PARALLEL MATRIX MULTIPLICATION: A SYSTEMATIC JOURNEY

MARTIN D. SCHATZ[†], ROBERT A. VAN DE GEIJN[†], AND JACK POULSON[§]

Abstract. We expose a systematic approach for developing distributed memory parallel matrix-matrix multiplication algorithms. The journey starts with a description of how matrices are distributed to meshes of nodes (e.g., MPI processes), relates these distributions to scalable parallel implementation of matrix-vector multiplication and rank-1 update, continues on to reveal a family of matrix-matrix multiplication algorithms that view the nodes as a two-dimensional mesh, and finishes with extending these 2D algorithms to so-called 3D algorithms that view the nodes as a three-dimensional mesh. A cost analysis shows that the 3D algorithms can attain the (order of magnitude) lower bound for the cost of communication. The paper introduces a taxonomy for the resulting family of algorithms and explains how all algorithms have merit depending on parameters like the sizes of the matrices and architecture parameters. The techniques described in this paper are at the heart of the Elemental distributed memory linear algebra library. Performance results from implementation within and with this library are given on a representative distributed memory architecture, the IBM Blue Gene/P supercomputer.

1. Introduction.

This paper serves a number of purposes:

- Parallel[*] implementation of matrix-matrix multiplication is a standard topic in a course on parallel high-performance computing. However, rarely is the student exposed to the algorithms that are used in practical cutting-edge parallel dense linear algebra (DLA) libraries. This paper exposes a systematic path that leads from parallel algorithms for matrix-vector multiplication and rank-1 update to a practical, scalable family of parallel algorithms for matrix-matrix multiplication, including the classic result in [1] and those implemented in the Elemental parallel DLA library [28].
- This paper introduces a set notation for describing the data distributions that underlie the Elemental library. The notation is motivated using parallelization of matrix-vector operations and matrix-matrix multiplication as the driving examples.
- Recently, research on parallel matrix-matrix multiplication algorithms have revisited so-called 3D algorithms, which view (processing) nodes as a logical three-dimensional mesh. These algorithms are known to attain theoretical (order of magnitude) lower bounds on communication. This paper exposes a systematic path from algorithms for two-dimensional meshes to their extensions for three-dimensional meshes. Among the resulting algorithms are classic results [2].
- A taxonomy is given for the resulting family of algorithms which are all related to what is often called the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [33].

Thus, the paper simultaneously serves a pedagogical role, explains abstractions that underlie the Elemental library, advances the state of science for parallel matrix-matrix multiplication by providing a framework to systematically derive known and new algorithms for matrix-matrix-multiplication when computing on two-dimensional or three-

[†] Department of Computer Science, Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX.
Emails: martin.schatz@utexas.edu, ltm@cs.utexas.edu, rvdg@cs.utexas.edu.

[§] Institute for Computational and Mathematical Engineering, Stanford University, Stanford, CA 94305, poulson@stanford.edu.

[*] Parallel in this paper implicitly means *distributed memory* parallel.

dimensional meshes. While much of the new innovation for this paper concerns the extension of parallel matrix-matrix multiplication algorithms from two-dimensional to three-dimensional meshes, we believe that developing the reader's intuition for algorithms on two-dimensional meshes renders most of this new innovation much like a corollary to a theorem.

2. Background.

The parallelization of dense matrix-matrix multiplication is a well-studied subject. Cannon's algorithm (sometimes called roll-roll-compute) dates back to 1969 [9] and Fox's algorithm (sometimes called broadcast-roll-compute) dates back to the 1980s [15]. Both suffer from a number of shortcomings:

- They assume that p processes are viewed as an $d_0 \times d_1$ grid, with $d_0 = d_1 = \sqrt{p}$. Removing this constraint on d_0 and d_1 is nontrivial for these algorithms.
- They do not deal well with the case where one of the matrix dimensions becomes relatively small. This is the most commonly encountered case in libraries like LAPACK [3] and `libflame` [35, 36], and their distributed-memory counterparts: ScaLAPACK [11], PLAPACK [34], and Elemental [28].

Attempts to generalize [12, 21, 22] led to implementations that were neither simple nor effective.

A practical algorithm, which also results from the systematic approach discussed in this paper, can be described as "allgather-allgather-multiply" [1]. It does not suffer from the shortcomings of Cannon's and Fox's algorithms. It did not gain in popularity in part because libraries like ScaLAPACK and PLAPACK used a 2D block-cyclic distribution, rather than the 2D elemental distribution advocated by that paper. The arrival of the Elemental library, together with what we believe is our more systematic and extensible explanation, will hopefully elevate awareness of this result.

The Scalable Universal Matrix Multiplication Algorithm [33] is another algorithm that overcomes all shortcomings of Cannon's algorithm and Fox's algorithm. We believe it is a more widely known result in part because it can already be explained for a matrix that is distributed with a 2D blocked (but not cyclic) distribution and in part because it was easy to support in the ScaLAPACK and PLAPACK libraries. The original SUMMA paper gives four algorithms:

- For $C := AB + C$, SUMMA casts the computation in terms of multiple rank-k updates. This algorithm is sometimes called the broadcast-broadcast-multiply algorithm, a label we will see is somewhat limiting. We also call this algorithm "stationary C" for reasons that will become clear later. By design, this algorithm continues to perform well in the case where the width of A is small relative to the dimensions of C.
- For $C := A^T B + C$, SUMMA casts the computation in terms of multiple panel of rows times matrix multiplies, so performance is not degraded in the case where the height of A is small relative to the dimensions of B. We have also called this algorithm "stationary B" for reasons that will become clear later.
- For $C := AB^T + C$, SUMMA casts the computation in terms of multiple matrix-panel (of columns) multiplies, and so performance does not deteriorate when the width of C is small relative to the dimensions of A. We call this algorithm "stationary A" for reasons that will become clear later.
- For $C := A^T B^T + C$, the paper sketches an algorithm that is actually not practical.

In [17], it was shown how stationary A, B, and C algorithms can be formulated for each of the four cases of matrix-matrix multiplication, including for $C := A^T B^T +$

C. This then yielded a general, practical family of 2D matrix-matrix multiplication algorithms all of which were incorporated into PLAPACK and Elemental, and some of which are supported by ScaLAPACK. Some of the observations about developing 2D algorithms in the current paper can already be found in that paper, but our exposition is much more systematic and we use the matrix distribution that underlies Elemental to illustate the basic principles. Although the work by Agarwal et al. describes algorithms for the different matrix-matrix multiplication transpose variants, it does not describe how to create stationary A and B variants.

In the 1990s, it was observed that for the case where matrices were relatively small (or, equivalently, a relatively large number of nodes were available), better theoretical and practical performance resulted from viewing the p nodes as a $d_0 \times d_1 \times d_2$ mesh, yielding a 3D algorithm [2]. More recently, a 3D algorithm for computing the LU factorization of a matrix was devised in Tiskin [27] and Solomonik and Demmel [31]. In addition to the LU factorization algorithm devised in [31], a 3D algorithm for matrix-matrix multiplication was given for nodes arranged as an $d_0 \times d_1 \times d_2$ mesh, with $d_0 = d_1$ and $0 \leq d_2 < \sqrt[3]{p}$. This was labeled a 2.5D algorithm. Although the primary contribution of that work was LU-related, the 2.5D algorithm for matrix-matrix multiplication is the relevant portion to this paper. The focus of that study on 3D algorithms was the simplest case of matrix-matrix multiplication, $C := AB$.

In [25], an early attempt was made to combine multiple algorithms for computing $C = AB$ into a poly-algorithm, which refers to "the use of two or more algorithms to solve the same problem with a high level decision-making process determining which of a set of algorithms performs best in a given situation." That paper was published right at the time when SUMMA algorithms first became popular and when it was not yet completely understood that these SUMMA algorithm are inherently more practical than the Cannon's and Fox's algorithms. It already talked about "stationary A, B, and C" algorithms. In the paper, an attempt was made to combine all these approaches, including SUMMA, targeting general 2D Cartesian data distributions, which was (and still would be) a very ambitious goal. Our paper benefits from decades of experience with the more practical SUMMA algorithms and their variants. It purposely limits the data distribution to simple distributions, namely elemental distributions. This, we hope, allows the reader to gain a deep understanding in a simpler setting so that even if elemental distribution is not best for an encountered situation, a generalization can be easily derived. The family of presented 2D algorithms is a poly-algorithm implemented in Elemental.

3. Notation.

Although the focus of this paper is parallel distributed-memory matrix-matrix multiplication, the notation used is designed to be extensible to computation with higher-dimensional objects (tensors), on higher-dimensional grids. Because of this, the notation used may seem overly complex when restricted to matrix-matrix-multiplication only. In this section, we describe the notation used and the reasoning behind the choice of notation.

Grid dimension: d_x. Since we focus on algorithms for distributed-memory architectures, we must describe information about the grid on which we are computing. To support arbitrary-dimensional grids, we must express the shape of the grid in an extensible way. For this reason, we have chosen the subscripted letter d to indicate the size of a particular dimension of the grid. Thus, d_x refers to the number of processes comprising the xth dimension of the grid. In this paper, the grid is typically $d_0 \times d_1$.

Process location: s_x. In addition to describing the shape of the grid, it is useful to be able to refer to a particular process's location within the mesh of processes. For this, we use the subscripted s letter to refer to a process's location within some given dimension of the mesh of processes. Thus, s_x refers to a particular process's location within the xth dimension of the mesh of processes. In this paper, a typical process is labeled with (s_0, s_1).

Distribution: $\mathcal{D}_{(x_0,x_1,\ldots,x_{k-1})}$. In subsequent sections, we will introduce a notation for describing how data is distributed among processes of the grid. This notation will require a description of which dimensions of the grid are involved in defining the distribution. We use the symbol $\mathcal{D}_{(x_0,x_1,\ldots,x_{k-1})}$ to indicate a distribution which involves dimensions $x_0, x_1, \ldots, x_{k-1}$ of the mesh.

For example, when describing a distribution which involves the column and row dimension of the grid, we refer to this distribution as $\mathcal{D}_{(0,1)}$. Later, we will explain why the symbol $\mathcal{D}_{(0,1)}$ describes a different distribution from $\mathcal{D}_{(1,0)}$.

4. Of Matrix-Vector Operations and Distribution.

In this section, we discuss how matrix and vector distributions can be linked to parallel 2D matrix-vector multiplication and rank-1 update operations, which then allows us to eventually describe the stationary C, A, and B 2D algorithms for matrix-matrix multiplication that are part of the Elemental library.

4.1. Collective communication.

Collectives are fundamental to the parallelization of dense matrix operations. Thus, the reader must be (or become) familiar with the basics of these communications and is encouraged to read Chan et al. [10], which presents collectives in a systematic way that dovetails with the present paper.

To make this paper self-contained, Figure 4.1 (similar to Figure 1 in [10]) summarizes the collectives. In Figure 4.2 we summarize lower bounds on the cost of the collective communications, under basic assumptions explained in [10] (see [8] for an analysis of all-to-all), and the cost expressions that we will use in our analyses.

4.2. Motivation: matrix-vector multiplication.

Suppose $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$, and label their individual elements so that

$$A = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{pmatrix}, x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix}, \text{ and } y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix}.$$

Recalling that $y = Ax$ (matrix-vector multiplication) is computed as

$$\begin{array}{rcccc} \psi_0 = & \alpha_{0,0}\chi_0 + & \alpha_{0,1}\chi_1 + \cdots + & \alpha_{0,n-1}\chi_{n-1} \\ \psi_1 = & \alpha_{1,0}\chi_0 + & \alpha_{1,1}\chi_1 + \cdots + & \alpha_{1,n-1}\chi_{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ \psi_{m-1} = & \alpha_{m-1,0}\chi_0 + & \alpha_{m-1,1}\chi_1 + \cdots + & \alpha_{m-1,n-1}\chi_{n-1} \end{array}$$

Operation	Before				After			
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Permute	x_0	x_1	x_2	x_3	x_1	x_0	x_3	x_2
Broadcast	x				x	x	x	x
Reduce(-to-one)	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$			
Scatter	x_0 x_1 x_2 x_3				x_0	x_1	x_2	x_3
Gather	x_0	x_1	x_2	x_3	x_0 x_1 x_2 x_3			
Allgather	x_0	x_1	x_2	x_3	x_0 x_1 x_2 x_3	x_0 x_1 x_2 x_3	x_0 x_1 x_2 x_3	x_0 x_1 x_2 x_3
Reduce-scatter	$x_0^{(0)}$ $x_1^{(0)}$ $x_2^{(0)}$ $x_3^{(0)}$	$x_0^{(1)}$ $x_1^{(1)}$ $x_2^{(1)}$ $x_3^{(1)}$	$x_0^{(2)}$ $x_1^{(2)}$ $x_2^{(2)}$ $x_3^{(2)}$	$x_0^{(3)}$ $x_1^{(3)}$ $x_2^{(3)}$ $x_3^{(3)}$	$\sum_j x_0^{(j)}$	$\sum_j x_1^{(j)}$	$\sum_j x_2^{(j)}$	$\sum_j x_3^{(j)}$
Allreduce	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$
All-to-all	$x_0^{(0)}$ $x_1^{(0)}$ $x_2^{(0)}$ $x_3^{(0)}$	$x_0^{(1)}$ $x_1^{(1)}$ $x_2^{(1)}$ $x_3^{(1)}$	$x_0^{(2)}$ $x_1^{(2)}$ $x_2^{(2)}$ $x_3^{(2)}$	$x_0^{(3)}$ $x_1^{(3)}$ $x_2^{(3)}$ $x_3^{(3)}$	$x_0^{(0)}$ $x_0^{(1)}$ $x_0^{(2)}$ $x_0^{(3)}$	$x_1^{(0)}$ $x_1^{(1)}$ $x_1^{(2)}$ $x_1^{(3)}$	$x_2^{(0)}$ $x_2^{(1)}$ $x_2^{(2)}$ $x_2^{(3)}$	$x_3^{(0)}$ $x_3^{(1)}$ $x_3^{(2)}$ $x_3^{(3)}$

Fig. 4.1. *Collective communications considered in this paper.*

Communication	Latency	Bandw.	Comput.	Cost used for analysis
Permute	α	$n\beta$	$-$	$\alpha + n\beta$
Broadcast	$\lceil \log_2(p) \rceil \alpha$	$n\beta$	$-$	$\log_2(p)\alpha + n\beta$
Reduce(-to-one)	$\lceil \log_2(p) \rceil \alpha$	$n\beta$	$\frac{p-1}{p} n\gamma$	$\log_2(p)\alpha + n(\beta + \gamma)$
Scatter	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	$-$	$\log_2(p)\alpha + \frac{p-1}{p} n\beta$
Gather	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	$-$	$\log_2(p)\alpha + \frac{p-1}{p} n\beta$
Allgather	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	$-$	$\log_2(p)\alpha + \frac{p-1}{p} n\beta$
Reduce-scatter	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	$\frac{p-1}{p} n\gamma$	$\log_2(p)\alpha + \frac{p-1}{p} n(\beta + \gamma)$
Allreduce	$\lceil \log_2(p) \rceil \alpha$	$2\frac{p-1}{p} n\beta$	$\frac{p-1}{p} n\gamma$	$2\log_2(p)\alpha + \frac{p-1}{p} n(2\beta + \gamma)$
All-to-all	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	$-$	$\log_2(p)\alpha + \frac{p-1}{p} n\beta$

FIG. 4.2. *Lower bounds for the different components of communication cost. Conditions for the lower bounds are given in [10] and [8]. The last column gives the cost functions that we use in our analyses. For architectures with sufficient connectivity, simple algorithms exist with costs that remain within a small constant factor of all but one of the given formulae. The exception is the all-to-all, for which there are algorithms that achieve the lower bound for the α and β term separately, but it is not clear whether an algorithm that consistently achieves performance within a constant factor of the given cost function exists.*

we notice that element $\alpha_{i,j}$ multiplies χ_j and contributes to ψ_i. Thus we may summarize the interactions of the elements of x, y, and A by

$$\begin{array}{c||c|c|c|c}
 & \chi_0 & \chi_1 & \cdots & \chi_{n-1} \\
\hline\hline
\psi_0 & \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\
\hline
\psi_1 & \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\
\hline
\vdots & \vdots & \vdots & \ddots & \vdots \\
\hline
\psi_{m-1} & \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1}
\end{array} \tag{4.1}$$

which is meant to indicate that χ_j must be multiplied by the elements in the jth column of A while the ith row of A contributes to ψ_i.

4.3. Two-Dimensional Elemental Cyclic Distribution. It is well established that (weakly) scalable implementations of DLA operations require nodes to be logically viewed as a two-dimensional mesh [32, 20].

It is also well established that to achieve load balance for a wide range of matrix operations, matrices should be cyclically "wrapped" onto this logical mesh. We start with these insights and examine the simplest of matrix distributions that result: 2D elemental cyclic distribution [28, 19]. Denoting the number of nodes by p, a $d_0 \times d_1$ mesh must be chosen such that $p = d_0 d_1$.

Matrix distribution. The elements of A are assigned using an elemental cyclic (round-robin) distribution where $\alpha_{i,j}$ is assigned to node $(i \bmod d_0, j \bmod d_1)$. Thus, node (s_0, s_1) stores submatrix

$$A(s_0 \!:\! d_0 \!:\! m{-}1, s_1 \!:\! d_1 \!:\! n{-}1) = \left(\begin{array}{ccc} \alpha_{s_0,s_1} & \alpha_{s_0,s_1+d_1} & \cdots \\ \alpha_{s_0+d_0,s_1} & \alpha_{s_0+d_0,s_1+d_1} & \cdots \\ \vdots & \vdots & \ddots \end{array} \right),$$

where the left-hand side of the expression uses the MATLAB convention for expressing submatrices, starting indexing from zero instead of one. This is illustrated in Figure 4.3.

$\begin{array}{ccccc} & \chi_0 & & & \cdots \\ \psi_0 & \alpha_{0,0} & \alpha_{0,3} & \alpha_{0,6} & \cdots \\ & \alpha_{2,0} & \alpha_{2,3} & \alpha_{2,6} & \cdots \\ & \alpha_{4,0} & \alpha_{4,3} & \alpha_{4,6} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$	$\begin{array}{ccccc} & \chi_1 & & & \cdots \\ & \alpha_{0,1} & \alpha_{0,4} & \alpha_{0,7} & \cdots \\ \psi_2 & \alpha_{2,1} & \alpha_{2,4} & \alpha_{2,7} & \cdots \\ & \alpha_{4,1} & \alpha_{4,4} & \alpha_{4,7} & \cdots \\ & \vdots & \vdots & \vdots & \ddots \end{array}$	$\begin{array}{ccccc} & \chi_2 & & & \cdots \\ & \alpha_{0,2} & \alpha_{0,5} & \alpha_{0,8} & \cdots \\ & \alpha_{2,2} & \alpha_{2,5} & \alpha_{2,8} & \cdots \\ \psi_4 & \alpha_{4,2} & \alpha_{4,5} & \alpha_{4,8} & \cdots \\ & \vdots & \vdots & \vdots & \ddots \end{array}$
$\begin{array}{ccccc} & & \chi_3 & & \\ \psi_1 & \alpha_{1,0} & \alpha_{1,3} & \alpha_{1,6} & \cdots \\ & \alpha_{3,0} & \alpha_{3,3} & \alpha_{3,6} & \cdots \\ & \alpha_{5,0} & \alpha_{5,3} & \alpha_{5,6} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$	$\begin{array}{ccccc} & & \chi_4 & & \\ & \alpha_{1,1} & \alpha_{1,4} & \alpha_{1,7} & \cdots \\ \psi_3 & \alpha_{3,1} & \alpha_{3,4} & \alpha_{3,7} & \cdots \\ & \alpha_{5,1} & \alpha_{5,4} & \alpha_{5,7} & \cdots \\ & \vdots & \vdots & \vdots & \ddots \end{array}$	$\begin{array}{ccccc} & & \chi_5 & & \\ & \alpha_{1,2} & \alpha_{1,5} & \alpha_{1,8} & \cdots \\ & \alpha_{3,2} & \alpha_{3,5} & \alpha_{3,8} & \cdots \\ \psi_5 & \alpha_{5,2} & \alpha_{5,5} & \alpha_{5,8} & \cdots \\ & \vdots & \vdots & \vdots & \ddots \end{array}$

FIG. 4.3. *Distribution of A, x, and y within a 2 × 3 mesh. Redistributing a column of A in the same manner as y requires simultaneous scatters within rows of nodes while redistributing a row of A consistently with x requires simultaneous scatters within columns of nodes. In the notation of Section 5, here the distribution of x and y are given by* $x\,[(1,0),()]$ *and* $y\,[(0,1),()]$, *respectively, and A by* $A\,[(0),(1)]$.

$\begin{array}{ccccc} & \chi_0 & \chi_3 & \chi_6 & \cdots \\ \psi_0 & \alpha_{0,0} & \alpha_{0,3} & \alpha_{0,6} & \cdots \\ \psi_2 & \alpha_{2,0} & \alpha_{2,3} & \alpha_{2,6} & \cdots \\ \psi_4 & \alpha_{4,0} & \alpha_{4,3} & \alpha_{4,6} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$	$\begin{array}{ccccc} & \chi_1 & \chi_4 & \chi_7 & \cdots \\ \psi_0 & \alpha_{0,1} & \alpha_{0,4} & \alpha_{0,7} & \cdots \\ \psi_2 & \alpha_{2,1} & \alpha_{2,4} & \alpha_{2,7} & \cdots \\ \psi_4 & \alpha_{4,1} & \alpha_{4,4} & \alpha_{4,7} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$	$\begin{array}{ccccc} & \chi_2 & \chi_5 & \chi_8 & \cdots \\ \psi_0 & \alpha_{0,2} & \alpha_{0,5} & \alpha_{0,8} & \cdots \\ \psi_2 & \alpha_{2,2} & \alpha_{2,5} & \alpha_{2,8} & \cdots \\ \psi_4 & \alpha_{4,2} & \alpha_{4,5} & \alpha_{4,8} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$
$\begin{array}{ccccc} & \chi_0 & \chi_3 & \chi_6 & \cdots \\ \psi_1 & \alpha_{1,0} & \alpha_{1,3} & \alpha_{1,6} & \cdots \\ \psi_3 & \alpha_{3,0} & \alpha_{3,3} & \alpha_{3,6} & \cdots \\ \psi_5 & \alpha_{5,0} & \alpha_{5,3} & \alpha_{5,6} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$	$\begin{array}{ccccc} & \chi_1 & \chi_4 & \chi_7 & \cdots \\ \psi_1 & \alpha_{1,1} & \alpha_{1,4} & \alpha_{1,7} & \cdots \\ \psi_3 & \alpha_{3,1} & \alpha_{3,4} & \alpha_{3,7} & \cdots \\ \psi_5 & \alpha_{5,1} & \alpha_{5,4} & \alpha_{5,7} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$	$\begin{array}{ccccc} & \chi_2 & \chi_5 & \chi_8 & \cdots \\ \psi_1 & \alpha_{1,2} & \alpha_{1,5} & \alpha_{1,8} & \cdots \\ \psi_3 & \alpha_{3,2} & \alpha_{3,5} & \alpha_{3,8} & \cdots \\ \psi_5 & \alpha_{5,2} & \alpha_{5,5} & \alpha_{5,8} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$

FIG. 4.4. *Vectors x and y respectively redistributed as row-projected and column-projected vectors. The column-projected vector* $y\,[(0),()]$ *here is to be used to compute local results that will become contributions to a column vector* $y\,[(0,1),()]$ *which will result from adding these local contributions within rows of nodes. By comparing and contrasting this figure with Figure 4.3 it becomes obvious that redistributing* $x\,[(1,0),()]$ *to* $x\,[(1),()]$ *requires an allgather within columns of nodes while* $y\,[(0,1),()]$ *results from scattering* $y\,[(0),()]$ *within process rows.*

Column-major vector distribution. A *column-major* vector distribution views the $d_0 \times d_1$ mesh of nodes as a linear array of p nodes, numbered in *column-major* order. A vector is distributed with this distribution if it is assigned to this linear array of nodes in a round-robin fashion, one element at a time. In other words, consider vector y. Its element ψ_i is assigned to node $(i \bmod d_0, (i/d_0) \bmod d_1)$, where / denotes integer division. Or, equivalently in MATLAB-like notation, node (s_0, s_1) stores subvector $y(u(s_0,s_1)\!:\!p\!:\!m{-}1)$, where $u(s_0,s_1) = s_0 + s_1 d_0$ equals the rank of node (s_0, s_1) when the nodes are viewed as a one-dimensional array, indexed in column-major order. This distribution of y is illustrated in Figure 4.3.

Row-major vector distribution. Similarly, a *row-major* vector distribution views the $d_0 \times d_1$ mesh of nodes as a linear array of p nodes, numbered in *row-major* order. In other words, consider vector x. Its element χ_j is assigned to node $(j \bmod d_1, (j/d_1) \bmod d_0)$. Or, equivalently, node (s_0, s_1) stores subvector $x(v(s_0,s_1)\!:\!p\!:\!n{-}1)$, where $v(s_0,s_1) = s_0 d_1 + s_1$ equals the rank of node (s_0, s_1) when the nodes are viewed as a one-dimensional array, indexed in row-major order. The distribution of x is illustrated in Figure 4.3.

4.4. Parallelizing matrix-vector operations. In the following discussion, we assume that A, x, and y are distributed as discussed above[†]. At this point, we suggest comparing (4.1) with Figure 4.3.

Computing $y := Ax$. The relation between the distributions of a matrix, column-major vector, and row-major vector is illustrated by revisiting the most fundamental of computations in linear algebra, $y := Ax$, already discussed in Section 4.2. An examination of Figure 4.3 suggests that the elements of x must be gathered within columns of nodes (allgather within columns) leaving elements of x distributed as illustrated in Figure 4.4. Next, each node computes the partial contribution to vector y with its local matrix and copy of x. Thus, in Figure 4.4, ψ_i in each node becomes a contribution to the final ψ_i. These must be added together, which is accomplished by a summation of contributions to y within rows of nodes. An experienced MPI programmer will recognize this as a reduce-scatter within each row of nodes.

Under our communication cost model, the cost of this parallel algorithm is given by

$$\begin{aligned}
T_{y=Ax}(m,n,r,c) &= \underbrace{2\left\lceil \frac{m}{d_0} \right\rceil \left\lceil \frac{n}{d_1} \right\rceil \gamma}_{\text{local mvmult}} \\
&\quad + \underbrace{\log_2(d_0)\alpha + \frac{d_0-1}{d_0}\left\lceil \frac{n}{d_1} \right\rceil \beta}_{\text{allgather } x} + \underbrace{\log_2(d_1)\alpha + \frac{d_1-1}{d_1}\left\lceil \frac{m}{d_0} \right\rceil \beta + \frac{d_1-1}{d_1}\left\lceil \frac{m}{d_0} \right\rceil \gamma}_{\text{reduce-scatter } y} \\
&\approx 2\frac{mn}{p}\gamma + \underbrace{C_0 \frac{m}{d_0}\gamma + C_1 \frac{n}{d_1}\gamma}_{\text{load imbalance}} + \log_2(p)\alpha + \frac{d_0-1}{d_0}\frac{n}{d_1}\beta + \frac{d_1-1}{d_1}\frac{m}{d_0}\beta + \frac{d_1-1}{d_1}\frac{m}{d_0}\gamma
\end{aligned}$$

[†] We suggest the reader print copies of Figures 4.3 and 4.4 for easy referral while reading the rest of this section.

for some constants C_0 and C_1. We simplify this further to

$$2\frac{mn}{p}\gamma + \underbrace{\log_2(p)\alpha + \frac{d_0-1}{d_0}\frac{n}{d_1}\beta + \frac{d_1-1}{d_1}\frac{m}{d_0}\beta + \frac{d_1-1}{d_1}\frac{m}{d_0}\gamma}_{T^{+}{}_{\mathrm{y:=Ax}}(m,n,d_0,d_1)} \tag{4.2}$$

since the load imbalance contributes a cost similar to that of the communication[‡]. Here, $T^{+}{}_{\mathrm{y:=Ax}}(m,n,k/h,d_0,d_1)$ is used to refer to the overhead associated with the above algorithm for the $y = Ax$ operation. In Appendix A we use these estimates to show that this parallel matrix-vector multiplication is, for practical purposes, weakly scalable if d_0/d_1 is kept constant, but not if $d_0 \times d_1 = p \times 1$ or $d_0 \times d_1 = 1 \times p$.

Computing $x := A^T y$. Let us next discuss an algorithm for computing $x := A^T y$, where A is an $m \times n$ matrix and x and y are distributed as before (x with a row-major vector distribution and y with a column-major vector distribution).

Recall that $x = A^T y$ (transpose matrix-vector multiplication) means

$$\begin{array}{rcccc}
\chi_0 = & \alpha_{0,0}\psi_0 + & \alpha_{1,0}\psi_1 + \cdots + & \alpha_{n-1,0}\psi_{n-1} \\
\chi_1 = & \alpha_{0,1}\psi_0 + & \alpha_{1,1}\psi_1 + \cdots + & \alpha_{n-1,1}\psi_{n-1} \\
\vdots & \vdots & \vdots & \vdots \\
\chi_{m-1} = & \alpha_{0,m-1}\psi_0 + & \alpha_{1,m-1}\psi_1 + \cdots + & \alpha_{n-1,m-1}\psi_{n-1}
\end{array}$$

or,

$$\begin{array}{c|c|c|c}
\chi_0 = & \chi_1 = & \cdots & \chi_{m-1} = \\
\alpha_{0,0}\psi_0 + & \alpha_{0,1}\psi_0 + & \cdots & \alpha_{0,n-1}\psi_0 + \\
\alpha_{1,0}\psi_1 + & \alpha_{1,1}\psi_1 + & \cdots & \alpha_{1,n-1}\psi_1 + \\
\vdots & \vdots & & \vdots \\
\alpha_{n-1,0}\psi_{n-1} & \alpha_{n-1,1}\psi_{n-1} & \cdots & \alpha_{n-1,n-1}\psi_{n-1}
\end{array} \tag{4.3}$$

An examination of (4.3) and Figure 4.3 suggests that the elements of y must be gathered within rows of nodes (allgather within rows) leaving elements of y distributed as illustrated in Figure 4.4. Next, each node computes the partial contribution to vector x with its local matrix and copy of y. Thus, in Figure 4.4 χ_j in each node becomes a contribution to the final χ_j. These must be added together, which is accomplished by a summation of contributions to x within columns of nodes. We again recognize this as a reduce-scatter, but this time within each column of nodes.

The cost for this algorithm, approximating as we did when analyzing the algorithm for $y = Ax$, is

$$2\frac{mn}{p}\gamma + \underbrace{\log_2(p)\alpha + \frac{d_1-1}{d_1}\frac{n}{d_0}\beta + \frac{d_0-1}{d_0}\frac{m}{d_1}\beta + \frac{d_0-1}{d_0}\frac{m}{d_1}\gamma}_{T^{+}{}_{\mathrm{x:=A^Ty}}(m,n,d_1,d_0)}$$

where, as before, we ignore overhead due to load imbalance since terms of the same order appear in the terms that capture communication overhead.

[‡] It is tempting to approximate $\frac{x-1}{x}$ by 1, but this would yield formulae for the cases where the mesh is $p \times 1$ ($d_1 = 1$) or $1 \times p$ ($d_0 = 1$) that are overly pessimistic.

Computing $y := A^T x$. What if we wish to compute $y := A^T x$, where A is an $m \times n$ matrix and y is distributed with a column-major vector distribution and x with a row-major vector distribution? Now x must first be redistributed to a column-major vector distribution, after which the algorithm that we just discussed can be executed, and finally the result (in row-major vector distribution) must be redistributed to leave it as y in column-major vector distribution. This adds the cost of the permutation that redistributes x and the cost of the permutation that redistributes the result to y to the cost of $y := A^T x$.

Other cases. What if, when computing $y := Ax$ the vector x is distributed like a row of matrix A? What if the vector y is distributed like a column of matrix A? We leave these cases as an exercise to the reader.

The point is that understanding the basic algorithms for multiplying with A and A^T allows one to systematically derive and analyze algorithms when the vectors that are involved are distributed to the nodes in different ways.

Computing $A := yx^T + A$. A second commonly encountered matrix-vector operation is the rank-1 update: $A := \alpha yx^T + A$. We will discuss the case where $\alpha = 1$. Recall that

$$A + yx^T = \begin{pmatrix} \alpha_{0,0} + \psi_0\chi_0 & \alpha_{0,1} + \psi_0\chi_1 & \cdots & \alpha_{0,n-1} + \psi_0\chi_{n-1} \\ \alpha_{1,0} + \psi_1\chi_0 & \alpha_{1,1} + \psi_1\chi_1 & \cdots & \alpha_{1,n-1} + \psi_1\chi_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} + \psi_{m-1}\chi_0 & \alpha_{m-1,1} + \psi_{m-1}\chi_1 & \cdots & \alpha_{m-1,n-1} + \psi_{m-1}\chi_{n-1} \end{pmatrix},$$

which, when considering Figures 4.3 and 4.4, suggests the following parallel algorithm: All-gather of y within rows. All-gather of x within columns. Update of the local matrix on each node.

The cost for this algorithm, approximating as we did when analyzing the algorithm for $y = Ax$, yields

$$2\frac{mn}{p}\gamma + \underbrace{\log_2(p)\alpha + \frac{d_0 - 1}{d_0}\frac{n}{d_1}\beta + \frac{d_1 - 1}{d_1}\frac{m}{d_0}\beta}_{T^+{}_{\mathrm{A:=yx^T+A}}(m, n, d_0, d_1)}$$

where, as before, we ignore overhead due to load imbalance since terms of the same order appear in the terms that capture communication overhead. Notice that the cost is the same as a parallel matrix-vector multiplication, except for the "γ" term that results from the reduction within rows.

As before, one can modify this algorithm when the vectors start with different distributions building on intuition from matrix-vector multiplication. A pattern is emerging.

5. Generalizing the Theme.

The reader should now have an understanding how vector and matrix distribution are related to the parallelization of basic matrix-vector operations. We generalize the insights using sets of indices as "filters" to indicate what parts of a matrix or vector a given process owns.

The insights in this section are similar to those that underlie Physically Based Matrix Distribution [14] which itself also underlies PLAPACK. However, we formalize the notation beyond that used by PLAPACK. The link between distribution of vectors and matrices was first observed by Bisseling [6, 7], and, around the same time, in [24].

5.1. Vector distribution.

The basic idea is to use two different partitions of the natural numbers as a means of describing the distribution of the row and column indices of a matrix.

DEFINITION 5.1 (**Subvectors and submatrices**). *Let $x \in \mathbb{R}^n$ and $\mathcal{S} \subset \mathbb{N}$. Then $x[\mathcal{S}]$ equals the vector with elements from x with indices in the set $\mathcal{S}$, in the order in which they appear in vector x. If $A \in \mathbb{R}^{m \times n}$ and $\mathcal{S}, \mathcal{T} \subset \mathbb{N}$, then $A[\mathcal{S}, \mathcal{T}]$ is the submatrix formed by keeping only the elements of A whose row-indices are in $\mathcal{S}$ and column-indices are in $\mathcal{T}$, in the order in which they appear in matrix A.*

We illustrate this idea with simple examples:

EXAMPLE 1. *Let* $x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \chi_2 \\ \chi_3 \end{pmatrix}$ *and* $A = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} & \alpha_{0,3} & \alpha_{0,4} \\ \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ \alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} \\ \alpha_{4,0} & \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} \end{pmatrix}$.
If $\mathcal{S} = \{0, 2, 4, ...\}$ and $\mathcal{T} = \{1, 3, 5, ...\}$, then

$$x[\mathcal{S}] = \begin{pmatrix} \chi_0 \\ \chi_2 \end{pmatrix} \quad \text{and} \quad A[\mathcal{S}, \mathcal{T}] = \begin{pmatrix} \alpha_{0,1} & \alpha_{0,3} \\ \alpha_{2,1} & \alpha_{2,3} \\ \alpha_{4,1} & \alpha_{4,3} \end{pmatrix}.$$

We now introduce two fundamental ways to distribute vectors relative to a logical $d_0 \times d_1$ process grid.

DEFINITION 5.2 (**Column-major vector distribution**). *Suppose that $p \in \mathbb{N}$ processes are available, and define*

$$\mathcal{V}_p^\sigma(q) = \{N \in \mathbb{N} : N \equiv q + \sigma \pmod p\}, \quad q \in \{0, 1, ..., p-1\},$$

where $\sigma \in \{0, 1, ..., p-1\}$ is an arbitrary alignment parameter. *When p is implied from context and σ is unimportant to the discussion, we will simply denote the above set by $\mathcal{V}(q)$.*

If the p processes have been configured into a logical $d_0 \times d_1$ grid, a vector x is said to be in a column-major vector distribution *if process (s_0, s_1), where $s_0 \in \{0, \ldots, d_0-1\}$ and $s_1 \in \{0, \ldots, d_1-1\}$, is assigned the subvector $x(\mathcal{V}_p^\sigma(s_0+s_1 d_0))$. This distribution is represented via the $d_0 \times d_1$ array of indices*

$$\mathcal{D}_{(0,1)}(s_0, s_1) \equiv \mathcal{V}(s_0 + s_1 d_0), \quad (s_0, s_1) \in \{0, \ldots, d_0 - 1\} \times \{0, \ldots, d_1 - 1\},$$

and the shorthand $x[(0,1)]$ will refer to the vector x distributed such that process (s_0, s_1) stores $x(\mathcal{D}_{(0,1)}(s_0, s_1))$.

DEFINITION 5.3 (**Row-major vector distribution**). *Similarly, the $d_0 \times d_1$ array*

$$\mathcal{D}_{(1,0)} \equiv \mathcal{V}(s_1 + s_0 d_1), \quad (s_0, s_1) \in \{0, \ldots, d_0 - 1\} \times \{0, \ldots, d_1 - 1\},$$

is said to define a row-major vector distribution. *The shorthand $y[(1,0)]$ will refer to the vector y distributed such that process (s_0, s_1) stores $y(\mathcal{D}_{(1,0)}(s_0, s_1))$.*

The members of any column-major vector distribution, $\mathcal{D}_{(0,1)}$, or row-major vector distribution, $\mathcal{D}_{(1,0)}$, form a partition of $\mathbb{N}$. The names *column-major vector distribution* and *row-major vector distribution* come from the fact that the mappings $(s_0, s_1) \mapsto s_0 + s_1 d_0$ and $(s_0, s_1) \mapsto s_1 + s_0 d_1$ respectively label the $d_0 \times d_1$ grid with a column-major and row-major ordering.

As row-major and column-major distributions differ only by which dimension of the grid is considered first when assigning an order to the processes in the grid, we can give one general definition for a vector distribution with two-dimensional grids. We give this definition now.

Definition 5.4 (**Vector distribution**). *We call the $d_0 \times d_1$ array $\mathcal{D}_{(i,j)}$ a* vector distribution *if $i, j \in \{0, 1\}$, $i \neq j$, and there exists some alignment parameter $\sigma \in \{0, \ldots, p-1\}$ such that, for every grid position $(s_0, s_1) \in \{0, \ldots, d_0-1\} \times \{0, \ldots, d_1-1\}$,*

$$\mathcal{D}_{(i,j)}(s_0, s_1) = \mathcal{V}_p^{\sigma}(s_i + s_j d_i). \tag{5.1}$$

The shorthand $y\,[(i,j)]$ will refer to the vector y distributed such that process (s_0, s_1) stores $y(\mathcal{D}_{(i,j)}(s_0, s_1))$.

Figure 4.3 illustrates that to redistribute $y\,[(0,1)]$ to $y\,[(1,0)]$, and vice versa, requires a permutation communication (simultaneous point-to-point communications). The effect of this redistribution can be seen in Figure 5.2. Via a permutation communication, the vector y distributed as $y\,[(0,1)]$, can be redistributed as $y\,[(1,0)]$ which is the same distribution as the vector x.

In the preceding discussions, our definitions of $\mathcal{D}_{(0,1)}$ and $\mathcal{D}_{(1,0)}$ allowed for arbitrary alignment parameters. For the rest of the paper, we will only treat the case where all alignments are zero, i.e., the top-left entry of every (global) matrix and top entry of every (global) vector is owned by the process in the top-left of the process grid.

5.2. Induced matrix distribution. We are now ready to discuss how matrix distributions are induced by the vector distributions. For this, it pays to again consider Figure 4.3. The element $\alpha_{i,j}$ of matrix A, is assigned to the row of processes in which ψ_i exists and the column of processes in which χ_j exists. This means that in $y = Ax$ elements of x need only be communicated within columns of processes and local contributions to y need only be summed within rows of processes. This induces a Cartesian matrix distribution: Column j of A is assigned to the same column of processes as is χ_j. Row i of A is assigned to the same row of processes as ψ_i. We now answer the related questions "What is the set $\mathcal{D}_{(0)}(s_0)$ of matrix row indices assigned to process row s_0?" and "What is the set $\mathcal{D}_{(1)}(s_1)$ of matrix column indices assigned to process column s_1?"

Elemental symbol	Introduced symbol
M_C	(0)
M_R	(1)
V_C	$(0,1)$
V_R	$(1,0)$
$*$	$()$

Fig. 5.1. *The relationships between distribution symbols found in the Elemental library implementation and those introduced here. For instance, the distribution $A[M_C, M_R]$ found in the Elemental library implementation corresponds to the distribution $A\left[(0),(1)\right]$.*

Definition 5.5. *Let*

$$\mathcal{D}_{(0)}(s_0) = \bigcup_{s_1=0}^{d_1-1} \mathcal{D}_{(0,1)}(s_0, s_1) \quad \textit{and} \quad \mathcal{D}_{(1)}(s_1) = \bigcup_{s_1=0}^{d_0-1} \mathcal{D}_{(1,0)}(s_0, s_1).$$

Given matrix A, $A\left[\mathcal{D}_{(0)}(s_0), \mathcal{D}_{(1)}(s_1)\right]$ denotes the submatrix of A with row indices in the set $\mathcal{D}_{(0)}(s_0)$ and column indices in $\mathcal{D}_{(1)}(s_1)$. Finally, $A\left[(0),(1)\right]$ denotes the distribution of A that assigns $A\left[\mathcal{D}_{(0)}(s_0), \mathcal{D}_{(1)}(s_1)\right]$ to process (s_0, s_1).

We say that $\mathcal{D}_{(0)}$ and $\mathcal{D}_{(1)}$ are *induced* respectively by $\mathcal{D}_{(0,1)}$ and $\mathcal{D}_{(1,0)}$ because the process to which $\alpha_{i,j}$ is assigned is determined by the row of processes, s_0, to which y_i is assigned and the column of processes, s_1, to which x_j is assigned, so that it is ensured that in the matrix-vector multiplication $y = Ax$ communication needs only be within rows and columns of processes. Notice in Figure 5.2 that to redistribute indices of the vector y as the matrix column indices in A requires a communication within rows of processes. Similarly, to redistribute indices of the vector x as matrix row indices requires a communication within columns of processes. The above definition lies at the heart of our communication scheme.

5.3. Vector duplication. Two vector distributions, encountered in Section 4.4 and illustrated in Figure 4.4, still need to be specified with our notation. The vector x, duplicated as needed for the matrix-vector multiplication $y = Ax$, can be specified as $x\left[(0)\right]$ or, viewing x as a $n \times 1$ matrix, $x\left[(0),()\right]$. The vector y, duplicated so as to store local contributions for $y = Ax$, can be specified as $y\left[(1)\right]$ or, viewing y as a $n \times 1$ matrix, $y\left[(1),()\right]$. Here the $()$ should be interpreted as "all indices". In other words, $\mathcal{D}_{()} \equiv \mathbb{N}$.

5.4. Notation in Elemental library. Readers familiar with the Elemental library will notice that the distribution symbols defined within that library's implementation follow a different convention than that used for distribution symbols introduced in the previous subsections. This is due to the fact that the notation used in this paper was devised after the implementation of the Elemental library and we wanted the notation to be extensible to higher-dimensional objects (tensors). However, for every symbol utilized in the Elemental library implementation, there exists a unique symbol in the notation introduced here. In Figure 5.1, the relationships between distribution symbols utilized in the Elemental library implementation and the symbols used in this paper are defined.

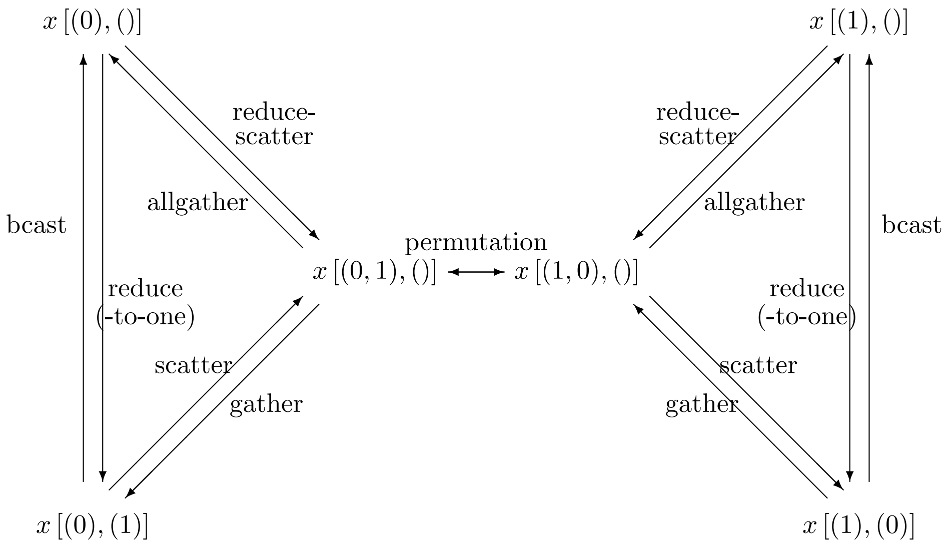


Fig. 5.2. *Summary of the communication patterns for redistributing a vector x. For instance, a method for redistributing x from a matrix column to a matrix row is found by tracing from the bottom-left to the bottom-right of the diagram.*

5.5. Of vectors, columns, and rows. A matrix-vector multiplication or rank-1 update may take as its input/output vectors (x and y) the rows and/or columns of matrices, as we will see in Section 6. This motivates us to briefly discuss the different communications needed to redistribute vectors to and from columns and rows. In our discussion, it will help to refer back to Figures 4.3 and 4.4.

Column to/from column-major vector. Consider Figure 4.3 and let a_j be a typical column in A. It exists within one single process column. Redistributing $a_j[(0),(1)]$ to $y[(0,1),()]$ requires simultaneous scatters within process rows. Inversely, redistributing $y[(0,1),()]$ to $a_j[(0),(1)]$ requires simultaneous gathers within process rows.

Column to/from row-major vector. Redistributing $a_j[(0),(1)]$ to $x[(1,0),()]$ can be accomplished by first redistributing to $y[(0,1),()]$ (simultaneous scatters within rows) followed by a redistribution of $y[(0,1),()]$ to $x[(1,0),()]$ (a permutation). Redistributing $x[(1,0)]$ to $a_j[(0),(1)]$ reverses these communications.

Column to/from column projected vector. Redistributing $a_j[(0),(1)]$ to $a_j[(0),()]$ (duplicated y in Figure 4.4) can be accomplished by first redistributing to $y[(0,1),()]$ (simultaneous scatters within rows) followed by a redistribution of $y[(0,1),()]$ to $y[(0),()]$ (simultaneous allgathers within rows). However, recognize that a scatter followed by an allgather is equivalent to a broadcast. Thus, redistributing $a_j[(0),(1)]$ to $a_j[(0),()]$ can be more directly accomplished by broadcasting within rows. Similarly, summing duplicated vectors $y[(0),()]$ leaving the result as $a_j[(0),(1)]$ (a column in A) can be accomplished by first summing them into $y[(0,1),()]$ (reduce-scatters within rows) followed by a redistribution to $a_j[(0),(1)]$ (gather within rows). But a reduce-scatter followed by a gather is equivalent to a reduce(-to-one) collective communication.

All communication patterns with vectors, rows, and columns. We summarize all the communication patterns that will be encounted when performing various matrix-vector multiplications or rank-1 updates, with vectors, columns, or rows as input, in

Algorithm: $y := Ax$ (GEMV)	Comments
$x\,[(1),()] \leftarrow x\,[(1,0),()]$	Redistribute x (allgather in columns)
$y^{(1)}\,[(0),()] := A\,[(0),(1)]\;\; x\,[(1),()]$	Local matrix-vector multiply
$y\,[(0,1),()] := \widehat{\sum}_1 y^{(1)}\,[(0),()]$	Sum contributions (reduce-scatter in rows)

FIG. 5.3. *Parallel algorithm for computing $y := Ax$.*

Algorithm $A := A + xy^T$ (GER)	Comments
$x\,[(0,1),()] \leftarrow x\,[(1,0),()]$	Redistribute x as a column-major vector (permutation)
$x\,[(0),()] \leftarrow x\,[(0,1),()]$	Redistribute x (allgather in rows)
$y\,[(1,0),()] \leftarrow y\,[(0,1),()]$	Redistribute y as a row-major vector (permutation)
$y((1),()) \leftarrow y((1,0),())$	Redistribute y (allgather in cols)
$A\,[(0),(1)] := x\,[(0),()]\,[y\,[(1),()]]^T$	Local rank-1 update

FIG. 5.4. *Parallel algorithm for computing $A := A + xy^T$.*

Algorithm: $\hat{c}_i := Ab_j$ (GEMV)	Comments
$x\,[(1),()] \leftarrow b_j\,[(0),(1)]$	Redistribute b_j: $x\,[(0,1),()] \leftarrow b_j\,[(0),(1)]$ (scatter in rows) $x\,[(1,0),()] \leftarrow x\,[(0,1),()]$ (permutation) $x\,[(1),()] \leftarrow x\,[(1,0),()]$ (allgather in columns).
$y^{(1)}\,[(0),()] := A\,[(0),(1)]\;\; x\,[(1),()]$	Local matrix-vector multiply
$\hat{c}_i\,[(0),(1)] := \widehat{\sum}_1 y^{(1)}\,[(0),()]$	Sum contributions: $y\,[(0,1),()] := \widehat{\sum}_t y^{(1)}\,[(0),()]$ (reduce-scatter in rows) $y\,[(1,0),()] \leftarrow y\,[(0,1),()]$ (permutation) $\hat{c}_i((0),(1)) \leftarrow y((1,0),())$ (gather in rows)

FIG. 5.5. *Parallel algorithm for computing $\hat{c}_i := Ab_j$ where $\hat{c}_i$ is a row of a matrix C and b_j is a column of a matrix B.*

Figure 5.2.

5.6. Parallelizing matrix-vector operations (revisited). We now show how the notation discussed in the previous subsection pays off when describing algorithms for matrix-vector operations.

Assume that A, x, and y are respectively distributed as $A\,[(0),(1)]$, $x\,[(1,0),()]$, and $y\,[(0,1),()]$. Algorithms for computing $y := Ax$ and $A := A + xy^T$ are given in Figures 5.3 and 5.4.

The discussion in Section 5.5 provides the insights to generalize these parallel matrix-vector operations to the cases where the vectors are rows and/or columns of

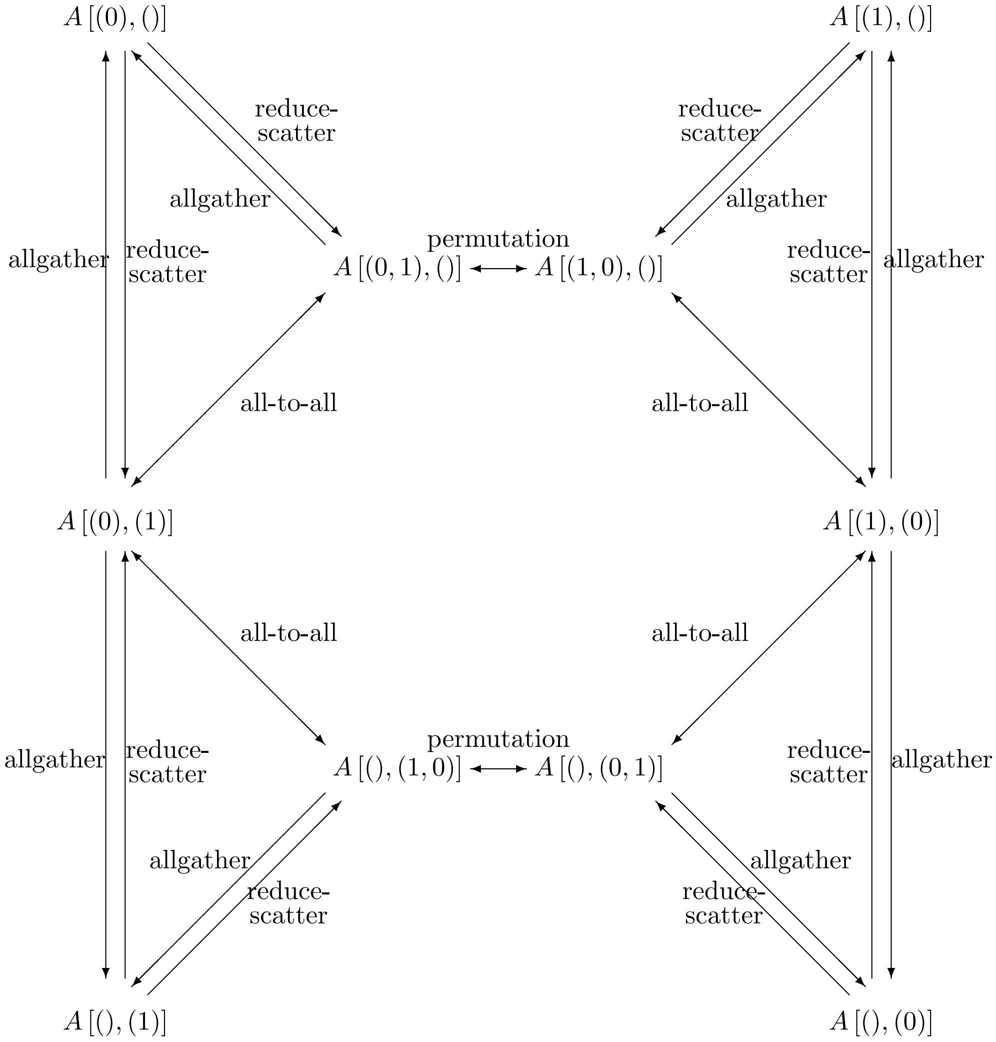


Fig. 6.1. *Summary of the communication patterns for redistributing a matrix A.*

matrices. For example, in Figure 5.5 we show how to compute a column of matrix C, $\hat{c}_i$, as the product of a matrix A times the column of a matrix B, b_j. Certain steps in Figures 5.3–5.5 have superscripts associated with outputs of local computations. These superscripts indicate that contributions rather than final results are computed by the operation. Further, the subscript to $\widehat{\sum}$ indicates along which dimension of the processing grid a reduction of contributions must occur.

5.7. Similar operations. What we have described is a general method. We leave it as an exercise to the reader to derive parallel algorithms for $x := A^T y$ and $A := yx^T + A$, starting with vectors that are distributed in various ways.

6. Elemental SUMMA: 2D algorithms (eSUMMA2D). We have now arrived at the point where we can discuss parallel matrix-matrix multiplication on a $d_0 \times d_1$ mesh, with $p = d_0 d_1$. In our discussion, we will assume an elemental distribution, but the ideas clearly generalize to other Cartesian distributions.

This section exposes a systematic path from the parallel rank-1 update and matrix-vector multiplication algorithms to highly efficient 2D parallel matrix-matrix multiplication algorithms. The strategy is to first recognize that a matrix-matrix multiplication can be performed by a series of rank-1 updates or matrix-vector multiplications. This gives us parallel algorithms that are inefficient. By then recognizing that the order of operations can be changed so that communication and computation can be separated and consolidated, these inefficient algorithms are transformed into efficient algorithms. While only explained for some of the cases of matrix multiplication, we believe the exposition is such that the reader can him/herself derive algorithms for the remaining cases by applying the ideas in a straight forward manner.

To fully understand how to attain high performance on a single processor, the reader should familiarize him/herself with, for example, the techniques in [16].

6.1. Elemental stationary C algorithms (eSUMMA2D-C). We first discuss the case where $C := C + AB$, where A and B have k columns each, with k relatively small [§] . We call this a rank-k update or panel-panel multiplication [16]. We will assume the distributions $C\left[(0),(1)\right]$, $A\left[(0),(1)\right]$, and $B\left[(0),(1)\right]$. Partition

$$A = \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{k-1} \end{array} \right) \text{ and } B = \left(\begin{array}{c} \hat{b}_0^T \\ \hline \hat{b}_1^T \\ \hline \vdots \\ \hline \hat{b}_{k-1}^T \end{array} \right) \text{ so that}$$

$$C := ((\cdots((C + a_0\hat{b}_0^T) + a_1\hat{b}_1^T) + \cdots) + a_{k-1}\hat{b}_{k-1}^T).$$

The following loop computes $C := AB + C$:

```
for $p = 0, \ldots, k-1$
    $a_p\left[(0),()\right] \leftarrow a_p\left[(0),(1)\right]$                                   (broadcasts within rows)
    $b_p^T\left[(),(1)\right] \leftarrow \hat{b}_p^T\left[(0),(1)\right]$                                   (broadcasts within cols)
    $C\left[(0),(1)\right] := C\left[(0),(1)\right] + a_p\left[(0),()\right]\hat{b}_p^T\left[(),(1)\right]$     (local rank-1 updates)
endfor
```

While Section 5.6 gives a parallel algorithm for GER, the problem with this algorithm is that (1) it creates a lot of messages and (2) the local computation is a rank-1 update, which inherently does not achieve high performance since it is memory bandwidth bound. The algorithm can be rewritten as

```
for $p = 0, \ldots, k-1$
    $a_p\left[(0),()\right] \leftarrow a_p\left[(0),(1)\right]$                                   (broadcasts within rows)
endfor
for $p = 0, \ldots, k-1$
    $b_p^T\left[(),(1)\right] \leftarrow \hat{b}_p^T\left[(0),(1)\right]$                                   (broadcasts within cols)
endfor
for $p = 0, \ldots, k-1$
    $C\left[(0),(1)\right] := C\left[(0),(1)\right] + a_p\left[(0),()\right]\hat{b}_p^T\left[(),(1)\right]$     (local rank-1 updates)
endfor
```

[§] There is an algorithmic block size, b_{alg}, for which a local rank-k update achieves peak performance [16]. Think of k as being that algorithmic block size for now.

```
Algorithm: $C := \text{GEMM\_C}(C, A, B)$
Partition $A \rightarrow \left( A_L \,\|\, A_R \right)$, $B \rightarrow \left( \frac{B_T}{B_B} \right)$
    where $A_L$ has 0 columns, $B_T$ has 0 rows
while $n(A_L) < n(A)$ do
    Determine block size $b$
    Repartition
      $\left( A_L \,\|\, A_R \right) \rightarrow \left( A_0 \,\|\, A_1 \,|\, A_2 \right)$,
      $\left( \frac{B_T}{B_B} \right) \rightarrow \begin{pmatrix} B_0 \\ \hline B_1 \\ \hline B_2 \end{pmatrix}$
        where $A_1$ has $b$ columns, $B_1$ has $b$ rows
    ----------------------------------------------------------------
      $A_1\,[(0),()] \leftarrow A_1\,[(0),(1)]$
      $B_1\,[(),(1)] \leftarrow B_1\,[(0),(1)]$
      $C\,[(0),(1)] := C\,[(0),(1)]$
                       $+\, A_1\,[(0),()] \quad B_1\,[(),(1)]$
    ----------------------------------------------------------------
    Continue with
      $\left( A_L \,\|\, A_R \right) \leftarrow \left( A_0 \,|\, A_1 \,\|\, A_2 \right)$,
      $\left( \frac{B_T}{B_B} \right) \leftarrow \begin{pmatrix} B_0 \\ \hline B_1 \\ \hline B_2 \end{pmatrix}$
endwhile
```

```
Algorithm: $C := \text{GEMM\_A}(C, A, B)$
Partition
    $C \rightarrow \left( C_L \,\|\, C_R \right)$, $B \rightarrow \left( B_L \,\|\, B_R \right)$
    where $C_L$ and $B_L$ have 0 columns
while $n(C_L) < n(C)$ do
    Determine block size $b$
    Repartition
      $\left( C_L \,\|\, C_R \right) \rightarrow \left( C_0 \,\|\, C_1 \,|\, C_2 \right)$,
      $\left( B_L \,\|\, B_R \right) \rightarrow \left( B_0 \,\|\, B_1 \,|\, B_2 \right)$
        where $C_1$ and $B_1$ have $b$ columns
    ----------------------------------------------------------------
      $B_1\,[(1),()] \leftarrow B_1\,[(0),(1)]$
      $C_1^{(1)}\,[(0),()] := A\,[(0),(1)]\, B_1\,[(1),()]$
      $C_1\,[(0),(1)] := \widehat{\sum}_1 C_1^{(1)}\,[(0),()]$
    ----------------------------------------------------------------
    Continue with
      $\left( C_L \,\|\, C_R \right) \leftarrow \left( C_0 \,|\, C_1 \,\|\, C_2 \right)$,
      $\left( B_L \,\|\, B_R \right) \leftarrow \left( B_0 \,|\, B_1 \,\|\, B_2 \right)$
endwhile
```

FIG. 6.2. *Algorithms for computing $C := AB + C$. Left: Stationary C. Right: Stationary A.*

and finally, equivalently,

$$\begin{array}{ll} A\,[(0),()] \leftarrow A\,[(0),(1)] & \text{(allgather within rows)} \\ B\,[(),(1)] \leftarrow B\,[(0),(1)] & \text{(allgather within cols)} \\ C\,[(0),(1)] := C\,[(0),(1)] + A\,[(0),()] \quad B\,[(),(1)] & \text{(local rank-k update)} \end{array}$$

Now the local computation is cast in terms of a local matrix-matrix multiplication (rank-k update), which can achieve high performance. Here (given that we assume elemental distribution) $A\,[(0),()] \leftarrow A\,[(0),(1)]$, within each row broadcasts k columns of A from different roots: an allgather if elemental distribution is assumed! Similarly $B\,[(),(1)] \leftarrow B\,[(0),(1)]$, within each column broadcasts k rows of B from different roots: another allgather if elemental distribution is assumed!

Based on this observation, the SUMMA-like algorithm can be expressed as a loop around such rank-k updates, as given in Figure 6.2 (left)[¶]. The purpose for the loop is to reduce workspace required to store duplicated data. Notice that, if an elemental distribution is assumed, the SUMMA-like algorithm should *not* be called a broadcast-broadcast-compute algorithm. Instead, it becomes an allgather-allgather-compute algorithm. We will also call it a stationary C algorithm, since C is not communicated (and hence "owner computes" is determined by what processor owns what element of C). The primary benefit from a having a loop around rank-k updates is that it reduces the required local workspace at the expense of an increase only in the α term of the communication cost.

We label this algorithm *eSUMMA2D-C*, an elemental SUMMA-like algorithm targeting a two-dimensional mesh of nodes, stationary C variant. It is not hard to extend

[¶] We use FLAME notation to express the algorithm, which has been used in our papers for more than a decade [18].

the insights to non-elemental distributions (as, for example, used by ScaLAPACK or PLAPACK).

An approximate cost for the described algorithm is given by

$$\begin{aligned}
&T_{\mathrm{eSUMMA2D-C}}(m,n,k,d_0,d_1)\\
&= \frac{2mnk}{p}\gamma + \frac{k}{b_{\mathrm{alg}}}\log_2(d_1)\alpha + \frac{d_1-1}{d_1}\frac{mk}{d_0}\beta + \frac{k}{b_{\mathrm{alg}}}\log_2(d_0)\alpha + \frac{d_0-1}{d_0}\frac{nk}{d_1}\beta\\
&= \frac{2mnk}{p}\gamma + \underbrace{\frac{k}{b_{\mathrm{alg}}}\log_2(p)\alpha + \frac{(d_1-1)mk}{p}\beta + \frac{(d_0-1)nk}{p}\beta.}_{T^{+}{}_{\mathrm{eSUMMA2D-C}}(m,n,k,d_0,d_1)}
\end{aligned}$$

This estimate ignores load imbalance (which leads to a γ term of the same order as the β terms) and the fact that the allgathers may be unbalanced if b_{alg} is not an integer multiple of both d_0 and d_1. As before and throughout this paper, T^+ refers to the communication overhead of the proposed algorithm (e.g. $T^+_{eSUMMA2D-C}$ refers to the communication overhead of the eSUMMA2D-C algorithm.)

It is not hard to see that, for practical purposes[‖], the weak scalability of the eSUMMA2D-C algorithm mirrors that of the parallel matrix-vector multiplication algorithm analyzed in Appendix A: it is weakly scalable when $m = n$ and $d_0 = d_1$, for arbitrary k.

At this point it is important to mention that this resulting algorithm may seem similar to an approach described in prior work [1]. Indeed, this allgather-allgather-compute approach to parallel matrix-matrix multiplication is described in that paper for the matrix-matrix multiplication variants $C = AB$, $C = AB^T$, $C = A^TB$, and $C = A^TB^T$ under the assumption that all matrices are approximately the same size; a surmountable limitation. As we have argued previously, the allgather-allgather-compute approach is particularly well-suited for situations where we wish not to communicate the matrix C. In the next section, we describe how to systematically derive algorithms for situations where we wish to avoid communicating the matrix A.

6.2. Elemental stationary A algorithms (eSUMMA2D-A). Next, we discuss the case where $C := C + AB$, where C and B have n columns each, with n relatively small. For simplicity, we also call that parameter b_{alg}. We call this a matrix-panel multiplication [16]. We again assume that the matrices are distributed as $C\,[(0),(1)]$, $A\,[(0),(1)]$, and $B\,[(0),(1)]$. Partition

$$C = \left(\begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1}\end{array}\right) \quad\text{and}\quad B = \left(\begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{n-1}\end{array}\right)$$

so that $c_j = Ab_j + c_j$. The following loop will compute $C = AB + C$:

```
for $j = 0, \ldots, n-1$
    $b_j\,[(0,1),()] \leftarrow b_j\,[(0),(1)]$                          (scatters within rows)
    $b_j\,[(1,0),()] \leftarrow b_j\,[(0,1),()]$                         (permutation)
    $b_j\,[(1),()] \leftarrow b_j\,[(1,0),()]$                           (allgathers within cols)
    $c_j\,[(0),()] := A\,[(0),(1)]\,b_j\,[(1),()]$                      (local matvec mult.)
    $c_j\,[(0),(1)] \leftarrow \widehat{\sum}_1 c_j\,[(0),()]$          (reduce-to-one within rows)
endfor
```

[‖] The very slow growing factor $\log_p(p)$ prevents weak scalability unless it is treated as a constant.

While Section 5.6 gives a parallel algorithm for GEMV, the problem again is that (1) it creates a lot of messages and (2) the local computation is a matrix-vector multiply, which inherently does not achieve high performance since it is memory bandwidth bound. This can be restructured as

```
for $j = 0, \ldots, n-1$
    $b_j\,[(0,1),()] \leftarrow b_j\,[(0),(1)]$                  (scatters within rows)
endfor
for $j = 0, \ldots, n-1$
    $b_j\,[(1,0),()] \leftarrow b_j\,[(0,1),()]$                  (permutation)
endfor
for $j = 0, \ldots, n-1$
    $b_j\,[(1),()] \leftarrow b_j\,[(1,0),()]$                    (allgathers within cols)
endfor
for $j = 0, \ldots, n-1$
    $c_j\,[(0),()] := A\,[(0),(1)]\,b_j\,[(1),()]$               (local matvec mult.)
endfor
for $j = 0, \ldots, n-1$
    $c_j\,[(0),(1)] \leftarrow \widehat{\sum}_1 c_j\,[(0),()]$    (simultaneous reduce-to-one
endfor                                                            within rows)
```

and finally, equivalently,

```
$B\,[(1),()] \leftarrow B\,[(0),(1)]$                             (all-to-all within rows, permu-
                                                                  tation, allgather within cols)
$C\,[(0),()] := A\,[(0),(1)]\,B\,[(1),()] + C\,[(0),()]$          (simultaneous local matrix
                                                                  multiplications)
$C\,[(0),(1)] \leftarrow \widehat{\sum} C\,[(0),()]$              (reduce-scatter within rows)
```

Now the local computation is cast in terms of a local matrix-matrix multiplication (matrix-panel multiply), which can achieve high performance. A stationary A algorithm for arbitrary n can now be expressed as a loop around such parallel matrix-panel multiplies, given in Figure 6.2 (right).

An approximate cost for the described algorithm is given by

$$
\begin{aligned}
&T_{\mathrm{eSUMMA2D-A}}(m,n,k,d_0,d_1) = \\
&\quad \tfrac{n}{b_{\mathrm{alg}}}\log_2(d_1)\alpha + \tfrac{d_1-1}{d_1}\tfrac{nk}{d_0}\beta && \text{(all-to-all within rows)} \\
&\quad +\tfrac{n}{b_{\mathrm{alg}}}\alpha + \tfrac{n}{d_1}\tfrac{k}{d_0}\beta && \text{(permutation)} \\
&\quad +\tfrac{n}{b_{\mathrm{alg}}}\log_2(d_0)\alpha + \tfrac{d_0-1}{d_0}\tfrac{nk}{d_1}\beta && \text{(allgather within cols)} \\
&\quad +\tfrac{2mnk}{p}\gamma && \text{(simultaneous local matrix-panel mult.)} \\
&\quad +\tfrac{n}{b_{\mathrm{alg}}}\log_2(d_1)\alpha + \tfrac{d_1-1}{d_1}\tfrac{mn}{d_0}\beta + \tfrac{d_1-1}{d_1}\tfrac{mn}{d_0}\gamma && \text{(reduce-scatter within rows)}
\end{aligned}
$$

As we discussed earlier, the cost function for the all-to-all operation is somewhat suspect. Still, if an algorithm that attains the lower bound for the α term is employed, the β term must at most increase by a factor of $\log_2(d_1)$ [8], meaning that it is not the dominant communication cost. The estimate ignores load imbalance (which leads to a γ term of the same order as the β terms) and the fact that various collective communications may be unbalanced if b_{alg} is not an integer multiple of both d_0 and d_1.

While the overhead is clearly greater than that of the eSUMMA2D-C algorithm when $m = n = k$, the overhead is *comparable* to that of the eSUMMA2D-C algorithm; so the weak scalability results are, asymptotically, the same. Also, it is not hard to see that if m and k are large while n is small, this algorithm achieves better parallelism since less communication is required: The stationary matrix, A, is then the largest matrix and not communicating it is beneficial. Similarly, if m and n are large while k is small, then the eSUMMA2D-C algorithm does not communicate the largest matrix, C, which is beneficial.

6.3. Communicating submatrices. In Figure 6.1 we illustrate the collective communications required to redistribute submatrices from one distribution to another and the collective communications required to implement them.

6.4. Other cases. We leave it as an exercise to the reader to propose and analyze the remaining stationary A, B, and C algorithms for the other cases of matrix-matrix multiplication: $C := A^T B + C$, $C := AB^T + C$, and $C := A^T B^T + C$.

The point is that we have presented a systematic framework for deriving a family of parallel matrix-matrix multiplication algorithms.

7. Elemental SUMMA: 3D algorithms (eSUMMA3D).

We now view the p processors as forming a $d_0 \times d_1 \times h$ mesh, which one should visualize as h stacked layers, where each layer consists of a $d_0 \times d_1$ mesh. The extra dimension is used to gain an extra level of parallelism, which reduces the overhead of the 2D SUMMA algorithms within each layer at the expense of communications between the layers.

The approach used to generalize Elemental SUMMA 2D algorithms to Elemental SUMMA 3D algorithms can be easily modified to use Cannon's or Fox's algorithms (with the constraints and complications that come from using those algorithms), or any other distribution for which SUMMA can be used (pretty much any Cartesian distribution).

7.1. 3D stationary C algorithms (eSUMMA3D-C). Partition, A and B so that $A = \left(\begin{array}{c|c|c} A_0 & \cdots & A_{h-1} \end{array} \right)$ and $B = \left(\begin{array}{c} B_0 \\ \hline \vdots \\ \hline B_{h-1} \end{array} \right)$, where A_p and B_p have approximately k/h columns and rows, respectively. Then

$$C + AB = \underbrace{(C + A_0 B_0)}_{\text{by layer 0}} + \underbrace{(0 + A_1 B_1)}_{\text{by layer 1}} + \cdots + \underbrace{(0 + A_{h-1} B_{h-1})}_{\text{by layer h-1}}.$$

This suggests the following 3D algorithm:

- Duplicate C to each of the layers, initializing the duplicates assigned to layers 1 through $h-1$ to zero. This requires no communication. We will ignore the cost of setting the duplicates to zero.
- Scatter A and B so that layer H receives A_H and B_H. This means that all processors $(I, J, 0)$ simultaneously scatter approximately $(m+n)k/(d_0 d_1)$ data to processors $(I, J, 0)$ through $(I, J, h-1)$. The cost of such a scatter can be approximated by

$$\log_2(h)\alpha + \frac{h-1}{h}\frac{(m+n)k}{d_0 d_1}\beta = \log_2(h)\alpha + \frac{(h-1)(m+n)k}{p}\beta. \qquad (7.1)$$

- Compute $C := C + A_K B_K$ simultaneously on all the layers. If eSUMMA2D-C is used for this in each layer, the cost is approximated by

$$(7.2)\frac{mnk}{p}\gamma + \underbrace{\frac{k}{hb_{\mathrm{alg}}}\left(\log_2(p) - \log_2(h)\right)\alpha + \frac{(d_1-1)mk}{p}\beta + \frac{(d_0-1)nk}{p}\beta.}_{T^+{}_{\mathrm{eSUMMA2D-C}}(m,n,k/h,d_0,d_1)}$$

- Perform reduce operations to sum the contributions from the different layers to the copy of C in layer 0. This means that contributions from processors $(I,J,0)$ through (I,J,K) are reduced to processor $(I,J,0)$. An estimate for this reduce-to-one is

$$(7.3) \qquad \log_2(h)\alpha + \frac{mn}{d_0 d_1}\beta + \frac{mn}{d_0 d_1}\gamma = \log_2(h)\alpha + \frac{mnh}{p}\beta + \frac{mnh}{p}\gamma.$$

Thus, an estimate for the total cost of this eSUMMA3D-C algorithm for this case of `gemm` results from adding (7.1)–(7.3).

Let us analyze the case where $m = n = k$ and $d_0 = d_1 = \sqrt{p/h}$ in detail. The cost becomes

$$\begin{aligned}
&C_{\mathrm{eSUMMA3D-C}}(n,n,n,d_0,d_0,h) \\
&= 2\frac{n^3}{p}\gamma + \frac{n}{hb_{\mathrm{alg}}}(\log_2(p) - \log_2(h))\alpha + 2\frac{(d_0-1)n^2}{p}\beta + \log_2(h)\alpha + 2\frac{(h-1)n^2}{p}\beta \\
&\qquad\qquad + \log_2(h)\alpha + \frac{n^2 h}{p}\beta + \frac{n^2 h}{p}\gamma \\
&= 2\frac{n^3}{p}\gamma + \left[\frac{n}{hb_{\mathrm{alg}}}(\log_2(p) - \log_2(h)) + 2\log_2(h)\right]\alpha + \left[2(\frac{\sqrt{p}}{\sqrt{h}} - 1) + 3h - 2 + \frac{\gamma}{\beta}h\right]\frac{n^2}{p}\beta.
\end{aligned}$$

Now, let us assume that the α term is inconsequential (which will be true if n is large enough). Then the minimum can be computed by taking the derivative (with respect to h) and setting this to zero: $-\sqrt{p}h^{-3/2} + (3+K) = 0$ or $h = ((3+K)/\sqrt{p})^{-2/3} = \sqrt[3]{p}/(3+K)^{2/3}$, where $K = \gamma/\beta$. Typically $\gamma/\beta \ll 1$ and hence $(3+K)^{-2/3} \approx 3^{-2/3} \approx 1/2$, meaning that the optimal h is given by $h \approx \sqrt[3]{p}/2$. Of course, details of how the collective communication algorithms are implemented, etc., will affect this optimal choice. Moreover, α is typically four to five orders of magnitude greater than β, and hence the α term cannot be ignored for more moderate matrix sizes, greatly affecting the analysis.

While the cost analysis assumes the special case where $m = n = k$ and $d_0 = d_1$, and that the matrices is perfectly balanced among the $d_0 \times d_0$ mesh, the description of the algorithm is general. It is merely the case that the cost analysis for the more general case becomes more complex.

The algorithm and the related insights are similar to those described in Agarwal, et.al [2], although we arrive at this algorithm via a different path.

Now, PLAPACK and Elemental both include stationary C algorithms for the other cases of matrix multiplication ($C := \alpha A^T B + \beta C$, $C := \alpha A B^T + \beta C$, and $C := \alpha A^T B^T + \beta C$). Clearly, 3D algorithms that utilize these implementations can be easily proposed. For example, if $C := A^T B^T + C$ is to be computed, one can partition

$$A = \left(\begin{array}{c} A_0 \\ \hline \vdots \\ \hline A_{h-1} \end{array}\right) \quad \text{and} \quad B = \left(\begin{array}{c|c|c} B_0 & \cdots & B_{h-1} \end{array}\right),$$

after which

$$C + A^T B^T = \underbrace{(C + A_0^T B_0^T)}_{\text{by layer 0}} + \underbrace{(0 + A_1^T B_1^T)}_{\text{by layer 1}} + \cdots + \underbrace{(0 + A_{h-1}^T B_{h-1}^T)}_{\text{by layer h-1}}.$$

The communication overhead for all four cases is similar, meaning that for all four cases, the resulting stationary C 3D algorithms have similar properties.

7.2. Stationary A algorithms (eSUMMA3D-A). Let us next focus on $C := AB + C$. Algorithms such that A is the stationary matrix are implemented in PLAPACK and Elemental. They have costs similar to that of the eSUMMA2D-C algorithm.

Let us describe a 3D algorithm, with a $d_0 \times d_1 \times h$ mesh, again viewed as h layers. If we partition, conformally, C and B so that $C = \left(\begin{array}{c|c|c} C_0 & \cdots & C_{h-1} \end{array} \right)$ and $B = \left(\begin{array}{c|c|c} B_0 & \cdots & B_{h-1} \end{array} \right)$, then

$$\underbrace{(C_0 := C_0 + AB_0}_{\text{by layer } 0} \mid C_1 := C_1 + AB_1 \mid \cdots \mid \underbrace{C_{h-1} := C_{h-1} + AB_{h-1})}_{\text{by layer } h-1}.$$

This suggests the following 3D algorithm:

- Duplicate (broadcast) A to each of the layers. If matrix A is perfectly balanced among the processors, the cost of this can be approximated by
$$\log_2(h)\alpha + \frac{mk}{d_0 d_1}\beta.$$
- Scatter C and B so that layer K recieves C_K and B_K. This means having all processors $(I, J, 0)$ simultaneously scatter approximately $(mn + nk)/(d_0 d_1)$ data to processors $(I, J, 0)$ through $(I, J, h-1)$. The cost of such a scatter can be approximated by
$$\log_2(h)\alpha + \frac{h-1}{h}\frac{(m+k)n}{d_1 d_0}\beta = \log_2(h)\alpha + \frac{(h-1)(m+k)n}{p}\beta.$$
- Compute $C_K := C_K + AB_K$ simultaneously on all the layers with a 2D stationary A algorithm. The cost of this is approximated by
$$\frac{2mnk}{p}\gamma + T^+{}_{\text{eSUMMA2D-A}}(m, n/h, k, d_0, d_1).$$
- Gather the C_K submatrices to Layer 0. The cost of such a gather can be approximated by
$$\log_2(h)\alpha + \frac{h-1}{h}\frac{mn}{d_1 d_0}\beta.$$

Rather than giving the total cost, we merely note that the stationary A 3D algorithms can similarly be stated for general m, n, k, d_0, and d_1, and that then the costs are similar.

Now, PLAPACK and Elemental both include stationary A algorithms for the other cases of matrix multiplication. Again, 3D algorithms that utilize these implementations can be easily proposed.

7.3. Stationary B algorithms (eSUMMA3D-B). Finally, let us again focus on $C := AB + C$. Algorithms such that B is the stationary matrix are also implemented in PLAPACK and Elemental. They also have costs similar to that of the SUMMA algorithm for $C := AB + C$.

Let us describe a 3D algorithm, with a $d_0 \times d_1 \times h$ mesh, again viewed as h layers. If we partition, conformally, C and A so that $C = \left(\begin{array}{c} C_0 \\ \hline \vdots \\ \hline C_{h-1} \end{array}\right)$ and $A = \left(\begin{array}{c} A_0 \\ \hline \vdots \\ \hline A_{h-1} \end{array}\right)$, then

$$\left(\begin{array}{c} C_0 + A_0 B \\ \hline C_1 + A_1 B \\ \hline \vdots \\ \hline C_{h-1} := C_{h-1} + A_{h-1} B \end{array}\right) \begin{array}{c} \text{by layer } 0 \\ \text{by layer } 1 \\ \vdots \\ \text{by layer } h-1 \end{array}$$

This suggests the following 3D algorithm:

- Duplicate (broadcast) B to each of the layers. If matrix B is perfectly balanced among the processors, the cost can be approximated by
$$\log_2(h)\alpha + \frac{nk}{d_0 d_1}\beta.$$
- Scatter C and A so that layer K recieves C_K and A_K. This means having all processors $(I, J, 0)$ simultaneously scatter approximately $(mn + mk)/(d_0 d_1)$ data to processors $(I, J, 0)$ through $(I, J, h-1)$. The cost of such a scatter can be approximated by
$$\log_2(h)\alpha + \frac{h-1}{h}\frac{m(n+k)}{d_1 d_0}\beta = \log_2(h)\alpha + \frac{(h-1)m(n+k)}{p}\beta$$
- Compute $C_K := C_K + A_K B$ simultaneously on all the layers with a 2D stationary B algorithm. The cost of this is approximated by
$$\frac{2mnk}{p}\gamma + T^{+}{}_{eSUMMA2D-B}(m/h, n, k, d_0, d_1)$$
- Gather the C_K submatrices to Layer 0. The cost of such a gather can be approximated by
$$\log_2(h)\alpha + \frac{h-1}{h}\frac{mn}{d_1 d_0}\beta$$

Again, a total cost similar to those for stationary C and A algorithms results. Again, PLAPACK and Elemental both include stationary B algorithms for the other cases of matrix multiplication. Again, 3D algorithms that utilize these implementations can be easily proposed.

7.4. Other cases. We leave it as an exercise to the reader to propose and analyze the remaining eSUMMA3D-A, eSUMMA3D-B, and eSUMMA3D-C algorithms for the other cases of matrix-matrix multiplication: $C := A^T B + C$, $C := AB^T + C$, and $C := A^T B^T + C$.

The point is that we have presented a systematic framework for deriving a family of parallel 3D matrix-matrix multiplication algorithms.

7.5. Discussion. This extra level of parallelism gained with 3D SUMMA algorithms allows us to parallelize computation across any one of three dimensions involved in the matrix-matrix multiplication (two dimensions for forming the output C, and one for the reduction of A and B). The particular 3D SUMMA algorithmic variant dictates the dimension along which the extra parallelism occurs. In [13], a geometric model is developed that views the set of scalar computations associated with a matrix-matrix multiplication as a set of lattice points forming a rectangular prism. This geometric model is based on the Loomis-Whitney inequality [26] that has been used to devise algorithms that achieve the parallel bandwidth cost lower bound for matrix-matrix multiplication [4, 5]. Considering this geometric model, each 3D SUMMA algorithmic variant corresponds to performing computations appearing in different slices[**] in parallel. The orientation of slices is dictated by the 3D SUMMA algorithmic variant chosen and the order in which computations are performed within a slice is dictated by the 2D SUMMA algorithm used within each layer of the processing mesh. We now discuss how the communication overhead of Elemental 2D and 3D SUMMA algorithms relate to the lower bounds of both the latency and bandwidth costs associated with parallel matrix-matrix multiplication.

In [23], it was shown that the lower bound on communicated data is $\Omega(n^2/\sqrt{p})$ for a matrix multiplication of two $n \times n$ matrices computed on a processing grid involving p processes arranged as a two-dimensional mesh and $\Omega(n^2/\sqrt[3]{p^2})$ for a matrix multiplication of two $n \times n$ matrices computed on a processing grid involving p processes arranged as a three-dimensional mesh. Examination of the cost functions associated with each eSUMMA2D algorithm and eSUMMA3D algorithm shows that each achieves the lower-bound on communication for such an operation.

With regards to latency, the lower bound on number of messages required is $\Omega(log(p))$ for a matrix multiplication of two $n \times n$ matrices computed on a processing grid involving p processes arranged as either a two-dimensional or three-dimensional mesh. Examination of the cost functions shows that each achieves the lower-bound on latency as well if we assume that the algorithmic block-size $b_{alg} = n$. Otherwise, the proposed algorithms do not achieve the lower bound.

8. Performance Experiments. In this section, we present performance results that support the insights in the previous sections. Implementations of the eSUMMA-2D algorithms are all part of the Elemental library. The eSUMMA-3D algorithms were implemented with Elemental, building upon its eSUMMA-2D algorithms and implementations. In all these experiments, it was assumed that the data started and finished distributed within one layer of the three-dimensional mesh of nodes so that all communication necessary to duplicate was included in the performance calculations.

As in [28, 29], performance experiments were carried out on the IBM Blue Gene/P architecture with compute nodes that consist of four 850 MHz PowerPC 450 processors for a combined theoretical peak performance of 13.6 GFlops per node using double-precision arithmetic. Nodes are interconnected by a three-dimensional torus topology and a collective network that each support a per-node bidirectional bandwidth of 2.55 GB/s. In all graphs, the top of the graph represents peak performance for this architecture so that the attained efficiency can be easily judged.

The point of the performance experiments was to demonstrate the merits of 3D algorithms. For this reason, we simply fixed the algorithmic block size, b_{alg}, to 128 for all experiments. The number of nodes, p, was chosen to be various powers of two,

[**] As used, the term “slice” refers to a set of “superbricks” in [13].

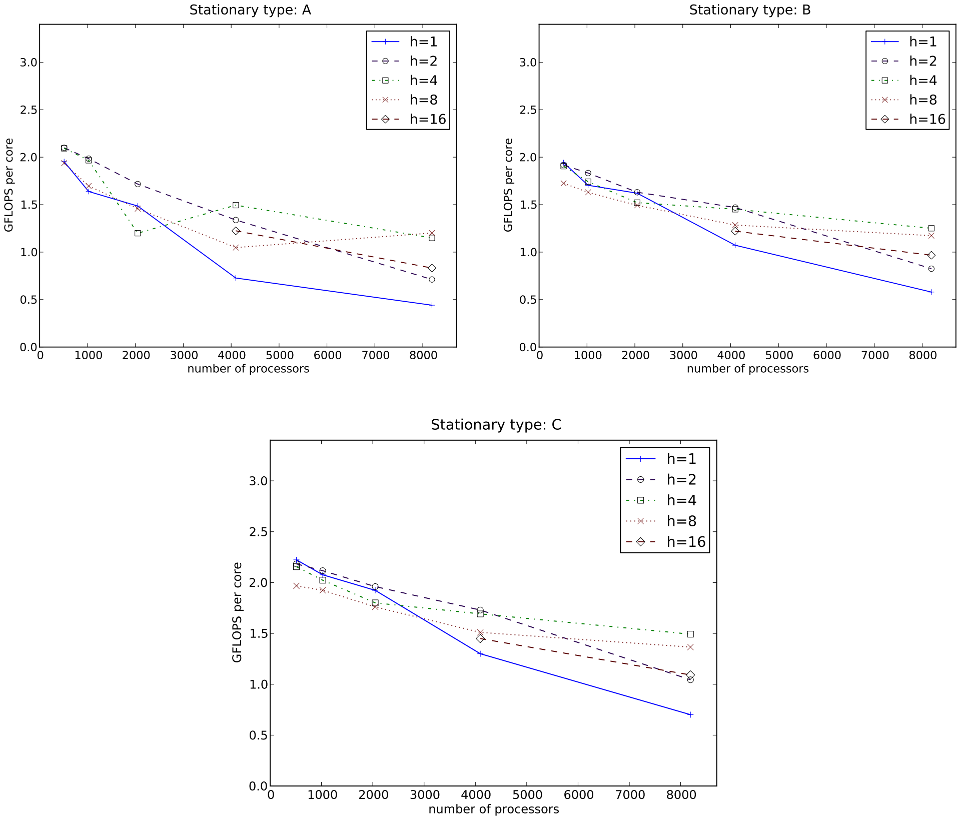


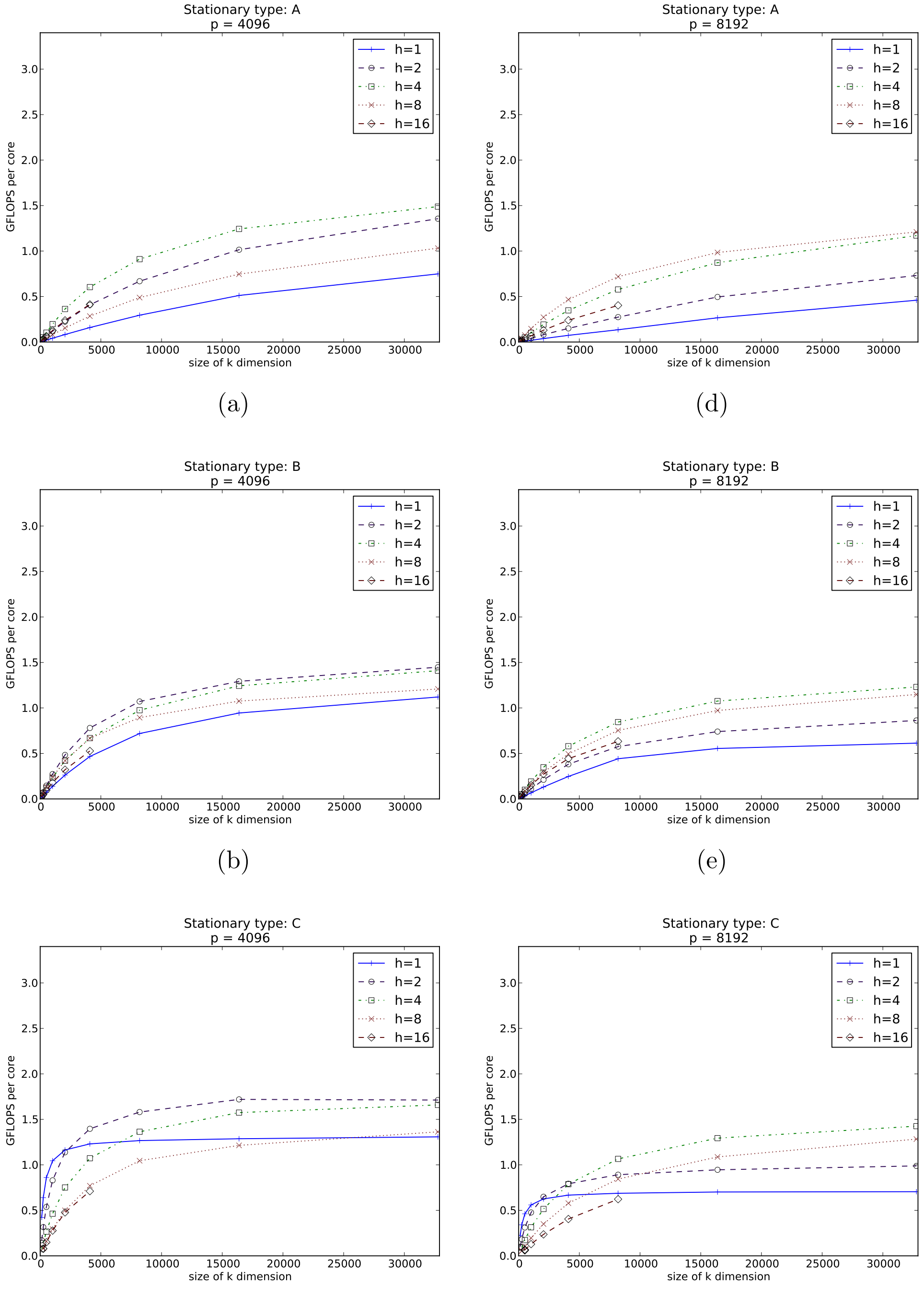
FIG. 8.1. *Performance of the different implementations when $m = n = k = 30,000$ and the number of nodes is varied.*

as was the number of layers, h. As a result, the $d_0 \times d_1$ mesh for a single layer was chosen so that $d_0 = d_1$ if p/h was a perfect square and $d_0 = d_1/2$ otherwise. The "zig-zagging" observed in some of the curves is attributed to this square vs. nonsquare choice of $d_0 \times d_1$. It would have been tempting to perform exhaustive experiments with various algorithmic block sizes and mesh configurations. However, the performance results were merely meant to verify that the insights of the previous sections have merit.

In our implementations, the eSUMMA3D-X algorithms utilize eSUMMA2D-X algorithms on each of the layers, where $\mathrm{X} \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$. As a result, the curve for eSUMMA3D-X with $h = 1$ is also the curve for the eSUMMA2D-X algorithm.

Figure 8.1 illustrates the benefits of the 3D algorithms. When the problem size is fixed, efficiency can inherently not be maintained. In other words, "strong" scaling is unattainable. Still, by increasing the number of layers, h, as the number of nodes, p, is increased, efficiency can be better maintained.

Figure 8.2: illustrates that the eSUMMA2D-C and eSUMMA3D-C algorithms attain high performance already when $m = n$ are relatively large and k is relatively small. This is not surprising: the eSUMMA2D-C algorithm already attains high performance



(a) (d)

(b) (e)

(c) (f)

Fig. 8.2. *Performance of the different implementations when $m = n = 30,000$ and k is varied. As expected, the stationary C algorithms ramp up to high performance faster than the other algorithms when k is small.*

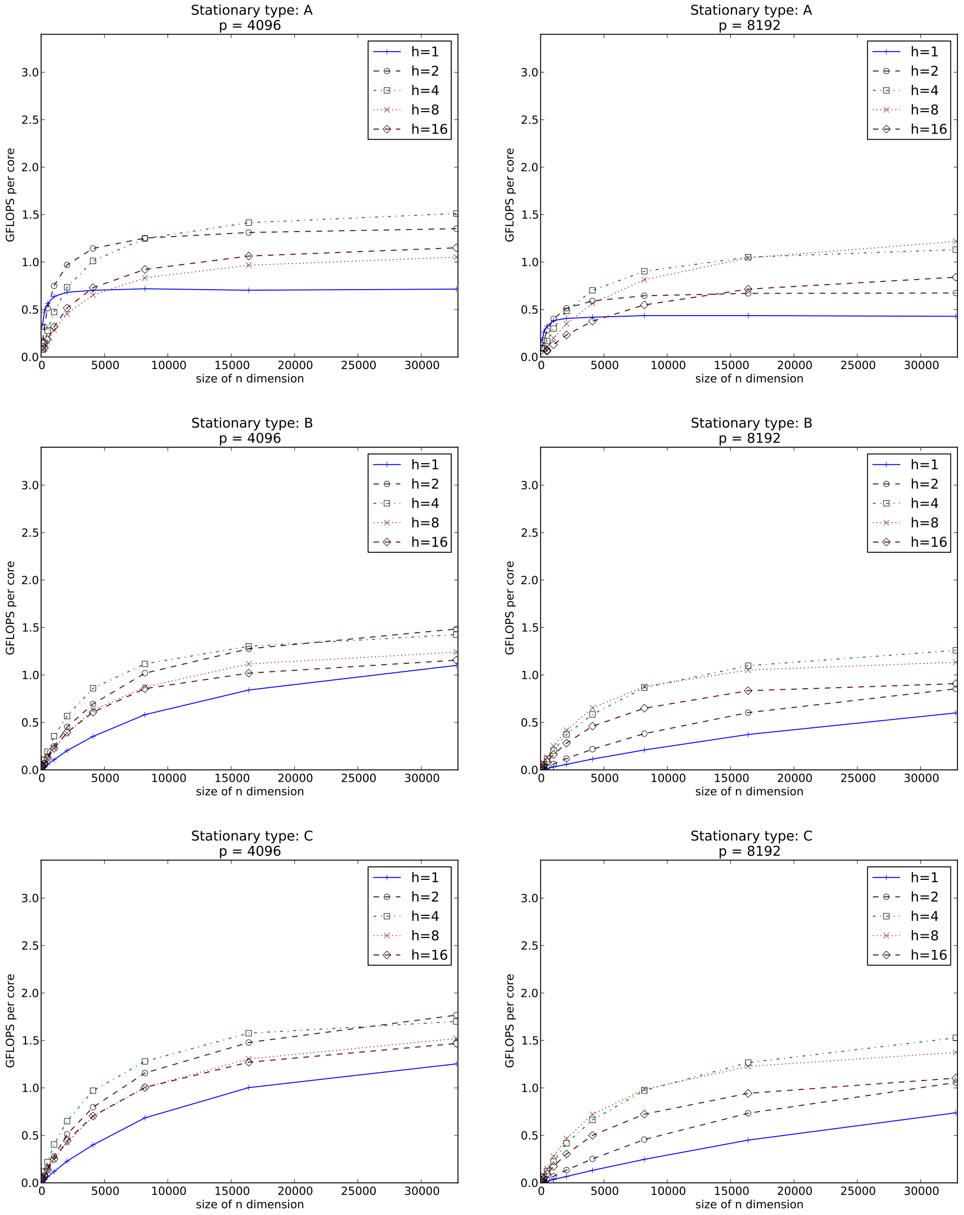


Fig. 8.3. *Performance of the different implementations when $m = k = 30,000$ and n is varied. As expected, the stationary A algorithms ramp up to high performance faster than the other algorithms when n is small.*

when k is small because the "large"' matrix C is not communicated and the local matrix-matrix multiplication can already attain high performance when the local k is small (if the local m and n are relatively large).

Figure 8.3 similarly illustrates that the eSUMMA2D-A and eSUMMA3D-A algorithms attain high performance already when $m = k$ are relatively large and n is rel-

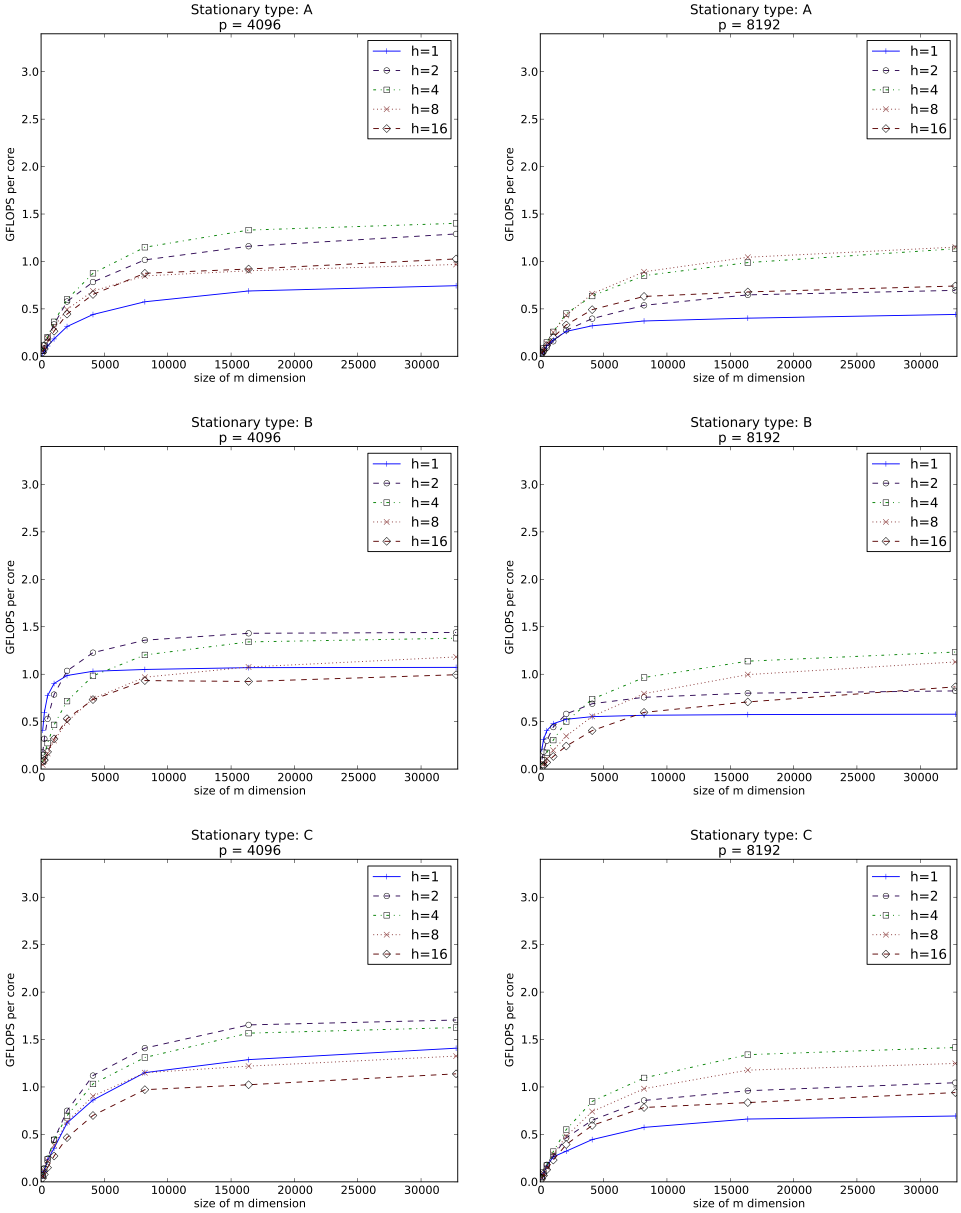


FIG. 8.4. *Performance of the different implementations when $n = k = 30,000$ and m is varied. As expected, the stationary B algorithms ramp up to high performance faster than the other algorithms when m is small.*

atively small and **Figure 8.4** illustrates that the eSUMMA2D-B and eSUMMA3D-B algorithms attain high performance already when $n = k$ are relatively large and m is relatively small.

Figure 8.2(c) vs. Figure 8.3(a) shows that the eSUMMA2D-A algorithm (Figure 8.3(a) with $h = 1$) asymptotes sooner than the eSUMMA2D-C algorithm (Figure 8.2(c) with $h = 1$). The primary reason for this is that it incurs more communication overhead. But as a result, increasing h benefits eSUMMA3D-A more in Figure 8.3(a) than does increasing h for eSUMMA3D-C in Figure 8.2(c). A similar observation can be made for eSUMMA2D-B and eSUMMA3D-B in Figure 8.4(b).

9. Extensions to Tensor Computations. Matrix computations and linear algebra are useful when the problem being modeled can be naturally described with up to two dimensions. The number of dimensions the object (linear or multi-linear) describes is often referred to as the *order* of the object. For problems naturally described as higher-order objects, tensor computations and multi-linear algebra are utilized.

As an example of tensor computations, the generalization of matrix-matrix multiplication to tensor computations is the tensor contraction. Not only is matrix-multiplication generalized in the respect that the objects represent a greater number of dimensions, but the number of dimensions involved in the summation or accumulation of the multiplication is generalized (up to all dimensions of a tensor can be involved in the summation of a tensor contraction), and the notion of a transposition of dimensions is generalized to incorporate the higher number of dimensions represented by each tensor.

A significant benefit of the notation introduced in this paper is that generalizing concepts to tensors and multi-linear algebra is relatively straight-forward. The notation used for an object's distribution is comprised of two pieces of information: how column-indices and how row-indices of the matrix object are distributed. To describe how a higher-order tensor is distributed, the notation needs only to extend to describe how the additional dimensions are distributed. Further, while this paper focuses predominately on processing grids that are two- and three-dimensional, modeling higher-order grids is straightforward. By design, we describe the shape of the grid as an array where each element is the size of the corresponding dimension of the grid. When targeting a higher-order grid, the array need only to be reshaped to match the order of the grid.

The challenge of formalizing how the different collective communications relate different distributions of tensors and how to systematically derive algorithms for tensor operations is beyond the scope of this paper but is a part of future work. Initial results of how the ideas in this paper are extended to the tensor contraction operation are given in the dissertation proposal of one of the authors [30].

10. Conclusion. We have given a systematic treatment of the parallel implementation of matrix-vector multiplication and rank-1 update. This motivates the vector and matrix distributions that underly PLAPACK and, more recently, Elemental. Based on this, we exposed a systematic approach for implementing parallel 2D matrix-matrix multiplication algorithms. With that in place, we then extended the observations to 3D algorithms.

The ideas in this paper primarily focus on aspects of distributed-memory architectures that utilize a bulk-synchronous communication model for network communication. The ideas presented do not preclude the use of many-core and/or GPU architectures within each node of such distributed-memory architectures. For distributed-memory architectures that are most appropriately modeled with bulk-synchronous communications, we hope that the ideas presented will allow others to investigate how to effectively utilize various on-node architectures. We recognize that future distributed-memory architectures may be better suited for more asynchronous communication models; however, it is important to understand when the ideas in this paper can be applied to better tune algorithms for given architectures.

We believe that sufficient detail has been given so that a reader can now easily extend our approach to alternative data distributions and/or alternative architectures. Throughout this paper, we have hinted how the ideas can be extended to the realm of tensor computation on higher-dimensional computing grids. A detailed presentation of how these ideas are extended will be given in future work. Another interesting future direction would be to analyze whether it would be worthwhile to use the proposed 3D parallelization, but with a different 2D SUMMA algorithms within each layer. For example, questions such as "would it be worthwhile to use the eSUMMA3D-C approach, but with a eSUMMA2D-A algorithm within each layer?" remain.

Acknowledgments. This research was partially sponsored by NSF grants OCI-0850750, CCF-0917167, ACI-1148125/1340293 and CCF-1320112, grants from Microsoft, and an unrestricted grant from Intel. Martin Schatz was partially supported by a Sandia Fellowship. Jack Poulson was partially supported by a fellowship from the Institute of Computational Engineering and Sciences. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357; early experiments were performed on the Texas Advanced Computing Center's Ranger Supercomputer.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

REFERENCES

[1] R. C. Agarwal, F. 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*, 38(6), 1994.

[2] R.C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39:39–5, 1995.

[3] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[4] G. Ballard. *Avoiding Communication in Dense Linear Algebra*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2013.

[5] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014.

[6] R. H. Bisseling. Parallel iterative solution of sparse linear systems on a transputer network. In A. E. Fincham and B. Ford, editors, *Parallel Computation*, volume 46 of *The Institute of Mathematics and its Applications Conference*, pages 253–271. Oxford University Press, Oxford, UK, 1993.

[7] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994.

[8] Jehoshua Bruck, Ching tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multi-port systems. In *IEEE Transactions on Parallel and Distributed Systems*, pages 298–309, 1997.

[9] Lynn Elliot Cannon. *A cellular computer to implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[10] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[11] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[12] J. Choi, D. W. Walker, and J. J.Dongarra. PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6, 1994.

[13] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick. Communication lower bounds and optimal algorithms for programs that reference arrays – Part 1. *ArXiv e-prints*, July 2013.

[14] C. Edwards, P. Geng, A. Patra, and R. van de Geijn. Parallel matrix distributions: have we been doing it all wrong? Technical Report TR-95-40, Department of Computer Sciences, The University of Texas at Austin, 1995.

[15] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice Hall, 1988.

[16] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3):12:1–12:25, May 2008.

[17] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.

[18] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[19] B. Hendrickson, R. Leland, and S. Plimpton. An efficient parallel algorithm for matrix-vector multiplication. Technical report, 1993.

[20] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput.*, 15(5):1201–1226, 1994.

[21] S. Huss-Lederman, E. Jacobson, and A. Tsao. Comparison of scalable parallel matrix multiplication libraries. In *Proceedings of the Scalable Parallel Libraries Conference*, 1993.

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

[23] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, September 2004.

[24] J. G. Lewis and R. A. van de Geijn. Implementing matrix-vector multiplication and conjugate gradient algorithms on distributed memory multicomputers. In *Proceedings of Supercomputing 1993*, 1993.

[25] J. Li, A. Skjellum, and R. D. Falgout. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency: Practice and Experience*, 9(5):345–389, May 1997.

[26] L. H. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Bull. Amer. Math. Soc.*, 55(10):961–962, 10 1949.

[27] F. W. McColl and A. Tiskin. Memory-efficient matrix multiplication in the bsp model. *Algorithmica*, 24(3):287–297.

[28] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2):13:1–13:24, February 2013.

[29] Jack Poulson, Robert van de Geijn, and Jeffrey Bennighof. Parallel algorithms for reducing the generalized hermitian-definite eigenvalue problem. FLAME Working Note #56. Technical Report TR-11-05, The University of Texas at Austin, Department of Computer Sciences, February 2011.

[30] Martin D. Schatz. Anatomy of parallel computation with tensors. Technical Report TR-13-21, Department of Computer Science, The University of Texas at Austin, 2013.

[31] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 90–109, Berlin, Heidelberg, 2011. Springer-Verlag.

[32] G. W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990.

[33] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

[34] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[35] Field G. Van Zee. `libflame`*: The Complete Reference*. `lulu.com`, 2009.

[36] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. The libflame library for dense matrix computations. *IEEE Computing in Science and Engineering*, 11(6):56–63, November 2009.

Appendix A. (Electronic supplement) Scalability of Matrix-vector Operations.

Here, we consider the scalability of various algorithms.

A.1. Weak scalability. A parallel algorithm is said to be weakly scalable when it can maintain efficiency as the number of nodes, p, increases.

More formally, let $T(n)$ and $T(n,p)$ be the cost (in time for execution) of a sequential and parallel algorithm (utilizing p nodes), respectively, when computing with a problem with a size parameterized by n. $n_{\max}(p)$ represents the largest problem that can fit in the combined memories of the p nodes, and the overhead $T^{+}(n,p)$ be given by

$$T^{+}(n,p) = T(n,p) - \frac{T(n)}{p}.$$

Then the efficiency of the parallel algorithm is given by

$$E(n,p) = \frac{T(n)}{pT(n,p)} = \frac{T(n)}{T(n) + pT^{+}(n,p)} = \frac{1}{1 + \frac{T^{+}(n,p)}{T(n)/p}}.$$

The efficiency attained by the largest problem that can be executed is then given

$$E(n_{\max}(p),p) = \frac{1}{1 + \frac{T^{+}(n_{\max}(p),p)}{T(n_{\max}(p))/p}}.$$

As long as

$$\lim_{p\to\infty} \frac{T^{+}(n_{\max}(p),p)}{T(n_{\max}(p))/p} \le R < \infty,$$

then the effective efficiency is bounded below away from 0, meaning that more nodes can be used effectively. In this case, the algorithm is said to be weakly scalable.

A.2. Weak scalability of parallel matrix-vector multiplication. Let us now analyze the weak scalability of some of the algorithms in Section 4.4.

Parallel $y := Ax$:. In (4.2), the cost of the parallel algorithm was approximated by

$$T_{y:=Ax}(m,n,d_0,d_1) = 2\frac{mn}{p}\gamma + \log_2(p)\alpha + \frac{d_0-1}{d_0}\frac{n}{d_1}\beta + \frac{d_1-1}{d_1}\frac{m}{d_0}\beta + \frac{d_1-1}{d_1}\frac{m}{d_0}\gamma.$$

Let us simplify the problem by assuming that $m = n$ so that $T_{y:=Ax}(n) = \frac{2n^2\gamma}{p}$ and

$$T_{y:=Ax}(n,d_0,d_1) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + \frac{d_0-1}{d_0}\frac{n}{d_1}\beta + \frac{d_1-1}{d_1}\frac{n}{d_0}\beta + \frac{d_1-1}{d_1}\frac{n}{d_0}\gamma.}_{{T^{+}}_{y:=Ax}(n,d_0,d_1)}$$

Now, let us assume that each node has memory to store a matrix with M entries. We will ignore memory needed for vectors, workspace, etc. in this analysis. Then $n_{\max}(p) = \sqrt{p}\sqrt{M}$ and

$$\frac{T^{+}(n_{\max}(p),p)}{T(n_{\max}(p))/p} = \frac{\log_2(p)\alpha + \frac{d_0-1}{d_0}\frac{n_{\max}}{d_1}\beta + \frac{d_1-1}{d_1}\frac{n_{\max}}{d_0}\beta + \frac{d_1-1}{d_1}\frac{n_{\max}}{d_0}\gamma}{2n_{\max}^2/p\gamma}$$

$$
\begin{aligned}
&= \frac{\log_2(p)\alpha + \frac{d_0-1}{p} n_{\max}\beta + \frac{d_1-1}{p} n_{\max}\beta + \frac{d_1-1}{p} n_{\max}\gamma}{2n_{\max}^2/p\gamma} \\
&= \frac{\log_2(p)\alpha + \frac{d_0-1}{\sqrt{p}}\sqrt{M}\beta + \frac{d_1-1}{\sqrt{p}}\sqrt{M}\beta + \frac{d_1-1}{\sqrt{p}}\sqrt{M}\gamma}{2M\gamma} \\
&= \log_2(p)\frac{1}{2M}\frac{\alpha}{\gamma} + \frac{d_0-1}{\sqrt{p}}\frac{1}{2\sqrt{M}}\frac{\beta}{\gamma} + \frac{d_1-1}{\sqrt{p}}\frac{1}{2\sqrt{M}}\frac{\beta}{\gamma} + \frac{d_1-1}{\sqrt{p}}\frac{1}{2\sqrt{M}}.
\end{aligned}
$$

We will use this formula to now analyze scalability.
Case 1: $d_0 \times d_1 = p \times 1$ Then

$$
\begin{aligned}
\frac{T^+(n_{\max}(p), p)}{T(n_{\max}(p))/p} &= \log_2(p)\frac{1}{2M}\frac{\alpha}{\gamma} + \frac{p-1}{\sqrt{p}}\frac{1}{2\sqrt{M}}\frac{\beta}{\gamma} \\
&\approx \log_2(p)\frac{1}{2M}\frac{\alpha}{\gamma} + \sqrt{p}\frac{1}{2\sqrt{M}}\frac{\beta}{\gamma}.
\end{aligned}
$$

Now, $\log_2(p)$ is generally regarded as a function that grows slowly enough that it can be treated almost like a constant. Not so for $\sqrt{p}$. Thus, even if $\log_2(p)$ is treated as a constant, $\lim_{p\to\infty} \frac{T^+(n_{\max}(p),p)}{T(n_{\max}(p))/p} \to \infty$ and eventually efficiency cannot be maintained. When $d_0 \times d_1 = p \times 1$, the proposed parallel matrix-vector multiply is not weakly scalable.

Case 2: $d_0 \times d_1 = 1 \times p$ We leave it as an exercise that the algorithm is not scalable in this case either.

The cases where $d_0 \times d_1 = 1 \times p$ or $d_0 \times d_1 = p \times 1$ can be viewed as partitioning the matrix by columns or rows, respectively, and assigning these in a round-robin fashion to the one-dimensional array of processors.
Case 3: $d_0 \times d_1 = \sqrt{p} \times \sqrt{p}$ Then

$$
\begin{aligned}
&\frac{T^+(n_{\max}(p), p)}{T(n_{\max}(p))/p} \\
&= \log_2(p)\frac{1}{2M}\frac{\alpha}{\gamma} + \frac{\sqrt{p}-1}{\sqrt{p}}\frac{1}{2\sqrt{M}}\frac{\beta}{\gamma} + \frac{\sqrt{p}-1}{\sqrt{p}}\frac{1}{2\sqrt{M}}\frac{\beta}{\gamma} + \frac{\sqrt{p}-1}{\sqrt{p}}\frac{1}{2\sqrt{M}} \\
&\approx \log_2(p)\frac{1}{2M}\frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}}\frac{\beta}{\gamma} + \frac{1}{2\sqrt{M}}\frac{\beta}{\gamma} + \frac{1}{2\sqrt{M}}.
\end{aligned}
$$

Now, if $\log_2(p)$ is treated like a constant, then $R(n_{\max}, \sqrt{p}, \sqrt{p})$ $\frac{T^+(n_{\max}(p),p)}{T(n_{\max}(p))/p}$ is a constant. Thus, the algorithm is considered weakly scalable, for practical purposes.

A.3. Other algorithms. All algorithms discussed, or derivable with the methods, in this paper can be analyzed similarly.