}_{C=AA^T} \approx$	$\frac{n}{n_b}\log(p)\alpha + \left(\log(c)\frac{mn}{2r} + \log(r)\frac{mn}{c}\right)\beta + \frac{mn^2}{p}\gamma$
$T^{\mathtt{SYRK}}_{C=A^TA} \approx$	$\frac{m}{n_b}\log(p)\alpha + \left(\log(c)\frac{mn}{2r} + \log(r)\frac{mn}{c}\right)\beta + \frac{m^2n}{p}\gamma$
xSYR2K ($m \times n$ matrices A and B)	
$T^{\mathtt{SYR2K}}_{C=AB^T+B^TA} \approx$	$T^{\mathtt{SYRK}}_{C=AA^T} + T^{\mathtt{SYRK}}_{C=A^TA}$
xTRSM ($m \times n$ matrix B)	
$T^{\mathtt{TRSM}}_{B=A^{-1}B} \approx$	$(\log(p) + \log(c))\frac{m}{n_b}\alpha + \left(\log(c)\frac{m^2}{2r} + \log(r)\frac{mn}{c}\right)\beta + \frac{nm^2}{p}\gamma$

The presented costs are representative of the costs of all parallel level 3 BLAS. Scalability can be analysed using these costs extimates as was done for matrix-matrix multiply. The resulting conclusions are similar to those for the matrix-matrix multiply.

5 Optimizations

In [41], we show how the performance of the matrix-matrix multiplication can be improved considerably by pipelining communication and computation. In effect, the broadcast is replaced by one that passes the messages around the ring that forms the nodes involved in the broadcast. Let us examine the simple case $C = AB$. Since the next iteration of an algorithm can start as soon as the next nodes that need to broadcast are ready, the net cost of the algorithm becomes

$$\begin{aligned} T_{C=AB}(m,n,k,r,c) &= \\ & K\left[2\left(\alpha + \left\lceil \frac{M}{r} \right\rceil n_b^2 \beta\right) + 2\left(\alpha + \left\lceil \frac{N}{c} \right\rceil n_b^2 \beta\right) + 2\left\lceil \frac{M}{r} \right\rceil \left\lceil \frac{N}{c} \right\rceil n_b^3 \gamma\right] \\ \approx & \ \frac{k}{n_b} 2\alpha + \left(2\frac{mk}{r} + 2\frac{nk}{c}\right)\beta + 2\frac{mnk}{p}\gamma \end{aligned}$$

This formula ignores the cost of filling and/or emptying the pipe. A similar improvement is seen for all the algorithms, if this "ring" broadcast is used. For some of the algorithms, some care must be taken to make the operation "flow" in the appropriate direction. Notice that the effect on scalability is considerable, since in the estimate for the efficiency, the "log" factors are essentially replaced by a 2, making the approach virtually perfectly scalable.

6 Performance Results

This section reports the performance of preliminary parallel level 3 BLAS implementations, obtained on an Intel Paragon with GP nodes. Our implementations use the highly optimized BLAS library available for computation on each node. Communication is accomplished using MPI collective communication calls. We present results from pipelined implementations.

In Fig. 4, we show the performance of representative parallel level 3 BLAS as a function of the matrix dimensions, on a single node. This gives us an indication of the peak performance for the different operations.

In Fig. 5, we show the performance of representative parallel level 3 BLAS as a function of the matrix dimensions, on an 8×8 mesh of nodes. All matrices are assumed to be square. Performance is reported in MFLOPS per node, as a function of problem size. Noting that peak performance for matrix-matrix multiply on a single node is around 45-46 MFLOPS, we see our implementations achieve very good performance, once the problem size is reasonably large. Additional performance graphs are given in the appendix.

In Fig. ??, we show the performance of representative parallel level 3 BLAS as a function of the number of nodes, where the problem size is scaled with the number of nodes to keep the memory used per node constant. Again, all matrices are assumed to be square and performance is reported in MFLOPS per node. Since reporting MFLOPS (performance) per node is equivalent to reporting efficiency, our analysis predicts that performance per node should be essentially constant as the number of nodes increases. Our performance graphs show a trend that verifies that prediction. Additional scalability graphs are given in the appendix.

Notice that we do not give performance results for $C = A^T B^T$ since we don't have a general implementation for that operation.

7 Availability

A considerable contribution of this work is in the implementations that we have produced. Complete implementations of the discussed BLAS, with the exception of $C = A^T B^T$, are available under GNU license from

`http://www.cs.utexas.edu/users/rvdg/sw/sB_BLAS`

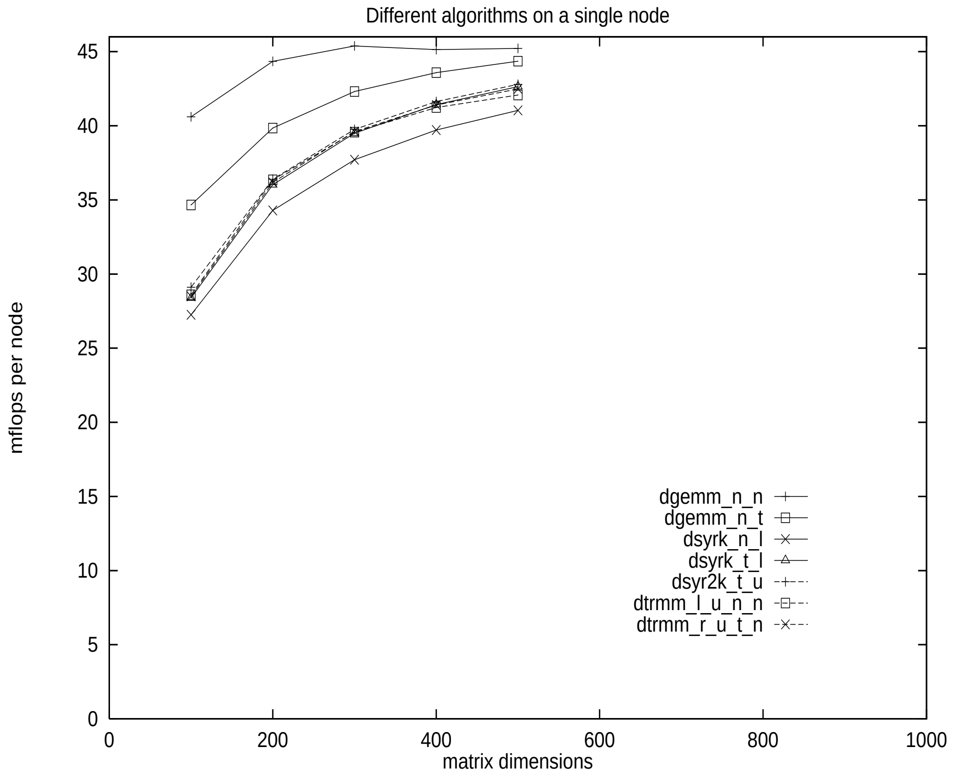

Figure 4: Performance of the various algorithms (on a single node), as a function of the global matrix dimension. The "_x_x_x_x" denoted the different options passed to the BLAS routine. E.g. _r_u_t_n stands for "Right", "Upper triangular", "Transpose", "No transpose"

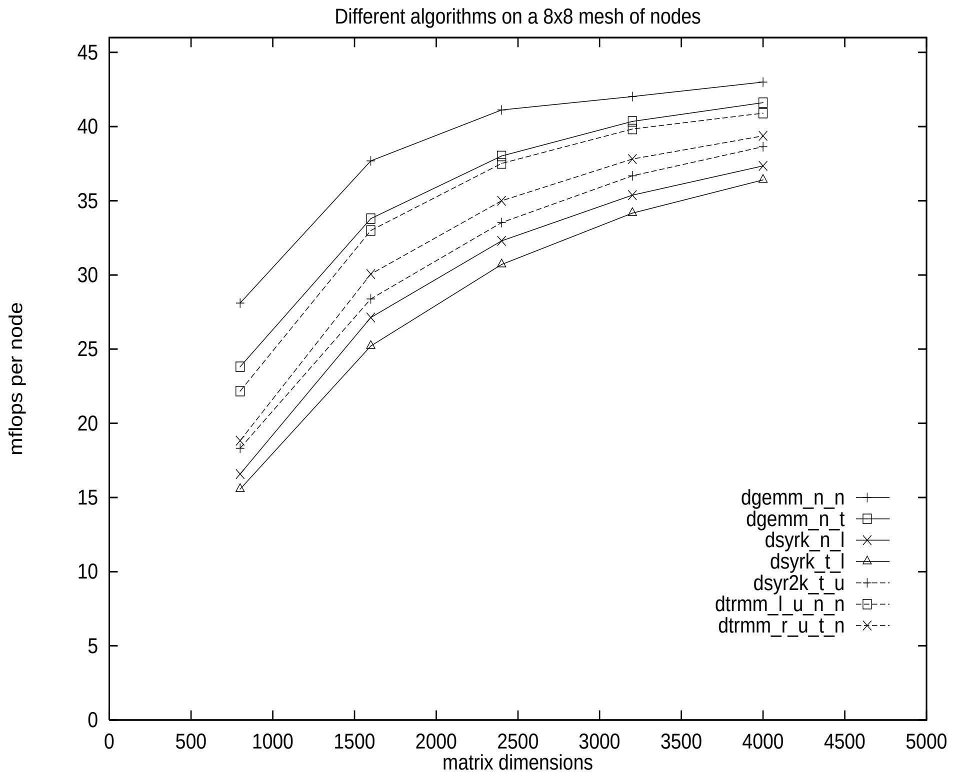

Figure 5: Performance of the various algorithms (on an 8×8 mesh), as a function of the global matrix dimension.

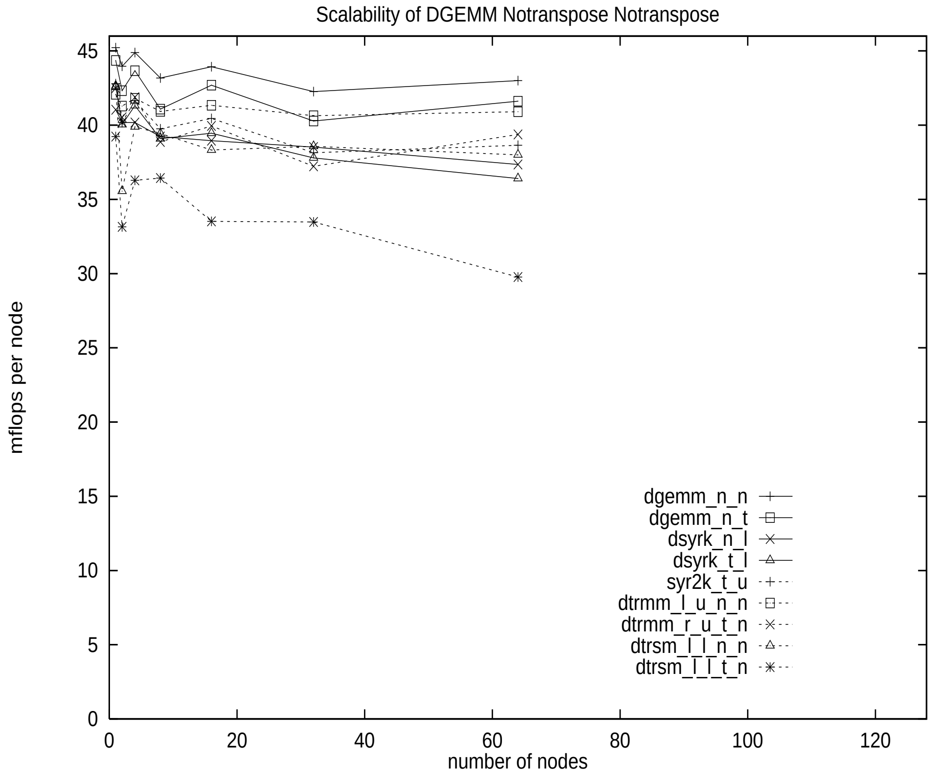

Figure 6: Scalability of the various algorithms when memory utilization per node is equivalent to a 500×500 matrix.

8 Conclusion

We have demonstrated a general, unified approached to parallel implemention of the level 3 BLAS. Very good performance is demonstrated. Furthermore, a considerable strength of the approach is that it allows simple implementation of all algorithms, making it ideal for library development.

It appears that one shortcoming of our approach is that is does not handle the operation $C = A^T B^T$ elegantly. Moreover, while our graphs show that performance is impressive, we must point out that this is not necessarily the case when the matrices are not nicely shaped. Consider the operation $C = AB$, where C and B are only single columns or narrow panels of columns. All useful computation will be performed by the single column of nodes that own C and B. Interestingly enough, if we start by transposing B, resulting $\tilde{B}$ distributed in a single row of nodes, and next apply the algorithm $C = AB^T$, useful computation is performed by all nodes.

We thus argue that the key to implementing all level 3 BLAS so that they are efficient for all shapes of matrices requires us to be able to transpose panels of rows and/or columns efficiently. This would also solve the $C = A^T B^T$ problem. Unfortunately, the traditional approach to matrix distribution does not allow this operation to be performed efficiently, elegantly, or generally. We believe that the key to solving this problem is to switch to what we call *Physically Based Matrix Distributions* (PBMD) [22]. We are currently pursuing implementation of all BLAS using this new approach to matrix distribution.

The above comments may make it appear we are sending mixed signals concerning the usefulness of the described techniques. We believe the presented paper is useful in that it exposes powerful techniques regardless of the matrix distribution choosen. We have tried to describe the techniques so that they can be easily customized for specific distributions. The library as it stands would be useful for implementation of a wide range of algorithms, including the LU factorization [21], the left-looking Cholesky factorization, matrix-multiplication based eigensolvers[6, 7, 8, 33], or out-of-core dense linear solvers [31].

Finally, in [26] we show how this approach can be used for parallel implementation of Strassen's algorithm for matrix-matrix multiplication. This suggests that an entire suite of variants of Strassen's algorithm for all the level 3 BLAS could be implemented on top of our standard parallel level 3 BLAS implementations.

Acknowledgements

This research was performed in part using the Intel Paragon 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] R. C. Agarwal, F. G. Gustavson, S. M. Balle, M. Joshi, P. Palkar, "A High Performance Matrix Multiplication Algorithm for MPPs" IBM T.J. Watson Research Center, 1995.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair, "A High Performance Matrix Multiplication Algorithm on a Distributed-Memory Parallel Computer, using Overlapped Communication," *IBM Journal of Research and Development,* pp. 673–681, 1994.

[3] Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbau m, 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.

[4] 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.

[5] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn, "LAPACK for Distributed Memory Architectures: Progress Report." in the *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing,* SIAM, Philadelphia, 1992. pp. 625–630.

[6] Auslander, L., Tsao, A., *On Parallelizable Eigensolvers*, Advanced Appl. Math., Vol. 13, pp. 253–261, 1992

[7] Bai, Z., Demmel, J., "Design of a Parallel Nonsymmetric Eigenroutine Toolbox, Part I", *Parallel Processing for Scientific Computing*, Editors R. Sincovec, D. Keyes, M. Leuze, L. Petzold, and D. Reed, pp. 391–398, SIAM Publications, Philadelphia, PA, 1993

[8] Bai, Z., Demmel, J., Dongarra, J., Petitet, A., Robinson, H., Stanley, K., *The Spectral Decomposition of Nonsymmetric Matrices on Distributed Memory Parallel Computers*, LAPACK working note 91, University of Tennessee, Jan. 1995

[9] M. Barnett, D.G. Payne, R. van de Geijn and J. Watts. "Broadcasting on Meshes with Worm-Hole Routing," University of Texas, Department of Computer Sciences, TR-93-24 (1993).

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

[11] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley "A Proposal for a Set of Parallel Basic Linear Algebra Subprograms", LAPACK Working Note 100, University of Tennessee, CS-95-292, May 1995.

[12] 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.

[13] 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.

[14] 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.

[15] James Demmel, Jack Dongarra, Robert van de Geijn and David Walker, "LAPACK for Distributed Memory Architectures: The Next Generation," in *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing,* Norfolk, March 1993.

[16] 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.

[17] Dongarra, J. J., J. Du Croz, S. Hammarling, and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms", *TOMS*, Vol. 14, No. 1, pp. 1–17, 1988.

[18] 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.

[19] Jack J. Dongarra, Robert A. van de Geijn and R. Clint Whaley, "Two Dimensional Basic Linear Algebra Communication Subprograms," in *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, March 1993.

[20] Jack Dongarra and Robert van de Geijn, "A Parallel Dense Linear Solve Library Routine," in *Proceedings of the 1992 Intel Supercomputer Users' Group Meeting*, Dallas, Oct. 1992.

[21] Jack Dongarra, Robert van de Geijn, and David Walker, "Scalability Issues Affecting the Design of a Dense Linear Algebra Library," Special Issue on Scalability of Parallel Algorithms, *Journal of Parallel and Distributed Computing,* Vol. 22, No. 3, Sept. 1994.

[22] C. Edwards, P. Geng, A. Patra, and R. van de Geijn, "Parallel Matrix Distributions: have we been doing it all wrong?" Department of Computer Sciences, UT-Austin, Report TR95-39, Oct. 1995.

[23] 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.

[24] Fox, G., S. Otto, and A. Hey, "Matrix Algorithms on a Hypercube I: Matrix Multiplication," Parallel Computing **3** (1987), pp 17-31.

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

[26] Brian Grayson and Robert van de Geijn "A High Performance Parallel Strassen Implementation," *Parallel Processing Letters,* to appear.

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

[28] 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*, pp. 640–648, IEEE, 1986.

[29] 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.

[30] 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.

[31] Ken Klimkowski and Robert van de Geijn, "Anatomy of an out-of-core dense linear solver", *International Conference on Parallel Processing 1995,* to appear.

[32] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage", *TOMS*, Vol. 5, No. 3, pp. 308–323, 1979.

[33] Lederman, S., Tsao, A., Turnbull, T., *A parallelizable eigensolver for real diagonalizable matrices with real eigenvalues*, Technical Report TR-91-042, Supercomputing Research Center, 1991

[34] J. G. Lewis and R. A. van de Geijn, "Implementing Matrix-Vector Multiplication and Conjugate Gradient Algorithms on Distributed Memory Multicomputers," *Supercomputing '93.*

[35] J. G. Lewis, D. G. Payne, and R. A. van de Geijn, "Matrix-Vector Multiplication and Conjugate Gradient Algorithms on Distributed Memory Computers," *Scalable High Performance Computing Conference 1994.*

[36] W. Lichtenstein and S. L. Johnsson, "Block-Cyclic Dense Linear Algebra", Harvard University, Center for Research in Computing Technology, TR-04-92, Jan. 1992.

[37] 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.

[38] Prasenjit Mitra, David Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts, "Fast Collective Communication Libraries, Please," in the *Proceedings of the Intel Supercomputing Users' Group Meeting 1995.*

[39] Robert van de Geijn, "Massively Parallel LINPACK Benchmark on the Intel Touchstone DELTA and iPSC/860 Systems: Preliminary Report," TR-91-28, Department of Computer Sciences, University of Texas, Aug. 1991.

[40] Robert van de Geijn, "On Global Combine Operations," *Journal of Parallel and Distributed Computing,* **22**, pp. 324-328 (1994).

[41] Robert van de Geijn and Jerrell Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," TR-95-13, Department of Computer Sciences, University of Texas, April 1995. Also: LAPACK Working Note 96, May 1. Submitted to *Concurrency: Practice and Experience.*

[42] Jerrell Watts and Robert van de Geijn, "A Pipelined Broadcast for Multidimensional Meshes," *Parallel Processing Letters*, to appear.

In Figs. 7–11, we show the performance of all the level 3 BLAS as a function of the matrix dimensions. All matrices are assumed to be square. Performance is reported in MFLOPS per node, as a function of problem size. Noting that peak performance for matrix-matrix multiply on a single node is approximately 45-46 MFLOPS, we observe that our implementations achieve very good performance, once the problem size is reasonably large.

In Figs. 12–16, we show the performance of all the level 3 BLAS as a function of the number of nodes, where the problem size is scaled with the number of nodes to keep the memory used per node constant. Again, all matrices are assumed to be square and performance is reported in MFLOPS per node. Since reporting MFLOPS (performance) per node is equivalent to reporting efficiency, our analysis predicts that performance per node should be essentially constant as the number of nodes increases. Our performance graphs show a trend that verifies that prediction.

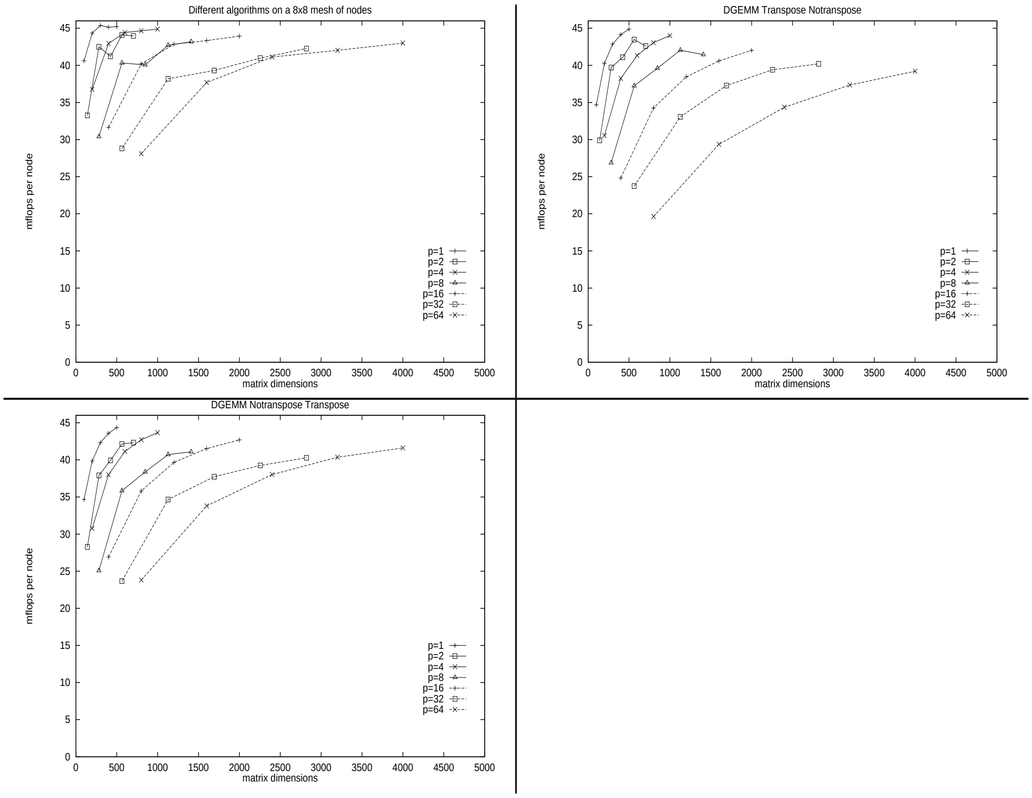

Figure 7: Performance of parallel DGEMM

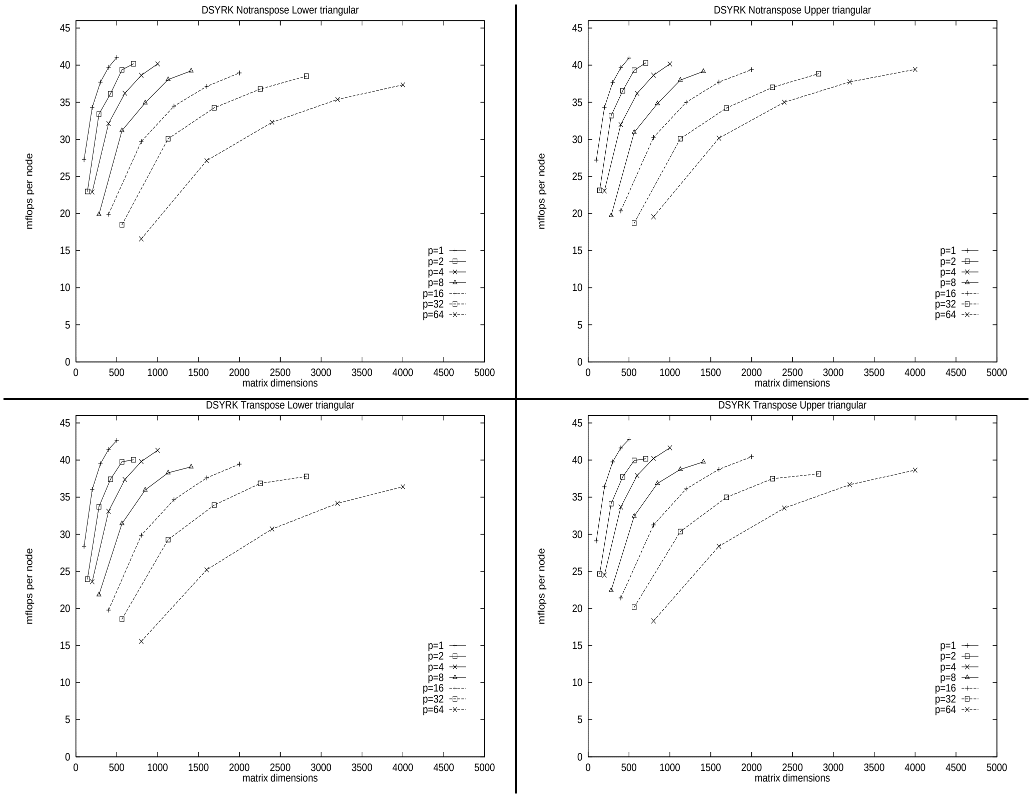

Figure 8: Performance of parallel DSYRK

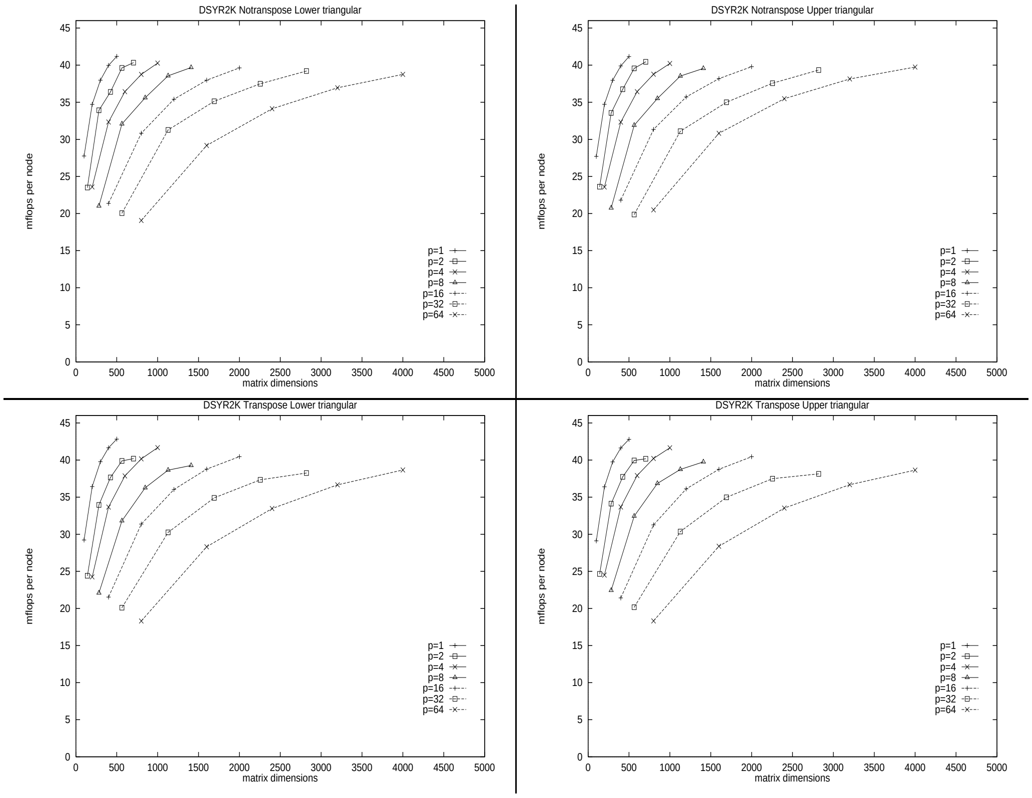

Figure 9: Performance of parallel DSYR2K

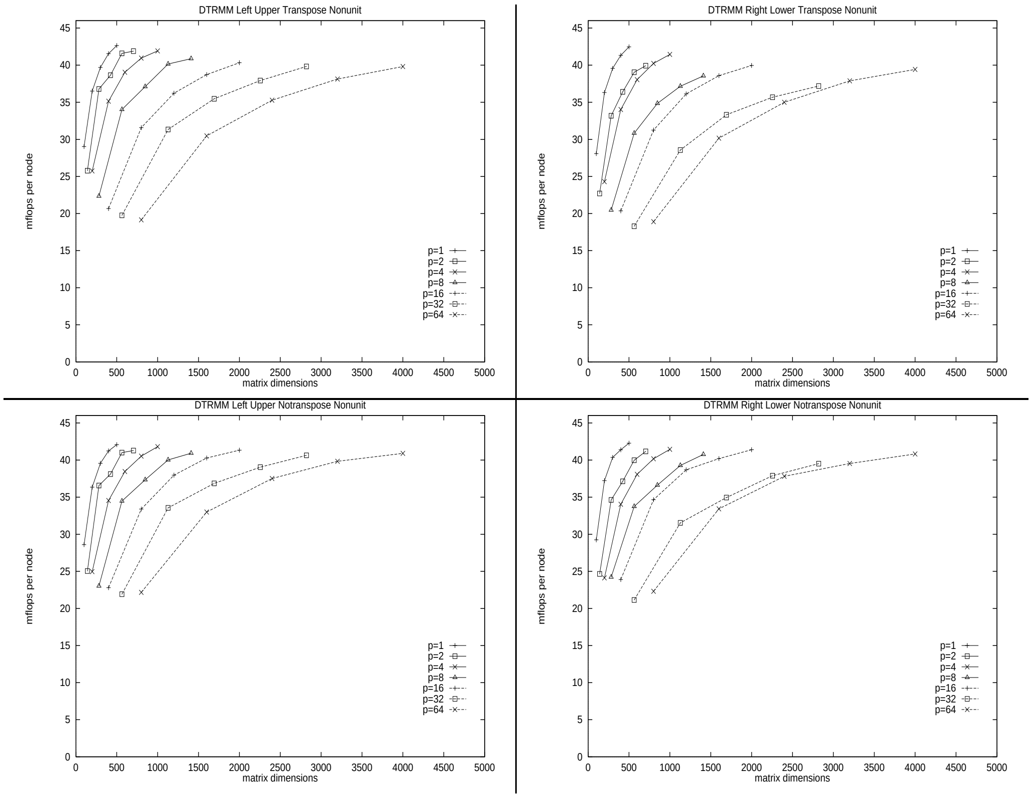

Figure 10: Performance of parallel DTRMM

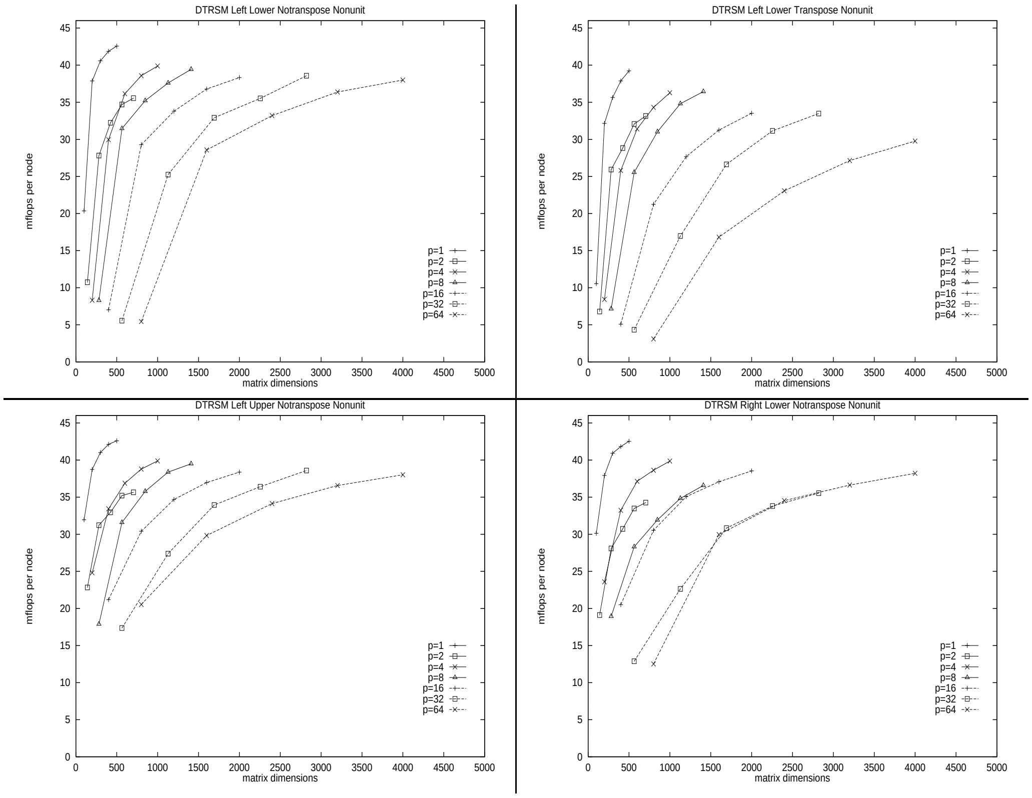

Figure 11: Performance of parallel DTRSM

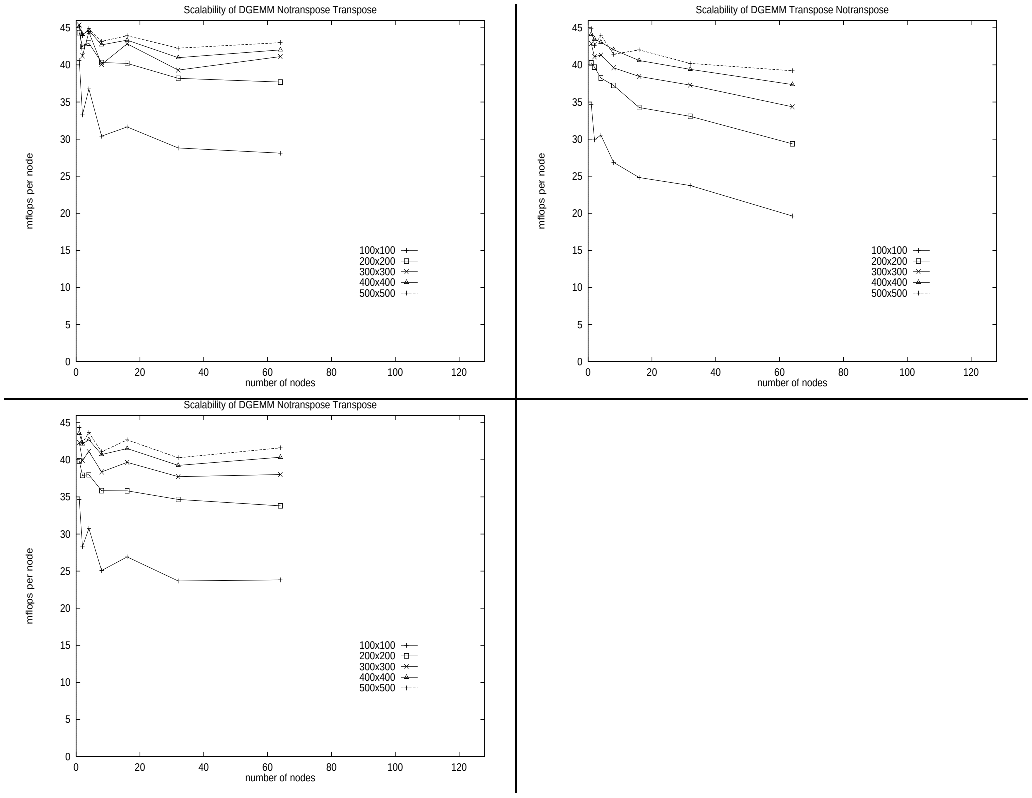

Figure 12: Scalability of parallel DGEMM

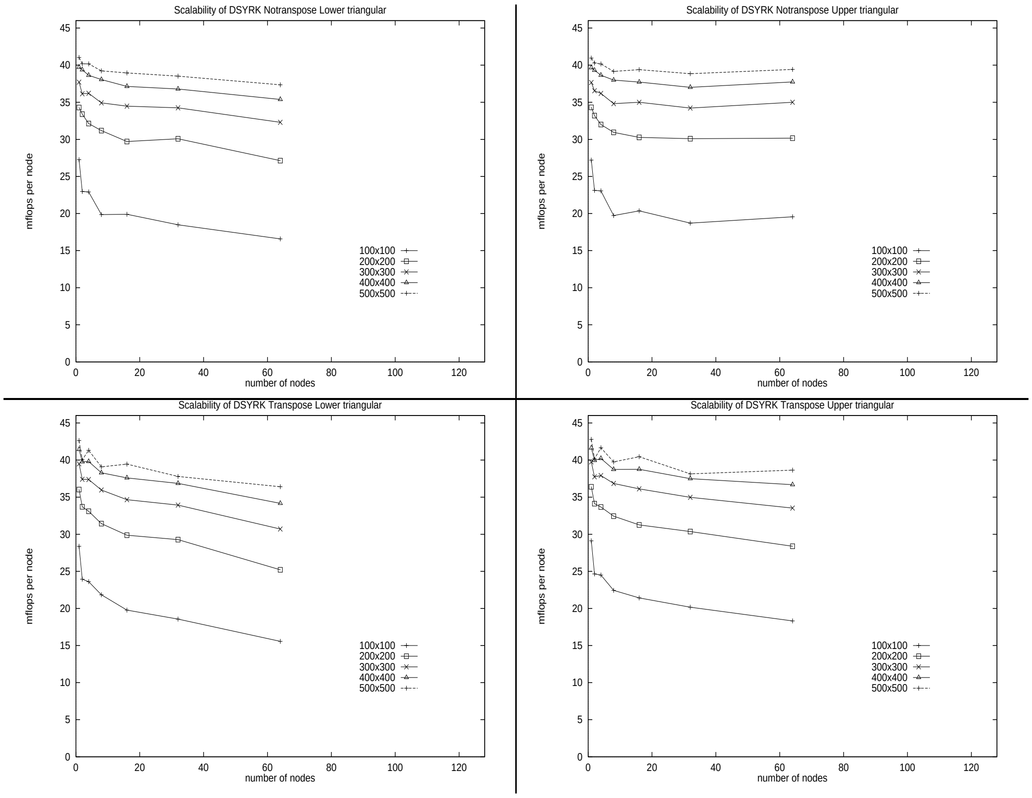

Figure 13: Scalability of parallel DSYRK

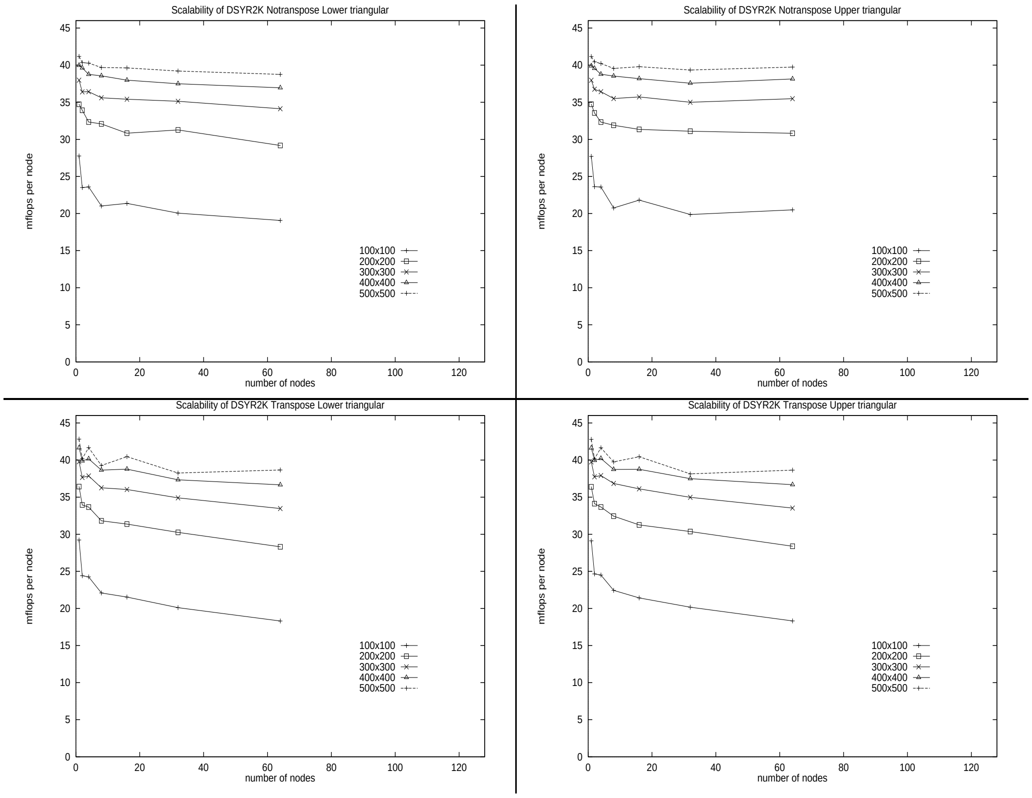

Figure 14: Scalability of parallel DSYR2K

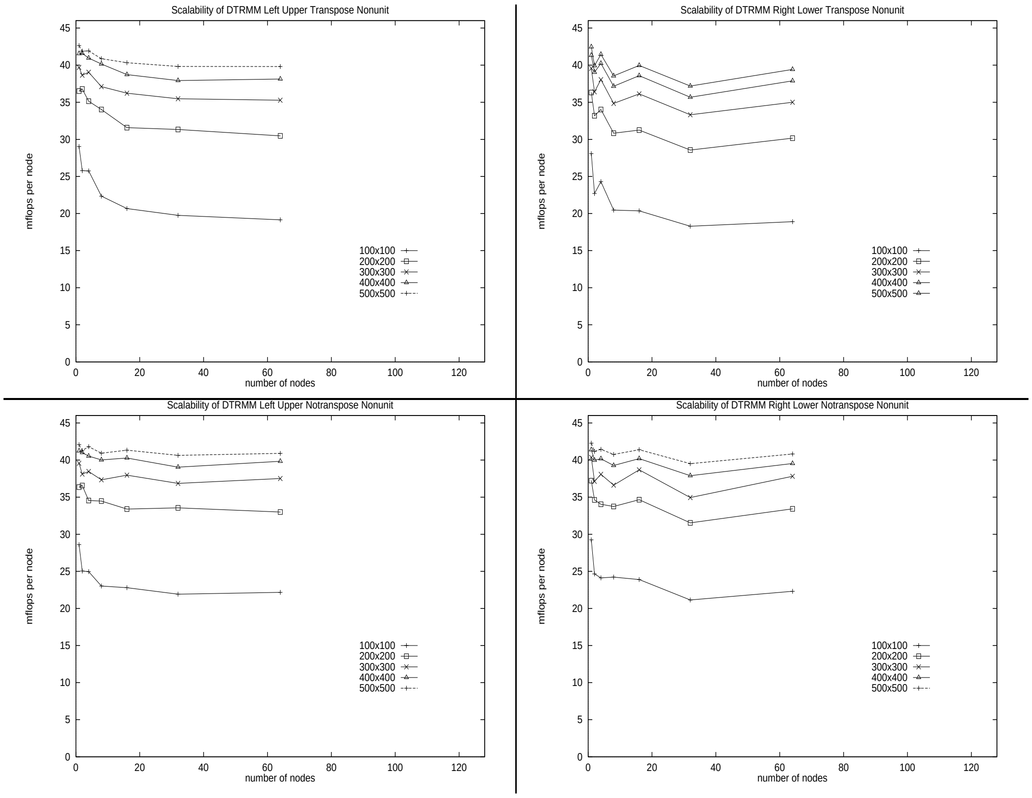

Figure 15: Scalability of parallel DTRMM

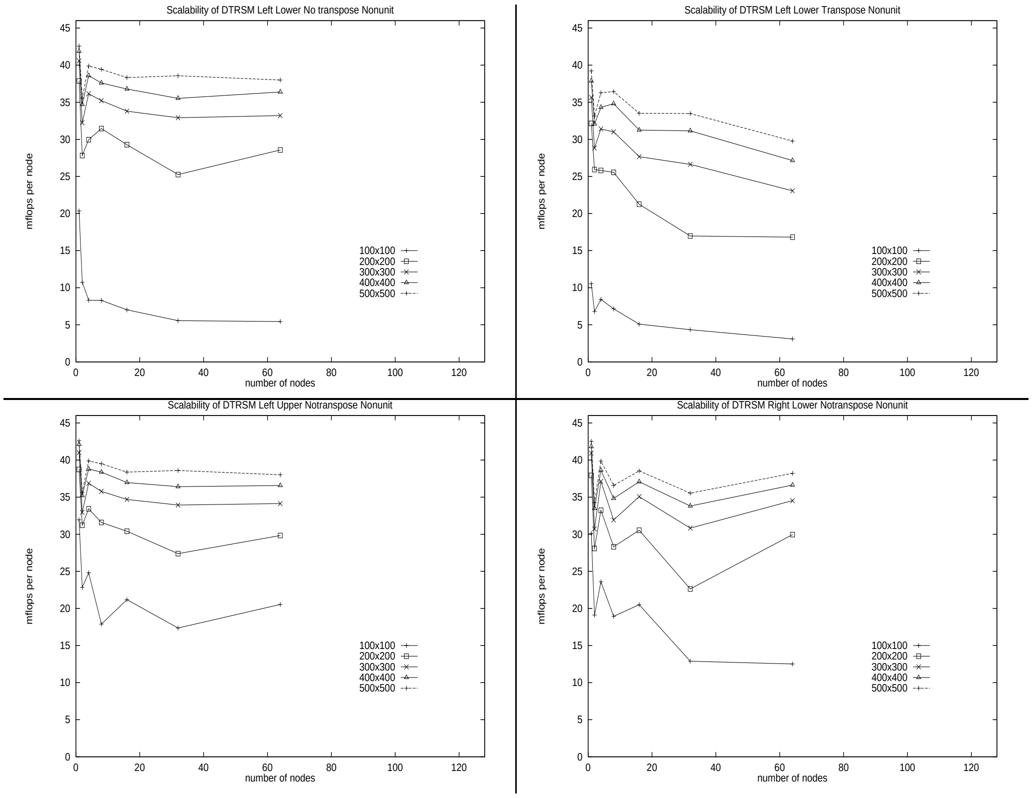

Figure 16: Scalability of parallel DTRSM