HPC for numerical methods and data analysis

Laura Grigori
text from book in progress with Grey Ballard, Erin Carson, James Demmel

October 15, 2024

Contents

1 Communication cost model and collective communication operations 5
- 1.1 Abstract model of a parallel machine . . . 5
- 1.2 Collective communication operations . . . 6
  - 1.2.1 Binomial Tree and Butterfly Patterns . . . 8
  - 1.2.2 Scatter and Gather . . . 9
  - 1.2.3 Reduce_scatter and Allgather . . . 9
  - 1.2.4 Broadcast, Reduce, and Allreduce . . . 11
  - 1.2.5 Alltoall . . . 11

2 Background 13
- 2.1 Notations, BLAS and LAPACK libraries . . . 13
  - 2.1.1 Matrices and vectors . . . 13
  - 2.1.2 BLAS and LAPACK linear algebra libraries . . . 14

3 Orthogonalization 17
- 3.1 Background on Orthogonalization and Least Squares . . . 17
  - 3.1.1 Distribution of data in parallel . . . 18
- 3.2 Gram-Schmidt Orthogonalization . . . 18
  - 3.2.1 Left-Looking BLAS-2 CGS . . . 19
  - 3.2.2 Left-Looking BLAS-1 MGS . . . 21
- 3.3 Cholesky-QR . . . 22
- 3.4 Householder QR and TSQR . . . 23
  - 3.4.1 Householder QR . . . 23
  - 3.4.2 TSQR . . . 26
- 3.5 Numerical Stability . . . 29
  - 3.5.1 Gram-Schmidt . . . 29
  - 3.5.2 Cholesky-QR . . . 30
  - 3.5.3 Householder QR and TSQR . . . 30

4 Matrix Multiplication 31
- 4.1 Lower Bounds . . . 31
  - 4.1.1 Sequential case . . . 31
  - 4.1.2 Parallel Case . . . 33
- 4.2 Parallel algorithms . . . 34
  - 4.2.1 SUMMA . . . 34
  - 4.2.2 3D Algorithm (square case) . . . 36

5 Linear Systems 41
- 5.1 Background on LU factorization . . . 41
  - 5.1.1 Blocked LU factorization with row pivoting . . . 43
    - 5.1.1.1 Partial pivoting . . . 44
    - 5.1.1.2 Tournament pivoting . . . 44

5.2 Numerical stability 46
5.3 Lower bounds on communication 47
5.4 Parallel algorithms 48
5.4.1 Parallel LU with partial pivoting 48
5.4.2 Parallel communication avoiding LU with tournament pivoting . 49

Bibliography **53**

Chapter 1

Communication cost model and collective communication operations

We discuss a basic parallel computer architecture, the basic communication cost model that we will use to evaluate algorithms, and the parallel communication primitives (reduction, broadcast, etc.) that parallel algorithms will use.

1.1 ▪ Abstract model of a parallel machine

We consider a simple model of a parallel machine that is formed by a collection of homogeneous processors connected through a fast network. This model is displayed in Figure 1.1. The time required to compute one floating point operation per processor is γ. The time required to communicate a message of n words from one processor to another is modeled as $\alpha + n\beta$, where α is the interprocessor latency and β is the inverse of the interprocessor bandwidth. The time of a parallel algorithm is estimated with the $\alpha - \beta - \gamma$ model as,

$$T = \gamma \cdot \#\text{ flops} + \beta \cdot \#\text{ words} + \alpha \cdot \#\text{ messages}, \tag{1.1}$$

where $\#flops$ represents the computation on the critical path of the algorithm, $\#words$ the volume of communication, and $\#messages$ the number of messages exchanged on the critical path of the parallel algorithm. An advantage of this model is that it can be applied to both sequential and parallel machines, with the appropriate choices of the parameters α, β, γ.

This model has several simplifying assumptions:

- The time required to compute one flop per processor is constant. Thus the model ignores the memory hierarchy of each processor.
- The communication cost is independent of the topology of the interconnect network and of the physical distance between processors. Network contention is ignored.
- At a given time, a processor can send and can receive a message.
- Any subset of disjoint pair of processors can communicate simultaneously and the links in the network are assumed to be bidirectional. With these assumptions, the communication cost of exchanging a message of n words between a pair of processors is estimated as $\alpha + n\beta$.

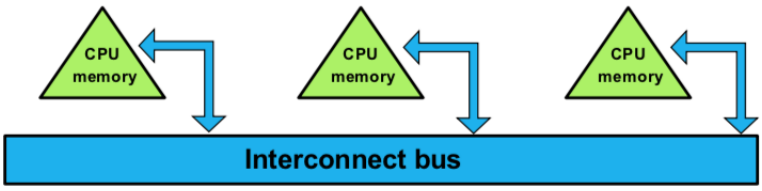


Figure 1.1: Abstract model of a parallel machine.

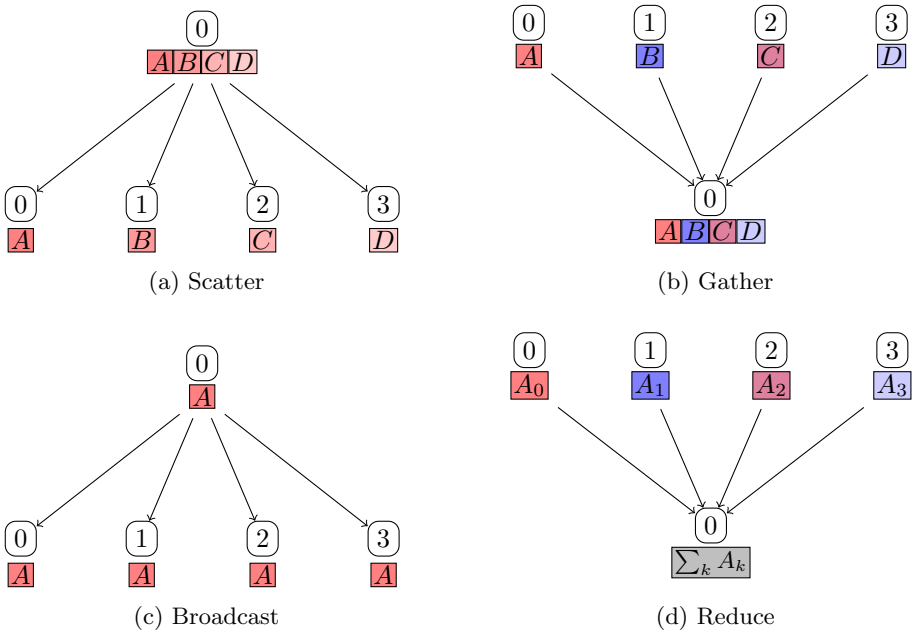


Figure 1.2: Rooted collective operations on 4 processors

1.2 ▪ Collective communication operations

We present several collective communication operations and their communication cost by considering the simplified model of a parallel machine described in Section 1.1. The presentation focuses on the most fundamental MPI collective operations.

The eight collectives presented in this section and their costs are summarized in Table 1.1, Figure 1.2, and Figure 1.3. We let n denote the size of the data involved in a given collective, as measured in words. Depending on the size of n, different algorithms can be used for these collective operations. Our focus is in particular on algorithms used for short or medium size messages, when the latency cost might dominate the communication cost, and in particular we consider the case when $n \geq P$. For simplicity, we assume that the number of processors P is a power of 2, and the size of the message n divides P when $n \geq P$.

We now describe what each collective does, and later we describe how the operations are performed efficiently. The most commonly used collectives are Broadcast and Reduce, and they are duals of each other. In a Broadcast, one processor, which we refer to as the root, has data that we want every other processor to have. The Reduce collective computes a global sum (or other accumulation operation) of the data stored on each

Table 1.1: Summary of collective operations involving P processors and their leading-order costs. Root is a designated processor among the P processors. Reduce, Reduce_scatter, and Allreduce are assumed to use an associative and commutative reduction operator and their cost includes the cost of arithmetic when the reduction of two words corresponds to one flop.

Routine	Description and cost of efficient algorithm	
Scatter	a root scatters n words, each processor receives n/P words	
	$\alpha \cdot \log_2 P + \beta \cdot n$	when $n \geq P$,
	$\alpha \cdot \log_2 n + \beta \cdot n$	otherwise
Gather	each processor sends n/P words, which are gathered on root	
	$\alpha \cdot \log_2 P + \beta \cdot n$	when $n \geq P$,
	$\alpha \cdot \log_2 n + \beta \cdot n$	otherwise
Reduce_scatter	reduction on n words from each processor, result scattered on all processors	
	$\alpha \cdot \log_2 P + \beta \cdot n + \gamma \cdot n$	when $n \geq P$,
	$\alpha \cdot \log_2 P + \beta \cdot (n + \log_2(P/n)) + \gamma \cdot (n + \log_2(P/n))$	otherwise
Allgather	each processor sends n/P words, which are gathered on all processors	
	$\alpha \cdot \log_2 P + \beta \cdot n$	when $n \geq P$
	$\alpha \cdot \log_2 P + \beta \cdot (n + \log_2(P/n))$	otherwise
Reduce	reduction on n words from each processor, result returned on root	
	$\alpha \cdot 2\log_2 P + \beta \cdot 2n + \gamma \cdot n$	when $n \geq P$,
	$\alpha \cdot \log_2(Pn) + \beta \cdot (2n + \log_2(P/n)) + \gamma \cdot (n + \log_2(P/n))$	otherwise
Broadcast	a root broadcasts n words to all processors	
	$\alpha \cdot 2\log_2 P + \beta \cdot 2n$	when $n \geq P$,
	$\alpha \cdot \log_2(Pn) + \beta \cdot (2n + \log_2(P/n))$	otherwise
Allreduce	reduction on n words from each processor, result returned on all processors	
	$\alpha \cdot 2\log_2 P + \beta \cdot 2n + \gamma \cdot n$	when $n \geq P$,
	$\alpha \cdot 2\log_2 P + \beta \cdot 2(n + \log_2(P/n)) + \gamma \cdot (n + \log_2(P/n))$	otherwise
Alltoall	each processor sends different n/P words to every other processor	
	$\alpha \cdot \log_2 P + \beta \cdot \frac{n}{2} \log_2 P$	

individual processor, storing the result on a single root processor. In each case, the data may be a single word or an array of arbitrary length, in which case the accumulation operation is performed elementwise. See Figs. 1.2c and 1.2d for a visualization.

Less common but simpler are the Scatter and Gather collectives, which are shown in Figs. 1.2a and 1.2b. In a Scatter, an array of data of size n starts on the root and is scattered evenly across all processors so that each processor stores n/P of the array. A Gather is the opposite: the data starts distributed across processors and is gathered onto a single root.

These first four collectives all specify a root processor, but the next four do not. As shown in Fig. 1.3a, the Reduce_scatter collective achieves the effect of first performing a Reduce followed by a Scatter (though it is not efficient to implement it this way): each processor starts with its own data array and then ends with a global accumulation of part of the array. An Allgather starts like a Gather but ends as if every processor is a root: all processors end with all the input data in their local memory (see Fig. 1.3b). Similarly, an Allreduce starts like a Reduce but ends as if every processor is the root: all processors end with a copy of the accumulated result as shown in Fig. 1.3c. Finally, the Alltoall collective performs a redistribution of the data: each processor starts with n data but ends with n/P data from each of the other processors, so it ends with a different set of n words than it started. An example is given in Fig. 1.3d.

These routines have different names in different communication libraries. For example

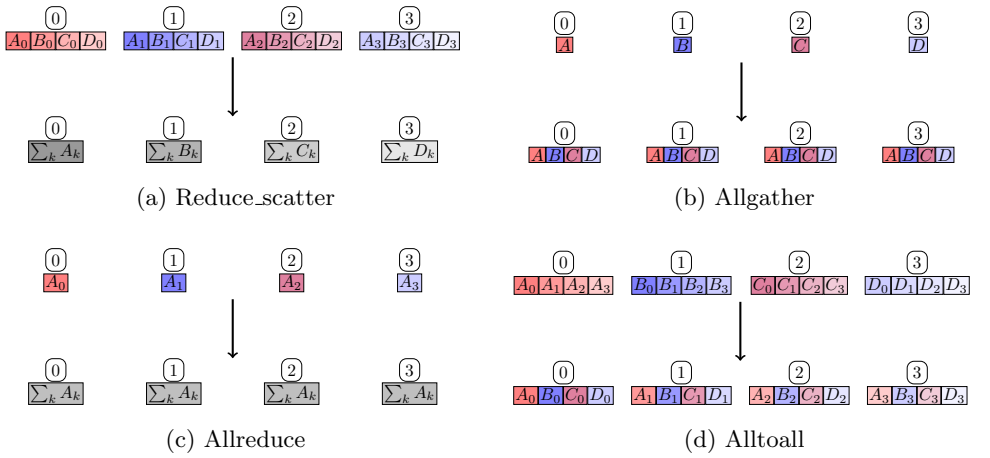


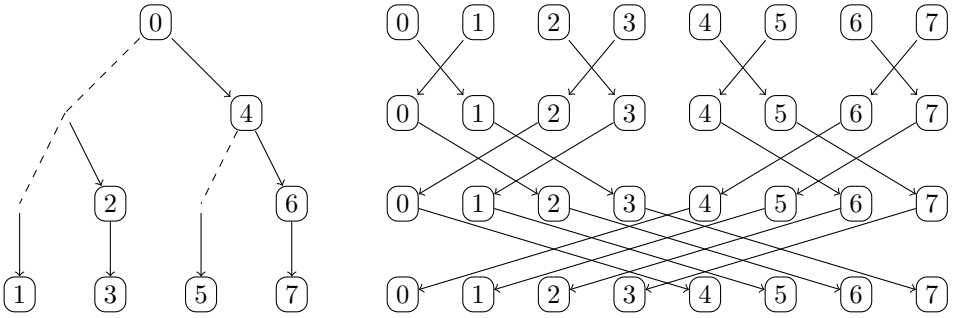
Figure 1.3: Unrooted collective operations on 4 processors

in MPI, the Broadcast, Allgather, and Alltoall are called MPI_Broadcast, MPI_Allgather, and MPI_Alltoall, respectively.

1.2.1 ▪ Binomial Tree and Butterfly Patterns

Efficient implementations of these collectives in our communication model are based on two communication patterns called binomial tree and butterfly, shown in Fig. 1.4. The binomial tree is like a binary tree except that some of the nodes in any given level are repeated from a previous level. For example, a simple algorithm for broadcasting n words uses a binomial tree of depth $\log_2 P$ and proceeds as follows. Suppose processor 0 is the root that starts with the data. First the root processor sends the n words to processor $P/2$. After this step, there are two subtrees, each having $P/2$ processors, for which the roots are processors 0 and $P/2$. Processors 0 and $P/2$ send simultaneously the n words to processors $P/4$ and $3P/4$. The algorithm continues recursively until depth $\log_2 P$. However, because every message is of size n, such an algorithm leads to a communication cost of $\alpha \cdot \log_2 P + \beta \cdot n \log_2 P$. A similar binomial-tree Reduce algorithm achieves the same communication cost as the binomial-tree Broadcast algorithm. While this latency cost is optimal, the bandwidth cost of Broadcast can be improved using a combination of Scatter and Allgather, as we describe in § 1.2.4.

The butterfly scheme is useful for the unrooted collectives. For example, to perform an Allgather on n words of data initially distributed across P processors, we can use a butterfly pattern of depth $\log_2 P$. In the first step, consecutive processors exchange their data, each ending with twice as many words of the data as they started. After this step, the n words are distributed across the even processors, and they are also distributed across the odd processors. The process continues recursively with pairwise exchanges among the even processors proceeding simultaneously with pairwise exchanges among the odd processors. After $\log_2 P$ steps, all processors have all the data. Because the size of each message doubles each step from n/P to $n/2$, the cost of the algorithm is $\alpha \cdot \log_2 P + \beta \cdot \frac{P-1}{P} n$. This algorithm is sometimes called recursive doubling, see § 1.2.3 for more details.



(a) binomial tree scheme (b) butterfly scheme

Figure 1.4: Communication schemes in collectives on 8 processors. Each level corresponds to a communication phase in the collective, with solid lines indicating messages.

1.2.2 ▪ Scatter and Gather

Examples for Scatter and Gather operations with 4 processors are presented in Figs. 1.2a and 1.2b, respectively. Both collectives are rooted, and efficient implementations of these algorithms use the binomial tree communication pattern.

In a Scatter operation, an array of n words, for $n \geq P$, is scattered from a root processor to all the other processors. At the end of this operation, processor i owns n/P words, which correspond to the elements starting at position in/P in the initial array, with the index of the first element of the array being 0. The binomial tree algorithm is called recursive halving. In the first step, the root 0 sends the 2nd half of the data to processor $P/2$. After the first step, the 1st $P/2$ processors recursively scatter the 1st half of the data, and simultaneously, the 2nd $P/2$ processors recursively scatter the 2nd half of the data. After $\log_2 P$ steps, each processor owns n/P of the data.

In a Gather operation, each processor starts with n/P data (assuming $n \geq P$) and the root ends with the entire array of n words. The binomial tree Gather algorithm works in the opposite direction of the Scatter algorithm and is called recursive doubling. For a Gather, the message size in the first step is n/P and occurs between consecutive processors. Then the Gather proceeds recursively on only the even processors. The message size in the last step is $n/2$ and occurs between processors 0 and $P/2$. The communication cost of both routines is given by

$$\alpha \cdot \log_2 P + \beta \cdot \frac{P-1}{P} n.$$

In the case that $n < P$, we can still perform a Scatter, but only n processors will end up with data. Similarly, we can perform a Gather across n processors, each of which has 1 word of data. In this case, assuming n is also a power of two, the cost above simplifies by setting $P = n$.

1.2.3 ▪ Reduce_scatter and Allgather

Reduce_scatter and Allgather are unrooted collectives. Like Scatter and Gather, Reduce_scatter and Allgather are duals of each other, as shown in Figs. 1.3a and 1.3b. Efficient algorithms for these collectives use the butterfly communication pattern. We assume $n \geq P$.

The Allgather collective starts with each processor owning n/P unique words of data and ends with all processors owning all n words. As in the efficient algorithm for Gather, we can perform a recursive doubling approach; however, in order for all processors to end with all the data, we use a butterfly pattern instead of a single binomial tree. In the first step, consecutive processors exchange their data, after which each processor owns $2n/P$ words of data. Then the full array is distributed across even processors and also across odd processors. The algorithm continues recursively and simultaneously on both sets of processors. After this step, each processor owns $2n/P$ words. In the final $\log_2 P$ step, each processor i such that $0 \leq i < P/2$ exchanges $n/2$ words with processor $i + P/2$. After this step, n words are gathered on all processors.

In a Reduce_scatter, each processor starts with n words and ends with n/P words of the global accumulated result. This works the same as Allgather but in the opposite order: we perform recursive halving using the butterfly pattern. We assume here that the reduce operation (e.g., scalar addition) is associative and commutative and that the reduction of two words costs one flop. In the first step, each processor i such that $0 \leq i < P/2$ exchanges $n/2$ words with processor $i + P/2$. All processors compute the reduction operation between the $n/2$ words they owned originally and the received data. At this point, all contributions to the left half of the array are distributed across the first $P/2$ processors, and all contributions to the right half of the array are distributed across the last $P/2$ processors, and the algorithm proceeds recursively on each half of processors simultaneously. In the last step, consecutive processors exchange n/P data and perform the final reductions.

Assuming $n \geq P$, the communication cost of each collective using these algorithms is given by

$$\alpha \cdot \log_2 P + \beta \cdot \frac{P-1}{P} n,$$

and the Reduce_scatter incurs an additional computation cost of $\gamma \frac{P-1}{P} n$.

When $n < P$, not all processors have data to contribute to an Allgather, and not all processors end up with data in a Reduce_scatter. Assuming n is also a power of two, we can combine binomial-tree algorithms with the algorithms described above.

In the case of an Allgather, n processors start with 1 word of data each, and all other processors start with no data. We start with a binomial-tree Broadcast among groups of P/n processors, each rooted at a different processor that started with data. After this step, there are P/n groups of processors, each consisting of n processors with each processor owning a unique word of data. Within each group, we simultaneously perform the recursive-doubling Allgather algorithm described above where the array size n matches the number of processors exactly.

In the case of a Reduce_scatter with $n < P$, we perform the same steps of the recursive-halving algorithm described above until the data size on each processor is 1 word, which requires $\log_2 n$ steps. At this point, groups of P/n processors own contributions to a single word of the final result. Within those n groups of processors, we simultaneously perform binomial-tree Reduce algorithms to obtain a final result with n processors each owning one word of the final result.

The cost of these algorithms for $n < P$ is the sum of the cost given above for $n \geq P$ (plugging in $P = n$) and the cost of the binomial-tree Broadcast or Reduce, which is $\alpha \cdot \log_2(P/n) + \beta \cdot \log_2(P/n)$ and additional cost of $\gamma \cdot \log_2(P/n)$ for Reduce. Thus, the communication cost is given by

$$\alpha \cdot \log_2 P + \beta \cdot \left(n - 1 + \log_2 \frac{P}{n} \right),$$

and the Reduce_scatter incurs an additional computation cost of $\gamma \cdot \left(n - 1 + \log_2 \frac{P}{n}\right)$.

1.2.4 ▪ Broadcast, Reduce, and Allreduce

Broadcast and Reduce are rooted collectives visualized in Figs. 1.2c and 1.2d. While each collective can be implemented straightforwardly using a binomial-tree algorithm, this simple approach is not bandwidth optimal. As described in § 1.2.1, the binomial-tree algorithm has communication cost $\alpha \cdot \log_2 P + \beta \cdot n \log_2 P$. Likewise, Allreduce is an unrooted collective (see Fig. 1.3c) whose straightforward implementation using a butterfly algorithm does not minimize bandwidth cost. Instead, we can implement these three collectives using pairs of simpler collectives analyzed above and reduce the bandwidth cost by a factor of $O(\log_2 P)$. More details and references can be found in [2, 7].

In a Broadcast, we start with n data stored on a root processor, and we end with all processors owning all the data. We can achieve this effect by first performing a Scatter and then performing an Allgather. Thus, we combine a binomial-tree algorithm that distributes the data from the root to all processors with a butterfly algorithm that gathers all the data to all the processors. The key observation is that within the Scatter and Allgather algorithms, the message sizes shrink and grow from $n/2$ to n/P and back, as opposed to each message having size n in the simple binomial-tree broadcast. This means that the bandwidth cost is proportional to n rather than $n \log_2 P$. Note that we double both the bandwidth and latency costs by using two collectives as subroutines, but both costs are within a factor of 2 of optimal. This algorithm works when $n \geq P$ and $n < P$, though the costs vary in the two cases. When $n \geq P$, the communication cost is

$$\alpha \cdot 2 \log_2 P + \beta \cdot 2 \frac{P-1}{P} n,$$

and when $n < P$, the cost is

$$\alpha \cdot (\log_2 P + \log_2 n) + \beta \cdot \left(\frac{2P-1}{P} n + \log_2 \frac{P}{n} - 1 \right).$$

In a Reduce, we start with n data on every processor and end with the global accumulation of that data onto a root processor. Instead of the simple binomial-tree algorithm, we implement a Reduce efficiently by using a Reduce_scatter followed by a Gather. Like the case of Broadcast, both bandwidth and latency costs are optimal to within a factor of 2, and the algorithm works for any data size n.

To implement an Allreduce efficiently, in which case the global accumulation of data ends redundantly on every processor, we perform a Reduce_scatter followed by an Allgather. In this case, both subroutines are implemented with the butterfly scheme. The simplified costs for both Reduce and Allreduce are given in Tab. 1.1.

1.2.5 ▪ Alltoall

In an Alltoall, each processor sends different n/P words to every other processor. At the end of the algorithm every processor has received $\frac{P-1}{P} n$ words. When latency is important, a butterfly algorithm can be used, which has cost

$$\alpha \cdot \log_2 P + \beta \cdot \frac{n}{2} \log_2 P.$$

Chapter 2
Background

This chapter will be updated progressively with notations and background needed throughout the lectures. It is organized as follows. Section 2.1 defines the vector and matrix notation that we will use throughout the book.

2.1 ▪ Notations, BLAS and LAPACK libraries

We introduce in this section notations and several basic matrix operations. Matlab notation is used predominantly throughout the book.

2.1.1 ▪ Matrices and vectors

Let $\mathbb{R}^{m\times n}$ denote the vector space of all real matrices of dimension $m \times n$ and $\mathbb{C}^{m\times n}$ denote the vector space of all complex matrices of dimension $m \times n$. Unless mentioned otherwise, we consider real vector spaces in this book. A matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ is a collection of $m \times n$ real numbers,

$$\mathbf{A} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}.$$

We denote the element of a matrix $\mathbf{A}$ at row i and column j as $\mathbf{A}(i, j)$, and sometimes also as a_{ij}. The submatrix of $\mathbf{A}$ formed by rows from i to j and columns from k to l is referred to as $\mathbf{A}(i : j, k : l)$.

$$\mathbf{A}(i : j, k : l) = \begin{bmatrix} a_{ik} & \dots & a_{il} \\ \vdots & & \vdots \\ a_{jk} & \dots & a_{jl} \end{bmatrix}.$$

The leading $k \times k$ minor of $\mathbf{A}$ is $\mathbf{A}(: k, : k)$. The matrix formed by concatenating two matrices $\mathbf{A}_1$, $\mathbf{A}_2$ stacked atop one another is referred to as $[\mathbf{A}_1; \mathbf{A}_2]$. The matrix formed by concatenating two matrices one next to another is referred to as $[\mathbf{A}_1, \mathbf{A}_2]$. For example, $\mathbf{A} = [\mathbf{A}(1 : i, :); \mathbf{A}(i + 1 : n, :)]$, or $\mathbf{A} = [\mathbf{A}(:, 1 : j); \mathbf{A}(:, j + 1 : m)]$, where $1 \le i < m$, $1 \le j < n$. The matrix $|\mathbf{A}|$ is the matrix formed by the absolute value of the elements of $\mathbf{A}$. The identity matrix of size $n \times n$ is referred to as $\mathbf{I}_n$.

There are several different ways to partition a matrix, by rows or blocks of rows, by columns or blocks of coumns, or simply by blocks. In general, algorithms use a partition into square blocks of dimension $b \times b$, and by assuming that m and n divide b, $M = m/b$, $N = n/b$, the matrix $\mathbf{A}$ is partitioned as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \dots & \mathbf{A}_{1N} \\ \vdots & & \vdots \\ \mathbf{A}_{M1} & \dots & \mathbf{A}_{MN} \end{bmatrix}.$$

A real m-vector $\mathbf{v}$ belongs to the vector space $\mathbf{R}^m$ and is defined as

$$\mathbf{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix}.$$

Its kth component is referred to as either v_k or $\mathbf{v}(k)$.

In general, matrices are denoted by upper case letters, while vectors are denoted by lower case letters. A scalar is denoted by a greek lower case letter, unless it is a specific element of a matrix or a vector.

Some of the basic matrix operations are the following:

- Transposition: $\mathbf{B} = \mathbf{A}^T$, where $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{B} \in \mathbb{R}^{n \times m}$ and

$$\mathbf{A}(i,j) = \mathbf{B}(j,i), \quad \text{for all } 1 \leq i \leq m, 1 \leq j \leq n.$$

- Addition: $\mathbf{C} = \mathbf{A} + \mathbf{B}$, where $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{m \times n}$ and

$$\mathbf{C}(i,j) = \mathbf{A}(i,j) + \mathbf{B}(i,j), \quad \text{for all } 1 \leq i \leq m, 1 \leq j \leq n.$$

- Matrix-matrix multiplication: $\mathbf{C} = \mathbf{AB}$, or sometimes for more clarity denoted as $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, where $\mathbf{A} \in \mathbb{R}^{m \times p}$, $\mathbf{B} \in \mathbb{R}^{p \times n}$, $\mathbf{C} \in \mathbb{R}^{m \times n}$, and

$$\mathbf{C}(i,j) = \sum_{k=1}^{p} \mathbf{A}(i,k) \cdot \mathbf{B}(k,j), \quad \text{for all } 1 \leq i \leq m, 1 \leq j \leq n.$$

2.1.2 ▪ BLAS and LAPACK linear algebra libraries

Given the importance of linear algebra, there is a standard interface for basic linear algebra routines, referred to as BLAS. Those routines are grouped into three categories, BLAS-1, BLAS-2 and BLAS-3.

- BLAS-1: 15 different operations. It implements vector-vector operations as dot product of two vectors, saxpy (y=a*x+y). Those routines are not very efficient, since there are few operations that are performed on each element of the vectors. Hence the cost of transferring data through memory hierarchies is important/

- BLAS-2: 25 different operations. It implements matrix-vector operations, as matrix vector multiplication. Those routines are slightly faster than BLAS1, but their runtimes remains dominated by data transfers between different levels of the memory hierarchy.

- BLAS-3: 9 different operations. It implements matrix-matrix operations as matrix matrix multiplication. There routines are efficient. They are implemented to use efficiently the memory hierarchy. Indeed this is possible since for multiplying two matrices for example of dimensions $n \times n$, n^3 operations for matrices of dimensions $n \times n$. Most efficient algorithms will aim to rely as much as possible on BLAS3 operations.

More advanced linear algebra algorithms that compute the factorization of a matrix, as QR, LU or Cholesky, or compute the eigenvalue decomposition of a matrix, are available in many libraries provided by vendors. Their reference implementation can be found in LAPACK, and a distributed memory implementation in ScaLAPACK.

Chapter 3

Orthogonalization

Given a set of vectors, the term *orthogonalization* refers to computing a set of orthonormal vectors, which are unit length and mutually orthogonal, that spans the same space as the input. Computing a QR decomposition of a matrix is one way to orthogonalize the columns of the matrix. There are a variety of algorithms for orthogonalization and computing QR decompositions. Gram-Schmidt methods can be thought of as triangular orthogonalization: each successive vector is orthogonalized against the previous ones, which yields a transformation matrix $\mathbf{R}$ with triangular structure. Other methods, including Householder QR, can be thought of as orthogonal triangularization: each step of the algorithm uses an orthogonal transformation to annihilate entries in the matrix $\mathbf{A}$ until it becomes the triangular factor $\mathbf{R}$. Methods based on Cholesky-QR exploit the fact that $\mathbf{A} = \mathbf{QR}$ implies $\mathbf{A}^T\mathbf{A} = \mathbf{R}^T\mathbf{Q}^T\mathbf{QR} = \mathbf{R}^T\mathbf{R}$, which means the Cholesky factor of $\mathbf{A}^T\mathbf{A}$ is the triangular factor of the QR decomposition of $\mathbf{A}$.

These methods have varying numerical stability properties as well as arithmetic and communication costs. In this chapter, we focus on the "tall-and-skinny" case, where the lengths of the vectors far exceeds the number of vectors or equivalently, the input matrix has many more rows than columns. This case is common for highly overdetermined least squares problems and large-scale Krylov methods. When the problem is not so tall and skinny, more general algorithms are required to achieve communication efficiency. We discuss these approaches, which utilize algorithms for the tall-and-skinny case as subroutines, in a later section.

Communication lower bounds exist for orthogonalization algorithms, and they are generally consistent with other matrix computations. We will discuss communication lower bounds in the following chapter.

3.1 ▪ Background on Orthogonalization and Least Squares

An orthogonal matrix $\mathbf{Q}$ is a square matrix that satisfies $\mathbf{Q}^T\mathbf{Q} = \mathbf{QQ}^T = \mathbf{I}$. Orthogonal matrices have very convenient properties, so they are fundamental to the solution of a wide range of problems in linear algebra. For example, we can solve linear systems quickly because $\mathbf{Q}^{-1} = \mathbf{Q}^T$. They also maintain norms of vectors ($\|\mathbf{Qx}\| = \|\mathbf{x}\|$).

An $m \times n$ matrix $\mathbf{Q}$ with orthonormal columns satisfies $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$, but if $m > n$, $\mathbf{QQ}^T \neq \mathbf{I}$. In this case, each column has unit norm and each pair of columns is mutually orthogonal. Suppose we have a set of n (column) vectors of length m. Arranging the vectors into the columns of a matrix $\mathbf{A}$, the QR decomposition $\mathbf{A} = \mathbf{QR}$, with $\mathbf{R}$ upper

triangular, achieves orthogonalization with the property that the first k columns of $\mathbf{Q}$ are an orthogonalization of the first k columns of $\mathbf{A}$ for each $1 \leq k \leq n$. In this case, we consider $\mathbf{Q}$ to be $m \times n$, a matrix with orthonormal columns, and $\mathbf{R}$ to be square, which is sometimes referred to as the compact QR decomposition. An equivalent decomposition can be written $\mathbf{A} = \begin{bmatrix}\mathbf{Q} & \tilde{\mathbf{Q}}\end{bmatrix} \begin{bmatrix}\mathbf{R} \\ \mathbf{0}\end{bmatrix}$, where $\tilde{\mathbf{Q}}$ is any orthogonal complement of $\mathbf{Q}$ (it is a matrix with orthonormal columns that are all orthogonal to the columns of $\mathbf{Q}$). In this full QR decomposition, the first factor $\begin{bmatrix}\mathbf{Q} & \tilde{\mathbf{Q}}\end{bmatrix}$ is an orthogonal matrix.

Three of the most compelling applications of orthogonalization within linear algebra are the solution of linear least squares problems, computing the low rank approximation of a matrix, and Krylov subspace methods for iteratively solving linear systems and eigenvalue problems, but there are many others. We will discuss low rank approximation algorithms and Krylov subspace methods later on.

A linear least squares problem is defined by an $m \times n$ coefficient matrix $\mathbf{A}$ and a length-n vector $\mathbf{b}$:

$$\arg\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\| \tag{3.1}$$

If we consider the full QR decomposition of $\mathbf{A}$ and use the property that orthogonal matrices maintain norms, we have

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 = \left\| \begin{bmatrix}\mathbf{R} \\ \mathbf{0}\end{bmatrix} \mathbf{x} - \begin{bmatrix}\mathbf{Q} & \tilde{\mathbf{Q}}\end{bmatrix}^T \mathbf{b} \right\|^2 = \left\| \mathbf{R}\mathbf{x} - \mathbf{Q}^T\mathbf{b} \right\|^2 + \left\| \tilde{\mathbf{Q}}^T\mathbf{b} \right\|^2.$$

This implies that

$$\arg\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\| = \arg\min_{\mathbf{x}} \left\| \mathbf{R}\mathbf{x} - \mathbf{Q}^T\mathbf{b} \right\|,$$

and so we can solve the linear least squares problem by solving the triangular linear system in terms of the QR decomposition factors.

3.1.1 ▪ Distribution of data in parallel

We now turn our attention to parallel algorithms for orthogonalization of an $m \times n$ matrix $\mathbf{A}$ using P processors. We assume that our matrices are sufficiently tall and skinny, with the explicit assumption that $m/n \geq P$.

For all the algorithms we consider, we impose a 1D row-wise distribution of the input matrix across processors, so that each processor owns $m_\ell = m/P$ rows of $\mathbf{A}$ and we assume m_ℓ is an integer. With this distribution, processor I owns the $m_\ell \times n$ submatrix

$$\mathbf{A}_I = \mathbf{A}((I-1)m_\ell + 1 : Im_\ell, 1 : n).$$

Under the tall-and-skinny assumption, each processor's local submatrix has more rows than columns, as $m_\ell \geq n$.

3.2 ▪ Gram-Schmidt Orthogonalization

Gram-Schmidt orthogonalization is accomplished via a sequence of projections. In step j, the next orthogonal vector is produced by subtracting from the jth column of $\mathbf{A}$ the orthogonal projection of the jth column of $\mathbf{A}$ onto the previously computed orthogonal vectors. The difference between classical Gram-Schmidt (CGS) and modified Gram-Schmidt (MGS) is that the computations are performed in a different order. Whereas in CGS, the next vector is computed as $(\mathbf{I} - \mathbf{Q}(:, 1 : j-1)\mathbf{Q}(:, 1 : j-1)^T)\mathbf{A}(:, j)$, in MGS the

computation is performed via $(\mathbf{I} - \mathbf{Q}(:, j-1)\mathbf{Q}(:, j-1)^T) \cdots (\mathbf{I} - \mathbf{Q}(:, 1)\mathbf{Q}(:, 1)^T)\mathbf{A}(:, j)$. Although the approaches are mathematically equivalent, this seemingly small change makes a significant difference numerically.

The two main variants of Gram-Schmidt orthogonalization are classical and modified, but for each variant, there are several approaches for organizing the computation. We will first distinguish between left-looking and right-looking algorithms.

Both types of algorithms work column by column, left to right. Left-looking algorithms repeatedly access columns to the left in order to compute the current column, and they do not access or update columns to the right at that iteration. Right-looking algorithms repeatedly access and update columns to the right after computing the current column, and they do not access columns to the left ever again. We note that left-looking algorithms are required in the case that vectors (columns) are added to the set one at a time, as is the case in many Krylov methods. Right-looking algorithms cannot update future vectors unless they are known from the start.

Next, we distinguish between BLAS-1 and BLAS-2 algorithms. Removing vector projections in the Gram-Schmidt process involves dot products (to compute the length of the projection) and axpys (to subtract the projection from the vector), both of which are BLAS-1 (vector-vector) operations. Because these operations are performed for multiple vectors at a time, they can usually be cast into BLAS-2 (matrix-vector) operations. BLAS-2 operations offer modest improvements in communication costs, allow for more flexibility of low-level optimizations to be applied within the BLAS implementation, and also simplify the algorithms.

We focus in the following only on left-looking algorithms. We present left-looking CGS, sequential and parallel, and parallel left-looking MGS, the other variants can be adapted from these.

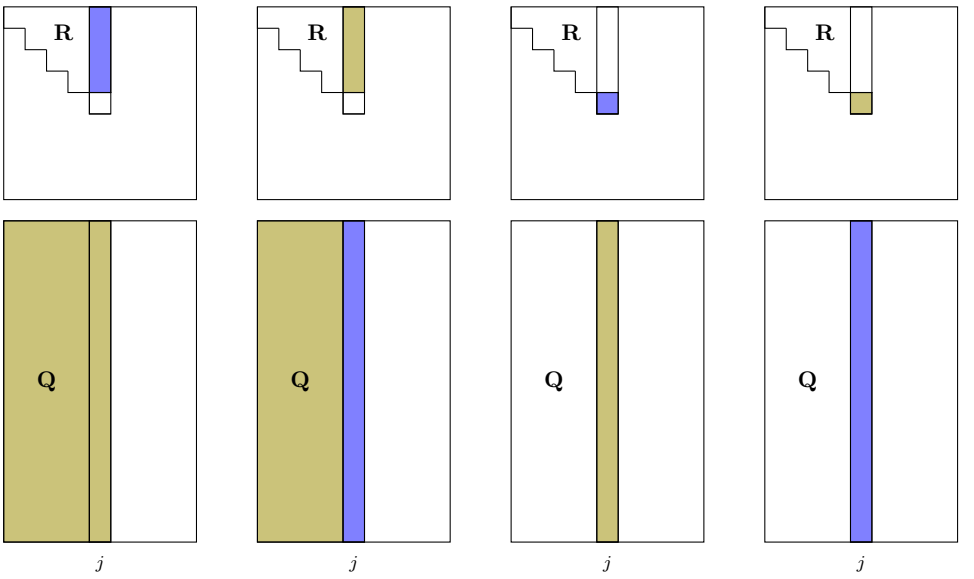
3.2.1 ▪ Left-Looking BLAS-2 CGS

Algorithm 3.1. Classical Gram-Schmidt (Left-looking, BLAS-2 version).

Require: $\mathbf{A}$ is an $m \times n$ matrix with $m \geq n$
Ensure: $\mathbf{QR} = \mathbf{A}$ where $\mathbf{R}$ is upper triangular

```
1:  function [Q, R] = CGS(A)
2:      R = 0
3:      R(1, 1) = ‖A(:, 1)‖_2
4:      Q(:, 1) = A(:, 1) / R(1, 1)
5:      for j = 2 to n do
6:          R(1 : j−1, j) = Q(:, 1 : j−1)^T · A(:, j)
7:          Q(:, j) = A(:, j) − Q(:, 1 : j−1) · R(1 : j−1, j)
8:          R(j, j) = ‖Q(:, j)‖_2
9:          Q(:, j) = Q(:, j) / R(j, j)
10:     end for
11: end function
```

Here we consider the BLAS-2, left-looking version of CGS, which is presented as Alg. 3.1 and visualized in Fig. 3.1. The algorithm works column by column, and for each column j, it performs four steps: it computes the lengths of the projections of the vector on the previously computed vectors and stores them in the jth column of $\mathbf{R}$ in line 6 (Fig. 3.1a), it subtracts those projections from the vector in line 7 (Fig. 3.1b), and then it



(a) Comp. projections (line 6) (b) Update from left (line 7) (c) Compute norm (line 8) (d) Normalize $\mathbf{Q}(:, j)$ (line 9)

Figure 3.1: Steps of Left-looking CGS (Alg. 3.1) at iteration j. Green highlighting indicates access (reads) and blue highlighting indicates update (writes).

normalizes the vector and stores its magnitude in the diagonal entry of $\mathbf{R}$ in lines 8 and 9 (Figs. 3.1c and 3.1d). The algorithm is left looking because at each iteration it computes the projections of the vector on the previously computed columns (stored to the left) and never accesses or updates any columns to the right. The algorithm is BLAS-2 because the projections are computed and removed using matrix-vector products.

In the case of parallel left-looking CGS, whose access pattern is illustrated in Fig. 3.1, the projections of previously computed columns of $\mathbf{Q}$ are removed from the current column of $\mathbf{A}$. The dot products that represent the magnitude of these projections in the jth iteration are given by the matrix-vector product $\mathbf{Q}(:, 1 : j-1)^T \mathbf{A}(:, j)$. This matrix-vector product involves distributed data and is performed in parallel as shown in Alg. 3.2. In the 1D block-row distribution, this consists of a matrix-vector product involving local data (line 8) followed by a sum across all processors. Because the result will be used in the subsequent matrix-vector computation, we compute the sum using an all-reduce (line 9), after which the update of the jth column can be performed locally with no further communication (line 10). Normalizing the vector that remains after the projections have been removed follows a similar computational pattern in lines 11 to 13.

The computation and communication costs are dominated by the matrix-vector products and corresponding collective. The number of rows of $\mathbf{Q}_I$ is m/P, and the number of columns involved in the matrix-vector products at iteration j is $j-1$, so the total arithmetic cost over the entire algorithm is $2mn^2/P$. The all-reduce collective involves data of size $j-1$ at iteration j. By using the communication costs from Tab. 1.1 and ignoring some lower order terms, the cost over all n iterations is $\beta \cdot O(n^2 + n \log P) + \alpha \cdot O(n \log P)$ (see details in § 1.2).

Algorithm 3.2. Parallel Classical Gram-Schmidt (Left-looking).

Require: $\mathbf{A}$ is an $m \times n$ matrix 1D-row-distributed over P processors
Ensure: $\mathbf{QR} = \mathbf{A}$ where $\mathbf{R}$ is upper triangular and $\mathbf{Q}$ is $m \times n$
Ensure: $\mathbf{R}$ is stored redundantly on all processors and $\mathbf{Q}$'s distribution matches $\mathbf{A}$
1: **function** $[\mathbf{Q}, \mathbf{R}] = $ 1D-CGS($\mathbf{A}$)
2: $I = $ MyProcID()
3: $\mathbf{R} = \mathbf{0}$
4: $\overline{\beta} = \|\mathbf{A}_I(:, 1)\|_2^2$
5: All-reduce $\overline{\beta}$ over all processors, take square root, and store in $\mathbf{R}(1, 1)$
6: $\mathbf{Q}_I(:, 1) = \mathbf{A}_I(:, 1)$ / $\mathbf{R}(1, 1)$
7: **for** $j = 2$ to n **do**
8: $\overline{\mathbf{r}} = \mathbf{Q}_I(:, 1 : j-1)^T \cdot \mathbf{A}_I(:, j)$
9: All-reduce $\overline{\mathbf{r}}$ over all processors, store in $\mathbf{R}(1 : j-1, j)$
10: $\mathbf{Q}_I(:, j) = \mathbf{A}_I(:, j) - \mathbf{Q}_I(:, 1 : j-1) \cdot \mathbf{R}(1 : j-1, j)$
11: $\overline{\beta} = \|\mathbf{Q}_I(:, j)\|_2^2$
12: All-reduce $\overline{\beta}$ over all processors, take square root, and store in $\mathbf{R}(j, j)$
13: $\mathbf{Q}_I(:, j) = \mathbf{Q}_I(:, j)$ / $\mathbf{R}(j, j)$
14: **end for**
15: **end function**

3.2.2 ▪ Left-Looking BLAS-1 MGS

In the case of left-looking MGS, the parallel algorithm given in Alg. 3.3 must perform BLAS-1 style operations to compute and remove projections. Note that the main difference between Alg. 3.3 and Alg. 3.2 is the doubly nested loops and the communication that occurs in the inner loop. In this algorithm, the dot product between each previous column of $\mathbf{Q}$ and the current vector must be computed independently and removed from the vector before proceeding, and each operation occurs over the 1D-row-distributed data. The computational cost of the inner loop is that of a dot product and an axpy of vectors of local dimension m/P, and the communication cost is that of an all-reduce of a single element. Over all $n^2/2$ inner iterations, the total costs are $\gamma \cdot O(2mn^2/P)$ and $\beta \cdot O(n^2 \log P) + \alpha \cdot O(n^2 \log P)$. Note that no more efficient all-reduce algorithm exists when the data consists of a single element.

Algorithm 3.3. Parallel Modified Gram-Schmidt (Left-looking).

Require: $\mathbf{A}$ is an $m \times n$ matrix 1D-row-distributed over P processors
Ensure: $\mathbf{QR} = \mathbf{A}$ where $\mathbf{R}$ is upper triangular and $\mathbf{Q}$ is $m \times n$
Ensure: $\mathbf{R}$ is stored redundantly on all processors and $\mathbf{Q}$'s distribution matches $\mathbf{A}$
1: **function** $[\mathbf{Q}, \mathbf{R}] = $ 1D-MGS($\mathbf{A}$)
2: $I = $ MyProcID()
3: $\mathbf{R} = \mathbf{0}$
4: $\mathbf{Q}_I = \mathbf{A}_I$
5: **for** $j = 1$ to n **do**
6: **for** $i = 1$ to $j - 1$ **do**
7: $\overline{\rho} = \mathbf{Q}_I(:, i)^T \cdot \mathbf{Q}_I(:, j)$
8: All-reduce $\overline{\rho}$ over all processors, store in $\mathbf{R}(i, j)$
9: $\mathbf{Q}_I(:, j) = \mathbf{Q}_I(:, j) - \mathbf{Q}_I(:, i) \cdot \mathbf{R}(i, j)$

```
10:         end for
11:         β̄ = ‖Q_I(:, j)‖_2^2
12:         All-reduce β̄ over all processors, take square root, and store in R(j, j)
13:         Q_I(:, j) = Q_I(:, j) / R(j, j)
14:     end for
15: end function
```

We highlight the extra factor of n that appears in the latency cost of left-looking parallel MGS compared to parallel CGS. For this reason, many users resort to the more communication-efficient CGS.

3.3 ▪ Cholesky-QR

In contrast to iterative, projection-based approaches like Gram-Schmidt, the Cholesky-QR approach is based on the fact that in exact arithmetic, the $\mathbf{R}$ in the QR decomposition of $\mathbf{A}$ is the Cholesky factor of $\mathbf{A}^T\mathbf{A}$. The Cholesky-QR algorithm (abbreviated CholQR) first computes $\mathbf{A}^T\mathbf{A}$, computes it Cholesky factor to find $\mathbf{R}$, and then computes $\mathbf{Q} = \mathbf{A}\mathbf{R}^{-1}$. It is clear that explicitly forming $\mathbf{A}^T\mathbf{A}$ can have consequences numerically; it is shown in [10] that if $O(\varepsilon)\kappa^2(\mathbf{A}) < 1$, then the loss of orthogonality is bounded by $O(\varepsilon)\kappa^2(\mathbf{A})$.

Algorithm 3.4. Cholesky QR.

Require: $\mathbf{A}$ is an $m \times n$ matrix with $m \geq n$
Ensure: $\mathbf{QR} = \mathbf{A}$ where $\mathbf{R}$ is upper triangular $n \times n$ and $\mathbf{Q}$ is $m \times n$

```
1: function [Q, R] = CholQR(A, b)
2:     G = 0
3:     G = A^T A
4:     R = Cholesky(G)
5:     Q = AR^{-1}                          ▷ Triangular solve with multiple RHS
6: end function
```

The Cholesky-QR algorithm has three simple steps: compute $\mathbf{A}^T\mathbf{A}$, compute its Cholesky decomposition to find $\mathbf{R}$, and use triangular solve to recover $\mathbf{Q}$.

We consider now parallel Cholesky-QR. The three steps of Cholesky-QR are (1) form $\mathbf{A}^T\mathbf{A}$, (2) compute Cholesky factorization for $\mathbf{R}$, and (3) perform a triangular solve (TRSM) for $\mathbf{Q}$. When $\mathbf{A}$ (and $\mathbf{Q}$) are 1D-row-distributed, the first and third steps are easily parallelized. Under the assumption that $m/P \geq n$, the most communication efficient algorithm for computing $\mathbf{A}^T\mathbf{A}$ is by assuming that the matrix has a 1D block row distribution. After symmetric multiplication of each local block of dimension $m/P \times n$, the final result is the sum over all processors. We perform an all-reduce in line 4 of Alg. 3.5 so that the Cholesky factorization of the result can be performed redundantly to obtain $\mathbf{R}$ on all processors. In this way, the parallel TRSM requires no further communication, as each processor computes its local block of $\mathbf{Q}$ from its local block of $\mathbf{A}$ using $\mathbf{R}$.

The computational cost of Parallel Cholesky-QR is thus $2mn^2/P + O(n^3)$, and the communication cost is that of the all-reduce: $\beta \cdot O(n^2) + \alpha \cdot O(\log P)$ when $n^2 \geq P$.

Algorithm 3.5. Parallel Cholesky-QR.

Require: $\mathbf{A}$ is an $m \times n$ matrix 1D-row-distributed over P processors
Ensure: $\mathbf{QR} = \mathbf{A}$ where $\mathbf{R}$ is upper triangular and $\mathbf{Q}$ is $m \times n$
Ensure: $\mathbf{R}$ is stored redundantly on all processors and $\mathbf{Q}$'s distribution matches $\mathbf{A}$
1: **function** $[\mathbf{Q}, \mathbf{R}] =$ ParCholQR($\mathbf{A}$)
2: $I =$ MyProcID()
3: $\overline{\mathbf{G}} = \mathbf{A}_I^T \mathbf{A}_I$ ▷ Local symmetric rank-k update
4: All-reduce $\overline{\mathbf{G}}$ over all processors, store in $\mathbf{G}$
5: $\mathbf{R} =$ Cholesky($\mathbf{G}$) ▷ Performed redundantly on all processors
6: $\mathbf{Q}_I = \mathbf{A}_I \mathbf{R}^{-1}$ ▷ Local triangular solve with multiple RHS
7: **end function**

3.4 ▪ Householder QR and TSQR

3.4.1 ▪ Householder QR

The Householder QR factorization is an orthogonalization procedure that transforms the input matrix into an upper triangular factor $\mathbf{R}$ using orthogonal transformations based on Householder reflectors. A Householder reflector $\mathbf{H} \in \mathbb{R}^{m \times m}$ is constructed from a Householder vector $\mathbf{y} \in \mathbb{R}^m \setminus \{0\}$ as:

$$\mathbf{H} = \mathbf{I}_m - \frac{2}{\mathbf{y}^T\mathbf{y}}\mathbf{y}\mathbf{y}^T, \quad \mathbf{H}^2 = \mathbf{H}^T\mathbf{H} = \mathbf{I}_m, \tag{3.2}$$

where $\mathbf{I}_m$ is the identity matrix of dimensions $m \times m$. Note that $\mathbf{H}$ is symmetric and orthogonal and is independent of the scaling of $\mathbf{y}$. We illustrate the construction of the Householder reflector that allows to annihilate all the elements of the first column of $\mathbf{A}$, except the first one. We refer to this first column as $\mathbf{a}_1$, $\mathbf{a}_1 \in \mathbb{R}^m$. Note first that when applied to a vector $\mathbf{a}_1$, $\mathbf{H}$ reflects $\mathbf{a}_1$ about the hyperplane $\text{span}(\mathbf{y})^\perp$, since $\mathbf{H}\cdot\mathbf{a}_1 = \mathbf{a}_1 - \frac{2\mathbf{y}^T\mathbf{a}_1}{\mathbf{y}^T\mathbf{y}}\mathbf{y} = \mathbf{a}_1 - \xi\mathbf{y}$, where ξ is a scalar. Suppose $\mathbf{a}_1$ is not a multiple of $\mathbf{e}_1$ (the vector whose first element is 1 and all the other elements are zero). Let $\tilde{\alpha} = \mathbf{a}_1(1)$ be its first element. By setting $\tilde{\beta} = -\operatorname{sgn}(\tilde{\alpha})\cdot\|\mathbf{a}_1\|$ and $\mathbf{y} = \mathbf{a}_1 - \tilde{\beta}\mathbf{e}_1$, we obtain the Householder reflector $\mathbf{H} = \mathbf{I}_m - \tau\mathbf{y}\mathbf{y}^T$, where $\tau = 2/(\mathbf{y}^T\mathbf{y})$. One can verify that $\mathbf{H}\cdot\mathbf{a}_1 = \tilde{\beta}\mathbf{e}_1$, hence the Householder reflector can be used to annihilate all the elements of a vector, except its first entry. The choice of sign has been made such that cancellations are avoided when computing $\mathbf{y}(1) = \mathbf{a}_1(1) - \tilde{\beta}$.

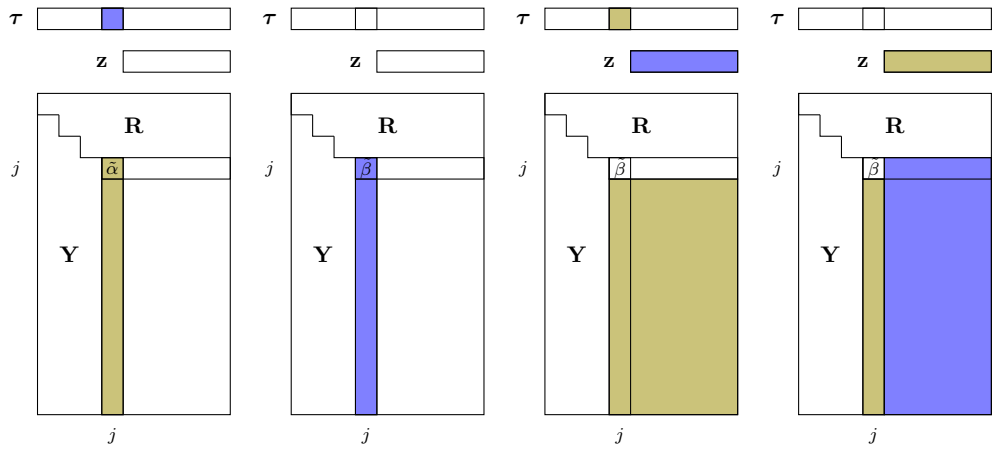
Let's note the Householder matrix used to annihilate the elements of the first column below the diagonal as $\mathbf{H}_1 = \mathbf{I}_m - \tau_1\mathbf{y}_1\mathbf{y}_1^T$. The same reasoning can be applied to the following columns of $\mathbf{A}$ to progressively annihilate the elements below its diagonal. Indeed let's consider the j-th column of $\mathbf{A}$, $\mathbf{a}_j$. To annihilate all the elements below the diagonal, we build a transformation $\mathbf{H}_j$ that does not modify the first $j-1$ elements of that column as:

$$\mathbf{H}_j \cdot \mathbf{a}_j = (\mathbf{I}_m - \tau_j\mathbf{y}_j\mathbf{y}_j^T)\cdot\mathbf{a}_j = \begin{bmatrix} \mathbf{a}_j(1:j-1) \\ \pm\|\mathbf{a}_j(j:m)\| \\ \mathbf{0}_{m-j} \end{bmatrix}, \quad \text{where } \mathbf{y}_j = \begin{bmatrix} \mathbf{0}_{j-1} \\ \mathbf{a}_j(1) + sgn(\mathbf{a}_j(1))\|\mathbf{a}_j(j:m)\|_2 \\ \mathbf{a}_j(j+1:m) \end{bmatrix}$$

where $\mathbf{0}_m$ denotes a vector of dimension m formed by zeros.

We obtain thus a factorization

$$\mathbf{H}_n \cdots \mathbf{H}_1\mathbf{A} = \begin{bmatrix} \mathbf{R} \\ \mathbf{0}_{m-n,n} \end{bmatrix} \iff \mathbf{A} = \mathbf{H}_1 \cdots \mathbf{H}_n \begin{bmatrix} \mathbf{R} \\ \mathbf{0}_{n-m,m} \end{bmatrix} = \mathbf{QR}, \tag{3.3}$$



(a) Norm of subcolumn (lines 5 and 6) (b) Scale subcolumn (lines 7 and 8) (c) Trailing matrix-vector product (line 9) (d) Outer product update (lines 10 and 11)

Figure 3.2: Major steps of Householder QR (Alg. 3.6) at iteration j. Green highlighting indicates access (reads) and blue highlighting indicates update (writes).

where $\mathbf{R}$ is upper-triangular and $\mathbf{0}_{n-m,m}$ is a matrix of dimensions $(n-m) \times m$.

The standard Householder QR algorithm is given as Alg. 3.6, and it corresponds to the algorithm implemented in LAPACK. The algorithm works column by column, computes a Householder vector whose transformation will annihilate entries below the diagonal in that column and then applies the transformation to the "trailing matrix," which is the submatrix consisting of the rows affected by the transformation and the columns that have not yet been annihilated. See Fig. 3.2 for an illustration of a step of the algorithm. In Alg. 3.6, we use the convention that the topmost nonzero entry of each Householder vector is 1 and follow LAPACK notation for the vector $\boldsymbol{\tau}$ and the scalar quantities $\tilde{\alpha}$ and $\tilde{\beta}$ in the computation of Householder vectors $\{\mathbf{y}_i\}$, which are stored as columns in a lower triangular matrix $\mathbf{Y}$. We also note that we overwrite the input matrix $\mathbf{A}$ through the course of the algorithm (for ease of presentation), so the results of the algorithm form a QR decomposition of the original input and not the status of the matrix $\mathbf{A}$ at the end of the algorithm. The output of the algorithm includes the triangular factor $\mathbf{R}$ but not the explicit orthogonal factor $\mathbf{Q}$. In many cases, we need not form $\mathbf{Q}$ (or its first n columns) explicitly, as we can apply it or its transpose to other matrices using its implicit Householder vector structure. We discuss how to generate $\mathbf{Q}$ explicitly below.

Algorithm 3.6. Householder QR.

Require: $\mathbf{A}$ is an $m \times n$ matrix with $m \geq n$
Ensure: $\hat{\mathbf{Q}}\hat{\mathbf{R}} = \mathbf{A}$ where $\hat{\mathbf{R}}$ is upper triangular $(m \times n)$, $\mathbf{R}$ is its upper triangular part $(n \times n)$, and $\hat{\mathbf{Q}} = (\mathbf{I} - \tau_1 \mathbf{y}_1 \mathbf{y}_1^T) \cdots (\mathbf{I} - \tau_n \mathbf{y}_n \mathbf{y}_n^T)$
1: **function** $[\mathbf{Y}, \boldsymbol{\tau}, \mathbf{R}] =$ HouseholderQR($\mathbf{A}$)
2: $\quad \mathbf{Y} = \mathbf{0}$, $\mathbf{R} = \mathbf{0}$
3: $\quad$ **for** $j = 1$ to n **do**
$\quad\quad$ ▷ Compute the Householder vector
4: $\quad\quad \tilde{\alpha} = \mathbf{A}(j, j)$
5: $\quad\quad \tilde{\beta} = -\operatorname{sgn}(\tilde{\alpha}) \cdot \|\mathbf{A}(j:m, j)\|_2$

Algorithm	# flops	# words	# messages
CGS	$\frac{2mn^2}{P}$	$O(n^2 + n \log P)$	$O(n \log P)$
MGS	$\frac{2mn^2}{P}$	$O(n^2 \log P)$	$O(n^2 \log P)$
Cholesky-QR	$\frac{2mn^2}{P} + \frac{n^3}{3}$	$O(n^2)$	$O(\log P)$
Householder QR	$\frac{2mn^2}{P}$	$O(n^2)$	$O(n \log P)$
TSQR	$\frac{2mn^2}{P} + \frac{2n^3}{3} \log P$	$O(n^2 \log P)$	$O(\log P)$

Table 3.1: Algorithmic costs for various parallel orthogonalization routines ($P < m/n$) Cost of CholQR assumes $n^2 \geq P$.

6: $\boldsymbol{\tau}(j) = (\tilde{\beta} - \tilde{\alpha})/\tilde{\beta}$
7: $\mathbf{Y}(j{+}1 : m, j) = 1/(\tilde{\alpha} - \tilde{\beta}) \cdot \mathbf{A}(j{+}1 : m, j)$
8: $\mathbf{R}(j, j) = \tilde{\beta}$
▷ Apply the Householder transformation to the trailing matrix
9: $\mathbf{z} = \boldsymbol{\tau}(j) \cdot (\mathbf{A}(j, j{+}1 : n) + \mathbf{Y}(j{+}1 : m, j)^T \cdot \mathbf{A}(j{+}1 : m, j{+}1 : n))$
10: $\mathbf{R}(j, j{+}1 : n) = \mathbf{A}(j, j{+}1 : n) - \mathbf{z}$
11: $\mathbf{A}(j{+}1 : m, j{+}1 : n) = \mathbf{A}(j{+}1 : m, j{+}1 : n) - \mathbf{Y}(j{+}1 : m, j) \cdot \mathbf{z}$
12: **end for**
13: **end function**

Note that it is possible to block the Householder vectors. Given n Householder vectors, the implicit structure of the corresponding orthogonal factor is

$$\hat{\mathbf{Q}} = (\mathbf{I} - \tau_1 \mathbf{y}_1 \mathbf{y}_1^T) \cdots (\mathbf{I} - \tau_n \mathbf{y}_n \mathbf{y}_n^T) = \mathbf{I} - \mathbf{Y}\mathbf{T}\mathbf{Y}^T, \tag{3.4}$$

where $\mathbf{Y}$ is a unit lower triangular matrix whose ith column is $\mathbf{y}_i$ and $\mathbf{T}$ is an upper triangular matrix that satisfies $\mathbf{Y}^T\mathbf{Y} = \mathbf{T}^{-1} + \mathbf{T}^{-T}$. The more detailed construction is explained in the slides. Thus, applying $\hat{\mathbf{Q}}^T$ to a vector can be computed by multiplying with the product $\mathbf{Y}\mathbf{T}\mathbf{Y}^T$ in an efficient order, and each operation is a matrix multiplication.

Given the implicit structure of $\hat{\mathbf{Q}}$, we can apply it to a matrix efficiently using a sequence of Householder transformations. Note that applying $\hat{\mathbf{Q}}^T$ can be done by applying the (symmetric) transformations in opposite order. In order to generate $\hat{\mathbf{Q}}$ explicitly, we can apply it to the identity matrix and further improve efficiency by exploiting the sparsity structure throughout the process. More commonly, we wish to form only the first n columns of $\hat{\mathbf{Q}}$, similar to the output of Gram-Schmidt or Cholesky-QR, and thus we apply $\hat{\mathbf{Q}}$ to the first n columns of the $m \times m$ identity matrix, which we denote by $\mathbf{I}_{m,n}$:

$$\hat{\mathbf{Q}}(1 : m, 1 : n) = (\mathbf{I} - \tau_1 \mathbf{y}_1 \mathbf{y}_1^T) \cdots (\mathbf{I} - \tau_n \mathbf{y}_n \mathbf{y}_n^T)\mathbf{I}_{m,n}.$$

Note that the first Householder transformation involves $\mathbf{y}_n$, which has zeros in the first $n - 1$ entries and therefore fills in values only in the last column of $\mathbf{I}_{m,n}$. Likewise, the ith transformation involves $\mathbf{y}_{n-i+1}$ and need be applied only to the last i columns. Exploiting this structure, the algorithm for generating the first n columns of $\mathbf{Q}$ operates nearly identically to lines 9 to 11 of Alg. 3.6, only in reverse order. The leading order costs, in terms of both computation and communication, are the same as Alg. 3.6.

The 1D parallel Householder QR approach is not discussed in details here. We only note that, as in the sequential case, the algorithm proceeds column by column. Each iteration consists of computing a norm of a subcolumn and scaling it to compute the Householder vector, followed by computing a matrix-vector product and outer-product update with the trailing matrix to apply the Householder transformation. Given the 1D distribution of the matrix across processors, each of these operations involves distributed data. We note that computing the norm of a subcolumn requires communication among all processors. Specifically it requires an all-reduce collective, and thus the number of messages exchanged by this algorithm is $O(n \log P)$. We discuss in the following a faster algorithm that reduces the number of messages exchanged during Householder QR, referred to as TSQR.

3.4.2 ▪ TSQR

The goal of the Tall-Skinny QR (TSQR) algorithm is to match simultaneously the communication cost of Cholesky-QR and the numerical stability of Householder QR. That is, we want to obtain a QR factorization while relying on a constant number of collective exchanges among processors, and use orthogonal transformations based on Householder to annihilate entries such that we preserve the numerical stability. The key idea is to exploit the flexibility in the order of annihilation for obtaining an upper triangular factor. Given that the input matrix $\mathbf{A}$ is 1D-row-distributed across processors, the first observation of the parallel TSQR algorithm is that each processor can annihilate many of its entries independently by performing a local QR decomposition. After this first step, P triangles remain, and $P-1$ of them need to be annihilated to obtain a single upper triangular factor $\mathbf{R}$. The second observation of the TSQR algorithm is that these triangles can be annihilated using a parallel reduction where the reduction operator is a QR decomposition of stacked triangles. Thus, the parallel reduction can be optimized by tuning the reduction tree. In particular, a binomial reduction tree achieves a latency cost of $O(\log P)$, which improves upon the latency cost of Householder QR. Algorithm 3.7 specifies the TSQR algorithm using a binomial tree, where we assume P is a power of two. The algorithm is illustrated in Fig. 3.3 for $P = 4$. We note that there are many possibilities for reduction trees, and they can be tuned for particular architectures and matrix dimensions. The implicit structure of the orthogonal factor depends on the shape of the reduction tree. Further, we can use all-reduction algorithms, which have the same asymptotic cost as reduction algorithms, to produce the resulting triangular factor on all processors at the end of the computation.

Consider the parallel TSQR algorithm with 4 processors and a binomial tree, as depicted in Fig. 3.3. We can express the orthogonal factor mathematically as follows. In the first step, we determine a matrix $\hat{\mathbf{Q}}^{(2)}$ such that

$$\hat{\mathbf{Q}}^{(2)^T} \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \mathbf{A}_3 \\ \mathbf{A}_4 \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{R}}_1^{(2)} \\ \hat{\mathbf{R}}_2^{(2)} \\ \hat{\mathbf{R}}_3^{(2)} \\ \hat{\mathbf{R}}_4^{(2)} \end{bmatrix}, \quad \text{where} \quad \hat{\mathbf{Q}}^{(2)} = \begin{bmatrix} \hat{\mathbf{Q}}_1^{(2)} & & & \\ & \hat{\mathbf{Q}}_2^{(2)} & & \\ & & \hat{\mathbf{Q}}_3^{(2)} & \\ & & & \hat{\mathbf{Q}}_4^{(2)} \end{bmatrix}$$

and $\mathbf{A}_I = \hat{\mathbf{Q}}_I^{(2)} \hat{\mathbf{R}}_I^{(2)}$ for each I. Here $\hat{\mathbf{Q}}_I^{(2)}$ has dimension $(m/4) \times (m/4)$ and $\hat{\mathbf{R}}_I^{(2)}$ has dimension $(m/4) \times n$. We use the notation $\mathbf{R}_I^{(2)}$ to denote the first n rows of $\hat{\mathbf{R}}_I^{(2)}$ so that $\hat{\mathbf{R}}_I^{(2)} = \begin{bmatrix} \mathbf{R}_I^{(2)} \\ \mathbf{0} \end{bmatrix}$.

In the second level of the binomial tree, we eliminate $\mathbf{R}_2^{(2)}$ and $\mathbf{R}_4^{(2)}$ in parallel. We do this by determining two orthogonal matrices that satisfy

$$\begin{bmatrix} \hat{\mathbf{Q}}_{11}^{(1)} & \hat{\mathbf{Q}}_{12}^{(1)} \\ \hat{\mathbf{Q}}_{21}^{(1)} & \hat{\mathbf{Q}}_{22}^{(1)} \end{bmatrix}^T \begin{bmatrix} \mathbf{R}_1^{(2)} \\ \mathbf{R}_2^{(2)} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_1^{(1)} \\ \mathbf{0} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \hat{\mathbf{Q}}_{33}^{(1)} & \hat{\mathbf{Q}}_{34}^{(1)} \\ \hat{\mathbf{Q}}_{43}^{(1)} & \hat{\mathbf{Q}}_{44}^{(1)} \end{bmatrix}^T \begin{bmatrix} \mathbf{R}_3^{(2)} \\ \mathbf{R}_4^{(2)} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_3^{(1)} \\ \mathbf{0} \end{bmatrix}.$$

Here, each $\hat{\mathbf{Q}}_{IJ}^{(1)}$ is $n \times n$. The corresponding sets of Householder vectors have the structure

$$\mathbf{Y}_{12}^{(1)} = \begin{bmatrix} \mathbf{I} \\ \mathbf{Y}_1^{(1)} \end{bmatrix} \quad \text{and} \quad \mathbf{Y}_{34}^{(1)} = \begin{bmatrix} \mathbf{I} \\ \mathbf{Y}_3^{(1)} \end{bmatrix},$$

where $\mathbf{Y}_1^{(1)}$ and $\mathbf{Y}_3^{(1)}$ are each $n \times n$ and upper triangular. Mathematically, we have

$$\hat{\mathbf{Q}}^{(1)^T} \begin{bmatrix} \hat{\mathbf{R}}_1^{(2)} \\ \hat{\mathbf{R}}_2^{(2)} \\ \hat{\mathbf{R}}_3^{(2)} \\ \hat{\mathbf{R}}_4^{(2)} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_1^{(1)} \\ \mathbf{0} \\ \mathbf{R}_3^{(1)} \\ \mathbf{0} \end{bmatrix},$$

where

$$\hat{\mathbf{Q}}^{(1)} = \begin{bmatrix} \hat{\mathbf{Q}}_{11}^{(1)} & & \hat{\mathbf{Q}}_{12}^{(1)} & & & & & \\ & \mathbf{I} & & & & & & \\ \hat{\mathbf{Q}}_{21}^{(1)} & & \hat{\mathbf{Q}}_{22}^{(1)} & & & & & \\ & & & \mathbf{I} & & & & \\ & & & & \hat{\mathbf{Q}}_{33}^{(1)} & & \hat{\mathbf{Q}}_{34}^{(1)} & \\ & & & & & \mathbf{I} & & \\ & & & & \hat{\mathbf{Q}}_{43}^{(1)} & & \hat{\mathbf{Q}}_{44}^{(1)} & \\ & & & & & & & \mathbf{I} \end{bmatrix}.$$

To get to the root of the tree, the final step consists of determining a $2n \times 2n$ orthogonal matrix that satisfies

$$\begin{bmatrix} \hat{\mathbf{Q}}_{11}^{(0)} & \hat{\mathbf{Q}}_{13}^{(0)} \\ \hat{\mathbf{Q}}_{31}^{(0)} & \hat{\mathbf{Q}}_{33}^{(0)} \end{bmatrix}^T \begin{bmatrix} \mathbf{R}_1^{(1)} \\ \mathbf{R}_3^{(1)} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_1^{(0)} \\ \mathbf{0} \end{bmatrix}$$

(with the same Householder vector structure as the 2nd step) so that

$$\hat{\mathbf{Q}}^{(0)^T} \begin{bmatrix} \mathbf{R}_1^{(1)} \\ \mathbf{0} \\ \mathbf{R}_3^{(1)} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_1^{(0)} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \quad \text{where} \quad \hat{\mathbf{Q}}^{(0)} = \begin{bmatrix} \hat{\mathbf{Q}}_{11}^{(0)} & & \hat{\mathbf{Q}}_{13}^{(0)} & \\ & \mathbf{I} & & \\ \hat{\mathbf{Q}}_{31}^{(0)} & & \hat{\mathbf{Q}}_{33}^{(0)} & \\ & & & \mathbf{I} \end{bmatrix}.$$

Thus, we obtain an orthogonal factor represented mathematically as $\hat{\mathbf{Q}} = \hat{\mathbf{Q}}^{(2)}\hat{\mathbf{Q}}^{(1)}\hat{\mathbf{Q}}^{(0)}$.

In general, the orthogonal factor $\hat{\mathbf{Q}}^{(\log P)}$ is a block diagonal with diagonal blocks denoted by $\hat{\mathbf{Q}}_I^{(\log P)}$ corresponding to the QR factorization of $\mathbf{A}_I$. Each diagonal block $\hat{\mathbf{Q}}_I^{(\log P)}$ is represented using a set of Householder vectors $\mathbf{Y}_I^{(\log P)}$. For $k < \log P$, the orthogonal factor $\hat{\mathbf{Q}}^{(k)}$ encodes the elimination at step k, which is a collection of QR decompositions of stacked triangles. If processor I is involved in the elimination at step k, then it stores the upper triangular Householder vector information in the matrix $\mathbf{Y}_I^{(k)}$.

The parallel algorithm is specified in Alg. 3.7, where we assume for simplicity that P is a power of two. A visualization of the binomial tree for $P = 4$ is given in Fig. 3.3. When P is not a power of two, the pseudocode is complicated slightly, but the asymptotic costs are the same. As explained above, the first step involves independent QR decompositions of local data (line 3), which is of size $(m/P) \times n$. The remaining loop specifies the binomial reduction tree, where after each loop iteration, half the processors drop out of the computation. Each processor with work to do at a given iteration receives triangular data from its partner processor (line 8) and performs a QR decomposition of stacked triangles (line 10).

The computational cost of the initial QR of local data is $2mn^2/P + O(n^3)$, and the cost of a QR decomposition of two stacked $n \times n$ triangles is $2n^3/3 + O(n^2)$ taking the triangular structure into account. The communication cost of the algorithm is that of sending an $n \times n$ triangular matrix between partner processors at each level of the tree. As the depth of the tree is $O(\log P)$, we obtain the costs given in Tab. 3.1.

In order to apply $\hat{\mathbf{Q}}$ (or $\hat{\mathbf{Q}}^T$) to a matrix, we apply each step in sequence, using the implicit Householder structure of each $\hat{\mathbf{Q}}^{(k)}$ and obtaining the same parallelism as when each factor was computed. The explicit representation of the first n columns of $\hat{\mathbf{Q}}$ can be generated in the same computation and communication cost of the TSQR algorithm.

Algorithm 3.7. Parallel TSQR (binomial tree).

Require: $\mathbf{A}$ is an $m \times n$ matrix 1D-row-distributed over power-of-two P processors
Ensure: $\hat{\mathbf{Q}}\mathbf{R} = \mathbf{A}$ where $\mathbf{R}$ is upper triangular and $\hat{\mathbf{Q}} = \hat{\mathbf{Q}}^{(\log P)} \cdots \hat{\mathbf{Q}}^{(0)}$
Ensure: $\mathbf{R}$ is stored on processor 1 and each $\mathbf{Y}^{(k)}$ is distributed across 2^k processors

1: **function** $\left[\left\{\mathbf{Y}_I^{(k)}\right\}, \mathbf{R}\right] =$ ParTSQR($\mathbf{A}$)
2: $I =$ MyProcID()
3: $\left[\mathbf{Y}_I^{(\log P)}, \mathbf{R}_I^{(\log P)}\right] =$ HouseholderQR($\mathbf{A}_I$) ▷ Eliminate lower triangle of local block
4: **for** $k = \log P - 1$ down to 0 **do**
5: Break if I doesn't have a partner proc
6: Determine J, partner proc ID
7: **if** $I > J$ **then**
8: Send $\mathbf{R}_I^{(k+1)}$ to processor J
9: **else**
10: Receive $\mathbf{R}_J^{(k+1)}$ from processor J
11: $\left[\mathbf{Y}_I^{(k)}, \mathbf{R}_I^{(k)}\right] =$ HouseholderQR$\left(\begin{bmatrix}\mathbf{R}_I^{(k+1)} \\ \mathbf{R}_J^{(k+1)}\end{bmatrix}\right)$ ▷ Eliminate Jth triangle
12: **end if**
13: **end for**
14: **if** $I = 1$ **then**
15: $\mathbf{R} = \mathbf{R}_1^{(0)}$
16: **end if**
17: **end function**

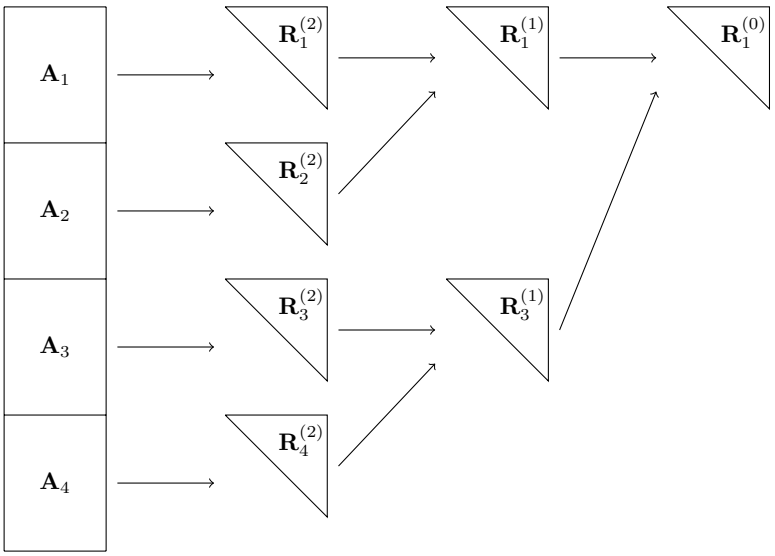


Figure 3.3: Visualization of the binomial elimination tree used by parallel TSQR (Alg. 3.7)

3.5 ▪ Numerical Stability

In finite precision arithmetic, the computation of the QR factorization of a matrix $\mathbf{A}$ is, of course, not exact. There are generally two main quantities we can consider when discussing the numerical stability of orthogonalization methods. The first is the orthogonality error, or the *loss of orthogonality*, typically measured as $\|\mathbf{I}-\hat{\mathbf{Q}}^T\hat{\mathbf{Q}}\|$. This quantity measures how close the matrix $\hat{\mathbf{Q}}$ is to being orthogonal, which can affect the behavior of downstream applications such as the convergence of Krylov subspace methods. This quantity is also refected by the condition number of $\hat{\mathbf{Q}}$, that is $\kappa(\hat{\mathbf{Q}}) = \sigma_{max}(\hat{\mathbf{Q}})/\sigma_{min}(\hat{\mathbf{Q}})$, where $\sigma_{max}(\hat{\mathbf{Q}}), \sigma_{min}\hat{\mathbf{Q}}$ are the largest and the smallest singular values of $\hat{\mathbf{Q}}$, respectively.

Another quantity that is important is the residual norm or the *decomposition error*, measured as $\|\mathbf{A}-\hat{\mathbf{Q}}\mathbf{R}\|$, which describes how close the computed $\hat{\mathbf{Q}}\mathbf{R}$ is to $\mathbf{A}$. Generally, most reasonable approaches to QR factorization produce a residual norm on the order of machine epsilon times the norm of the input matrix. Guarantees on the orthogonality of $\hat{\mathbf{Q}}$, however, can vary greatly amongst algorithms. We briefly discuss the different numerical stability properties of the algorithms presented in this section in order to motivate their inclusion; these properties are summarized in Table 3.2.

3.5.1 ▪ Gram-Schmidt

Gram-Schmidt orthogonalization, discussed in Section 3.2, is accomplished via a sequence of projections. In step j, the next orthogonal vector is produced by subtracting from the jth column of $\mathbf{A}$ the orthogonal projection of the jth column of $\mathbf{A}$ onto the previously computed orthogonal vectors. The difference between classical Gram-Schmidt (CGS) and modified Gram-Schmidt (MGS) is that the computations are performed in a different order. Whereas in CGS, the next vector is computed as $(\mathbf{I} - \hat{\mathbf{Q}}(:, 1 : j-1)\hat{\mathbf{Q}}(:, 1 : j-1)^T)\mathbf{A}(:, j)$, in MGS the computation is performed via $(\mathbf{I}-\hat{\mathbf{Q}}(:, j-1)\hat{\mathbf{Q}}(:, j-1)^T)\cdots(\mathbf{I}-\hat{\mathbf{Q}}(:, 1)\hat{\mathbf{Q}}(:, 1)^T)\mathbf{A}(:, j)$. Although the approaches are mathematically equivalent, this seemingly small change makes a significant difference numerically.

Algorithm	$\|\mathbf{I} - \mathbf{Q}^T\mathbf{Q}\|$	Constraint	References
CGS	$O(\varepsilon)\kappa^2(\mathbf{A})$	$O(\varepsilon)\kappa^2(\mathbf{A}) < 1$	[4]
MGS	$O(\varepsilon)\kappa(\mathbf{A})$	$O(\varepsilon)\kappa(\mathbf{A}) < 1$	[1]
Cholesky-QR	$O(\varepsilon)\kappa^2(\mathbf{A})$	$O(\varepsilon)\kappa^2(\mathbf{A}) < 1$	[10]
Householder QR	$O(\varepsilon)$	none	[8]
TSQR	$O(\varepsilon)$	none	[3],[6]

Table 3.2: Various orthogonalization routines and their stability in terms of loss of orthogonality, associated constraints on condition number, and references. Note that there are dimensional constants hidden in the $O(\varepsilon)$ factors.

The bound on the loss of orthogonality for CGS is due to Giraud, Langou, Rozložník, and van den Eshof with a constraint: if $O(\varepsilon)\kappa^2(\mathbf{A}) < 1$, then $\|\mathbf{I} - \hat{\mathbf{Q}}^T\hat{\mathbf{Q}}\| \leq O(\varepsilon)\kappa^2(\mathbf{A})$ [4]. The reordering of computations in MGS leads to a much more stable algorithm than CGS without incurring any extra arithmetic cost. For MGS, Björck has shown that if $O(\varepsilon)\kappa(\mathbf{A}) < 1$, then $\|\mathbf{I} - \hat{\mathbf{Q}}^T\hat{\mathbf{Q}}\| \leq O(\varepsilon)\kappa(\mathbf{A})$ [1].

3.5.2 ▪ Cholesky-QR

In contrast to iterative, projection-based approaches like Gram-Schmidt, the Cholesky-QR approach is based on the fact that in exact arithmetic, the $\mathbf{R}$ in the QR decomposition of $\mathbf{A}$ is the Cholesky factor of $\mathbf{A}^T\mathbf{A}$. As explained in § 3.3, the Cholesky-QR algorithm (abbreviated CholQR) first computes $\mathbf{A}^T\mathbf{A}$, computes its Cholesky factor to find $\mathbf{R}$, and then computes $\hat{\mathbf{Q}} = \mathbf{A}\mathbf{R}^{-1}$. It is clear that explicitly forming $\mathbf{A}^T\mathbf{A}$ can have consequences numerically; it is shown in [10] that if $O(\varepsilon)\kappa^2(\mathbf{A}) < 1$, then the loss of orthogonality is bounded by $O(\varepsilon)\kappa^2(\mathbf{A})$.

3.5.3 ▪ Householder QR and TSQR

The Householder QR algorithm is often considered to be the canonical numerically stable approach to computing a QR decomposition. This nice numerical behavior stems from the fact that this approach involves a series of orthogonal transformations; see Section 3.4.1 for details. The study of the stability of Householder transformations and Householder QR goes back to the work of Wilkinson; see [8] and [9]. In summary, the loss of orthogonality in Householder QR is bounded by $O(\varepsilon)$, and this algorithm is unconditionally stable. In other words, the algorithm will return a factor $\hat{\mathbf{Q}}$ that is orthogonal to working precision regardless of the conditioning of the input matrix $\mathbf{A}$.

Like Householder QR, TSQR produces a $\hat{\mathbf{Q}}$ factor that is orthogonal to working precision with no constraint on the conditioning of $\mathbf{A}$ [3],[6].

Chapter 4

Matrix Multiplication

We consider a fundamental computation in numerical linear algebra: *classical matrix multiplication*. By "classical", we mean that the multiplication of an $m \times n$ matrix $\mathbf{A}$ and an $n \times r$ matrix $\mathbf{B}$ is performed via the mathematical definition and excludes faster algorithms as Strassen. To simplify later algorithms and analysis, we consider the form of matrix multiplication that updates existing entries of a matrix: $\mathbf{C} := \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$. In general, this can be expressed as

$$\mathbf{C}(i,j) = \mathbf{C}(i,j) + \sum_{k=1}^{n} \mathbf{A}(i,k) \cdot \mathbf{B}(k,j), \tag{4.1}$$

where the order of summation and the order of computing output entries is unspecified. Thus, when $m = n = r$, all classical algorithms perform exactly $2n^3$ flops, but the amount of communication the algorithms perform can vary widely.

We first present lower bounds on communication, in the sequential case with a two-level memory model, along with the proof, and then we discuss several algorithms along with their communication cost analysis.

4.1 ▪ Lower Bounds

4.1.1 ▪ Sequential case

Here we focus on the sequential two-level memory model, where we have two levels of memory, a fast memory of size M words and a slow memory of infinite size. Computations are performed on data that resides in fast memory, and data is transferred between slow and fast memory. The idea of the lower bound is to establish a minimum number of reads and writes that applies to any order of evaluation of the summations given by eq. (4.1), and we use a geometric inequality that applies to both sequential and parallel models.

Theorem 4.1.1 *Assume that we want to compute $\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$, where $\mathbf{A}$ is $m \times n$ and $\mathbf{B}$ is $n \times r$, and that the memory locations storing entries of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ do not overlap. Then the bandwidth cost lower bound for classical matrix multiplication on a sequential machine is*

$$W \geq \frac{2mnr}{\sqrt{M}} - 2M,$$

where M is the size of the fast memory.

Proof. The general idea of the proof is as follows. We consider the classical matrix multiplication algorithm as a stream of instructions involving computations and memory operations: loads and stores between fast and slow memory. We break this instruction stream into segments, where each segment contains exactly x load and store instructions, where x is a parameter we can optimize to obtain the tightest lower bound. We will then derive an upper bound F on the number of scalar multiplications that can be performed during any given segment using a geometric inequality. Then, we derive a lower bound on the number of complete segments by dividing by the total number of operations mnr by F, giving $\lfloor mnr/F \rfloor$. Finally, we can bound from below the total number of loads and stores needed by taking the product of x (the number of loads/stores per segment) times the minimum number of complete segments, giving $x\lfloor mnr/F \rfloor$, and choosing a particular x.

We note that in order to compute the scalar multiplication $\mathbf{A}(i,k) \cdot \mathbf{B}(k,j)$ for a particular (i,k,j), we require that $\mathbf{A}(i,k)$, $\mathbf{B}(k,j)$, as well as the output operand (a variable for accumulating the sum of the k scalar multiplications) reside in fast memory. Now, consider the number of input/output operands that are available in fast memory during a given segment. At the start of the segment, there can be at most M distinct operands in fast memory. During the segment, there can be at most x additional operands read into fast memory (since a segment contains exactly x load/store operations). Since there are at most M words in fast memory at the start of a segment and at most x store operations, we have a maximum of $M + x$ distinct operands available. Recall that we have assumed that the memory locations storing entries of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ cannot overlap, so this means that

$$\#\text{ of entries of } \mathbf{A} + \#\text{ of entries of } \mathbf{B} + \#\text{ of entries of } \mathbf{C} \leq M + x. \tag{4.2}$$

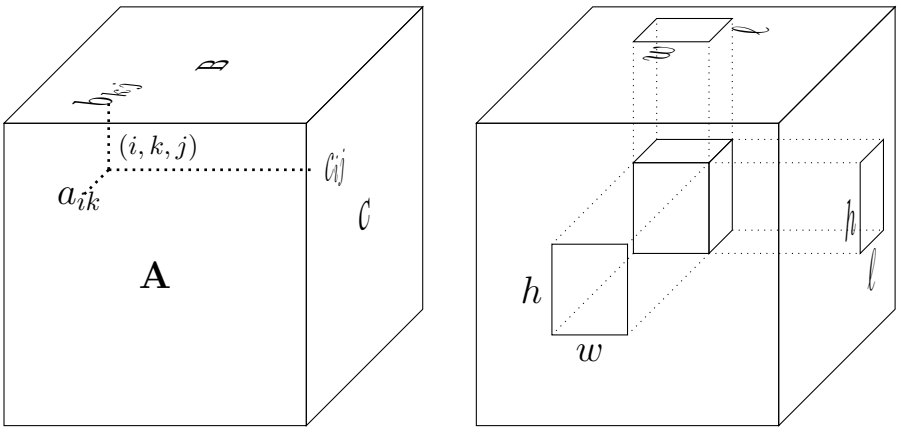
This inequality provides an upper bound on the amount of data available in a segment, which we want to leverage to establish an upper bound on the number of operations F that can be performed on that data. The key to this step is to use geometry: in the case of matrix multiplication the iteration space (i.e., the computation) is three dimensional and the matrices (i.e., the data) are two dimensional, with a clear relationship between them. We state the key geometric inequality in Lemma 4.1.2, visualize it in Fig. 4.1, and apply it to matrix multiplication below.

Lemma 4.1.2 (Loomis and Whitney [5]) *Let V be a finite set of lattice points in $\mathbb{Z}^3$, that is, points (x,y,z) with integer coordinates. Let V_x be the projection of V in the x-direction, that is, all points (y,z) such that there exists an x' such that $(x',y,z) \in V$. Define V_y and V_z similarly. Let $|\cdot|$ denote the cardinality of a set. Then*

$$|V| \leq \sqrt{|V_x||V_y||V_z|}.$$

One can gain intuition for the inequality by first picturing a rectangular prism with dimensions $w \times h \times \ell$. In this case, the volume V is $wh\ell$. The areas of the three types of faces are wh, $w\ell$, and $h\ell$, and the square root of the product of those areas is $\sqrt{wh \cdot w\ell \cdot h\ell} = wh\ell$, the same as the volume. Lemma 4.1.2 implies that the rectangular prism is the most efficient shape for maximizing volume subject to its projections; more amorphous blobs will yield strict inequality.

Let the set of lattice points (i,j,k) represent the scalar multiplications. For a given segment, let V be the set of indices (i,j,k) of the scalar multiplications performed during the segment, let $V_{\mathbf{C}}$ be the set of memory locations storing the entries of $\mathbf{C}$ for the set of indices (i,j), let $V_{\mathbf{A}}$ be the set of memory locations storing the entries of $\mathbf{A}$ for the set



(a) One lattice point and its projections (b) Rectangular prism and its projections

Figure 4.1: Loomis-Whitney cube visualization: interior lattice points correspond to scalar multiplications and projections onto the faces correspond to data required.

of indices (i, k), and let $V_{\mathbf{B}}$ be the set of memory locations storing entries of $\mathbf{B}$ for the set of indices (k, j). These sets of memory locations are exactly the projections of the computation in the three coordinate directions, and their cardinality is the number of entries of each matrix that are accessible in the segment. Then applying Lemma 4.1.2, the inequality of arithmetic and geometric means (AM-GM), and eq. (4.2), we have

$$|V| \leq \sqrt{|V_{\mathbf{A}}||V_{\mathbf{B}}||V_{\mathbf{C}}|} \leq \left(\frac{|V_{\mathbf{A}}| + |V_{\mathbf{B}}| + |V_{\mathbf{C}}|}{3}\right)^{3/2} \leq \left(\frac{M+x}{3}\right)^{3/2} \equiv F$$

total scalar multiplications per segment. Then we need at least

$$\left\lfloor \frac{mnr}{\left(\frac{M+x}{3}\right)^{3/2}} \right\rfloor$$

segments. Thus the total number of loads and stores over all segments is lower bounded by

$$x \left\lfloor \frac{mnr}{\left(\frac{M+x}{3}\right)^{3/2}} \right\rfloor,$$

and choosing $x = 2M$, which would be the maximizer above absent the floor function, we get the lower bound of

$$2M \left\lfloor \frac{mnr}{M^{3/2}} \right\rfloor \geq \frac{2mnr}{\sqrt{M}} - 2M$$

words moved between fast and slow memory, which proves Theorem 4.1.1. □

4.1.2 ▪ Parallel Case

We now consider the distributed-memory parallel memory model, where each of P processors has a local memory of size M. For matrix multiplication, we must have that $PM = \Omega(n^2)$ in order to store the input and output matrices.

Theorem 4.1.1 in § 4.1 establishes the communication lower bound for the sequential case. We note that the argument applies not only to the entire matrix multiplication, but also to any subset of the computation, where the only update to the argument is the value of the total number of scalar multiplications. In the case of parallel matrix multiplication of $m \times n$ and $n \times r$ matrices, some processor must perform at least mnr/P scalar multiplications. Applying the argument to that processor, and interpreting fast memory as that processor's local memory, we obtain the following memory-dependent parallel lower bound.

Theorem 4.1.3 *The bandwidth cost lower bound for classical matrix multiplication of $m \times n$ and $n \times r$ matrices on a parallel machine with P processors, each with local memory of size M, is*

$$W \geq \frac{2mnr}{P\sqrt{M}} - M.$$

Assuming that the matrices are square ($m = n = r$) and there is just enough memory to store in the input and output matrices, i.e., $M = \Theta(n^2/P)$, Theorem 4.1.3 gives that $W = \Omega(n^2/P^{1/2})$. If $M = \Theta(n^2/P^{2/3})$ words of fast memory are available, then there is enough memory to replicate the matrices $O(P^{1/3})$ times. This leads to a reduction in the required communication; the lower bound in this case becomes $W = \Omega(n^2/P^{2/3})$. If there is effectively infinite memory, then the lower bound degenerates to zero. However it can be shown that the communication cannot decrease more than $W = \Omega(n^2/P^{2/3})$, since otherwise one of the processors will need to perform too much computation.

4.2 ▪ Parallel algorithms

We now consider parallel algorithms implementing classical matrix multiplication $\mathbf{C} = \mathbf{AB}$. We present here the Scalable Universal Matrix Multiplication Algorithm (SUMMA) in § 4.2.1. SUMMA attains the parallel lower bounds to within a factor of $\log P$ when there is only enough memory for one or two copies of the data. However, the matrices may be sized such that we can afford to store more than one copy in memory, and in that case, SUMMA communicates more than necessary. Algorithms running on machines with $\Omega(n^2/P^{2/3})$ memory can attain the bound in **??** (which is tighter than Theorem 4.1.3 when M is large), and we present one such "3D" algorithm in § 4.2.2.

4.2.1 ▪ SUMMA

We first consider the Scalable Universal Matrix Multiplication Algorithm (SUMMA). While one of the advantages of SUMMA is that it is easily extended to rectangular matrices, for simplicity we present it in Alg. 4.1 for square matrices and assume a square processor grid. In this case, the algorithm requires that the input matrices $\mathbf{A}$ and $\mathbf{B}$ start out distributed in a 2D distribution across a $\sqrt{P} \times \sqrt{P}$ processor grid. This means that each processor owns a local submatrix of each input matrix of dimension $n_\ell \times n_\ell$ with $n_\ell = n/\sqrt{P}$, and we use the notation (I, J) to index the processor that owns submatrices $\mathbf{A}_{IJ}$ and $\mathbf{B}_{IJ}$ in block row I and block column J. Likewise, at the end of the algorithm, processor (I, J) will own submatrix $\mathbf{C}_{IJ}$ of the output matrix.

The outer-product formulation of matrix multiplication considers the output matrix as a sum of outer products of corresponding columns and rows of the input matrices: $\mathbf{C} = \mathbf{C} + \sum_k \mathbf{A}(:, k)\mathbf{B}(k, :)$. The idea of SUMMA is to perform each outer product in parallel, where the result of the outer product is distributed and accumulated across the entire processor grid. In the standard formulation of SUMMA, the output matrix is never

communicated, so the vectors of the outer product must be shared across processors so that they can be used to update the local submatrix of $\mathbf{C}$. The algorithm typically blocks the n outer products into blocks of size $b \geq 1$, so that outer products, or rank-one updates, become rank-b updates. This tuning parameter b trades off the temporary memory requirement with the number of messages sent among the processors, and it also affects the performance of the local matrix multiplications.

To understand the communication pattern, consider the extreme case of choosing $b = n_\ell = n/\sqrt{P}$. In this case, the inner loop starting at line 4 collapses to a single iteration and the subindexing of blocks $\mathbf{A}_{IK}$ and $\mathbf{B}_{KJ}$ in lines 5 and 6 simplify to $(:,:)$, or including the entire blocks. Thus, the algorithm works in $\sqrt{P}$ steps, where at step K, each processor in the Kth column of the processor grid broadcasts its local block of $\mathbf{A}$ to all processors in its processor row. Likewise, each processor in the Kth processor row broadcasts its local block of $\mathbf{B}$ to all processors in its processor column. After this communication, each processor (I, J) temporarily stores blocks $\mathbf{A}_{IK}$ and $\mathbf{B}_{KJ}$ and can multiply them and accumulate the result into its local $\mathbf{C}_{IJ}$. After all $\sqrt{P}$ steps of the algorithm are complete, each processor will have fully computed the block inner product to obtain its local submatrix of $\mathbf{C}$. Note that the temporary space required to run the algorithm with $b = n_\ell$ is as much as it takes to store the input matrices.

In order to reduce the memory footprint, SUMMA allows the outer products to be blocked into smaller groups of size $b < n_\ell$. In this case, the size of temporaries $\mathbf{A}_{\text{tmp}}$ and $\mathbf{B}_{\text{tmp}}$ is reduced from n_ℓ^2 to bn_ℓ. The tradeoff is that more broadcasts of smaller data will be performed, which increases the latency cost of the algorithm (the bandwidth cost is not affected by b). We also note that the local matrix multiplications occur between matrices of dimensions $n_\ell \times b$ and $b \times n_\ell$, and the possible cache re-use of matrix entries is limited to $O(b)$, which will adversely affect local performance when b is much smaller than the square root of the cache size.

Figure 4.2 shows the data involved at a particular iteration K and k. The highlighted data in matrix $\mathbf{A}$ is the subset of columns involved in the global rank-b update; each processor owning a subset of the rows of this block column broadcasts it to all other processors in its row (line 5). The highlighted data in matrix $\mathbf{B}$ is the subset of row involved in the global rank-b update; each processor owning a subset of the columns of this block row broadcasts it to all other processors in its column (line 6). We highlight one particular processor's local computation in matrix $\mathbf{C}$: after receiving the submatrices corresponding to the global rank-b update it needs, it performs local matrix multiplication and accumulates the result into its local submatrix of $\mathbf{C}$ (line 7).

The communication cost of this algorithm is the cost of the broadcasts in lines 5 and 6. Each involves data of size $n_\ell \times b$, so using an efficient broadcast algorithm, the cost of each is given by Tab. 1.1: $\beta \cdot 2bn_\ell + \alpha \cdot \log P$, assuming $P \leq b^2 n_\ell^2$. The two broadcasts are performed $\sqrt{P} \cdot \frac{n_\ell}{b} = \frac{n}{b}$ times for a total communication cost of

$$\beta \cdot 4\frac{n^2}{\sqrt{P}} + \alpha \cdot 2\frac{n}{b}\log P.$$

The computation cost of line 6 is $\gamma \cdot 2n_\ell^2 b$, and it is performed $\frac{n}{b}$ times, so the overall computation cost is $\gamma \cdot 2\frac{n^3}{P}$. Note that the computation is perfectly load balanced and the communication cost attains the lower bound of Theorem 4.1.3 assuming the memory size is limited to $M = O\left(\frac{n^2}{P}\right)$ (at most a constant factor more than memory required to store the input and output matrices).

Algorithm 4.1. Scalable Universal Matrix Multiplication Algorithm (SUMMA).

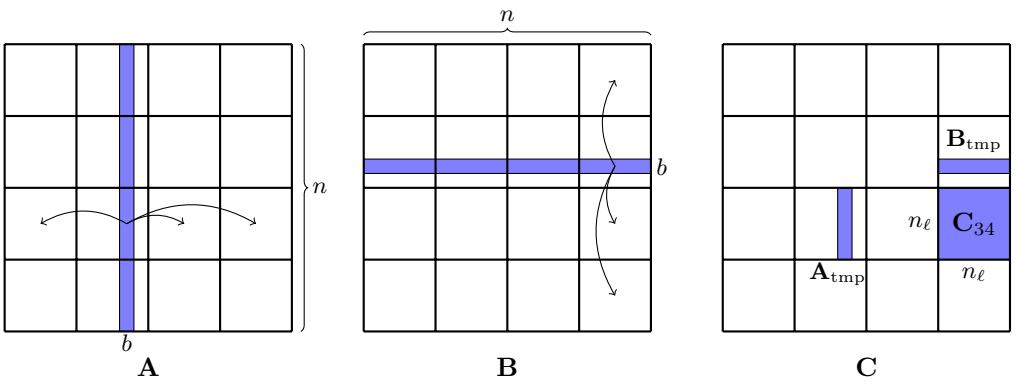


Figure 4.2: Illustration of one of SUMMA's inner loops as viewed by processor $(3, 4)$ of a 4×4 processor grid, depicting the broadcast of block of $\mathbf{A}$ in processor row 3, the broadcast of block of $\mathbf{B}$ in processor column 4, and the local rank-b update on processor $(3, 4)$.

Require: $\mathbf{A}, \mathbf{B}$ are $n \times n$ matrices in identical 2D block distribution across processors
Require: Processors arranged in $\sqrt{P} \times \sqrt{P}$ grid where $n_\ell = n/\sqrt{P}$ is an integer
Require: Processor (I, J) owns $n_\ell \times n_\ell$ submatrix

$$\mathbf{M}_{IJ} = \mathbf{M}((I-1)n_\ell+1 : In_\ell, (J-1)n_\ell+1 : Jn_\ell)$$

Require: Block size b divides n_ℓ evenly
Ensure: $\mathbf{C} = \mathbf{C}+\mathbf{AB}$ is $n \times n$ matrix in identical 2D block distribution across processors

```
1: function C = SUMMA(C, A, B, b)
2:     (I, J) = MyProcID()
3:     for K = 1 to √P do
4:         for k = 1 to n_ℓ / b do
5:             Proc(I, K) broadcasts A_IK(:, (k−1)b+1:kb) to Proc(I, :), store in A_tmp
6:             Proc(()K, J) broadcasts B_KJ((k−1)b+1:kb, :) to Proc(:, J), store in B_tmp
7:             C_IJ = C_IJ + A_tmp · B_tmp
8:         end for
9:     end for
10: end function
```

4.2.2 ▪ 3D Algorithm (square case)

While SUMMA is a communication-optimal algorithm when the memory is limited to $M = O(n^2/P)$, the lower bound of Theorem 4.1.3 suggests that a larger local memory admits a smaller lower bound. Indeed, 3D algorithms for matrix multiplication exploit this memory-communication tradeoff to reduce communication at the expense of a larger memory footprint. We present a simple variant of 3D matrix multiplication for square matrices here.

For intuition, we note that the name "3D" stems from the fact the iteration space is three dimensional and that 3D algorithms parallelize the computation in each of the three dimensions. SUMMA follows an "owner computes" rule so that the only processor that

performs scalar multiplications that contribute to a particular entry in $\mathbf{C}$ is the processor that owns that entry of $\mathbf{C}$. Visualizing the iteration space as the Loomis-Whitney cube (Fig. 4.1), SUMMA parallelizes over dimensions corresponding to both $\mathbf{A}$ and $\mathbf{B}$, but not $\mathbf{C}$. That is, different processors must share entries of $\mathbf{A}$ and $\mathbf{B}$ to perform their local computations, but no communication is required of entries of $\mathbf{C}$; thus, SUMMA can be considered a 2D algorithm. 3D algorithms, on the other hand, parallelize across all three dimensions of the iteration space and require communication of all three matrices.

We present the 3D algorithm in Alg. 4.2 and visualize the steps of the algorithm in Fig. 4.3. The algorithm assumes a rectangular block distribution (as opposed to a square block distribution as used by SUMMA). Additionally, we note that the distributions of the three matrices are not identical. These distribution assumptions are somewhat arbitrary because an algorithm could perform redistribution from other reasonable distributions without affecting the leading order communication costs, but the assumptions simplify the presentation of the algorithm. In particular, we assume that each matrix is distributed over a $\sqrt[3]{P} \times (\sqrt[3]{P})^2$ processor grid, so that each processor owns submatrices of dimension $n_\ell \times n_b$, where $n_\ell = n/\sqrt[3]{P}$ and $n_b = n/(\sqrt[3]{P})^2$. We use the notation $\mathbf{M}_{IJ}$ to denote the (I, J) submatrix of matrix $\mathbf{M}$ with dimensions $n_\ell \times n_\ell$ and the notation $\mathbf{M}_{IJK}$ to denote the Kth column block of $\mathbf{M}_{IJ}$ with dimension $n_\ell \times n_b$. Considering the processors to be organized into a logical $\sqrt[3]{P} \times \sqrt[3]{P} \times \sqrt[3]{P}$ processor grid, we assign processor (I, J, K) the blocks $\mathbf{A}_{IKJ}$, $\mathbf{B}_{KJI}$, and $\mathbf{C}_{IJK}$. The initial distribution of $\mathbf{A}$ and $\mathbf{B}$ is presented in Fig. 4.3b.

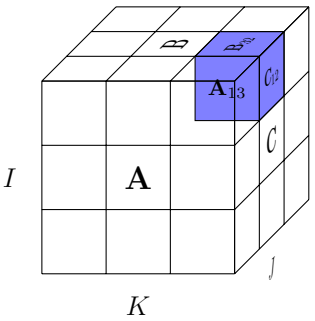
Given the input and output matrix distributions, the algorithm proceeds in four steps: gather the necessary entries of $\mathbf{A}$, gather the necessary entries of $\mathbf{B}$, perform local matrix multiplication, and sum the corresponding entries of $\mathbf{C}$. Processor (I, J, K) is responsible for the matrix multiplication $\mathbf{A}_{IK}\mathbf{B}_{KJ}$, which is accumulated into output submatrix $\mathbf{C}_{IJ}$. To obtain the submatrix $\mathbf{A}_{IK}$, processor (I, J, K) performs an all-gather collective with processors that also own subsets of entries of that submatrix, those that share indices I and K. Similarly, to obtain the submatrix $\mathbf{B}_{KJ}$, processor (I, J, K) performs an all-gather collective with processors that also own subsets of entries of that submatrix, those that share indices J and K. These two steps are shown in Fig. 4.3c. After performing the local computation, processor (I, J, K) performs the summation with other processors that share the indices I and J; to obtain a final distribution so that each processor owns n^2/P entries, a reduce-scatter collective operation is used. This step is shown in Fig. 4.3d.

The costs of Alg. 4.2 is that of the three communication collectives and the local matrix multiplication, each of which involves matrices of dimension $n_\ell \times n_\ell$. Thus, from Tab. 1.1, the cost is given by

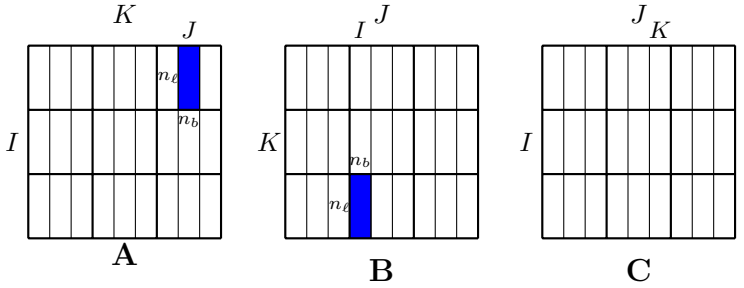
$$\gamma \cdot 2\frac{n^3}{P} + \beta \cdot 3\frac{n^2}{P^{2/3}} + \alpha \cdot \log P.$$

The memory required for the algorithm is that of storing $\mathbf{A}_{IK}$, $\mathbf{B}_{KJ}$, and $\bar{\mathbf{C}}_{IJ}$, which is $3n^2/P^{2/3}$. Thus, Alg. 4.2 attains the lower bounds specified by Theorem 4.1.3 and **??**.

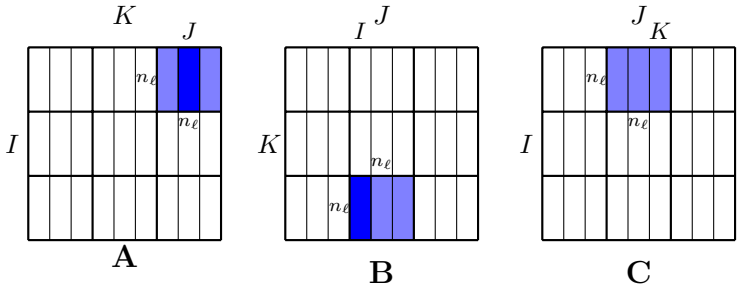
In the case that there exists substantially more than $O(n^2/P)$ memory but not the $O(n^2/P^{2/3})$ required by Alg. 4.2, 3D algorithms can be adapted to use as much memory as is available, continuously navigating the memory-communication tradeoff and obtaining the memory-dependent lower bound (Theorem 4.1.3), which dominates the memory-independent bound (**??**) in this case. For example, we can parametrize by $\alpha \in [0, 1/6]$ and obtain bandwidth cost $O(n^2/P^{2/3-\alpha})$ and latency cost $O(P^{3\alpha})$ with $O(n^2/P^{2\alpha+2/3})$ memory.



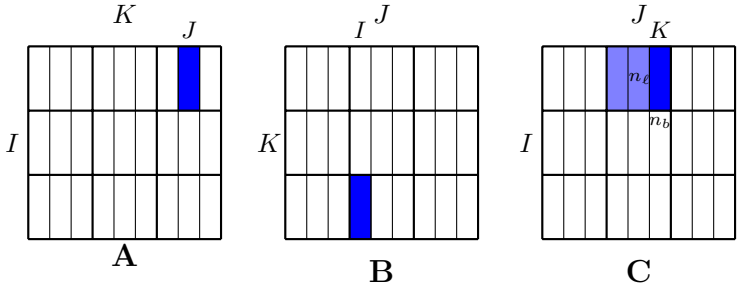
(a) Visualization of computation assigned to processor $(1, 2, 3)$.



(b) Initial distribution of $\mathbf{A}$ and $\mathbf{B}$. Processor $(1, 2, 3)$ initially owns $\mathbf{A}_{132}$ and $\mathbf{B}_{321}$, each of dimension $n_\ell \times n_b$.



(c) All-gathers and local computation. Processor $(1, 2, 3)$ participates in an all-gather of $\mathbf{A}$ with processors $(1, :, 3)$ and in an all-gather of $\mathbf{B}$ with processors $(:, 2, 3)$, each of dimension $n_\ell \times n_\ell$, and then computes a local matrix multiplication.



(d) Reduce-scatter to obtain final distribution of $\mathbf{C}$. Processor $(1, 2, 3)$ participates in a reduce-scatter of $\mathbf{C}$ with processors $(1, 2, :)$ of dimension $n_\ell \times n_\ell$ and finally owns $\mathbf{C}_{123}$ of dimension $n_\ell \times n_b$.

Figure 4.3: Steps of 3D Parallel Matrix Multiplication (Alg. 4.2) with computation and data associated with processor $(1, 2, 3)$ highlighted.

Algorithm 4.2. 3D Parallel Matrix Multiplication.

Require: Processors arranged in $\sqrt[3]{P} \times \sqrt[3]{P} \times \sqrt[3]{P}$ grid
Require: $\mathbf{A}, \mathbf{B}$ are $n \times n$ matrices in 2D block distribution across $\sqrt[3]{P} \times (\sqrt[3]{P})^2$ processor grid where $n_\ell = n/\sqrt[3]{P}$ and $n_b = n/(\sqrt[3]{P})^2$ are integers
Require: All matrices partitioned into $n_\ell \times n_\ell$ blocks so that

$$\mathbf{M}_{IJ} = \mathbf{M}((I-1)n_\ell+1 : In_\ell, (J-1)n_\ell+1 : Jn_\ell)$$

Require: Processor (I, J, K) owns $n_\ell \times n_b$ submatrices

$$\mathbf{A}_{IKJ} = \mathbf{A}_{IK}(:, (J-1)n_b+1 : Jn_b)$$

$$\mathbf{B}_{KJI} = \mathbf{B}_{KJ}(:, (I-1)n_b+1 : In_b)$$

$$\mathbf{C}_{IJK} = \mathbf{C}_{IJ}(:, (K-1)n_b+1 : Kn_b)$$

Ensure: $\mathbf{C} = \mathbf{C} + \mathbf{AB}$ is $n \times n$ matrix in 2D block distribution across processors so that processor (I, J, K) owns $\mathbf{C}_{IJK}$

1: **function** $\mathbf{C} = $ 3D-MATMUL$(\mathbf{C}, \mathbf{A}, \mathbf{B})$
2: $(I, J, K) =$ MYPROCID()
3: All-gather $\mathbf{A}_{IKJ}$ across Proc$(I, :, K)$, store in $\mathbf{A}_{IK}$
4: All-gather $\mathbf{B}_{KJI}$ across Proc$(:, J, K)$, store in $\mathbf{B}_{KJ}$
5: $\overline{\mathbf{C}}_{IJ} = \mathbf{A}_{IK} \cdot \mathbf{B}_{KJ}$
6: Reduce-scatter $\overline{\mathbf{C}}_{IJ}$ across Proc$(I, J, :)$, combine result with $\mathbf{C}_{IJK}$
7: **end function**

Chapter 5

Linear Systems

This chapter focuses on solving linear systems of equations $\mathbf{A}x = b$, with $\mathbf{A} \in \mathbb{R}^{n\times n}$ by using direct methods of factorization as LU factorization. We consider in particular the case when $\mathbf{A}$ is a dense matrix. Solving a linear system of equations is one of the most used operations in scientific computing. Direct methods are used in particular when the matrix $\mathbf{A}$ is ill conditioned and iterative methods converge slowly or even fail to converge.

5.1 ▪ Background on LU factorization

Gaussian elimination solves the linear system $\mathbf{A}x = b$ by using linear combinations of the equations that transform progressively the system into a triangular system. This can be expressed as factoring the matrix $\mathbf{A}$ as $\mathbf{A} = \mathbf{LU}$, where $\mathbf{L}$ is unit lower triangular and $\mathbf{U}$ is upper triangular. Then the equivalent linear system $\mathbf{LU}x = b$ is solved by finding the solution of two triangular systems: first $\mathbf{L}y = b$ by using forward substitution and second $\mathbf{U}x = y$ by using backward substitution. If the matrix $\mathbf{A}$ is symmetric positive definite, then the Cholesky factorization $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ can be used.

The LU factorization of a matrix $A \in \mathbb{R}^{n\times n}$ is obtained by progressively transforming A into upper triangular through elementary row operations. This is achieved by adding, for each column k, appropriate multiples of the kth row to subsequent rows $k+1$ to n such that the elements below the kth diagonal are annihilated. Consider first a simple matrix $\mathbf{A} \in \mathbb{R}^{3\times 3}$,

$$\mathbf{A} = \begin{pmatrix} 3 & 2 & 1 \\ 6 & 9 & 6 \\ 9 & 16 & 17 \end{pmatrix}.$$

The elements below the first diagonal are zeroed by adding multiples of the first row to the two subsequent rows. This is equivalent to multiplying $\mathbf{A}$ with a matrix $\mathbf{M_1}$ whose first column is formed by the multipliers,

$$\mathbf{M_1} = \begin{pmatrix} 1 & & \\ -2 & 1 & \\ -3 & & 1 \end{pmatrix}, \quad \mathbf{M_1 A} = \begin{pmatrix} 3 & 2 & 1 \\ & 5 & 4 \\ & 10 & 14 \end{pmatrix}.$$

The multipliers are computed by dividing the elements of the first column of A by the diagonal element a_{11}. Following the same procedure to annihilate a_{32}, the upper

triangular factor $\mathbf{U}$ is obtained as,

$$\mathbf{M_2} = \begin{pmatrix} 1 & & \\ & 1 & \\ & -2 & 1 \end{pmatrix}, \quad \mathbf{U} = \mathbf{M_2 M_1 A} = \begin{pmatrix} 3 & 2 & 1 \\ & 5 & 4 \\ & & 6 \end{pmatrix}.$$

The unit lower triangular factor L is hence,

$$\mathbf{L} = \mathbf{M}_1^{-1}\mathbf{M}_2^{-1} = \begin{pmatrix} 1 & & \\ 2 & 1 & \\ 3 & 2 & 1 \end{pmatrix}.$$

In general, suppose that the first $k-1$ steps of the LU factorization of a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ have been computed. The partially factored matrix $\mathbf{A}^{(k)}$,

$$\mathbf{A}^{(k)} = \mathbf{M}_{k-1} \dots \mathbf{M}_1 \mathbf{A},$$

is upper triangular in the first $k-1$ columns. If $a_{k,k}^{(k)}$ is non zero, the factorization can proceed. The multipliers for this step are computed as $l_{j,k} = a_{j,k}^{(k)}/a_{k,k}^{(k)}$, for $j = 1$ to k. They are used to form $\mathbf{M}_k$, a *Gauss transformation* matrix,

$$\mathbf{M}_k = \begin{bmatrix} \mathbf{I}_{k-1} & & & & \\ & 1 & & & \\ & -l_{k+1,k} & 1 & & \\ & \dots & & \ddots & \\ & -l_{n,k} & & & 1 \end{bmatrix} = \mathbf{I} - \mathbf{m}_k \mathbf{e}_k^T,$$

where $\mathbf{I}_{k-1} \in \mathbb{R}^{(k-1)\times(k-1)}$ is the identity matrix, $\mathbf{m}_k = (0, \dots, 0, 1, m_{k+1,k}, \dots, m_{n,k})^T$, and $\mathbf{e}_k$ is the k-th unit vector. It is easy to check that $e_i^T m_k = 0$, for all $i \leq k$. A new matrix $\mathbf{A}^{(k+1)}$, which is upper triangular in the first k columns, is obtained by applying the Gauss transformation to $\mathbf{A}^{(k)}$,

$$\mathbf{A}^{(k+1)} \quad = \quad \mathbf{M}_k \mathbf{A}^{(k)}.$$

Gauss transforms are easy to operate with, their inverse is $\mathbf{M}_k^{-1} = \mathbf{I} + \mathbf{m}_k \mathbf{e}_k^T$, while their multiplication is simplely computed as $\mathbf{M}_1^{-1} \dots \mathbf{M}_k^{-1} = (\mathbf{I} + \mathbf{m}_1 \mathbf{e}_1^T) \dots (\mathbf{I} + \mathbf{m}_k \mathbf{e}_k^T) = \mathbf{I} + \sum_{i=1}^{k} \mathbf{m}_i \mathbf{e}_i^T$, with the result being a unit lower triangular matrix.

If $n-1$ such steps can be computed, the upper triangular U factor is obtained as

$$\mathbf{U} = \mathbf{A}^{(n)} = \mathbf{M}_{n-1} \dots \mathbf{M}_1 \mathbf{A},$$

and hence

$$\mathbf{A} = \mathbf{LU}, \tag{5.1}$$

where

$$\mathbf{L} = \mathbf{M}_1^{-1} \dots \mathbf{M}_{n-1}^{-1} = \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix}. \tag{5.2}$$

However the factorization might not exist if at some iteration k the diagonal element $a_{kk}^{(k)}$ is zero. Even division by small diagonal elements needs to be avoided for numerical

stability. This can be achieved by permuting the rows or the columns of A. The most used technique is partial pivoting, which consists in permuting at each step k of the factorization the element of maximum magnitude in absolute value of $\mathbf{A}^{(k)}(k{+}1:n,k)$ to the diagonal position. Let a_{kj} be that element. Let $\mathbf{\Pi}_k$ be the permutation matrix that corresponds to inderchanging rows k and j of the identity matrix,

$$\mathbf{\Pi}_k = \begin{bmatrix} \mathbf{I}_{k-1} & & & & \\ & & & 1 & \\ & & \mathbf{I}_{j-k-1} & & \\ & 1 & & & \\ & & & & \mathbf{I}_{n-j} \end{bmatrix},$$

where $\mathbf{I}_l \in \mathbb{R}^{(l)\times(l)}$, with $l \in \{k-1, j-k-1, n-j\}$ are identity matrices. The rows k and j of $A^{(k)}$ are interchanged, then the multipliers are computed, and a matrix upper triangular in the first k columns is obtained as

$$\mathbf{A}^{(k+1)} \quad = \quad \mathbf{M}_k \mathbf{\Pi}_k \mathbf{A}^{(k)}.$$

Upon completion, we obtain

$$\mathbf{U} = \mathbf{A}^{(n)} = \mathbf{M}_{n-1}\mathbf{\Pi}_{n-1}\ldots\mathbf{M}_1\mathbf{\Pi}_1\mathbf{A}.$$

The LU factorization with partial pivoting is written as

$$\mathbf{\Pi A} = \mathbf{LU}, \tag{5.3}$$

where $\mathbf{\Pi} = \mathbf{\Pi}_{n-1}\ldots\mathbf{\Pi}_1$ and if $\mathbf{M}_k = \mathbf{I} - \mathbf{m}_k\mathbf{e}_k^T$, then it can be easily shown that $\mathbf{L}(k{+}1:n) = (\mathbf{\Pi}_{n-1}\ldots\mathbf{\Pi}_{k+1}\mathbf{m}_k)(k{+}1:n)$. Partial pivoting allows to bound the multipliers $l_{ik} \leq 1$ and hence $|\mathbf{L}| \leq 1$. If at step k, all elements of $\mathbf{A}^{(k)}(k:n,k)$ are zero, then the first k columns of A are linearly dependent in exact arithmetic. However the factorization can proceed by omitting step k. This factorization is also referred to as Gaussian elimination with partial pivoting, or GEPP.

5.1.1 ▪ Blocked LU factorization with row pivoting

The blocked LU factorization with row pivoting computes the factorization of a matrix $A \in \mathbb{R}^{n\times n}$ by traversing it by blocks of b columns, where for simplicity we assume b divides n. Consider the matrix A is partitioned as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \tag{5.4}$$

where $\mathbf{A}_{11} \in \mathbb{R}^{b\times b}$, $\mathbf{A}_{21} \in \mathbb{R}^{(n-b)\times b}$, $\mathbf{A}_{12} \in \mathbb{R}^{b\times(n-b)}$ and $\mathbf{A}_{22} \in \mathbb{R}^{(n-b)\times(n-b)}$. The first iteration computes the factorization

$$\mathbf{\Pi}_1\mathbf{A} = \begin{bmatrix} \bar{\mathbf{A}}_{11} & \bar{\mathbf{A}}_{12} \\ \bar{\mathbf{A}}_{21} & \bar{\mathbf{A}}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & \\ \mathbf{L}_{21} & \mathbf{I}_{n-b} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{I}_b & \\ & \mathbf{A}_{22}^{(1)} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ & \mathbf{I}_{n-b} \end{bmatrix}. \tag{5.5}$$

This is obtained as follows. First, the LU factorization with row pivoting of the first block column is performed,

$$\mathbf{\Pi}_1 \begin{bmatrix} \mathbf{A}_{11} \\ \mathbf{A}_{21} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} \\ \mathbf{L}_{21} \end{bmatrix} \mathbf{U}_{11}. \tag{5.6}$$

This step is also referred to as panel factorization. Letting $\bar{\mathbf{A}} = \mathbf{\Pi}_1^T \mathbf{A}$, the block $\mathbf{U}_{12}$ is computed by solving the triangular system

$$\mathbf{L}_{11}\mathbf{U}_{12} = \bar{\mathbf{A}}_{12}, \tag{5.7}$$

and finally the trailing matrix is updated,

$$\mathbf{A}_{22}^{(1)} = \bar{\mathbf{A}}_{22} - \mathbf{L}_{21}\mathbf{U}_{12}. \tag{5.8}$$

The algorithm continues recursively on the trailing matrix $\mathbf{A}_{22}^{(1)}$.

Except for the first step, which requires row pivoting for stability and has not yet been specified, it can be seen that the subsequent steps rely on matrix-matrix operations and thus can be implemented efficiently on modern architectures.

5.1.1.1 ▪ Partial pivoting

Blocked LU factorization with partial pivoting computes the LU factorization with partial pivoting of each panel, following the algebra described in section 5.1, which processes the panel column by column. For each column, the algorithm identifies the element with the largest absolute magnitude. The row containing this element is permuted to the diagonal position and then the remaining columns in the panel are updated. This process is repeated for each column until the factorization is complete. The panel factorization relies on vector and matrix-vector operations, which can require more data transfers than the other steps in the blocked LU factorization. The communication cost is discussed in detail in Section 5.4.1 for the parallel algorithm.

5.1.1.2 ▪ Tournament pivoting

Blocked LU factorization with tournament pivoting computes the LU factorization of each panel in two steps. The first step is a preprocessing phase that identifies at low communication cost a set of b pivot rows. These rows are then used as pivots in the second step for the LU factorization of the entire panel. That is, the identified pivot rows are permuted into the leading b positions of the panel, maintaining the order determined in the first step, and the LU factorization of the panel is performed without additional pivoting. Since the panel is a matrix with many more rows than columns, being thus tall and skinny, the panel factorization is referred to as TSLU.

The preprocessing step considers that the panel is partitioned into several blocks and the selection of the pivot rows is performed as a reduction operation, using LU with partial pivoting as the operator to select new pivot rows at each node of the reduction tree. The blocks are selected such that they either fit into fast memory in the sequential case or they match the number of processors in the parallel case. To illustrate tournament pivoting, consider a binomial reduction tree and the factorization of the first panel of dimension $m \times b$ that is partitioned into four blocks, assuming that 4 evenly divides m and that $m \geq 4b$. The preprocessing starts by performing the LU factorization with partial pivoting of each block $\mathbf{A}_I \in \mathbb{R}^{m/4 \times b}$ of the first panel,

$$\begin{aligned}
\mathbf{\Pi}_1^{(2)}\mathbf{A}_1 &= \mathbf{L}_1^{(2)}\mathbf{U}_1^{(2)}, \\
\mathbf{\Pi}_2^{(2)}\mathbf{A}_2 &= \mathbf{L}_2^{(2)}\mathbf{U}_2^{(2)}, \\
\mathbf{\Pi}_3^{(2)}\mathbf{A}_3 &= \mathbf{L}_3^{(2)}\mathbf{U}_3^{(2)}, \\
\mathbf{\Pi}_4^{(2)}\mathbf{A}_4 &= \mathbf{L}_4^{(2)}\mathbf{U}_4^{(2)}.
\end{aligned}$$

This corresponds to the reductions performed at the leaves of the binomial tree, with the superscript (2) indicating the level in the reduction tree. For each factorization $\mathbf{\Pi}_I^{(2)}\mathbf{A}_I = \mathbf{L}_I^{(2)}\mathbf{U}_I^{(2)}$, $\mathbf{\Pi}_I^{(2)} \in \mathbb{R}^{m/4\times m/4}$ is a permutation matrix, $\mathbf{L}_I^{(2)} \in \mathbb{R}^{m/4\times m/4}$ is a lower unit trapezoidal matrix, and $\mathbf{U}_I^{(2)} \in \mathbb{R}^{m/4\times b}$ is an upper triangular matrix. In the second level of the binomial tree, two sets of pivot rows are identified in parallel from the four sets identified at the leaves of the binomial tree. For this, the first two sets are stacked atop one another to form the matrix $\mathbf{A}_1^{(1)} \in \mathbb{R}^{2b\times b}$ and the remaining two sets are stacked to form the matrix $\mathbf{A}_3^{(1)} \in \mathbb{R}^{2b\times b}$. Two LU factorizations with partial pivoting are computed in parallel to obtain two new sets of pivot rows,

$$\mathbf{A}_1^{(1)} := \begin{bmatrix} \mathbf{\Pi}_1^{(2)}\mathbf{A}_1(1:b,:) \\ \mathbf{\Pi}_2^{(2)}\mathbf{A}_2(1:b,:) \end{bmatrix}, \quad \mathbf{\Pi}_1^{(1)}\mathbf{A}_1^{(1)} = \mathbf{L}_1^{(1)}\mathbf{U}_1^{(1)},$$
$$\mathbf{A}_3^{(1)} := \begin{bmatrix} \mathbf{\Pi}_3^{(2)}\mathbf{A}_3(1:b,:) \\ \mathbf{\Pi}_4^{(2)}\mathbf{A}_4(1:b,:) \end{bmatrix}, \quad \mathbf{\Pi}_3^{(1)}\mathbf{A}_3^{(1)} = \mathbf{L}_3^{(1)}\mathbf{U}_3^{(1)}$$

At the root of the depth-2 binomial tree tree, the two new sets of pivot rows are merged into the matrix $\mathbf{A}_1^{(0)}$ and the global pivot rows are obtained by applying LU with partial pivoting on this matrix,

$$\mathbf{A}_1^{(0)} := \begin{bmatrix} \mathbf{\Pi}_1^{(1)}\mathbf{A}_1^{(1)}(1:b,:) \\ \mathbf{\Pi}_3^{(1)}\mathbf{A}_3^{(1)}(1:b,:) \end{bmatrix}, \quad \mathbf{\Pi}_1^{(0)}\mathbf{A}_1^{(0)} = \mathbf{L}_1^{(0)}\mathbf{U}_1^{(0)}$$

The global pivot rows $\mathbf{\Pi}_1^{(0)}\mathbf{A}_1^{(0)}(1:b,:)$ are permuted to the leading positions of the first panel $(\mathbf{\Pi}_1\mathbf{A})(1:b,1:b)$. Let $\mathbf{\Pi}_1 \in \mathbb{R}^{m\times m}$ be the matrix that reflects this permutation. Then the LU factorization of the first panel is computed with no more permutations to obtain the factorization of the first panel as in (5.6),

$$\mathbf{\Pi}_1 \begin{bmatrix} \mathbf{A}_{11} \\ \mathbf{A}_{21} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} \\ \mathbf{L}_{21} \end{bmatrix} \mathbf{U}_{11}. \tag{5.9}$$

Note that $\mathbf{U}_{11} = \mathbf{U}_1^{(0)}$. The blocked LU factorization continues with computing the first block row of U and the update of the trailing matrix.

Different reduction trees can be used during the preprocessing step of TSLU. We illustrate them using an arrow notation with the following meaning. The function $f(B)$ computes GEPP of matrix B, and returns the b rows used as pivots. The input matrix B is formed by stacking atop one another the matrices situated at the left side of the arrows pointing to $f(B)$. A binary tree of height two is represented in the following picture:

$$\begin{array}{lll} \mathbf{A}_1 \to \mathbf{\Pi}_1^{(2)}\mathbf{A}_1 & \searrow & \\ & & \mathbf{\Pi}_1^{(1)}\mathbf{A}_1^{(1)} \searrow \\ \mathbf{A}_2 \to \mathbf{\Pi}_2^{(2)}\mathbf{A}_2 & \nearrow & \qquad\qquad \mathbf{\Pi}_1^{(0)}\mathbf{A}_1^{(0)} \\ \mathbf{A}_3 \to \mathbf{\Pi}_3^{(2)}\mathbf{A}_3 & \searrow & \\ & & \mathbf{\Pi}_3^{(1)}\mathbf{A}_3^{(1)} \nearrow \\ \mathbf{A}_4 \to \mathbf{\Pi}_4^{(2)}\mathbf{A}_4 & \nearrow & \end{array}$$

A reduction tree of height one leads to the following factorization:

$$\begin{array}{ll} \mathbf{A}_1 \to \mathbf{\Pi}_1^{(1)}\mathbf{A}_1 & \searrow \\ \mathbf{A}_2 \to \mathbf{\Pi}_2^{(1)}\mathbf{A}_2 & \to \mathbf{\Pi}_1^{(0)}\mathbf{A}_1^{(0)} \\ \mathbf{A}_3 \to \mathbf{\Pi}_3^{(1)}\mathbf{A}_3 & \nearrow \\ \mathbf{A}_4 \to \mathbf{\Pi}_4^{(1)}\mathbf{A}_4 & \end{array}$$

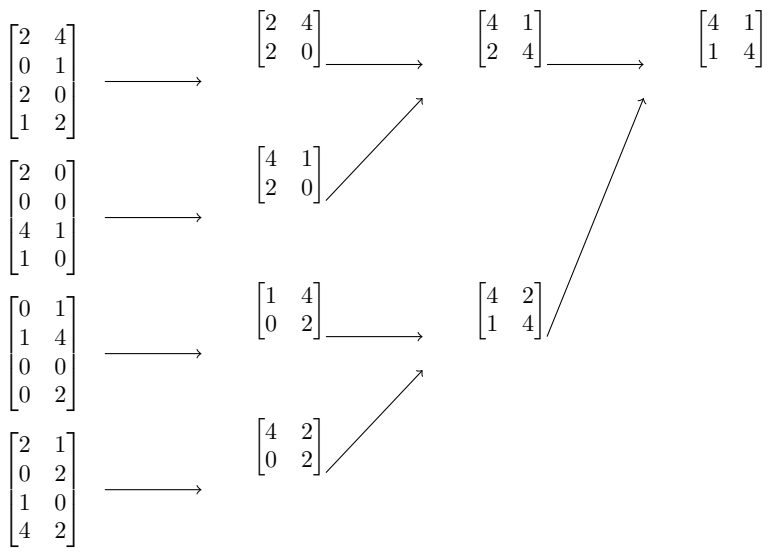
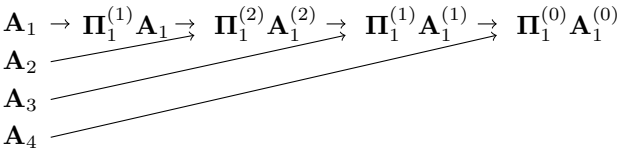


Figure 5.1: Visualization on a simple matrix of the binomial elimination tree used by parallel TSLU (Fig. 5.2)

The flat-tree-based TSLU is illustrated using the arrow abbreviation as:



5.2 ▪ Numerical stability

The numerical stability of the LU factorization is well studied in the literature. In fact it is the first algorithm for which a rounding error analysis was studied. We recall here only the main backward error stability result. The stability of the LU factorization depends on the growth factor g_W defined as

$$g_W(n) = \frac{\max_{i,j,k} |a_{ij}^k|}{\max_{i,j} |a_{ij}|}. \tag{5.10}$$

Note that while the elements of L are bounded by 1 when partial pivoting is used, the elements of U can be bounded as

$$|u_{ij}| = |a_{ij}^i| \leq g_W \max_{i,j} |a_{ij}|.$$

Theorem 5.2.1 (Wilkinson's backward error stability result) *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ and let $\hat{x}$ be the computed solution of $Ax = b$ obtained by using GEPP. Then*

$$(\mathbf{A} + \Delta\mathbf{A})\hat{x} = b, \qquad \|\Delta\mathbf{A}\|_\infty \leq n^2 \gamma_{3n} g_W(n) \|A\|_\infty,$$

where $\gamma_n = n\epsilon/(1 - n\epsilon)$, ϵ is machine precision and assuming $n\epsilon < 1$.

The LU factorization is backward stable if the growth factor is small. For partial pivoting, the growth factor can be as large as 2^{n-1}. As noted by Wilkinson, this bound is attained on the following matrix:

$$\mathbf{A} = diag(\pm 1) \begin{bmatrix} 1 & 0 & 0 & \cdots & 1 \\ -1 & 1 & 0 & ... & 1 \\ -1 & -1 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 1 \\ -1 & -1 & \cdots & -1 & 1 \end{bmatrix} \tag{5.11}$$

Since all the elements below the diagonal are 1 in absolute value, no row permutation occurs during the LU factorization with partial pivoting. Exponential growth appears in the last column as only additions are performed at each step of the factorization. While there are a few other classes of matrices for which the growth factor is large, in practice for most of the applications the growth factor is small, on the order of $n^{2/3}$. Two reasons contribute to this. First, the multipliers in L are on average much smaller than 1. Second, the signs of the multipliers in L and of the elements of U are not independent, this being related to the fact that a rank-1 update of the trailing matrix is performed at each step of the factorization. As a result, cancellations are favored and growth is delayed.

A comparison of the upper bound of the growth factors obtained by different pivoting strategies is given in Table 5.1. All the results discussed in this section hold in exact arithmetic. The growth factor of CALU is obtained by using the fact that performing CALU on a matrix A is equivalent with performing GEPP on a larger matrix formed by blocks from the original matrix A and blocks of zeros. We observe that the upper bound of the growth factor is larger for CALU than for GEPP. However many experiments

Table 5.1: Bounds for the growth factor g_W obtained from tournament pivoting in CALU and partial pivoting in GEPP for a matrix of size $n \times n$. The reduction tree used during tournament pivoting is of height $\log P$.

	CALU	GEPP
Upper bound	$2^{n(\log P+1)-1}$	2^{n-1}

5.3 ▪ Lower bounds on communication

It can be shown by reduction that the lower bounds on communication identified in section 4 apply as well to LU factorization and other direct methods of factorization as QR or Cholesky factorizations. Consider the LU factorization of the following matrix:

$$\begin{pmatrix} I & & -\mathbf{B} \\ \mathbf{A} & I & \\ & & \mathbf{I} \end{pmatrix} = \begin{pmatrix} \mathbf{I} & & \\ \mathbf{A} & \mathbf{I} & \\ & & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{I} & & -\mathbf{B} \\ & \mathbf{I} & \mathbf{AB} \\ & & \mathbf{I} \end{pmatrix}.$$

Since this factorization involves multiplying matrices $\mathbf{A}$ and $\mathbf{B}$, by reduction, the lower bounds from matrix multiplication apply to LU factorization as well. We consider here the case where the memory size available on each processor is $M = \Omega(n^2/P)$, and thus the lower bounds are:

$$\#\text{words} \geq \Omega\left(\frac{n^2}{\sqrt{P}}\right), \quad \#\text{messages} \geq \Omega(\sqrt{P}) \tag{5.12}$$

5.4 ▪ Parallel algorithms

5.4.1 ▪ Parallel LU with partial pivoting

Parallel blocked LU distributes the matrix A over a $P_r \times P_c$ grid of processors using a bidimensional (2D) block cyclic layout with blocks of dimension $b \times b$. As an example, with a 2×2 grid of processors, the blocks of the matrix are distributed over processors as

$$\begin{bmatrix} (1,1) & (1,2) & (1,1) & (1,2) & \dots \\ (2,1) & (2,2) & (2,1) & (2,2) & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}.$$

Algorithm 5.1 presents the main operations executed at each iteration of the blocked LU factorization. First the panel factorization using partial pivoting is computed by processors in the same column of the process grid that own the current panel. For each column i in the block column, the pivot is found and the the i-th row is swapped with the pivot row of the current block column. Once the i-th column of L is computed, the i-row is broadcasted in the current column of processes and the trailing block column is updated. By ignoring some lower order terms, the cost of the panel factorization is

$$2n \log_2 P_r \alpha + nb\beta + \frac{1}{P_r}\left(mnb - \frac{n^2 b}{2}\right)\gamma$$

The processors owning block column K broadcast the pivot information along the rows of the process grid. The pivot information is applied to the remainder of the rows to obtain $\mathbf{A} = \mathbf{\Pi_K A}$. Considering that all to all is used for the communication, the cost becomes

$$\frac{n}{b}(\log_2 P_r + \log_2 P_c)\alpha + \frac{n^2 - nb}{P_c}\log_2 P_r \beta.$$

The $b \times b$ upper part of the block column j of L is broadcasted along row of processes owning block row j of U. Then the block row j of U is computed, with the cost being

$$\frac{n}{b}\log_2 P_c \alpha_r + \frac{nb}{2}\log_2 P_c \beta_r + \frac{n^2 b}{2P_c}\gamma$$

The block column j of L is broadcasted along rows of processors of the grid and the block row j of U is broadcasted along columns of processors of the grid. A rank-b update on the trailing matrix is performed. The cost is

$$\frac{n}{b}(\log_2 P_c + \log_2 P_r)\alpha + (\frac{1}{P_r}\left(mn - \frac{n^2}{2}\right)\log_2 P_c + \frac{n^2}{2P_c}\log_2 P_r)\beta + \frac{1}{P}\left(mn^2 - \frac{n^3}{3}\right)\gamma$$

By summing up all the costs and ignoring further lower order terms, the overall cost of this algorithm is:

$$(2n\left(1 + \frac{2}{b}\right)\log_2 P_r + \frac{3n}{b}\log_2 P_c)\alpha + \tag{5.13}$$

$$\left(\frac{nb}{2} + \frac{3n^2}{2P_c}\right)\log_2 P_r + \log_2 P_c \frac{1}{P_r}\left(mn - \frac{n^2}{2}\right)\beta + \tag{5.14}$$

$$(\frac{1}{P}\left(mn^2 - \frac{n^3}{3}\right) + \frac{1}{P_r}\left(mn - \frac{n^2}{2}\right)b + \frac{n^2 b}{2P_c})\gamma \tag{5.15}$$

In terms of number of messages, it can be seen that, except for the panel factorization, all the other operations rely on collective communications which require exchanging $O(\log P_r)$ or $O(\log P_c)$ messages. Hence, the latency bottleneck lies in the panel factorization, where the LU factorization is performed column by column.

Algorithm 5.1. Parallel LU with partial pivoting.

Require: $\mathbf{A} \in \mathbb{R}^{n \times n}$ in 2D block cyclic distribution across processors
Require: Processors arranged in $P_r \times P_c$ grid where m/P_r and n/P_c are integers
Require: Processor (I, J) owns $b \times b$ submatrices $\mathbf{M}_{KS}$ for all K, S...

$$\mathbf{M}_{IJ} = \mathbf{M}((I-1)n_\ell+1 : In_\ell, (J-1)n_\ell+1 : Jn_\ell)$$

Require: Block size b divides m/P_r and n/P_c evenly
function $[\mathbf{L}, \mathbf{U}, \mathbf{\Pi}] =$ 2D-LU($\mathbf{A}$)
  $\mathbf{L} \in \mathbb{R}^{m \times n}$ is initialized with identity matrix and $\mathbf{U} \in \mathbb{R}^{n \times n}$ with zero matrix
  for $K = 1$ to n/b **do**
    Let Proc(I, J) be the owner of $\mathbf{A}_{KK}$
    Procs$(:, J)$ compute pivoted LU factorization
$$\Pi_k \mathbf{A}_{K:,K} = \mathbf{L}_{K:,K} \cdot \mathbf{U}_{K,K}$$
    for $S = 1$ to P_r in parallel **do**
      Proc(S, J) broadcasts pivot information to Proc$(S, :)$
    end for
    Alltoall among all processors to permute b rows and obtain
$$\mathbf{A} = \Pi_k \mathbf{A}$$
    Proc(I, J) broadcasts $\mathbf{L}_{K,K}$ to Procs$(:, J)$
    Procs$(I, :)$ compute their blocks of $\mathbf{U}_{K,K+1:}$ as
$$\mathbf{U}_{K,K+1:} = \mathbf{L}_{K,K}^{-1} \mathbf{A}_{K,K+1:}$$
    for $S = 1$ to P_r in parallel **do**
      Proc(S, J) broadcasts its blocks of $\mathbf{L}_{K+1:,K}$ to Proc$(S, :)$
    end for
    for $S = 1$ to P_c in parallel **do**
      Proc(I, S) broadcasts its blocks of $\mathbf{U}_{K,K+1:}$ to Proc$(:, S)$
    end for
    All processors update their blocks of the trailing matrix,
$$\mathbf{A}_{K+1:,K+1:} = \mathbf{A}_{K+1:,K+1:} - \mathbf{L}_{K+1:,K} \cdot \mathbf{U}_{K,K+1}$$
  end for
end function

5.4.2 ▪ Parallel communication avoiding LU with tournament pivoting

A more efficient algorithm is obtained if tournament pivoting is used instead of partial pivoting for each panel factorization. This algorithm is presented in Figure 5.2 and the factorization is referred to as TSLU.

Algorithm 5.2. Parallel TSLU (binomial tree).

Require: $\mathbf{A}$ is an $m \times n$ matrix 1D-row-distributed over power-of-two P processors
Ensure: $\mathbf{\Pi A} = \mathbf{LU}$, where $\mathbf{U}$ is upper triangular and $\mathbf{L}$ is unit lower triangular.

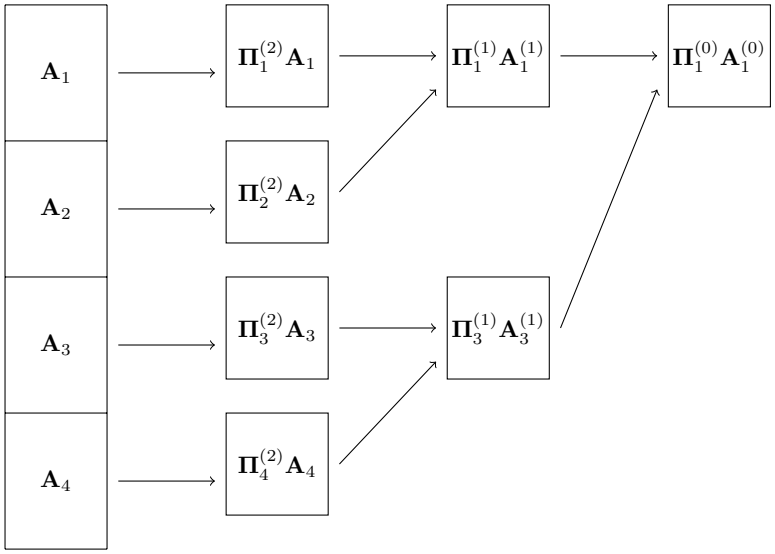


Figure 5.2: Visualization of the binomial reduction tree used by parallel TSLU (**??**)

Ensure: $\mathbf{U}$ is stored on processor 1 and each $\mathbf{L}^{(k)}$ is distributed across 2^k processors
function $\left[\Pi, \left\{\mathbf{L}_I^{(k)}\right\}, \mathbf{R}\right]$ = ParTSLU($\mathbf{A}$)
  I = MyProcID()
  Compute $\mathbf{\Pi}_I^{(\log P)}\mathbf{A}_I^{(\log P)} = \mathbf{L}_I^{(\log P)}\mathbf{U}_I^{(\log P)}$ using GEPP, where $\mathbf{A}_I^{(\log P)} := \mathbf{A}_I$
  for $k = \log P - 1$ down to 0 **do**
    Break if I doesn't have a partner proc
    Determine J, partner proc ID
    if $I > J$ **then**
      Send $\mathbf{\Pi}_I^{(k+1)}\mathbf{A}_I^{(k+1)}(1:b,:)$ to processor J
    else
      Receive $\mathbf{\Pi}_J^{(k+1)}\mathbf{A}_J^{(k+1)}(1:b,:)$ from processor J
      Form the matrix $\mathbf{A}_I^{(k)} := \begin{bmatrix}\mathbf{\Pi}_I^{(k+1)}\mathbf{A}_I^{(k+1)}(1:b,:) \\ \mathbf{\Pi}_J^{(k+1)}\mathbf{A}_J^{(k+1)}(1:b,:)\end{bmatrix}$
      Compute $\mathbf{\Pi}_I^{(k)}\mathbf{A}_I^{(k)} = \mathbf{L}_I^{(k)}\mathbf{U}_I^{(k)}$ using GEPP
    end if
  end for
  Permute the pivot rows $\mathbf{\Pi}_1^{(0)}\mathbf{A}_1^{(0)}(1:b,:)$ to the leading positions of $\mathbf{A}$
  Let $\mathbf{\Pi}$ be the matrix that reflects this permutation
  if $I = 1$ **then**
    $\mathbf{U} = \mathbf{U}_1^{(0)}$
    Broadcast $\mathbf{U}$ to all processors
  end if
  Compute $\mathbf{L}_I = \mathbf{A}_I\mathbf{U}^{-1}$
end function

The LU factorization of a square matrix that relies on TSLU for its panel factorization is referred to as CALU. TSLU requires exchanging $\log P$ messages among processors.

This allows the overall CALU algorithm to attain the lower bounds on communication in terms of both number of messages and volume of communication. When the LU factorization of a matrix of size $n \times n$ is computed by using CALU on a grid of $P = P_r \times P_c$ processors, the parallel performance of CALU in terms of number of messages, volume of communication, and flops, is

$$\begin{aligned} T_{CALU}(m,n,P) &\approx \gamma \cdot \left(\frac{1}{P}\left(mn^2 - \frac{n^3}{3}\right) + \frac{1}{P_r}\left(2mn - n^2\right) b + \frac{n^2 b}{2P_c} + \frac{nb^2}{3}(5\log_2 P_r - 1) \right) \\ &+ \beta \cdot \left(\left(nb + \frac{3n^2}{2P_c} \right) \log_2 P_r + \frac{1}{P_r}\left(mn - \frac{n^2}{2} \right) \log_2 P_c \right) \\ &+ \alpha \cdot \left(\frac{3n}{b} \log_2 P_r + \frac{3n}{b} \log_2 P_c \right). \end{aligned} \tag{5.16}$$

To attain the lower bounds on communication, an optimal layout can be chosen with $P_r = P_c = \sqrt{P}$ and $b = \log^{-2}\left(\sqrt{P}\right) \cdot \frac{n}{\sqrt{P}}$. The blocking parameter b is chosen such that the number of messages attains the lower bound on communication from equation (**??**), while the number of flops increases only by a lower order term. With this layout, the performance of CALU becomes,

$$\begin{aligned} T_{CALU}(m,n,P = \sqrt{P} \times \sqrt{P}) &\approx \gamma \cdot \left(\frac{1}{P}\frac{2n^3}{3} + \frac{5n^3}{2P\log^2 P} + \frac{5n^3}{3P\log^3 P} \right) \\ &+ \beta \cdot \frac{n^2}{\sqrt{P}} \left(2\log^{-1} P + 1.25 \log P \right) \\ &+ \alpha \cdot 3\sqrt{P}\log^3 P. \end{aligned} \tag{5.17}$$

We note that GEPP as implemented for example in ScaLAPACK (`PDGETRF` routine) has the same volume of communication as CALU, but requires exchanging a factor on the order of b more messages than CALU.

Bibliography

[1] Åke Björck. Solving linear least squares problems by Gram-Schmidt orthogonalization. *BIT*, 7:1–21, 1967.

[2] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[3] James W. Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comput.*, 34(1):A206–A239, 2012.

[4] Luc Giraud, Julien Langou, Miroslav Rozložník, and Jasper Van Den Eshof. Rounding error analysis of the classical Gram-Schmidt orthogonalization process. *Numer. Math.*, 101:87–100, 2005.

[5] L. H. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the AMS*, 55:961–962, 1949.

[6] Daisuke Mori, Yusaku Yamamoto, and Shao Liang Zhang. Backward error analysis of the AllReduce algorithm for Householder QR decomposition. *Jpn. J. Ind. Appl. Math.*, 29(1):111–130, 2012.

[7] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[8] James Hardy Wilkinson. *The algebraic eigenvalue problem*, volume 87. Oxford University Press, 1965.

[9] James Hardy Wilkinson. Error analysis of transformations based on the use of matrices of the form $i - 2ww^h$. *Error in Digital Computation*, 2:77–101, 1965.

[10] Yusaku Yamamoto, Yuji Nakatsukasa, Yuka Yanagisawa, and Takeshi Fukaya. Roundoff error analysis of the CholeskyQR2 algorithm. *Electron. Trans. Numer. Anal.*, 44:306–326, 2015.