Chapter 1

Matrix Multiplication

The study of matrix computations properly begins with the study of various matrix multiplication problems. Although simple mathematically, these calculations are sufficiently rich to develop a wide range of essential algorithmic skills.

In §1.1 we examine several formulations of the matrix multiplication update problem $C = C + AB$. Partitioned matrices are introduced and used to identify linear algebraic "levels" of computation.

If a matrix has special properties, then various economies are generally possible. For example, a symmetric matrix can be stored in half the space of a general matrix. A matrix-vector product may require much less time to execute if the matrix has many zero entries. These matters are considered in §1.2.

A block matrix is a matrix whose entries are themselves matrices. The "language" of block matrices is developed in §1.3. It supports the easy derivation of matrix factorizations by enabling us to spot patterns in a computation that are obscured at the scalar level. Algorithms phrased at the block level are typically rich in matrix-matrix multiplication, the operation of choice in many high-performance computing environments. Sometimes the block structure of a matrix is recursive, meaning that the block entries have an exploitable resemblance to the overall matrix. This type of connection is the foundation for "fast" matrix-vector product algorithms such as various fast Fourier transforms, trigonometric transforms, and wavelet transforms. These calculations are among the most important in all of scientific computing and are discussed in §1.4. They provide an excellent opportunity to develop a facility with block matrices and recursion.

The last two sections set the stage for effective, "large-n" matrix computations. In this context, data locality affects efficiency more than the volume of actual arithmetic. Having an ability to reason about memory hierarchies and multiprocessor computation is essential. Our goal in §1.5 and §1.6 is to build an appreciation for the attendant issues without getting into system-dependent details.

Reading Notes

The sections within this chapter depend upon each other as follows:

$$\begin{array}{ccccccc} \S1.1 & \rightarrow & \S1.2 & \rightarrow & \S1.3 & \rightarrow & \S1.4 \\ & & & & \downarrow & & \\ & & & & \S1.5 & \rightarrow & \S1.6 \end{array}$$

Before proceeding to later chapters, §1.1, §1.2, and §1.3 are essential. The fast transform ideas in §1.4 are utilized in §4.8 and parts of Chapters 11 and 12. The reading of §1.5 and §1.6 can be deferred until high-performance linear equation solving or eigenvalue computation becomes a topic of concern.

1.1 Basic Algorithms and Notation

Matrix computations are built upon a hierarchy of linear algebraic operations. Dot products involve the scalar operations of addition and multiplication. Matrix-vector multiplication is made up of dot products. Matrix-matrix multiplication amounts to a collection of matrix-vector products. All of these operations can be described in algorithmic form or in the language of linear algebra. One of our goals is to show how these two styles of expression complement each other. Along the way we pick up notation and acquaint the reader with the kind of thinking that underpins the matrix computation area. The discussion revolves around the matrix multiplication problem, a computation that can be organized in several ways.

1.1.1 Matrix Notation

Let $\mathbb{R}$ designate the set of real numbers. We denote the vector space of all m-by-n real matrices by $\mathbb{R}^{m\times n}$:

$$A \in \mathbb{R}^{m\times n} \iff A = (a_{ij}) = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}, \quad a_{ij} \in \mathbb{R}$$

If a capital letter is used to denote a matrix (e.g., A, B, Δ), then the corresponding lower case letter with subscript ij refers to the (i,j) entry (e.g., a_{ij}, b_{ij}, δ_{ij}). Sometimes we designate the elements of a matrix with the notation $[\,A\,]_{ij}$ or $A(i,j)$.

1.1.2 Matrix Operations

Basic matrix operations include *transposition* ($\mathbb{R}^{m\times n} \rightarrow \mathbb{R}^{n\times m}$),

$$C = A^T \qquad \Longrightarrow \qquad c_{ij} = a_{ji},$$

addition ($\mathbb{R}^{m\times n} \times \mathbb{R}^{m\times n} \rightarrow \mathbb{R}^{m\times n}$),

$$C = A + B \quad \Longrightarrow \quad c_{ij} = a_{ij} + b_{ij},$$

scalar-matrix multiplication ($\mathbb{R} \times \mathbb{R}^{m\times n} \rightarrow \mathbb{R}^{m\times n}$),

$$C = \alpha A \quad \Longrightarrow \quad c_{ij} = \alpha a_{ij},$$

and *matrix-matrix multiplication* ($\mathbb{R}^{m\times p} \times \mathbb{R}^{p\times n} \rightarrow \mathbb{R}^{m\times n}$),

$$C = AB \quad \Longrightarrow \quad c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}.$$

Pointwise matrix operations are occasionally useful, especially pointwise multiplication ($\mathbb{R}^{m\times n} \times \mathbb{R}^{m\times n} \rightarrow \mathbb{R}^{m\times n}$),

$$C = A .* B \quad \Longrightarrow \quad c_{ij} = a_{ij} b_{ij}$$

and pointwise division ($\mathbb{R}^{m\times n} \times \mathbb{R}^{m\times n} \rightarrow \mathbb{R}^{m\times n}$),

$$C = A ./ B \quad \Longrightarrow \quad c_{ij} = a_{ij} / b_{ij}.$$

Of course, for pointwise division to make sense, the "denominator matrix" must have nonzero entries.

1.1.3 Vector Notation

Let $\mathbb{R}^n$ denote the vector space of real n-vectors:

$$x \in \mathbb{R}^n \quad \Longleftrightarrow \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad x_i \in \mathbb{R}\,.$$

We refer to x_i as the ith component of x. Depending upon context, the alternative notations $[x]_i$ and $x(i)$ are sometimes used.

Notice that we are identifying $\mathbb{R}^n$ with $\mathbb{R}^{n\times 1}$ and so the members of $\mathbb{R}^n$ are *column* vectors. On the other hand, the elements of $\mathbb{R}^{1\times n}$ are *row* vectors:

$$x \in \mathbb{R}^{1\times n} \quad \Longleftrightarrow \quad x = [x_1, \ldots, x_n].$$

If x is a column vector, then $y = x^T$ is a row vector.

1.1.4 Vector Operations

Assume that $a \in \mathbb{R}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^n$. Basic vector operations include *scalar-vector multiplication*,

$$z = ax \quad \Longrightarrow \quad z_i = a x_i,$$

vector addition,

$$z = x + y \quad \Longrightarrow \quad z_i = x_i + y_i,$$

and the *inner product* (or *dot product*),

$$c = x^T y \qquad \Longrightarrow \qquad c = \sum_{i=1}^{n} x_i y_i.$$

A particularly important operation, which we write in update form, is the *saxpy*:

$$y = ax + y \quad \Longrightarrow \quad y_i = ax_i + y_i$$

Here, the symbol "=" is used to denote assignment, not mathematical equality. The vector y is being updated. The name "saxpy" is used in LAPACK, a software package that implements many of the algorithms in this book. "Saxpy" is a mnemonic for "scalar a x plus y." See `LAPACK`.

Pointwise vector operations are also useful, including *vector multiplication*,

$$z = x.*y \qquad \Longrightarrow \qquad z_i = x_i y_i,$$

and *vector division*,

$$z = x./y \qquad \Longrightarrow \qquad z_i = x_i / y_i.$$

1.1.5 The Computation of Dot Products and Saxpys

Algorithms in the text are expressed using a stylized version of the MATLAB language. Here is our first example:

Algorithm 1.1.1 (Dot Product) If $x, y \in \mathbb{R}^n$, then this algorithm computes their dot product $c = x^T y$.

```
c = 0
for i = 1:n
    c = c + x(i)y(i)
end
```

It is clear from the summation that the dot product of two n-vectors involves n multiplications and n additions. The dot product operation is an "$O(n)$" operation, meaning that the amount of work scales linearly with the dimension. The saxpy computation is also $O(n)$:

Algorithm 1.1.2 (Saxpy) If $x, y \in \mathbb{R}^n$ and $a \in \mathbb{R}$, then this algorithm overwrites y with $y + ax$.

```
for i = 1:n
    y(i) = y(i) + ax(i)
end
```

We stress that the algorithms in this book are encapsulations of important computational ideas and are not to be regarded as "production codes."

1.1.6 Matrix-Vector Multiplication and the Gaxpy

Suppose $A \in \mathbb{R}^{m\times n}$ and that we wish to compute the update

$$y = y + Ax$$

where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ are given. This generalized saxpy operation is referred to as a *gaxpy*. A standard way that this computation proceeds is to update the components one-at-a-time:

$$y_i = y_i + \sum_{j=1}^{n} a_{ij}x_j, \qquad i = 1{:}m.$$

This gives the following algorithm:

Algorithm 1.1.3 (Row-Oriented Gaxpy) If $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$, then this algorithm overwrites y with $Ax + y$.

```
for i = 1:m
    for j = 1:n
        y(i) = y(i) + A(i,j)x(j)
    end
end
```

Note that this involves $O(mn)$ work. If each dimension of A is doubled, then the amount of arithmetic increases by a factor of 4.

An alternative algorithm results if we regard Ax as a linear combination of A's columns, e.g.,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}\begin{bmatrix} 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 1\cdot 7 + 2\cdot 8 \\ 3\cdot 7 + 4\cdot 8 \\ 5\cdot 7 + 6\cdot 8 \end{bmatrix} = 7\begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} + 8\begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} = \begin{bmatrix} 23 \\ 53 \\ 83 \end{bmatrix}.$$

Algorithm 1.1.4 (Column-Oriented Gaxpy) If $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$, then this algorithm overwrites y with $Ax + y$.

```
for j = 1:n
    for i = 1:m
        y(i) = y(i) + A(i,j)·x(j)
    end
end
```

Note that the inner loop in either gaxpy algorithm carries out a saxpy operation. The column version is derived by rethinking what matrix-vector multiplication "means" at the vector level, but it could also have been obtained simply by interchanging the order of the loops in the row version.

1.1.7 Partitioning a Matrix into Rows and Columns

Algorithms 1.1.3 and 1.1.4 access the data in A by row and by column, respectively. To highlight these orientations more clearly, we introduce the idea of a *partitioned matrix.*

From one point of view, a matrix is a stack of row vectors:

$$A \in \mathbb{R}^{m\times n} \quad \Longleftrightarrow \quad A = \begin{bmatrix} r_1^T \\ \vdots \\ r_m^T \end{bmatrix}, \quad r_k \in \mathbb{R}^n . \tag{1.1.1}$$

This is called a *row partition* of A. Thus, if we row partition

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix},$$

then we are choosing to think of A as a collection of rows with

$$r_1^T = [\, 1 \;\; 2 \,], \qquad r_2^T = [\, 3 \;\; 4 \,], \qquad r_3^T = [\, 5 \;\; 6 \,].$$

With the row partitioning (1.1.1), Algorithm 1.1.3 can be expressed as follows:

for $i = 1{:}m$
  $y_i = y_i + r_i^T x$
end

Alternatively, a matrix is a collection of column vectors:

$$A \in \mathbb{R}^{m\times n} \quad \Longleftrightarrow \quad A = [\, c_1 \,|\, \cdots \,|\, c_n \,], \quad c_k \in \mathbb{R}^m. \tag{1.1.2}$$

We refer to this as a *column partition* of A. In the 3-by-2 example above, we thus would set c_1 and c_2 to be the first and second columns of A, respectively:

$$c_1 = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}, \qquad c_2 = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}.$$

With (1.1.2) we see that Algorithm 1.1.4 is a saxpy procedure that accesses A by columns:

for $j = 1{:}n$
  $y = y + x_j c_j$
end

In this formulation, we appreciate y as a running vector sum that undergoes repeated saxpy updates.

1.1.8 The Colon Notation

A handy way to specify a column or row of a matrix is with the "colon" notation. If $A \in \mathbb{R}^{m\times n}$, then $A(k,:)$ designates the kth row, i.e.,

$$A(k,:) = [\, a_{k1}, \ldots, a_{kn} \,].$$

The kth column is specified by

$$A(:,k) = \begin{bmatrix} a_{1k} \\ \vdots \\ a_{mk} \end{bmatrix}.$$

With these conventions we can rewrite Algorithms 1.1.3 and 1.1.4 as

for $i = 1{:}m$
  $y(i) = y(i) + A(i,:)\cdot x$
end

and

for $j = 1{:}n$
  $y = y + x(j)\cdot A(:,j)$
end

respectively. By using the colon notation, we are able to suppress inner loop details and encourage vector-level thinking.

1.1.9 The Outer Product Update

As a preliminary application of the colon notation, we use it to understand the *outer product update*

$$A = A + xy^T, \qquad A \in \mathbb{R}^{m\times n},\ x \in \mathbb{R}^m,\ y \in \mathbb{R}^n.$$

The outer product operation xy^T "looks funny" but is perfectly legal, e.g.,

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 4 & 5 \end{bmatrix} = \begin{bmatrix} 4 & 5 \\ 8 & 10 \\ 12 & 15 \end{bmatrix}.$$

This is because xy^T is the product of two "skinny" matrices and the number of columns in the left matrix x equals the number of rows in the right matrix y^T. The entries in the outer product update are prescribed by

for $i = 1{:}m$
  for $j = 1{:}n$
    $a_{ij} = a_{ij} + x_i y_j$
  end
end

This involves $O(mn)$ arithmetic operations. The mission of the j loop is to add a multiple of y^T to the ith row of A, i.e.,

for $i = 1{:}m$
  $A(i,:) = A(i,:) + x(i)\cdot y^T$
end

On the other hand, if we make the i-loop the inner loop, then its task is to add a multiple of x to the jth column of A:

for $j = 1{:}n$
  $A(:,j) = A(:,j) + y(j)\cdot x$
end

Note that both implementations amount to a set of saxpy computations.

1.1.10 Matrix-Matrix Multiplication

Consider the 2-by-2 matrix-matrix multiplication problem. In the dot product formulation, each entry is computed as a dot product:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1\cdot 5+2\cdot 7 & 1\cdot 6+2\cdot 8 \\ 3\cdot 5+4\cdot 7 & 3\cdot 6+4\cdot 8 \end{bmatrix}.$$

In the saxpy version, each column in the product is regarded as a linear combination of left-matrix columns:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 5\begin{bmatrix} 1 \\ 3 \end{bmatrix} + 7\begin{bmatrix} 2 \\ 4 \end{bmatrix}, & 6\begin{bmatrix} 1 \\ 3 \end{bmatrix} + 8\begin{bmatrix} 2 \\ 4 \end{bmatrix} \end{bmatrix}.$$

Finally, in the outer product version, the result is regarded as the sum of outer products:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}\begin{bmatrix} 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 \\ 4 \end{bmatrix}\begin{bmatrix} 7 & 8 \end{bmatrix}.$$

Although equivalent mathematically, it turns out that these versions of matrix multiplication can have very different levels of performance because of their memory traffic properties. This matter is pursued in §1.5. For now, it is worth detailing the various approaches to matrix multiplication because it gives us a chance to review notation and to practice thinking at different linear algebraic levels. To fix the discussion, we focus on the matrix-matrix update computation:

$$C = C + AB, \qquad C \in \mathbb{R}^{m\times n},\ A \in \mathbb{R}^{m\times r},\ B \in \mathbb{R}^{r\times n}.$$

The update $C = C + AB$ is considered instead of just $C = AB$ because it is the more typical situation in practice.

1.1.11 Scalar-Level Specifications

The starting point is the familiar triply nested loop algorithm:

Algorithm 1.1.5 (ijk Matrix Multiplication) If $A \in \mathbb{R}^{m\times r}$, $B \in \mathbb{R}^{r\times n}$, and $C \in \mathbb{R}^{m\times n}$ are given, then this algorithm overwrites C with $C + AB$.

for $i = 1{:}m$
  for $j = 1{:}n$
    for $k = 1{:}r$
      $C(i,j) = C(i,j) + A(i,k)\cdot B(k,j)$
    end
  end
end

This computation involves $O(mnr)$ arithmetic. If the dimensions are doubled, then work increases by a factor of 8.

Each loop index in Algorithm 1.1.5 has a particular role. (The subscript i names the row, j names the column, and k handles the dot product.) Nevertheless, the ordering of the loops is arbitrary. Here is the (mathematically equivalent) jki variant:

```
for j = 1:n
    for k = 1:r
        for i = 1:m
            C(i,j) = C(i,j) + A(i,k)B(k,j)
        end
    end
end
```

Altogether, there are six (= 3!) possibilities:

$$ijk, \quad jik, \quad ikj, \quad jki, \quad kij, \quad kji.$$

Each features an inner loop operation (dot product or saxpy) and each has its own pattern of data flow. For example, in the ijk variant, the inner loop oversees a dot product that requires access to a row of A and a column of B. The jki variant involves a saxpy that requires access to a column of C and a column of A. These attributes are summarized in Table 1.1.1 together with an interpretation of what is going on when

Loop Order	Inner Loop	Inner Two Loops	Inner Loop Data Access
ijk	dot	vector × matrix	A by row, B by column
jik	dot	matrix × vector	A by row, B by column
ikj	saxpy	row gaxpy	B by row, C by row
jki	saxpy	column gaxpy	A by column, C by column
kij	saxpy	row outer product	B by row, C by row
kji	saxpy	column outer product	A by column, C by column

Table 1.1.1. *Matrix multiplication: loop orderings and properties*

the middle and inner loops are considered together. Each variant involves the same amount of arithmetic, but accesses the A, B, and C data differently. The ramifications of this are discussed in §1.5.

1.1.12 A Dot Product Formulation

The usual matrix multiplication procedure regards $A{\cdot}B$ as an array of dot products to be computed one at a time in left-to-right, top-to-bottom order. This is the idea behind Algorithm 1.1.5 which we rewrite using the colon notation to highlight the mission of the innermost loop:

Algorithm 1.1.6 (Dot Product Matrix Multiplication) If $A \in \mathbb{R}^{m\times r}$, $B \in \mathbb{R}^{r\times n}$, and $C \in \mathbb{R}^{m\times n}$ are given, then this algorithm overwrites C with $C + AB$.

```
for i = 1:m
    for j = 1:n
        C(i,j) = C(i,j) + A(i,:)·B(:,j)
    end
end
```

In the language of partitioned matrices, if

$$A = \begin{bmatrix} a_1^T \\ \vdots \\ a_m^T \end{bmatrix} \quad \text{and} \quad B = [\, b_1 \,|\, \cdots \,|\, b_n \,],$$

then Algorithm 1.1.6 has this interpretation:

```
for i = 1:m
    for j = 1:n
        c_ij = c_ij + a_i^T b_j
    end
end
```

Note that the purpose of the j-loop is to compute the ith row of the update. To emphasize this we could write

```
for i = 1:m
    c_i^T = c_i^T + a_i^T B
end
```

where

$$C = \begin{bmatrix} c_1^T \\ \vdots \\ c_m^T \end{bmatrix}$$

is a row partitioning of C. To say the same thing with the colon notation we write

```
for i = 1:m
    C(i,:) = C(i,:) + A(i,:)·B
end
```

Either way we see that the inner two loops of the ijk variant define a transposed gaxpy operation.

1.1.13 A Saxpy Formulation

Suppose A and C are column-partitioned as follows:

$$A = [\, a_1 \,|\, \cdots \,|\, a_r \,], \qquad C = [\, c_1 \,|\, \cdots \,|\, c_n \,].$$

By comparing jth columns in $C = C + AB$ we see that

$$c_j = c_j + \sum_{k=1}^{r} a_k b_{kj}, \qquad j = 1{:}n.$$

These vector sums can be put together with a sequence of saxpy updates.

Algorithm 1.1.7 (Saxpy Matrix Multiplication) If the matrices $A \in \mathbb{R}^{m\times r}$, $B \in \mathbb{R}^{r\times n}$, and $C \in \mathbb{R}^{m\times n}$ are given, then this algorithm overwrites C with $C + AB$.

```
for j = 1:n
    for k = 1:r
        C(:,j) = C(:,j) + A(:,k)·B(k,j)
    end
end
```

Note that the k-loop oversees a gaxpy operation:

```
for j = 1:n
    C(:,j) = C(:,j) + AB(:,j)
end
```

1.1.14 An Outer Product Formulation

Consider the kij variant of Algorithm 1.1.5:

```
for k = 1:r
    for j = 1:n
        for i = 1:m
            C(i,j) = C(i,j) + A(i,k)·B(k,j)
        end
    end
end
```

The inner two loops oversee the outer product update

$$C = C + a_k b_k^T$$

where

$$A = [\, a_1 \,|\, \cdots \,|\, a_r \,] \quad \text{and} \quad B = \begin{bmatrix} b_1^T \\ \vdots \\ b_r^T \end{bmatrix} \tag{1.1.3}$$

with $a_k \in \mathbb{R}^m$ and $b_k \in \mathbb{R}^n$. This renders the following implementation:

Algorithm 1.1.8 (Outer Product Matrix Multiplication) If the matrices $A \in \mathbb{R}^{m\times r}$, $B \in \mathbb{R}^{r\times n}$, and $C \in \mathbb{R}^{m\times n}$ are given, then this algorithm overwrites C with $C + AB$.

```
for k = 1:r
    C = C + A(:,k)·B(k,:)
end
```

Matrix-matrix multiplication is a sum of outer products.

1.1.15 Flops

One way to quantify the volume of work associated with a computation is to count flops. A *flop* is a floating point add, subtract, multiply, or divide. The number of flops in a given matrix computation is usually obtained by summing the amount of arithmetic associated with the most deeply nested statements. For matrix-matrix multiplication, e.g., Algorithm 1.1.5, this is the 2-flop statement

$$C(i,j) = C(i,j) + A(i,k)\cdot B(k,j).$$

If $A \in \mathbb{R}^{m\times r}$, $B \in \mathbb{R}^{r\times n}$, and $C \in \mathbb{R}^{m\times n}$, then this statement is executed mnr times. Table 1.1.2 summarizes the number of flops that are required for the common operations detailed above.

Operation	Dimension	Flops
$\alpha = x^T y$	$x, y \in \mathbb{R}^n$	$2n$
$y = y + ax$	$a \in \mathbb{R}$, $x, y \in \mathbb{R}^n$	$2n$
$y = y + Ax$	$A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$	$2mn$
$A = A + yx^T$	$A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$	$2mn$
$C = C + AB$	$A \in \mathbb{R}^{m\times r}$, $B \in \mathbb{R}^{r\times n}$, $C \in \mathbb{R}^{m\times n}$	$2mnr$

Table 1.1.2. *Important flop counts*

1.1.16 Big-Oh Notation/Perspective

In certain settings it is handy to use the "Big-Oh" notation when an order-of-magnitude assessment of work suffices. (We did this in §1.1.1.) Dot products are $O(n)$, matrix-vector products are $O(n^2)$, and matrix-matrix products are $O(n^3)$. Thus, to make efficient an algorithm that involves a mix of these operations, the focus should typically be on the highest order operations that are involved as they tend to dominate the overall computation.

1.1.17 The Notion of "Level" and the BLAS

The dot product and saxpy operations are examples of *level-1* operations. Level-1 operations involve an amount of data and an amount of arithmetic that are linear in the dimension of the operation. An m-by-n outer product update or a gaxpy operation involves a quadratic amount of data ($O(mn)$) and a quadratic amount of work ($O(mn)$). These are *level-2* operations. The matrix multiplication update $C = C+AB$ is a *level-3* operation. Level-3 operations are quadratic in data and cubic in work.

Important level-1, level-2, and level-3 operations are encapsulated in the "BLAS," an acronym that stands for Basic Linear Algebra Subprograms. See `LAPACK`. The design of matrix algorithms that are rich in level-3 BLAS operations is a major preoccupation of the field for reasons that have to do with data reuse (§1.5).

1.1.18 Verifying a Matrix Equation

In striving to understand matrix multiplication via outer products, we essentially established the matrix equation

$$AB = \sum_{k=1}^{r} a_k b_k^T, \tag{1.1.4}$$

where the a_k and b_k are defined by the partitionings in (1.1.3).

Numerous matrix equations are developed in subsequent chapters. Sometimes they are established algorithmically as above and other times they are proved at the ij-component level, e.g.,

$$\left[\sum_{k=1}^{r} a_k b_k^T\right]_{ij} = \sum_{k=1}^{r} \left[a_k b_k^T\right]_{ij} = \sum_{k=1}^{r} a_{ik} b_{kj} = [AB]_{ij}.$$

Scalar-level verifications such as this usually provide little insight. However, they are sometimes the only way to proceed.

1.1.19 Complex Matrices

On occasion we shall be concerned with computations that involve complex matrices. The vector space of m-by-n complex matrices is designated by $\mathbb{C}^{m\times n}$. The scaling, addition, and multiplication of complex matrices correspond exactly to the real case. However, transposition becomes *conjugate transposition*:

$$C = A^H \quad \Longrightarrow \quad c_{ij} = \overline{a}_{ji}.$$

The vector space of complex n-vectors is designated by $\mathbb{C}^n$. The dot product of complex n-vectors x and y is prescribed by

$$s = x^H y = \sum_{i=1}^{n} \overline{x}_i y_i.$$

If $A = B + iC \in \mathbb{C}^{m\times n}$, then we designate the real and imaginary parts of A by $\mathsf{Re}(A) = B$ and $\mathsf{Im}(A) = C$, respectively. The conjugate of A is the matrix $\bar{A} = (\bar{a}_{ij})$.

Problems

P1.1.1 Suppose $A \in \mathbb{R}^{n\times n}$ and $x \in \mathbb{R}^r$ are given. Give an algorithm for computing the first column of $M = (A - x_1 I)\cdots(A - x_r I)$.

P1.1.2 In a conventional 2-by-2 matrix multiplication $C = AB$, there are eight multiplications: $a_{11}b_{11}$, $a_{11}b_{12}$, $a_{21}b_{11}$, $a_{21}b_{12}$, $a_{12}b_{21}$, $a_{12}b_{22}$, $a_{22}b_{21}$, and $a_{22}b_{22}$. Make a table that indicates the order that these multiplications are performed for the ijk, jik, kij, ikj, jki, and kji matrix multiplication algorithms.

P1.1.3 Give an $O(n^2)$ algorithm for computing $C = (xy^T)^k$ where x and y are n-vectors.

P1.1.4 Suppose $D = ABC$ where $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times p}$, and $C \in \mathbb{R}^{p\times q}$. Compare the flop count of an algorithm that computes D via the formula $D = (AB)C$ versus the flop count for an algorithm that computes D using $D = A(BC)$. Under what conditions is the former procedure more flop-efficient than the latter?

P1.1.5 Suppose we have real n-by-n matrices C, D, E, and F. Show how to compute real n-by-n matrices A and B with just three real n-by-n matrix multiplications so that

$$A + iB = (C + iD)(E + iF).$$

Hint: Compute $W = (C + D)(E - F)$.

P1.1.6 Suppose $W \in \mathbb{R}^{n\times n}$ is defined by

$$w_{ij} = \sum_{p=1}^{n}\sum_{q=1}^{n} x_{ip}y_{pq}z_{qj}$$

where $X, Y, Z \in \mathbb{R}^{n\times n}$. If we use this formula for each w_{ij} then it would require $O(n^4)$ operations to set up W. On the other hand,

$$w_{ij} = \sum_{p=1}^{n} x_{ip}\left(\sum_{q=1}^{n} y_{pq}z_{qj}\right) = \sum_{p=1}^{n} x_{ip}u_{pj}$$

where $U = YZ$. Thus, $W = XU = XYZ$ and only $O(n^3)$ operations are required.

Use this methodology to develop an $O(n^3)$ procedure for computing the n-by-n matrix A defined by

$$a_{ij} = \sum_{k_1=1}^{n}\sum_{k_2=1}^{n}\sum_{k_3=1}^{n} E(k_1,i)F(k_1,i)G(k_2,k_1)H(k_2,k_3)F(k_2,k_3)G(k_3,j)$$

where $E, F, G, H \in \mathbb{R}^{n\times n}$. Hint. Transposes and pointwise products are involved.

Notes and References for §1.1

For an appreciation of the BLAS and their foundational role, see:

C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh (1979). "Basic Linear Algebra Subprograms for FORTRAN Usage," *ACM Trans. Math. Softw. 5,* 308–323.

J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson (1988). "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. Math. Softw. 14,* 1–17.

J.J. Dongarra, J. Du Croz, I.S. Duff, and S.J. Hammarling (1990). "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Softw. 16*, 1–17.

B. Kågström, P. Ling, and C. Van Loan (1991). "High-Performance Level-3 BLAS: Sample Routines for Double Precision Real Data," in *High Performance Computing II*, M. Durand and F. El Dabaghi (eds.), North-Holland, Amsterdam, 269–281.

L.S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R.C. Whaley (2002). "An Updated Set of Basic Linear Algebra Subprograms (BLAS)", *ACM Trans. Math. Softw. 28*, 135–151.

The order in which the operations in the matrix product $A_1 \cdots A_r$ are carried out affects the flop count if the matrices vary in dimension. (See P1.1.4.) Optimization in this regard requires dynamic programming, see:

T.H. Corman, C.E. Leiserson, R.L. Rivest, and C. Stein (2001). *Introduction to Algorithms*, MIT Press and McGraw-Hill, 331–339.

1.2 Structure and Efficiency

The efficiency of a given matrix algorithm depends upon several factors. Most obvious and what we treat in this section is the amount of required arithmetic and storage. How to reason about these important attributes is nicely illustrated by considering examples that involve triangular matrices, diagonal matrices, banded matrices, symmetric matrices, and permutation matrices. These are among the most important types of structured matrices that arise in practice, and various economies can be realized if they are involved in a calculation.

1.2.1 Band Matrices

A matrix is *sparse* if a large fraction of its entries are zero. An important special case is the *band matrix*. We say that $A \in \mathbb{R}^{m\times n}$ has *lower bandwidth* p if $a_{ij} = 0$ whenever $i > j+p$ and *upper bandwidth* q if $j > i+q$ implies $a_{ij} = 0$. Here is an example of an 8-by-5 matrix that has lower bandwidth 1 and upper bandwidth 2:

$$A = \begin{bmatrix} \times & \times & \times & 0 & 0 \\ \times & \times & \times & \times & 0 \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The $\times$'s designate arbitrary nonzero entries. This notation is handy to indicate the structure of a matrix and we use it extensively. Band structures that occur frequently are tabulated in Table 1.2.1.

Type of Matrix	Lower Bandwidth	Upper Bandwidth
Diagonal	0	0
Upper triangular	0	$n-1$
Lower triangular	$m-1$	0
Tridiagonal	1	1
Upper bidiagonal	0	1
Lower bidiagonal	1	0
Upper Hessenberg	1	$n-1$
Lower Hessenberg	$m-1$	1

Table 1.2.1. *Band terminology for m-by-n matrices*

1.2.2 Triangular Matrix Multiplication

To introduce band matrix "thinking" we look at the matrix multiplication update problem $C = C + AB$ where A, B, and C are each n-by-n and upper triangular. The 3-by-3 case is illuminating:

$$AB = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12}+a_{12}b_{22} & a_{11}b_{13}+a_{12}b_{23}+a_{13}b_{33} \\ 0 & a_{22}b_{22} & a_{22}b_{23}+a_{23}b_{33} \\ 0 & 0 & a_{33}b_{33} \end{bmatrix}.$$

It suggests that the product is upper triangular and that its upper triangular entries are the result of abbreviated inner products. Indeed, since $a_{ik}b_{kj} = 0$ whenever $k < i$ or $j < k$, we see that the update has the form

$$c_{ij} = c_{ij} + \sum_{k=i}^{j} a_{ik}b_{kj}$$

for all i and j that satisfy $i \leq j$. This yields the following algorithm:

Algorithm 1.2.1 (Triangular Matrix Multiplication) Given upper triangular matrices $A, B, C \in \mathbb{R}^{n\times n}$, this algorithm overwrites C with $C + AB$.

```
for i = 1:n
    for j = i:n
        for k = i:j
            C(i,j) = C(i,j) + A(i,k)·B(k,j)
        end
    end
end
```

1.2.3 The Colon Notation—Again

The dot product that the k-loop performs in Algorithm 1.2.1 can be succinctly stated if we extend the colon notation introduced in §1.1.8. If $A \in \mathbb{R}^{m\times n}$ and the integers p, q, and r satisfy $1 \leq p \leq q \leq n$ and $1 \leq r \leq m$, then

$$A(r,p{:}q) = \left[\, a_{rp} \,|\, \cdots \,|\, a_{rq} \,\right] \in \mathbb{R}^{1\times(q-p+1)}.$$

Likewise, if $1 \leq p \leq q \leq m$ and $1 \leq c \leq n$, then

$$A(p{:}q,c) = \begin{bmatrix} a_{pc} \\ \vdots \\ a_{qc} \end{bmatrix} \in \mathbb{R}^{q-p+1}.$$

With this notation we can rewrite Algorithm 1.2.1 as

```
for i = 1:n
    for j = i:n
        C(i,j) = C(i,j) + A(i,i:j)·B(i:j,j)
    end
end
```

This highlights the abbreviated inner products that are computed by the innermost loop.

1.2.4 Assessing Work

Obviously, upper triangular matrix multiplication involves less arithmetic than full matrix multiplication. Looking at Algorithm 1.2.1, we see that c_{ij} requires $2(j-i+1)$ flops if $(i \leq j)$. Using the approximations

$$\sum_{p=1}^{q} p = \frac{q(q+1)}{2} \approx \frac{q^2}{2}$$

and

$$\sum_{p=1}^{q} p^2 = \frac{q^3}{3} + \frac{q^2}{2} + \frac{q}{6} \approx \frac{q^3}{3},$$

we find that triangular matrix multiplication requires one-sixth the number of flops as full matrix multiplication:

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 2(j-i+1) \;=\; \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} 2j \;\approx\; \sum_{i=1}^{n}\frac{2(n-i+1)^2}{2} \;=\; \sum_{i=1}^{n} i^2 \;\approx\; \frac{n^3}{3}.$$

We throw away the low-order terms since their inclusion does not contribute to what the flop count "says." For example, an exact flop count of Algorithm 1.2.1 reveals that precisely $n^3/3 + n^2 + 2n/3$ flops are involved. For large n (the typical situation of interest) we see that the exact flop count offers no insight beyond the simple $n^3/3$ accounting.

Flop counting is a necessarily crude approach to the measurement of program efficiency since it ignores subscripting, memory traffic, and other overheads associated with program execution. We must not infer too much from a comparison of flop counts. We cannot conclude, for example, that triangular matrix multiplication is six times faster than full matrix multiplication. Flop counting captures just one dimension of what makes an algorithm efficient in practice. The equally relevant issues of vectorization and data locality are taken up in §1.5.

1.2.5 Band Storage

Suppose $A \in \mathbb{R}^{n\times n}$ has lower bandwidth p and upper bandwidth q and assume that p and q are much smaller than n. Such a matrix can be stored in a $(p+q+1)$-by-n array $A.band$ with the convention that

$$a_{ij} \;=\; A.band(i-j+q+1,\,j) \tag{1.2.1}$$

for all (i,j) that fall inside the band, e.g.,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix} \Rightarrow \begin{bmatrix} * & * & a_{13} & a_{24} & a_{35} & a_{46} \\ * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \end{bmatrix}.$$

Here, the "$*$" entries are unused. With this data structure, our column-oriented gaxpy algorithm (Algorithm 1.1.4) transforms to the following:

Algorithm 1.2.2 (Band Storage Gaxpy) Suppose $A \in \mathbb{R}^{n\times n}$ has lower bandwidth p and upper bandwidth q and is stored in the $A.band$ format (1.2.1). If $x, y \in \mathbb{R}^n$, then this algorithm overwrites y with $y + Ax$.

for $j = 1{:}n$
  $\alpha_1 = \max(1, j-q),\ \alpha_2 = \min(n, j+p)$
  $\beta_1 = \max(1, q+2-j),\ \beta_2 = \beta_1 + \alpha_2 - \alpha_1$
  $y(\alpha_1{:}\alpha_2) = y(\alpha_1{:}\alpha_2) \;+\; A.band(\beta_1{:}\beta_2, j)x(j)$
end

Notice that by storing A column by column in $A.band$, we obtain a column-oriented saxpy procedure. Indeed, Algorithm 1.2.2 is derived from Algorithm 1.1.4 by recognizing that each saxpy involves a vector with a small number of nonzeros. Integer arithmetic is used to identify the location of these nonzeros. As a result of this careful zero/nonzero analysis, the algorithm involves just $2n(p+q+1)$ flops with the assumption that p and q are much smaller than n.

1.2.6 Working with Diagonal Matrices

Matrices with upper and lower bandwidth zero are *diagonal*. If $D \in \mathbb{R}^{m\times n}$ is diagonal, then we use the notation

$$D = \text{diag}(d_1,\ldots,d_q), \quad q = \min\{m,n\} \quad \Longleftrightarrow \quad d_i = d_{ii}.$$

Shortcut notations when the dimension is clear include $\text{diag}(d)$ and $\text{diag}(d_i)$. Note that if $D = \text{diag}(d) \in \mathbb{R}^{n\times n}$ and $x \in \mathbb{R}^n$, then $Dx = d.*x$. If $A \in \mathbb{R}^{m\times n}$, then pre-multiplication by $D = \text{diag}(d_1,\ldots,d_m) \in \mathbb{R}^{m\times m}$ scales rows,

$$B = DA \qquad \Longleftrightarrow \qquad B(i,:) = d_i \cdot A(i,:),\ i = 1{:}m$$

while post-multiplication by $D = \text{diag}(d_1,\ldots,d_n) \in \mathbb{R}^{n\times n}$ scales columns,

$$B = AD \qquad \Longleftrightarrow \qquad B(:,j) = d_j \cdot A(:,j),\ j = 1{:}n.$$

Both of these special matrix-matrix multiplications require mn flops.

1.2.7 Symmetry

A matrix $A \in \mathbb{R}^{n\times n}$ is *symmetric* if $A^T = A$ and *skew-symmetric* if $A^T = -A$. Likewise, a matrix $A \in \mathbb{C}^{n\times n}$ is *Hermitian* if $A^H = A$ and *skew-Hermitian* if $A^H = -A$. Here are some examples:

$$\text{Symmetric:} \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}, \qquad \text{Hermitian:} \begin{bmatrix} 1 & 2-3i & 4-5i \\ 2+3i & 6 & 7-8i \\ 4+5i & 7+8i & 9 \end{bmatrix},$$

$$\text{Skew-Symmetric:} \begin{bmatrix} 0 & -2 & 3 \\ 2 & 0 & -5 \\ -3 & 5 & 0 \end{bmatrix}, \quad \text{Skew-Hermitian:} \begin{bmatrix} i & -2+3i & -4+5i \\ 2+3i & 6i & -7+8i \\ 4+5i & 7+8i & 9i \end{bmatrix}.$$

For such matrices, storage requirements can be halved by simply storing the lower triangle of elements, e.g.,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix} \quad \Leftrightarrow \quad A.vec = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}.$$

For general n, we set

$$A.vec((n-j/2)(j-1)+i) \ = \ a_{ij} \quad 1 \le j \le i \le n. \tag{1.2.2}$$

Here is a column-oriented gaxpy with the matrix A represented in $A.vec$.

Algorithm 1.2.3 (Symmetric Storage Gaxpy) Suppose $A \in \mathbb{R}^{n\times n}$ is symmetric and stored in the $A.vec$ style (1.2.2). If $x, y \in \mathbb{R}^n$, then this algorithm overwrites y with $y + Ax$.

```
for j = 1:n
    for i = 1:j − 1
        y(i) = y(i) + A.vec((i − 1)n − i(i − 1)/2 + j)x(j)
    end
    for i = j:n
        y(i) = y(i) + A.vec((j − 1)n − j(j − 1)/2 + i)x(j)
    end
end
```

This algorithm requires the same $2n^2$ flops that an ordinary gaxpy requires.

1.2.8 Permutation Matrices and the Identity

We denote the n-by-n *identity matrix* by I_n, e.g.,

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

We use the notation e_i to designate the ith column of I_n. If the rows of I_n are reordered, then the resulting matrix is said to be a *permutation matrix*, e.g.,

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \tag{1.2.3}$$

The representation of an n-by-n permutation matrix requires just an n-vector of integers whose components specify where the 1's occur. For example, if $v \in \mathbb{R}^n$ has the property that v_i specifies the column where the "1" occurs in row i, then $y = Px$ implies that $y_i = x_{v_i}$, $i = 1{:}n$. In the example above, the underlying v-vector is $v = [\,2\;4\;3\;1\,]$.

1.2.9 Specifying Integer Vectors and Submatrices

For permutation matrix work and block matrix manipulation (§1.3) it is convenient to have a method for specifying structured integer vectors of subscripts. The MATLAB colon notation is again the proper vehicle and a few examples suffice to show how it works. If $n = 8$, then

$$\begin{aligned} v = 1{:}2{:}n &\quad\Longrightarrow\quad v = [\,1\;3\;5\;7\,], \\ v = n{:}{-}1{:}1 &\quad\Longrightarrow\quad v = [\,8\;7\;6\;5\;4\;3\;2\;1\,], \\ v = [\,(1{:}2{:}n)\;(2{:}2{:}n)\,] &\quad\Longrightarrow\quad v = [\,1\;3\;5\;7\;2\;4\;6\;8\,]. \end{aligned}$$

Suppose $A \in \mathbb{R}^{m\times n}$ and that $v \in \mathbb{R}^r$ and $w \in \mathbb{R}^s$ are integer vectors with the property that $1 \le v_i \le m$ and $1 \le w_i \le n$. If $B = A(v,w)$, then $B \in \mathbb{R}^{r\times s}$ is the matrix defined by $b_{ij} = a_{v_i,w_j}$ for $i = 1{:}r$ and $j = 1{:}s$. Thus, if $A \in \mathbb{R}^{8\times 8}$, then

$$A(1{:}2{:}8, 2{:}2{:}8) = \begin{bmatrix} a_{12} & a_{14} & a_{16} & a_{18} \\ a_{32} & a_{34} & a_{36} & a_{38} \\ a_{52} & a_{54} & a_{56} & a_{58} \\ a_{72} & a_{74} & a_{76} & a_{78} \end{bmatrix}.$$

1.2.10 Working with Permutation Matrices

Using the colon notation, the 4-by-4 permutation matrix in (1.2.3) is defined by $P = I_4(v,:)$ where $v = [\,2\ 4\ 3\ 1\,]$. In general, if $v \in \mathbb{R}^n$ is a permutation of the vector $1{:}n = [1, 2, \ldots, n]$ and $P = I_n(v,:)$, then

$$\begin{aligned} y = Px \quad &\Longrightarrow \quad y = x(v) \quad \Longrightarrow \quad y_i = x_{v_i},\ i = 1{:}n \\ y = P^Tx \quad &\Longrightarrow \quad y(v) = x \quad \Longrightarrow \quad y_{v_i} = x_i,\ i = 1{:}n \end{aligned}$$

The second result follows from the fact that v_i is the *row* index of the "1" in column i of P^T. Note that $P^T(Px) = x$. The inverse of a permutation matrix is its transpose.

The action of a permutation matrix on a given matrix $A \in \mathbb{R}^{m\times n}$ is easily described. If $P = I_m(v,:)$ and $Q = I_n(w,:)$, then $PAQ^T = A(v,w)$. It also follows that $I_n(v,:)\cdot I_n(w,:) = I_n(w(v),:)$. Although permutation operations involve no flops, they move data and contribute to execution time, an issue that is discussed in §1.5.

1.2.11 Three Famous Permutation Matrices

The *exchange permutation* $\mathcal{E}_n$ turns vectors upside down, e.g.,

$$y = \mathcal{E}_4x = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_4 \\ x_3 \\ x_2 \\ x_1 \end{bmatrix}.$$

In general, if $v = n{:}-1{:}1$, then the n-by-n exchange permutation is given by $\mathcal{E}_n = I_n(v,:)$. No change results if a vector is turned upside down twice and thus, $\mathcal{E}_n^T\mathcal{E}_n = \mathcal{E}_n^2 = I_n$.

The *downshift permutation* $\mathcal{D}_n$ pushes the components of a vector down one notch with wraparound, e.g.,

$$y = \mathcal{D}_4x = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_4 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

In general, if $v = [\,(2{:}n)\ 1\,]$, then the n-by-n downshift permutation is given by $\mathcal{D}_n = I_n(v,:)$. Note that $\mathcal{D}_n^T$ can be regarded as an *upshift* permutation.

The *mod-p perfect shuffle permutation* $\mathcal{P}_{p,r}$ treats the components of the input vector $x \in \mathbb{R}^n$, $n = pr$, as cards in a deck. The deck is cut into p equal "piles" and

reassembled by taking one card from each pile in turn. Thus, if $p = 3$ and $r = 4$, then the piles are $x(1{:}4)$, $x(5{:}8)$, and $x(9{:}12)$ and

$$y = \mathcal{P}_{3,4}x = I_{pr}([\,1\ 5\ 9\ 2\ 6\ 10\ 3\ 7\ 11\ 4\ 8\ 12\,],:)x = \begin{bmatrix} x(1{:}4{:}12) \\ x(2{:}4{:}12) \\ x(3{:}4{:}12) \\ x(4{:}4{:}12) \end{bmatrix}.$$

In general, if $n = pr$, then

$$\mathcal{P}_{p,r} = I_n([\,(1{:}r{:}n)\ \ (2{:}r{:}n)\ \cdots\ (r{:}r{:}n)],:)$$

and it can be shown that

$$\mathcal{P}_{p,r}^T = I_n([\,(1{:}p{:}n)\ \ (2{:}p{:}n)\ \cdots\ (p{:}p{:}n)\,],:). \tag{1.2.4}$$

Continuing with the card deck metaphor, $\mathcal{P}_{p,r}^T$ reassembles the card deck by placing all the x_i having $i \bmod p = 1$ first, followed by all the x_i having $i \bmod p = 2$ second, and so on.

Problems

P1.2.1 Give an algorithm that overwrites A with A^2 where $A \in \mathbb{R}^{n\times n}$. How much extra storage is required? Repeat for the case when A is upper triangular.

P1.2.2 Specify an algorithm that computes the first column of the matrix $M = (A-\lambda_1 I)\cdots(A-\lambda_r I)$ where $A \in \mathbb{R}^{n\times n}$ is upper Hessenberg and $\lambda_1,\ldots,\lambda_r$ are given scalars. How many flops are required assuming that $r \ll n$?

P1.2.3 Give a column saxpy algorithm for the n-by-n matrix multiplication problem $C = C + AB$ where A is upper triangular and B is lower triangular.

P1.2.4 Extend Algorithm 1.2.2 so that it can handle rectangular band matrices. Be sure to describe the underlying data structure.

P1.2.5 If $A = B + iC$ is Hermitian with $B \in \mathbb{R}^{n\times n}$, then it is easy to show that $B^T = B$ and $C^T = -C$. Suppose we represent A in an array $A.herm$ with the property that $A.herm(i,j)$ houses b_{ij} if $i \geq j$ and c_{ij} if $j > i$. Using this data structure, write a matrix-vector multiply function that computes $\mathsf{Re}(z)$ and $\mathsf{Im}(z)$ from $\mathsf{Re}(x)$ and $\mathsf{Im}(x)$ so that $z = Ax$.

P1.2.6 Suppose $X \in \mathbb{R}^{n\times p}$ and $A \in \mathbb{R}^{n\times n}$ are given and that A is symmetric. Give an algorithm for computing $B = X^TAX$ assuming that both A and B are to be stored using the symmetric storage scheme presented in §1.2.7.

P1.2.7 Suppose $a \in \mathbb{R}^n$ is given and that $A \in \mathbb{R}^{n\times n}$ has the property that $a_{ij} = a_{|i-j|+1}$. Give an algorithm that overwrites y with $y + Ax$ where $x, y \in \mathbb{R}^n$ are given.

P1.2.8 Suppose $a \in \mathbb{R}^n$ is given and that $A \in \mathbb{R}^{n\times n}$ has the property that $a_{ij} = a_{((i+j-1) \bmod n)+1}$. Give an algorithm that overwrites y with $y + Ax$ where $x, y \in \mathbb{R}^n$ are given.

P1.2.9 Develop a compact storage scheme for symmetric band matrices and write the corresponding gaxpy algorithm.

P1.2.10 Suppose $A \in \mathbb{R}^{n\times n}$, $u \in \mathbb{R}^n$, and $v \in \mathbb{R}^n$ are given and that $k \leq n$ is an integer. Show how to compute $X \in \mathbb{R}^{n\times k}$ and $Y \in \mathbb{R}^{n\times k}$ so that $(A + uv^T)^k = A^k + XY^T$. How many flops are required?

P1.2.11 Suppose $x \in \mathbb{R}^n$. Write a single-loop algorithm that computes $y = \mathcal{D}_n^k x$ where k is a positive integer and $\mathcal{D}_n$ is defined in §1.2.11.

P1.2.12 (a) Verify (1.2.4). (b) Show that $\mathcal{P}_{p,r}^T = \mathcal{P}_{r,p}$.

P1.2.13 The number of n-by-n permutation matrices is $n!$. How many of these are symmetric?

Notes and References for §1.2

See LAPACK for a discussion about appropriate data structures when symmetry and/or bandedness is present in addition to

F.G. Gustavson (2008). "The Relevance of New Data Structure Approaches for Dense Linear Algebra in the New Multi-Core/Many-Core Environments," in *Proceedings of the 7th international Conference on Parallel Processing and Applied Mathematics,* Springer-Verlag, Berlin, 618–621.

The exchange, downshift, and perfect shuffle permutations are discussed in Van Loan (FFT).

1.3 Block Matrices and Algorithms

A block matrix is a matrix whose entries are themselves matrices. It is a point of view. For example, an 8-by-15 matrix of scalars can be regarded as a 2-by-3 block matrix with 4-by-5 entries. Algorithms that manipulate matrices at the block level are often more efficient because they are richer in level-3 operations. The derivation of many important algorithms is often simplified by using block matrix notation.

1.3.1 Block Matrix Terminology

Column and row partitionings (§1.1.7) are special cases of matrix blocking. In general, we can partition both the rows and columns of an m-by-n matrix A to obtain

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1r} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix}$$
$$\begin{matrix} n_1 & & n_r \end{matrix}$$

where $m_1 + \cdots + m_q = m$, $n_1 + \cdots + n_r = n$, and $A_{\alpha\beta}$ designates the (α, β) block (submatrix). With this notation, block $A_{\alpha\beta}$ has dimension m_α-by-n_β and we say that $A = (A_{\alpha\beta})$ is a q-by-r block matrix.

Terms that we use to describe well-known band structures for matrices with scalar entries have natural block analogs. Thus,

$$\text{diag}(A_{11}, A_{22}, A_{33}) = \begin{bmatrix} A_{11} & 0 & 0 \\ 0 & A_{22} & 0 \\ 0 & 0 & A_{33} \end{bmatrix}$$

is *block diagonal* while the matrices

$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix}, \quad U = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}, \quad T = \begin{bmatrix} T_{11} & T_{12} & 0 \\ T_{21} & T_{22} & T_{23} \\ 0 & T_{32} & T_{33} \end{bmatrix},$$

are, respectively, *block lower triangular*, *block upper triangular*, and *block tridiagonal*. The blocks *do not* have to be square in order to use this *block sparse* terminology.

1.3.2 Block Matrix Operations

Block matrices can be scaled and transposed:

$$\mu \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix} = \begin{bmatrix} \mu A_{11} & \mu A_{12} \\ \mu A_{21} & \mu A_{22} \\ \mu A_{31} & \mu A_{32} \end{bmatrix},$$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}^T = \begin{bmatrix} A_{11}^T & A_{21}^T & A_{31}^T \\ A_{12}^T & A_{22}^T & A_{32}^T \end{bmatrix}.$$

Note that the transpose of the original (i,j) block becomes the (j,i) block of the result. Identically blocked matrices can be added by summing the corresponding blocks:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix} = \begin{bmatrix} A_{11}+B_{11} & A_{12}+B_{12} \\ A_{21}+B_{21} & A_{22}+B_{22} \\ A_{31}+B_{31} & A_{32}+B_{32} \end{bmatrix}.$$

Block matrix multiplication requires more stipulations about dimension. For example, if

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22} \\ A_{31}B_{11}+A_{32}B_{21} & A_{31}B_{12}+A_{32}B_{22} \end{bmatrix}$$

is to make sense, then the column dimensions of A_{11}, A_{21}, and A_{31} must each be equal to the row dimension of both B_{11} and B_{12}. Likewise, the column dimensions of A_{12}, A_{22}, and A_{32} must each be equal to the row dimensions of both B_{21} and B_{22}.

Whenever a block matrix addition or multiplication is indicated, it is assumed that the row and column dimensions of the blocks satisfy all the necessary constraints. In that case we say that the operands are *partitioned conformably* as in the following theorem.

Theorem 1.3.1. *If*

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1s} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qs} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix}, \qquad B = \begin{bmatrix} B_{11} & \cdots & B_{1r} \\ \vdots & & \vdots \\ B_{s1} & \cdots & B_{sr} \end{bmatrix} \begin{matrix} p_1 \\ \\ p_s \end{matrix},$$

$$\begin{matrix} p_1 & & p_s \end{matrix} \qquad \begin{matrix} n_1 & & n_r \end{matrix}$$

and we partition the product $C = AB$ as follows,

$$C = \begin{bmatrix} C_{11} & \cdots & C_{1r} \\ \vdots & & \vdots \\ C_{q1} & \cdots & C_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix},$$

$$\begin{matrix} n_1 & & n_r \end{matrix}$$

then for $\alpha = 1{:}q$ and $\beta = 1{:}r$ we have $C_{\alpha\beta} = \sum_{\gamma=1}^{s} A_{\alpha\gamma}B_{\gamma\beta}$.

Proof. The proof is a tedious exercise in subscripting. Suppose $1 \leq \alpha \leq q$ and $1 \leq \beta \leq r$. Set $M = m_1 + \cdots + m_{\alpha-1}$ and $N = n_1 + \cdots n_{\beta-1}$. It follows that if $1 \leq i \leq m_\alpha$ and $1 \leq j \leq n_\beta$ then

$$[C_{\alpha\beta}]_{ij} = \sum_{k=1}^{p_1+\cdots p_s} a_{M+i,k} b_{k,N+j} = \sum_{\gamma=1}^{s} \sum_{k=p_1+\cdots+p_{\gamma-1}+1}^{p_1+\cdots+p_\gamma} a_{M+i,k} b_{k,N+j}$$

$$= \sum_{\gamma=1}^{s} \sum_{k=1}^{p_\gamma} [A_{\alpha\gamma}]_{ik} [B_{\gamma\beta}]_{kj} = \sum_{\gamma=1}^{s} [A_{\alpha\gamma} B_{\gamma\beta}]_{ij} = \left[\sum_{\gamma=1}^{s} A_{\alpha\gamma} B_{\gamma\beta} \right]_{ij}.$$

Thus, $C_{\alpha\beta} = A_{\alpha,1} B_{1,\beta} + \cdots + A_{\alpha,s} B_{s,\beta}$. □

If you pay attention to dimension and remember that matrices do not commute, i.e., $A_{11}B_{11} + A_{12}B_{21} \neq B_{11}A_{11} + B_{21}A_{12}$, then block matrix manipulation is just ordinary matrix manipulation with the a_{ij}'s and b_{ij}'s written as A_{ij}'s and B_{ij}'s!

1.3.3 Submatrices

Suppose $A \in \mathbb{R}^{m\times n}$. If $\alpha = [\alpha_1, \ldots, \alpha_s]$ and $\beta = [\beta_1, \ldots, \beta_t]$ are integer vectors with distinct components that satisfy $1 \leq \alpha_i \leq m$, and $1 \leq \beta_i \leq n$, then

$$A(\alpha, \beta) = \begin{bmatrix} a_{\alpha_1,\beta_1} & \cdots & a_{\alpha_1,\beta_t} \\ \vdots & \ddots & \vdots \\ a_{\alpha_s,\beta_1} & \cdots & a_{\alpha_s,\beta_t} \end{bmatrix}$$

is an s-by-t *submatrix* of A. For example, if $A \in \mathbb{R}^{8\times 6}$, $\alpha = [2\ 4\ 6\ 8]$, and $\beta = [4\ 5\ 6]$, then

$$A(\alpha, \beta) = \begin{bmatrix} a_{24} & a_{25} & a_{26} \\ a_{44} & a_{45} & a_{46} \\ a_{64} & a_{65} & a_{66} \\ a_{84} & a_{85} & a_{86} \end{bmatrix}.$$

If $\alpha = \beta$, then $A(\alpha, \beta)$ is a *principal submatrix*. If $\alpha = \beta = 1{:}k$ and $1 \leq k \leq \min\{m, n\}$, then $A(\alpha, \beta)$ is a *leading principal submatrix*.

If $A \in \mathbb{R}^{m\times n}$ and

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1s} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qs} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix}, \qquad \begin{matrix} n_1 & & n_r \end{matrix}$$

then the colon notation can be used to specify the individual blocks. In particular,

$$A_{ij} = A(\tau + 1{:}\tau + m_i, \mu + 1{:}\mu + n_j)$$

where $\tau = m_1 + \cdots + m_{i-1}$ and $\mu = n_1 + \cdots + n_{j-1}$. Block matrix notation is valuable for the way in which it hides subscript range expressions.

1.3.4 The Blocked Gaxpy

As an exercise in block matrix manipulation and submatrix designation, we consider two block versions of the gaxpy operation $y = y + Ax$ where $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$. If

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_q \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix} \quad \text{and} \quad y = \begin{bmatrix} y_1 \\ \vdots \\ y_q \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix},$$

then

$$\begin{bmatrix} y_1 \\ \vdots \\ y_q \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_q \end{bmatrix} + \begin{bmatrix} A_1 \\ \vdots \\ A_q \end{bmatrix} x,$$

and we obtain

$\alpha = 0$
for $i = 1{:}q$
  $idx = \alpha{+}1 : \alpha{+}m_i$
  $y(idx) = y(idx) + A(idx, :)\cdot x$
  $\alpha = \alpha + m_i$
end

The assignment to $y(idx)$ corresponds to $y_i = y_i + A_i x$. This row-blocked version of the gaxpy computation breaks the given gaxpy into q "shorter" gaxpys. We refer to A_i as the ith block row of A.

Likewise, with the partitionings

$$A = \begin{bmatrix} A_1 \\ n_1 \end{bmatrix} \cdots \begin{bmatrix} A_r \\ n_r \end{bmatrix} \quad \text{and} \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_r \end{bmatrix} \begin{matrix} n_1 \\ \\ n_r \end{matrix},$$

we see that

$$y = y + \begin{bmatrix} A_1 \\ \end{bmatrix} \cdots \begin{bmatrix} A_r \\ \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_r \end{bmatrix} = y + \sum_{j=1}^{r} A_j x_j$$

and we obtain

$\beta = 0$
for $j = 1{:}r$
  $jdx = \beta{+}1 : \beta{+}n_j$
  $y = y + A(:, jdx)\cdot x(jdx)$
  $\beta = \beta + n_j$
end

The assignment to y corresponds to $y = y + A_j x_j$. This column-blocked version of the gaxpy computation breaks the given gaxpy into r "thinner" gaxpys. We refer to A_j as the jth block column of A.

1.3.5 Block Matrix Multiplication

Just as ordinary, scalar-level matrix multiplication can be arranged in several possible ways, so can the multiplication of block matrices. To illustrate this with a minimum of subscript clutter, we consider the update

$$C = C + AB$$

where we regard $A = (A_{\alpha\beta})$, $B = (B_{\alpha\beta})$, and $C = (C_{\alpha\beta})$ as N-by-N block matrices with ℓ-by-ℓ blocks. From Theorem 1.3.1 we have

$$C_{\alpha\beta} = C_{\alpha\beta} + \sum_{\gamma=1}^{N} A_{\alpha\gamma}B_{\gamma\beta}, \qquad \alpha = 1{:}N, \quad \beta = 1{:}N.$$

If we organize a matrix multiplication procedure around this summation, then we obtain a block analog of Algorithm 1.1.5:

```
for α = 1:N
    i = (α − 1)ℓ + 1:αℓ
    for β = 1:N
        j = (β − 1)ℓ + 1:βℓ
        for γ = 1:N
            k = (γ − 1)ℓ + 1:γℓ
            C(i, j) = C(i, j) + A(i, k)·B(k, j)
        end
    end
end
```

Note that, if $\ell = 1$, then $\alpha \equiv i$, $\beta \equiv j$, and $\gamma \equiv k$ and we revert to Algorithm 1.1.5.

Analogously to what we did in §1.1, we can obtain different variants of this procedure by playing with loop orders and blocking strategies. For example, corresponding to

$$\begin{bmatrix} C_{11} & \cdots & C_{1N} \\ \vdots & \ddots & \vdots \\ C_{N1} & \cdots & C_{NN} \end{bmatrix} + \begin{bmatrix} A_1 \\ \vdots \\ A_N \end{bmatrix} \begin{bmatrix} B_1 & \cdots & B_N \end{bmatrix}$$

where $A_i \in \mathbb{R}^{\ell\times n}$ and $B_j \in \mathbb{R}^{n\times\ell}$, we obtain the following block outer product computation:

```
for i = 1:N
    for j = 1:N
        C_ij = C_ij + A_i B_j
    end
end
```

1.3.6 The Kronecker Product

It is sometimes the case that the entries in a block matrix A are all scalar multiples of the same matrix. This means that A is a *Kronecker product*. Formally, if $B \in \mathbb{R}^{m_1 \times n_1}$ and $C \in \mathbb{R}^{m_2 \times n_2}$, then their Kronecker product $B \otimes C$ is an m_1-by-n_1 block matrix whose (i,j) block is the m_2-by-n_2 matrix $b_{ij}C$. Thus, if

$$A = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} \otimes \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

then

$$A = \left[\begin{array}{ccc|ccc} b_{11}c_{11} & b_{11}c_{12} & b_{11}c_{13} & b_{12}c_{11} & b_{12}c_{12} & b_{12}c_{13} \\ b_{11}c_{21} & b_{11}c_{22} & b_{11}c_{23} & b_{12}c_{21} & b_{12}c_{22} & b_{12}c_{23} \\ b_{11}c_{31} & b_{11}c_{32} & b_{11}c_{33} & b_{12}c_{31} & b_{12}c_{32} & b_{12}c_{33} \\ \hline b_{21}c_{11} & b_{21}c_{12} & b_{21}c_{13} & b_{22}c_{11} & b_{22}c_{12} & b_{22}c_{13} \\ b_{21}c_{21} & b_{21}c_{22} & b_{21}c_{23} & b_{22}c_{21} & b_{22}c_{22} & b_{22}c_{23} \\ b_{21}c_{31} & b_{21}c_{32} & b_{21}c_{33} & b_{22}c_{31} & b_{22}c_{32} & b_{22}c_{33} \\ \hline b_{31}c_{11} & b_{31}c_{12} & b_{31}c_{13} & b_{32}c_{11} & b_{32}c_{12} & b_{32}c_{13} \\ b_{31}c_{21} & b_{31}c_{22} & b_{31}c_{23} & b_{32}c_{21} & b_{32}c_{22} & b_{32}c_{23} \\ b_{31}c_{31} & b_{31}c_{32} & b_{31}c_{33} & b_{32}c_{31} & b_{32}c_{32} & b_{32}c_{33} \end{array}\right].$$

This type of highly structured blocking occurs in many applications and results in dramatic economies when fully exploited.

Note that if B has a band structure, then $B \otimes C$ "inherits" that structure at the block level. For example, if

$$B \text{ is } \left\{ \begin{array}{l} \text{diagonal} \\ \text{tridiagonal} \\ \text{lower triangular} \\ \text{upper triangular} \end{array} \right\} \text{ then } B \otimes C \text{ is } \left\{ \begin{array}{l} \text{block diagonal} \\ \text{block tridiagonal} \\ \text{block lower triangular} \\ \text{block upper triangular} \end{array} \right\}.$$

Important Kronecker product properties include:

$$(B \otimes C)^T = B^T \otimes C^T, \tag{1.3.1}$$

$$(B \otimes C)(D \otimes F) = BD \otimes CF, \tag{1.3.2}$$

$$(B \otimes C)^{-1} = B^{-1} \otimes C^{-1}, \tag{1.3.3}$$

$$B \otimes (C \otimes D) = (B \otimes C) \otimes D. \tag{1.3.4}$$

Of course, the products BD and CF must be defined for (1.3.2) to make sense. Likewise, the matrices B and C must be nonsingular in (1.3.3).

In general, $B \otimes C \neq C \otimes B$. However, there is a connection between these two matrices via the perfect shuffle permutation that is defined in §1.2.11. If $B \in \mathbb{R}^{m_1 \times n_1}$ and $C \in \mathbb{R}^{m_2 \times n_2}$, then

$$P(B \otimes C)Q^T = C \otimes B \tag{1.3.5}$$

where $P = \mathcal{P}_{m_1,m_2}$ and $Q = \mathcal{P}_{n_1,n_2}$.

1.3.7 Reshaping Kronecker Product Expressions

A matrix-vector product in which the matrix is a Kronecker product is "secretly" a matrix-matrix-matrix product. For example, if $B \in \mathbb{R}^{3\times 2}$, $C \in \mathbb{R}^{m\times n}$, and $x_1, x_2 \in \mathbb{R}^n$, then

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = (B \otimes C)\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_{11}C & b_{12}C \\ b_{21}C & b_{22}C \\ b_{31}C & b_{32}C \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$
$$= \begin{bmatrix} b_{11}Cx_1 + b_{12}Cx_2 \\ b_{21}Cx_1 + b_{22}Cx_2 \\ b_{31}Cx_1 + b_{32}Cx_2 \end{bmatrix}$$

where $y_1, y_2, y_3 \in \mathbb{R}^m$. On the other hand, if we define the matrices

$$X = [\, x_1 \; x_2 \,] \quad \text{and} \quad Y = [\, y_1 \; y_2 \; y_3 \,],$$

then $Y = CXB^T$.

To be precise about this reshaping, we introduce the vec operation. If $X \in \mathbb{R}^{m\times n}$, then $\mathsf{vec}(X)$ is an nm-by-1 vector obtained by "stacking" X's columns:

$$\mathsf{vec}(X) = \begin{bmatrix} X(:,1) \\ \vdots \\ X(:,n) \end{bmatrix}.$$

If $B \in \mathbb{R}^{m_1\times n_1}$, $C \in \mathbb{R}^{m_2\times n_2}$, and $X \in \mathbb{R}^{n_1\times m_2}$, then

$$Y = CXB^T \;\Leftrightarrow\; \mathsf{vec}(Y) = (B \otimes C)\mathsf{vec}(X). \tag{1.3.6}$$

Note that if $B, C, X \in \mathbb{R}^{n\times n}$, then $Y = CXB^T$ costs $O(n^3)$ to evaluate while the disregard of Kronecker structure in $y = (B \otimes C)x$ leads to an $O(n^4)$ calculation. This is why reshaping is central for effective Kronecker product computation. The reshape operator is handy in this regard. If $A \in \mathbb{R}^{m\times n}$ and $m_1 n_1 = mn$, then

$$B = \mathsf{reshape}(A, m_1, n_1)$$

is the m_1-by-n_1 matrix defined by $\mathsf{vec}(B) = \mathsf{vec}(A)$. Thus, if $A \in \mathbb{R}^{3\times 4}$, then

$$\mathsf{reshape}(A, 2, 6) = \begin{bmatrix} a_{11} & a_{31} & a_{22} & a_{13} & a_{33} & a_{24} \\ a_{21} & a_{12} & a_{32} & a_{23} & a_{14} & a_{34} \end{bmatrix}.$$

1.3.8 Multiple Kronecker Products

Note that $A = B \otimes C \otimes D$ can be regarded as a block matrix whose entries are block matrices. In particular, $b_{ij}c_{k\ell}D$ is the (k,ℓ) block of A's (i,j) block.

As an example of a multiple Kronecker product computation, let us consider the calculation of $y = (B \otimes C \otimes D)x$ where $B, C, D \in \mathbb{R}^{n\times n}$ and $x \in \mathbb{R}^N$ with $N = n^3$. Using (1.3.6) it follows that

$$\mathsf{reshape}(y, n^2, n) = (C \otimes D) \cdot \mathsf{reshape}(x, n^2, n) \cdot B^T.$$

Thus, if

$$F = \text{reshape}(x, n^2, n) \cdot B^T,$$

then $G = (C \otimes D)F \in \mathbb{R}^{n^2 \times n}$ can computed column-by-column using (1.3.6):

$$G(:,k) = \text{reshape}(D \cdot \text{reshape}(F(:,k), n, n) \cdot C^T, n^2, 1) \qquad k = 1{:}n.$$

It follows that $y = \text{reshape}(G, N, 1)$. A careful accounting reveals that $6n^4$ flops are required. Ordinarily, a matrix-vector product of this dimension would require $2n^6$ flops.

The Kronecker product has a prominent role to play in tensor computations and in §13.1 we detail more of its properties.

1.3.9 A Note on Complex Matrix Multiplication

Consider the complex matrix multiplication update

$$C_1 + iC_2 = (C_1 + iC_2) + (A_1 + iA_2)(B_1 + iB_2)$$

where all the matrices are real and $i^2 = -1$. Comparing the real and imaginary parts we conclude that

$$\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} + \begin{bmatrix} A_1 & -A_2 \\ A_2 & A_1 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}.$$

Thus, complex matrix multiplication corresponds to a structured real matrix multiplication that has expanded dimension.

1.3.10 Hamiltonian and Symplectic Matrices

While on the topic of 2-by-2 block matrices, we identify two classes of structured matrices that arise at various points later on in the text. A matrix $M \in \mathbb{R}^{2n \times 2n}$ is a *Hamiltonian matrix* if it has the form

$$M = \begin{bmatrix} A & G \\ F & -A^T \end{bmatrix}$$

where $A, F, G \in \mathbb{R}^{n \times n}$ and F and G are symmetric. Hamiltonian matrices arise in optimal control and other application areas. An equivalent definition can be given in terms of the permutation matrix

$$J = \begin{bmatrix} 0 & I_n \\ -I_n & 0 \end{bmatrix}.$$

In particular, if

$$JMJ^T = -M^T,$$

then M is Hamiltonian. A related class of matrices are the symplectic matrices. A matrix $S \in \mathbb{R}^{2n \times 2n}$ is *symplectic* if

$$S^T J S = J.$$

If

$$S = \begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix}$$

where the blocks are n-by-n, then it follows that both $S_{11}^T S_{21}$ and $S_{22}^T S_{12}$ are symmetric and $S_{11}^T S_{22} = I_n + S_{21}^T S_{12}$.

1.3.11 Strassen Matrix Multiplication

We conclude this section with a completely different approach to the matrix-matrix multiplication problem. The starting point in the discussion is the 2-by-2 block matrix product

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where each block is square. In the ordinary algorithm, $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$. There are 8 multiplies and 4 adds. Strassen (1969) has shown how to compute C with just 7 multiplies and 18 adds:

$$\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\
P_2 &= (A_{21} + A_{22})B_{11}, \\
P_3 &= A_{11}(B_{12} - B_{22}), \\
P_4 &= A_{22}(B_{21} - B_{11}), \\
P_5 &= (A_{11} + A_{12})B_{22}, \\
P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}), \\
C_{11} &= P_1 + P_4 - P_5 + P_7, \\
C_{12} &= P_3 + P_5, \\
C_{21} &= P_2 + P_4, \\
C_{22} &= P_1 + P_3 - P_2 + P_6.
\end{aligned}$$

These equations are easily confirmed by substitution. Suppose $n = 2m$ so that the blocks are m-by-m. Counting adds and multiplies in the computation $C = AB$, we find that conventional matrix multiplication involves $(2m)^3$ multiplies and $(2m)^3 - (2m)^2$ adds. In contrast, if Strassen's algorithm is applied *with conventional multiplication at the block level,* then $7m^3$ multiplies and $7m^3 + 11m^2$ adds are required. If $m \gg 1$, then the Strassen method involves about 7/8 the arithmetic of the fully conventional algorithm.

Now recognize that we can recur on the Strassen idea. In particular, we can apply the Strassen algorithm to each of the half-sized block multiplications associated with the P_i. Thus, if the original A and B are n-by-n and $n = 2^q$, then we can repeatedly apply the Strassen multiplication algorithm. At the bottom "level," the blocks are 1-by-1.

Of course, there is no need to recur down to the $n = 1$ level. When the block size gets sufficiently small, $(n \leq n_{\min})$, it may be sensible to use conventional matrix multiplication when finding the P_i. Here is the overall procedure:

Algorithm 1.3.1 (Strassen Matrix Multiplication) Suppose $n = 2^q$ and that $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times n}$. If $n_{\min} = 2^d$ with $d \le q$, then this algorithm computes $C = AB$ by applying Strassen procedure recursively.

function $C = \mathsf{strass}(A, B, n, n_{\min})$
  if $n \le n_{\min}$
    $C = AB$ (conventionally computed)
  else
    $m = n/2;\ u = 1{:}m;\ v = m+1{:}n$
    $P_1 = \mathsf{strass}(A(u,u) + A(v,v), B(u,u) + B(v,v), m, n_{\min})$
    $P_2 = \mathsf{strass}(A(v,u) + A(v,v), B(u,u), m, n_{\min})$
    $P_3 = \mathsf{strass}(A(u,u), B(u,v) - B(v,v), m, n_{\min})$
    $P_4 = \mathsf{strass}(A(v,v), B(v,u) - B(u,u), m, n_{\min})$
    $P_5 = \mathsf{strass}(A(u,u) + A(u,v), B(v,v), m, n_{\min})$
    $P_6 = \mathsf{strass}(A(v,u) - A(u,u), B(u,u) + B(u,v), m, n_{\min})$
    $P_7 = \mathsf{strass}(A(u,v) - A(v,v), B(v,u) + B(v,v), m, n_{\min})$
    $C(u,u) = P_1 + P_4 - P_5 + P_7$
    $C(u,v) = P_3 + P_5$
    $C(v,u) = P_2 + P_4$
    $C(v,v) = P_1 + P_3 - P_2 + P_6$
  end

Unlike any of our previous algorithms, strass is recursive. Divide and conquer algorithms are often best described in this fashion. We have presented strass in the style of a MATLAB function so that the recursive calls can be stated with precision.

The amount of arithmetic associated with strass is a complicated function of n and $n_{\min}$. If $n_{\min} \gg 1$, then it suffices to count multiplications as the number of additions is roughly the same. If we just count the multiplications, then it suffices to examine the deepest level of the recursion as that is where all the multiplications occur. In strass there are $q - d$ subdivisions and thus 7^{q-d} conventional matrix-matrix multiplications to perform. These multiplications have size $n_{\min}$ and thus strass involves about $s = (2^d)^3 7^{q-d}$ multiplications compared to $c = (2^q)^3$, the number of multiplications in the conventional approach. Notice that

$$\frac{s}{c} = \left(\frac{2^d}{2^q}\right)^3 7^{q-d} = \left(\frac{7}{8}\right)^{q-d}.$$

If $d = 0$, i.e., we recur on down to the 1-by-1 level, then

$$s = (7/8)^q\, c = 7^q = n^{\log_2 7} \approx n^{2.807}.$$

Thus, asymptotically, the number of multiplications in Strassen's method is $O(n^{2.807})$. However, the number of additions (relative to the number of multiplications) becomes significant as $n_{\min}$ gets small.

Problems

P1.3.1 Rigorously prove the following block matrix equation:

$$\begin{bmatrix} A_{11} & \cdots & A_{1r} \\ \vdots & \ddots & \vdots \\ A_{q1} & \cdots & A_{qr} \end{bmatrix}^T = \begin{bmatrix} A_{11}^T & \cdots & A_{q1}^T \\ \vdots & \ddots & \vdots \\ A_{1r}^T & \cdots & A_{qr}^T \end{bmatrix}.$$

P1.3.2 Suppose $M \in \mathbb{R}^{n\times n}$ is Hamiltonian. How many flops are required to compute $N = M^2$?

P1.3.3 What can you say about the 2-by-2 block structure of a matrix $A \in \mathbb{R}^{2n\times 2n}$ that satisfies $\mathcal{E}_{2n}A\mathcal{E}_{2n} = A^T$ where $\mathcal{E}_{2n}$ is the exchange permutation defined in §1.2.11. Explain why A is symmetric about the "antidiagonal" that extends from the $(2n, 1)$ entry to the $(1, 2n)$ entry.

P1.3.4 Suppose

$$A = \begin{bmatrix} 0 & B \\ B^T & 0 \end{bmatrix}$$

where $B \in \mathbb{R}^{n\times n}$ is upper bidiagonal. Describe the structure of $T = PAP^T$ where $P = \mathcal{P}_{2,n}$ is the perfect shuffle permutation defined in §1.2.11.

P1.3.5 Show that if B *and* C are each permutation matrices, then $B \otimes C$ is also a permutation matrix.

P1.3.6 Verify Equation (1.3.5).

P1.3.7 Verify that if $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$, then $y \otimes x = \mathbf{vec}(xy^T)$.

P1.3.8 Show that if $B \in \mathbb{R}^{p\times p}$, $C \in \mathbb{R}^{q\times q}$, and

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_p \end{bmatrix} \qquad x_i \in \mathbb{R}^q,$$

then

$$x^T (B \otimes C)\, x = \sum_{i=1}^{p}\sum_{j=1}^{p} b_{ij}\left(x_i^T C x_j\right).$$

P1.3.9 Suppose $A^{(k)} \in \mathbb{R}^{n_k\times n_k}$ for $k = 1{:}r$ and that $x \in \mathbb{R}^n$ where $n = n_1 \cdots n_r$. Give an efficient algorithm for computing $y = \left(A^{(r)} \otimes \cdots \otimes A^{(2)} \otimes A^{(1)}\right) x$.

P1.3.10 Suppose n is even and define the following function from $\mathbb{R}^n$ to $\mathbb{R}$:

$$f(x) = x(1{:}2{:}n)^T x(2{:}2{:}n) = \sum_{i=1}^{n/2} x_{2i-1}x_{2i}.$$

(a) Show that if $x, y \in \mathbb{R}^n$ then

$$x^T y = \sum_{i=1}^{n/2}(x_{2i-1} + y_{2i})(x_{2i} + y_{2i-1}) - f(x) - f(y).$$

(b) Now consider the n-by-n matrix multiplication $C = AB$. Give an algorithm for computing this product that requires $n^3/2$ multiplies once f is applied to the rows of A and the columns of B. See Winograd (1968) for details.

P1.3.12 Adapt **strass** so that it can handle square matrix multiplication of any order. Hint: If the "current" A has odd dimension, append a zero row and column.

P1.3.13 Adapt **strass** so that it can handle nonsquare products, e.g., $C = AB$ where $A \in \mathbb{R}^{m\times r}$ and $B \in \mathbb{R}^{r\times n}$. Is it better to augment A and B with zeros so that they become square and equal in size or to "tile" A and B with square submatrices?

P1.3.14 Let W_n be the number of flops that **strass** requires to compute an n-by-n product where n is a power of 2. Note that $W_2 = 25$ and that for $n \geq 4$

$$W_n = 7W_{n/2} + 18(n/2)^2$$

Show that for every $\epsilon > 0$ there is a constant c_ϵ so $W_n \leq c_\epsilon n^{\omega+\epsilon}$ where $\omega = \log_2 7$ and n is any power of two.

P1.3.15 Suppose $B \in \mathbb{R}^{m_1 \times n_1}$, $C \in \mathbb{R}^{m_2 \times n_2}$, and $D \in \mathbb{R}^{m_3 \times n_3}$. Show how to compute the vector $y = (B \otimes C \otimes D)x$ where $x \in \mathbb{R}^n$ and $n = n_1 n_2 n_3$ is given. Is the order of operations important from the flop point of view?

Notes and References for §1.3

Useful references for the Kronecker product include Horn and Johnson (TMA, Chap. 4), Van Loan (FFT), and:

C.F. Van Loan (2000). "The Ubiquitous Kronecker Product," *J. Comput. Appl. Math., 123*, 85–100.

For quite some time fast methods for matrix multiplication have attracted a lot of attention within computer science, see:

S. Winograd (1968). "A New Algorithm for Inner Product," *IEEE Trans. Comput. C-17,* 693–694.
V. Strassen (1969). "Gaussian Elimination is not Optimal," *Numer. Math. 13,* 354–356.
V. Pan (1984). "How Can We Speed Up Matrix Multiplication?," *SIAM Review 26,* 393–416.
I. Kaporin (1999). "A Practical Algorithm for Faster Matrix Multiplication," *Num. Lin. Alg. 6*, 687–700.
H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans (2005). "Group-theoretic Algorithms for Matrix Multiplication," *Proceeedings of the 2005 Conference on the Foundations of Computer Science (FOCS)*, 379–388.
J. Demmel, I. Dumitriu, O. Holtz, and R. Kleinberg (2007). "Fast Matrix Multiplication is Stable," *Numer. Math. 106*, 199–224.
P. D'Alberto and A. Nicolau (2009). "Adaptive Winograd's Matrix Multiplication," *ACM Trans. Math. Softw. 36*, Article 3.

At first glance, many of these methods do not appear to have practical value. However, this has proven not to be the case, see:

D. Bailey (1988). "Extra High Speed Matrix Multiplication on the Cray-2," *SIAM J. Sci. Stat. Comput. 9,* 603–607.
N.J. Higham (1990). "Exploiting Fast Matrix Multiplication within the Level 3 BLAS," *ACM Trans. Math. Softw. 16*, 352–368.
C.C. Douglas, M. Heroux, G. Slishman, and R.M. Smith (1994). "GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm," *J. Comput. Phys. 110*, 1–10.

Strassen's algorithm marked the beginning of a search for the fastest possible matrix multiplication algorithm from the complexity point of view. The *exponent of matrix multiplication* is the smallest number ω such that, for all $\epsilon > 0$, $O(n^{\omega+\epsilon})$ work suffices. The best known value of ω has decreased over the years and is currently around 2.4. It is interesting to speculate on the existence of an $O(n^{2+\epsilon})$ procedure.

1.4 Fast Matrix-Vector Products

In this section we refine our ability to think at the block level by examining some matrix-vector products $y = Ax$ in which the n-by-n matrix A is so highly structured that the computation can be carried out with many fewer than the usual $O(n^2)$ flops. These results are used in §4.8.

1.4.1 The Fast Fourier Transform

The *discrete Fourier transform* (DFT) of a vector $x \in \mathbb{C}^n$ is a matrix-vector product

$$y = F_n x$$

where the *DFT matrix* $F_n = (f_{kj}) \in \mathbb{C}^{n\times n}$ is defined by

$$f_{kj} = \omega_n^{(k-1)(j-1)} \tag{1.4.1}$$

with

$$\omega_n = \exp(-2\pi i/n) \;=\; \cos(2\pi/n) - i\cdot\sin(2\pi/n). \tag{1.4.2}$$

Here is an example:

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ 1 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}.$$

The DFT is ubiquitous throughout computational science and engineering and one reason has to do with the following property:

> If n is highly composite, then it is possible to carry out the DFT in many fewer than the $O(n^2)$ flops required by conventional matrix-vector multiplication.

To illustrate this we set $n = 2^t$ and proceed to develop the *radix-2 fast Fourier transform.*

The starting point is to examine the block structure of an even-order DFT matrix after its columns are reordered so that the odd-indexed columns come first. Consider the case

$$F_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{bmatrix} \qquad (\omega = \omega_8).$$

(Note that ω_8 is a root of unity so that high powers simplify, e.g., $[F_8]_{4,7} = \omega^{3\cdot 6} = \omega^{18} = \omega^2$.) If $cols = [1\ 3\ 5\ 7\ 2\ 4\ 6\ 8]$, then

$$F_8(:, cols) = \left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega & \omega^3 & \omega^5 & \omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega & \omega^7 & \omega^5 \\ \hline 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & -\omega & -\omega^3 & -\omega^5 & -\omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & -\omega^2 & -\omega^6 & -\omega^2 & -\omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & -\omega^3 & -\omega & -\omega^7 & -\omega^5 \end{array}\right].$$

The lines through the matrix are there to help us think of $F_8(:, cols)$ as a 2-by-2 matrix with 4-by-4 blocks. Noting that $\omega^2 = \omega_8^2 = \omega_4$, we see that

$$F_8(:, cols) \;=\; \left[\begin{array}{c|c} F_4 & \Omega_4 F_4 \\ \hline F_4 & -\Omega_4 F_4 \end{array}\right]$$

where $\Omega_4 = \mathrm{diag}(1, \omega_8, \omega_8^2, \omega_8^3)$. It follows that if $x \in \mathbb{R}^8$, then

$$F_8 x = F_8(:, cols) \cdot x(cols) = \left[\begin{array}{c|c} F_4 & \Omega_4 F_4 \\ \hline F_4 & -\Omega_4 F_4 \end{array}\right] \left[\begin{array}{c} x(1{:}2{:}8) \\ \hline x(2{:}2{:}8) \end{array}\right] = \left[\begin{array}{c|c} I_4 & \Omega_4 \\ \hline I_4 & -\Omega_4 \end{array}\right] \left[\begin{array}{c} F_4 x(1{:}2{:}8) \\ \hline F_4 x(2{:}2{:}8) \end{array}\right].$$

Thus, by simple scalings we can obtain the 8-point DFT $y = F_8 x$ from the 4-point DFTs $y_T = F_4 \cdot x(1{:}2{:}8)$ and $y_B = F_4 \cdot x(2{:}2{:}8)$. In particular,

$$\begin{aligned} y(1{:}4) &= y_T + d \mathbin{.*} y_B, \\ y(5{:}8) &= y_T - d \mathbin{.*} y_B \end{aligned}$$

where

$$d = \begin{bmatrix} 1 \\ \omega \\ \omega^2 \\ \omega^3 \end{bmatrix}.$$

More generally, if $n = 2m$, then $y = F_n x$ is given by

$$\begin{aligned} y(1{:}m) &= y_T + d \mathbin{.*} y_B, \\ y(m+1{:}n) &= y_T - d \mathbin{.*} y_B \end{aligned}$$

where $d = \begin{bmatrix} 1, & \omega_n, \ldots, & \omega_n^{m-1} \end{bmatrix}^T$ and

$$\begin{aligned} y_T &= F_m x(1{:}2{:}n), \\ y_B &= F_m x(2{:}2{:}n). \end{aligned}$$

For $n = 2^t$, we can recur on this process until $n = 1$, noting that $F_1 x = x$.

Algorithm 1.4.1 If $x \in \mathbb{C}^n$ and $n = 2^t$, then this algorithm computes the discrete Fourier transform $y = F_n x$.

function $y = \mathsf{fft}(x, n)$
  if $n = 1$
    $y = x$
  else
    $m = n/2$
    $y_T = \mathsf{fft}(x(1{:}2{:}n), m)$
    $y_B = \mathsf{fft}(x(2{:}2{:}n), m)$
    $\omega = \exp(-2\pi i/n)$
    $d = \begin{bmatrix} 1, & \omega, \cdots, & \omega^{m-1} \end{bmatrix}^T$
    $z = d \mathbin{.*} y_B$
    $y = \begin{bmatrix} y_T + z \\ y_T - z \end{bmatrix}$
  end

The flop analysis of fft requires an assessment of complex arithmetic and the solution of an interesting recursion. We first observe that the multiplication of two complex numbers involves six (real) flops while the addition of two complex numbers involves two flops. Let f_n be the number of flops that fft needs to produce the DFT of $x \in \mathbb{C}^n$. Scrutiny of the method reveals that

$$\left\{\begin{array}{c} y_T \\ y_B \\ d \\ z \\ y \end{array}\right\} \text{ requires } \left\{\begin{array}{l} f_m \text{ flops} \\ f_m \text{ flops} \\ 6m \text{ flops} \\ 6m \text{ flops} \\ 2n \text{ flops} \end{array}\right\}$$

where $n = 2m$. Thus,

$$f_n = 2f_m + 8n \qquad (f_1 = 0).$$

Conjecturing that $f_n = c{\cdot}n\log_2(n)$ for some constant c, it follows that

$$f_n = c{\cdot}n\log_2(n) = 2c{\cdot}m\log_2(m) + 8n = c{\cdot}n(\log_2(n) - 1) + 8n,$$

from which we conclude that $c = 8$. Thus, fft requires $8n\log_2(n)$ flops. Appreciate the speedup over conventional matrix-vector multiplication. If $n = 2^{20}$, it is a factor of about 10,000. We mention that the fft flop count can be reduced to $5n\log_2(n)$ by precomputing $\omega_n, \ldots, \omega_n^{n/2-1}$. See P1.4.1.

1.4.2 Fast Sine and Cosine Transformations

In the *discrete sine transform* (DST) problem, we are given real values $x_1, \ldots, x_{m-1}$ and compute

$$y_k = \sum_{j=1}^{m-1} \sin\left(\frac{kj\pi}{m}\right) x_j \tag{1.4.3}$$

for $k = 1{:}m-1$. In the *discrete cosine transform* (DCT) problem, we are given real values $x_0, x_1, \ldots, x_m$ and compute

$$y_k = \frac{x_0}{2} + \sum_{j=1}^{m-1} \cos\left(\frac{kj\pi}{m}\right) x_j + \frac{(-1)^k x_m}{2} \tag{1.4.4}$$

for $k = 0{:}m$. Note that the sine and cosine evaluations "show up" in the DFT matrix. Indeed, for $k = 0{:}2m-1$ and $j = 0{:}2m-1$ we have

$$[F_{2m}]_{k+1,j+1} = \omega_{2m}^{kj} = \cos\left(\frac{kj\pi}{m}\right) - i\sin\left(\frac{kj\pi}{m}\right). \tag{1.4.5}$$

This suggests (correctly) that there is an exploitable connection between each of these trigonometric transforms and the DFT. The key observation is to block properly the real and imaginary parts of F_{2m}. To that end, define the matrices $S_r \in \mathbb{R}^{r\times r}$ and $C_r \in \mathbb{R}^{r\times r}$ by

$$\begin{aligned} [S_r]_{kj} &= \sin\left(\frac{kj\pi}{r+1}\right), \\ [C_r]_{kj} &= \cos\left(\frac{kj\pi}{r+1}\right), \end{aligned} \qquad k = 1{:}r,\ j = 1{:}r. \tag{1.4.6}$$

Recalling from §1.2.11 the definition of the exchange permutation $\mathcal{E}_n$, we have

Theorem 1.4.1. *Let m be a positive integer and define the vectors e and v by*

$$e^T = (\underbrace{1, 1, \ldots, 1}_{m-1}), \qquad v^T = (\underbrace{-1, 1, \ldots, (-1)^{m-1}}_{m-1}).$$

If $E = \mathcal{E}_{m-1}$, $C = C_{m-1}$, and $S = S_{m-1}$, then

$$F_{2m} = \begin{bmatrix} 1 & e^T & 1 & e^T \\ e & C - iS & v & (C + iS)E \\ 1 & v^T & (-1)^m & v^T E \\ e & E(C + iS) & Ev & E(C - iS)E \end{bmatrix}. \tag{1.4.7}$$

Proof. It is clear from (1.4.5) that $F_{2m}(:, 1)$, $F_{2m}(1, :1)$, $F_{2m}(:, m{+}1)$, and $F_{2m}(m{+}1, :)$ are correctly specified. It remains for us to show that equation (1.4.7) holds in blocks positions (2,2), (2,4), (4,2), and (4,4). The (2,2) verification is straightforward:

$$\begin{aligned} [F_{2m}(2{:}m, 2{:}m)]_{kj} &= \cos\left(\frac{kj\pi}{m}\right) - i \sin\left(\frac{kj\pi}{m}\right) \\ &= [C - iS]_{kj}. \end{aligned}$$

A little trigonometry is required to verify correctness in the (2,4) position:

$$\begin{aligned} [F_{2m}(2{:}m, m+2{:}2m)]_{kj} &= \cos\left(\frac{k(m+j)\pi}{m}\right) - i \sin\left(\frac{k(m+j)\pi}{m}\right) \\ &= \cos\left(\frac{kj\pi}{m} + k\pi\right) - i \sin\left(\frac{kj\pi}{m} + k\pi\right) \\ &= \cos\left(-\frac{kj\pi}{m} + k\pi\right) + i \sin\left(-\frac{kj\pi}{m} + k\pi\right) \\ &= \cos\left(\frac{(k(m-j)\pi}{m}\right) + i \sin\left(\frac{k(m-j)\pi}{m}\right) \\ &= [(C + iS)E]_{kj}. \end{aligned}$$

We used the fact that post-multiplying a matrix by the permutation $E = \mathcal{E}_{m-1}$ has the effect of reversing the order of its columns. The recipes for $F_{2m}(m + 2{:}2m, 2{:}m)$ and $F_{2m}(m + 2{:}2m, m + 2{:}2m)$ are derived similarly. □

Using the notation of the theorem, we see that the sine transform (1.4.3) is a matrix-vector product

$$y(1{:}m{-}1) = \mathrm{DST}(m{-}1) \cdot x(1{:}m{-}1)$$

where

$$\mathrm{DST}(m-1) \;=\; S_{m-1}. \tag{1.4.8}$$

If $x = x(1{:}m-1)$ and

$$x_{\sin} \;=\; \begin{bmatrix} 0 \\ x \\ 0 \\ -Ex \end{bmatrix} \in \mathbb{R}^{2m}, \tag{1.4.9}$$

then since $e^TE = e$ and $E^2 = E$ we have

$$\frac{i}{2}F_{2m}x_{\sin} = \frac{i}{2}\begin{bmatrix} 1 & e^T & 1 & e^T \\ e & C-iS & v & (C+iS)E \\ 1 & v^T & (-1)^m & v^TE \\ e & E(C+iS) & Ev & E(C-iS)E \end{bmatrix}\begin{bmatrix} 0 \\ x \\ 0 \\ -Ex \end{bmatrix}$$

$$= \frac{i}{2}\begin{bmatrix} e^Tx - e^TEx \\ -2iSx \\ v^Tx - v^TE^2x \\ i(ESx + ESE^2x) \end{bmatrix} = \begin{bmatrix} 0 \\ Sx \\ 0 \\ -ESx \end{bmatrix}.$$

Thus, the DST of $x(1{:}m-1)$ is a scaled subvector of $F_{2m}x_{\sin}$.

Algorithm 1.4.2 The following algorithm assigns the DST of $x_1, \ldots, x_{m-1}$ to y.

Set up the vector $x_{\sin}$ defined by (1.4.9).

Use fft (e.g., Algorithm 1.4.1) to compute $\tilde{y} = F_{2m}x_{\sin}$

$y \;=\; i \cdot \tilde{y}(2{:}m)/2$

This computation involves $O(m\log_2(m))$ flops. We mention that the vector $x_{\sin}$ is real and highly structured, something that would be exploited in a truly efficient implementation.

Now let us consider the discrete cosine transform defined by (1.4.4). Using the notation from Theorem 1.4.1, the DCT is a matrix-vector product

$$y(0{:}m) \;=\; \mathrm{DCT}(m+1)\cdot x(0{:}m)$$

where

$$\mathrm{DCT}(m+1) \;=\; \begin{bmatrix} 1/2 & e^T & 1/2 \\ e/2 & C_{m-1} & v/2 \\ 1/2 & v^T & (-1)^m/2 \end{bmatrix} \tag{1.4.10}$$

If $\tilde{x} = x(1{:}m-1)$ and

$$x_{\cos} \;=\; \begin{bmatrix} x_0 \\ \tilde{x} \\ x_m \\ E\tilde{x} \end{bmatrix} \in \mathbb{R}^{2m}, \tag{1.4.11}$$

then

$$\frac{1}{2}F_{2m}x_{\cos} = \frac{1}{2}\begin{bmatrix} 1 & e^T & 1 & e^T \\ e & C-iS & v & (C+iS)E \\ 1 & v^T & (-1)^m & v^TE \\ e & E(C+iS) & Ev & E(C-iS)E \end{bmatrix}\begin{bmatrix} x_0 \\ \tilde{x} \\ x_m \\ E\tilde{x} \end{bmatrix}$$

$$= \begin{bmatrix} (x_0/2) & + & e^T\tilde{x} & + & (x_m/2) \\ (x_0/2)e & + & C\tilde{x} & + & (x_m/2)v \\ (x_0/2) & + & v^T\tilde{x} & + & (-1)^m(x_m/2) \\ (x_0/2)e & + & EC\tilde{x} & + & (x_m/2)Ev \end{bmatrix}.$$

Notice that the top three components of this block vector define the DCT of $x(0{:}m)$. Thus, the DCT is a scaled subvector of $F_{2m}x_{\cos}$.

Algorithm 1.4.3 The following algorithm assigns to $y \in \mathbb{R}^{m+1}$ the DCT of $x_0, \ldots, x_m$.

Set up the vector $x_{\cos} \in \mathbb{R}^{2m}$ defined by (1.4.11).

Use fft (e.g., Algorithm 1.4.1) to compute $\tilde{y} = F_{2m}x_{\cos}$

$y = \tilde{y}(1{:}m+1)/2$

This algorithm requires $O(m \log m)$ flops, but as with Algorithm 1.4.2, it can be more efficiently implemented by exploiting symmetries in the vector x_{cos}.

We mention that there are important variants of the DST and the DCT that can be computed fast:

$$\begin{aligned}
&\text{DST-II:} && y_k = \sum_{j=1}^{m} \sin\left(\frac{k(2j-1)\pi}{2m}\right)x_j, && k = 1{:}m, \\
&\text{DST-III:} && y_k = \sum_{j=1}^{m} \sin\left(\frac{(2k-1)j\pi}{2m}\right)x_j, && k = 1{:}m, \\
&\text{DST-IV:} && y_k = \sum_{j=1}^{m} \sin\left(\frac{(2k-1)(2j-1)\pi}{2m}\right)x_j, && k = 1{:}m, \\
&\text{DCT-II:} && y_k = \sum_{j=0}^{m-1} \cos\left(\frac{k(2j-1)\pi}{2m}\right)x_j, && k = 0{:}m-1, \\
&\text{DCT-III:} && y_k = \frac{x_0}{2} = \sum_{j=1}^{m-1} \cos\left(\frac{(2k-1)j\pi}{2m}\right)x_j, && k = 0{:}m-1, \\
&\text{DCT-IV:} && y_k = \sum_{j=0}^{m-1} \cos\left(\frac{(2k-1)(2j-1)\pi}{2m}\right)x_j, && k = 0{:}m-1.
\end{aligned} \tag{1.4.12}$$

For example, if $\tilde{y} \in \mathbb{R}^{2m-1}$ is the DST of $\tilde{x} = [\, x_1,\ 0,\ x_2,\ 0, \ldots, 0,\ x_{m-1},\ x_m \,]^T$, then $\tilde{y}(1{:}m)$ is the DST-II of $x \in \mathbb{R}^m$. See Van Loan (FFT) for further details.

1.4.3 The Haar Wavelet Transform

If $n = 2^t$, then the *Haar wavelet transform* $y = W_n x$ is a matrix-vector product in which the transform matrix $W_n \in \mathbb{R}^{n\times n}$ is defined recursively:

$$W_n = \begin{cases} \left[\begin{array}{c|c} W_m \otimes \begin{pmatrix} 1 \\ 1 \end{pmatrix} & I_m \otimes \begin{pmatrix} 1 \\ -1 \end{pmatrix} \end{array}\right] & \text{if } n = 2m, \\ [\,1\,] & \text{if } n = 1. \end{cases}$$

Here are some examples:

$$W_2 = \left[\begin{array}{c|c} 1 & 1 \\ \hline 1 & -1 \end{array}\right],$$

$$W_4 = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 0 \\ \hline 1 & -1 & 0 & 1 \\ 1 & -1 & 0 & -1 \end{array}\right],$$

$$W_8 = \left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & -1 & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 & -1 & 0 & 0 \\ \hline 1 & -1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & -1 & 0 \\ 1 & -1 & 0 & -1 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & -1 & 0 & 0 & 0 & -1 \end{array}\right].$$

An interesting block pattern emerges if we reorder the rows of W_n so that the odd-indexed rows come first:

$$\mathcal{P}_{2,m}^T W_n = \begin{bmatrix} W_m & I_m \\ W_m & -I_m \end{bmatrix} = (W_2 \otimes I_m) \begin{bmatrix} W_m & 0 \\ 0 & I_m \end{bmatrix}. \tag{1.4.13}$$

Thus, if $x \in \mathbb{R}^n$, $x_T = x(1{:}m)$, and $x_B = x(m+1{:}n)$, then

$$y = W_n x = \mathcal{P}_{2,m} \begin{bmatrix} I_m & I_m \\ I_m & -I_m \end{bmatrix} \begin{bmatrix} W_m & 0 \\ 0 & I_m \end{bmatrix} \begin{bmatrix} x_T \\ x_B \end{bmatrix}$$

$$= \mathcal{P}_{2,m} \begin{bmatrix} W_m x_T + x_B \\ W_m x_T - x_B \end{bmatrix}.$$

In other words,

$$y(1{:}2{:}n) = W_m x_T + x_B, \qquad y(2{:}2{:}n) = W_m x_T - x_B.$$

This points the way to a fast recursive procedure for computing $y = W_n x$.

Algorithm 1.4.4 (Haar Wavelet Transform) If $x \in \mathbb{R}^n$ and $n = 2^t$, then this algorithm computes the Haar transform $y = W_n x$.

```
function y = fht(x, n)
    if n = 1
        y = x
    else
        m = n/2
        z = fht(x(1:m), m)
        y(1:2:m) = z + x(m+1:n)
        y(2:2:m) = z - x(m+1:n)
    end
```

It can be shown that this algorithm requires $2n$ flops.

Problems

P1.4.1 Suppose $w = \begin{bmatrix} 1, & \omega_n, & \omega_n^2, \ldots, & \omega_n^{n/2-1} \end{bmatrix}$ where $n = 2^t$. Using the colon notation, express

$$\begin{bmatrix} 1, & \omega_r, & \omega_r^2, \ldots, & \omega_r^{r/2-1} \end{bmatrix}$$

as a subvector of w where $r = 2^q$, $q = 1{:}t$. Rewrite Algorithm 1.4.1 with the assumption that w is precomputed. Show that this maneuver reduces the flop count to $5n \log_2 n$.

P1.4.2 Suppose $n = 3m$ and examine

$$G = [\, F_n(:, 1{:}3{:}n-1) \mid F_n(:, 2{:}3{:}n-1) \mid F_n(:, 3{:}3{:}n-1) \,]$$

as a 3-by-3 block matrix, looking for scaled copies of F_m. Based on what you find, develop a recursive radix-3 FFT analogous to the radix-2 implementation in the text.

P1.4.3 If $n = 2^t$, then it can be shown that $F_n = (A_t \Gamma_t) \cdots (A_1 \Gamma_1)$ where for $q = 1{:}t$

$$L_q = 2^q, \quad r_q = n/L_q,$$

$$A_q = I_{r_q} \otimes \begin{bmatrix} I_{L_{q-1}} & \Omega_q \\ I_{L_{q-1}} & -\Omega_q \end{bmatrix},$$

$$\Gamma_q = \mathcal{P}_{2,r_q} \otimes I_{L_{q-1}},$$

$$\Omega_q = \mathrm{diag}(1, \omega_{L_q}, \ldots, \omega_{L_q}^{L_{q-1}-1}).$$

Note that with this factorization, the DFT $y = F_n x$ can be computed as follows:

```
y = x
for q = 1:t
    y = A_q(Γ_q y)
end
```

Fill in the details associated with the y updates and show that a careful implementation requires $5n \log_2(n)$ flops.

P1.4.4 What fraction of the components of W_n are zero?

P1.4.5 Using (1.4.13), verify by induction that if $n = 2^t$, then the Haar tranform matrix W_n has the factorization $W_n = H_t \cdots H_1$ where

$$H_q = \begin{bmatrix} \mathcal{P}_{2,L_*} & 0 \\ 0 & I_{n-L} \end{bmatrix} \begin{bmatrix} W_2 \otimes I_{L_*} & 0 \\ 0 & I_{n-L} \end{bmatrix} \qquad L = 2^q,\ L_* = L/2.$$

Thus, the computation of $y = W_n x$ may proceed as follows:

```
y = x
for q = 1:t
    y = H_q y
end
```

Fill in the details associated with the update $y = H_q y$ and confirm that $W_n x$ costs $2n$ flops.

P1.4.6 Using (1.4.13), develop an $O(n)$ procedure for solving $W_n y = x$ where $x \in \mathbb{R}^n$ is given and $n = 2^t$.

Notes and References for §1.4

In Van Loan (FFT) the FFT family of algorithms is described in the language of matrix-factorizations. A discussion of various fast trigonometric transforms is also included. See also:

W.L. Briggs and V.E. Henson (1995). *The DFT: An Owners' Manual for the Discrete Fourier Transform*, SIAM Publications, Philadelphia, PA.

The design of a high-performance FFT is a nontrivial task. An important development in this regard is a software tool known as "the fastest Fourier transform in the west":

M. Frigo and S.G. Johnson (2005). "The Design and Implementation of FFTW3", *Proceedings of the IEEE, 93*, 216–231.

It automates the search for the "right" FFT given the underlying computer architecture. FFT references that feature interesting factorization and approximation ideas include:

A. Edelman, P. McCorquodale, and S. Toledo (1998). "The Future Fast Fourier Transform?," *SIAM J. Sci. Comput. 20*, 1094–1114.
A. Dutt and and V. Rokhlin (1993). "Fast Fourier Transforms for Nonequally Spaced Data," *SIAM J. Sci. Comput. 14,* 1368–1393.
A. F. Ware (1998). "Fast Approximate Fourier Transforms for Irregularly Spaced Data," *SIAM Review 40*, 838 –856.
N. Nguyen and Q.H. Liu (1999). "The Regular Fourier Matrices and Nonuniform Fast Fourier Transforms," *SIAM J. Sci. Comput. 21*, 283–293.
A. Nieslony and G. Steidl (2003). "Approximate Factorizations of Fourier Matrices with Nonequispaced Knots," *Lin. Alg. Applic. 366,* 337–351.
L. Greengard and J.–Y. Lee (2004). "Accelerating the Nonuniform Fast Fourier Transform," *SIAM Review 46*, 443–454.
K. Ahlander and H. Munthe-Kaas (2005). "Applications of the Generalized Fourier Transform in Numerical Linear Algebra," *BIT 45*, 819–850.

The fast multipole method and the fast Gauss transform represent another type of fast transform that is based on a combination of clever blocking and approximation.

L. Greengard and V. Rokhlin (1987). "A Fast Algorithm for Particle Simulation," *J. Comput. Phys. 73*, 325–348.
X. Sun and N.P. Pitsianis (2001). "A Matrix Version of the Fast Multipole Method," *SIAM Review 43*, 289–300.
L. Greengard and J. Strain (1991). "The Fast Gauss Transform," *SIAM J. Sci. Stat. Comput. 12*, 79–94.
M. Spivak, S.K. Veerapaneni, and L. Greengard (2010). "The Fast Generalized Gauss Transform," *SIAM J. Sci. Comput. 32*, 3092–3107.
X. Sun and Y. Bao (2003). "A Kronecker Product Representation of the Fast Gauss Transform," *SIAM J. Matrix Anal. Applic. 24*, 768–786.

The Haar transform is a simple example of a wavelet transform. The wavelet idea has had a profound impact throughout computational science and engineering. In many applications, wavelet basis functions work better than the sines and cosines that underly the DFT. Excellent monographs on this subject include

I Daubechies (1992). *Ten Lectures on Wavelets,* SIAM Publications, Philadelphia, PA.
G. Strang (1993). "Wavelet Transforms Versus Fourier Transforms," *Bull. AMS 28,* 288–305.
G. Strang and T. Nguyan (1996). *Wavelets and Filter Banks*, Wellesley-Cambridge Press.

1.5 Vectorization and Locality

When it comes to designing a high-performance matrix computation, it is not enough simply to minimize flops. Attention must be paid to how the arithmetic units interact with the underlying memory system. Data structures are an important part of the picture because not all matrix layouts are "architecture friendly." Our aim is to build a practical appreciation for these issues by presenting various simplified models of execution. These models are *qualitative* and are just informative pointers to complex implementation issues.

1.5.1 Vector Processing

An individual floating point operation typically requires several cycles to complete. A 3-cycle addition is depicted in Figure 1.5.1. The input scalars x and y proceed along

Figure 1.5.1. *A 3-Cycle adder*

a computational "assembly line," spending one cycle at each of three work "stations." The sum z emerges after three cycles. Note that, during the execution of a single, "free standing" addition, only one of the three stations would be active at any particular instant.

Vector processors exploit the fact that a vector operation is a very regular sequence of scalar operations. The key idea is *pipelining*, which we illustrate using the vector addition computation $z = x + y$. With pipelining, the x and y vectors are streamed through the addition unit. Once the pipeline is filled and steady state reached, a z-vector component is produced every cycle, as shown in Figure 1.5.2. In

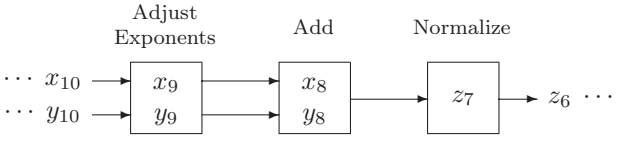

Figure 1.5.2. *Pipelined addition*

this case, we would anticipate vector processing to proceed at about three times the rate of scalar processing.

A vector processor comes with a repertoire of *vector instructions*, such as vector add, vector multiply, vector scale, dot product, and saxpy. These operations take place in *vector registers* with input and output handled by *vector load* and *vector store* instructions. An important attribute of a vector processor is the length v_L of the vector registers that carry out the vector operations. A length-n vector operation must be broken down into subvector operations of length v_L or less. Here is how such a partitioning might be managed for a vector addition $z = x + y$ where x and y are n-vectors:

$$
\begin{array}{l}
first = 1 \\
\textbf{while } first \leq n \\
\quad last = \min\{n, first + v_L - 1\} \\
\quad \text{Vector load: } r_1 \leftarrow x(first{:}last) \\
\quad \text{Vector load: } r_2 \leftarrow y(first{:}last) \\
\quad \text{Vector add: } r_1 = r_1 + r_2 \\
\quad \text{Vector store: } z(first{:}last) \leftarrow r_1 \\
\quad first = last + 1 \\
\textbf{end}
\end{array}
\tag{1.5.1}
$$

The vector addition is a register-register operation while the "flopless" movement of data to and from the vector registers is identified with the left arrow "$\leftarrow$". Let us model the number of cycles required to carry out the various steps in (1.5.1). For clarity, assume that n is very large and an integral multiple of v_L, thereby making it safe to ignore the final cleanup pass through the loop.

Regarding the vectorized addition $r_1 = r_1 + r_2$, assume it takes τ_{add} cycles to fill the pipeline and that once this happens, a component of z is produced each cycle. It follows that

$$
N_{\text{arith}} = \left(\frac{n}{v_L}\right)(\tau_{\text{add}} + v_L) = \left(\frac{\tau_{\text{add}}}{v_L} + 1\right) n
$$

accounts for the total number cycles that (1.5.1) requires for arithmetic.

For the vector loads and stores, assume that $\tau_{\text{data}} + v_L$ cycles are required to transport a length-v_L vector from memory to a register or from a register to memory, where τ_{data} is the number of cycles required to fill the data pipeline. With these assumptions we see that

$$
N_{\text{data}} = 3\left(\frac{n}{v_L}\right)(\tau_{\text{data}} + v_L) = 3\left(\frac{\tau_{\text{add}}}{v_L} + 1\right) n
$$

specifies the number of cycles that are required by (1.5.1) to get data to and from the registers.

The arithmetic-to-data-motion ratio

$$
N_{\text{arith}}/N_{\text{data}} = \frac{\tau_{\text{add}} + v_L}{3(\tau_{\text{data}} + v_L)}
$$

and the total cycles sum

$$
N_{\text{arith}} + N_{\text{data}} = \left(\frac{\tau_{\text{arith}} + 3\tau_{\text{data}}}{v_L} + 4\right) n
$$

are illuminating statistics, but they are not necessarily good predictors of performance. In practice, vector loads, stores, and arithmetic are "overlapped" through the chaining together of various pipelines, a feature that is not captured by our model. Nevertheless, our simple analysis is a preliminary reminder that data motion is an important factor when reasoning about performance.

1.5.2 Gaxpy versus Outer Product

Two algorithms that involve the same number of flops can have substantially different data motion properties. Consider the n-by-n gaxpy

$$y = y + Ax$$

and the n-by-n outer product update

$$A = A + yx^T.$$

Both of these level-2 operations involve $2n^2$ flops. However, if we assume (for clarity) that $n = v_L$, then we see that the gaxpy computation

$r_x \leftarrow x$
$r_y \leftarrow y$
for $j = 1{:}n$
  $r_a \leftarrow A(:,j)$
  $r_y = r_y + r_a r_x(j)$
end
$y \leftarrow r_y$

requires $(3+n)$ load/store operations while for the outer product update

$r_x \leftarrow x$
$r_y \leftarrow y$
for $j = 1{:}n$
  $r_a \leftarrow A(:,j)$
  $r_a = r_a + r_y r_x(j)$
  $A(:,j) \leftarrow r_a$
end

the corresponding count is $(2+2n)$. Thus, the data motion overhead for the outer product update is worse by a factor of 2, a reality that could be a factor in the design of a high-performance matrix computation.

1.5.3 The Relevance of Stride

The time it takes to load a vector into a vector register may depend greatly on how the vector is laid out in memory, a detail that we did not consider in §1.5.1. Two concepts help frame the issue. A vector is said to have *unit stride* if its components are contiguous in memory. A matrix is said to be stored in *column-major order* if its columns have unit stride.

Let us consider the matrix multiplication update calculation

$$C = C + AB$$

where it is assumed that the matrices $C \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{m\times r}$, and $B \in \mathbb{R}^{r\times n}$ are stored in column-major order. Suppose the loading of a unit-stride vector proceeds much more quickly than the loading of a non-unit-stride vector. If so, then the implementation

```
for j = 1:n
    for k = 1:r
        C(:,j) = C(:,j) + A(:,k)·B(k,j)
    end
end
```

which accesses C, A, and B by column would be preferred to

```
for i = 1:m
    for j = 1:n
        C(i,j) = C(i,j) + A(i,:)·B(:,j)
    end
end
```

which accesses C and A by row. While this example points to the possible importance of stride, it is important to keep in mind that the penalty for non-unit-stride access varies from system to system and may depend upon the value of the stride itself.

1.5.4 Blocking for Data Reuse

Matrices reside in memory but *memory has levels*. A typical arrangement is depicted in Figure 1.5.3. The *cache* is a relatively small high-speed memory unit that sits

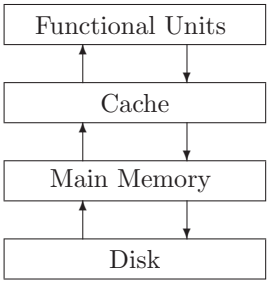

Figure 1.5.3. *A memory hierarchy*

just below the functional units where the arithmetic is carried out. During a matrix computation, matrix elements move up and down the memory hierarchy. The *cache*, which is a small high-speed memory situated in between the functional units and main memory, plays a particularly critical role. The overall design of the hierarchy varies from system to system. However, two maxims always apply:

- Each level in the hierarchy has a limited capacity and for economic reasons this capacity usually becomes smaller as we ascend the hierarchy.
- There is a cost, sometimes relatively great, associated with the moving of data between two levels in the hierarchy.

The efficient implementation of a matrix algorithm requires an ability to reason about the flow of data between the various levels of storage.

To develop an appreciation for cache utilization we again consider the update $C = C + AB$ where each matrix is n-by-n and blocked as follows:

$$C = \begin{bmatrix} C_{11} & \cdots & C_{1r} \\ \vdots & \ddots & \vdots \\ C_{qr} & \cdots & C_{qr} \end{bmatrix} \quad A = \begin{bmatrix} A_{11} & \cdots & A_{1p} \\ \vdots & \ddots & \vdots \\ A_{qr} & \cdots & A_{qp} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & \cdots & B_{1r} \\ \vdots & \ddots & \vdots \\ B_{pr} & \cdots & B_{pr} \end{bmatrix}.$$

Assume that these three matrices reside in main memory and that we plan to update C block by block:

$$C_{ij} = C_{ij} + \sum_{k=1}^{p} A_{ik}B_{kj}.$$

The data in the blocks must be brought up to the functional units via the cache which we assume is large enough to hold a C-block, an A-block, and a B-block. This enables us to structure the computation as follows:

```
for i = 1:q
    for j = 1:r
        Load C_ij from main memory into cache
        for k = 1:p
            Load A_ik from main memory into cache
            Load B_kj from main memory into cache
            C_ij = C_ij + A_ik B_kj
        end
        Store C_ij in main memory.
    end
end
```

(1.5.4)

The question before us is how to choose the blocking parameters q, r, and p so as to minimize memory traffic to and from the cache. Assume that the cache can hold M floating point numbers and that $M \ll 3n^2$, thereby forcing us to block the computation. We assume that

$$\left.\begin{matrix} C_{ij} \\ A_{ik} \\ B_{kj} \end{matrix}\right\} \text{ is roughly } \left\{\begin{matrix} (n/q)\text{-by-}(n/r) \\ (n/q)\text{-by-}(n/p) \\ (n/p)\text{-by-}(n/r) \end{matrix}\right. .$$

We say "roughly" because if q, r, or p does not divide n, then the blocks are not quite uniformly sized, e.g.,

$$A = \left[\begin{array}{ccc|ccc|ccc|c} \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \hline \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \hline \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \end{array}\right], \qquad \begin{matrix} n = 10, \\ q = 3, \\ p = 4. \end{matrix}$$

However, nothing is lost in glossing over this detail since our aim is simply to develop an intuition about cache utilization for large-n problems. Thus, we are led to impose the following constraint on the blocking parameters:

$$\left(\frac{n}{q}\right)\left(\frac{n}{r}\right) + \left(\frac{n}{q}\right)\left(\frac{n}{p}\right) + \left(\frac{n}{p}\right)\left(\frac{n}{r}\right) \leq M. \tag{1.5.5}$$

Proceeding with the optimization, it is reasonable to *maximize* the amount of arithmetic associated with the update $C_{ij} = C_{ij} + A_{ik}B_{kj}$. After all, we have moved matrix data from main memory to cache and should make the most of the investment. This leads to the problem of maximizing $2n^3/(qrp)$ subject to the constraint (1.5.5). A straightforward Lagrange multiplier argument leads us to conclude that

$$q_{\text{opt}} = p_{\text{opt}} = r_{\text{opt}} \approx \sqrt{\frac{n^2}{3M}}. \tag{1.5.6}$$

That is, each block of C, A, and B should be approximately square and occupy about one-third of the cache.

Because blocking affects the amount of memory traffic in a matrix computation, it is of paramount importance when designing a high-performance implementation. In practice, things are never as simple as in our model example. The optimal choice of q_{opt}, r_{opt}, and p_{opt} will also depend upon transfer rates between memory levels and upon all the other architecture factors mentioned earlier in this section. Data structures are also important; storing a matrix by block rather than in column-major order could enhance performance.

Problems

P1.5.1 Suppose $A \in \mathbb{R}^{n\times n}$ is tridiagonal and that the elements along its subdiagonal, diagonal, and superdiagonal are stored in vectors $e(1{:}n-1)$, $d(1{:}n)$, and $f(2{:}n)$. Give a vectorized implementation of the n-by-n gaxpy $y = y + Ax$. Hint: Make use of the vector multiplication operation.

P1.5.2 Give an algorithm for computing $C = C + A^TBA$ where A and B are n-by-n and B is symmetric. Innermost loops should oversee unit-stride vector operations.

P1.5.3 Suppose $A \in \mathbb{R}^{m\times n}$ is stored in column-major order and that $m = m_1M$ and $n = n_1N$. Regard A as an M-by-N block matrix with m_1-by-n_1 blocks. Give an algorithm for storing A in a vector $A.block(1{:}mn)$ with the property that each block A_{ij} is stored contiguously in column-major order.

Notes and References for §1.5

References that address vector computation include:

J.J. Dongarra, F.G. Gustavson, and A. Karp (1984). "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Review 26,* 91–112.
B.L. Buzbee (1986) "A Strategy for Vectorization," *Parallel Comput. 3,* 187–192.
K. Gallivan, W. Jalby, U. Meier, and A.H. Sameh (1988). "Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design," *Int. J. Supercomput. Applic. 2,* 12–48.
J.J. Dongarra and D. Walker (1995). "Software Libraries for Linear Algebra Computations on High Performance Computers," *SIAM Review 37,* 151–180.

One way to realize high performance in a matrix computation is to design algorithms that are rich in matrix multiplication and then implement those algorithms using an optimized level-3 BLAS library. For details on this philosophy and its effectiveness, see:

B. Kågström, P. Ling, and C. Van Loan (1998). "GEMM-based Level-3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark," *ACM Trans. Math. Softw. 24*, 268–302.
M.J. Dayde and I.S. Duff (1999). "The RISC BLAS: A Blocked Implementation of Level 3 BLAS for RISC Processors," *ACM Trans. Math. Softw. 25*, 316–340.
E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström (2004). "Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software," *SIAM Review 46*, 3–45.
K. Goto and R. Van De Geign (2008). "Anatomy of High-Performance Matrix Multiplication," *ACM Trans. Math. Softw. 34,* 12:1–12:25.

Advanced data structures that support high performance matrix computations are discussed in:

F.G. Gustavson (1997). "Recursion Leads to Automatic Variable Blocking for Dense Linear Algebra Algorithms," *IBM J. Res. Dev. 41*, 737–755.
V. Valsalam and A. Skjellum (2002). "A Framework for High-Performance Matrix Multiplication Based on Hierarchical Abstractions, Algorithms, and Optimized Low-Level Kernels," *Concurrency Comput. Pract. Exper. 14*, 805–839.
S.R. Chatterjee, P. Patnala, and M. Thottethodi (2002). "Recursive Array Layouts and Fast Matrix Multiplication," *IEEE Trans. Parallel. Distrib. Syst. 13*, 1105–1123.
F.G. Gustavson (2003). "High-Performance Linear Algebra Algorithms Using Generalized Data Structures for Matrices," *IBM J. Res. Dev. 47*, 31–54.
N. Park, B. Hong, and V.K. Prasanna (2003). "Tiling, Block Data Layout, and Memory Hierarchy Performance," *IEEE Trans. Parallel Distrib. Systems, 14*, 640–654.
J.A. Gunnels, F.G. Gustavson, G.M. Henry, and R.A. van de Geijn (2005). "A Family of High-Performance Matrix Multiplication Algorithms," *PARA 2004, LNCS 3732,* 256–265.
P. D'Alberto and A. Nicolau (2009). "Adaptive Winograd's Matrix Multiplications," *ACM Trans. Math. Softw. 36*, 3:1–3:23.

A great deal of effort has gone into the design of software tools that automatically block a matrix computation for high performance, e.g.,

S. Carr and R.B. Lehoucq (1997) "Compiler Blockability of Dense Matrix Factorizations," *ACM Trans. Math. Softw. 23*, 336–361.
J.A. Gunnels, F. G. Gustavson, G.M. Henry, and R. A. van de Geijn (2001). "FLAME: Formal Linear Algebra Methods Environment," *ACM Trans. Math. Softw. 27*, 422–455.
P. Bientinesi, J.A. Gunnels, M.E. Myers, E. Quintana-Orti, and R.A. van de Geijn (2005). "The Science of Deriving Dense Linear Algebra Algorithms," *ACM Trans. Math. Softw. 31*, 1–26.
J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R.C. Whaley, and K. Yelick (2005). "Self-Adapting Linear Algebra Algorithms and Software,", *Proc. IEEE 93,* 293–312.
K. Yotov, X.Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill (2005). "Is Search Really Necessary to Generate High-Performance BLAS?," *Proc. IEEE 93*, 358–386.

For a rigorous treatment of communication lower bounds in matrix computations, see:

G. Ballard, J. Demmel, O. Holtz, and O. Schwartz (2011). "Minimizing Communication in Numerical Linear Algebra," *SIAM J. Matrix Anal. Applic. 32*, 866–901.

1.6 Parallel Matrix Multiplication

The impact of matrix computation research in many application areas depends upon the development of parallel algorithms that *scale*. Algorithms that scale have the property that they remain effective as problem size grows and the number of involved processors increases. Although powerful new programming languages and related system tools continue to simplify the process of implementing a parallel matrix computation, being able to "think parallel" is still important. This requires having an intuition about load balancing, communication overhead, and processor synchronization.

1.6.1 A Model Computation

To illustrate the major ideas associated with parallel matrix computations, we consider the following *model computation*:

> Given $C \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{m\times r}$, and $B \in \mathbb{R}^{r\times n}$, effectively compute the matrix multiplication update $C = C + AB$ assuming the availability of p processors. Each processor has its own *local memory* and executes its own *local program*.

The matrix multiplication update problem is a good choice because it is an inherently parallel computation and because it is at the heart of many important algorithms that we develop in later chapters.

The design of a parallel procedure begins with the breaking up of the given problem into smaller parts that exhibit a measure of independence. In our problem we assume the blocking

$$C = \begin{bmatrix} C_{11} & \cdots & C_{1N} \\ \vdots & \ddots & \vdots \\ C_{M1} & \cdots & C_{MN} \end{bmatrix},\ A = \begin{bmatrix} A_{11} & \cdots & A_{1R} \\ \vdots & \ddots & \vdots \\ A_{M1} & \cdots & A_{MR} \end{bmatrix},\ B = \begin{bmatrix} B_{11} & \cdots & B_{1N} \\ \vdots & \ddots & \vdots \\ B_{R1} & \cdots & B_{RN} \end{bmatrix}, \tag{1.6.1}$$

$$m = m_1 M, \qquad r = r_1 R, \qquad n = n_1 N$$

with $C_{ij} \in \mathbb{R}^{m_1\times n_1}$, $A_{ij} \in \mathbb{R}^{m_1\times r_1}$, and $B_{ij} \in \mathbb{R}^{r_1\times n_1}$. It follows that the $C + AB$ update partitions nicely into MN smaller *tasks*:

$$\text{Task}(i,j)\text{:} \qquad C_{ij} = C_{ij} + \sum_{k=1}^{R} A_{ik}B_{kj}. \tag{1.6.2}$$

Note that the block-block products $A_{ik}B_{kj}$ are all the same size.

Because the tasks are naturally double-indexed, we double index the available processors as well. Assume that $p = p_{\text{row}}p_{\text{col}}$ and designate the (i,j)th processor by $\text{Proc}(i,j)$ for $i = 1{:}p_{\text{row}}$ and $j = 1{:}p_{\text{col}}$. The double indexing of the processors is just a notation and is *not* a statement about their physical connectivity.

1.6.2 Load Balancing

An effective parallel program equitably partitions the work among the participating processors. Two subdivision strategies for the model computation come to mind. The *2-dimensional block distribution* assigns contiguous block updates to each processor. See Figure 1.6.1. Alternatively, we can have $\text{Proc}(\mu,\tau)$ oversee the update of C_{ij} for $i = \mu{:}\,p_{\text{row}}{:}M$ and $j = \tau{:}\,p_{\text{col}}{:}N$. This is called the *2-dimensional block-cyclic distribution*. See Figure 1.6.2. For the displayed example, both strategies assign twelve C_{ij} updates to each processor and each update involves R block-block multiplications, i.e., $12(2m_1n_1r_1)$ flops. Thus, from the flop point of view, both strategies are *load balanced*, by which we mean that the amount of arithmetic computation assigned to each processor is roughly the same.

Proc(1,1) $\begin{Bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \\ C_{41} & C_{42} & C_{43} \end{Bmatrix}$	Proc(1,2) $\begin{Bmatrix} C_{14} & C_{15} & C_{16} \\ C_{24} & C_{25} & C_{26} \\ C_{34} & C_{35} & C_{36} \\ C_{44} & C_{45} & C_{46} \end{Bmatrix}$	Proc(1,3) $\begin{Bmatrix} C_{17} & C_{18} & C_{19} \\ C_{27} & C_{28} & C_{29} \\ C_{37} & C_{38} & C_{39} \\ C_{47} & C_{48} & C_{49} \end{Bmatrix}$
Proc(2,1) $\begin{Bmatrix} C_{51} & C_{52} & C_{53} \\ C_{61} & C_{62} & C_{63} \\ C_{71} & C_{72} & C_{73} \\ C_{81} & C_{82} & C_{83} \end{Bmatrix}$	Proc(2,2) $\begin{Bmatrix} C_{54} & C_{55} & C_{56} \\ C_{64} & C_{65} & C_{66} \\ C_{74} & C_{75} & C_{76} \\ C_{84} & C_{85} & C_{86} \end{Bmatrix}$	Proc(2,3) $\begin{Bmatrix} C_{57} & C_{58} & C_{59} \\ C_{67} & C_{68} & C_{69} \\ C_{77} & C_{78} & C_{79} \\ C_{87} & C_{88} & C_{89} \end{Bmatrix}$

Figure 1.6.1. *The block distribution of tasks* ($M = 8$, $p_{\text{row}} = 2$, $N = 9$, and $p_{\text{col}} = 3$).

Proc(1,1) $\begin{Bmatrix} C_{11} & C_{14} & C_{17} \\ C_{31} & C_{34} & C_{37} \\ C_{51} & C_{54} & C_{57} \\ C_{71} & C_{74} & C_{77} \end{Bmatrix}$	Proc(1,2) $\begin{Bmatrix} C_{12} & C_{15} & C_{18} \\ C_{32} & C_{35} & C_{38} \\ C_{52} & C_{55} & C_{58} \\ C_{72} & C_{75} & C_{78} \end{Bmatrix}$	Proc(1,3) $\begin{Bmatrix} C_{13} & C_{16} & C_{19} \\ C_{33} & C_{36} & C_{39} \\ C_{53} & C_{56} & C_{59} \\ C_{73} & C_{76} & C_{79} \end{Bmatrix}$
Proc(2,1) $\begin{Bmatrix} C_{21} & C_{24} & C_{27} \\ C_{41} & C_{44} & C_{47} \\ C_{61} & C_{64} & C_{67} \\ C_{81} & C_{84} & C_{87} \end{Bmatrix}$	Proc(2,2) $\begin{Bmatrix} C_{22} & C_{25} & C_{28} \\ C_{42} & C_{45} & C_{48} \\ C_{62} & C_{65} & C_{68} \\ C_{82} & C_{85} & C_{88} \end{Bmatrix}$	Proc(2,3) $\begin{Bmatrix} C_{23} & C_{26} & C_{29} \\ C_{43} & C_{46} & C_{49} \\ C_{63} & C_{66} & C_{69} \\ C_{83} & C_{86} & C_{89} \end{Bmatrix}$

Figure 1.6.2. *The block-cyclic distribution of tasks* ($M = 8$, $p_{\text{row}} = 2$, $N = 9$, *and* $p_{\text{col}} = 3$).

If M is not a multiple of p_{row} or if N is not a multiple of p_{col}, then the distribution of work among processors is no longer balanced. Indeed, if

$$\begin{aligned} M &= \alpha_1 p_{\text{row}} + \beta_1, & 0 \le \beta_1 < p_{\text{row}}, \\ N &= \alpha_2 p_{\text{col}} + \beta_2, & 0 \le \beta_2 < p_{\text{col}}, \end{aligned}$$

then the number of block-block multiplications per processor can range from $\alpha_1\alpha_2 R$ to $(\alpha_1+1)(\alpha_2+1)R$. However, this variation is insignificant in a large-scale computation with $M \gg p_{\text{row}}$ and $N \gg p_{\text{col}}$:

$$\frac{(\alpha_1+1)(\alpha_2+1)R}{(\alpha_1\alpha_2)R} = 1 + O\left(\frac{p_{\text{row}}}{M} + \frac{p_{\text{col}}}{N}\right).$$

We conclude that both the block distribution and the block-cyclic distribution strategies are load balanced for the general $C + AB$ update.

This is *not* the case for certain block-sparse situations that arise in practice. If A is block lower triangular and B is block upper triangular, then the amount of work associated with Task(i,j) depends upon i and j. Indeed from (1.6.2) we have

$$C_{ij} = C_{ij} + \sum_{k=1}^{\min\{i,j,R\}} A_{ik}B_{kj}.$$

A very uneven allocation of work for the block distribution can result because the number of flops associated with Task(i,j) increases with i and j. The tasks assigned to Proc$(p_{\text{row}}, p_{\text{col}})$ involve the most work while the tasks assigned to Proc(1,1) involve the least. To illustrate the ratio of workloads, set $M = N = R = \tilde{M}$ and assume that $p_{\text{row}} = p_{\text{col}} = \tilde{p}$ divides $\tilde{M}$. It can be shown that

$$\frac{\text{Flops assigned to Proc}(\tilde{p},\tilde{p})}{\text{Flops assigned to Proc}(1,1)} = O(\tilde{p}) \tag{1.6.3}$$

if we assume $\tilde{M}/\tilde{p} \gg 1$. Thus, load balancing does not depend on problem size and gets worse as the number of processors increase.

This is not the case for the block-cyclic distribution. Again, Proc(1,1) and Proc$(\tilde{p},\tilde{p})$ are the least busy and most busy processors. However, now it can be verified that

$$\frac{\text{Flops assigned to Proc}(\tilde{p},\tilde{p})}{\text{Flops assigned to Proc}(1,1)} = 1 + O\left(\frac{\tilde{p}}{\tilde{M}}\right), \tag{1.6.4}$$

showing that the allocation of work becomes increasingly balanced as the problem size grows.

Another situation where the block-cyclic distribution of tasks is preferred is the case when the first q block rows of A are zero and the first q block columns of B are zero. This situation arises in several important matrix factorization schemes. Note from Figure 1.6.1 that if q is large enough, then some processors have absolutely nothing to do if tasks are assigned according to the block distribution. On the other hand, the block-cyclic distribution is load balanced, providing further justification for this method of task distribution.

1.6.3 Data Motion Overheads

So far the discussion has focused on load balancing from the flop point of view. We now turn our attention to the costs associated with data motion and processor coordination. How does a processor get hold of the data it needs for an assigned task? How does a processor know enough to wait if the data it needs is the output of a computation being performed by another processor? What are the overheads associated with data transfer and synchronization and how do they compare to the costs of the actual arithmetic?

The importance of data locality is discussed in §1.5. However, in a parallel computing environment, the data that a processor needs can be "far away," and if that is the case too often, then it is possible to lose the multiprocessor advantage. Regarding synchronization, time spent waiting for another processor to finish a calculation is time lost. Thus, the design of an effective parallel computation involves paying attention to the number of synchronization points and their impact. Altogether, this makes it difficult to model performance, especially since an individual processor can typically compute and communicate at the same time. Nevertheless, we forge ahead with our analysis of the model computation to dramatize the cost of data motion relative to flops. For the remainder of this section we assume:

(a) The block-cyclic distribution of tasks is used to ensure that arithmetic is load balanced.

(b) Individual processors can perform the computation $C_{ij} = C_{ij} + A_{ik}B_{kj}$ at a rate of F flops per second. Typically, a processor will have its own local memory hierarchy and vector processing capability, so F is an attempt to capture in a single number all the performance issues that we discussed in §1.5.

(c) The time required to move η floating point numbers into or out of a processor is $\alpha + \beta\eta$. In this model, the parameters α and β respectively capture the *latency* and *bandwidth* attributes associated with data transfer.

With these simplifications we can roughly assess the effectiveness of assigning p processors to the update computation $C = C + AB$.

Let $T_{\text{arith}}(p)$ be the time that each processor must spend doing arithmetic as it carries out its share of the computation. It follows from assumptions (a) and (b) that

$$T_{\text{arith}}(p) \approx \frac{2mnr}{pF}. \tag{1.6.5}$$

Similarly, let $T_{\text{data}}(p)$ be the time that each processor must spend acquiring the data it needs to perform its tasks. Ordinarily, this quantity would vary significantly from processor to processor. However, the implementation strategies outlined below have the property that the communication overheads are roughly the same for each processor. It follows that if $T_{\text{arith}}(p) + T_{\text{data}}(p)$ approximates the total execution time for the p-processor solution, then the quotient

$$S(p) = \frac{T_{\text{arith}}(1)}{T_{\text{arith}}(p) + T_{\text{data}}(p)} = \frac{p}{1 + \dfrac{T_{\text{data}}(p)}{T_{\text{arith}}(p)}} \tag{1.6.6}$$

is a reasonable measure of *speedup*. Ideally, the assignment of p processors to the $C = C + AB$ update would reduce the single-processor execution time by a factor of p. However, from (1.6.6) we see that $S(p) < p$ with the compute-to-communicate ratio $T_{\text{data}}(p)/T_{\text{arith}}(p)$ explaining the degradation. To acquire an intuition about this all-important quotient, we need to examine more carefully the data transfer properties associated with each task.

1.6.4 Who Needs What

If a processor carries out Task(i, j), then at some time during the calculation, blocks $C_{ij}, A_{i1}, \ldots, A_{iR}, B_{1j}, \ldots, B_{Rj}$ must find their way into its local memory. Given assumptions (a) and (c), Table 1.6.1 summarizes the associated data transfer overheads for an individual processor:

Required Blocks			Data Transfer Time per Block
C_{ij}	$i = \mu{:}p_{\text{row}}{:}M$	$j = \tau{:}p_{\text{col}}{:}N$	$\alpha + \beta m_1 n_1$
A_{ij}	$i = \mu{:}p_{\text{row}}{:}M$	$j = 1{:}R$	$\alpha + \beta m_1 r_1$
B_{ij}	$i = 1{:}R$	$j = \tau{:}p_{\text{col}}{:}N$	$\alpha + \beta r_1 n_1$

TABLE 1.6.1. *Communication overheads for* Proc(μ, τ)

It follows that if

$$\gamma_C = \text{total number of required } C\text{-block transfers}, \tag{1.6.7}$$

$$\gamma_A = \text{total number of required } A\text{-block transfers}, \tag{1.6.8}$$

$$\gamma_B = \text{total number of required } B\text{-block transfers}, \tag{1.6.9}$$

then

$$T_{\text{data}}(p) \approx \gamma_C(\alpha + \beta m_1 n_1) + \gamma_A(\alpha + \beta m_1 r_1) + \gamma_B(\alpha + \beta r_1 n_1),$$

and so from (1.6.5) we have

$$\frac{T_{\text{data}}(p)}{T_{\text{arith}}(p)} \approx \frac{Fp}{2}\left(\alpha\frac{\gamma_C + \gamma_A + \gamma_B}{mnr} + \beta\left(\frac{\gamma_C}{MNr} + \frac{\gamma_A}{MnR} + \frac{\gamma_B}{mNR}\right)\right). \tag{1.6.10}$$

To proceed further with our analysis, we need to estimate the γ-factors (1.6.7)–(1.6.9), and that requires assumptions about how the underlying architecture stores and accesses the matrices A, B, and C.

1.6.5 The Shared-Memory Paradigm

In a *shared-memory system* each processor has access to a common, global memory. See Figure 1.6.3. During program execution, data flows to and from the global memory and this represents a significant overhead that we proceed to assess. Assume that the matrices C, A, and B are in global memory at the start and that Proc(μ, τ) executes the following:

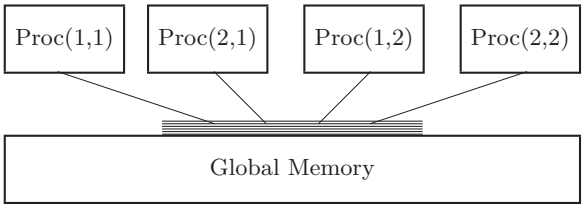

Figure 1.6.3. *A four-processor shared-memory system*

for $i = \mu{:}p_{\text{row}}{:}M$
  for $j = \tau{:}p_{\text{col}}{:}N$
    $C^{(\text{loc})} \leftarrow C_{ij}$
    for $k = 1{:}R$
      $A^{(\text{loc})} \leftarrow A_{ik}$
      $B^{(\text{loc})} \leftarrow B_{kj}$ (Method 1)
      $C^{(\text{loc})} = C^{(\text{loc})} + A^{(\text{loc})}B^{(\text{loc})}$
    end
    $C_{ij} \leftarrow C^{(\text{loc})}$
  end
end

As a reminder of the interactions between global and local memory, we use the "←" notation to indicate data transfers between these memory levels and the "loc" superscript to designate matrices in local memory. The block transfer statistics (1.6.7)-(1.6.9) for Method 1 are given by

$$\begin{aligned} \gamma_C &\approx 2(MN/p), \\ \gamma_A &\approx R(MN/p), \\ \gamma_B &\approx R(MN/p), \end{aligned}$$

and so from (1.6.10) we obtain

$$\frac{T_{\text{data}}(p)}{T_{\text{arith}}(p)} \approx \frac{F}{2}\left(\alpha\frac{2+2R}{m_1 n_1 r} + \beta\left(\frac{2}{r}+\frac{1}{n_1}+\frac{1}{m_1}\right)\right). \tag{1.6.11}$$

By substituting this result into (1.6.6) we conclude that (a) speed-up degrades as the flop rate F increases and (b) speedup improves if the communication parameters α and β decrease or the block dimensions m_1, n_1, and r_1 increase. Note that the communicate-to-compute ratio (1.6.11) for Method 1 does not depend upon the number of processors.

Method 1 has the property that it is only necessary to store one C-block, one A-block, and one B-block in local memory at any particular instant, i.e., $C^{(\mathrm{loc})}$, $A^{(\mathrm{loc})}$, and $B^{(\mathrm{loc})}$. Typically, a processor's local memory is much smaller than global memory, so this particular solution approach is attractive for problems that are very large relative to local memory capacity. However, there is a hidden cost associated with this economy because in Method 1, each A-block is loaded N/p_{col} times and each B-block is loaded M/p_{row} times. This redundancy can be eliminated if each processor's local memory is large enough to house simultaneously all the C-blocks, A-blocks, and B-blocks that are required by its assigned tasks. Should this be the case, then the following method involves much less data transfer:

```
for k = 1:R
    A_ik^(loc) ← A_ik      (i = μ:p_row:M)
    B_kj^(loc) ← B_kj      (j = τ:p_col:N)
end
for i = μ:p_row:M
    for j = τ:p_col:N
        C^(loc) ← C_ij
        for k = 1:R                                   (Method 2)
            C^(loc) = C^(loc) + A_ik^(loc) B_kj^(loc)
        end
        C_ij ← C^(loc)
    end
end
```

The block transfer statistics γ'_C, γ'_A, and γ'_B, for Method 2 are more favorable than for Method 1. It can be shown that

$$\gamma'_C = \gamma_C, \qquad \gamma'_A = \gamma_A f_{\mathrm{col}}, \qquad \gamma'_B = \gamma_B f_{\mathrm{row}}, \tag{1.6.12}$$

where the quotients $f_{\mathrm{col}} = p_{\mathrm{col}}/N$ and $f_{\mathrm{row}} = p_{\mathrm{row}}/M$ are typically much less than unity. As a result, the communicate-to-compute ratio for Method 2 is given by

$$\frac{T_{\mathrm{data}}(p)}{T_{\mathrm{arith}}(p)} \approx \frac{F}{2}\left(\alpha\frac{2+R\,(f_{\mathrm{col}}+f_{\mathrm{row}})}{m_1 n_1 r} + \beta\left(\frac{2}{r}+\frac{1}{n_1}f_{\mathrm{col}}+\frac{1}{m_1}f_{\mathrm{row}}\right)\right), \tag{1.6.13}$$

which is an improvement over (1.6.11). Methods 1 and 2 showcase the trade-off that frequently exists between local memory capacity and the overheads that are associated with data transfer.

1.6.6 Barrier Synchronization

The discussion in the previous section assumes that C, A, and B are available in global memory at the start. If we extend the model computation so that it includes the

multiprocessor initialization of these three matrices, then an interesting issue arises. How does a processor "know" when the initialization is complete and it is therefore safe to begin its share of the $C = C + AB$ update?

Answering this question is an occasion to introduce a very simple synchronization construct known as the *barrier*. Suppose the C-matrix is initialized in global memory by assigning to each processor some fraction of the task. For example, Proc(μ, τ) could do this:

for $i = \mu{:}p_{\text{row}}{:}M$
  for $j = \tau{:}p_{\text{col}}{:}N$
    Compute the (i, j) block of C and store in $C^{(\text{loc})}$.
    $C_{ij} \leftarrow C^{(\text{loc})}$
  end
end

Similar approaches can be taken for the setting up of $A = (A_{ij})$ and $B = (B_{ij})$. Even if this partitioning of the initialization is load balanced, it cannot be assumed that each processor completes its share of the work at exactly the same time. This is where the barrier synchronization is handy. Assume that Proc(μ, τ) executes the following:

$$\begin{array}{lllll}
\text{Initialize } C_{ij}, & i = \mu : p_{\text{row}} : M, & j = \tau : p_{\text{col}} : N & & \\
\text{Initialize } A_{ij}, & i = \mu : p_{\text{row}} : M, & j = \tau : p_{\text{col}} : R & & \\
\text{Initialize } B_{ij}, & i = \mu : p_{\text{row}} : R, & j = \tau : p_{\text{col}} : N & & (1.6.14) \\
\mathsf{barrier} & & & & \\
\text{Update } C_{ij}, & i = \mu : p_{\text{row}} : M, & j = \tau : p_{\text{col}} : N & &
\end{array}$$

To understand the barrier command, regard a processor as being either "blocked" or "free." Assume in (1.6.14) that all processors are free at the start. When it executes the barrier command, a processor becomes blocked and suspends execution. After the last processor is blocked, all the processors return to the free state and resume execution. In (1.6.14), the barrier does not allow the C_{ij} updating via Methods 1 or 2 to begin until all three matrices are fully initialized in global memory.

1.6.7 The Distributed-Memory Paradigm

In a *distributed-memory system* there is no global memory. The data is collectively housed in the local memories of the individual processors which are connected to form a network. There are many possible network topologies. An example is displayed in Figure 1.6.4. The cost associated with sending a message from one processor to another is likely to depend upon how "close" they are in the network. For example, with the torus in Figure 1.6.4, a message from Proc(1,1) to Proc(1,4) involves just one "hop" while a message from Proc(1,1) to Proc(3,3) would involve four.

Regardless, the message-passing costs in a distributed memory system have a serious impact upon performance just as the interactions with global memory affect performance in a shared memory system. Our goal is to approximate these costs as they might arise in the model computation. For simplicity, we make no assumptions about the underlying network topology.

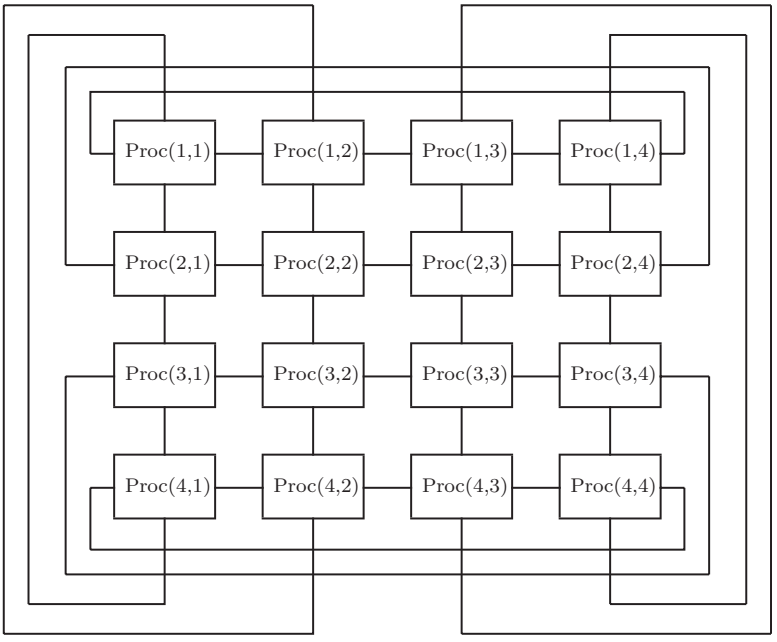

Figure 1.6.4. *A 2-Dimensional Torus*

Let us first assume that $M = N = R = p_{\text{row}} = p_{\text{col}} = 2$ and that the C, A, and B matrices are distributed as follows:

Proc(1,1)
$C_{11},\ A_{11},\ B_{11}$

Proc(1,2)
$C_{12},\ A_{12},\ B_{12}$

Proc(2,1)
$C_{21},\ A_{21},\ B_{21}$

Proc(2,2)
$C_{22},\ A_{22},\ B_{22}$

Assume that Proc(i, j) oversees the update of C_{ij} and notice that the required data for this computation is not entirely local. For example, Proc(1,1) needs to receive a copy of A_{12} from Proc(1,2) and a copy of B_{21} from Proc(2,1) before it can complete the update $C_{11} = C_{11} + A_{11}B_{11} + A_{12}B_{21}$. Likewise, it must send a copy of A_{11} to Proc(1,2) and a copy of B_{11} to Proc(2,1) so that they can carry out their respective updates. Thus, the local programs executing on each processor involve a mix of computational steps and message-passing steps:

Proc(1,1)
Send a copy of A_{11} to Proc(1,2)
Receive a copy of A_{12} from Proc(1,2)
Send a copy of B_{11} to Proc(2,1)
Receive a copy of B_{21} from Proc(2,1)
$C_{11} = C_{11} + A_{11}B_{11} + A_{12}B_{21}$

Proc(1,2)
Send a copy of A_{12} to Proc(1,1)
Receive a copy of A_{11} from Proc(1,1)
Send a copy of B_{12} to Proc(2,2)
Receive a copy of B_{22} from Proc(2,2)
$C_{12} = C_{12} + A_{11}B_{12} + A_{12}B_{22}$

Proc(2,1)
Send a copy of A_{21} to Proc(2,2)
Receive a copy of A_{22} from Proc(2,2)
Send a copy of B_{21} to Proc(1,1)
Receive a copy of B_{11} from Proc(1,1)
$C_{21} = C_{21} + A_{21}B_{11} + A_{22}B_{21}$

Proc(2,2)
Send a copy of A_{22} to Proc(2,1)
Receive a copy of A_{21} from Proc(2,1)
Send a copy of B_{22} to Proc(1,2)
Receive a copy of B_{12} from Proc(1,2)
$C_{22} = C_{22} + A_{21}B_{12} + A_{22}B_{22}$

This informal specification of the local programs does a good job delineating the duties of each processor, but it hides several important issues that have to do with the timeline of execution. (a) Messages do not necessarily arrive at their destination in the order that they were sent. How will a receiving processor know if it is an A-block or a B-block? (b) Receive-a-message commands can block a processor from proceeding with the rest of its calculations. As a result, it is possible for a processor to wait forever for a message that its neighbor never got around to sending. (c) Overlapping computation with communication is critical for performance. For example, after A_{11} arrives at Proc(1,2), the "half" update $C_{12} = C_{12} + A_{11}B_{12}$ can be carried out while the wait for B_{22} continues.

As can be seen, distributed-memory matrix computations are quite involved and require powerful systems to manage the packaging, tagging, routing, and reception of messages. The discussion of such systems is outside the scope of this book. Nevertheless, it is instructive to go beyond the above 2-by-2 example and briefly anticipate the data transfer overheads for the general model computation. Assume that Proc(μ, τ) houses these matrices:

$$\begin{aligned} &C_{ij}, \quad i = \mu : p_{\text{row}} : M, \quad j = \tau : p_{\text{col}} : N, \\ &A_{ij}, \quad i = \mu : p_{\text{row}} : M, \quad j = \tau : p_{\text{col}} : R, \\ &B_{ij}, \quad i = \mu : p_{\text{row}} : R, \quad j = \tau : p_{\text{col}} : N. \end{aligned}$$

From Table 1.6.1 we conclude that if Proc(μ, τ) is to update C_{ij} for $i = \mu : p_{\text{row}} : M$ and $j = \tau : p_{\text{col}} : N$, then it must

(a) For $i = \mu : p_{\text{row}} : M$ and $j = \tau : p_{\text{col}} : R$, send a copy of A_{ij} to

$$\text{Proc}(\mu, 1), \ldots, \text{Proc}(\mu, \tau - 1), \text{Proc}(\mu, \tau + 1), \ldots, \text{Proc}(\mu, p_{\text{col}}).$$

Data transfer time $\approx (p_{\text{col}} - 1)(M/p_{\text{row}})(R/p_{\text{col}})\,(\alpha + \beta m_1 r_1)$

(b) For $i = \mu : p_{\text{row}} : R$ and $j = \tau : p_{\text{col}} : N$, send a copy of B_{ij} to

$$\text{Proc}(1, \tau), \ldots, \text{Proc}(\mu - 1), \tau), \text{Proc}(\mu + 1, \tau), \ldots, \text{Proc}(p_{\text{row}}, \tau).$$

Data transfer time $\approx (p_{\text{row}} - 1)(R/p_{\text{row}})(N/p_{\text{col}})\,(\alpha + \beta r_1 n_1)$

(c) Receive copies of the A-blocks that are sent by processors

$$\text{Proc}(\mu,1),\ldots,\ \text{Proc}(\mu,\tau-1),\ \text{Proc}(\mu,\tau+1),\ldots,\ \text{Proc}(\mu,p_{\text{col}}).$$

Data transfer time $\approx (p_{\text{col}}-1)(M/p_{\text{row}})(R/p_{\text{col}})\,(\alpha+\beta m_1 r_1)$

(d) Receive copies of the B-blocks that are sent by processors

$$\text{Proc}(1,\tau),\ldots,\ \text{Proc}(\mu-1),\tau),\ \text{Proc}(\mu+1,\tau),\ldots,\ \text{Proc}(p_{\text{row}},\tau).$$

Data transfer time $\approx (p_{\text{row}}-1)(R/p_{\text{row}})(N/p_{\text{col}})\,(\alpha+\beta r_1 n_1)$

Let T_{data} be the summation of these data transfer overheads and recall that $T_{\text{arith}} = (2mnr)/(Fp)$ since arithmetic is evenly distributed around the processor network. It follows that

$$\frac{T_{\text{data}}(p)}{T_{\text{arith}}(p)} \approx F\left(\alpha\left(\frac{p_{\text{col}}}{m_1 r_1 n}+\frac{p_{\text{row}}}{m r_1 n_1}\right)+\beta\left(\frac{p_{\text{col}}}{n}+\frac{p_{\text{row}}}{m}\right)\right). \tag{1.6.15}$$

Thus, as problem size grows, this ratio tends to zero and speedup approaches p according to (1.6.6).

1.6.8 Cannon's Algorithm

We close with a brief description of the Cannon (1969) matrix multiplication scheme. The method is an excellent way to showcase the toroidal network displayed in Figure 1.6.4 together with the idea of "nearest-neighbor" thinking which is quite important in distributed matrix computations. For clarity, let us assume that $A=(A_{ij})$, $B=(B_{ij})$, and $C=(C_{ij})$ are 4-by-4 block matrices with n_1-by-n_1 blocks. Define the matrices

$$A^{(1)} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{22} & A_{23} & A_{24} & A_{21} \\ A_{33} & A_{34} & A_{31} & A_{32} \\ A_{44} & A_{41} & A_{42} & A_{43} \end{bmatrix}, \quad B^{(1)} = \begin{bmatrix} B_{11} & B_{22} & B_{33} & B_{44} \\ B_{21} & B_{32} & B_{43} & B_{14} \\ B_{31} & B_{42} & B_{13} & B_{24} \\ B_{41} & B_{12} & B_{23} & B_{34} \end{bmatrix},$$

$$A^{(2)} = \begin{bmatrix} A_{14} & A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{32} & A_{33} & A_{34} & A_{31} \\ A_{43} & A_{44} & A_{41} & A_{42} \end{bmatrix}, \quad B^{(2)} = \begin{bmatrix} B_{41} & B_{12} & B_{23} & B_{34} \\ B_{11} & B_{22} & B_{33} & B_{44} \\ B_{21} & B_{32} & B_{43} & B_{14} \\ B_{31} & B_{42} & B_{13} & B_{24} \end{bmatrix},$$

$$A^{(3)} = \begin{bmatrix} A_{13} & A_{14} & A_{11} & A_{12} \\ A_{24} & A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{42} & A_{43} & A_{44} & A_{41} \end{bmatrix}, \quad B^{(3)} = \begin{bmatrix} B_{31} & B_{42} & B_{13} & B_{24} \\ B_{41} & B_{12} & B_{23} & B_{34} \\ B_{11} & B_{22} & B_{33} & B_{44} \\ B_{21} & B_{32} & B_{43} & B_{14} \end{bmatrix},$$

$$A^{(4)} = \begin{bmatrix} A_{12} & A_{13} & A_{14} & A_{11} \\ A_{23} & A_{24} & A_{21} & A_{22} \\ A_{34} & A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}, \quad B^{(4)} = \begin{bmatrix} B_{21} & B_{32} & B_{43} & B_{14} \\ B_{31} & B_{42} & B_{13} & B_{24} \\ B_{41} & B_{12} & B_{23} & B_{34} \\ B_{11} & B_{22} & B_{33} & B_{44} \end{bmatrix},$$

and note that

$$C_{ij} = A_{ij}^{(1)}B_{ij}^{(1)} + A_{ij}^{(2)}B_{ij}^{(2)} + A_{ij}^{(3)}B_{ij}^{(3)} + A_{ij}^{(4)}B_{ij}^{(4)}. \tag{1.6.16}$$

Refer to Figure 1.6.4 and assume that Proc(i,j) is in charge of computing C_{ij} and that at the start it houses both $A_{ij}^{(1)}$ and $B_{ij}^{(1)}$. The message passing required to support the updates

$$C_{ij} = C_{ij} + A_{ij}^{(1)} B_{ij}^{(1)}, \tag{1.6.17}$$

$$C_{ij} = C_{ij} + A_{ij}^{(2)} B_{ij}^{(2)}, \tag{1.6.18}$$

$$C_{ij} = C_{ij} + A_{ij}^{(3)} B_{ij}^{(3)}, \tag{1.6.19}$$

$$C_{ij} = C_{ij} + A_{ij}^{(4)} B_{ij}^{(4)}, \tag{1.6.20}$$

involves communication with Proc(i,j)'s four neighbors in the toroidal network. To see this, define the block downshift permutation

$$P = \begin{bmatrix} 0 & 0 & 0 & I_{n_1} \\ I_{n_1} & 0 & 0 & 0 \\ 0 & I_{n_1} & 0 & 0 \\ 0 & 0 & I_{n_1} & 0 \end{bmatrix}$$

and observe that $A^{(k+1)} = A^{(k)} P^T$ and $B^{(k+1)} = P B^{(k)}$. That is, the transition from $A^{(k)}$ to $A^{(k+1)}$ involves shifting A-blocks to the right one column (with wraparound) while the transition from $B^{(k)}$ to $B^{(k+1)}$ involves shifting the B-blocks down one row (with wraparound). After each update (1.6.17)–(1.6.20), the housed A-block is passed to Proc(i,j)'s "east" neighbor and the next A-block is received from its "west" neighbor. Likewise, the housed B-block is sent to its "south" neighbor and the next B-block is received from its "north" neighbor.

Of course, the Cannon algorithm can be implemented on any processor network. But we see from the above that it is particularly well suited when there are toroidal connections for then communication is always between adjacent processors.

Problems

P1.6.1 Justify Equations (1.6.3) and (1.6.4).

P1.6.2 Contrast the two task distribution strategies in §1.6.2 for the case when the first q block rows of A are zero and the first q block columns of B are zero.

P1.6.3 Verify Equations (1.6.13) and (1.6.15).

P1.6.4 Develop a shared memory method for overwriting A with A^2 where it is assumed that $A \in \mathbb{R}^{n\times n}$ resides in global memory at the start.

P1.6.5 Develop a shared memory method for computing $B = A^T A$ where it is assumed that $A \in \mathbb{R}^{m\times n}$ resides in global memory at the start and that B is stored in global memory at the end.

P1.6.6 Prove (1.6.16) for general N. Use the block downshift matrix to define $A^{(i)}$ and $B^{(i)}$.

Notes and References for §1.6

To learn more about the practical implementation of parallel matrix multiplication, see `scaLAPACK` as well as:

L. Cannon (1969). "A Cellular Computer to Implement the Kalman Filter Algorithm," PhD Thesis, Montana State University, Bozeman, MT.

K. Gallivan, W. Jalby, and U. Meier (1987). "The Use of BLAS3 in Linear Algebra on a Parallel Processor with a Hierarchical Memory," *SIAM J. Sci. Stat. Comput. 8,* 1079–1084.
P. Bjørstad, F. Manne, T.Sørevik, and M. Vajteršic (1992). "Efficient Matrix Multiplication on SIMD Computers," *SIAM J. Matrix Anal. Appl. 13*, 386–401.
S.L. Johnsson (1993). "Minimizing the Communication Time for Matrix Multiplication on Multiprocessors," *Parallel Comput. 19*, 1235–1257.
K. Mathur and S.L. Johnsson (1994). "Multiplication of Matrices of Arbitrary Shape on a Data Parallel Computer," *Parallel Comput. 20*, 919–952.
J. Choi, D.W. Walker, and J. Dongarra (1994) "Pumma: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers," *Concurrency: Pract. Exper. 6*, 543-570.
R.C. Agarwal, F.G. Gustavson, and M. Zubair (1994). "A High-Performance Matrix-Multiplication Algorithm on a Distributed-Memory Parallel Computer, Using Overlapped Communication," *IBM J. Res. Devel. 38*, 673–681.
D. Irony, S. Toledo, and A. Tiskin (2004). "Communication Lower Bounds for Distributed Memory Matrix Multiplication," *J. Parallel Distrib. Comput. 64*, 1017–1026.

Lower bounds for communication overheads are important as they establish a target for implementers, see:

G. Ballard, J. Demmel, O. Holtz, and O. Schwartz (2011). "Minimizing Communication in Numerical Linear Algebra," *SIAM. J. Matrix Anal. Applic. 32,* 866–901.

Matrix transpose in a distributed memory environment is surprisingly complex. The study of this central, no-flop calculation is a reminder of just how important it is to control the costs of data motion. See

S.L. Johnsson and C.T. Ho (1988). "Matrix Transposition on Boolean N-cube Configured Ensemble Architectures," *SIAM J. Matrix Anal. Applic. 9*, 419–454.
J. Choi, J.J. Dongarra, and D.W. Walker (1995). "Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers," *Parallel Comput. 21*, 1387–1406.

The parallel matrix computation literature is a vast, moving target. Ideas come and go with shifts in architectures. Nevertheless, it is useful to offer a small set of references that collectively trace the early development of the field:

D. Heller (1978). "A Survey of Parallel Algorithms in Numerical Linear Algebra," *SIAM Review 20,* 740–777.
J.M. Ortega and R.G. Voigt (1985). "Solution of Partial Differential Equations on Vector and Parallel Computers," *SIAM Review 27,* 149–240.
D.P. O'Leary and G.W. Stewart (1985). "Data Flow Algorithms for Parallel Matrix Computations," *Commun. ACM 28,* 841–853.
J.J. Dongarra and D.C. Sorensen (1986). "Linear Algebra on High Performance Computers," *Appl. Math. Comput. 20,* 57–88.
M.T. Heath, ed. (1987). *Hypercube Multiprocessors*, SIAM Publications, Philadelphia, PA.
Y. Saad and M.H. Schultz (1989). "Data Communication in Parallel Architectures," *J. Dist. Parallel Comput. 11*, 131–150.
J.J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst (1990). *Solving Linear Systems on Vector and Shared Memory Computers,* SIAM Publications, Philadelphia, PA.
K.A. Gallivan, R.J. Plemmons, and A.H. Sameh (1990). "Parallel Algorithms for Dense Linear Algebra Computations," *SIAM Review 32*, 54–135.
J.W. Demmel, M.T. Heath, and H.A. van der Vorst (1993). "Parallel Numerical Linear Algebra," in *Acta Numerica 1993*, Cambridge University Press.
A. Edelman (1993). "Large Dense Numerical Linear Algebra in 1993: The Parallel Computing Influence," *Int. J. Supercomput. Applic. 7*, 113–128.