Parallel Matrix Multiplication: 2D and 3D

FLAME Working Note #62

Martin Schatz
Jack Poulson
Robert van de Geijn
The University of Texas at Austin
Austin, Texas 78712

June 11, 2012

Abstract

We describe an extension of the Scalable Universal Matrix Multiplication Algorithms (SUMMA) from 2D to 3D process grids; the underlying idea is to lower the communication volume through storing redundant copies of one or more matrices. While SUMMA was originally introduced for block-wise matrix distributions, so that most of its communication was within broadcasts, this paper focuses on element-wise matrix distributions, which lead to allgather-based algorithms. We begin by describing an allgather-based 2D SUMMA, describe its generalization to 3D process grids, and then discuss theoretical and experimental performance benefits of the new algorithms.

1 Introduction

The parallelization of dense matrix-matrix multiplication is a well-studied subject. Cannon's algorithm (sometimes called roll-roll-compute) dates back to 1969 [7], and Fox's algorithm (sometimes called broadcast-roll-compute) dates back to the 1980s [12]. Both suffer from a number of shortcomings:

- They assume that p processes are viewed as an $r \times c$ grid, with $r = c = \sqrt{p}$. Removing this constraint on r and c is nontrivial.
- 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` [22, 23], and their distributed-memory counterparts: ScaLAPACK [9], PLAPACK [21], and Elemental [17].
- They do not treat the other common cases of matrix-matrix multiplication: $C := A^T B + C$, $C := AB^T + C$, and $C := A^T B^T + C$.[1]

These shortcomings were overcome by SUMMA. The original paper [20] gives four algorithms:

- For $C := AB + C$, SUMMA casts the computation in terms of multiple rank-k updates, and algorithm already independently reported in [1]. 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-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.

[1] cf. the general form of `dgemm` [10]

- For $C := AB^T + C$, SUMMA casts the computation in terms of multiple matrix-panel 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 [14], 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 matrix-matrix multiplication algorithms all of which were incorporated into PLAPACK and Elemental, and some of which are supported by ScaLAPACK.

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 $\sqrt[3]{p} \times \sqrt[3]{p} \times \sqrt[3]{p}$ mesh, yielding a 3D algorithm [2] It was recently observed that the nodes could instead be viewed as an $r \times c \times h$ mesh, with $r = c$ but general h, building on Cannon's algorithm. This was labeled a 2.5D algorithm [19] since it was believed that h would always be chosen to be less than $\sqrt[3]{p}$ for theoretical optimality reasons. It was subsequently observed by the authors of [19] that, by using the SUMMA algorithm instead, at the expense of a slightly higher latency cost, the constraint that $r = c$ could be removed.

This paper attempts to give the most up-to-date treatment yet of practical parallel matrix-matrix multiplication algorithms. It does so by first discussing how collective communication is related to matrix-vector multiplication ($y := Ax$) and rank-1 update ($C := yx^T + C$). This is then used to link matrix distribution to a 2D mesh of nodes to the communications required for these matrix operations. This then leads naturally into various practical algorithms for all cases of matrix-matrix multiplication on 2D meshes of nodes. We finally generalizes the notion of 2.5D algorithms to yield a generally family of 3D algorithms that build on 2D stationary C, A, and B algorithms. We call this general class of algorithms General 3D algorithms, or G3D algorithms for short.

The paper makes a number of contributions:

- It systematically exposes the link between collective communication, matrix and vector distribution, and matrix-vector operations like matrix-vector multiplication and rank-1 update.
- It next shows how parallel algorithms for these matrix-vector operations systematically yield a family of matrix-matrix multiplication algorithms that view nodes as a 2D mesh.
- It exposes a set based notation for describing distribution of matrices and vectors that is used by the Elemental library for dense matrix computations on distributed memory architectures.
- It shows how the family of 2D algorithms can be used to build a family of algorithms that view the nodes as a 3D mesh.
- It provides performance results that illustrate the benefits of the family of 2D and 3D algorithms.

The exposition builds the reader's understanding so that he/she can easily customize the algorithms for other matrix distributions.

2 Of Matrix-Vector Operations and Distribution

In this section, we discuss how matrix and vector distribution 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.

2.1 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. [8], which presents collectives in a systematic way that dovetails perfectly with the present paper.

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

Figure 1: Collective communications considered in this paper.

Communication	Latency	Bandwidth	Computation	Cost used for analysis
Broadcast	$\lceil \log_2(p) \rceil \alpha$	$n\beta$	–	$\log_2(p)\alpha + n\beta$
Reduce(-to-one)	$\lceil \log_2(p) \rceil \alpha$	$n\beta$	$\frac{p-1}{p} n\gamma$	$\log_2(p)\alpha + n\beta + n\gamma$
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	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	$\frac{p-1}{p} n\gamma$	$\log_2(p)\alpha + \frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma$
Allreduce	$\lceil \log_2(p) \rceil \alpha$	$2\frac{p-1}{p} n\beta$	$\frac{p-1}{p} n\gamma$	$2\log_2(p)\alpha + 2\frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma$
All-to-all	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p} n\beta$	–	$\log_2(p)\alpha + \frac{p-1}{p} n\beta$

Figure 2: Lower bounds for the different components of communication cost. Conditions for the lower bounds given in [8] and [6]. 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.

To make this paper self-contained, Figure 1 (similar to Figure 1 in [8]) summarizes the collectives. In Figure 2 we summarize lower bounds on the cost of the collective communications, under basic assumptions explained in [8] (see [6] for analysis of all-to-all), and the cost expressions that we will use in our analyses.

2.2 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

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

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

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

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

	χ_0	χ_1	$\cdots$	χ_{n-1}
ψ_0	$\alpha_{0,0}$	$\alpha_{0,1}$	$\cdots$	$\alpha_{0,n-1}$
ψ_1	$\alpha_{1,0}$	$\alpha_{1,1}$	$\cdots$	$\alpha_{1,n-1}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
ψ_{m-1}	$\alpha_{m-1,0}$	$\alpha_{m-1,1}$	$\cdots$	$\alpha_{m-1,n-1}$

(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.

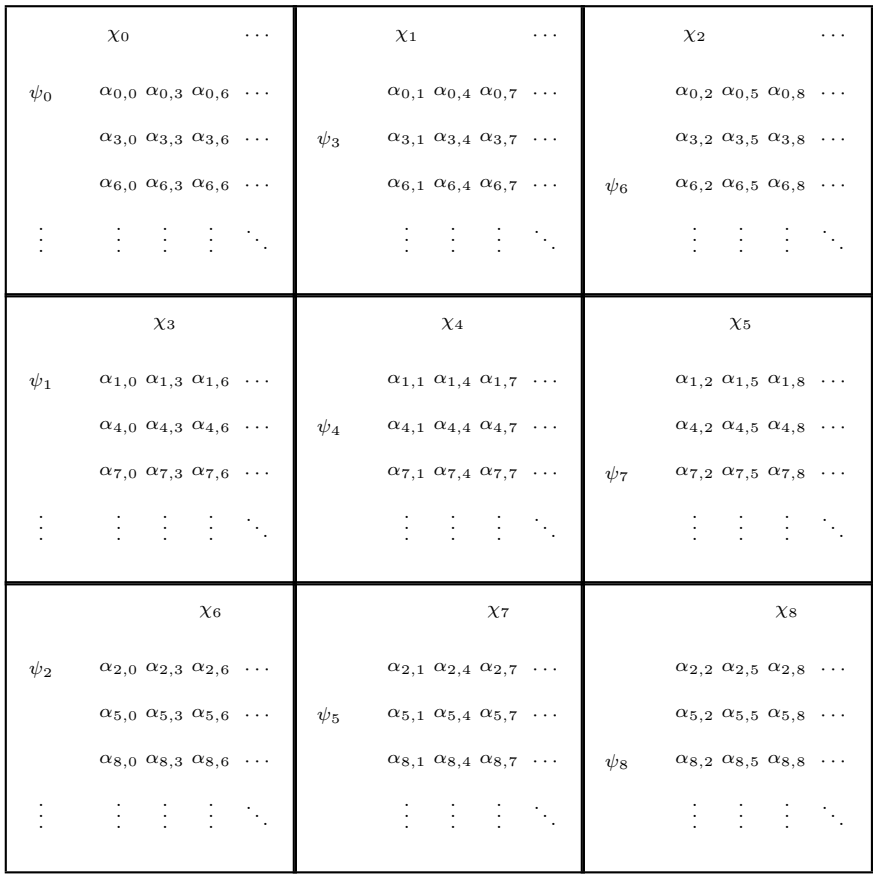

Figure 3: Distribution of A, x, and y within a 3×3 mesh. Notice that redistributing a column of A in the same manner as y requires simultaneous gathers within rows of nodes while redistributing a row of A consistently with x requires simultaneous gathers within columns of nodes. In the notation of Section 3, here the distribution of x and y are given by $x(V_R, *)$ and $y(V_C, *)$, respectively, and A by $A(M_C, M_R)$. While the presented mesh of nodes is square, none of the results depend on the mesh being square.

2.3 Two-Dimensional Elemental Cyclic Distribution

It is well established that scalable implementations of dense linear algebra operations require nodes to be logically viewed as a two-dimensional mesh. 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.

Denoting the number of nodes by p, an $r \times c$ mesh must be chosen such that $p = rc$.

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

$$A(s\!:\!r\!:\!m\!-\!1, t\!:\!c\!:\!n\!-\!1) = \begin{pmatrix} \alpha_{s,t} & \alpha_{s,t+c} & \cdots \\ \alpha_{s+r,t} & \alpha_{s+r,t+c} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix},$$

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 3.

Column-major vector distribution. A *column-major* vector distribution views the $r \times c$ mesh of nodes as a linear array of p nodes, numbered in *column-major* order. A vector is distributed with this

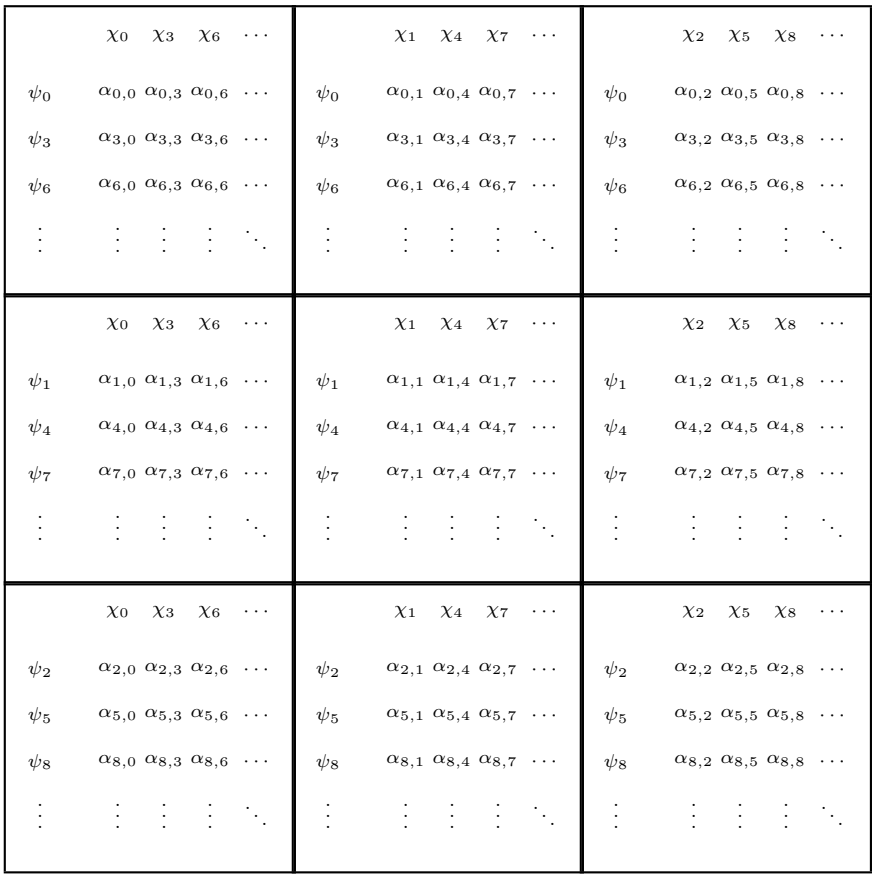

Figure 4: Vectors x and y respectively redistributed as row-projected and column-projected vectors. The column-projected vector $y(M_C, *)$ here is to be used to compute local results that will become contributions to a column vector $y(V_C, *)$ which will result from adding these local contributions within rows of nodes. By comparing and contrasting this figure with Figure 3 it becomes obvious that redistributing $x(V_R)$ to $x(M_R)$ requires an allgather within colums of nodes while $y(V_C)$ results from scattering $y(M_C)$ within process rows.

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 $(i \bmod r, (i/r) \bmod c)$, where / denotes integer division. Or, equivalently in matlab-like notation, node (s,t) stores subvector $y(u : p : m-1)$, where $u(s,t) = s + tr$ equals the rank of node (s,t) when the nodes are viewed as a one-dimensional array, indexed in column-major order. The distribution of y is illustrated in Figure 3.

Row-major vector distribution Similarly, a *row-major* vector distribution views the $r \times c$ mesh of nodes as a linear array of p nodes, numbered in *row-major* order. The vector is then assigned to this linear array of nodes in a round-robin fashion, one element at a time.

In other words, consider vector x. Its element χ_j is assigned to node $(j \bmod c, (j/c) \bmod r)$. Or, equivalently, node (s,t) stores subvector $x(v : p : n-1)$, where $v = sc + t$ equals the rank of node (s,t) when the nodes are viewed as a one-dimensional array, indexed in row-major order. The distribution of x is illustrated in Figure 3.

At this point, we suggest comparing Eqn. 1 with Figure 3.

2.4 Parallelizing matrix-vector operations

In the following discussion, we assume that A, x, and y are distributed as discussed above[2] .

Computing $y := Ax$: The relation between these 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 2.2. An examination of Figure 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. Next, each node computes the partial contribution to vector y with its local matrix and copy of x. Thus, in Figure 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

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

We simplify this further to

$$
2\frac{mn}{p}\gamma + \log_2(p)\alpha + \frac{r-1}{r}\frac{n}{c}\beta + \frac{c-1}{c}\frac{m}{r}\beta + \frac{c-1}{c}\frac{m}{r}\gamma
$$

since the load imbalance contributes a cost similar to that of the communication[3]. In Appendix A we use these estimates to show that this parallel matrix-vector multiplication is weakly scalable if r/c is kept constant, but not if $r = p$ or $c = p$.

Computing $x := A^T y$: 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,

$$
\begin{array}{c|c|c|c}
\chi_0 = & \chi_1 = & \cdots & \chi_{m-1} = \\
\alpha_{0,0}\psi_0 + & \alpha_{0,1}\psi_0 + & \cdots & \alpha_{0,n-1}\psi_0 + \\
\alpha_{1,0}\psi_1 + & \alpha_{1,1}\psi_1 + & \cdots & \alpha_{1,n-1}\psi_1 + \\
\vdots & \vdots & & \vdots \\
\alpha_{n-1,0}\psi_{n-1} & \alpha_{n-1,1}\psi_{n-1} & \cdots & \alpha_{n-1,n-1}\psi_{n-1}
\end{array}
\tag{2}
$$

An examination of Eqn. 2 and Figure 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. Next, each node computes the partial contribution to vector x with its local matrix and copy of y. Thus, in Figure 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.

[2]We suggest the reader print copies of Figures 3 and 4 for easy referral while reading the rest of this section.

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

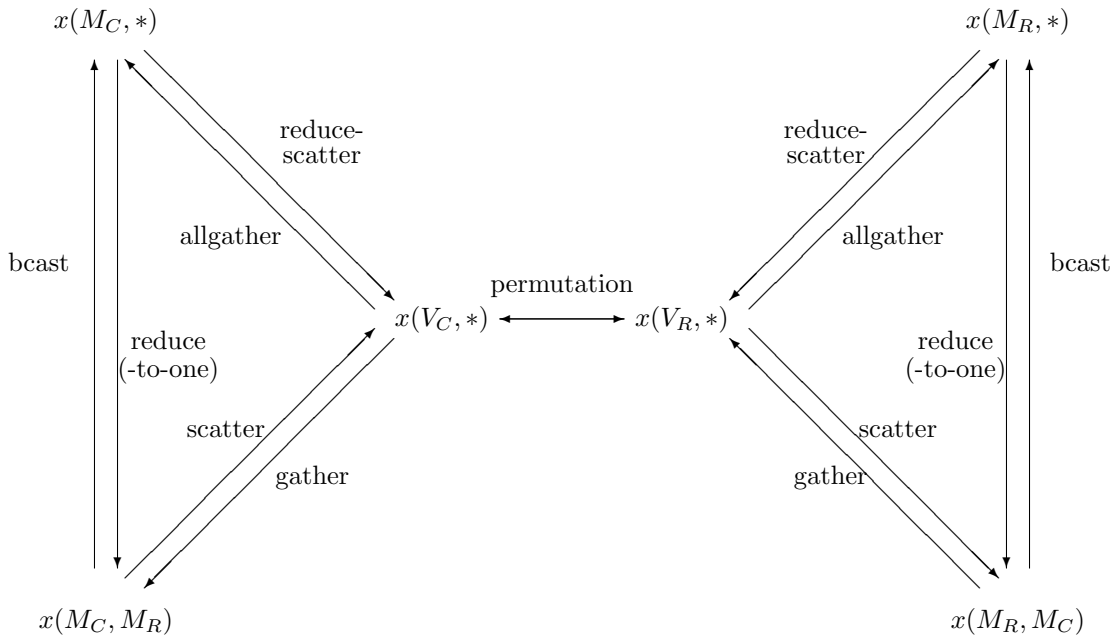

Figure 5: Summary of the communication patterns for redistributing a vector x. For instance, a method for redistributing x from a matrix column to a matrix row is found by tracing from the bottom-left to the bottom-right of the diagram.

Exercise 1. *Give a cost estimate for the above parallel algorithm for computing $x = A^T y$. What can you say about the weak scalability of the algorithm?*
Exercise 2. *What additional communication needs to be added to the above approaches if one wants to compute $y = A^T x$ and $x = Ay$? How does this affect weak scalability?*

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 3 and 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.
Exercise 3. *Analyze the weak scalability for the proposed parallel algorithm that computes $A = yx^T + A$.*
Exercise 4. *What additional communication needs to be added to the above approaches if one wants to compute $A = xy^T + A$?*

3 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 (PBMD) [11] 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 [4, 5], and, around the same time, in [16].

3.1 Vector distribution

The partitioning of indices among processes is fundamental to our coming discussions, and the following formalism simplifies our notation.

Definition 1 (Partition of $\mathbb{N}$). A collection of k sets $\{\mathcal{S}_0, \ldots, \mathcal{S}_{k-1}\}$ is said to be a *partition* of the natural numbers (including zero), $\mathbb{N}$, if $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset$ when $i \neq j$, and $\cup_{j=0}^{k-1} \mathcal{S}_j = \mathbb{N}$.

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 2 (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.

Note that the above notation is similar to MATLAB notation, but the members of $\mathcal{S}$ and $\mathcal{T}$ need not be constrained by the sizes of x and A. We illustrate this idea with simple examples:

Example 3. *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* $x(\mathcal{S}) = \begin{pmatrix} \chi_0 \\ \chi_2 \end{pmatrix}$ *and* $A(\mathcal{S}, \mathcal{T}) = \begin{pmatrix} \alpha_{0,1} & \alpha_{0,3} \\ \alpha_{2,1} & \alpha_{2,3} \\ \alpha_{4,1} & \alpha_{4,3} \end{pmatrix}$.

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

Definition 4 (Column-major vector distribution). Given $p = rc$ processes configured into a logical $r \times c$ grid, we say that the tuple $V_C = (V_C^0, V_C^1, ..., V_C^{p-1})$ is a *column-major vector distribution* if there exists some alignment parameter $\sigma \in \{0, ..., p-1\}$ such that, for any given grid positions $s \in \{0, ..., r-1\}$ and $t \in \{0, ..., c-1\}$,

$$V_C^{s+tr} = \{N \in \mathbb{N} : N \equiv s + tr + \sigma \pmod{p}\}. \tag{3}$$

The shorthand $y(V_C)$ will refer to the vector y distributed such that process (s, t) stores $y(V_C^{s+tr})$.

Definition 5 (Row-major vector distribution). Similarly, we call the tuple $V_R = (V_R^0, V_R^1, ..., V_R^{p-1})$ a *row-major vector distribution* if there exists some alignment parameter σ such that, for every grid position (s, t), where $0 \leq s < r$ and $0 \leq t < c$,

$$V_R^{t+sc} = \{N \in \mathbb{N} : N \equiv t + sc + \sigma \pmod{p}\}. \tag{4}$$

The shorthand $y(V_R)$ will refer to the vector y distributed such that process (s, t) stores $y(V_R^{t+sc})$.

Remark 6. The members of any column-major vector distribution, V_C, or row-major vector distribution, V_R, form a partition of $\mathbb{N}$.

Remark 7. The names *column-major vector distribution* and *row-major vector distribution* come from the fact that the mappings $(s,t) \mapsto s + tr$ and $(s,t) \mapsto t + sc$ respectively label the $r \times c$ grid with a column-major and row-major ordering.

Remark 8. A comparison of (3) and (4) reveals that to redistribute $y(V_C)$ to $y(V_R)$, and vise versa, requires a permutation communication (simultaneous point-to-point communications).

Remark 9. In the preceding discussions, our definitions of V_C and V_R distributions 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 matrix is owned by the process in the top-left of the process grid.

3.2 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 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 with 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 M_C^s of row indices assigned to process row s?" and "What is the set M_R^t or column indices assigned to process column t?"

Definition 10. Let $M_C^s = \bigcup_{t=0}^{c-1} V_C^{s,t}$ and $M_R^t = \bigcup_{t=0}^{r-1} V_R^{s,t}$. Given matrix A, $A(M_C^s, M_R^t)$ denotes the submatrix of A with row indices in the set M_C^s and column indices in M_R^t. Finally, $A(M_C, M_R)$ denotes the distribution of A that assigns $A(M_C^s, M_R^t)$ to process (s,t).

We say that (M_C, M_R) is *induced* by (V_C, V_R) because the process to which $\alpha_{i,j}$ is assigned is determined by the row of processes, s, to which y_i is assigned and the column of processes, t, 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. The above definition lies at the heart of our communication scheme.

3.3 Vector duplication

Two vector distributions, encountered in Section 2.4, 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 $x(M_C)$ or, viewing x as a $n \times 1$ matrix, $x(M_C, *)$. The vector y, duplicated so as to store local contributions for $y = Ax$, can be specified as $y(M_R)$ or, viewing y as a $n \times 1$ matrix, $y(M_R, *)$. Here the $*$ should be intrepreted as "all indices". In other words, $* \equiv \mathbb{N}$.

3.4 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 4. This motivates us to briefly discuss the different communications needed to redistributed vectors to and from columns and rows. In our discussion, it will help to refer back to Figures 3 and 4.

Algorithm: $y := Ax$ (GEMV)	Comments
$x(M_R, *) \leftarrow x(V_R, *)$	Redistribute x (allgather in columns)
$y^{(t)}(M_C^s, *) := A(M_C^s, M_R^t)\, x(M_R^t, *)$	Local matrix-vector multiply
$y(V_C, *) := \widehat{\sum}_t y^{(t)}(M_C, *)$	Sum contributions (reduce-scatter in rows)

Figure 6: Parallel algorithm for computing $y := Ax$.

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

Figure 7: Parallel algorithm for computing $A := A + xy^T$.

Column to and from column-major vector Consider Figure 3 and let a_j be a typical column in A. It exists within one single process column. Redistributing $a_j(M_C, M_R)$ to $y(V_C)$ requires simultaneous scatters within process rows. Redistributing $y(V_C)$ to $a_j(M_C, M_R)$ requires simultaneous gathers within process rows.

Column to and from row-major vector Redistributing $a_j(M_C, M_R)$ to $x(V_R)$ can be accomplished by first redistributing to $y(V_C)$ (simultaneous scatters within rows) followed by a redistribution of $y(V_C)$ to $x(V_R)$ (a permutation). Redistributing $x(V_R)$ to $a_j(M_C, M_R)$ reverses these communications.

Column to and from column projected vector Redistributing $a_j(M_C, M_R)$ to $a_j(M_C, *)$ (duplicated y in Figure 4) can be accomplished by first redistributing to $y(V_C)$ (simultaneous scatters within rows) followed by a redistribution of $y(V_C)$ to $y(M_C, *)$ (simultaneous allgathers within rows). However, recognize that a scatter followed by an allgather is equivalent to a broadcast. Thus, redistributing $a_j(M_C, M_R)$ to $a_j(M_C, *)$ can be more directly accomplished by broadcasting within rows. Similarly, summing duplicated vectors $y(M_C, *)$ leaving the result as $a_j(M_C, M_R)$ (a column in A) can either be accomplished by first summing them into $y(V_C)$ (reduce-scatters within rows) followed by a redistribution to $a_j(M_C, M_R)$. 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 performancing various matrix-vector multiplications or rank-1 updates, with vectors, columns, or rows as input, in Figure 5.

3.5 Parallelizing matrix-vector operations (revisited)

We now show how the notation discussed in the previous section pays off when describing algorithms for matrix-vector operations.

Assume that A, x, and y are distributed as $A(M_C, M_R)$, $x(V_R, *)$, and $y(V_C, *)$, respectively. Algorithms for computing $y := Ax$ and $A := A + xy^T$ are given in Figures 6 and 7.

Algorithm: $\hat{c}_i := Ab_j$ (GEMV)	Comments
$x(M_R, *) \leftarrow b_j(M_C, M_R)$	Redistribute b_j as a PBMD. This can be implemented as
	$x(V_C, *) \leftarrow b_j(M_C, M_R)$ (scatter in rows)
	$x(V_R, *) \leftarrow x(V_C, *)$ (permutation)
	$x(M_R, *) \leftarrow x(V_R, *)$ (allgather in columns).
$y^{(t)}(M_C^s, *) := A(M_C^s, M_R^t)\, x(M_R^t, *)$	Local matrix-vector multiply
$\hat{c}_i(M_C, M_R) := \widehat{\sum}_t y^{(t)}(M_C, *)$	Sum contributions. This can be implemented as
	$y(V_C, *) := \widehat{\sum}_t y^{(t)}(M_C, *)$ (reduce-scatter in rows)
	$y(V_R, *) \leftarrow y(V_C, *)$ (permutation)
	$\hat{c}_i(M_C, M_R) \leftarrow y(V_R, *)$ (gather in rows)

Figure 8: 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.

Exercise 5. *Assume that A, x, and y are distributed as $A(M_C, M_R)$, $x(V_R, *)$, and $y(V_C, *)$, respectively. Propose parallel algorithms for*

1. $x = A^T y$,
2. $y = A^T x$, *and*
3. $A = A + yx^T$.

The discussion in Section 3.4 provides the insights to generalize these parallel matrix-vector operations to the cases where the vectors are rows and/or columns of matrices. For example, in Figure 8 we show how to compute a row of a matrix C, $\hat{c}_i$, as the product of a matrix A times the column of a matrix B, b_j.

4 Elemental SUMMA: 2D algorithms (eSUMMA2D)

We have now arrived at the point where we can discuss parallel matrix-matrix multiplication on a $r \times c$ mesh, with $p = rc$. In our discussion, we will assume an elemental distribution: $\mathcal{V}_j = \{j, j + p, j + 2p, \cdots\}$, but the ideas clearly generalize.

To fully understand how to attain high performance on a single processor, the reader should familiarize him/herself with [13].

4.1 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[4]. We call this a rank-k update or panel-panel multiplication [13]. We will assume the distributions $C(M_C, M_R)$, $A(M_C, M_R)$, and $B(M_C, M_R)$. Partition

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

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 $C := AB + C$:

[4]There is an algorithmic block size, b_{alg}, for which a local rank-k update achieves peak performance [13]. Think of k as being that algorithmic block size for now.

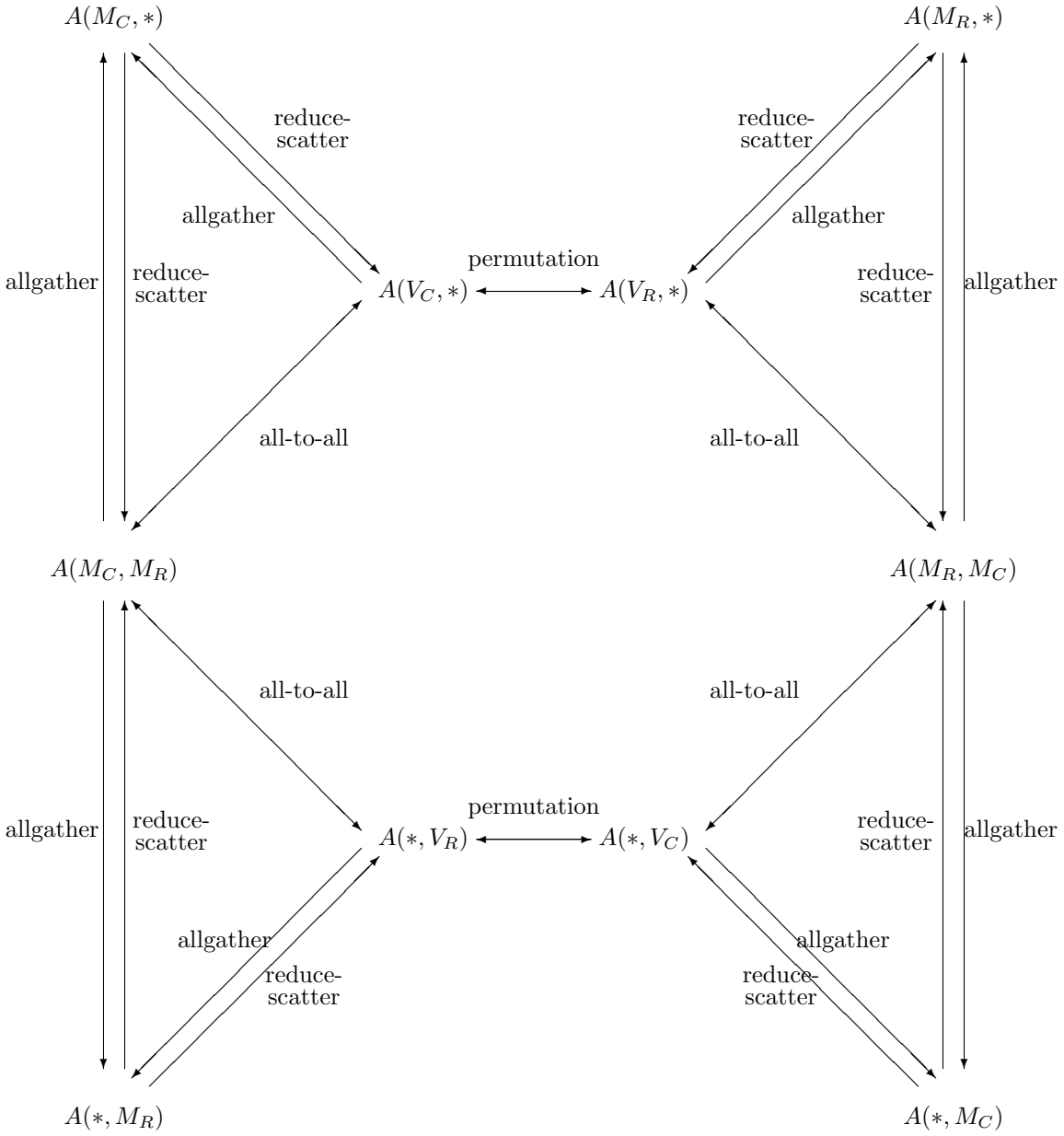

Figure 9: Summary of the communication patterns for redistributing a matrix A.

for $p = 0, \ldots, k-1$	
$a_p(M_C, *) \leftarrow a_p(M_C, M_R)$	(broadcasts within rows)
$b_p^T(M_R, *) \leftarrow \hat{b}_p^T(M_C, M_R)$	(broadcasts within columns)
$C(M_C, M_R) := C(M_C, M_R) + a_p(M_C, *)\hat{b}_p^T(M_R, *)$	(local rank-1 updates)
endfor	

While Section 3.5 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

Algorithm: $C := \text{GEMM\_C}(C, A, B)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_L & A_R \end{array} \right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right)$

where A_L has 0 columns, B_T has 0 rows

while $n(A_L) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_L & A_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array} \right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array} \right)$$

where A_1 **has** b **columns,** B_1 **has** b **rows**

$A_1(M_C, *) \leftarrow A_1(M_C, M_R)$
$B_1(*, M_R) \leftarrow B_1(M_C, M_R)$
$C(M_C, M_R) := C(M_C, M_R) + A_1(M_C, *) \;\; B_1(*, M_R)$

Continue with

$$\left(\begin{array}{c|c} A_L & A_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array} \right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array} \right)$$

endwhile

Figure 10: Stationary C algorithm for computing $C := AB + C$.

for $p = 0, \ldots, k-1$	
$a_p(M_C, *) \leftarrow a_p(M_C, M_R)$	(broadcasts within rows)
endfor	
for $p = 0, \ldots, k-1$	
$b_p^T(M_R, *) \leftarrow \hat{b}_p^T(M_C, M_R)$	(broadcasts within columns)
endfor	
for $p = 0, \ldots, k-1$	
$C(M_C, M_R) := C(M_C, M_R) + a_p(M_C, *)\hat{b}_p^T(M_R, *)$	(local rank-1 updates)
endfor	

and finally, equivalently,

$A(M_C, *) \leftarrow A(M_C, M_R)$	(allgather within rows)
$B(*, M_R) \leftarrow B(M_C, M_R)$	(allgather within columns)
$C(M_C, M_R) := C(M_C, M_R) + A(M_C, *) \;\; B(*, M_R)$	(local rank-k update)

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) $A(M_C, *) \leftarrow A(M_C, M_R)$, within each row broadcasts k columns of A from different roots: an allgather if elemental distribution is assumed! Similarly $B(*, M_R) \leftarrow B(M_C, M_R)$, within each column broadcasts k rows of B from different roots: another allgather if elemental distribution is assumed!

Based on this observation, the SUMMA algorithm can be expressed as a loop around such rank-k updates, as given in Figure 10[5]. Notice that, if an elemental distribution is assumed, the SUMMA 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

[5]We use FLAME notation to express the algorithm, which has been used in our papers for more than a decade [15].

a loop around rank-k updates is that is reduces the required local workspace at the expense of an increase only in the α term.

Remark 11. We label this algorithm *eSUMMA2D-C*, an elemental SUMMA algorithm targeting a 2D mesh of nodes, stationary C variant. It is not hard to extend the insights to non-elemental distributions (ala ScaLAPACK or PLAPACK).

An approximate cost for the described algorithm is given by

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

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 r and c.

It is not hard to see that 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 $r = c$, for arbitrary k.

4.2 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 [13]. We again assume that the matrices are distributed as $C(M_C, M_R)$, $A(M_C, M_R)$, and $B(M_C, M_R)$. 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 $j = 0, \ldots, n-1$	
$b_j(V_C, *) \leftarrow b_j(M_C, M_R)$	(scatters within rows)
$b_j(V_R, *) \leftarrow b_j(V_C, *)$	(permutation)
$b_j(M_R, *) \leftarrow b_j(V_R, *)$	(allgathers within cols)
$c_j(M_C, *) := A(M_C, M_R)b_j(M_R, *)$	(local matrix-vector multiplications)
$c_j(M_C, M_R) \leftarrow \widehat{\sum} c_j(M_C, *)$	(reduce-to-one within rows)
endfor	

While Section 3.5 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

Algorithm: $C := \text{GEMM\_A}(C, A, B)$

Partition $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array}\right)$, $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array}\right)$
where C_L and B_L have 0 columns

while $n(C_L) < n(C)$ **do**
Determine block size b
Repartition
$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$, $\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$
where C_1 **and** B_1 **have** b **columns**

$B_1(*, M_R) \leftarrow B_1(M_C, M_R)$
$C_1^{(t)}(M_C^s, *) := A(M_C^s, M_R^t) B_1(M_R^t, *)$
$C_1(M_C, M_R) := \widehat{\sum}_t C_1^{(t)}(M_C, *)$

Continue with
$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right)$, $\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$
endwhile

Figure 11: Stationary A algorithm for computing $C := AB + C$.

for $j = 0, \ldots, n-1$	
$b_j(V_C, *) \leftarrow b_j(M_C, M_R)$	(scatters within rows)
endfor	
for $j = 0, \ldots, n-1$	
$b_j(V_R, *) \leftarrow b_j(V_C, *)$	(permutation)
endfor	
for $j = 0, \ldots, n-1$	
$b_j(M_R, *) \leftarrow b_j(V_R, *)$	(allgathers within columns)
endfor	
for $j = 0, \ldots, n-1$	
$c_j(M_C, *) := A(M_C, M_R) b_j(M_R, *)$	(local matrix-vector multiplications)
endfor	
for $j = 0, \ldots, n-1$	
$c_j(M_C, M_R) \leftarrow \widehat{\sum} c_j(M_C, *)$	(simultaneous reduce-to-one within rows)
endfor	

and finally, equivalently,

$B(*, M_R) \leftarrow B(M_C, M_R)$	(all-to-all within rows, permutation, allgather within columns)
$C(M_C, *) := AB(*, M_R) + C(M_C, *)$	(simultaneous local matrix multiplications)
$C(M_C, M_R) \leftarrow \widehat{\sum} C(M_C, *)$	(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 arbitray n can now be expressed as a loop around such parallel matrix-panel multiplies, given in Figure 11.

An approximate cost for the described algorithm is given by

$$T_{\text{eSUMMA2D-A}}(m, n, k, r, c) =$$

$\frac{n}{b_{\text{alg}}}\log_2(c)\alpha + \frac{c-1}{c}\frac{k}{r}\frac{n}{c}\beta$ (all-to-all within rows)
$+\frac{n}{b_{\text{alg}}}\alpha + \frac{n}{c}\frac{k}{r}\beta$ (permutation)
$+\frac{n}{b_{\text{alg}}}\log_2(r)\alpha + \frac{r-1}{r}\frac{nk}{c}\beta$ (allgather within columns)
$+\frac{2mnk}{p}\gamma$ (simultaneous local matrix-panel multiplications)
$+\frac{n}{b_{\text{alg}}}\log_2(c)\alpha + \frac{c-1}{c}\frac{mk}{r}\beta + \frac{c-1}{c}\frac{mk}{r}\gamma$ (reduce-scatter within rows)

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 $\log_2(c)$ [6], 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 r and c.

While the overhead is clearly greater than that of the eSUMMA2D-C algorithm, when $m = n = k$ the overhead is comparable to that of that algorithm; so the scalability results are similar. 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.

4.3 Communicating submatrices

In Figure 9 we show the collective communications required to redistribute submatrices from one distribution to another and the collective communications required to implement them.

4.4 Exercises

Exercise 6. *Propose and analyze eSUMMA2D-C algorithms for $C := A^TB + C$, $C := AB^T + C$, and $C := A^TB^T + C$.*
Exercise 7. *Design and analyze stationary A algorithms for $C := A^TB + C$, $C := AB^T + C$, and $C := A^TB^T + C$.*
Exercise 8. *Design and analyze stationary B algorithms for $C := AB + C$, $C := A^TB + C$, $C := AB^T + C$, and $C := A^TB^T + C$.*

5 Elemental SUMMA: 3D algorithms (eSUMMA3D)

We now view the p processors as forming a $r \times c \times h$ mesh, which one should visualize as h stacked layers, where each layer consists of a $r \times c$ mesh. The extra dimension will be used to gain an extra level of parallelism, which reduces the overhead of the 2D SUMMA algorithms at the expense of communications between the layers. The approach on how 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 that come from using those algorithms), or a more traditional distribution for which SUMMA can be used (pretty much any Cartesian distribution).

5.1 3D stationary C algorithms (eSUMMA3D-C)

Partition, conformally, A and B so that

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

where A_p and B_p have approximately k/h columns and rows, respectively. Then

$$C + AB = \underbrace{(C + A_0B_0)}_{\text{by layer 0}} + \underbrace{(0 + A_1B_1)}_{\text{by layer 1}} + \cdots + \underbrace{(0 + A_{h-1}B_{h-1}).}_{\text{by layer h-1}}$$

This suggests the following 3D algorithm:

- Duplicate C to each of the layers, initializing the duplicates by layeres 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 recieves A_H and B_H. This means that all processors $(I, J, 0)$ simultaneously scatter approximately $(m+n)k/(rc)$ data to processors $(I, J, 0)$ through $(I, J, h-1)$. The cost of such a scatter can be approximated by

$$\log_2(h)\alpha + \frac{h-1}{h}\frac{(m+n)k}{rc}\beta = \log_2(h)\alpha + \frac{(h-1)(m+n)k}{p}\beta. \tag{5}$$

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

$$2\frac{mnk}{p}\gamma + \underbrace{\frac{k}{hb_{\text{alg}}}\left(\log_2(p) - \log_2(h)\right)\alpha + \frac{(c-1)mk}{p}\beta + \frac{(r-1)nk}{p}\beta}_{T^{+}_{\text{eSUMMA2D-C}}(m,n,k/h,r,c)} \tag{6}$$

- 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

$$\log_2(h)\alpha + \frac{mn}{rc}\beta + \frac{mn}{rc}\gamma = \log_2(h)\alpha + \frac{mnh}{p}\beta + \frac{mnh}{p}\gamma. \tag{7}$$

Thus, an estimate for the total cost of this eSUMMA3D-C algorithm for this case of gemm results from adding (5)–(7).

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

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

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 $r = c$, and that the matrices is perfectly balanced among the $r \times r$ 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.

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.

5.2 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 SUMMA algorithm for $C := AB + C$.

Let us describe a 3D algorithm, with a $r \times c \times h$ mesh, again viewed as h layers. If we partition, conformally, C and B so that

$$C = \left(\begin{array}{c|c|c} C_0 & \cdots & C_{h-1} \end{array} \right) \quad \text{and} \quad B = \left(\begin{array}{c|c|c} B_0 & \cdots & B_{h-1} \end{array} \right),$$

then

$$\left(\begin{array}{c|c|c} \underbrace{C_0 := C_0 + AB_0}_{\text{by layer 0}} & \cdots & \underbrace{C_{h-1} := C_{h-1} + AB_{h-1}}_{\text{by layer } h-1} \end{array} \right).$$

This suggests the following 3D algorithm:

- Duplicate (broadcast) A to each of the layers, initializing the duplicates meshes 1 through $h-1$ to zero. If matrix A is perfectly balanced among the processors, a lower bound for this is given by

  $$\log_2(h)\alpha + \frac{mk}{rc}\beta$$

  Simple algorithms that achieve within a factor two of this lower bound are known.

- 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 $(mn + nk)/(rc)$ data to processors $(I, J, 0)$ through $(I, J, h-1)$. The cost of such a scatter can be approximated by

  $$\log_2(h)\alpha + \frac{h-1}{h}\frac{(m+k)n}{cr}\beta = \log_2(h)\alpha + \frac{(h-1)(m+k)n}{p}\beta$$

- 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

  $$\frac{2mnk}{p}\gamma + T^{+}_{\text{eSUMMA2D-A}}(m, n/h, k, r, c)$$

- Gather the C_K submatrices to Layer 0. The cost of such a gather can be approximated by

  $$\log_2(h)\alpha + \frac{h-1}{h}\frac{mn}{cr}\beta$$

Rather than giving the total cost, we merely note that the stationary A 3D algorithms can similarly be stated for general m, n, k, r, and c, 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.

5.3 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 $r \times c \times h$ mesh, again viewed as h layers. If we partition, conformally, C and A so that

$$C = \left(\begin{array}{c} C_0 \\ \hline \vdots \\ \hline C_{h-1} \end{array}\right) \quad \text{and} \quad A = \left(\begin{array}{c} A_0 \\ \hline \vdots \\ \hline A_{h-1} \end{array}\right),$$

then

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

This suggests the following 3D algorithm:

- Duplicate (broadcast) B to each of the layers, initializing the duplicates on meshes 1 through $h-1$ to zero. If matrix B is perfectly balanced among the processors, a lower bound for this is given by

  $$\log_2(h)\alpha + \frac{nk}{rc}\beta$$

  Simple algorithms that achieve within a factor two of this lower bound are known.

- 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 $(mn + mk)/(rc)$ data to processors $(I, J, 0)$ through $(I, J, h-1)$. The cost of such a scatter can be approximated by

  $$\log_2(h)\alpha + \frac{h-1}{h}\frac{m(n+k)}{cr}\beta = \log_2(h)\alpha + \frac{(h-1)m(n+k)}{p}\beta$$

- 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

  $$\frac{2mnk}{p}\gamma + T^+_{eSUMMA2D-B}(m/h, n, k, r, c)$$

- Gather the C_K submatrices to Layer 0. The cost of such a gather can be approximated by

  $$\log_2(h)\alpha + \frac{h-1}{h}\frac{mn}{cr}\beta$$

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.

5.4 Exercises

Exercise 9. *Propose and analyse eSUMMA3D-C for computing $C := A^T B + C$, $C := AB^T + C$, and $C := A^T B^T + C$.*
Exercise 10. *Propose and analyse eSUMMA3D-A algorithms for computing $C := A^T B + C$, $C := AB^T + C$, and $C := A^T B^T + C$.*
Exercise 11. *Propose and analyse eSUMMA3D-B algorithms for computing $C := A^T B + C$, $C := AB^T + C$, and $C := A^T B^T + C$.*

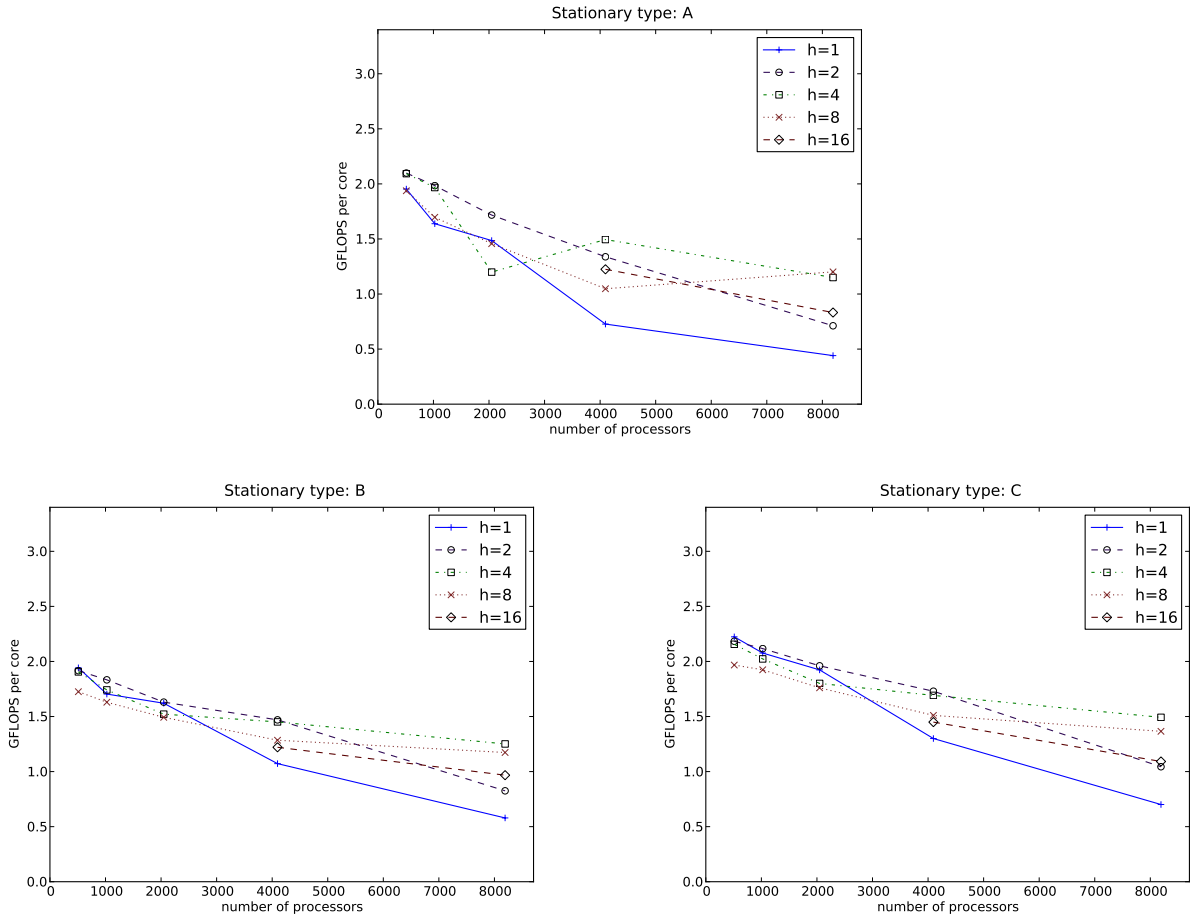

Figure 12: Performance of the different implementations when $m = n = k = 30,000$ and the number of nodes is varied.

6 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 3D mesh of nodes so that all communication necessary to duplicate was included in the performance calculations.

As in [17, 18], 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 was the number of layers, h. As a result, the $r \times c$ mesh for a single layer was chosen so that $r = c$ if p/h was a perfect square and $r = c/2$ otherwise. The "zig-zagging" observed in some of the curves is attributed to this square vs. nonsquare choice of $r \times c$. 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 12 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 13: 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 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 14 similarly illustrates that the eSUMMA2D-A and eSUMMA3D-A algorithms attain high performance already when $m = k$ are relatively large and n is relatively small and **Figure 15** 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 13(c) vs. Figure 14(a) shows that the eSUMMA2D-A algorithm, Figure 14(a) with $h = 1$, asymptotes sooner than the eSUMMA2D-C algorithm, Figure 13(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 14(a) than does increasing h for eSUMMA3D-C in Figure 13(c). A similar observation can be made for eSUMMA2D-B and eSUMMA3D-B in Figure 15(b).

7 Conclusion

We have given a systematic treatment of the parallel implementation of matrix-vector multiplication and rank-1 update. These motivate 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. We believe that sufficient detail has been given such that a reader can now easily extend our approach to alternative data distributions and/or more sophisticated architectures. Another interesting direction would be to analyze whether it would be worthwhile to use the proposed 3D parallelization, but with a different 2D parallelization. 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 and CCF-0917167, grants from Microsoft, and an unrestricted grant from Intel. 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).

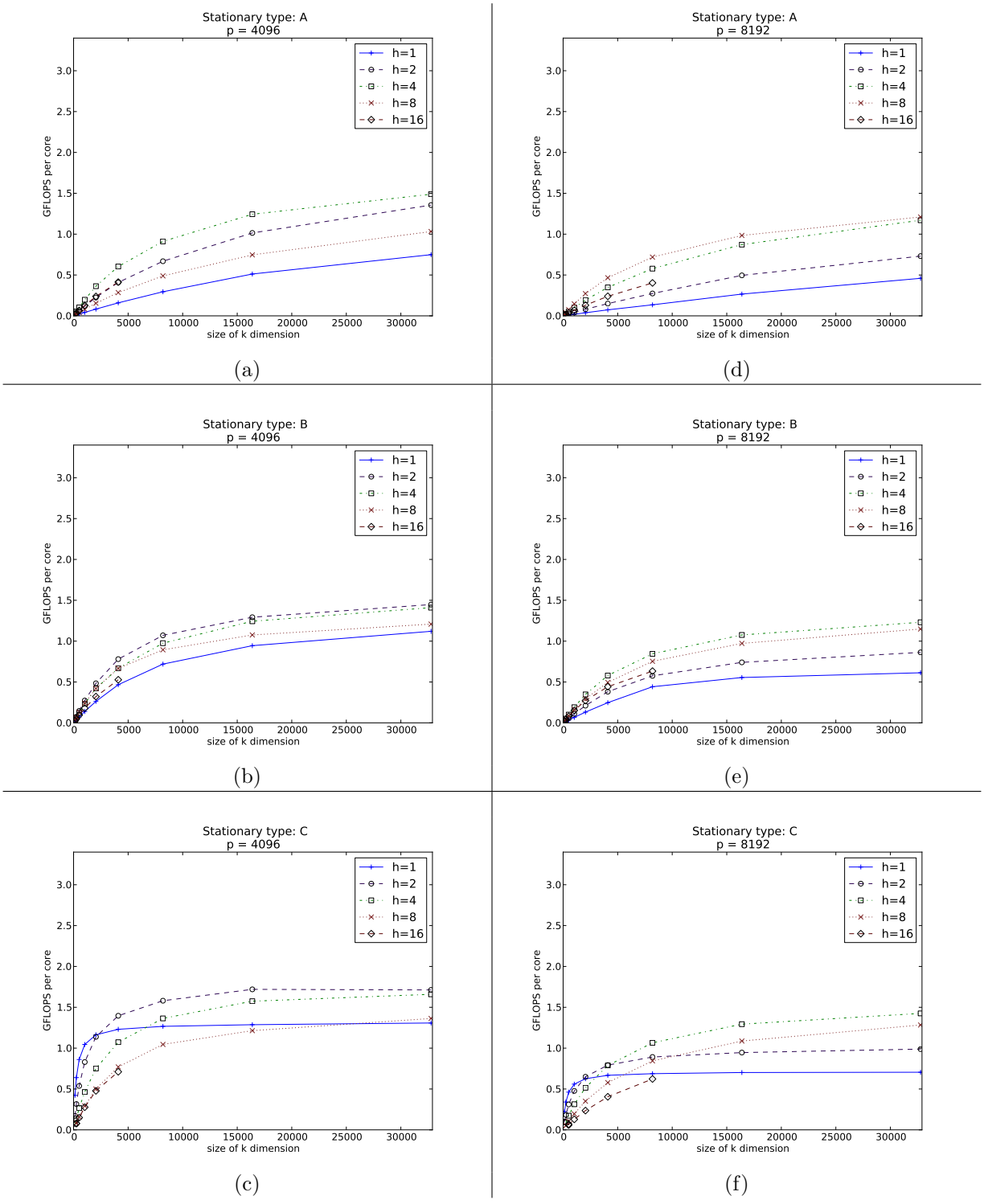

Figure 13: Performance of the different implementations when $m = n = 30,000$ and k is varied. As expected, the stationary C algorithms ramp up to high performance faster than the other algorithms when k is small.

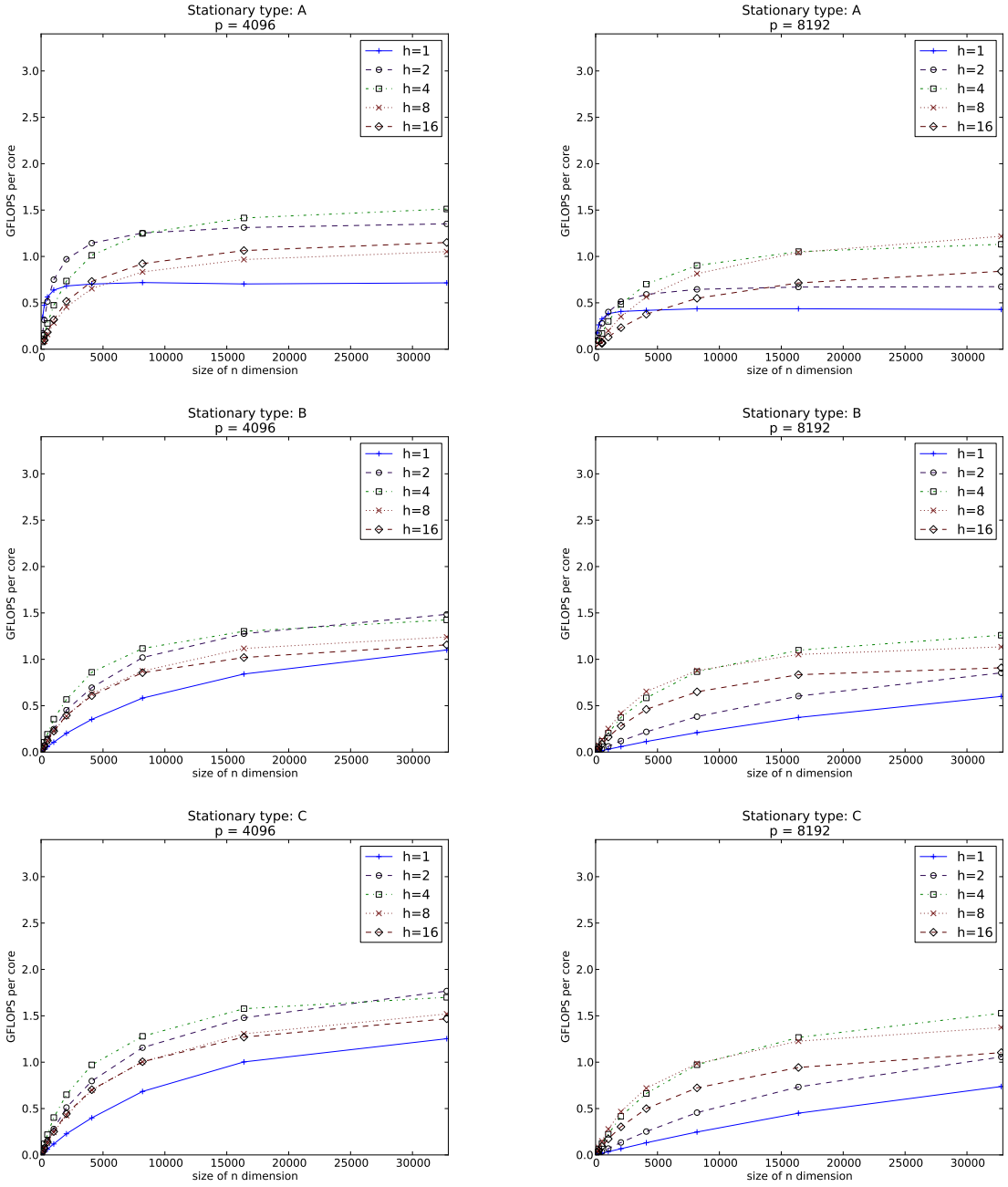

Figure 14: Performance of the different implementations when $m = k = 30,000$ and n is varied. As expected, the stationary A algorithms ramp up to high performance faster than the other algorithms when n is small.

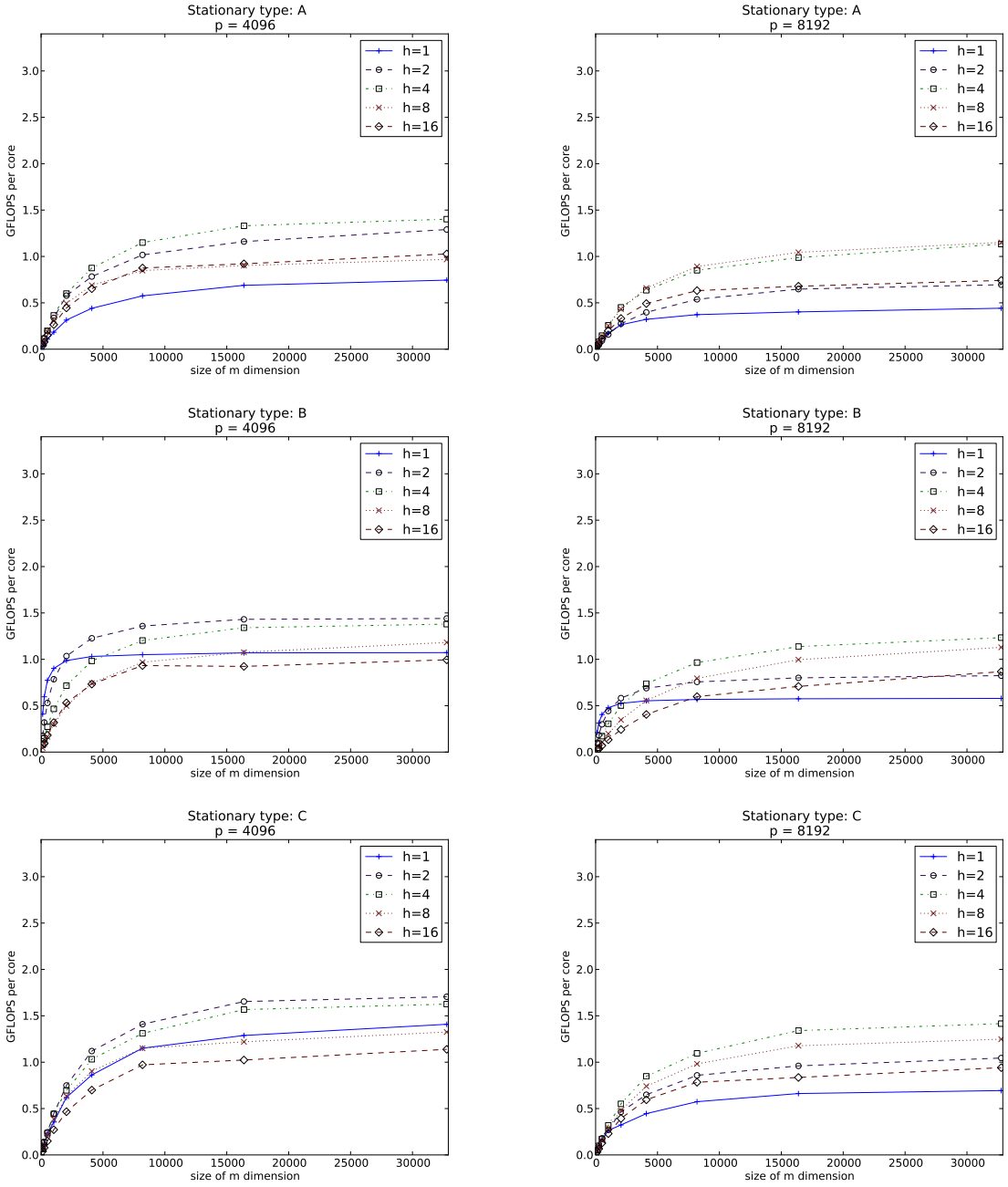

Figure 15: Performance of the different implementations when $n = k = 30,000$ and m is varied. As expected, the stationary B algorithms ramp up to high performance faster than the other algorithms when m is small.

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] 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 Series. New Series*, pages 253–271. Oxford University Press, Oxford, UK, 1993.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Jack Poulson, Bryan Marker, Jeff R. Hammond, Nichols A. Romero, and Robert van de Geijn. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Soft.* to appear.

[18] Jack Poulson, Robert van de Geijn, and Jeffrey Bennighof. (Parallel) algorithms for two-sided triangular solves and matrix multiplication. *ACM Trans. Math. Soft.* submitted.

[19] 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.

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

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

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

[23] 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 Des. Test*, 11(6):56–63, November 2009.

A Scalability of Matrix-vector Operations

In this section, 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 speedup 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

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

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 2.4.

Parallel $y := Ax$: As discussed in the body of the paper, the cost of the parallel algorithm was approximated by

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

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

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

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

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

$$
\begin{array}{rcl}
 & = & \frac{\log_2(p)\alpha + \frac{r-1}{\sqrt{p}}\sqrt{M}\beta + \frac{c-1}{\sqrt{p}}\sqrt{M}\beta + \frac{c-1}{\sqrt{p}}\sqrt{M}\gamma}{2M\gamma} \\
 & = & \log_2(p)\frac{1}{2M}\frac{\alpha}{\gamma} + \frac{r-1}{\sqrt{p}}\frac{1}{2\sqrt{M}}\frac{\beta}{\gamma} + \frac{c-1}{\sqrt{p}}\frac{1}{2\sqrt{M}}\frac{\beta}{\gamma} + \frac{c-1}{\sqrt{p}}\frac{1}{2\sqrt{M}}
\end{array}
$$

We will use this formula to now analyze scalability.

Case 1: $r \times c = p \times 1$**.** Then

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

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, $\lim_{p\rightarrow\infty} \frac{T^+(n_{\max}(p),p)}{T(n_{\max}(p))/p} \rightarrow \infty$ and eventually efficiency cannot be maintained. When $r \times c = p \times 1$, the proposed parallel matrix-vector multiply is not weakly scalable.

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

The cases where $r \times c = 1 \times p$ or $r \times c = p \times 1$ 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: $r \times c = \sqrt{p} \times \sqrt{p}$**.** Then

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

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 Exercises

Exercise 12. *Show that the parallel matrix-vector multiplication is not scalable when* $r \times c = 1 \times p$.
Exercise 13. *Show that the parallel matrix-vector multiplication is scalable (for practical purposes) if* r/c *is kept constant as* p *increases.*