LAFF-On
Programming for High Performance

Robert A. van de Geijn

Devangi N. Parikh

Jianyu Huang

Margaret E. Myers

Release Date
Monday 25th June, 2018

This is a work in progress

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data not yet available
Draft Edition, Monday 25th June, 2018
This "Draft Edition" allows this material to be used while we sort out through what mechanism we will publish the book.

Contents

Preface

Acknowledgments

The cover was derived from an image that is copyrighted by the Texas Advanced Computing Center (TACC). It is used with permission.

Week 1

Matrix-Matrix Multiplication Basics

1.1 Opening Remarks

1.1.1 Launch

Homework 1.1.1.1 Compute

$$\begin{pmatrix} 1 & -2 & 2 \\ -1 & 1 & 3 \\ -2 & 2 & -1 \end{pmatrix} \begin{pmatrix} -2 & 1 \\ 1 & 3 \\ -1 & 2 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ -1 & 2 \\ -2 & 1 \end{pmatrix} =$$

☛ SEE ANSWER

Let A, B, and C be $m \times k$, $k \times n$, and $m \times n$ matrices, respectively. We can expose their individual entries as

$$A = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{pmatrix}, B = \begin{pmatrix} \beta_{0,0} & \beta_{0,1} & \cdots & \beta_{0,n-1} \\ \beta_{1,0} & \beta_{1,1} & \cdots & \beta_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \beta_{k-1,0} & \beta_{k-1,1} & \cdots & \beta_{k-1,n-1} \end{pmatrix},$$

and

$$C = \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{pmatrix}.$$

The computation $C := AB + C$, which adds the result of matrix-matrix multiplication AB to a matrix

```
void Gemm_IJP( int m, int n, int k,
               double *A, int ldA,
               double *B, int ldB,
               double *C, int ldC )
{
  for ( int i=0; i<m; i++ )
    for ( int j=0; j<n; j++ )
      for ( int p=0; p<k; p++ )
        gamma( i,j ) += alpha( i,p ) * beta( p,j );
}
```

Figure 1.1: C implementation of IJP ordering for computing MMM.

C, is defined as

$$\gamma_{i,j} = \sum_{p=0}^{k-1} \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$$

for all $0 \leq i < m$ and $0 \leq j < n$. This leads to the following pseudo-code for computing $C := AB + C$:

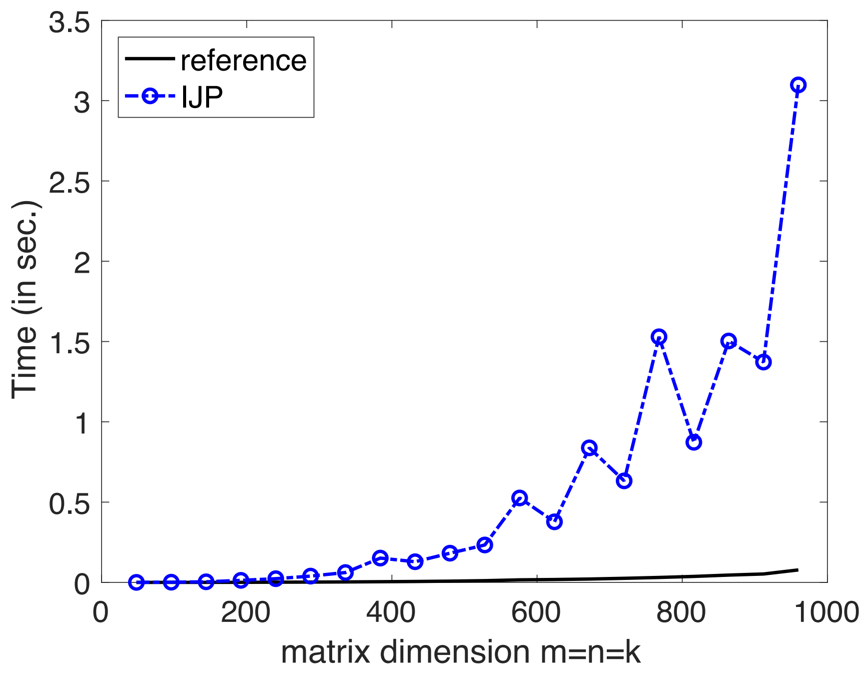
```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
    for $p := 0, \ldots, k-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end
```

The outer two loops visit each element of C, and the inner loop updates $\gamma_{i,j}$ with the dot product of the ith row of A with the jth column of B.

Homework 1.1.1.2 In the file ☛ `Assignments/Week1/C/Gemm_IJP.c` you will find a simple implementation that computes $C := AB + C$ (GEMM). Part of the code is given in Figure 1.1. Compile, link, and execute it by following the instructions in the MATLAB Live Script ☛ `Plot_IJP.mlx` in the same directory.

☛ SEE ANSWER

On Robert's laptop, Homework 1.1.1.2 yields the two graphs given in Figure 1.2 (top) as the curve labeled with IJP. In the first, the time required to compute GEMM as a function of the matrix size is plotted, where $m = n = k$ (each matrix is square). The "dips" in the time required to complete can be attributed to a number of factors, including that other processes that are executing on the same processor may be disrupting the computation. One should not be too concerned about those. In the graph, we also show the time it takes to complete the same computations with a highly optimized implementation. To the uninitiated in high-performance computing (HPC), the



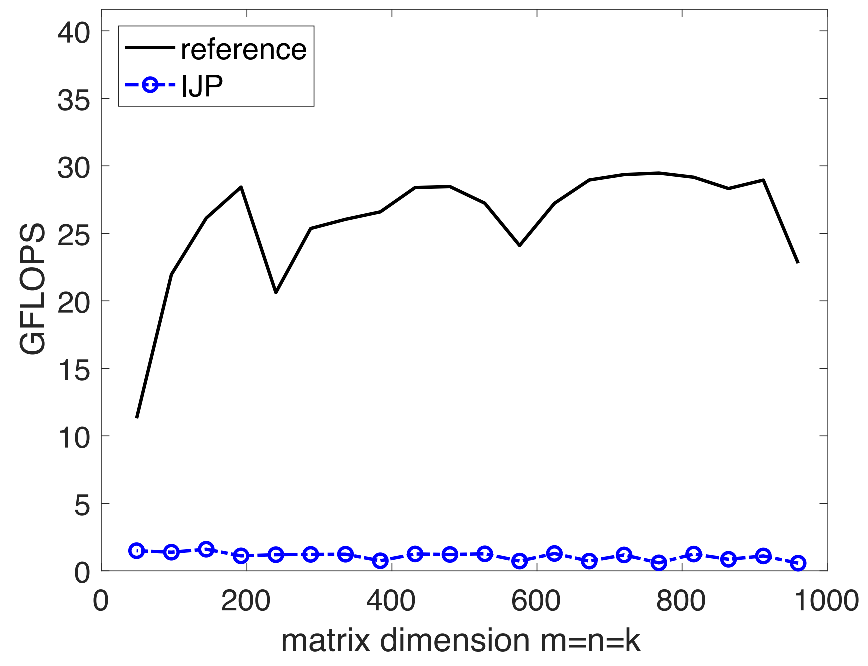


Figure 1.2: Performance comparison of a simple triple-nested loop and a high-performance implementation of matrix-matrix multiplication. Top: Time for computing $C := AB + C$. Bottom: Rate of computation, in billions of floating point operations per second (GFLOPS).

difference is the time required may be a bit of a shock. It makes one realize who inefficient many of the programs we write are.

The performance of a MMM implementation is measured in billions of floating point operations (flops) per second (GFLOPS). The idea is that we know that it takes $2mnk$ flops to compute $C := AB + C$ where C is $m \times n$, A is $m \times k$, and B is $k \times n$. If we measure the time it takes to complete the computation, $T(m,n,k)$, then the rate at which we compute is given by

$$\frac{2mnk}{T(m,n,k)} \times 10^{-9} \text{ GFLOPS}.$$

For our implementation and the reference implementation, this is reported in Figure 1.2 (Bottom). Again, don't worry too much about the dips in the curves. If we controlled the environment in which we performed the experiments, these would largely disappear.

Of course, we don't at this point even know if the reference implementation attains good performance. To investigate this, it pays to compute the theoretical peak performance of the processor. In order to do this,

- You will want to find out what exact processor is in your computer. On a Mac, you can type

  ```
  sysctl -n machdep.cpu.brand_string
  ```

  in a terminal session. On Robert's laptop, this yields

  ```
  Intel(R) Core(TM) i5-4250U CPU @ 1.30GHz.
  ```

  On a Linux system, you can instead type

  ```
  lscpu
  ```

  at the command line, which will give you much more detailed information.

- If you then do an internet search, you may find a page like

  ☛ Intel Core i5-4250U Processor

  which tells Robert's laptop is a Haswell processor (look for the "Code name").

- Now this is where it gets interesting: On this page it tells you the Processor Base Frequency is 1.3 GHz, but that the Max Turbo Frequency is 2.60 GHz. The turbo frequency is the frequency at which the processor can operate under special circumstance. The more cores are active, the lower the frequency at which the processor operates to reduce the temperature and power use of the processor.

- You can find out more about what clock rate can be expected by consulting a document like

  ☛ Intel Xeon Processor E5 v3 Product Family.

  This document may make your head spin...

You will want to investigate the above for the processor of the machine on which you perform your experiments.

For the sake of discussion, let's use the clock rate of 2.6 GHz in turbo for Robert's laptop.

- The next thing you need to know is how many flops each core can perform per clock cycle. We happen to know that a Haswell core can perform sixteen flops per cycle.
- Thus, the peak performance of a single core is given by $2.6 \times 16 = 41.6$ GFLOPS when performing double precision real computations (which is what we will focus on in this course).

What we conclude is that the best we can expect, under idealized circumstances, is that our implementation will achieve 41.6 GFLOPS. Looking at Figure 1.2 (bottom), we see that the reference implementation achieves a reasonable rate of computation, while the simple triple loop is dismal in its performance. In that graph, and many that will follow, we make the top of the graph represent the theoretical peak.

1.1.2 Outline Week 1

1.1.3 What you will learn

Upon completion of this week, you should be able to

- map matrices to memory;
- apply conventions to describe how to index into arrays that store matrices;
- observe the effects of loop order on performance;
- recognize that simple implementations may not provide the performance that can be achieved;
- realize that compilers don't automagically do all the optimization for you.

1.2 Mapping Matrices to Memory

In this section, learners find out

- How to map matrices to memory.
- Conventions we will use to describe how we index into the arrays that store matrices.
- That loop order affects performance.
- The performance of a simple implementation leaves a lot to be desired.
- High-performance can be achieved.

The learner is left wondering "why?"

1.2.1 Column-major ordering

Matrices are stored in two-dimensional arrays while computer memory is inherently one-dimensional in the way it is addressed. So, we need to agree on how we are going to store matrices in memory.

Consider the matrix

$$\left(\begin{array}{rrr} 1 & -2 & 2 \\ -1 & 1 & 3 \\ -2 & 2 & -1 \end{array}\right)$$

from the opener of this week. In memory, this may be stored in an array `A` by columns:

$$
\begin{array}{lc|r|}
\cline{3-3}
\texttt{A} & \longrightarrow & 1 \\
\texttt{A[1]} & \longrightarrow & -1 \\
\texttt{A[2]} & \longrightarrow & -2 \\
\cline{3-3}
\texttt{A[3]} & \longrightarrow & -2 \\
\vdots & & 1 \\
 & & 2 \\
\cline{3-3}
 & & 2 \\
 & & 3 \\
 & & -1 \\
\cline{3-3}
\end{array}
$$

which is known as *column-major ordering*.

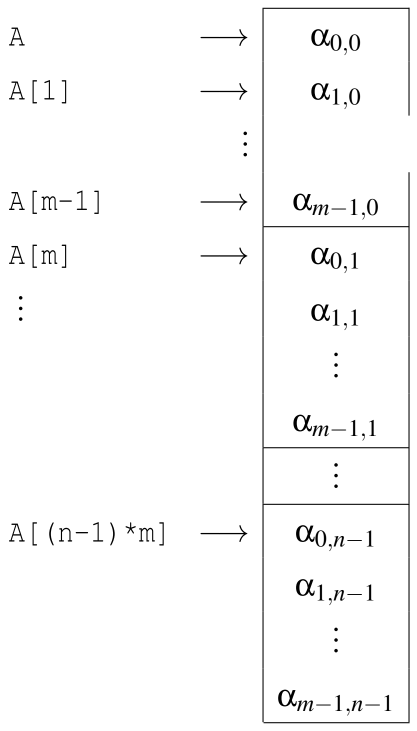


Figure 1.3: Mapping of $m \times n$ matrix A to memory with column-major order.

More generally, consider the matrix

$$\begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{pmatrix}.$$

Column-major ordering would store this in array A as illustrated in Figure 1.3. Obviously, one could use the alternative known as *row-major ordering*.

Homework 1.2.1.1 Exercise on mapping matrices to memory here.

☛ SEE ANSWER

1.2.2 The leading dimension

Very frequently, we will work with a matrix that is a submatrix of a larger matrix. Consider Figure 1.4. What we depict there is a matrix that is embedded in a larger matrix. The larger matrix

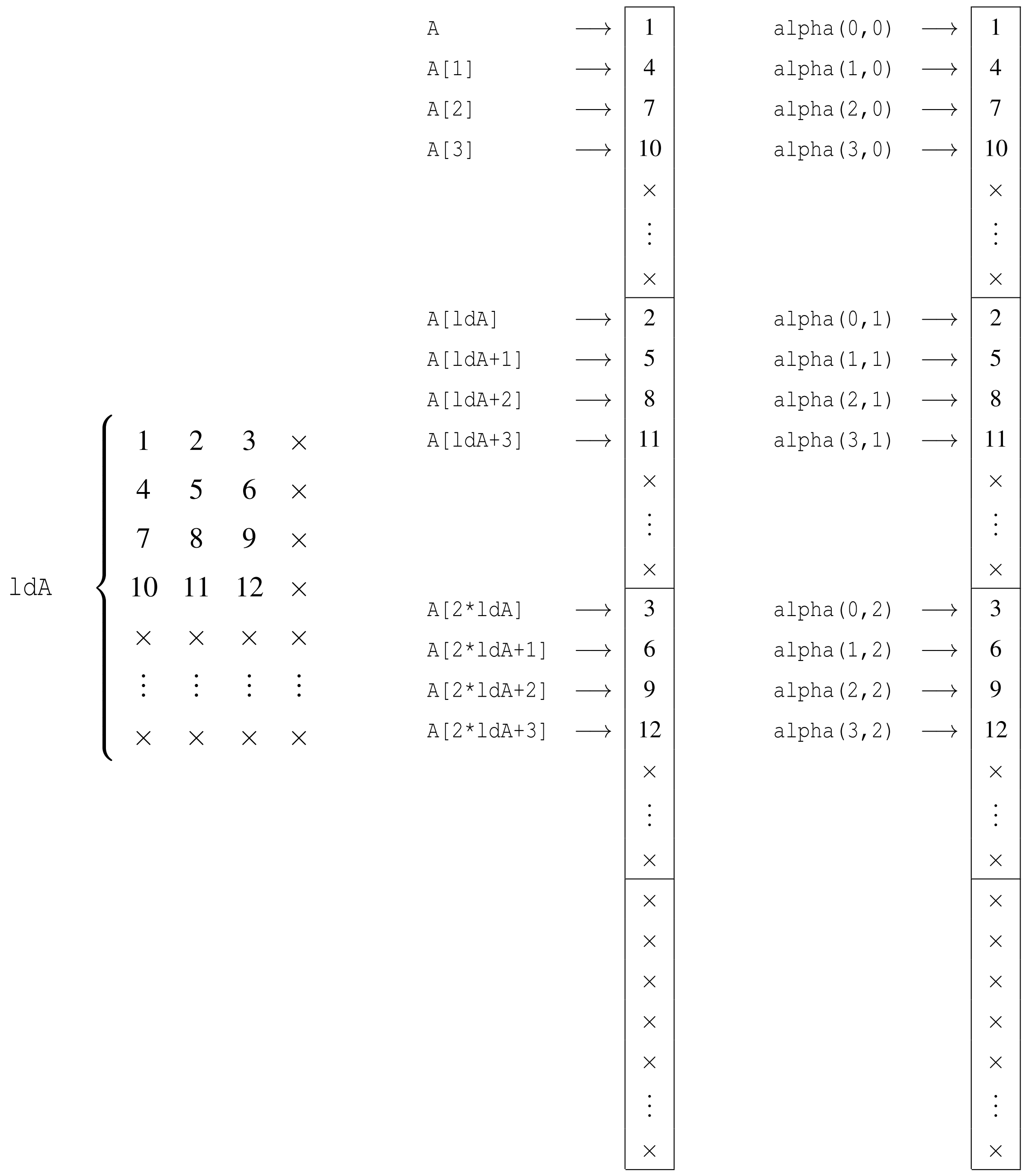


Figure 1.4: Addressing a matrix embedded in an array with ldA rows. At the left we illustrate a 4×3 submatrix of a ldA $\times$ 4 matrix. In the middle, we illustrate how this is mapped into an linear array a. In the right, we show how defining the C macro `#define alpha(i,j) A[ (j)*ldA + (i)]` allows us to address the matrix in a more natural way.

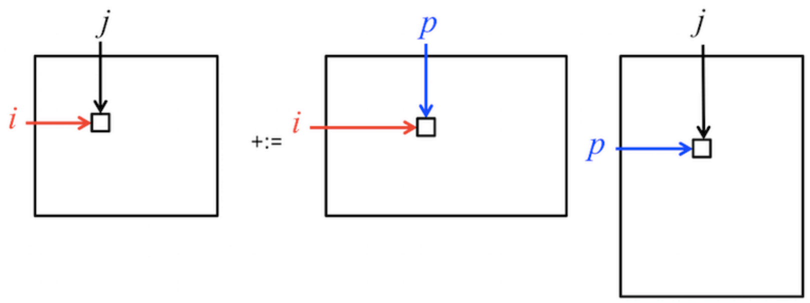


Figure 1.5: We embrace the convention that in loops that implement MMM, the variable i indexes the current row of matrix C (and hence the current row of A), the variable j indexes the current column of matrix C (and hence the current column of B), and variable p indexes the "other" dimension, namely the current column of A (and hence the current row of B). In other literature, the index k is often used instead of the index p. However, it is customary to talk about $m \times n$ matrix C, $m \times k$ matrix A, and $k \times n$ matrix B. Thus, the letter k is "overloaded" and we use p instead.

consists of ldA (the *leading dimension*) rows and some number of columns. If column-major order is used to store the larger matrix, then addressing the elements in the submatrix requires knowledge of the leading dimension of the larger matrix. In the C programming language, if the top-left element of the submatrix is stored at address `A`, then one can address the (i, j) element as `A[ j*ldA + i ]`. We will typically' define a macro that makes addressing the elements more natural:

```
#define alpha(i,j) A[ (j)*ldA + (i) ]
```

where we assume that the variable or constant `ldA` holds the leading dimension parameter.

Homework 1.2.2.1 Exercise on leading dimension of matrices to memory here.
☛ SEE ANSWER

1.2.3 A convention regarding the letter used for the loop index

When we talk about loops for matrix-matrix multiplication (MMM), it helps to keep the picture in Figure 1.5 in mind, which illustrates which loop index (variable name) is used for what row or column of the matrices. We try to be consistent in this use, as should you.

1.2.4 Ordering the loops

Consider again a simple algorithm for computing $C := AB + C$.

```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
    for $p := 0, \ldots, k-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end
```

Given that we now embrace the convention that i indexes rows of C and A, j indexes columns of C and B, and p indexes "the other dimension," we can call this the IJP ordering of the loops around the assignment statement.

The order in which the elements of C are updated, and the order in which terms of

$$\alpha_{i,0}\beta_{0,j} + \cdots + \alpha_{i,k-1}\beta_{k-1,j}$$

are added to $c_{i,j}$ is mathematically equivalent. (There would be a possible change in the result due to round-off errors when floating point arithmetic occurs, but this is ignored.) What this means is that the three loops can be reordered without changing the result[1].

Homework 1.2.4.2 The IJP ordering is one possible ordering of the loops. How many distinct reorderings of those loops are there?

☛ SEE ANSWER

Homework 1.2.4.3 In directory `Assignments/Week1/C/` make copies of ☛ `Gemm_IJP.c` into files with names that reflect the different loop orderings (`Gemm_JIP.c`, etc.). Next, make the necessary changes to loops in each file to reflect the ordering encoded in its name. Test the implementions by executing 'make JIP', etc., for each of the implementations and view the resulting performance by making the indicated changes to the Live Script in ☛ data/Plot_All_Orderings.mlx.

☛ SEE ANSWER

In Figure 1.6, the results of Homework 1.2.4.3 on Robert's laptop are reported. What is obvious is that ordering the loops matters. Changing the order changes how data in memory are accessed, and that can have a considerable impact on the performance that is achieved. The bottom graph includes a plot of the performance of the reference implementation and scales the y-axis so that the top of the graph represents the theoretical peak of a single core of the processor. What it demonstrates is that there is a lot of room for improvement.

1.3 Thinking in terms of vector-vector operations

In this section, learners find out

[1]In exact arithmetic, the result does not change. However, computation is typically performed in floating point arithmetic, in which case roundoff error may accumulate slightly differently, depending on the order of the loops. For more details, see Unit **??**.

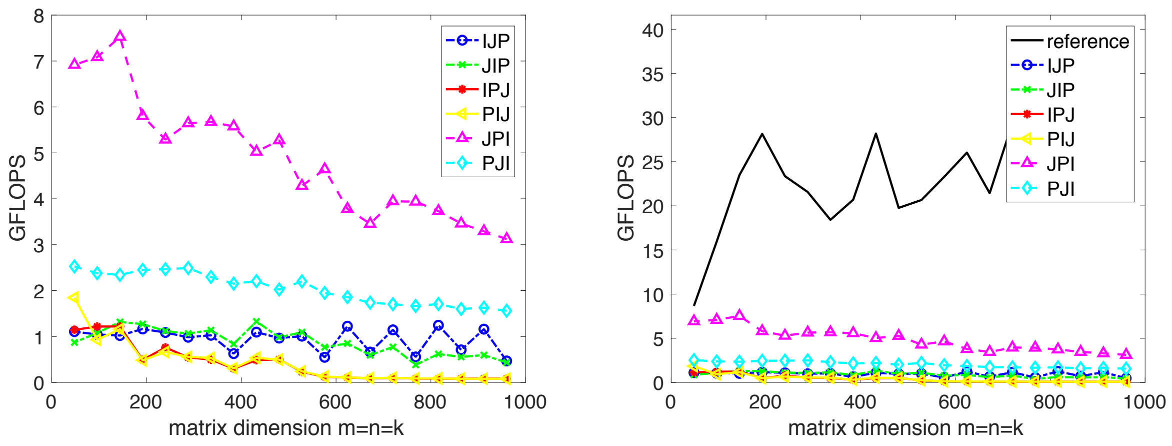


Figure 1.6: Performance comparison of all different orderings of the loops, on Robert's laptop. In the graph on the right, the performance of the reference implementation is added and the top of the graph is chosen to represent the theoretical peak of the processor. This captures that simple implementations aren't magically transformed into high-performance implementations by compilers or architectural features. The most pronounced "zigzagging" of the reference plot is likely due to other tasks being run on the same processor.

- Our notational conventions for matrices and vectors.
- How to think of a matrix in terms of its rows or columns.
- How the inner-most loop of MMM expresses one of two vector-vector operations: the dot product (dot) or the update of a vector bu adding a multiple of another vector (AXPY).
- That vector-vector operations attained high performance on the vector supercomputers of the 1970s. In Week 2, we will see how similarly modern cores attain high performance with vector instructions.
- How to implement vector-vector operations with calls to the Basic Linear Algebra Subprograms (BLAS)
- blah blah blah
- make use of memory access patterns
- Link to high-performance implementations of vector-vector operations.

1.3.1 The Basic Linear Algebra Subprograms (BLAS)

Linear algebra operations are fundamental to computational science. In the 1970s, when vector supercomputers reigned supreme, it was recognized that if applications and software libraries are written in terms of a standardized interface to routines that implement operations with vectors, and vendors of computers then provide high-performance instantiations for that interface, then applications would attain portable high-performance across different computer platforms. This observation yielded the original Basic Linear Algebra Subprograms (BLAS) interface [**?**] for Fortran 77. These original BLAS are now referred to as the *level-1 BLAS*. The interface was expanded in the 1980 to encompass matrix-vector operations (level-2 BLAS) and matrix-matrix operations (level-3 BLAS). We will learn more about the level-2 and level-3 BLAS later in the course.

Expressing code in terms of the BLAS has another benefit: the call to the routine hides the loop that otherwise implements the vector-vector operation and clearly reveals the operation being performed, thus improving readability of the code.

1.3.2 Notation

In our discussions, we use capital letters for matrices ($A, B, C, \ldots$), lower case letters for vectors ($a, b, c, \ldots$), and lower case Greek letters for scalars ($\alpha, \beta, \gamma, \ldots$). Exceptions are integer scalars, for which we will use i, j, k, m, n, and p.

Vectors in our universe are column vectors or, equivalently, $n \times 1$ matrices if the vector has n components (size n). A row vector we view as a column vector that has been transposed. So, x is a column vector and x^T is a row vector.

In the subsequent discussion, we will want to expose the rows or columns of a matrix. If X is an $m \times n$ matrix, then we expose its columns as

$$X = \left(\begin{array}{c|c|c|c} x_0 & x_1 & \cdots & x_{n-1} \end{array} \right)$$

so that x_j equals the column with index j. We expose its rows as

$$X = \left(\begin{array}{c} \widetilde{x}_0^T \\ \widetilde{x}_1^T \\ \vdots \\ \widetilde{x}_{m-1}^T \end{array} \right)$$

so that $\widetilde{x}_i^T$ equals the row with index i. Here the ^{T} indicates it is a row (a column vector that has been transposed). The tilde is added for clarity since x_i^T would in this setting equal the column indexed with i that has been transposed, rather than the row indexed with i. When there isn't a cause for confusion, we will sometimes leave the $\widetilde{}$ off. We use the lower case letter that corresponds to the upper case letter used to denote the matrix, as an added visual clue that x_j is a column of X and $\widetilde{x}_i^T$ is a row of X.

We have already seen that the scalars that constitute the elements of a matrix or vector are denoted with the lower Greek letter that corresponds to the letter used for the matrix of vector:

$$X = \begin{pmatrix} \chi_{0,0} & \chi_{0,1} & \cdots & \chi_{0,n-1} \\ \chi_{1,0} & \chi_{1,1} & \cdots & \chi_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \chi_{m-1,0} & \chi_{m-1,1} & \cdots & \chi_{m-1,n-1} \end{pmatrix} \quad \text{and} \quad x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix}.$$

If you look carefully, you will notice the difference between x and χ. The latter is the lower case Greek letter "chi."

1.3.3 The dot product (inner product)

Given two vectors x and y of size n

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix},$$

their dot product is given by

$$x^T y = \sum_{i=0}^{n-1} \chi_i \psi_i.$$

The notation $x^T y$ comes from the fact that the dot product equals the result of multiplying $1 \times n$ matrix x^T times $n \times 1$ matrix y.

A routine. coded in C, that computes $x^T y + \gamma$ where x and y are stored at location `x` with stride `incx` and location `y` with stride `incy`, respectively, and γ is stored at location `gamma` is given by

```c
#define chi( i ) x[ (i)*incx ]   // map chi( i ) to array x
#define psi( i ) y[ (i)*incy ]   // map psi( i ) to array y

void Dots( int n, double *x, int incx, double *y, int incy, double *gamma )
{
  for ( int i=0; i<n; i++ )
    *gamma += chi( i ) * psi( i );
}
```

in ☛ `Dots.c`. Here stride refers to the number of items in memory between the stored components of the vector. For example, the stride when accessing a row of a matrix, that is stored in column-major order with leading dimension `lda`, is `lda`.

The BLAS include a function for computing the dot operation. Its calling sequence in Fortran, for double precision data, is

```
DDOT( N, X, INCX, Y, INCY )
```

where

- (input) `N` is an integer that equals the size of the vectors.
- (input) `X` is the address where x is stored.
- (input) `INCX` is the stride in memory between entries of x.
- (input) `Y` is the address where y is stored.
- (input) `INCY` is the stride in memory between entries of y.

The function returns the result as a scalar of type double precision. If the datatype were single precision, complex double precision, or complex single precision, then the first `D` is replaced by `S`, `Z`, or `C`, respectively.

To call the same routine in a code written in C, it is important to keep in mind that Fortran passes data by address. The call

```
Dots( n, x, incx, y, incy, &gamma );
```

which, recall, adds the result of the dot product to the value in `gamma`, translates to

```
gamma += ddot_( &n, x, &incx, y, &incy );
```

When one of the strides equals one, as in

```
Dots( n, x, 1, y, incy, &gamma );
```

one has to declare an integer variable (e.g, `i_one`) with value one and pass the address of that variable:

```
int i_one=1;
gamma += ddot_( &n, x, &i_one, y, &incy );
```

We will see examples of this later in this section.

In this course, we use the BLIS implementation of the BLAS as our library. This library also has its own (native) BLAS-like interface. A "quickguide" for this BLIS native interface can be found at

https://github.com/flame/blis/wiki/BLISAPIQuickReference.

There, we find the routine `bli_ddotxv` that computes $\gamma := \alpha x^T y + \beta\gamma$, optionally conjugating the elements of the vectors. The call

```
Dots( n, x, incx, y, incy, &gamma );
```

translates to

```
double one=1.0;

bli_ddotxv( BLIS_NO_CONJUGATE, BLIS_NO_CONJUGATE,
            n, &one, x, incx, y, incy,
               &one, &gamma, NULL );
```

The `BLIS_NO_CONJUGATE` is to indicate that the vectors are *not* to be conjugated. Those parameters are there for consistency with the complex versions of this routine (`bli_zdotv` and `bli_cdotsv`). The, `NULL` at the end corresponds to a parameters that experts can use to pass certain environment information to the routine. For our purposes, passing the `NULL` pointer suffices.

1.3.4 The IJP and JIP orderings

Let us return once again to the IJP ordering of the loops that compute MMM:

```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
    for $p := 0, \ldots, k-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end
```

This pseudo-code translates into the routine coded in C given in Figure 1.1.

Using the notation that we introduced in Unit 1.3.2, one way to think of the above algorithm is to view matrix C as its individual elements, matrix A as its rows, and matrix B as its columns:

$$C = \left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \hline \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \hline \vdots & & \vdots & \\ \hline \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{array}\right), \quad A = \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right), \quad \text{and } B = \left(\begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{n-1} \end{array}\right).$$

We then notice that

$$\left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \hline \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{array}\right)$$

$$
:= \left(\begin{array}{c} \widetilde a_0^T \\ \hline \widetilde a_1^T \\ \hline \vdots \\ \hline \widetilde a_{m-1}^T \end{array} \right) \left(\begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{n-1} \end{array} \right) + \left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \hline \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{array} \right)
$$

$$
= \left(\begin{array}{c|c|c|c} \widetilde a_0^T b_0 + \gamma_{0,0} & \widetilde a_0^T b_1 + \gamma_{0,1} & \cdots & \widetilde a_0^T b_{n-1} + \gamma_{0,n-1} \\ \hline \widetilde a_1^T b_0 + \gamma_{1,0} & \widetilde a_1^T b_1 + \gamma_{1,1} & \cdots & \widetilde a_1^T b_{n-1} + \gamma_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \widetilde a_{m-1}^T b_0 + \gamma_{m-1,0} & \widetilde a_{m-1}^T b_1 + \gamma_{m-1,1} & \cdots & \widetilde a_{m-1}^T b_{n-1} + \gamma_{m-1,n-1} \end{array} \right).
$$

If this makes your head spin, you may want to quickly go through Weeks 3-5 of our MOOC titled "Linear Algebra: Foundations to Fontiers." It captures that the outer two loops visit all of the elements in C, and the inner loop implements the dot product of the appropriate row of A with the appropriate column of B: $\gamma_{i,j} := \widetilde a_i^T b_j + \gamma_{i,j}$, as illustrated by

```
**for** $i := 0, \ldots, m-1$
  **for** $j := 0, \ldots, n-1$
    **for** $p := 0, \ldots, k-1$                    }
        $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   }  $\gamma_{i,j} := \widetilde a_i^T b_j + \gamma_{i,j}$
    **end**                                         }
  **end**
**end**
```

which is, again, the IJP ordering of the loops.

Homework 1.3.4.1 In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IJP.c` into file `Gemm_IJ_Dots.c`. Change `Gemm_IJP` to `Gemm_IJ_Dots` in all places in this new file and then replace the inner-most loop with a call to `Dots`. View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_IJP_JIP.mlx`.

☛ SEE ANSWER

Homework 1.3.4.2 In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IJP.c` into file `Gemm_IJ_ddot.c`. Change `Gemm_IJP` to `Gemm_IJ_ddot` in all places in this new file and then replace the inner-most loop with a call to the BLAS routine `ddot`. View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_IJP_JIP.mlx`.

☛ SEE ANSWER

Homework 1.3.4.3 In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IJP.c` into file `Gemm_IJ_bli_ddotxv.c`. Change `Gemm_IJP` to `Gemm_IJ_bli_ddotxv` in all places in this new file and then replace the inner-most loop with a call to the BLIS routine `bli_ddotxv`. View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_IJP_JIP.mlx`.

☛ SEE ANSWER

Obviously, one can equally well switch the order of the outer two loops, which just means that the elements of C are computed a column at a time rather than a row at a time:

```
for $j := 0, \ldots, n-1$
  for $i := 0, \ldots, m-1$
    for $p := 0, \ldots, k-1$
        $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end
```

$$\left.\begin{array}{l}\textbf{for } p := 0, \ldots, k-1 \\ \quad \gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j} \\ \textbf{end}\end{array}\right\} \quad \gamma_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j}$$

Homework 1.3.4.4 Repeat the last exercises with the implementation in ☛ `Gemm_JIP.c`. In other words, copy this file into files `Gemm_JI_Dots.c`, `Gemm_JI_ddot.c`, and `Gemm_JI_bli_dotsxv.c`. Then, make the necessary changes to these files and view the resulting performance with the Live Script in ☛ `data/Plot_IJP_JIP.mlx`.

☛ SEE ANSWER

In Figure 1.7 we report the performance of the various implementations from the last homeworks. What we notice is that, at least when using Apple's clang compiler, not much difference results from hiding the inner-most loop in a subroutine.

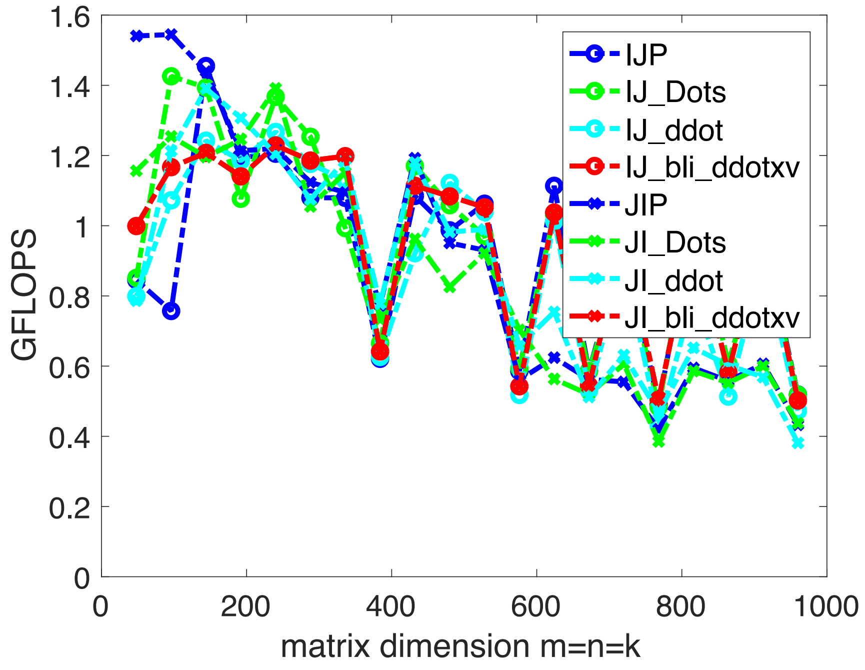
1.3.5 The AXPY operation

Given a scalar, α, and two vectors, x and y, of size n with elements

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix},$$

the scaled vector addition (AXPY) operation is given by

$$y := \alpha x + y$$



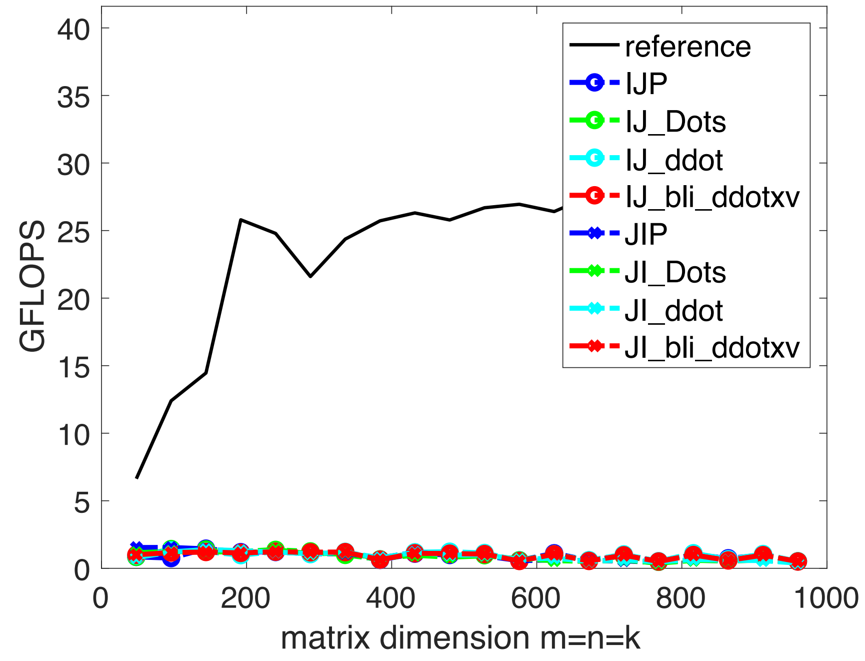


Figure 1.7: Performance of the various implementations of the IJP and JIP orderings. In the graph on the bottom, we include the reference to illustrate there is still a lot of room for improvement.

which in terms of the elements of the vectors equals

$$\begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix} := \alpha \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} + \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix} = \begin{pmatrix} \alpha\chi_0 + \psi_0 \\ \alpha\chi_1 + \psi_1 \\ \vdots \\ \alpha\chi_{n-1} + \psi_{n-1} \end{pmatrix}.$$

The name `axpy` comes from the fact that in Fortran 77 only six characters and numbers could be used to designate the names of variables and functions. The operation $\alpha x + y$ can be read out loud as "scalar Alpha times X Plus Y" which yields the acronym AXPY.

An outline for a routine that implements the AXPY operation is given by

```
#define chi( i ) x[ (i)*incx ]   // map chi( i ) to array x
#define psi( i ) y[ (i)*incy ]   // map psi( i ) to array y

void Axpy( int n, double alpha, double *x, int incx, double *y, int incy )
{
  int i;

  for ( i=0; i<n; i++ )
    psi( i ) =                    ;
}
```

in file ☛ `Axpy.c`.

Homework 1.3.5.1 Complete the routine in `Axpy.c`. You can find the partial implementation in `Axpy.c`. You will test the implementation by using it in subsequent homeworks.

☛ SEE ANSWER

The BLAS include a function for computing the AXPY operation. Its calling sequence in Fortran, for double precision data, is

```
DAXPY( N, ALPHA, X, INCX, Y, INCY )
```

where

- (input) `N` is the address of the integer that equals the size of the vectors.
- (input) `ALPHA` is the address where α is stored.
- (input) `X` is the address where x is stored.
- (input) `INCX` is the address where the stride between entries of x is stored.
- (input/output) `Y` is the address where y is stored.
- (input) `INCY` is the address where the stride between entries of y is stored.

(It may appear strange that the addresses of `N`, `ALPHA`, `INCX`, and `INCY` are passed in. This is because Fortran passes parameters by address rather than value.) If the datatype were single precision, complex double precision, or complex single precision, then the first `D` is replaced by `S`, `Z`, or `C`, respectively.

In C, the call

```
Axpy( n, alpha, x, incx, y, incy );
```

translates to

```
daxpy_( &n, &alpha, x, &incx, y, &incy );
```

which then calls the Fortran interface. Notice that the scalar parameters `n`, `alpha`, `incx`, and `incy` are passed "by address" because Fortran passes parameters to subroutines by address. We will see examples of its use later in this section.

The BLIS native routine for AXPY is `bli_dapyv`. The call

```
Axpy( n, alpha, x, incx, y, incy );
```

translates to

```
bli_daxpyv( BLIS_NO_CONJUGATE, n, alpha, x, incx, y, incy, NULL );
```

The `BLIS_NO_CONJUGATE` is to indicate that vector x is *not* to be conjugated. That parameter is there for consistency with the complex versions of this routine (`bli_zaxpyv` and `bli_caxpyv`). The, `NULL` at the end corresponds to a parameters that experts can use to pass certain environment information to the routine. For our purposes, passing the `NULL` pointer suffices.

1.3.6 The IPJ and PIJ orderings

What we notice is that there are 3! ways of order the loops: Three choices for the outer loop, two for the second loop, and one choice for the final loop. Let's consider the IPJ ordering:

```
for $i := 0, \ldots, m-1$
  for $p := 0, \ldots, k-1$
    for $j := 0, \ldots, n-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end
```

One way to think of the above algorithm is to view matrix C as its rows, matrix A as its individual elements, and matrix B as its rows:

$$C = \left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array} \right), \quad A = \left(\begin{array}{c|c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{array} \right), \quad \text{and} \quad B = \left(\begin{array}{c} \widetilde{b}_0^T \\ \hline \widetilde{b}_1^T \\ \hline \vdots \\ \hline \widetilde{b}_{k-1}^T \end{array} \right).$$

We then notice that

$$\left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) := \left(\begin{array}{c|c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{array}\right) \left(\begin{array}{c} \widetilde{b}_0^T \\ \hline \widetilde{b}_1^T \\ \hline \vdots \\ \hline \widetilde{b}_{k-1}^T \end{array}\right) + \left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right)$$

$$= \left(\begin{array}{ccccccccc} \widetilde{c}_0^T & + & \alpha_{0,0}\widetilde{b}_0^T & + & \alpha_{0,1}\widetilde{b}_1^T & + & \cdots & + & \alpha_{0,k-1}\widetilde{b}_{k-1}^T \\ \hline \widetilde{c}_1^T & + & \alpha_{1,0}\widetilde{b}_0^T & + & \alpha_{1,1}\widetilde{b}_1^T & + & \cdots & + & \alpha_{1,k-1}\widetilde{b}_{k-1}^T \\ \hline & & & & \vdots & & & & \\ \hline \widetilde{c}_{m-1}^T & + & \alpha_{m-1,0}\widetilde{b}_0^T & + & \alpha_{m-1,1}\widetilde{b}_1^T & + & \cdots & + & \alpha_{m-1,k-1}\widetilde{b}_{k-1}^T \end{array}\right).$$

This captures that the outer two loops visit all of the elements in A, and the inner loop implements the updating of the ith row of C by adding $\alpha_{i,p}$ times the pth row of B to it, as captured by

```
**for** $i := 0, \ldots, m-1$
  **for** $p := 0, \ldots, k-1$
    **for** $j := 0, \ldots, n-1$
        $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$          } $\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i$
    **end**
  **end**
**end**
```

Homework 1.3.6.1 In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IPJ.c` into file `Gemm_IP_Axpy.c`. Change `Gemm_IPJ` to `Gemm_IP_Axpy` in all places in this new file and then replace the inner-most loop with a call to `Axpy`. View the resulting performance by making the necessary changes to the Live Script in `data/Plot_IPJ_PIJ.mlx`.

☛ SEE ANSWER

Homework 1.3.6.2 In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IPJ.c` into file `Gemm_IP_daxpy.c`. Change `Gemm_IPJ` to `Gemm_IP_daxpy` in all places in this new file and then replace the inner-most loop with a call to the BLAS routine `daxpy`. View the resulting performance by making the necessary changes to the Live Script in `data/Plot_IPJ_PIJ.mlx`.

☛ SEE ANSWER

Homework 1.3.6.3 In directory `Assignments/Week1/C/` copy file ☛ `Gemm_IPJ.c` into file `Gemm_IP_daxpyv.c`. Change `Gemm_IPJ` to `Gemm_IP_daxpyv` in all places in this new file and then replace the inner-most loop with a call to the BLIS routine `daxpyv`. View the resulting performance by making the necessary changes to the Live Script in `data/Plot_IPJ_PIJ.mlx`.
☛ SEE ANSWER

One can switch the order of the outer two loops to get

```
for $p := 0, \ldots, k-1$
  for $i := 0, \ldots, m-1$
    for $j := 0, \ldots, n-1$
        $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
    } $\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i^T$
  end
end
```

The outer loop in this second algorithm fixes the row of B that is used to to update all rows of C, using the appropriate element from A to scale. In the first iteration of the outer loop ($p = 0$), the following computations occur:

$$\left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) := \left(\begin{array}{c} \widetilde{c}_0^T + \alpha_{0,0}\widehat{b}_0^T \\ \hline \widetilde{c}_1^T + \alpha_{1,0}\widehat{b}_0^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T + \alpha_{m-1,0}\widehat{b}_0^T \end{array}\right).$$

In the second iteration of the outer loop ($p = 1$) it computes

$$\left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) := \left(\begin{array}{c} \widetilde{c}_0^T + \alpha_{0,0}\widehat{b}_0^T + \alpha_{0,1}\widehat{b}_1^T \\ \hline \widetilde{c}_1^T + \alpha_{1,0}\widehat{b}_0^T + \alpha_{1,1}\widehat{b}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T + \alpha_{m-1,0}\widehat{b}_0^T + \alpha_{m-1,1}\widehat{b}_1^T \end{array}\right),$$

and so forth.

Homework 1.3.6.4 Repeat the last exercises with the implementation in ☛ `Gemm_PIJ.c`. In other words, copy this file into files `Gemm_PI_Axpy.c`, `Gemm_PI_daxpy.c`, and `Gemm_PI_bli_axpyv.c`. Then, make the necessary changes to these files and view the resulting performance with the Live Script in ☛ `data/Plot_IJP_JIP.mlx`.
☛ SEE ANSWER

1.3.7 The JPI and PJI orderings

Let us consider the JPI ordering:

```
for $j := 0, \ldots, n-1$
  for $p := 0, \ldots, k-1$
    for $i := 0, \ldots, m-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end
```

One way to think of the above algorithm is to view matrix C as its columns, matrix A as its columns, and matrix B as its individual elements. Then

$$\left(\begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1}\end{array}\right) :=$$

$$\left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{k-1}\end{array}\right)\left(\begin{array}{c|c|c|c} \beta_{0,0} & \beta_{0,1} & \cdots & \beta_{0,n-1} \\ \hline \beta_{1,0} & \beta_{1,1} & \cdots & \beta_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \beta_{k-1,0} & \beta_{k-1,1} & \cdots & \beta_{k-1,n-1}\end{array}\right) + \left(\begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1}\end{array}\right).$$

so that

$$\left(\begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1}\end{array}\right) :=$$
$$\left(\begin{array}{c|c|c|c} c_0 + \beta_{0,0}a_0 + \beta_{1,0}a_1 + \cdots & c_1 + \beta_{0,1}a_0 + \beta_{1,1}a_1 + \cdots & \cdots & c_{n-1} + \beta_{0,n-1}a_0 + \beta_{1,n-1}a_1 + \cdots\end{array}\right).$$

The algorithm captures this as

```
for $j := 0, \ldots, n-1$
  for $p := 0, \ldots, k-1$

    for $i := 0, \ldots, m-1$                    \
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$     }  $c_j := \beta_{p,j} a_p + c_j$
    end                                          /

  end
end
```

Homework 1.3.7.1 In directory `Assignments/Week1/C/` copy file ☛ `Gemm_JPI.c` into file `Gemm_JP_Axpy.c`. Change `Gemm_JPI` to `Gemm_JP_Axpy` in all places in this new file and then replace the inner-most loop with a call to `Axpy`. View the resulting performance by making the necessary changes to the Live Script in `data/Plot_Inner_I.mlx`.

☛ SEE ANSWER

Homework 1.3.7.2 In directory `Assignments/Week1/C/` copy file ☛ `Gemm_JPI.c` into file `Gemm_JP_daxpy.c`. Change `Gemm_JPI` to `Gemm_JP_daxpy` in all places in this new file and then replace the inner-most loop with a call to the BLAS routine `daxpy`. View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_Inner_I.mlx`.
☛ SEE ANSWER

Homework 1.3.7.3 In directory `Assignments/Week1/C/` copy file ☛ `Gemm_JPI.c` into file `Gemm_JP_daxpyv.c`. Change `Gemm_JPI` to `Gemm_JP_daxpyv` in all places in this new file and then replace the inner-most loop with a call to the BLIS routine `daxpyv`. View the resulting performance by making the necessary changes to the Live Script in ☛ `data/Plot_Inner_I.mlx`.
☛ SEE ANSWER

One can switch the order of the outer two loops to get

```
for $p := 0, \ldots, k-1$
  for $j := 0, \ldots, n-1$
    for $i := 0, \ldots, m-1$                   \
        $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   } $c_j := \beta_{p,j} a_p + c_j$
    end                                          /
  end
end
```

The outer loop in this algorithm fixes the column of A that is used to to update all columns of C, using the appropriate element from B to scale. In the first iteration of the outer loop, the following computations occur:

$$\left(c_0 \,\middle|\, c_1 \,\middle|\, \cdots \,\middle|\, c_{n-1} \right) := \left(c_0 + \beta_{0,0} a_0 \,\middle|\, c_1 + \beta_{0,1} a_0 \,\middle|\, \cdots \,\middle|\, c_{n-1} + \beta_{0,n-1} a_0 \right).$$

In the second iteration of the outer loop it computes

$$\left(c_0 \,\middle|\, c_1 \,\middle|\, \cdots \,\middle|\, c_{n-1} \right) := \left(c_0 + \beta_{0,0} a_0 + \beta_{1,0} a_1 \,\middle|\, c_1 + \beta_{0,1} a_0 + \beta_{1,1} a_1 \,\middle|\, \cdots \,\middle|\, c_{n-1} + \beta_{0,n-1} a_0 + \beta_{1,n-1} a_1 \right).$$

and so forth.

Homework 1.3.7.4 Repeat the last exercises with the implementation in ☛ `Gemm_PJI.c`. In other words, copy this file into files `Gemm_PJ_Axpy.c`, `Gemm_PJ_daxpy.c`, and `Gemm_PJ_bli_axpyv.c`. Then, make the necessary changes to these files and view the resulting performance with the Live Script in ☛ `data/Plot_IJP_JIP.mlx`.
☛ SEE ANSWER

1.4 Thinking in terms of matrix-vector operations

1.4.1 Matrix-vector multiplication via dot products or AXPYoperations

Consider the matrix-vector multiplication (MVM)

$$y := Ax + y.$$

The way one is usually taught to compute this operations is that each element of y, ψ_i, is updated with the dot product of the corresponding row of A, $\widetilde{a}_i^T$, with vector x. With our notation, we can describe this as

$$\left(\begin{array}{c} \psi_0 \\ \hline \psi_1 \\ \hline \vdots \\ \hline \psi_{m-1} \end{array}\right) := \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) x + \left(\begin{array}{c} \psi_0 \\ \hline \psi_1 \\ \hline \vdots \\ \hline \psi_{m-1} \end{array}\right) = \left(\begin{array}{c} \widetilde{a}_0^T x + \psi_0 \\ \hline \widetilde{a}_1^T x + \psi_1 \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T x + \psi_{m-1} \end{array}\right).$$

If we then expose the individual elements of A and y we get

$$\begin{array}{rcl} \left(\begin{array}{c} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{array}\right) & := & \left(\begin{array}{c} \alpha_{0,0}\ \chi_0 + \alpha_{0,1}\ \chi_1 + \cdots\ \alpha_{0,n-1}\ \chi_{n-1} + \psi_0 \\ \alpha_{1,0}\ \chi_0 + \alpha_{1,1}\ \chi_1 + \cdots\ \alpha_{1,n-1}\ \chi_{n-1} + \psi_1 \\ \vdots \\ \alpha_{m-1,0}\ \chi_0 + \alpha_{m-1,1}\ \chi_1 + \cdots\ \alpha_{m-1,n-1}\ \chi_{n-1} + \psi_{m-1} \end{array}\right) \\ & = & \left(\begin{array}{c} \chi_0\ \alpha_{0,0} + \chi_1\ \alpha_{0,1} + \cdots\ \chi_{n-1}\ \alpha_{0,n-1} + \psi_0 \\ \chi_0\ \alpha_{1,0} + \chi_1\ \alpha_{1,1} + \cdots\ \chi_{n-1}\ \alpha_{1,n-1} + \psi_1 \\ \vdots \\ \chi_0\ \alpha_{m-1,0} + \chi_1\ \alpha_{m-1,1} + \cdots\ \chi_{n-1}\ \alpha_{m-1,n-1} + \psi_{m-1} \end{array}\right) \end{array}$$

This discussion explains the IJ loop for computing $y := Ax + y$:

```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
      $\psi_i := \alpha_{i,j}\chi_j + \psi_i$        } $\psi_i := \widetilde{a}_j^T x + \psi_i$
  end
end
```

What it demonstrates is how matrix-vector multiplication can be implemented as a sequence of DOT operations.

Homework 1.4.1.1 In directory `Assignments/Week1/C/` complete the implementation of MVM in terms of dot operations

```
#define alpha( i,j ) A[ (j)*ldA + i ]   // map alpha( i,j ) to array A
#define chi( i )  x[ (i)*incx ]         // map chi( i )  to array x
#define psi( i )  y[ (i)*incy ]         // map psi( i )  to array x

void Dots( int, const double *, int, const double *, int, double * );

void Gemv_I_Dots( int m, int n, double *A, int ldA,
                  double *x, int incx, double *y, int incy )
{
  for ( int i=0; i<m; i++ )
    Dots(   ,      ,     ,     ,    ,      );
}
```

in file ☛ `Gemv_I_Dots.c`. You will test this with an implementation of MMM in a later homework.

☛ SEE ANSWER

Next, partitioning $m \times n$ matrix A by columns and x by individual elements, we find that

$$\begin{aligned} y &:= \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1} \end{array} \right) \left(\begin{array}{c} \chi_0 \\ \hline \chi_1 \\ \hline \vdots \\ \hline \chi_{n-1} \end{array} \right) + y \\ &= \chi_0 a_0 + \chi_1 a_1 + \cdots + \chi_{n-1} a_{n-1} + y. \end{aligned}$$

If we then expose the individual elements of A and y we get

$$\begin{aligned} \left(\begin{array}{c} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{array} \right) &:= \chi_0 \left(\begin{array}{c} \alpha_{0,0} \\ \alpha_{1,0} \\ \vdots \\ \alpha_{m-1,0} \end{array} \right) + \chi_1 \left(\begin{array}{c} \alpha_{0,1} \\ \alpha_{1,1} \\ \vdots \\ \alpha_{m-1,1} \end{array} \right) + \cdots + \chi_{n-1} \left(\begin{array}{c} \alpha_{0,n-1} \\ \alpha_{1,n-1} \\ \vdots \\ \alpha_{m-1,n-1} \end{array} \right) + \left(\begin{array}{c} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{array} \right) \\ &= \left(\begin{array}{c} \chi_0 \;\; \alpha_{0,0} \;\; + \chi_1 \;\; \alpha_{0,1} \;\; + \cdots \; \chi_{n-1} \;\; \alpha_{0,n-1} \;\; + \;\; \psi_0 \\ \chi_0 \;\; \alpha_{1,0} \;\; + \chi_1 \;\; \alpha_{1,1} \;\; + \cdots \; \chi_{n-1} \;\; \alpha_{1,n-1} \;\; + \;\; \psi_1 \\ \vdots \\ \chi_0 \; \alpha_{m-1,0} + \chi_1 \; \alpha_{m-1,1} + \cdots \; \chi_{n-1} \; \alpha_{m-1,n-1} + \psi_{m-1} \end{array} \right) \end{aligned}$$

This discussion explains the JI loop for computing $y := Ax + y$:

```
for $j := 0, \ldots, n-1$
  for $i := 0, \ldots, m-1$
    $\psi_i := \alpha_{i,j}\chi_j + \psi_i$        } $y := \chi_j a_j + y$
  end
end
```

What it also demonstrates is how matrix-vector multiplication can be implemented as a sequence of AXPY operations.

Homework 1.4.1.2 In directory `Assignments/Week1/C/` complete the implementation of MVM in terms of AXPY operations

```
#define alpha( i,j ) A[ (j)*ldA + i ]   // map alpha( i,j ) to array A
#define chi( i )  x[ (i)*incx ]         // map chi( i )  to array x
#define psi( i )  y[ (i)*incy ]         // map psi( i )  to array x

void Dots( int, const double *, int, const double *, int, double * );

void Gemv_J_Axpy( int m, int n, double *A, int ldA,
                  double *x, int incx, double *y, int incy )
{
  for ( int j=0; j<n; j++ )
    Axpy(   ,    ,    ,    ,    ,     );
}
```

in file ☛ `Gemv_J_Axpy.c`. You will test this with an implementation of MMM in a later homework.

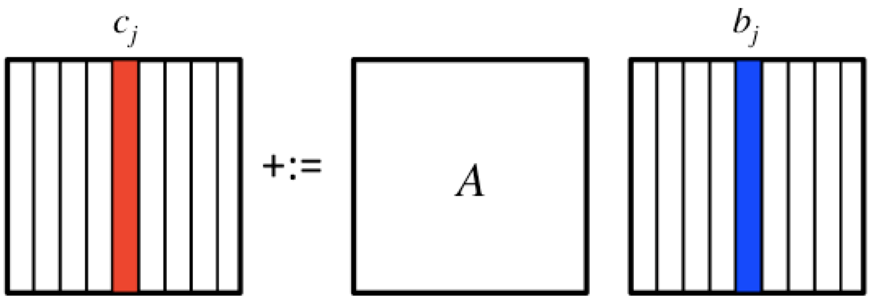
☛ SEE ANSWER

1.4.2 Matrix-matrix multiplication via matrix-vector multiplications

Now that we are getting comfortable with partitioning matrices and vectors, we can view the six algorithms for $C := AB + C$ is a more layered fashion. If we partition C and B by columns, we find that

$$\begin{aligned}\left(\begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1}\end{array}\right) &:= A\left(\begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{n-1}\end{array}\right) + \left(\begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1}\end{array}\right)\\ &= \left(\begin{array}{c|c|c|c} Ab_0 + c_0 & Ab_1 + c_1 & \cdots & Ab_{n-1} + c_{n-1}\end{array}\right)\end{aligned}$$

A picture that captures this is given by



This illustrates how the JIP and JPI algorithms can be viewed as a loop around matrix-vector multiplications:

$$\begin{array}{l}
\textbf{for } j := 0, \ldots, n-1 \\
\quad \left.\begin{array}{l}
\textbf{for } p := 0, \ldots, k-1 \\
\quad \left.\begin{array}{l}
\textbf{for } i := 0, \ldots, m-1 \\
\quad \gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j} \\
\textbf{end}
\end{array}\right\} \; c_j := \beta_{p,j} a_p + c_j \\
\textbf{end}
\end{array}\right\} \; c_j := A b_j + c_j \\
\textbf{end}
\end{array}$$

and

$$\begin{array}{l}
\textbf{for } j := 0, \ldots, n-1 \\
\quad \left.\begin{array}{l}
\textbf{for } i := 0, \ldots, m-1 \\
\quad \left.\begin{array}{l}
\textbf{for } p := 0, \ldots, k-1 \\
\quad \gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j} \\
\textbf{end}
\end{array}\right\} \; \gamma_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j} \\
\textbf{end}
\end{array}\right\} \; c_j := A b_j + c_j \\
\textbf{end}
\end{array}$$

Homework 1.4.2.1 In directory `Assignments/Week1/C/` complete the code in file `Gemm_J_Gemv_I_Dots.c`. View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_J.mlx`.

☛ SEE ANSWER

Homework 1.4.2.2 In directory `Assignments/Week1/C/` complete the code in file `Gemm_J_Gemv_J_Axpy.c`. View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Gemm_J.mlx`.

☛ SEE ANSWER

The BLAS include the routine `dgemv` that computes

$$y := \alpha A x + \beta y \quad \text{or} \quad y := \alpha A^T x + \beta y.$$

If

- $m \times n$ matrix A is stored in array `A` with its leading dimension stored in variable`ldA`,
- m is stored in variable `m` and n is stored in variable `n`,
- vector x is stored in array `x` with its stride stored in variable `incx`,
- vector y is stored in array `y` with its stride stored in variable `incy`, and
- α and β are stored in variables `alpha` and `beta`, respectively,

the $y := \alpha A x + \beta y$ translates, from C, into the call

```
dgemv_( "No transpose", &m, &n, &alpha, A, &ldA, x, &incx, &beta, y, &incy );
```

Homework 1.4.2.3 In directory `Assignments/Week1/C/` complete the code in file `Gemm_J_dgemv.c` that casts MMM in terms of the `dgemv` BLAS routine. View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_J.mlx`.

☛ SEE ANSWER

The BLIS native call that is similar to the BLAS `dgemv` routine in this setting translates to

```
bli_dgemv( BLIS_NO_TRANSPOSE, BLIS_NO_CONJUGATE, m, n, &alpha, A, 1, ldA, x, incx,
                                                              &beta, y, incy );
```

Notice the two parameters after `A`. BLIS requires a row and column stride to be specified for matrices, thus generalizing beyond column-major order storage.

Homework 1.4.2.4 In directory `Assignments/Week1/C/` complete the code in file `Gemm_J_bli_dgemv.c` that casts MMM in terms of the `bli_dgemv` BLIS routine. View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_J.mlx`.

☛ SEE ANSWER

Homework 1.4.2.5 What do you notice from the various performance graphs that result from executing the Live Script in ☛ `Plot_Outer_J.mlx`?

☛ SEE ANSWER

1.4.3 Rank-1 update RANK-1

An operation that is going to become very important in future discussion and optimization of MMM is the rank-1 update:

$$A := xy^T + A.$$

Partitioning $m \times n$ matrix A by columns, and x and y by individual elements, we find that

$$\left(\begin{array}{c|c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{array}\right) :=$$

$$\left(\begin{array}{c} \chi_0 \\ \hline \chi_1 \\ \hline \vdots \\ \hline \chi_{m-1} \end{array}\right) \left(\begin{array}{c|c|c|c} \psi_0 & \psi_1 & \cdots & \psi_{n-1} \end{array}\right) + \left(\begin{array}{c|c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c} \chi_0\psi_0 + \alpha_{0,0} & \chi_0\psi_1 + \alpha_{0,1} & \cdots & \chi_0\psi_{n-1} + \alpha_{0,n-1} \\ \hline \chi_1\psi_0 + \alpha_{1,0} & \chi_1\psi_1 + \alpha_{1,1} & \cdots & \chi_1\psi_{n-1} + \alpha_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \chi_{m-1}\psi_0 + \alpha_{m-1,0} & \chi_{m-1}\psi_1 + \alpha_{m-1,1} & \cdots & \chi_{m-1}\psi_{n-1} + \alpha_{m-1,n-1} \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c} \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix}\psi_0 + \begin{pmatrix} \alpha_{0,0} \\ \alpha_{1,0} \\ \vdots \\ \alpha_{m-1,0} \end{pmatrix} & \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix}\psi_1 + \begin{pmatrix} \alpha_{0,1} \\ \alpha_{1,1} \\ \vdots \\ \alpha_{m-1,1} \end{pmatrix} & \cdots & \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix}\psi_{n-1} + \begin{pmatrix} \alpha_{0,n-1} \\ \alpha_{1,n-1} \\ \vdots \\ \alpha_{m-1,n-1} \end{pmatrix} \end{array}\right).$$

What this illustrates is that we could have partitioned A by columns and y by elements to find that

$$\begin{aligned} \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1} \end{array}\right) &:= x\left(\begin{array}{c|c|c|c} \psi_0 & \psi_1 & \cdots & \psi_{n-1} \end{array}\right) + \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1} \end{array}\right) \\ &= \left(\begin{array}{c|c|c|c} x\psi_0 + a_0 & x\psi_1 + a_1 & \cdots & x\psi_{n-1} + a_{n-1} \end{array}\right) \\ &= \left(\begin{array}{c|c|c|c} \psi_0 x + a_0 & \psi_1 x + a_1 & \cdots & \psi_{n-1} x + a_{n-1} \end{array}\right). \end{aligned}$$

This discussion explains the JI loop ordering for computing $A := xy^T + A$:

```
for $j := 0, \ldots, n-1$
  for $i := 0, \ldots, m-1$
    $\alpha_{i,j} := \chi_i \psi_j + \alpha_{i,j}$      } $a_j := \psi_i x + a_j$
  end
end
```

What it also demonstrates is how the rank-1 operation can be implemented as a sequence of AXPY operations.

Homework 1.4.3.1 In directory `Assignments/Week1/C/` complete the implementation of RANK-1 in terms of AXPY operations in file ☛ `Ger_J_Axpy.c`. You will test this with an implementation of MMM in a later homework.

☛ SEE ANSWER

Notice that there is also an IJ loop ordering that can be explained by partitioning A by rows and x by elements:

$$\left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) := \left(\begin{array}{c} \chi_0 \\ \hline \chi_1 \\ \hline \vdots \\ \hline \chi_{m-1} \end{array}\right) y^T + \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) = \left(\begin{array}{c} \chi_0 y^T + \widetilde{a}_0^T \\ \hline \chi_1 y^T + \widetilde{a}_1^T \\ \hline \vdots \\ \hline \chi_{m-1} y^T + \widetilde{a}_{m-1}^T \end{array}\right)$$

leading to the algorithm

```
for $i := 0, \ldots, n-1$
  for $j := 0, \ldots, m-1$
      $\alpha_{i,j} := \chi_i \psi_j + \alpha_{i,j}$        } $\widetilde{a}_i^T := \chi_i y^T + \widetilde{a}_i^T$
  end
end
```

and corresponding implementation.

Homework 1.4.3.2 In directory `Assignments/Week1/C/` complete the implementation of RANK-1 in terms of AXPY operations in file ☛ `Ger_I_Axpy.c`. You will test this with an implementation of MMM in a later homework.

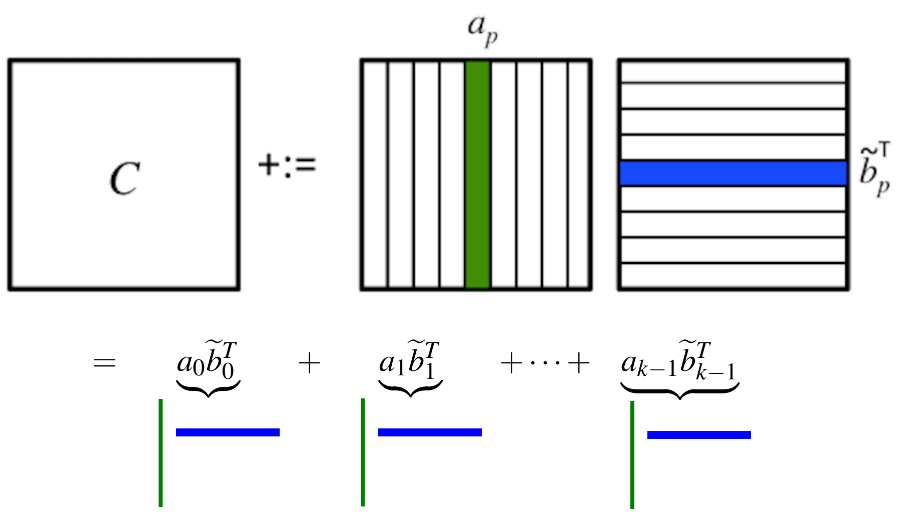
☛ SEE ANSWER

1.4.4 Matrix-matrix multiplication via rank-1 updates

Next, let us partition A by columns and B by rows, so that

$$\begin{array}{rcl} C & := & \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{k-1} \end{array}\right) \left(\begin{array}{c} \widetilde{b}_0^T \\ \hline \widetilde{b}_1^T \\ \hline \vdots \\ \hline \widetilde{b}_{k-1}^T \end{array}\right) + C \\ & = & a_0 \widetilde{b}_0^T + a_1 \widetilde{b}_1^T + \cdots | a_{k-1} \widetilde{b}_{k-1}^T + C \end{array}$$

A picture that captures this is given by


a_p
C
$+:=$
$\widetilde{b}_p^T$
$= \underbrace{a_0 \widetilde{b}_0^T} + \underbrace{a_1 \widetilde{b}_1^T} + \cdots + \underbrace{a_{k-1} \widetilde{b}_{k-1}^T}$

This illustrates how the PJI and PIJ algorithms can be viewed as a loop around matrix-vector multiplications:

```
for $p := 0, \ldots, k-1$
  for $j := 0, \ldots, n-1$
      for $i := 0, \ldots, m-1$                          \
         $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   } $c_j := \beta_{p,j} a_p + c_j$     } $C := a_p \widetilde{b}_p^T + C$
      end                                              /
    end
end
```

and

```
for $p := 0, \ldots, k-1$
  for $i := 0, \ldots, m-1$
      for $j := 0, \ldots, n-1$                          \
         $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$   } $\widetilde{c}_i^T := \alpha_{i,p} \widetilde{b}_p^T + \widetilde{c}_i^T$     } $C := a_p \widetilde{b}_p^T + C$
      end                                              /
    end
end
```

Homework 1.4.4.1 In directory `Assignments/Week1/C/` complete the code in file `Gemm_P_Ger_J_Axpy.c`. View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_P.mlx`.

☛ SEE ANSWER

Homework 1.4.4.2 In directory `Assignments/Week1/C/` complete the code in file `Gemm_P_Ger_I_Axpy.c`. View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_P.mlx`.

☛ SEE ANSWER

The BLAS include the routine `dger` that computes

$$A := \alpha x y^T + A$$

If

- $m \times n$ matrix A is stored in array `A` with its leading dimension stored in variable`ldA`,
- m is stored in variable `m` and n is stored in variable `n`,
- vector x is stored in array `x` with its stride stored in variable `incx`,
- vector y is stored in array `y` with its stride stored in variable `incy`, and
- α is stored in variable `alpha`,

the $A := \alpha x y^T + A$ translates, from C, into the call

```
dger_( &m, &n, &alpha, x, &incx, &beta, y, &incy, A, &ldA );
```

Homework 1.4.4.3 In directory `Assignments/Week1/C/` complete the code in file `Gemm_P_dger.c` that casts MMM in terms of the `dger` BLAS routine. View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_P.mlx`.

☛ SEE ANSWER

The BLIS native call that is similar to the BLAS `dger` routine in this setting translates to

```
bli_dger( BLIS_NO_CONJUGATE, BLIS_NO_CONJUGATE, m, n, &alpha, x, incx,
          &beta, y, incy, A, 1, ldA );
```

Again, notice the two parameters after `A`. BLIS requires a row and column stride to be specified for matrices, thus generalizing beyond column-major order storage.

Homework 1.4.4.4 In directory `Assignments/Week1/C/` complete the code in file `Gemm_P_bli_dger.c` that casts MMM in terms of the `bli_dger` BLIS routine. View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_P.mlx`.

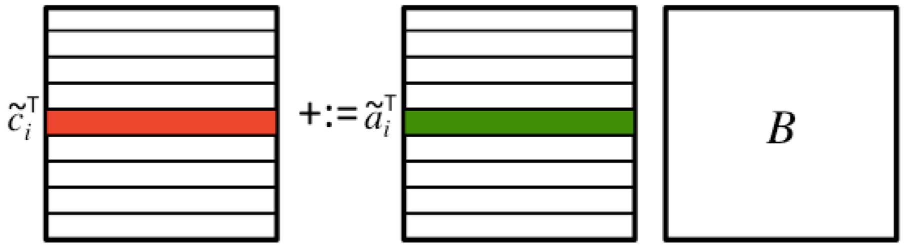
☛ SEE ANSWER

1.4.5 Matrix-matrix multiplication via row-times-matrix multiplications

Finally, let us partition C and A by rows so that

$$\left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) := \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) B + \left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) = \left(\begin{array}{c} \widetilde{a}_0^T B + \widetilde{c}_0^T \\ \hline \widetilde{a}_1^T B + \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T B + \widetilde{c}_{m-1}^T \end{array}\right)$$

A picture that captures this is given by



This illustrates how the IJP and IPJ algorithms can be viewed as a loop around the updating of a row of C with the product of the corresponding row of A times matrix B:

```
for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
      for $p := 0, \ldots, k-1$
          $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
      end
      } $\widetilde{\gamma}_{i,j} := \widetilde{a}_i^T b_j + \widetilde{\gamma}_{i,j}$
  end
  } $\widetilde{c}_i^T := \widetilde{a}_i^T B + \widetilde{c}_i^T$
end
```

and

```
for $i := 0, \ldots, m-1$
  for $p := 0, \ldots, k-1$
      for $j := 0, \ldots, n-1$
          $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
      end
      } $\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i^T$
  end
  } $\widetilde{c}_i^T := \widetilde{a}_i^T B + \widetilde{c}_i^T$
end
```

The problem with implementing the above algorithms is that the GEMV implementations `Gemv_I_Dots` and `Gemv_J_Axpy` implement $y := Ax + y$ rather than $y^T := x^T A + y^T$. Obviously, you could create new routines for this new operation. We will instead look at how to use the BLAS and BLIS interfaces.

Recall that the BLAS include the routine `dgemv` that computes

$$y := \alpha A x + \beta y \quad \text{or} \quad y := \alpha A^T x + \beta y.$$

What we want is a routine that computes

$$y^T := x^T A + y^T$$

What we remember from linear algebra is that if $A = B$ then $A^T = B^T$, that $(A + B)^T = A^T B^T$, and that $(AB)^T = B^T A^T$. Thus, starting with the the equality

$$y^T = x^T A + y^T,$$

and transposing both sides, we get that

$$(y^T)^T = (x^T A + y^T)^T$$

which is equivalent to

$$y = (x^T A)^T + (y^T)^T$$

and finally

$$y = A^T x + y.$$

So, updating

$$y^T := x^T A + y^T$$

is equivalent to updating

$$y := A^T x + y.$$

It this all seems unfamiliar, you may want to look at ☛ Unit 3.2.5 of our MOOC titled "Linear Algebra: Foundations to Frontiers."

Now, if

- $m \times n$ matrix A is stored in array `A` with its leading dimension stored in variable`ldA`,
- m is stored in variable `m` and n is stored in variable `n`,
- vector x is stored in array `x` with its stride stored in variable `incx`,
- vector y is stored in array `y` with its stride stored in variable `incy`, and
- α and β are stored in variables `alpha` and `beta`, respectively,

then $y := \alpha A^T x + \beta y$ translates, from C, into the call

```
dgemv_( "Transpose", &m, &n, &alpha, A, &ldA, x, &incx, &beta, y, &incy );
```

Homework 1.4.5.1 In directory `Assignments/Week1/C/` complete the code in file `Gemm_I_dgemv.c` that casts MMM in terms of the `dgemv` BLAS routine, but compute the result by rows. View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_I.mlx`.

☛ SEE ANSWER

The BLIS native call that is similar to the BLAS `dgemv` routine in this setting translates to

```
bli_dgemv( BLIS_TRANSPOSE, BLIS_NO_CONJUGATE, m, n, &alpha, A, ldA, 1,
                                              x, incx, &beta, y, incy );
```

Because of the two parameters after `A` that capture the stride between elements in a column (the row stride) and elements in rows (the column stride), one can alternatively swap these parameters:

```
bli_dgemv( BLIS_NO_TRANSPOSE, BLIS_NO_CONJUGATE, m, n, &alpha, A, ldA, 1,
                                                 x, incx, &beta, y, incy );
```

Homework 1.4.5.2 In directory `Assignments/Week1/C/` complete the code in file `Gemm_I_bli_dgemv.c` that casts MMM in terms of the `bli_dgemv` BLIS routine. View the resulting performance by making the necessary changes to the Live Script in ☛ `Plot_Outer_I.mlx`.

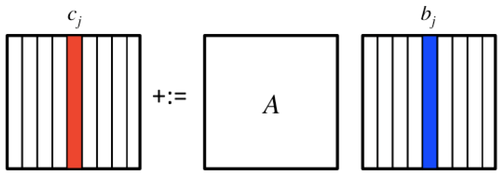
☛ SEE ANSWER

1.4.6 Discussion

The point of this section is that one can think of matrix-matrix multiplication as one loop around

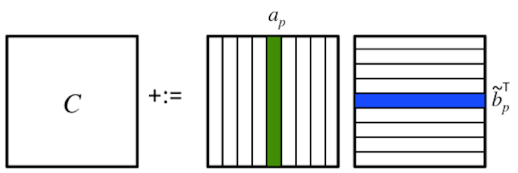
- multiple matrix-vector multiplications:

$$\left(c_0 \middle| \cdots \middle| c_{n-1} \right) = \left(Ab_0 + c_0 \middle| \cdots \middle| Ab_{n-1} + c_{n-1} \right)$$



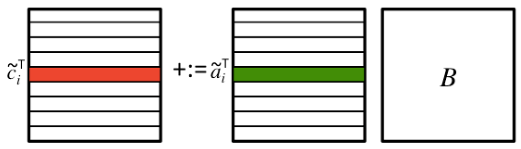
- multiple rank-1 updates:

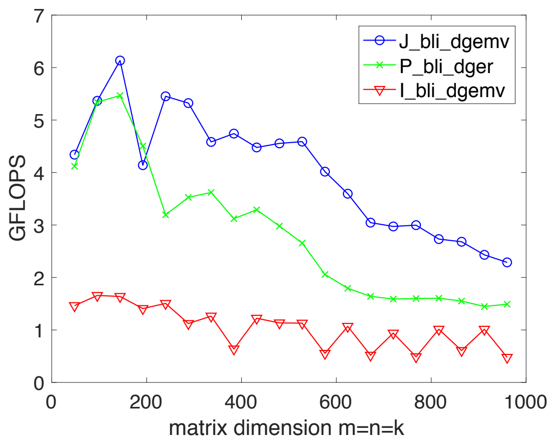
$$C = a_0 \widetilde{b}_0^T + a_1 \widetilde{b}_1^T + \cdots + a_{k-1} \widetilde{b}_{k-1}^T + C$$



- multiple row times matrix multiplications:

$$\begin{pmatrix} \widetilde{c}_0^T \\ \hline \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{pmatrix} = \begin{pmatrix} \widetilde{a}_0^T B + \widetilde{c}_0^T \\ \hline \widetilde{a}_1^T B + \widetilde{c}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T B + \widetilde{c}_{m-1}^T \end{pmatrix}$$





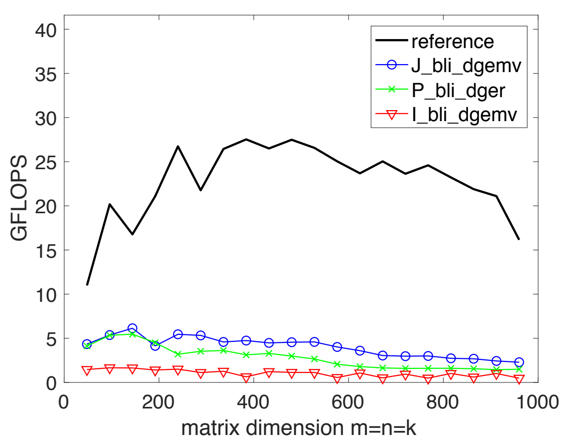


Figure 1.8: Performance for different choices of the outer loop, calling the BLIS typed interface, on Robert's laptop.

So, for the outer loop there are three choices: j, p, or i. This may give the appearance that there are only three algorithms. However, the matrix-vector multiply and rank-1 update hide a double loop and the order of these two loops can be chosen, to bring us back to six choices for algorithms. Importantly, matrix-vector multiplication can be organized so that matrices are addressed by columns in the inner-most loop (the JI order for computing GEMV) as can the rank-1 update (the JI order for computing GER). For the implementation that picks the I loop as the outer loop, GEMV is utilized but with the transposed matrix. As a result, there is no way of casting the inner two loops in terms of operations with columns.

Homework 1.4.6.1 In directory `Assignments/Week1/C/` use the Live Script in ☛ `Plot_Summary.mlx` to compare and contrast the performance of algorithms that use the I, J, and P loop as the outer-most loop.
What do you notice?

☛ SEE ANSWER

Homework 1.4.6.2 Homework that matches loops with partitioned (FLAME) notation?

☛ SEE ANSWER

From the performance experiments reported in Figure 1.8, we have observed that accessing matrices by columns, so that the most frequently loaded data is contiguous in memory, yields better performance. Still, the observed performance is much worse than can be achieved.

We have seen many examples of blocked matrix-matrix multiplication already, where matrices were partitioned by rows, columns, or individual entries. This will be explored and exploited further next week.

1.5 Enrichment

1.5.1 Counting flops

Floating point multiplies or adds are examples of floating point operation (flops). What we have noticed is that for all of our computations (DOTS, AXPY, GEMV, GER, and GEMM) every floating point multiply is paired with a floating point add into a fused multiply-add (FMA).

Determining how many floating point operations are required for the different operations is relatively straight forward: If x and y are of size n, then

- $\gamma := x^T y + \gamma = \chi_0\psi_0 + \chi_1\psi_1 + \cdots + \chi_{n-1}\psi_{n-1} + \gamma$ requires n FMAs and hence $2n$ flops.
- $y := \alpha x + y$ requires n FMAs (one per pair of element, one from from x and one from y) and hence $2n$ flops.

Similarly, it is pretty easy to establish that if A is $m \times n$, then

- $y := Ax + y$ requires mn FMAs and hence $2mn$ flops.
  n AXPY operations each of size m for $n \times 2m$ flops or m ... operations each of size n for $m \times 2n$ flops.
- $A := xy^T + A$ required mn FMAs and hence $2mn$ flops.
  n AXPY operations each of size m for $n \times 2m$ flops or m AXPY operations each of size n for $m \times 2n$ flops.

Finally, if C is $m \times n$, A is $m \times k$, and B is $k \times n$, then $C := AB + C$ requires $2mnk$ flops. We can estiblish this by recognizing that if C is updated one column at a time, this takes n GEMV operations each with a matrix of size $m \times k$, for a total of $n \times 2mk$. Alternativesly, if C is updated with rank-1 updates, then we get $k \times 2mn$.

When we run experiments, we tend to execute with matrices that are $n \times n$, where n ranges from small to large. Thus, the total operations required equal

$$\sum_{n=\mathtt{n_{first}}}^{\mathtt{n_{last}}} 2n^3 \text{ flops,}$$

where $n_{\mathtt{first}}$ is the first problem size, $n_{\mathtt{last}}$ the last, and the summation is in increments of $n_{\mathtt{inc}}$.

If $\mathtt{n_{last}} = \mathtt{n_{inc}} \times N$ and $n_{\mathtt{inc}} = n_{\mathtt{first}}$, then we get

$$\sum_{n=n_{\mathtt{first}}}^{n_{\mathtt{last}}} 2n^3 = \sum_{i=1}^{N} 2(i \times n_{\mathtt{inc}})^3 = 2n_{\mathtt{inc}}^3 \sum_{i=1}^{N} i^3.$$

A standard trick is to recognize that

$$\sum_{i=1}^{N} i^3 \approx \int_0^N x^3 dx = \frac{1}{4}N^4.$$

So,

$$\sum_{n=n_{\texttt{first}}}^{n_{\texttt{last}}} 2n^3 \approx \frac{1}{2} n_{\texttt{inc}}^3 N^4 = \frac{1}{2} \frac{n_{\texttt{inc}}^4 N^4}{n_{\texttt{ninc}}} = \frac{1}{2n_{\texttt{ninc}}} n^4.$$

The important thing is that every time we double $n_{\texttt{last}}$, we have to wait, approximately, sixteen times as long for our experiments to finish...

An interesting question is why we count flops rather than FMAs. By now, you have noticed we like to report the rate of computation in billions of floating point operations per second, GFLOPS. Now, if we counted FMAs rather than flops, the number that represents the rate of computation would be half a large. For marketing purposes, bigger is better and hence flops are reported!

1.6 Wrapup

1.6.1 Additional exercises

1.6.2 Summary

Week 2

Start Your Engines!

2.1 Opener

2.1.1 Launch

The current plan is to discuss the memory hierarchy and how MMM allows data movement to be amortized.

The inconvenient truth is that floating point computations can be performed very fast while bringing data in from main memory is relatively slow. How slow? On a typical architecture it takes two orders of magnitude more time to bring a floating point number in from main memory than it takes to compute with it.

The reason why main memory is slow is relatively simple: there is not enough room on a chip for the large memories that we are accustomed to, and hence they are off chip. The mere distance creates a latency for retrieving the data. This could then be offset by retrieving a lot of data simultaneously. Unfortunately there are inherent bandwidth limitations: there are only so many pins that can connect the central processing unit with main memory.

To overcome this limitation, a modern processor has a hierarchy of memories. We have already encountered the two extremes: registers and main memory. In between, there are smaller but faster *cache memories*. These cache memories are on-chip and hence do not carry the same latency as does main memory and also can achieve greater bandwidth. The hierarchical nature of these memories is often depicted as a pyramid as illustrated in Figure 2.1. To put things in perspective: We have discussed that the Haswell processor has sixteen vector processors that can store 64 double precision floating point numbers (doubles). Close to the CPU it has a 32Kbytes level-1 cache (L1 cache) that can thus store 4,096 doubles. Somewhat further it has a 256Kbytes level-2 cache (L2 cache) that can store 32,768 doubles. Further away yet, but still on chip, it has a 8Mbytes level-3 cache (L3 cache) that can hold 524,288 doubles.

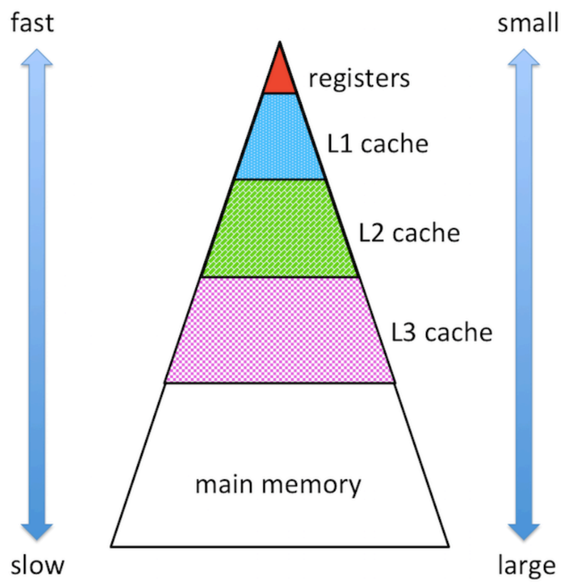


Figure 2.1: Illustration of the memory hierarchy.

In the below discussion, we will pretend that one can place data in a specific cache and keep it there for the duration of computations. In fact, caches retain data using some replacement policy that evicts data that has not been recently used. By carefully ordering computations, we can encourage data to remain in cache, which is what happens in practice.

2.1.2 Outline Week 2

2.1.3 What you will learn

2.2 Using registers and vector instructions

2.2.1 Blocked MMM

If the material in this section makes you scratch your head, you may want to go through the materials in Week 5 of our other MOOC: ☛ Linear Algebra: Foundations to Frontiers.

Key to understanding how we are going to optimize MMM is to understand blocked MMM (the multiplication of matrices that have been partitioned into submatrices).

Consider $C := AB + C$. In terms of the elements of the matrices, this means that each $\gamma_{i,j}$ is computed by

$$\gamma_{i,j} := \sum_{p=0}^{k-1} \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}.$$

Now, partition the matrices into submatrices:

$$C = \left(\begin{array}{c|c|c|c} C_{0,0} & C_{0,1} & \cdots & C_{0,N-1} \\ \hline C_{1,0} & C_{1,1} & \cdots & C_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline C_{M-1,0} & C_{M,1} & \cdots & C_{M-1,N-1} \end{array}\right), A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,K-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,K-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{M-1,0} & A_{M,1} & \cdots & A_{M-1,K-1} \end{array}\right),$$

and

$$B = \left(\begin{array}{c|c|c|c} B_{0,0} & B_{0,1} & \cdots & B_{0,N-1} \\ \hline B_{1,0} & B_{1,1} & \cdots & B_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline B_{K-1,0} & B_{K,1} & \cdots & B_{K-1,N-1} \end{array}\right),$$

where

- $C_{i,j}$ is $m_i \times n_j$,
- $A_{i,p}$ is $m_i \times k_p$, and
- $B_{p,j}$ is $k_p \times n_j$

with
$\sum_{i=0}^{M-1} m_i = m$, $\sum_{j=0}^{N-1} n_i = m$, and $\sum_{p=0}^{K-1} k_i = m$. Then

$$C_{i,j} := \sum_{p=0}^{K-1} A_{i,p}B_{p,j} + C_{i,j}.$$

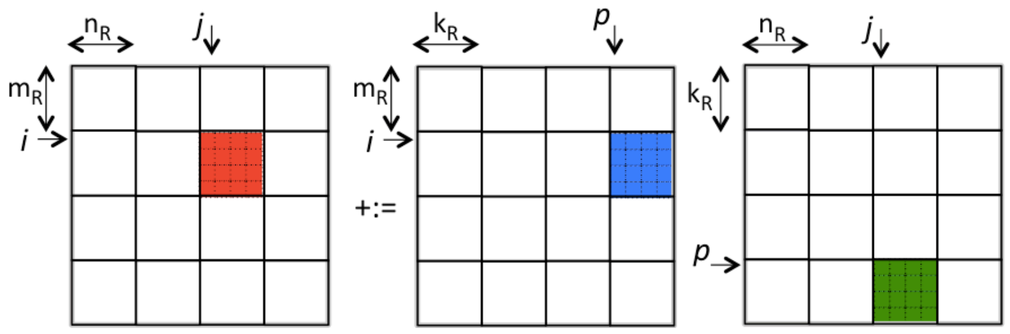


Figure 2.2: An illustration of a blocked algorithm where $m_R = n_R = k_R = 4$.

```
void Gemm_IJP_MRxNRxKR( int m, int n, int k, double *A, int ldA,
                              double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
    for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
      for ( int p=0; p<k; p+=KR ) /* k is assumed to be a multiple of KR */
        Gemm_MRxNRxKR( &alpha( i,p ), ldA, &beta( p,j ), ldB, &gamma( i,j ), ldC );
}

void Gemm_MRxNRxKR( double *A, int ldA, double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<MR; i++ )
    for ( int j=0; j<NR; j++ )
      for ( int p=0; p<KR; p++ )
        gamma( i,j ) += alpha( i,p ) * beta( p,j );
}
```

Assignments/Week2/C/Gemm_IJP_MRxNRxKR.c

Figure 2.3: Simple blocked algorithm that calls a kernel that updates a $m_R \times n_R$ submatrix of C with the result of multiplying a $m_R \times k_R$ submatrix of A times a $k_R \times n_R$ submatrix of B . Obviously, it only works when m, n, and k are multiples of m_R, n_R, and k_R, respectively.

In other words, provided the partitioning of the matrices is "conformal," MMM with partitioning matrices proceeds exactly as does MMM with the elements of the matrices *except* that the individual multiplications with the submatrices do not necessarily commute.

We illustrate how blocking C into $m_R \times n_R$ submatrices, A into $m_R \times n_R$ submatrices, and B into $k_R \times n_R$ submatrices in Figure 2.2. An implementation of one such algorithm is given in Figure 2.3.

Homework 2.2.1.1 In directory `Assignments/Week2/C/` time the code in file ☛ Gemm_IJP_MRxNRxKR.c by executing `make test_IJP_MRxNRxKR`. View its performance with ☛ Plot_MRxNR_Kernels.mlx.

☛ SEE ANSWER

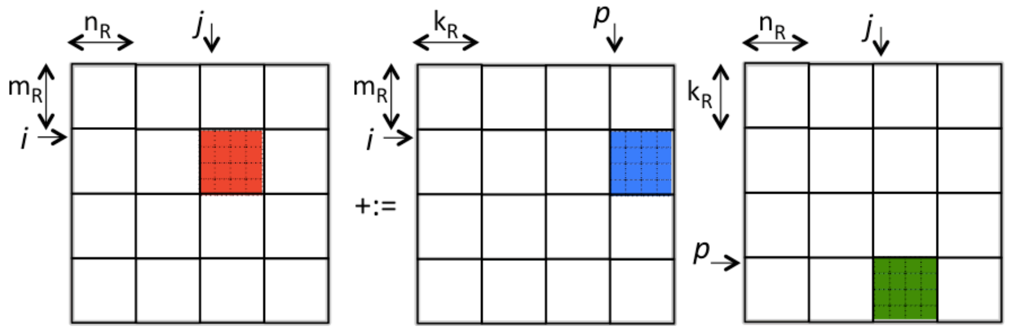
2.2.2 Computing with matrices in registers

For computation to happen, input data must be brought from memory into registers and output data written back to memory. For now let us make the following assumptions:

- Our processor has only one core.
- That core only has two levels of memory: registers and main memory.
- Moving data between main memory and registers takes time β per double.
- The registers can hold 64 doubles.
- Performing a flop with data in registers takes time γ_R.
- Data movement and computation cannot overlap.

Later, we will change some of these assumptions.

Given that the registers can hold 64 doubles, let's first consider blocking C, A, and B into 4×4 submatrices. In other words, consider the example from the last unit, given in Figure 2.2, with $m_R = n_R = k_R = 4$. This means $3 \times 4 \times 4 = 48$ registers are being used for storing the submatrices. (Yes, we are not using all registers. We'll get to that later.) Let us call the routine that computes with three blocks "the kernel". Thus, the blocked algorithm is a triple-nested loop around a call to this kernel:



Here is a naive algorithm, that exposes the loading of submatrices into registers:

```
for $i := 0, \ldots, M-1$
  for $j := 0, \ldots, N-1$
    for $p := 0, \ldots, K-1$
      Load $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$
      $C_{i,j} := A_{i,p} B_{p,j} + C_{i,j}$
      Store $C_{i,j}$
    end
  end
end
```

The time required to complete this algorithm can be estimated as

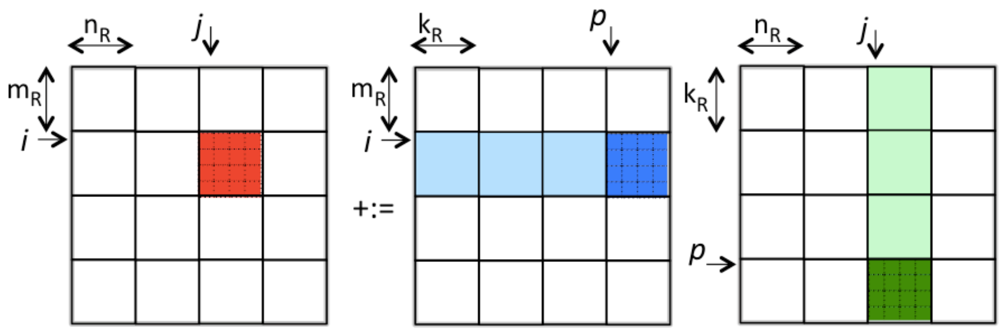
$$\begin{aligned} MNK(2m_R n_R k_R)\gamma_R + MNK(2m_R n_R + m_R k_R + k_R n_R)\beta \\ = \quad 2mnk\gamma_R + mnk(\frac{2}{k_R} + \frac{1}{n_R} + \frac{1}{m_R})\beta, \end{aligned}$$

since $M = m/m_R$, $N = n/n_R$, and $K = k/k_R$,

We next recognize that since the loop indexed with p is the inner-most loop, the loading and storing of $C_{i,j}$ does not need to happen before every call to the kernel, yielding the algorithm

```
for $i := 0, \ldots, M-1$
  for $j := 0, \ldots, N-1$
    Load $C_{i,j}$
    for $p := 0, \ldots, K-1$
      Load $A_{i,p}$, and $B_{p,j}$
      $C_{i,j} := A_{i,p} B_{p,j} + C_{i,j}$
    end
    Store $C_{i,j}$
  end
end
```

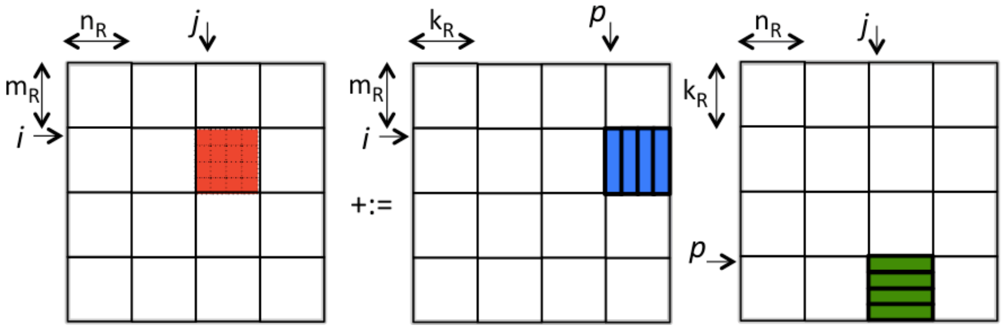
as illustrated by



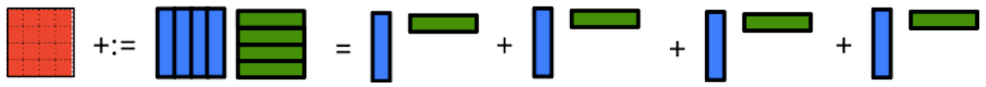
Homework 2.2.2.1 Compute the estimated cost of the last algorithm.

☛ SEE ANSWER

Now, if the submatrix $C_{i,j}$ were larger (i.e., m_R and n_R were larger), then the overhead can be reduced. Obviously, we can use more registers (we are only storing 48 of a possible 64 doubles in registers when $m_R = n_R = k_R = 4$). However, let's first discuss how to require fewer than 48 doubles to be stored while keeping $m_R = n_R = k_R = 4$. Recall from Week 1 that if the P loop of MMM is the outer-most loop, then the inner two loops implement a rank-1 update. If we do this for the kernel, then the result is illustrated by

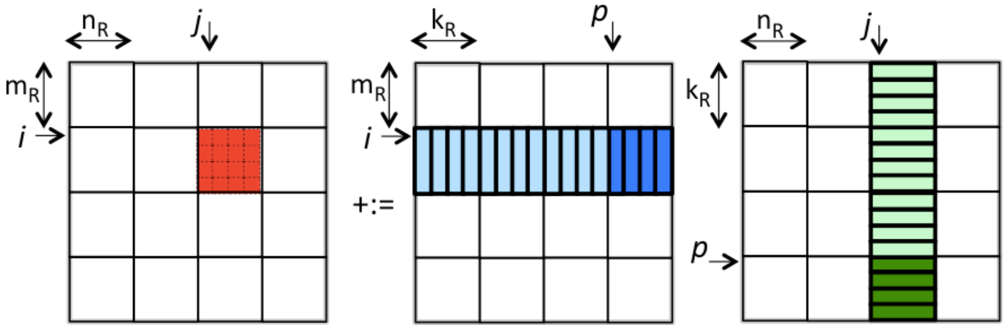


If we now consider the computation



We recognize that after each rank-1 update, the column of $A_{i,p}$ and row of $B_{p,j}$ that participate in that rank-1 update are not needed again in the computation $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$. Thus, only room for one such column of $A_{i,p}$ and one such row of $B_{p,j}$ needs to be reserved in the registers, reducing the total use of registers to $m_R \times n_R + m_R + n_R$. If $m_R = n_R = 4$ this equals $16 + 4 + 4 = 24$ doubles. That means in turn that, later, m_R and n_R can be chosen to be larger than if the entire blocks of A and B were stored.

If we now view the entire computation performed by the loop indexed with p, this can be illustrated by



and implemented, in terms of a call to the BLIS rank-1 update routine, in Figure 2.4.

What we now recognize is that matrices A and B need not be partitioned into $m_R \times k_R$ and $k_R \times n_R$ submatrices (respectively). Instead, A can be partitioned into $m_R \times k$ row panels and B into $k \times n_R$ column panels:

```c
void Gemm_IJP_P_bli_dger( int m, int n, int k, double *A, int ldA,
        double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
    for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
      for ( int p=0; p<k; p+=KR ) /* k is assumed to be a multiple of KR */
  Gemm_P_bli_dger( MR, NR, KR, &alpha( i,p ), ldA,
       &beta( p,j ), ldB, &gamma( i,j ), ldC );
}

void Gemm_P_bli_dger( int m, int n, int k, double *A, int ldA,
          double *B, int ldB, double *C, int ldC )
{
  double d_one = 1.0;

  for ( int p=0; p<k; p++ )
    bli_dger( BLIS_NO_CONJUGATE, BLIS_NO_CONJUGATE, m, n, &d_one, &alpha( 0,p ), 1,
              &beta( p,0 ), ldB, C, 1, ldC, NULL );
}
```

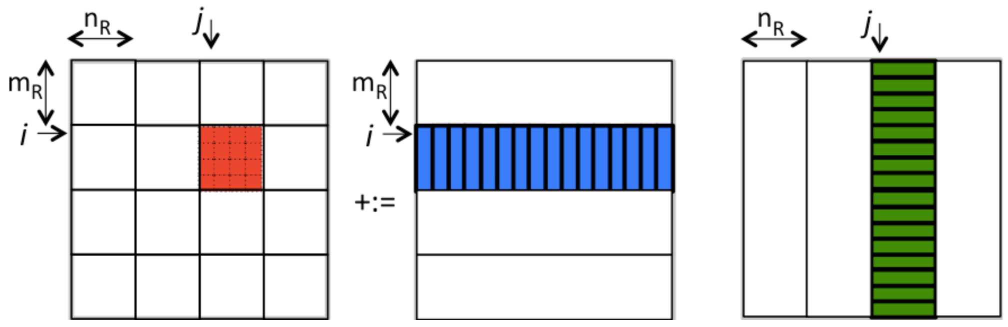
Assignments/Week2/C/Gemm_IJP_P_bli_dger.c

Figure 2.4: Blocked matrix-matrix multiplication with kernel that casts $C_{i,j} = A_{i,p}B_{p,j} + C_{i,j}$ in terms of rank-1 updates.

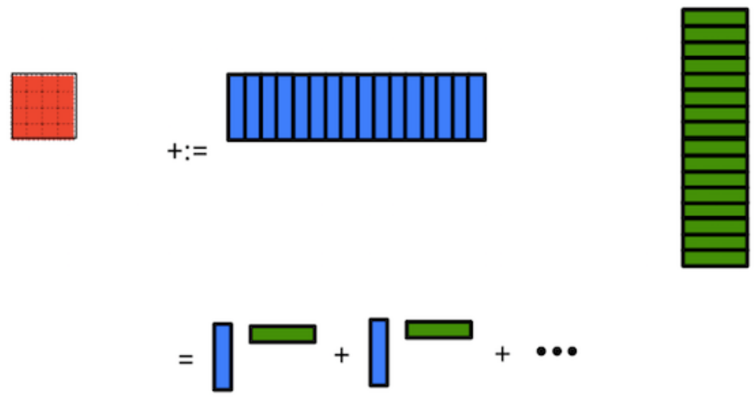
```c
void Gemm_IJ_P_bli_dger( int m, int n, int k, double *A, int ldA,
                                       double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
    for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
      Gemm_P_bli_dger( MR, NR, k, &alpha( i,p ), ldA,
           &beta( 0,j ), ldB, &gamma( i,j ), ldC );
}

void Gemm_P_bli_dger( int m, int n, int k, double *A, int ldA,
          double *B, int ldB, double *C, int ldC )
{
  double d_one = 1.0;

  for ( int p=0; p<k; p++ )
    bli_dger( BLIS_NO_CONJUGATE, BLIS_NO_CONJUGATE, m, n, &d_one, &alpha( 0,p ), 1,
              &beta( p,0 ), ldB, C, 1, ldC, NULL );
}
```

Assignments/Week2/C/Gemm_IJ_P_bli_dger.c

Figure 2.5: Blocked matrix-matrix multiplication that partitions A into row panels and B into column panels and casts the kernel in terms of rank-1 updates.



Another way of looking at this is that the inner loop indexed with p in the blocked algorithm can be merged with the outer loop of the kernel. The entire update of the $m_R \times n_R$ submatrix of C can then be pictured as



This is the computation that we will call the *micro-kernel* from here on.

Bringing $A_{i,p}$ one column at a time into registers and $B_{p,j}$ one row at a time into registers does not change the cost of reading that data. So, the cost of the resulting approach is still estimated as

$$\begin{aligned} & MNK(2m_R n_R k_R)\gamma_R + [MN(2m_R n_R) + MNK(m_R k_R + k_R n_R)]\,\beta \\ = \;& 2mnk\gamma_R + \left[2mn + mnk(\frac{1}{n_R} + \frac{1}{m_R})\right]\beta. \end{aligned}$$

We can arrive at the same algorithm by partitioning

$$C = \left(\begin{array}{c|c|c|c} C_{0,0} & C_{0,1} & \cdots & C_{0,N-1} \\ \hline C_{1,0} & C_{1,1} & \cdots & C_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline C_{M-1,0} & C_{M,1} & \cdots & C_{M-1,N-1} \end{array}\right), A = \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline \vdots \\ \hline A_{M-1} \end{array}\right), \text{ and } B = \left(\begin{array}{c|c|c|c} B_0 & B_1 & \cdots & B_{0,N-1} \end{array}\right),$$

where $C_{i,j}$ is $m_R \times n_R$, A_i is $m_R \times k$, and B_j is $k \times n_R$. Then computing $C := AB + C$ means updating $C_{i,j} := A_i B_j + C_{i,j}$ for all i, j:

```
for $i := 0, \ldots, M-1$
  for $j := 0, \ldots, N-1$
    $C_{i,j} := A_i B_j + C_{i,j}$        } Computed with the micro-kernel
  end
end
```

Obviously, the order of the two loops can be switched. Again, the computation $C_{i,j} := A_i B_j + C_{i,j}$ where $C_{i,j}$ fits in registers now becomes what we will call the *micro-kernel*.

2.2.3 Of vector registers and instructions, and instruction-level parallelism

As the reader should have noticed by now, in MMM for every floating point multiplication a corresponding floating point addition is encountered to accumulate the result. For this reason, such floating point computations are usually cast in terms of *fused multiply add* (FMA) operations, performed by a floating point unit (FPU) of the core.

What is faster than computing one FMA at a time? Computing multiple FMAs at a time! To accelerate computation, modern cores are equipped with vector registers, vector instructions, and vector floating point units that can simultaneously perform multiple FMA operations. This is called *instruction-level parallelism*.

In this section, we are going to use Intel's *vector intrinsic instructions* for Intel's AVX instruction set to illustrate the ideas. Vector intrinsics are supported in the C programming language for performing these vector instructions. We illustrate how the intrinsic operations are used by incorporating them into an implemention of the micro-kernel for the case where $m_R \times n_R = 4 \times 4$. Thus, for the remainder of this unit, C is $m_R \times n_R = 4 \times 4$, A is $m_R \times k = 4 \times k$, and B is $k \times n_R = k \times 4$.

Now, $C+ := AB$ when C is 4×4 translates to the computation

$$
\left(\begin{array}{c|c|c|c}
\gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\
\gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\
\gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\
\gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3}
\end{array}\right)
$$

$$
+:= \left(\begin{array}{c|c|c}
\alpha_{0,0} & \color{red}{\alpha_{0,1}} & \cdots \\
\alpha_{1,0} & \color{red}{\alpha_{1,1}} & \cdots \\
\alpha_{2,0} & \color{red}{\alpha_{2,1}} & \cdots \\
\alpha_{3,0} & \color{red}{\alpha_{3,1}} & \cdots
\end{array}\right)
\left(\begin{array}{cccc}
\beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} \\
\beta_{1,0} & \color{blue}{\beta_{1,1}} & \beta_{1,2} & \beta_{1,3} \\
\vdots & \vdots & \vdots & \vdots
\end{array}\right)
$$

$$
= \left(\begin{array}{c|c|c|c}
\alpha_{0,0}\beta_{0,0} + \color{red}{\alpha_{0,1}}\beta_{1,0} + \cdots & \alpha_{0,0}\beta_{0,1} + \color{red}{\alpha_{0,1}}\color{blue}{\beta_{1,1}} + \cdots & \alpha_{0,0}\beta_{0,2} + \color{red}{\alpha_{0,1}}\beta_{1,2} + \cdots & \alpha_{0,0}\beta_{0,2} + \color{red}{\alpha_{0,1}}\beta_{1,2} + \cdots \\
\alpha_{1,0}\beta_{0,0} + \color{red}{\alpha_{1,1}}\beta_{1,0} + \cdots & \alpha_{1,0}\beta_{0,1} + \color{red}{\alpha_{1,1}}\color{blue}{\beta_{1,1}} + \cdots & \alpha_{1,0}\beta_{0,2} + \color{red}{\alpha_{1,1}}\beta_{1,2} + \cdots & \alpha_{1,0}\beta_{0,3} + \color{red}{\alpha_{1,1}}\beta_{1,3} + \cdots \\
\alpha_{2,0}\beta_{0,0} + \color{red}{\alpha_{2,1}}\beta_{1,0} + \cdots & \alpha_{2,0}\beta_{0,1} + \color{red}{\alpha_{2,1}}\color{blue}{\beta_{1,1}} + \cdots & \alpha_{2,0}\beta_{0,2} + \color{red}{\alpha_{2,1}}\beta_{1,2} + \cdots & \alpha_{2,0}\beta_{0,3} + \color{red}{\alpha_{2,1}}\beta_{1,3} + \cdots \\
\alpha_{3,0}\beta_{0,0} + \color{red}{\alpha_{3,1}}\beta_{1,0} + \cdots & \alpha_{3,0}\beta_{0,1} + \color{red}{\alpha_{3,1}}\color{blue}{\beta_{1,1}} + \cdots & \alpha_{3,0}\beta_{0,2} + \color{red}{\alpha_{3,1}}\beta_{1,2} + \cdots & \alpha_{3,0}\beta_{0,3} + \color{red}{\alpha_{3,1}}\beta_{1,3} + \cdots
\end{array}\right)
$$

$$= \begin{pmatrix} \alpha_{0,0} \\ \alpha_{1,0} \\ \alpha_{2,0} \\ \alpha_{3,0} \end{pmatrix} \begin{pmatrix} \beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} \end{pmatrix} + \begin{pmatrix} \alpha_{0,1} \\ \alpha_{1,1} \\ \alpha_{2,1} \\ \alpha_{3,1} \end{pmatrix} \begin{pmatrix} \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} \end{pmatrix} + \cdots$$

Thus, updating 4×4 matrix C can be implemented as a loop around rank-1 updates:

```
for $p = 0, \ldots, k-1$
    $\left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{array}\right) +:= \begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix} \begin{pmatrix} \beta_{p,0} & \beta_{p,1} & \beta_{p,2} & \beta_{p,3} \end{pmatrix}$
endfor
```

or, equivalently to emphasize computations with vectors,

```
for $p = 0, \ldots, k-1$
    $\left(\begin{array}{c|c|c|c} \gamma_{0,0} +:= \alpha_{0,p} \times \beta_{p,0} & \gamma_{0,1} +:= \alpha_{0,p} \times \beta_{p,1} & \gamma_{0,2} +:= \alpha_{0,p} \times \beta_{p,2} & \gamma_{0,3} +:= \alpha_{0,p} \times \beta_{p,3} \\ \gamma_{1,0} +:= \alpha_{1,p} \times \beta_{p,0} & \gamma_{1,1} +:= \alpha_{1,p} \times \beta_{p,1} & \gamma_{1,2} +:= \alpha_{1,p} \times \beta_{p,2} & \gamma_{1,3} +:= \alpha_{1,p} \times \beta_{p,3} \\ \gamma_{2,0} +:= \alpha_{2,p} \times \beta_{p,0} & \gamma_{2,1} +:= \alpha_{2,p} \times \beta_{p,1} & \gamma_{2,2} +:= \alpha_{2,p} \times \beta_{p,2} & \gamma_{2,3} +:= \alpha_{2,p} \times \beta_{p,3} \\ \gamma_{3,0} +:= \alpha_{3,p} \times \beta_{p,0} & \gamma_{3,1} +:= \alpha_{3,p} \times \beta_{p,1} & \gamma_{3,2} +:= \alpha_{3,p} \times \beta_{p,2} & \gamma_{3,3} +:= \alpha_{3,p} \times \beta_{p,3} \end{array}\right)$
endfor
```

This, once again, shows how MMM can be cast in terms of a sequence of rank-1 updates, and that a rank-1 update can be implemented in terms of AXPY operations with columns of C.

Now we translate this into code that employs vector instructions, illustrated in Figure 2.8 and given in Figures 2.6-2.7. To use the intrinsics, we start by including the header file `immintrin.h`.

```
#include<immintrin.h>
```

The declarations

```
  __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
```

creates `gamma_0123_0` as a variable that references a vector register with four double precision numbers and loads it with the four numbers that are stored starting at address `&gamma( 0,0 )`. In other words, it loads the vector register with the original values

$$\begin{pmatrix} \gamma_{0,0} \\ \gamma_{1,0} \\ \gamma_{2,0} \\ \gamma_{3,0} \end{pmatrix}.$$

This is repeated for the other three columns of C:

```c
#include <stdio.h>
#include <stdlib.h>

#define alpha( i,j ) A[ (j)*ldA + (i) ]   // map alpha( i,j ) to array A
#define beta( i,j )  B[ (j)*ldB + (i) ]   // map beta( i,j ) to array B
#define gamma( i,j ) C[ (j)*ldC + (i) ]   // map gamma( i,j ) to array C

#define MR 4
#define NR 4

void Gemm_IJ_4x4Kernel( int, int, int, double *, int, double *, int, double *, int );

void Gemm_4x4Kernel( int, double *, int, double *, int, double *, int );

void GemmWrapper( int m, int n, int k, double *A, int ldA,
      double *B, int ldB, double *C, int ldC )
{
  if ( m % MR != 0 || n % NR != 0 ){
    printf( "m and n must be multiples of MR and NR, respectively \n" );
    exit( 0 );
  }

  Gemm_IJ_4x4Kernel( m, n, k, A, ldA, B, ldB, C, ldC );
}

void Gemm_IJ_4x4Kernel( int m, int n, int k, double *A, int ldA,
                                  double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
    for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
      Gemm_4x4Kernel( k, &alpha( i,0 ), ldA, &beta( 0,j ), ldB, &gamma( i,j ), ldC );
}
```

Assignments/Week2/C/Gemm_IJ_4x4Kernel.c

Figure 2.6: General framework for calling a kernel, instantiated for the case where $m_R = n_R = 4$. (Continued in Figure 2.7.)

```c
#include<immintrin.h>

void Gemm_4x4Kernel( int k, double *A, int ldA, double *B, int ldB,
    double *C, int ldC )
{
  /* Declare vector registers to hold 4x4 C and load them */
  __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
  __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );
  __m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );
  __m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );

  for ( int p=0; p<k; p++ ){
    /* Declare vector register for load/broadcasting beta( b,j ) */
    __m256d beta_p_j;

    /* Declare a vector register to hold the current column of A and load
       it with the four elements of that column. */
    __m256d alpha_0123_p = _mm256_loadu_pd( &alpha( 0,p ) );

    /* Load/broadcast beta( p,0 ). */
    beta_p_j = _mm256_broadcast_sd( &beta( p, 0) );

    /* update the first column of C with the current column of A times
       beta ( p,0 ) */
    gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );

    /* REPEAT for second, third, and fourth columns of C.  Notice that the
       current column of A needs not be reloaded. */
  }

  /* Store the updated results */
  _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
  _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
  _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
  _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
}
```

Assignments/Week2/C/Gemm_IJ_4x4Kernel.c

Figure 2.7: Continued: General framework for calling a kernel, instantiated for the case where $m_R = n_R = 4$.)

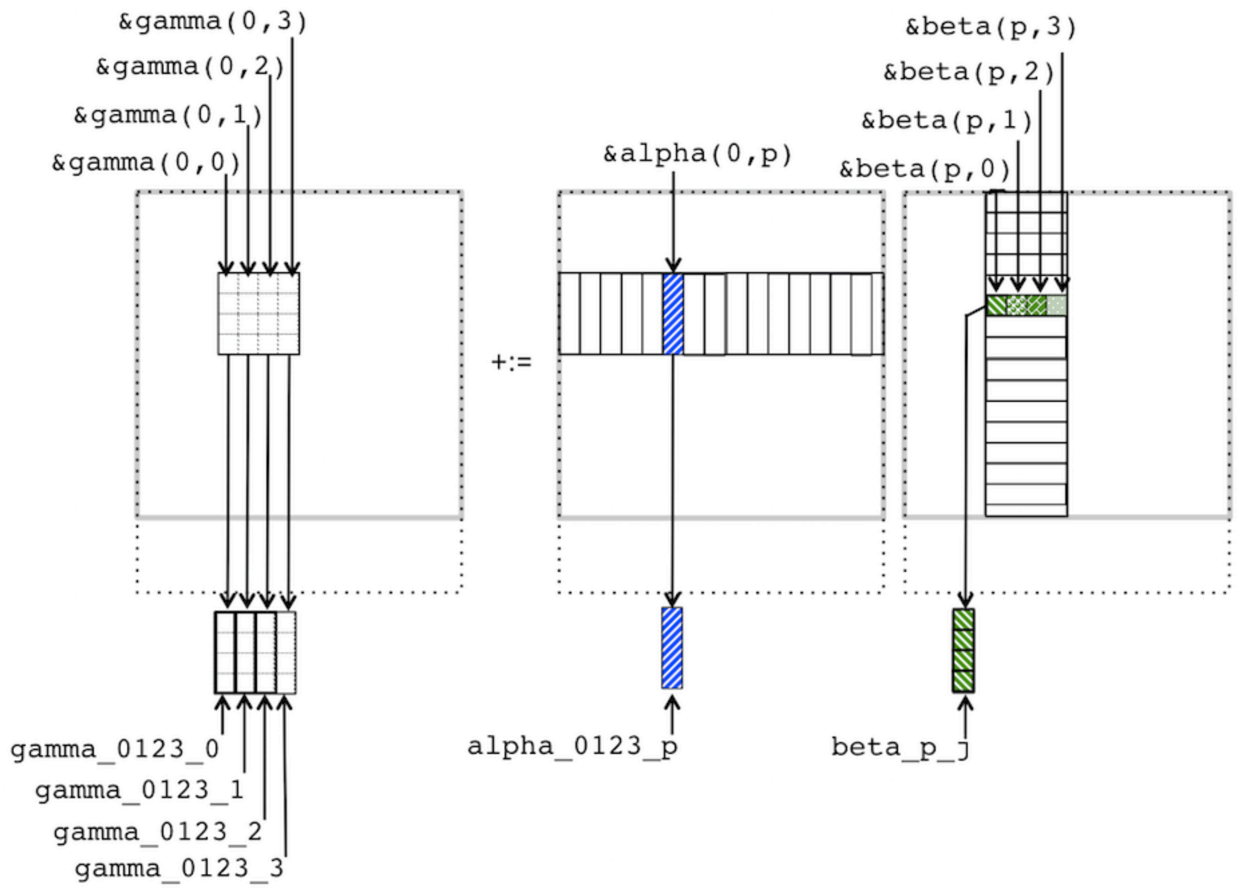


Figure 2.8: Illustration of how the routine in Figure 2.7 indexes into the matrices.

```
  __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );
  __m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );
  __m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );
```

The loop in Figure 2.7 implements

$$
\begin{array}{l}
\textbf{for } p = 0, \ldots, k-1 \\
\left(\begin{array}{c|c|c|c}
\gamma_{0,0} +:= \alpha_{0,p} \times \beta_{p,0} & \gamma_{0,1} +:= \alpha_{0,p} \times \beta_{p,1} & \gamma_{0,2} +:= \alpha_{0,p} \times \beta_{p,2} & \gamma_{0,3} +:= \alpha_{0,p} \times \beta_{p,3} \\
\gamma_{1,0} +:= \alpha_{1,p} \times \beta_{p,0} & \gamma_{1,1} +:= \alpha_{1,p} \times \beta_{p,1} & \gamma_{1,2} +:= \alpha_{1,p} \times \beta_{p,2} & \gamma_{1,3} +:= \alpha_{1,p} \times \beta_{p,3} \\
\gamma_{2,0} +:= \alpha_{2,p} \times \beta_{p,0} & \gamma_{2,1} +:= \alpha_{2,p} \times \beta_{p,1} & \gamma_{2,2} +:= \alpha_{2,p} \times \beta_{p,2} & \gamma_{2,3} +:= \alpha_{2,p} \times \beta_{p,3} \\
\gamma_{3,0} +:= \alpha_{3,p} \times \beta_{p,0} & \gamma_{3,1} +:= \alpha_{3,p} \times \beta_{p,1} & \gamma_{3,2} +:= \alpha_{3,p} \times \beta_{p,2} & \gamma_{3,3} +:= \alpha_{3,p} \times \beta_{p,3}
\end{array} \right) \\
\textbf{endfor}
\end{array}
$$

leaving the result in the vector registers. Each iteration starts by declaring vector register variable `alpha_0123_p` and loading it with the contents of

$$\begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix}.$$

```
    __m256d alpha_0123_p = _mm256_loadu_pd( &alpha( 0,p ) );
```

Next, $\beta_{p,0}$ is loaded into a vector register, broadcasting (duplicating) that value to each entry in that register:

```
    beta_p_j = _mm256_broadcast_sd( &beta( p, 0) );
```

The command

```
    gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );
```

then performs the computation

$$\begin{pmatrix} \gamma_{0,0} +:= \alpha_{0,p} \times \beta_{p,0} \\ \gamma_{1,0} +:= \alpha_{1,p} \times \beta_{p,0} \\ \gamma_{2,0} +:= \alpha_{2,p} \times \beta_{p,0} \\ \gamma_{3,0} +:= \alpha_{3,p} \times \beta_{p,0} \end{pmatrix}.$$

We leave it to the reader to add the commands that compute

$$\begin{pmatrix} \gamma_{0,1} +:= \alpha_{0,p} \times \beta_{p,1} \\ \gamma_{1,1} +:= \alpha_{1,p} \times \beta_{p,1} \\ \gamma_{2,1} +:= \alpha_{2,p} \times \beta_{p,1} \\ \gamma_{3,1} +:= \alpha_{3,p} \times \beta_{p,1} \end{pmatrix}, \quad \begin{pmatrix} \gamma_{0,2} +:= \alpha_{0,p} \times \beta_{p,2} \\ \gamma_{1,2} +:= \alpha_{1,p} \times \beta_{p,2} \\ \gamma_{2,2} +:= \alpha_{2,p} \times \beta_{p,2} \\ \gamma_{3,2} +:= \alpha_{3,p} \times \beta_{p,2} \end{pmatrix}, \quad \text{and} \quad \begin{pmatrix} \gamma_{0,3} +:= \alpha_{0,p} \times \beta_{p,3} \\ \gamma_{1,3} +:= \alpha_{1,p} \times \beta_{p,3} \\ \gamma_{2,3} +:= \alpha_{2,p} \times \beta_{p,3} \\ \gamma_{3,3} +:= \alpha_{3,p} \times \beta_{p,3} \end{pmatrix}.$$

Upon completion of the loop, the results are stored back into the original arrays with the commands

```
  _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
  _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
  _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
  _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
```

Homework 2.2.3.1 In directory `Assignments/Week2/C/` complete the code in file ☛ `Gemm_IJ_4x4Kernel.c`. View its performance with ☛ `data/Plot_MRxNR_Kernels.mlx`.

☛ SEE ANSWER

Homework 2.2.3.2 In directory `Assignments/Week2/C/` copy file ☛`Gemm_IJ_4x4Kernel.c` to `Gemm_JI_4x4Kernel.c` and reverse the I and J loops. View its performance with ☛ `data/Plot_MRxNR_Kernels.mlx`.

☛ SEE ANSWER

The form of parallelism illustrated here is often referred to as Single Instruction, Multiple Data (SIMD) parallelism. The same (FMA) instruction is performed with all data in the vector registers.

On Robert's laptop, we are starting to see some progress towards high performance, as illustrated in Figure 2.9.

2.3 Optimizing the Microkernel

2.3.1 Reuse of data in registers

In Section 2.2.3, we introduced the fact that modern architectures have vector registers and then showed how to (somewhat) optimize the update of a 4×4 submatrix of C.

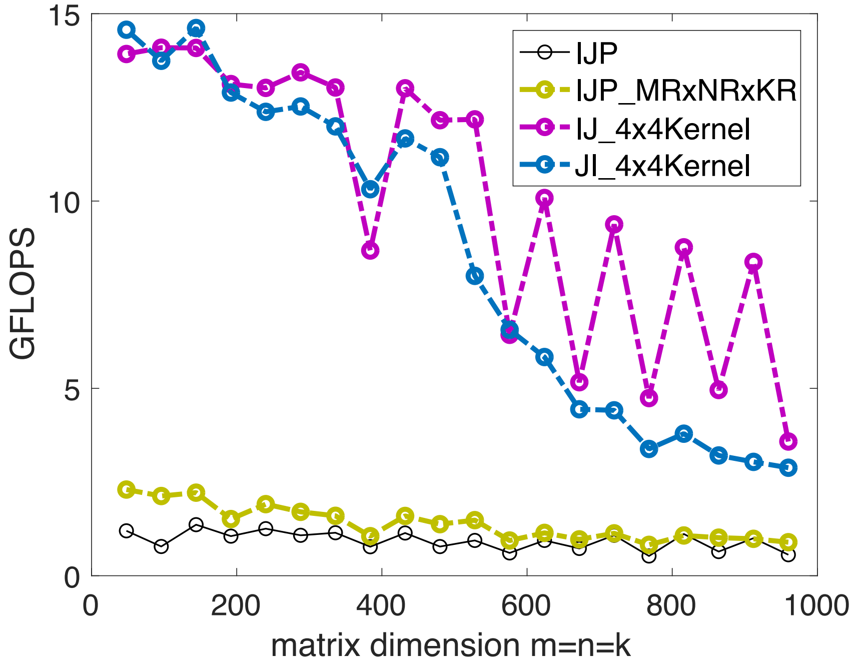
Some observations:

- Four vector registers are used for the submatrix of C. Once loaded, those values remain in the registers for the duration of the computation, until they are finally written out once they have been completely updated. It is important to keep values of C in registers that long, because values of C are read as well as written, unlike values from A and B, which are read but not written.

- The block is 4×4. In general, the block is $m_R \times n_R$, where in this case $m_R = n_R = 4$. If we assume we will use R_C registers for elements of the submatrix of C, what should m_R and n_R be, given the constraint that $m_R \times n_R \leq R_C$?

- Given that some registers need to be used for elements of A and B, how many registers should be used for elements of each of the matrices? In our implementation in the last section, we needed one vector register for four elements of A and one vector register in which one element of B is broadcast (duplicated). Thus, $16 + 4 + 1$ elements of matrices were being stored in 6 out of 16 available vector registers.

What we notice is that, ignoring the cost of loading and storing elements of C, loading four doubles from matrix A and broadcasting four doubles from matrix B is amortized over 32 flops. The ratio is 32 flops for 8 loads, or 4 flops/load. Here for now we ignore the fact that loading the four elements of A is less costly than broadcasting the four elements of B.



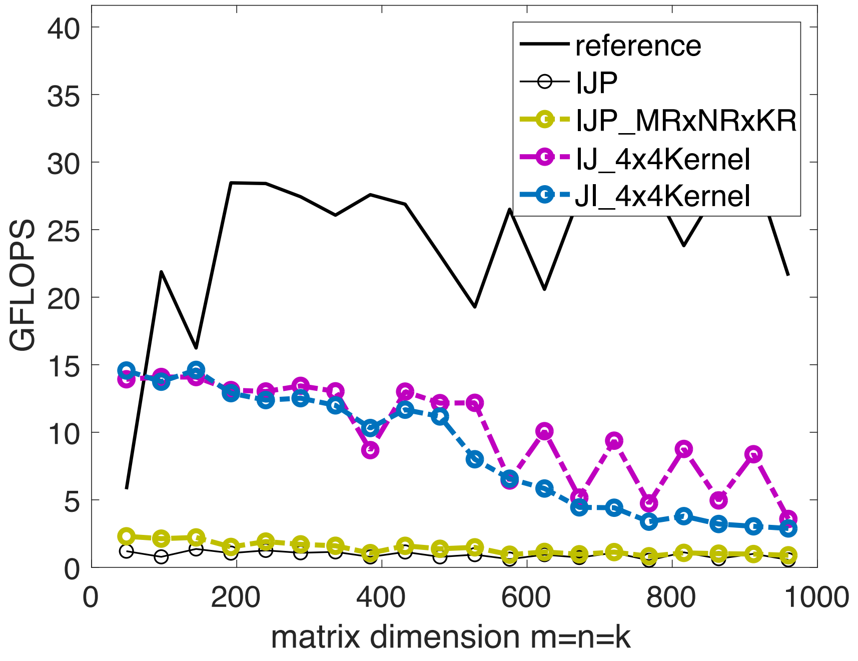


Figure 2.9: Performance of a double loop around a micro-kernel implemented with vector instructions, for $m_R = n_R = 4$.

2.3.2 More options

Homework 2.3.2.1 Modify ☛ `Gemm_JI_4x4Kernel.c` to implement the case where $m_R = 8$ and $n_R = 4$, storing the result in `Gemm_JI_8x4Kernel.c`. Don't forget to update `MR` and `NR`! You can test the result by typing

```
make test_JI_8x4Kernel
```

in that directory and view the resulting performance by appropriately modifying ☛ `data/Plot_MRxNR_Kernels.mlx`.

☛ SEE ANSWER

Homework 2.3.2.2 How many vector registors are needed for the implementation in Homework 2.3.2.1?
Ignoring the cost of loading the registers with the 8×4 submatrix of C, analyze the ratio of flops to loads for the implementation in Homework 2.3.2.1.

☛ SEE ANSWER

Homework 2.3.2.3 In the last few exercises, we considered $m_R \times n_R = 4 \times 4$ and $m_R \times n_R = 8 \times 4$, where elements of A are loaded without duplication into vector registers (and hence m_R must be a multiple of 4), and elements of B are loaded/broadcast. Extending this approach to loading A and B, complete the entries in the following table:

$m_R \times n_R$	# vector registers	flops/load
4×1		/ =
4×2		/ =
4×4	6	32 / 8 = 4
4×12		/ =
4×14		/ =
8×4		/ =
8×6		/ =
12×4		/ =
16×2		/ =
16×3		/ =

☛ SEE ANSWER

```c
#include <stdio.h>
#include <stdlib.h>

#define alpha( i,j ) A[ (j)*ldA + (i) ]   // map alpha( i,j ) to array A
#define beta( i,j )  B[ (j)*ldB + (i) ]   // map beta( i,j ) to array B
#define gamma( i,j ) C[ (j)*ldC + (i) ]   // map gamma( i,j ) to array C

#define MR 4
#define NR 4

void Gemm_JI_4x4Kernel( int, int, int, double *, int, double *, int, double *, int );

void Gemm_4x4Kernel( int, double *, int, double *, int, double *, int );

void GemmWrapper( int m, int n, int k, double *A, int ldA,
      double *B, int ldB, double *C, int ldC )
{
  if ( m % MR != 0 || n % NR != 0 ){
    printf( "m and n must be multiples of MR and NR, respectively \n" );
    exit( 0 );
  }

  Gemm_JI_4x4Kernel( m, n, k, A, ldA, B, ldB, C, ldC );
}

void Gemm_JI_4x4Kernel( int m, int n, int k, double *A, int ldA,
                                  double *B, int ldB, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
    for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
      Gemm_4x4Kernel( k, &alpha( i,0 ), ldA, &beta( 0,j ), ldB, &gamma( i,j ), ldC );
}
```

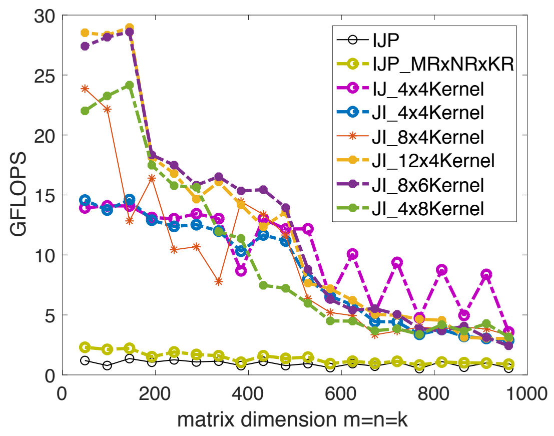
Assignments/Week2/C/Gemm_JI_4x4Kernel.c

Figure 2.10: General framework for calling a kernel, instantiated for the case where $m_R = n_R = 4$. (Continued in Figure 2.11.)

```c
#include<immintrin.h>

void Gemm_4x4Kernel( int k, double *A, int ldA, double *B, int ldB,
    double *C, int ldC )
{
  /* Declare vector registers to hold 4x4 C and load them */
  __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
  __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );
  __m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );
  __m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );

  for ( int p=0; p<k; p++ ){
    /* Declare vector register for load/broadcasting beta( b,j ) */
    __m256d beta_p_j;

    /* Declare a vector register to hold the current column of A and load
       it with the four elements of that column. */
    __m256d alpha_0123_p = _mm256_loadu_pd( &alpha( 0,p ) );

    /* Load/broadcast beta( p,0 ). */
    beta_p_j = _mm256_broadcast_sd( &beta( p, 0) );

    /* update the first column of C with the current column of A times
       beta ( p,0 ) */
    gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );

    /* REPEAT for second, third, and fourth columns of C.  Notice that the
       current column of A needs not be reloaded. */
  }

  /* Store the updated results */
  _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
  _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
  _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
  _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
}
```

Assignments/Week2/C/Gemm_JI_4x4Kernel.c

Figure 2.11: Continued: General framework for calling a kernel, instantiated for the case where $m_R = n_R = 4$.)



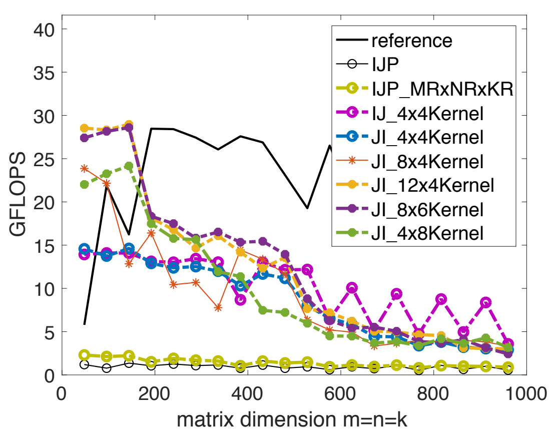


Figure 2.12: Performance of implementations for various choices of $m_R \times n_R$.

Homework 2.3.2.4 At this point, you have already implemented the following kernels:

`Gemm_JI_4x4Kernel.c` and `Gemm_JI_8x4Kernel.c`.

Implement as many of the more promising of the kernels you analyzed in the last homework as you like. They can be executed by typing

```
make test_JI_?x?Kernel
```

where the ?'s are replaced with the obvious choices of m_R and n_R. Don't forget to update `MR` and `NR`! The resulting performance can again be viewed with Live Script in ☛ `data/Plot_MRxNR_Kernels.mlx`.

☛ SEE ANSWER

In Figure 2.12, we show the performance of MMM cast in terms of various micro-kernel calls on Robert's laptop.

2.3.3 Amortizing data movement

As we have already discussed, one of the fundamental principles behind high performance is that when data is moved from one layer of memory to another, one needs to amortize the cost of that data movement over many computations. In our discussion so far, we have only discussed two layers of memory: (vector) registers and a cache.

Let us focus on the current approach:

- Four vector registers are loaded with a $m_R \times n_R$ submatrix of C. The cost of loading and unloading this data is negligible if k size is large.

- $m_R/4$ vector register is loaded with the current column of A, with m_R elements. In our picture, we call this register `alpha_0123_p`. The cost of this load is amortized over $2m_R \times n_R$ floating point operations.

- One vector register is loaded with broadcast elements of the current row of B. The loading of the n_R of the elements is also amortized over $2m_R \times n_R$ floating point operations.

To summarize, ignoring the cost of loading the submatrix of C, this would then require

$$m_R + n_R$$

floating point number loads from slow memory, m_R elements of A and n_R elements of B, which are amortized over $2 \times m_R \times n_R$ flops, for each rank-1 update.

If loading a double as part of vector load costs the same as the loading and broadcasting of a double, then the problem of finding the optimal choices of m_R and n_R can be stated as finding the m_R and n_R that optimize

$$\frac{2m_R n_R}{m_R + n_R}$$

under the constraint that

$$m_R n_R + m_R + 1 = r_C,$$

where r_c equals the number of doubles that can be stored in vector registers.

Finding the optimal solution is somewhat nasty, even if m_R and n_R are allowed to be real valued rather than integers. If we recognize that $m_R + 1$ is small relative to $m_R n_R$, and therefore ignore that those need to be kept in registers, we instead end up with having to find the m_R and n_R that optimize

$$\frac{2m_R n_R}{m_R + n_R}$$

under the constraint that

$$m_R n_R = r_C.$$

This is essentially a standard problem in calculus, used to find the sides of the rectangle with length x and width y that maximizes the area of the rectangle, xy, under the constraint that the length of the perimeter is constant. The answer for that problem is that the rectangle should be a square. Thus, we want the submatrix of C that is kept in registers to be squarish.

In practice, we have seen that the shape of the block of C kept in registers is noticeably not square. The reason for this is that, all other issues being equal, loading a double as part of a vector load requires fewer cycles than loading a double as part of a load/broadcast.

2.3.4 Discussion

Notice that we have analyzed that we must amortize the loading of elements of A and B over, at best, a few flops per such element. The "latency" from main memory (the time it takes for a double precision number to be fetched from main memory) is equal to the time it takes to perform on the order of 100 flops. Thus, "feeding the beast" from main memory spells disaster. Fortunately, modern architectures are equipped with multiple layers of cache memories that have a lower latency.

In Figure 2.12, we saw that for small matrices that fit in the L-2 cache, performance is very good, especially for the right choices of $m_R \times n_R$. The performance drops when problem sizes spill out of the L2 cache and drops again when they spill out of the L3 cache.

2.4 When Optimal Means Optimal

2.4.1 Reasoning about optimality

Early in our careers, we learned that if you say that an implementation is optimal, you better prove that it is optimal.

In our empirical studies, graphing the measured performance, we can compare our achieved results to the theoretical peak. Obviously, if we achieved theoretical peak performance, then we would know that the implementation is optimal. The problem is that we rarely achieve the theoretical peak performance as computed so far (multiplying the clock rate by the number of floating point operations that can be performed per clock cycle).

In order to claim optimality, one must carefully model an architecture and compute the exact limit of what it theoretical can achieve. Then, one can check achieved performance against the theoretical limit, and make a claim. Usually, this is also not practical.

In practice, one creates a model of computation for a simplified architecture. With that, one then computes a theoretical limit on performance. The next step is to show that the theoretical limit can be achieved by an algorithm that executes on that simplified architecture. This then says something about the optimality of the algorithm under idealized circumstances. By finally comparing and contrasting the simplified architecture with an actual architecture, and the algorithm that targets the simplified architecture with an actual algorithm designed for the actual architecture, one can reason about the optimality of the practical algorithm.

In this section, we will illustrate this process with MMM.

2.4.2 A simple model

Let us give a simple model of computation that matches what we have assumed so for when programming MMM:

- We wish to compute $C := AB + C$ where C, A, and C are $m \times n$, $m \times k$, and $k \times n$, respectively.
- The computation is cast in terms of FMAs.
- Our machine has two layers of memory: fast memory (registers) and slow memory (main memory).
- Initially, data reside in main memory.
- To compute a FMA, all three operands must be in fast memory.
- Fast memory can hold at most S floats.

- Slow memory is large enough that its size is not relevant to this analysis.
- Computation cannot be overlapped with data movement.

Notice that this model matches pretty well how we have viewed our processor so far.

2.4.3 Minimizing data movement

We have seen that MMM requires $m \times n \times k$ FMA operations, or $2mnk$ flops. Executing floating point operations constitutes useful computation. Moving data between slow memory and fast memory is overhead. Hence, under our simplified model, if an algorithm only performs the minimum number of flops (namely $2mnk$), minimizes the time spent moving data between memory layers, and we at any given time are either performing useful computation (flops) or moving data, then we can argue that (under our model) the algorithm is optimal.

We now focus the argument by reasoning about a lower bound on the number of data that must be moved from fast memory to slow memory. We will build the argument with a sequence of observations.

Consider the loop

```
for $p := 0, \ldots, k-1$
  for $j := 0, \ldots, n-1$
    for $i := 0, \ldots, m-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end
```

One can view the computations

$$\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$$

as a cube of points in 3D, (i, j, k) for $0 \leq i < m$, $0 \leq j < n$, $0 \leq p < k$. The set of all possible algorithms that execute each such update only once can be viewed as an arbitrary ordering on that set of points. We can this as indexing the set of all such triples with i_r, j_r, p_r, $0 \leq r < m \times n \times k$:

$$(i_r, j_r, k_r).$$

so that the algorithm that computes $C := AB + C$ can then be written as

```
for $r := 0, \ldots, mnk-1$
  $\gamma_{i_r,j_r} := \alpha_{i_r,p_r}\beta_{p_r,j_r} + \gamma_{i_r,j_r}$
end
```

Obviously, this puts certain restrictions on i_r, j_r, and p_r. Articulating those exactly is not important right now.

We now partition the ordered set $0, \ldots, mnk-1$ into ordered contiguous subranges (phases) each of which requires $S+M$ distinct elements from A, B, and C[1], except for the last phase, which will contain fewer[2]. Recall that S equals the number of floats that fit in fast memory. A typical phase will start with S elements in fast memory, and will in addition read M elements from slow memory (except for the final phase). Such a typical phase will start at $r = R$ and consists of F triples, (i_R, j_R, p_R) through $(i_{R+F-1}, j_{R+F-1}, p_{R+F-1})$. Let us denote the set of these triples by $\mathbf{D}$. These represent F FMAs being performed in our algorithm. Even the first phase needs to read *at least* M elements since it will require $S+M$ elements to be read.

The key question now becomes what the upper bound on the number of FMAs is that can be performed with $S+M$ elements. Let us denote this bound by $F_{\max}$. If we know $F_{\max}$ as a function of $S+M$, then we know that at least $\frac{mnk}{F_{\max}} - 1$ phases of FMAs need to be executed, where each of those phases requires at least S reads from slow memory. The total number of reads requires for *any* algorithm is thus at least

$$\left(\frac{mnk}{F_{\max}} - 1 \right) M. \tag{2.1}$$

To find $F_{\max}$ we make a few observations:

- A typical triple $(i_r, j_r, p_r) \in \mathbf{D}$ represents a FMA that requires one element from each of matrices C, A, and B: $\gamma_{i,j}$, $\alpha_{i,p}$, and $\beta_{p,j}$.
- The set of all elements from C, γ_{i_r,j_r}, that are needed for the computations represented by the triples in $\mathbf{D}$ are indexed with the tuples $\mathbf{C_D} = \{(i_r, j_r) \,|R \leq r < R+F\}$. If we think of the triples in $\mathbf{D}$ as points in 3D, then $\mathbf{C_D}$ is the projection of those points onto the i, j plane. Its size, $|\mathbf{C_D}|$, tells us how many elements of C must at some point be in fast memory during that phase of computations.
- The set of all elements from A, α_{i_r,p_r}, that are needed for the computations represented by the triples in $\mathbf{D}$ are indexed with the tuples $\mathbf{A_D} = \{(i_r, p_r) \,|R \leq r < R+F\}$. If we think of the triples (i_r, j_r, p_r) as points in 3D, then $\mathbf{A_D}$ is the projection of those points onto the i, p plane. Its size, $|\mathbf{A_D}|$, tells us how many elements of A must at some point be in fast memory during that phase of computations.
- The set of all elements from C, β_{p_r,j_r}, that are needed for the computations represented by the triples in $\mathbf{D}$ are indexed with the tuples $\mathbf{B_D} = \{(p_r, j_r) \,|R \leq r < R+F\}$. If we think of the triples (i_r, j_r, p_r) as points in 3D, then $\mathbf{B_D}$ is the projection of those points onto the p, j plane. Its size, $|\mathbf{B_D}|$, tells us how many elements of B must at some point be in fast memory during that phase of computations.

Now, there is an result known as the *discrete Loomis-Whitney inequality* that tells us that in our situation $|\mathbf{D}| \leq \sqrt{|\mathbf{C_D}||\mathbf{A_D}||\mathbf{B_D}|}$. In other words, $F_{\max} \leq \sqrt{|\mathbf{C_D}||\mathbf{A_D}||\mathbf{B_D}|}$. The name of the game

[1]E.g., S elements from A, $M/3$ elements of B, and $2M/3$ elements of C.

[2]Strictly speaking, it is a bit more complicated than splitting the range of the iterations, since the last FMA may require anywhere from 0 to 3 new elements to be loaded from slow memory. Fixing this is a matter of thinking of the loads that are required as separate from the computation (as our model does) and then splitting the operations (loads, FMAs, and stores) into phases rather than the range. This does not change our analysis.

now becomes to find the largest value $F_{\max}$ that satisfies

$$\text{maximize } F_{\max} \text{ such that } \begin{cases} F_{\max} \leq \sqrt{|\mathbf{C_D}||\mathbf{A_D}||\mathbf{B_D}|} \\ |\mathbf{C_D}| > 0, |\mathbf{A_D}| > 0, |\mathbf{B_D}| > 0 \\ |\mathbf{C_D}| + |\mathbf{A_D}| + |\mathbf{B_D}| = S + M. \end{cases}$$

An application known as Lagrange multipliers yields the solution

$$|\mathbf{C_D}| = |\mathbf{A_D}| = |\mathbf{B_D}| = \frac{S+M}{3} \quad \text{and} \quad F_{\max} = \frac{(S+M)\sqrt{S+M}}{3\sqrt{3}}.$$

With that largest $F_{\max}$ we can then establish a lower bound on the number of memory reads given by Equation (2.1):

$$\left(\frac{mnk}{F_{\max}} - 1\right) M = \left(3\sqrt{3}\frac{mnk}{(S+M)\sqrt{S+M}} - 1\right) M.$$

Now, M is a free variable. To come up with the sharpest (best) lower bound, we want the largest lower bound. It turns out that, using techniques from calculus, one can show that $M = 2S$ maximizes the lower bound. Thus, the best lower bound our analysis yields is given by

$$\left(3\sqrt{3}\frac{mnk}{(3S)\sqrt{3S}} - 1\right)(2S) = 2\frac{mnk}{\sqrt{S}} - 2S.$$

2.4.4 A nearly optimal algorithm

We now discuss a (nearly) optimal algorithm for our simplified architecture. Recall that we assume fast memory can hold S elements. For simplicity, assume S is a perfect square. Partition

$$C = \begin{pmatrix} C_{0,0} & C_{0,1} & \cdots \\ C_{1,0} & C_{1,1} & \cdots \\ \vdots & \vdots & \end{pmatrix}, \quad A = \begin{pmatrix} A_0 \\ A_1 \\ \vdots \end{pmatrix}, \quad \text{and} \quad B = \begin{pmatrix} B_0 & B_1 & \cdots \end{pmatrix}.$$

where $C_{i,j}$ is $m_R \times n_R$, A_i is $m_R \times k$, and B_j is $k \times n_R$. Here we choose $m_R \times n_R = (\sqrt{S} - 1) \times \sqrt{S}$ so that fast memory can hold one submatrix $C_{i,j}$, one column of A_i, and one element of B_j: $m_R \times n_R + m_R + 1 = (\sqrt{S} - 1) \times \sqrt{S} + \sqrt{S} - 1 + 1 = S$.

When computing $C := AB + C$, we recognize that $C_{i,j} := A_i B_j + C_{i,j}$. Now, let's further partition

$$A_i = \begin{pmatrix} a_{i,0} & a_{i,1} & \cdots \end{pmatrix} \quad \text{and} \quad B_j = \begin{pmatrix} b_{0,j}^T \\ b_{1,j}^T \\ \vdots \end{pmatrix}.$$

We now recognize that $C_{i,j} := A_i B_j + C_{i,j}$ can be computed as

$$C_{i,j} := a_{i,0} b_{0,j}^T + a_{i,1} b_{1,j}^T + \cdots,$$

the by now very familiar sequence of rank-1 updates that makes up the microkernel discussed in Section **??**. The following loop exposes the computation $C := AB + C$, including the loads and stores from and to slow memory:

```
for $j := 0, \ldots, N-1$
  for $i := 0, \ldots, M-1$
    load $C_{i,j}$ into fast memory
    for $p := 0, \ldots, k-1$
      load $a_{i,p}$ and $b^T_{p,j}$ into fast memory
      $C_{i,j} := a_{i,p} b^T_{p,j} + C_{i,j}$
    end
    store $C_{i,j}$ to slow memory
  end
end
```

For simplicity, here $M = m/m_r$ and $N = n/n_r$.

On the surface, this seems to require $C_{i,j}$, $a_{i,p}$, and $b^T_{p,j}$ to be in fast memory at the same time, placing $m_R \times n_R + m_R + n_r = S + \sqrt{S} - 1$ floats in fast memory. However, we have seen before that the rank-1 update $C_{i,j} := a_{i,p} b^T_{p,j} + C_{i,j}$ can be implemented as a loop around AXPY operations, so that the elements of $b^T_{p,j}$ only need to be in fast memory one at a time.

Let us now analyze the number of memory operations incurred by this algorithm:

- Loading and storing all $C_{i,j}$ incurs mn loads and mn stores, for a total of $2mn$ memory operations. (Each such block is loaded once and stored once, meaning every element of C is loaded once and stored once.)
- Loading all $a_{i,p}$ requires
$$MNkm_R = (Mm_R)Nk = m\frac{n}{n_R}k = \frac{mnk}{\sqrt{S}}$$
memory operations.
- Loading all $b^T_{p,j}$ requires
$$MNkn_R = M(Nn_R)k = \frac{m}{m_R}nk = \frac{mnk}{\sqrt{S}-1}$$
memory operations.

The total number of memory operations is hence
$$2mn + \frac{mnk}{\sqrt{S}} + \frac{mnk}{\sqrt{S}-1} = 2\frac{mnk}{\sqrt{S}} + 2mn + \frac{mnk}{S-\sqrt{S}}.$$

We can now compare this to the lower bound from the last unit:
$$2\frac{mnk}{\sqrt{S}} - 2S.$$

The cost of reading and writing elements of C contributes a lower order term, as does $\frac{mnk}{S-\sqrt{S}}$ if S (the size of fast memory) is reasonably large. Thus, the proposed algorithm is nearly optimal with regards to the amount of data that is moved between slow memory and fast memory.

2.4.5 Discussion

What we notice is that the algorithm presented in the last unit is quite similar to the algorithm that in the end delivered good performance in Section 2.3. Both utilize most of fast memory (registers in Section 2.3) with a submatrix of C. Both organize the computation in terms of a kernel that performs rank-1 updates of that submatrix of C.

What is different is that the theory indicates that the number of memory operations are minimized if the block of C is chosen to be square. In Section 2.3, the best performance was observed with a kernel that chose the submatrix of C in registers to be 12×4. The reason is that loading elements of A into registers (four at a time) is cheaper (on a per-element basis) than broadcasting elements of B. While in Unit 2.4.4 we try to minimize memory operations, in Section 2.3 we are trying to minimize time spent in those memory operations.

2.5 Leveraging the caches

We intend to move this section to Week 3.

2.5.1 Adding cache memory into the mix

We now refine our model of the processor slightly, adding one cache memory into the mix.

- Our processor has only one core.
- That core has three levels of memory: registers, a cache memory, and main memory.
- Moving data between the cache and registers takes time $\beta_{C \leftrightarrow R}$ per double while moving it between main memory and the cache takes time $\beta_{M \leftrightarrow C}$
- The registers can hold 64 doubles.
- The cache memory can hold one or more smallish matrices.
- Performing a flop with data in registers takes time γ_R.
- Data movement and computation cannot overlap.

The idea now is to figure out how to block the matrices into submatrices and then compute while these submatrices are in cache to avoid having to access memory more than necessary.

A naive approach partitions C, A, and B into (roughly) square blocks:

$$C = \left(\begin{array}{c|c|c|c} C_{0,0} & C_{0,1} & \cdots & C_{0,N-1} \\ \hline C_{1,0} & C_{1,1} & \cdots & C_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline C_{M-1,0} & C_{M,1} & \cdots & C_{M-1,N-1} \end{array}\right), A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,K-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,K-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{M-1,0} & A_{M,1} & \cdots & A_{M-1,K-1} \end{array}\right),$$

and

$$B = \left(\begin{array}{c|c|c|c} B_{0,0} & B_{0,1} & \cdots & B_{0,N-1} \\ \hline B_{1,0} & B_{1,1} & \cdots & B_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline B_{K-1,0} & B_{K,1} & \cdots & B_{K-1,N-1} \end{array}\right),$$

where $C_{i,j}$ is $m_C \times n_C$, $A_{i,p}$ is $m_C \times k_C$, and $B_{p,j}$ is $k_C \times n_C$. Then

$$C_{i,j} := \sum_{p=0}^{K-1} A_{i,p} B_{p,j} + C_{i,j},$$

which can be written as the triple-nested loop

```
**for** $i := 0, \ldots, M-1$
  **for** $j := 0, \ldots, N-1$
      **for** $p := 0, \ldots, K-1$
          $C_{i,j} := A_{i,p} B_{p,j} + B_{i,j}$
      **end**
    **end**
**end**
```

which is one of $3! = 6$ possible loop orderings.

If we choose m_C, n_C, and k_C such that $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ all fit in the cache, then we meet our conditions. We can then compute $C_{i,j} := A_{i,p} B_{p,j} + C_{i,j}$ by "bringing these blocks into cache" and computing with them before writing out the result, as before. The difference here is that while one can explicitly load registers, the movement of data into caches is merely encouraged by careful ordering of the computation, since replacement of data in cache is handled by the hardware, which has some cache replacement policy similar to "least recently used" data gets evicted.

```c
void Gemm_IJP_JI_4x4Kernel( int m, int n, int k, double *A, int ldA,
          double *B, int ldB, double *C, int ldC )
{
  int ib, jb, pb;

  for ( int i=0; i<m; i+=MC ) {
    ib = min( MC, m-i );         /* Last block may not be a full block */
    for ( int j=0; j<n; j+=NC ) {
      jb = min( NC, n-j );         /* Last block may not be a full block */
      for ( int p=0; p<k; p+=KC ) {
  pb = min( KC, k-p );         /* Last block may not be a full block */

  Gemm_JI_4x4Kernel
    ( ib, jb, pb, &alpha( i,p ), ldA, &beta( p,j ), ldB, &gamma( i,j ), ldC );
      }
    }
  }
}
```

Assignments/Week2/C/Gemm_IJP_JI_4x4Kernel.c

Figure 2.13: Triple loop around $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$.

Homework 2.5.1.1 In Figure 2.13, we give an IJP loop ordering around the computation of $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$, which itself is implemented by `Gemm_JI_4x4Kernel`. It can be found in ☛ `Gemm_IJP_JI_4x4Kernel.c`
and executed by typing

```
make test_Gemm_IJP_JI_4x4Kernel
```

in that directory. The resulting performance can be viewed with Live Script ☛ `data/Plot_XYZ_JI_MRxNRKernel.mlx`.

☛ SEE ANSWER

Homework 2.5.1.2 Copy ☛ `Gemm_IJP_JI_4x4Kernel.c`. into `Gemm_IJP_JI_12x4Kernel.c` and modify it to call your implementation of the 12×4 kernel. Collect performance data by executing

```
make test_Gemm_IJP_JI_12x4Kernel
```

in that directory. The resulting performance can be viewed with Live Script ☛ `data/Plot_XYZ_JI_MRxNRKernel.mlx`.

☛ SEE ANSWER

Homework 2.5.1.3 Copy ☛ `Gemm_IJP_JI_12x4Kernel.c` into `Assignments/Week2/C/Gemm_JPI_JI_12x4Kernel.c` and modify the order of the loops to JPI. Collect performance data by executing

```
make test_Gemm_JPI_JI_12x4Kernel
```

in that directory. The resulting performance can be viewed with Live Script ☛ `data/Plot_XYZ_JI_MRxNRKernel.mlx`.

☛ SEE ANSWER

Homework 2.5.1.4 Copy `Assignments/Week2/C/Gemm_IJP_JI_12x4Kernel.c` from the last homework into `Assignments/Week2/C/Gemm_PIJ_JI_12x4Kernel.c` and modify the order of the loops to PIJ. Collect performance data by executing

make test_Gemm_PIJ_JI_12x4Kernel

in that directory. The resulting performance can be viewed with Live Script ☛ `data/Plot_XYZ_JI_MRxNRKernel.mlx`.

☛ SEE ANSWER

The performance attained by our implementation is shown in Figure **??** (top-right).

2.5.2 Streaming submatrices of C and B

We illustrate the computation with one set of three submatrices (one per matrix A, B, and C) in Figure 2.14 for the case where $m_R = n_R = 4$ and $m_C = n_C = k_C = 4m_R$. What we notice is that each the $m_R \times n_R$ submatrices of $C_{i,j}$ is not reused again once it has been updated. This means that at any given time only one such matrix needs to be in the cache memory (as well as registers). Similarly, a panel of $B_{p,j}$ is not reused and hence only one such panel needs to be in cache. It is submatrix $A_{i,p}$ that must remain in cache during the entire computation. By not keeping the entire submatrices $C_{i,j}$ and $B_{p,j}$ in the cache memory, the size of the submatrix $A_{i,p}$ can be increased, which can be expected to improve performance. We will refer to this as "streaming" "micro-blocks" of $C_{i,j}$ and small panels of $B_{p,j}$, while keeping all of block $A_{i,p}$ in cache.

We could have chosen a different order of the loops, and then we may have concluded that submatrix $B_{p,j}$ needs to stay in cache while streaming a micro-blocks of C and small panels of A. Obviously, there is a symmetry here and hence we will just focus on the case where $A_{i,p}$ is kept in cache.

The second observation is that now that $C_{i,j}$ and $B_{p,j}$ are being streamed, there is no need for those submatrices to be square. Furthermore, the larger n_C, the more effectively the cost of bringing $A_{i,p}$ into cache is amortized over computation. Perhaps we should pick $n_C = n$. (Later, we will reintroduce n_C.) Another way of think of this is that if we choose the PIJ loop around the JI ordering around the micro-kernel (in other words, if we consider the `Gemm_PIJ_JI_12x4Kernel`

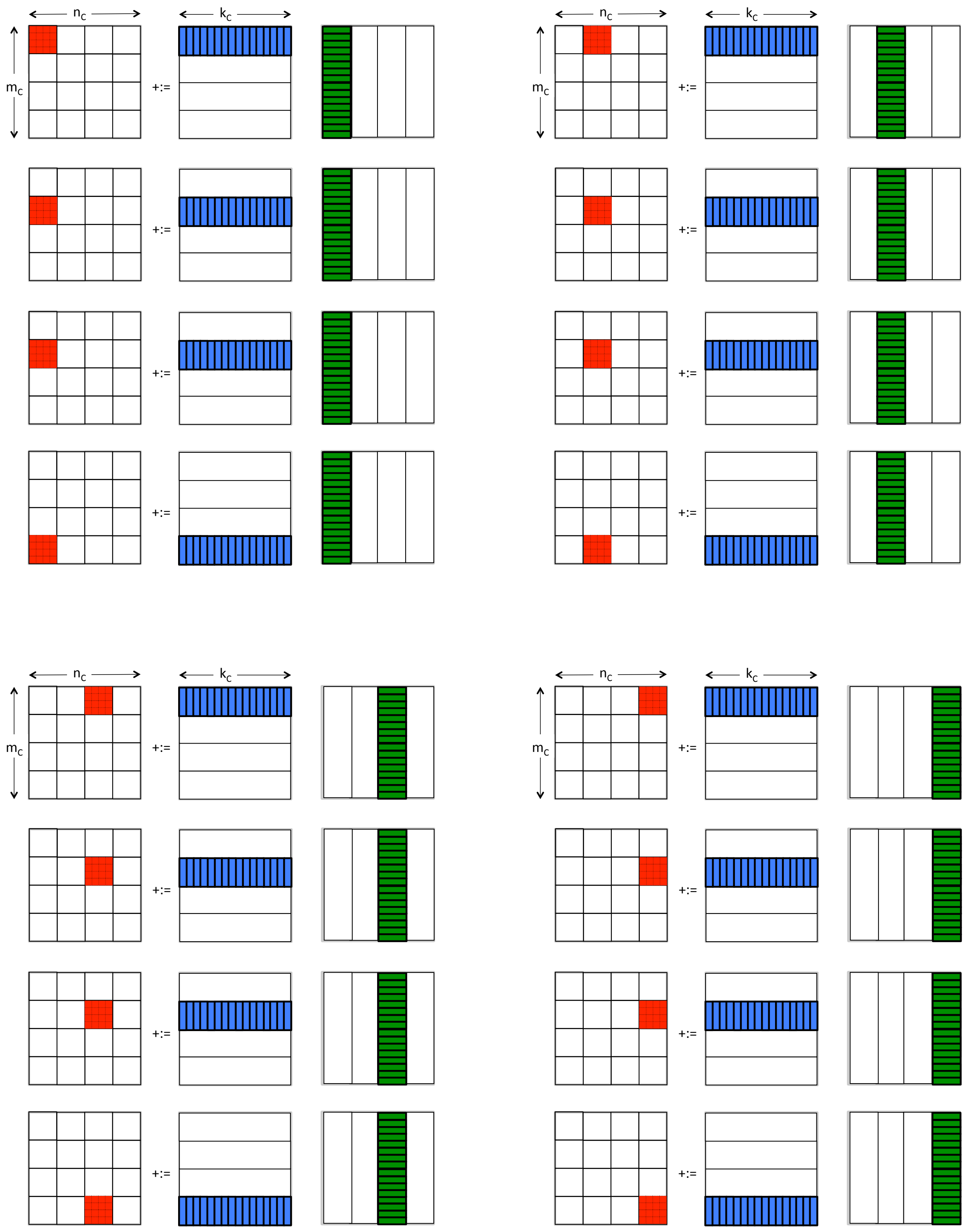


Figure 2.14: Illustration of computation in the cache with one block from each of A, B, and C.

implementation), then we notice that the inner loop of PIJ (indexed with J) matches the outer loop of the JI double loop, and hence those two loops can be combined, leaving us with an implementation that is a double loop (PI) around the double loop JI.

Homework 2.5.2.1 Copy ☛ `Gemm_PIJ_JI_12x4Kernel.c` into `Gemm_PI_JI_12x4Kernel.c` and remove the loop indexed with j. Collect performance data by executing

```
make test_Gemm_PI_JI_12x4Kernel
```

in that directory. The resulting performance can be viewed with Live Script ☛ `data/Plot_XYZ_JI_MRxNRKernel.mlx`.

☛ SEE ANSWER

Homework 2.5.2.2 Execute

make test_PI_JI_MCxKC

and view the performance results with Live Script

☛ `data/Plot_MC_KC_Performance.mlx`.

This experiment tries many different choices for m_C and k_C, and presents them as a scatter graph so that the optimal choice can be determined and visualized. (It will take a long time to execute. Go take a nap!)

☛ SEE ANSWER

The result of this last homework on Robert's laptop is shown in Figure 2.15.

To summarize, our insights suggest

1. Bring an $m_C \times k_C$ submatrix of A into the cache, at a cost of $m_C \times k_C$ memops. (It is the order of the loops and instructions that encourages the submatrix of A to stay in cache.) This cost is amortized over $2m_C n_C k_C$ flops for a ratio of $2m_C n k_C/(m_C k_C) = 2n$. The larger n, the better.

2. The cost of reading an $k_C \times n_R$ submatrix of B is amortized over $2m_C n_R k_C$ flops, for a ratio of $2m_C n_R k_C/(2m_C n_R) = m_C$. Obviously, the greater m_C, the better.

3. The cost of reading and writing an $m_R \times n_R$ submatrix of C is now amortized over $2m_R n_R k_C$ flops, for a ratio of $2m_R n_R k_C/(2m_R n_R) = k_C$. Obviously, the greater k_C, the better.

Items 2 and 3 suggest that $m_C \times k_C$ submatrix $A_{i,p}$ be roughly square. (This comes with a caviat. See video!)

If we revisit the performance data plotted in Figure 2.15, we notice that matrix $A_{i,p}$ fits in part of the L2 cache, but is too big for the L1 cache. What this means is that the bandwidth between the L2 and registers is enough to allow these blocks of A to reside in the L2 cache. Since the L2 cache is larger than the L1 cache, this benefits performance, since the m_C and k_C can be larger.

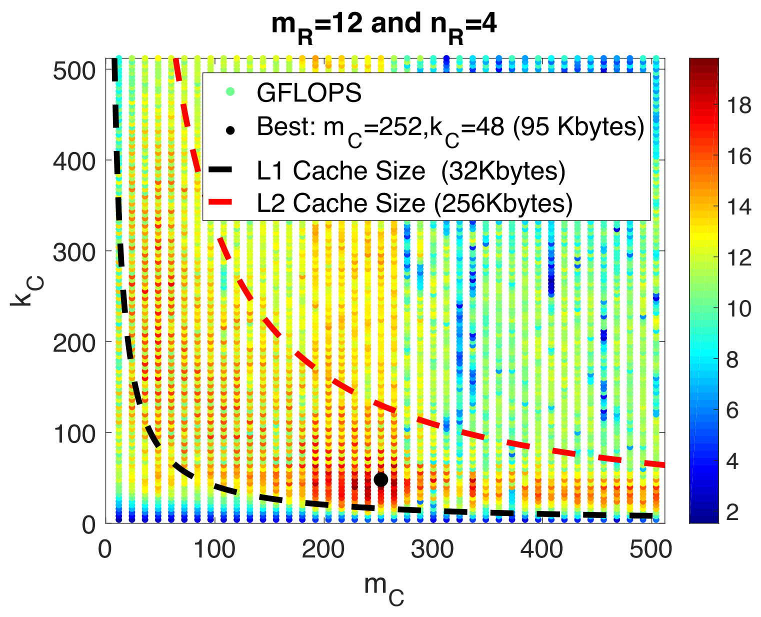


Figure 2.15: Performance when $m_R = 12$ and $n_R = 4$ and m_C and k_C are varied. The best performance is empirically determined by searching the results. We notice that $\widetilde{A}$ is sized to fit in the L2 cache. As we apply additional optimizations, the optimal choice for m_C and k_C will change. Notice that on Robert's laptop, the optimal choice varies greatly from one experiment to the next. You may observe the same level of variability.

In our future discussions, we will use the following terminology:

- A $m_R \times n_R$ submatrix of C that is begin updated we will call a *micro-tile*.
- The $m_R \times k_C$ submatrix of A and $k_C \times n_R$ submatrix of B we will call *micro-panels*.
- The routine that updates a micro-tile by multiplying two micro-panels we will call the micro-kernel.

2.5.3 Blocking for multiple caches

Figure 2.16 describes one way for blocking for multiple levels of cache. While some details still remain, this brings us very close to how matrix-matrix multiplication is implemented in practice in libraries that are widely used. The approach illustrated there was first suggested by Kazushige Goto [**?**] and is referred to as the Goto approach (to matrix-matrix multiplication) or GotoBLAS approach, since he incorporated it in an implementation of the Basic Linear Algebra Subprograms (BLAS) by that name.

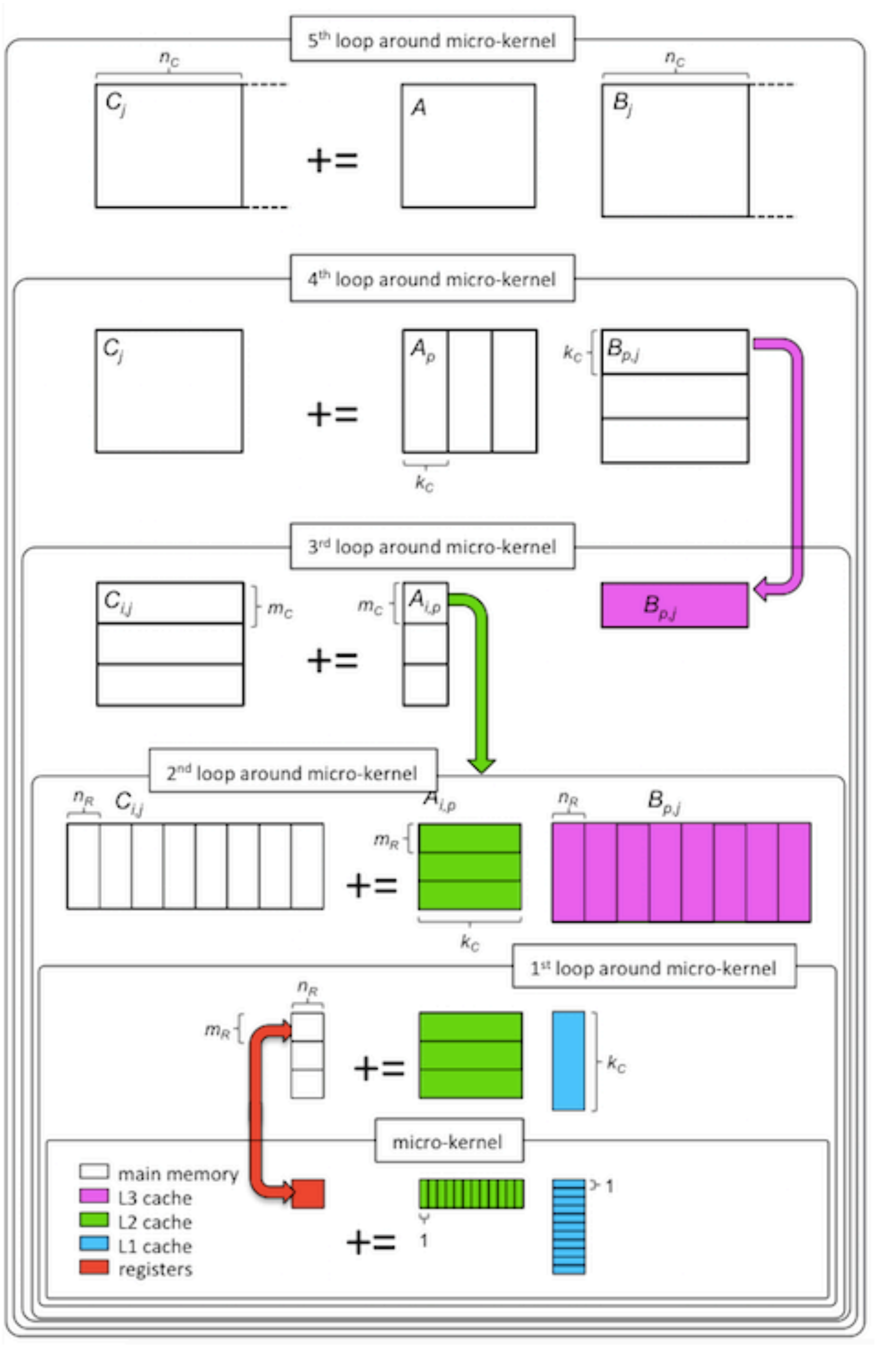

5th loop around micro-kernel
n_C
C_j
+=
A
B_j
4th loop around micro-kernel
A_p
k_C
$B_{p,j}$
3rd loop around micro-kernel
$C_{i,j}$
m_C
$A_{i,p}$
2nd loop around micro-kernel
n_R
m_R
1st loop around micro-kernel
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

Figure 2.16: Blocking for multiple levels of cache. Figure adapted from [**?**].

Each nested box represents a single loop that partitions one of the three dimensions (m, n, or k). The submatrices $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ are those already discussed in Section 2.5.1.

- The roughly square matrix $A_{i,p}$ is "placed" in the L2 cache. If data from $A_{i,p}$ can be fetched from the L2 cache fast enough, it is better to place it there since there is an advantage to making m_C and k_C larger and the L2 cache is larger than the L1 cache.
- Our analysis in Section 2.5.1 also suggests that k_C be chosen to be large. Since the submatrix $B_{p,j}$ is reused multiple times for many iterations of the "fourth loop around the micro-kernel" it may be beneficial to choose n_C so that $k_C \times n_C$ submatrix $B_{p,j}$ stays in the L3 cache. Later we will see there are other benefits to this.
- In this scheme, the $m_R \times n_R$ micro-block of $C_{i,j}$ end up in registers and the $k_C \times n_R$ "micro-panel" of $B_{p,j}$ ends up in the L1 cache.

Data in the L1 cache typically are also in the L2 and L3 caches. The update of the $m_R \times n_R$ micro-block of C also brings that in to the various caches. Thus, the described situation is similar to what we described in Section 2.5.1, where "the cache" discussed there corresponds to the L2 cache here. As our discussion proceeds, in Week 3, we will try to make this all more precise.

Homework 2.5.3.1 Which of the implementations

- `Gemm_IJP_JI_12x4Kernel.c`
- `Gemm_JPI_JI_12x4Kernel.c`
- `Gemm_PJI_JI_12x4Kernel.c`

best captures the loop structure illustrated in Figure 2.16? Modify that implementation with reasonable choices for `MC`, `NC`, and `KC`, re-execute to gather data and view with Live Script

`Assignments/Week2/C/Plot_XYZ_JI_MRxNRKernel.mlx`.

☛ SEE ANSWER

Homework 2.5.3.2 We always advocate, when it does not substantially impede performance, to instantiate an algorithm in code in a way that closely resembles how one explains it. In this spirit, the algorithm described in Figure 2.16 can be coded by making each loop (box) in the figure a separate routine. An outline of how this might be accomplished is given in Figure 2.17 and file ☛ `Gemm_Five_Loops_12x4Kernel.c`. Complete the code and execute it with

```
make test_Five_Loops
```

Then, view the performance with Live Script ☛ `data/Plot_Five_Loops.mlx`.

☛ SEE ANSWER

```c
void LoopFive( int m, int n, int k, double *A, int ldA,
       double *B, int ldB, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NC ) {
    int jb = min( NC, n-j );    /* Last loop may not involve a full block */
    LoopFour( m, jb, k, A, ldA, &beta( 0,j ), ldB, &gamma( 0,j ), ldC );
  }
}

void LoopFour( int m, int n, int k, double *A, int ldA,
         double *B, int ldB, double *C, int ldC )
{
  for ( int p=0; p<k; p+=KC ) {
    int pb = min( KC, k-p );    /* Last loop may not involve a full block */
    LoopThree( m, n, pb, &alpha( 0, p ), ldA, &beta( p, 0 ), ldB, C, ldC );
  }
}

void LoopThree( int m, int n, int k, double *A, int ldA,
    double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MC ) {
    int ib = min( MC, m-i );    /* Last loop may not involve a full block */
    LoopTwo( ib, n, k, &alpha( i, 0), ldA, B, ldB, &gamma( i,0 ), ldC );
  }
}

void LoopTwo( int m, int n, int k, double *A, int ldA,
        double *B, int ldB, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NR ) {
    int jb = min( NR, n-j );
    LoopOne( m, jb, k, A, ldA, &beta( 0,j ), ldB, &gamma( 0,j ), ldC );
  }
}

void LoopOne( int m, int n, int k, double *A, int ldA,
        double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MR ) {
    int ib = min( MR, m-i );
    Gemm_12x4Kernel( k, &alpha( i, 0 ), ldA, B, ldB, &gamma( i,0 ), ldC );
  }
}
```

Assignments/Week3/Answers/Gemm_Five_Loops_12x4Kernel.c

Figure 2.17: Blocking for multiple levels of cache.

2.6 Enrichment

2.7 Wrapup

Week 3

Pushing the Limits

3.1 Opener

3.1.1 Launch

3.1.2 Outline Week 2

3.1.3 What you will learn

3.2 Further Optimizing the Micro-kernel

3.2.1 Packing

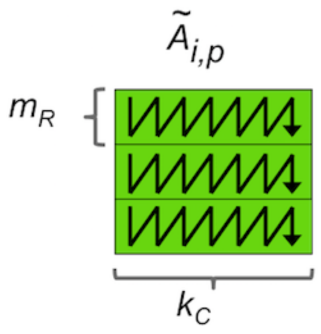
The next step towards optimizing the micro-kernel recognizes that computing with contiguous data (accessing data with a "stride one" access pattern) improves performance. The fact that the $m_R \times n_R$ micro-tile of C is not in contiguous memory is not particularly important. The cost of bringing it into the vector registers from some layer in the memory is mostly inconsequential because a lot of computation is performed before it is written back out. It is the repeated accessing of the elements of A and B that can benefit from stride one access.

Two successive rank-1 updates of the micro-tile can be given by

$$\begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{0,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{0,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{0,3} \end{pmatrix} +:=$$

$$\begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix} \begin{pmatrix} \beta_{p,0} & \beta_{p,1} & \beta_{p,2} & \beta_{p,3} \end{pmatrix} + \begin{pmatrix} \alpha_{0,p+1} \\ \alpha_{1,p+1} \\ \alpha_{2,p+1} \\ \alpha_{3,p+1} \end{pmatrix} \begin{pmatrix} \beta_{p+1,0} & \beta_{p+1,1} & \beta_{p+1,2} & \beta_{p+1,3} \end{pmatrix}.$$

Since A and B are stored with column-major order, the four elements of $\alpha_{[0:3],p}$ are contiguous in memory, but they are (generally) not contiguously stored with $\alpha_{[0:3],p+1}$. Elements $\beta_{p,[0:3]}$ are (generally) also not contiguous. The access pattern during the computation by the micro-kernel would be much more favorable if the A involved in the micro-kernel was packed in column-major order with leading dimension m_R:



and B was packed in row-major order with leading dimension n_R:

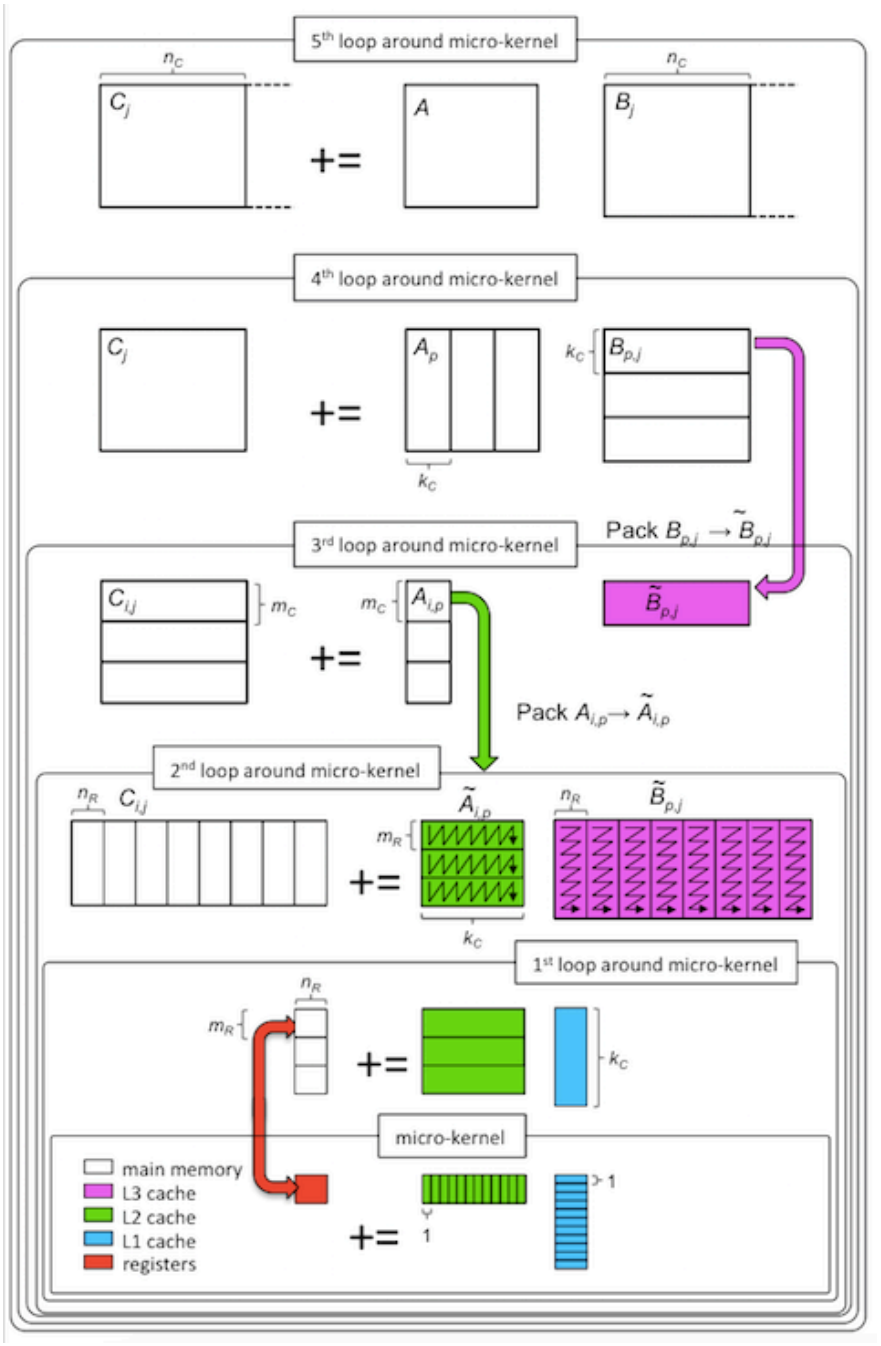


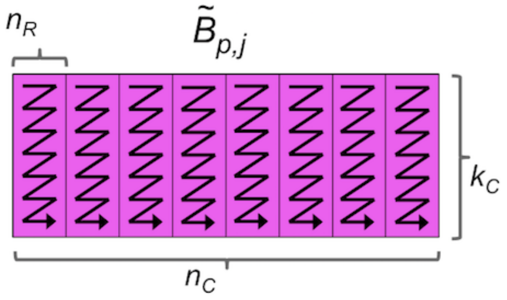
Figure 3.1: Blocking for multiple levels of cache, with packing. Picture adapted from []

```
void PackMicroPanelA_MRxKC( int m, int k, double *A, int ldA, double *Atilde )
/* Pack a micro-panel of A into buffer pointed to by Atilde.
   This is an unoptimized implementation for general MR and KC. */
{
  /* March through A in column-major order, packing into Atilde as we go. */

  if ( m == MR )   /* Full row size micro-panel.*/
    for ( int p=0; p<k; p++ )
      for ( int i=0; i<MR; i++ )
        *Atilde++ = alpha( i, p );
  else /* Not a full row size micro-panel.  We pad with zeroes. */
    for ( int p=0; p<k; p++ ) {
      for ( int i=0; i<m; i++ )
        *Atilde++ = alpha( i, p );
      for ( int i=m; i<MR; i++ )
        *Atilde++ = 0.0;
    }
}

void PackBlockA_MCxKC( int m, int k, double *A, int ldA, double *Atilde )
/* Pack a MC x KC block of A.  MC is assumed to be a multiple of MR.  The block is packed
   into Atilde a micro-panel at a time. If necessary, the micro-panel is padded with rows
   of zeroes. */
{
  for ( int i=0; i<m; i+=MR ){
    int ib = min( MR, m-i );

    PackMicroPanelA_MRxKC( ib, k, &alpha( i, 0 ), ldA, Atilde );
    Atilde += MR * k;
  }
}
```

Figure 3.2: A reference implementation of routines for packing $A_{i,p}$.

```
void PackMicroPanelB_KCxNR( int k, int n, double *B, int ldB, double *Btilde )
/* Pack a micro-panel of B into buffer pointed to by Btilde.
   This is an unoptimized implementation for general KC and NR. */
{
  /* March through B in row-major order, packing into Btilde as we go. */

  if ( n == NR ) /* Full column width micro-panel.*/
    for ( int p=0; p<k; p++ )
      for ( int j=0; j<NR; j++ )
        *Btilde++ = beta( p, j );
  else /* Not a full row size micro-panel.  We pad with zeroes. */
    for ( int p=0; p<k; p++ ) {
      for ( int j=0; j<n; j++ )
        *Btilde++ = beta( p, j );
      for ( int j=n; j<NR; j++ )
        *Btilde++ = 0.0;
    }
}

void PackPanelB_KCxNC( int k, int n, double *B, int ldB, double *Btilde )
/* Pack a KC x NC panel of B.  NC is assumed to be a multiple of NR.  The
   block is packed into Btilde a micro-panel at a time. If necessary, the
   micro-panel is padded with columns of zeroes. */
{
  for ( int j=0; j<n; j+= NR ){
    int jb = min( NR, n-j );

    PackMicroPanelB_KCxNR( k, jb, &beta( 0, j ), ldB, Btilde );
    Btilde += k * jb;
  }
}
```

Figure 3.3: A reference implementation of routines for packing $B_{p,j}$.



If this packing were performed at a strategic point in the computation, so that the packing is amortized over many computations, then a benefit might result. These observations are captured in Figure 3.1 and translated into an implementation in Figure 3.4. Reference implementations of packing routines can be found in Figures 3.2 and 3.3. While these implementations can be optimized, the fact is that the cost when packing is in the data movement between main memory and faster memory. As a result, optimizing the packing has relatively little effect.

How to modify the five loops to incorporate packing is illustrated in Figure 3.4. A micro-kernel to compute with the packed data when $m_R \times n_R = 4 \times 4$ is illustrated in Figure 3.5.

Homework 3.2.1.1 Copy the file `Gemm_Five_Loops_Pack_4x4Kernel.c` into file `Gemm_Five_Loops_Pack_12x4Kernel.c`. Modify that file so that it uses $m_R \times n_R = 12 \times 4$. Test the result with

```
make test_Five_Loops_Pack_12x4
```

and view the resulting performance with Live Script

```
Plot_Five_Loops.mlx
```

☛ SEE ANSWER

3.2.2 Further single core optimizations to try.

We now give a laundry list of further optimizations that can be tried.

Different $m_R \times n_R$ choices As mentioned, there are many choices for $m_R \times n_R$. On the Haswell architecture, it has been suggested that 12×4 or 4×12 are good choices. The choice 4×12 where elements of B are loaded into vector registers and elements of A are broadcast has some advantage.

Prefetching. Elements of $A_{i,p}$ in theory remain in the L2 cache. Loading them into registers may be faster if elements of a next micro-panel of A are prefetched into the L1 cache while computing with the current micro-panel, overlapping that movement with computation. There are intrinsic instructions for that. Some choices of m_R and n_R lend themselves better to exploiting this.

```c
void LoopFive( int m, int n, int k, double *A, int ldA,
       double *B, int ldB, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NC ) {
    int jb = min( NC, n-j );    /* Last loop may not involve a full block */
    LoopFour( m, jb, k, A, ldA, &beta( 0,j ), ldB, &gamma( 0,j ), ldC );
  }
}

void LoopFour( int m, int n, int k, double *A, int ldA, double *B, int ldB,
         double *C, int ldC )
{
  double *Btilde = ( double * ) malloc( KC * NC * sizeof( double ) );

  for ( int p=0; p<k; p+=KC ) {
    int pb = min( KC, k-p );    /* Last loop may not involve a full block */
    PackPanelB_KCxNC( pb, n, &beta( p, 0 ), ldB, Btilde );
    LoopThree( m, n, pb, &alpha( 0, p ), ldA, Btilde, C, ldC );
  }

  free( Btilde);
}

void LoopThree( int m, int n, int k, double *A, int ldA, double *Btilde, double *C, int ldC )
{
  double *Atilde = ( double * ) malloc( MC * KC * sizeof( double ) );

  for ( int i=0; i<m; i+=MC ) {
    int ib = min( MC, m-i );    /* Last loop may not involve a full block */
    PackBlockA_MCxKC( ib, k, &alpha( i, 0 ), ldA, Atilde );
    LoopTwo( ib, n, k, Atilde, Btilde, &gamma( i,0 ), ldC );
  }

  free( Atilde);
}

void LoopTwo( int m, int n, int k, double *Atilde, double *Btilde, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NR ) {
    int jb = min( NR, n-j );
    LoopOne( m, jb, k, Atilde, &Btilde[ j*k ], &gamma( 0,j ), ldC );
  }
}

void LoopOne( int m, int n, int k, double *Atilde, double *MicroPanelB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MR ) {
    int ib = min( MR, m-i );
    Gemm_4x4Kernel_Packed( k, &Atilde[ i*k ], MicroPanelB, &gamma( i,0 ), ldC );
  }
}
```

Assignments/Week3/C/Gemm_Five_Loops_Pack_4x4Kernel.c

Figure 3.4: Blocking for multiple levels of cache, with packing.

```
void Gemm_4x4Kernel_Packed( int k,
            double *BlockA, double *PanelB, double *C, int ldC )
{
  __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
  __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );
  __m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );
  __m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );

  __m256d beta_p_j;

  for ( int p=0; p<k; p++ ){
    /* load alpha( 0:3, p ) */
    __m256d alpha_0123_p = _mm256_loadu_pd( BlockA );

    /* load beta( p, 0 ); update gamma( 0:3, 0 ) */
    beta_p_j = _mm256_broadcast_sd( PanelB );
    gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );

    /* load beta( p, 1 ); update gamma( 0:3, 1 ) */
    beta_p_j = _mm256_broadcast_sd( PanelB+1 );
    gamma_0123_1 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_1 );

    /* load beta( p, 2 ); update gamma( 0:3, 2 ) */
    beta_p_j = _mm256_broadcast_sd( PanelB+2 );
    gamma_0123_2 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_2 );

    /* load beta( p, 3 ); update gamma( 0:3, 3 ) */
    beta_p_j = _mm256_broadcast_sd( PanelB+3 );
    gamma_0123_3 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_3 );

    BlockA += MR;
    PanelB += NR;
  }

  /* Store the updated results.  This should be done more carefully since
     there may be an incomplete micro-tile. */
  _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
  _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
  _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
  _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
}
```

Assignments/Week3/C/Gemm_Five_Loops_Pack_4x4Kernel.c

Figure 3.5: Blocking for multiple levels of cache, with packing.

Loop unrolling. There is some overhead that comes from having a loop in the micro-kernel. That overhead is in part due to loop indexing and branching overhead. The loop also can get in the way of the explicitly reordering of operations towards some benefit. So, one could turn the loop into a long sequence of instructions. A compromise is to "unroll" it more modestly, decreasing the number of iterations while increasing the number of operations performed in each iteration.

Different m_c, n_c and/or k_c choices. There may be a benefit to playing around with the various blocking parameters.

Using in-lined assembly code Even more control over where what happens from using in-lined assembly code. This will allow prefetching and how registers are used to be made more explicit. (The compiler often changes the order of the operations with intrinsics are used in C.)

Placing a block of B in the L2 cache. There is a symmetric way of blocking the various matrices that then places a block of B in the L2 cache. This has the effect of accessing C by rows, if C were accessed by columns before, and vise versa. For some choices of $m_R \times n_R$, this may show a benefit.

Repeat for single precision!

Repeat for another architecture!

Repeat for other MMM-like operations!

3.3 Multithreaded parallelization

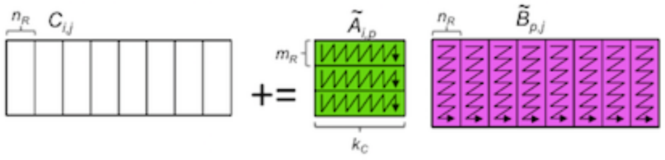
3.3.1 OpenMP

OpenMP is an Application Programming Interface (API) that allows one to communicate, partially to the compiler at compilation time and through a library of routines at execution time, how to achieve parallel execution. A good place to read up on OpenMP is, where else, Wikipedia: https://en.wikipedia.org/wiki/OpenMP.

In this section, we illustrate how OpenMP works by parallelizing our matrix-matrix multiplication implementations.

3.3.2 Parallelizing the second loop around the micro-kernel

Let's dive right in and parallelize one of the loops. We flipped a coin and it is the second loop around the micro-kernel that we are going to discuss first:



implemented by

```
void LoopTwo( int m, int n, int k, double *Atilde, double *Btilde, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NR ) {
    int jb = min( NR, n-j );
    LoopOne( m, jb, k, Atilde, &Btilde[ j*k ], &gamma( 0,j ), ldC );
  }
}
```

What we want to express is that each threads will perform a subset of multiplications of the submatrix $\tilde{A}_{i,p}$ times the micro-panels of $\tilde{B}_{p,j}$. Each should perform a roughly equal part of the necessary computation. This is indicated by inserting a *pragma*, a directive to the compiler:

```
#pragma omp parallel for
  for ( int j=0; j<n; j+=NR ) {
    int jb = min( NR, n-j );
    LoopOne( m, jb, k, Atilde, &Btilde[ j*k ], &gamma( 0,j ), ldC );
  }
```

To compile this, we must add `-fopenmp` to the `CFLAGS` in the makefile. Next, at execution time, you will want to indicate how many threads to use during the execution. First, to find out how many cores are available, you may want to execute

```
$ lscpu
```

which, on the machine where I tried it, yields

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                20
On-line CPU(s) list:   0-19
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
Stepping:              2
CPU MHz:               1326.273
CPU max MHz:           3000.0000
CPU min MHz:           1200.0000
BogoMIPS:              4594.96
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
NUMA node0 CPU(s):     0-19
```

and some additional information. What this tells us is that there are 10 cores, with two threads possible per core (this is called *hyperthreading*). On a mac you may want to try executing

```
$ sysctl -n hw.physicalcpu
```

instead. If you are using the BASH shell for Linux[1], you may then want to indicate, for example, that you want to use four threads:

```
   $ export OMP_NUM_THREADS=4
```

After you have added the directive to your code, compile it and execute.

[1]If you use a C-shell, the command is `setenv OMP_NUM_THREADS 4`.

Homework 3.3.2.2 So far, you have optimized your choice of MMM in file `GemmFiveLoops_??x??.c` where `??x??` reflects the register blocking for your micro-kernel. At this time, your implementation targets a single core. For the remainder of the exercises, you will start with *out* optimization for the case where $m_r \times n_r = 12 \times 4$. Now, in directory `Nov14`,

- Start with `Gemm_Parallel_Loop2_12x4.c`, in which you will find our implementation for a single core with $m_r \times n_r = 12 \times 4$.
- Modify it as described in this unit to parallelize Loop 2 around the micro-kernel.
- Set the number of threads to some number between 1 and the number of CPUs in the target processor.
- Execute `make test_Gemm_Parallel_Loop2_12x4`.
- View the resulting performance with `make ShowPerformance.mlx`.
- Be sure to check if you got the right answer!
- How does performance improve relative to the number of threads you indicated should be used?

☛ SEE ANSWER

3.3.3 Parallelizing the first loop around the micro-kernel

One can similarly optimize the first loop around the micro-kernel:

Homework 3.3.3.3 In directory `Nov14`,

- Copy `Gemm_Parallel_Loop2_12x4.c` into `Gemm_Parallel_Loop1_12x4.c`.
- Modify it so that only the first loop around the micro-kernel is parallelized.
- Set the number of threads to some number between 1 and the number of CPUs in the target processor.
- Execute `make test_Gemm_Parallel_Loop1_12x4`.
- View the resulting performance with `make ShowPerformance.mlx`, uncommenting the appropriate lines. (You should be able to do this so that you see previous performance curves as well.)
- Be sure to check if you got the right answer!
- How does the performance improve relative to the number of threads being used?

☛ SEE ANSWER

3.3.4 Parallelizing the third loop around the micro-kernel

One can similarly optimize the third loop around the micro-kernel:

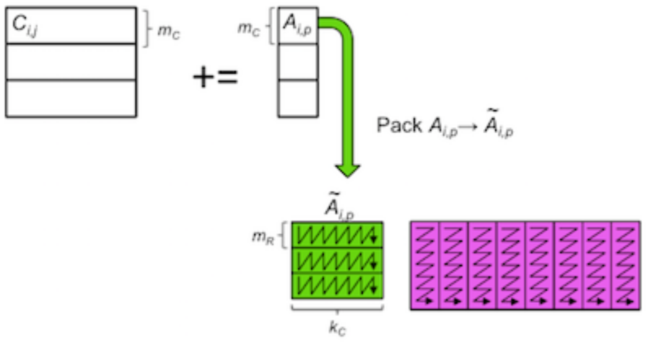
Homework 3.3.4.4 In directory `Nov14`,

- Copy `Gemm_Parallel_Loop2_12x4.c` into `Gemm_Parallel_Loop3_12x4.c`.
- Modify it so that only the third loop around the micro-kernel is parallelized.
- Set the number of threads to some number between 1 and the number of CPUs in the target processor.
- Execute `make test_Gemm_Parallel_Loop3_12x4`.
- View the resulting performance with `make ShowPerformance.mlx`, uncommenting the appropriate lines.
- Be sure to check if you got the right answer! This actually is trickier than you might at first think!

☛ SEE ANSWER

Observations.

- Parallelelizing the third loop is a bit trickier. Likely, you started by inserting the `#pragma` statement and got the wrong answer. The problem is that you are now parallelizing



implemented by

```
void LoopThree( int m, int n, int k, double *A, int ldA, double *Atilde,
    double *Btilde, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MC ) {
    int ib = min( MC, m-i );     /* Last loop may not involve a full block */
    PackBlockA_MCxKC( ib, k, &alpha( i, 0 ), ldA, Atilde );

    LoopTwo( ib, n, k, Atilde, Btilde, &gamma( i,0 ), ldC );
  }
}
```

The problem with a naive parallelization is that all threads pack their block of A into the same buffer `Atilde`. There are two solutions to this:

- Quick and dirty: allocate and free a new buffer in this loop, so that each iteration, and hence each thread, gets a different buffer.
- Cleaner: In Loop 5 around the micro-kernel, we currently allocate one buffer for `Atilde`:

```
  double *Atilde = ( double * ) malloc( MC * KC * sizeof( double ) );
```

What one could do is to allocate a larger buffer here, enough for as many blocks $\widetilde{A}$ as there are threads, and then in the third loop around the micro-kernel one can index into this larger buffer to give each thread its own space. For this, one needs to know about a few OpenMP inquiry routines:

  * `omp_get_num_threads()` returns the number of theads being used in a parallel section.
  * `omp_get_thread_num()` returns the thread number of the thread that calls it.

Now, this is all a bit tricky, because omp_get_num_threads() returns the number of threads being used only in a parallel section. So, to allocate enough space, one would try to change the code in Loop 5 to

```
double *Atilde;

#pragma omp parallel
{
   if ( omp_get_thread_num() == 0 )
     malloc( MC * NC * sizeof( double ) * omp_get_num_threads() );
}
```

Here the curly brackets indicate the scope of the parallel section.

The call to free must then be correspondingly modified so that only thread 0 releases the buffer.

Now, in Loop 2, one can index into this larger buffer.

3.3.5 Speedup, efficiency, and Ahmdahl's law

Amdahl's law is a simple observation regarding how much speedup can be expected when one parallelizes a code. First some definitions.

Sequential time. The *sequential time* for an operation is given by

$$T(n),$$

where n is a parameter that captures the size of the problem being solved. It is equal the time required to compute the operation on a single core. It is implicitly assumed that the implementation is the fastest implementation for a single core.

Parallel time. The *parallel time* for an operation and a given implementation using t threads is given by

$$T_t(n).$$

Speedup. The *speedup* for an operation and a given implementation using t threads is given by

$$T(n)/T_t(n).$$

It measures how much speedup is attained by using t threads. Generally speaking, it is assumed that

$$S_t(n) = T(n)/T_t(n) \leq t.$$

This is not always the case: sometimes t threads collective have resources (e.g., more cache memory) that allows speedup to exceed t. Think of the case where you make a bed with one person versus two people: You can often finish the job more than twice as fast because two people do not need to walk back and forth from one side of the bed to the other.

Efficiency. The *efficiency* for an operation and a given implementation using t threads is given by

$$S_t(n)/t.$$

It measures how much efficiently the t threads are being used.

GFLOPS. We like to report GFLOPS (billions of floating point operations per second). Notice that this gives us an easy way of judging efficiency: If we know what the theoretical peak of a processor is, in terms of GFLOPS, then we can easily judge how efficiently we are using the processor relative to the theoretical peak. How do we compute the theoretical peak? We can look up the number of cycles a core can perform per second and the number of floating point operations it can before per cycle. This can then be converted to a peak rate of computation for a single core (in billions of floating point operations per second). If there are multiple cores, we just multiply this peak by the number of cores.

Ahmdahl's law. Ahmdahl's law is a simple observation about the limits on parallel speed and efficiency. Consider an operation that requires sequential time

$$T$$

for computation. What if fraction f of this time is **not** parallelized, and the other fraction $(1-f)$ is parallelized with t threads. Then the total execution time is

$$T_t \geq (1-f)T/t + fT.$$

This, in turn, means that the speedup is bounded by

$$S \leq \frac{T}{(1-f)T/t + fT}.$$

Now, even if the parallelization with t threads is such that the fraction that is being parallelized takes no time at all ($(1-f)T/t = 0$), then we still find that

$$S \leq \frac{T}{(1-f)T/t + fT} \leq \frac{T}{(fT} = \frac{1}{f}.$$

What we notice is that the maximal speedup is bounded by the inverse of the fraction of the execution time that is not parallelized. So, if 20% of the code is not optimized, then no matter how many threads you use, you can at best compute the result five times faster. This means that one must think about parallelizing all significant parts of a code.

Ahmdahl's law says that if does not parallelize the part of a sequential code in which fraction f time is spent, then the speedup attained regardless of how many threads are used is bounded by $1/f$.

The point is: So far, we have focused on parallelizing the computational part of MMM. We should also parallelize the packing, since that is a nontrivial part of the total execution time.

3.3.6 Optimizing the packing

Some observations:

- If you choose to parallelize Loop 3 around the micro-kernel, then the packing into $\widetilde{A}$ is already parallelized, since each thread packs its own such block. So, you will want to parallelize the packing of $\widetilde{B}$.

- If you choose to parallelize Loop 2 around the micro-kernel, you will also want to parallelize the packing of $\widetilde{A}$.

- Be careful: we purposely wrote the routines that pack $\widetilde{A}$ and $\widetilde{B}$ so that a naive parallelization will give the wrong answer. Analyze carefully what happens when multiple threads execute the loop...

3.3.7 Overlapping data movement with computation

The next observation is that one may want to overlap the movement of data between memory layers. This is accomplished through *prefetching* of data. Prefetching (or, rather, preloading) into registers is not a simple matter: The $m_r \times n_r = 12 \times 4$ kernel already uses all sixteen vector registers.

The fundamental tools for preloading is the prefetch instructions:

```
// prefetch cache line that includes the address pntr to L1 cache
_mm_prefetch( pntr, 0 )

// prefetch cache line that includes the address pntr to L2 cache
_mm_prefetch( pntr, 1 )

// prefetch cache line that includes the address pntr to L2 or L3 cache
_mm_prefetch( pntr, 2 )
```

It is important to replace the allocation routines `malloc` and `free` with `_mm_malloc` and `_mm_free`, respectively, to ensure that data is properly aligned in memory.

Some things to try. You probably want to do this for one core first, before you move on to multiple cores.

- Before computing the update of the $m_r \times n_r$ micro-tile with the micro-kernel, prefetch the next micro-tile to the L3 cache.

- As you compute with the current micro-panel of $\widetilde{A}$, prefetch the next micro-panel of $\widetilde{A}$ into the appropriate cache level.

- As you compute with the current micro-panel of $\widetilde{B}$, prefetch the next micro-panel of $\widetilde{B}$ into the appropriate cache level. Notice that for the entire execution of Loop 1, the same micro-panel of $\widetilde{B}$ is used. So, you want to be selective as to what you prefetch.

3.4 Enrichment

3.5 Wrapup

Week 4

Message-Passing Interface Basics

4.1 Opener

4.1.1 Launch

4.1.2 Outline Week 3

4.1.3 What you will learn

4.2 MPI Basics

4.2.1 Programming distributed memory architectures

These days, the fastest computers in the world are composed of many individual computers, referred to as (computational) nodes, that can communicate through some high bandwidth interconnection network. Each individual node typically consists of a processor with multiple cores, and may have additional high-performance hardware in form of an accelerator like a GPU. We will not further concern ourselves with accelerators, mostly discussing nodes in a relatively abstract way.

To keep the cost of such architectures reasonable, each node is an off-the shelf computer, with its own CPU and memory hierarchy. In other words, each node looks very much like the processors for which you optimized MMM, in prior weeks of this course. Each node is therefore programmed as if it is a separate computer. This model of programming is known as *Multiple Instruction-Multiple Data*, since each node executes its own instruction stream, on data that is stored only in its memory. If data from other nodes is needed, then that data needs to be explicitly communicated between the nodes.

Given that a modern super computer may consist of tens of thousands or even hundreds of thousands of nodes, writing an individual program for each is obviously daunting. For this reason, one tends to write a single program, a copy of which executes on each node. This is known as the *Single Program-Multiple Data* (SPMD) paradigm of parallel programming.

Now, running the exact same program on thousands of nodes is obviously not interesting. What actually is done is that each node is given identifier so that its path through the program depends on that identifier. This also allows a node to communicate data to another node by identifier if data has to be shared.

Since the mid-1990s, SPMD programming is facilitated by a standardized interface, the *Message-Passing Interface* (MPI). While the latest standard (and even the first one) includes a vast number of routines and functionality, one fortunately only needs to know a few to get started.

4.2.2 MPI: Managing processes

MPI views the computation as being performed by a collection of *processes*. A particular node may host one or more such processes. The processes do not migrate between nodes. From a programming point of view, the processes have their separate memory spaces. Thus, from a programming point of view, they might as well be on separate nodes since even processes on the same node cannot access each other's data directly.

Let us start with the customary "Hello World" program in Figure 4.1. Normally, this would be compiled with a C compiler, say `gcc`

```
% gcc HelloWorld.c -o HelloWorld.x
```

and then executed:

```
% ./HelloWorld.x
```

which should yield the output

```
#include <stdio.h>

int main(int argc, char** argv) {
  printf ("Hello World!\n" );
}
```

Figure 4.1: Simple "Hello World!" program.

```
Hello World!
```

if all goes well.

At this point, you have a program that executes on one processor. To make it into a program that executes on many MPI processes, you compile with `mpicc`, which is just a convenient way of using your default compiler, but linking in MPI libraries and functionality.

```
% mpicc HelloWorld.c -o HelloWorld.x
```

To execute the result, type

```
% mpirun -np 4 HelloWorld.x
```

which creates four MPI processes on your computer, thus sort of simulating four compute nodes, and runs the program with each of the processes, yielding the output

```
Hello World!
Hello World!
Hello World!
Hello World!
```

Welcome to your first MPI program! Notice that we are actually simulating a distributed memory computer by running multiple processes on one processor, namely the computer on which you are compiling and executing. However, this is a convenient environment in which to first learn about MPI.

Now, redundantly executing the same program with the same data on multiple processes is rarely productive. Our kind of parallel computing is typically about using multiple processes to more quickly complete a single task.

To be able to take "a different execution path through the same program," a process is assigned a process number from 0 to `nprocs` where `nprocs` equals the number of processes indicated with the `-np` runtime parameter passed to `mpirun`. To be able to access that information, we need to add some commands to our simple program as shown in Figure 4.2. There

```
#include <mpi.h>
```

pulls in the MPI header file.

```
  MPI_Init( &argc, &argv );
```

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv)
{
  int nprocs, me, left, right;

  /* Extract MPI runtime parameters and update argc and argv
     leaving the other runtime parameters. */
  MPI_Init( &argc, &argv );

  /* Extract the number of processes */
  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );

  /* Extract the rank of this process */
  MPI_Comm_rank( MPI_COMM_WORLD, &me );

  /*  Hello! */
  printf( "Hello World from %d of %d processes\n", me, nprocs );

  /* Clean up the MPI environment. */
  MPI_Finalize();
}
```

Figure 4.2: Simple "Hello World!" program.

initializes the MPI environment, using the run time parameters passed to `mpirun`. It then strips out those run time parameters, leaving any additional run time parameters that are to be passed to the program. Eventually

```
  MPI_Finalize();
```

"finalizes" the environment.

MPI employs *object based* programming, which encapsulates detailed information in opague data types (objects), thus hiding intricate details. One important data type is the *communicator*, which encodes information about the group of processes that are involved in a communication between processes. As time goes on, how that fits into the picture will become clear. In our simple program, we only encounter the predefined `MPI_COMM_WORLD` communicator, which describes all processes created when `mpirun` is executed.

Inquiry routines are used to extract information from a communicator.

```
  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
```

sets integer `nprocs` to the number of processes while

```
  MPI_Comm_rank( MPI_COMM_WORLD, &me );
```

sets integer `me` to the rank of the calling process within `MPI_COMM_WORLD`. As you program, you will want to constantly ask "if this process has rank `me`, what computation should it perform?" In other words, `me` and `nprocs` will determine what path through the program will be taken by the process. In the case of this simple program, each process performs a slightly different computation, namely it prints out its rank. When executed with

```
mpicc HelloWorld2.c -o HelloWorld2.x
mpirun -np 4 ./HelloWorld2.x
```

The output might be

```
Hello World from 2 of 4 processes
Hello World from 0 of 4 processes
Hello World from 3 of 4 processes
Hello World from 1 of 4 processes
```

which illustrates that the order in which computation happens is not predetermined.

4.2.3 Sending and receiving

Let's see how we can force the results to be printed in order. There are simpler ways of doing this that presented here. However, the way we proceed allows us to illustrate simple sending and receiving calls.

Here is the idea: Each process with rank i will wait for a message from the process with rank $i-1$, except for the process with rank 0. Process 0 prints its message and then sends a message to process 1. This way each process i waits for the message from process $i-1$, prints its message, and then send a message to process $i+1$, except for the last such process.

This means replacing

```
  printf( "Hello World from %d of %d processes\n", me, nprocs );
```

with

```
  if ( me > 0 )
    MPI_Recv( &token, 1, MPI_INT, me-1, MPI_ANY_TAG, MPI_COMM_WORLD, &status );

  printf( "Hello World from %d of %d processes\n", me, nprocs );
  fflush( stdout );

  if ( me < nprocs-1 )
    MPI_Send( &token, 1, MPI_INT, me+1, 0, MPI_COMM_WORLD );
```

and adding the declarations

```
int token = 1;
MPI_Status status;
```

(Obviously, `token` can be initialized to any value. It is merely data to be passed around. Any data would work.) Take a moment to look at the piece of code. To create it (or interpret it), you need to think "if I am process `me`, what should I do?" The answer is

- If I am not the first process (`me > 0`) then I should wait for a message from the process to my left (`me-1`).

```
if ( me > 0 )
  MPI_Recv( &token, 1, MPI_INT, me-1, MPI_ANY_TAG, MPI_COMM_WORLD, &status );
```

- Once I receive the message, it is my turn to print.

```
  printf( "Hello World from %d of %d processes\n", me, nprocs );
  fflush( stdout );
```

  The call to `fflush` is required because output is typically send to process 0 for printing. Without the `fflush()` it might be buffered by the process that just called `printf`, which then could result in the printed messages still being out of order.

- Once I have printed the message, I need to pass the token to the next process, `me+1`, if there is such a process.

```
if ( me < nprocs-1 )
  MPI_Send( &token, 1, MPI_INT, me+1, 0, MPI_COMM_WORLD );
```

As long as you can learn to reason this way, putting yourself in the place of process `me`, message-passing programming is relatively straight forward.

Homework 4.2.3.1 Copy `HelloWorld2.c` into `HelloWorld3.c` and modify it according to the above description, passing messages to ensure that the output is serialized. Compile it, and execute it!

☛ SEE ANSWER

The above illustrates the use of the simplest MPI send routine:

```
int MPI_Send( void *buff, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm )
```

Here

- `buff` is the address where the data that is communicated is stored.
- `count` equals the number of items to be sent.
- `datatype` is a descriptor for the data type of each item. Commonly used are `MPI_INT` (integer), `MPI_CHAR` (character), and `MPI_DOUBLE` (double precision floating point). One can also custom create so-called *derived* data types, but that goes beyond this course.
- `dest` is the rank of the destination process, relative to the communicator given in `comm`.
- `tag` is a message tag that can be used to distinguish between multiple messages that are sent to the process `dest`.

- `comm` is the communicator that describes the group of processes involved in the communication, of which the sending receiving processes are part.

The simplest receive routine is given by

```
int MPI_Recv( void *buff, int count, MPI_Datatype datatype,
              int src, int tag, MPI_Comm comm, MPI_Status *status )
```

Here

- `buff` is the address where the data that is communicated is received.
- `count` equals the *maximum* number of items that can be received.
- `datatype` is a descriptor for the data type of each item.
- `src` is the rank of the source process, relative to the communicator given in `comm`.
- `tag` is a message tag that can be used to distinguish between multiple messages that are sent to the process.
- `comm` is the communicator that describes the group of processes involved in the communication, of which the sending receiving processes are part.
- `status` is an object that holds information about the received message, including, for example, its actually count.

For both, the return value should be `MPI_SUCCESS` if all went well.

It is important to realize how these operations proceed. The specification for `MPI_Send` says that the operation *may* block execution of the source process until a corresponding `MPI_Recv` happens on the receiving process. The specification for `MPI_Recv` says that the operation *will* block execution of the receiving process. Some scenarios:

- The `MPI_Send` is executed before the `MPI_Recv` happens on the receiving process. In this case *either* the call to `MPI_Send` blocks until the receive is posted *or* the message is stored (by the sending or receiving process) to be later matches with the receive. In this second case, the sending process can proceed with its execution.

  How these operations satisfy this specification depends on the implementation of MPI. The reason why `MPI_Send` *may* block is that this avoids the problem of there not being enough temporary space for the message to be temporarily stored "in flight."

  Notice that if the message is temporarily stored, then an extra memory-to-memory copy of the data is incurred along the way.

- The `MPI_Send` is executed after the `MPI_Recv` is posted. A reasonable MPI implementation would receive the data is directly into the buffer specified in the `MPI_Recv` call. This would usually mean that the communication incurs lower overhead.

4.2.4 Cost of sending/receiving a message

We will adopt the following model for the cost of sending n items between two processes:

$$\alpha + n\beta,$$

where

- α equals the *latency*: the time (in seconds) for the processes to connect.
- β is the cost per item in seconds (the inverse of the bandwidth).

This model is remarkably accurate even for machines with a very large number of nodes.

One may wonder whether this model shouldn't take into account the distance between the two nodes on which the processes exist. The latency (α term) is not particularly sensitive to this distance because the cost of initiating the message is mostly on the sending and receiving side. Forwarding the message in the network is supported by very fast hardware. It is the software overhead on the sending and receiving sides that is more significant. The bandwidth (β) term is not particularly affected by the distance because messages are pipelined.

Let's make this a little clearer. Let's say that a vector x of n items is to be communicated between two nodes. Let's for simplicity sake assume that to get from the sending node to the receiving node, the message has to be forwarded by $d-1$ nodes in between, so that the two nodes are a "distance" d apart. It would seem like the message is then sent from the sending nodes to the first node along the way, at a cost of $\alpha + n\beta$. To send it to the next node would be another $\alpha + n\beta$, and so forth. The total cost would be $d(\alpha + n\beta)$, which would typically be much greater than $\alpha + n\beta$.

Now, what if one breaks the message into k equally sized packets. In a first step, the first packet would be sent from the sending node to the first intermediate node, taking $\alpha + n/k\beta$. As this gets passed along to the next node, the sending process can send the next packet. It takes $d(\alpha + n/k\beta)$ time for the first packet to reach the final destination, and then another $(k-1)(\alpha + n/k\beta)$ time for the final packet to arrive. So the total cost is

$$(d+k-1)(\alpha + \frac{n}{k}\beta) = (d+k-1)\alpha + \frac{d_k - 1}{k}\beta.$$

If k is very large compared to d, then we get, approximately

$$(d+k-1)(\alpha + \frac{n}{k}\beta) = (d+k-1)\alpha + \frac{d+k-1}{k}n\beta \approx (d+k-1)\alpha + n\beta.$$

Now, the fact is that a modern network that connects nodes automatically forward messages within the communication network, rather than explicitly involving the nodes in between. So, one should really think of the steps along the way as the initiation of the communication , at a cost of α, and then a cost of forwarding the message within the network that involves a much smaller latency, α_{net}. The total cost then resembles more

$$\alpha + (d+k-1)\alpha_{\text{net}} + n\beta.$$

In practice, α is many orders of magnitude greater than α_{net}, and therefore a reasonable approximation becomes

$$\alpha + n\beta.$$

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv)
{
  int nprocs, me, dest, my_number=999, incoming=-999;
  MPI_Status status;

  MPI_Init( &argc, &argv );

  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
  MPI_Comm_rank( MPI_COMM_WORLD, &me );

  if ( me == 0 ){
    printf( "Input an integer:" );
    scanf( "%d", &my_number );

    /* send to all other processes */
    for ( dest=1; dest<nprocs; dest++ )
      MPI_Send( &my_number, 1, MPI_INT, dest, 0, MPI_COMM_WORLD );
  }
  else
    MPI_Recv( &incoming, 1, MPI_INT, 0, 0, MPI_COMM_WORLD );


  printf( "me = %d, my_message = %d, incoming = %d\n", me, my_message, incoming );

  MPI_Finalize();
}
```

Figure 4.3: Process zero reads and duplicates to all other processes.

4.3 Collective Communication

4.3.1 MPI_Bcast

It is a frequently encountered situation where one process, often process 0, reads data that is then shared with all other processes. A naive implementation of this is given in Figure 4.3.

Now, if that number of data being shared with all other processes is large and/or the number of processes being used is large, then sending individual messages to all other processes may become time consuming. Also, sharing data that is on one process, the root, with all other processes is a common operation. Perhaps it should be a primitive for broadcasting as part of MPI. Indeed, there is:

```
int MPI_Bcast( void *buff, int count, MPI_Datatype datatype,
                        int root, MPI_Comm comm )
```

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv)
{
  int nprocs, me, dest, my_number=999, incoming=-999;
  MPI_Status status;

  MPI_Init( &argc, &argv );

  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
  MPI_Comm_rank( MPI_COMM_WORLD, &me );

  if ( me == 0 ){
    printf( "Input an integer:" );
    scanf( "%d", &my_number );

    MPI_Bcast( &my_number, 1, MPI_INT, 0, MPI_COMM_WORLD );
  }
  else
    MPI_Bcast( &incoming, 1, MPI_INT, 0, MPI_COMM_WORLD );

  printf( "me = %d, my_message = %d, incoming = %d\n", me, my_message, incoming );

  MPI_Finalize();
}
```

Figure 4.4: Process zero reads and broadcasts to all other processes.

Here

- `buff` is the address where the data to be send/received is stored. On the root process, it is the address where data to be communicated is stored. On the other processes, it is where the data is to be received.
- `count` equals the *maximum* number of items being communicated.
- `datatype` is a descriptor for the data type of each item.
- `root` is the rank of the sending process, relative to the communicator given in `comm`.
- `comm` is the communicator that describes the group of processes involved in the communication, of which the sending receiving processes are part.

The operation is synchronous: all processes have to call it for it to complete successfully. With this, we can recode the example in Figure 4.3 as given in Figure 4.4.

4.3.2 Minimum spanning tree broadcast (MST Bcast)

Let us now discuss how to implement an efficient broadcast algorithm. We will illustrate this by implementing a routine

```
int My_Bcast( void *buff, int count, MPI_Datatype datatype,
               int root, MPI_Comm comm )
```

The most naive implementation sends individual messages from the root node to the other nodes:

```
int My_Bcast( void *buff, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm )
{
  int me, nprocs, dest, bcast_tag=99999;
  MPI_Status status;

  MPI_Comm_size( comm, &nprocs );
  MPI_Comm_rank( comm, &me );

  if ( me == root )
    for ( dest=0; dest<nprocs; dest++)
      if ( dest != root )
      MPI_Send( buff, count, datatype, dest, bcast_tag, comm );
  else
    MPI_Recv( buff, count, datatype, root, bcast_tag, comm, &status );
}
```

In our subsequent discussion, we are going to assume that at any given time, a process can only send one message to one other process and receive one message from another process. The problem with the naive implementation is that under our model the cost is

$$(p-1)\alpha + (p-1)n\beta,$$

where p equals the number of processes (`nprocs`) and n is the size of the message (`count`).

Can we do better? We next describe what is often referred to as a *minimum spanning tree* algorithm for broadcasting. The algorithm is broken down into steps, during each of which we can do a round of communication where each process can send and receive at most one message. When it starts, only one process, the root, has the message. In the first step, it sends this message to some other process, at a cost of $\alpha + n\beta$. At this point, the set of processes are partitioned into two roughly equal components (disjoint subsets), where the root and the destination of this first message are in separate components. These two processes now become the roots for two separate broadcasts, one in each component.

Proceedings this way, we find that after one communication step two processes have the message. After two communication steps, four processes have the message. After k communication steps, 2^k processes have the message. This way, it will take $\lceil \log_2(p) \rceil$ communication steps for all

p processes to get the message. Here $\lceil x \rceil$ is equals the *ceiling* of x, meaning the smallest integer greater than or equal to x. The cost of this approach is now

$$\lceil \log_2(p) \rceil(\alpha + n\beta) = \lceil \log_2(p) \rceil \alpha + \lceil \log_2(p) \rceil n\beta,$$

which is better than $(p-1)\alpha + (p-1)n\beta$ if $p \geq 4$ and much better if p is large.

How might we implement this? The above description is inherently recursive: the broadcast is described as partitioning of the set of processes, a communication, and then a broadcast within the components. It is usually best to have the implementation reflect how the algorithm is naturally described. So, we start with p processes indexed 0 through $p-1$. Partitioning these into two components can be accomplished by computing the index of the middle process: `mid = p/2` (integer division). If the root is in the set $\{0,\ldots,\texttt{mid}\}$, then a destination (`new_root`) is picked in the set $\{\texttt{mid}+1,\ldots,p-1\}$. Otherwise, a destination (`new_root`) is picked from the set $\{0,\cdots,\texttt{mid}\}$. The message is sent from `root` to `new_root`, and the broadcast continues in the two separate components.

Figure 4.5 translate the observations into code. Here `new_root` is taken to equal the left-most process if the current root is in the right component and the right-most process if the current root is in the left component. There are other choices that could be considered. Notice that to facilitate the recursion, we create an "auxiliary" routine that has two extra parameters: the ranks of the left-most and right-most processes that are participating in the broadcast. Initially, `left` equals zero while `right` equals $p-1$. As the recursion unfolds, the broadcast is called on the left and right partition relative to what processes are participating. Notice that the recursive call only needs to happen for the component of which the calling process (`me`) is part, since the calling process will never participate in a send or receive for a component of which it is not part.

Homework 4.3.2.1 Examine file `test_bcast.c` to see how it tests the `MST_Bcast` implementation. Then compile and execute it with

```
mpicc MST_Bcast.c test_bcast.c -o test_bcast.x
mpirun -np test_bcast.x
```

☛ SEE ANSWER

4.3.3 Can we do better yet?

A lot of people believe that the MST broadcast is the best algorithm. In order to answer such question, it always pays to ask the question "What is the lower bound for the time required to ..." In this case, we want a reasonably tight lower bound for the broadcast operations involving a message of size n among p processes.

Under our model (a message costs time $\alpha + n\beta$ and a process can only send to one process at a time and can only receive from one process at a time), we can make two observations:

- Broadcasting requires at least $\lceil \log_2 p \rceil$ steps and hence a lower bound on the latency (α) term is $\lceil \log_2 p \rceil \alpha$.
- Broadcasting among at least two processes requires a β term of at least $n\beta$ since the message must leave the root node.

```c
#include <mpi.h>

int MST_Bcast_aux( void *, int, MPI_Datatype, int, MPI_Comm, int, int );

int MST_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm )
{
  int nprocs;

  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );

  MST_Bcast_aux( buf, count, datatype, root, comm, 0, nprocs-1 );
}

int MST_Bcast_aux( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm,
                  int left, int right )
{
  int me, mid, new_root, bcast_tag=9999;
  MPI_Status status;

  if ( left != right ){
    MPI_Comm_rank( MPI_COMM_WORLD, &me );

    mid = ( left + right ) / 2;
    new_root = ( root <= mid ? right : left );

    if ( me == root )
      MPI_Send( buf, count, datatype, new_root, bcast_tag, comm );

    if ( me == new_root )
      MPI_Recv( buf, count, datatype, root, bcast_tag, comm, &status );

    if ( me <= mid )  // me is in left component
      MST_Bcast_aux( buf, count, datatype,
                    ( root <= mid ? root : new_root ), comm, left, mid );
    else              // me is in right component
      MST_Bcast_aux( buf, count, datatype,
                    ( root <= mid ? new_root : root ), comm, mid+1, right );
  }
}
```

Figure 4.5: Minimum Spanning Tree broadcast.

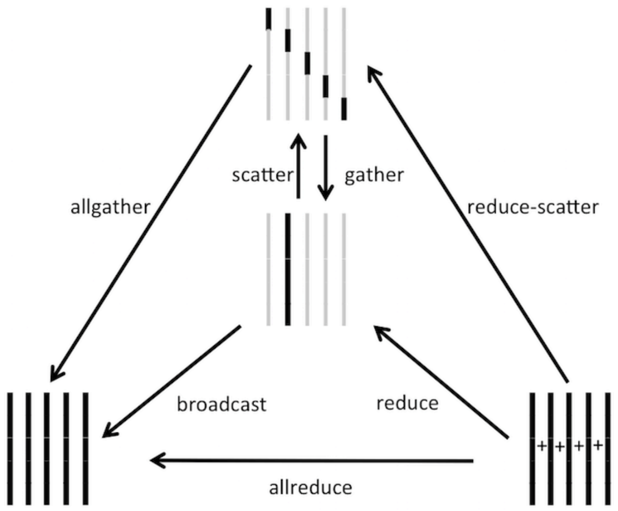


Figure 4.6: The seven collective communications discussed in this week.

This does not mean that a lower bound is given by $\lceil \log_2 p \rceil \alpha + n\beta$. After all, our model doesn't say anything about overlapping latency with communication. What we observe is that under our model the MST broadcast attains the lower bound on the α term, but is off by a factor $\log_2 p$ when it comes to the β term. So, if very little is being communicated, it should be the algorithm of choice (again: under our model of communication cost). If the message is long (n is large) then there may be an opportunity for improvement.

We are going to move on to other collective communication operations, and then will return to better algorithms for broadcast when the size of the message is large.

4.3.4 Various collective communications

In this unit, we discuss seven commonly encountered collective communications. How they are related is illustrated in Figure 4.6.

Broadcast. A vector of data, x, of size n items, starts on the root process. Upon completion, the broadcast leaves a copy of the data with each process.

Operation	Before				After			
	Process 0	Process 1	Process 2	Process 3	Process 0	Process 1	Process 2	Process 3
Broadcast	x				x	x	x	x

Scatter. The scatter operation again starts with a vector, x, of size n items, on the root process. This time, the vector is viewed as being partitioned into p subvectors (components),

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix},$$

where x_0 has n_i items, with $\sum_{i=0}^{p-1} n_i = n$. Upon completion, subvector x_i resides with process i.

Operation	Before				After			
	Process 0	Process 1	Process 2	Process 3	Process 0	Process 1	Process 2	Process 3
Scatter	x_0				x_0			
	x_1					x_1		
	x_2						x_2	
	x_3							x_3

Gather. The gather operation is the inverse of the scatter operation. Upon commencement each process i owns a subvector of data x_i of size n_i. Upon completion, one process, the root, owns the complete vector of data x.

Operation	Before				After			
	Process 0	Process 1	Process 2	Process 3	Process 0	Process 1	Process 2	Process 3
Gather	x_0				x_0			
		x_1			x_1			
			x_2		x_2			
				x_3	x_3			

Allgather. The allgather operation is like the gather operation, except that a copy of the gathered vector is left on all processes.

Operation	Before				After			
	Process 0	Process 1	Process 2	Process 3	Process 0	Process 1	Process 2	Process 3
Allgather	x_0				x_0	x_0	x_0	x_0
		x_1			x_1	x_1	x_1	x_1
			x_2		x_2	x_2	x_2	x_2
				x_3	x_3	x_3	x_3	x_3

Reduce(to-one). Reduce operations start with (different) vectors of the same size on each of the processes, and reduce these vectors element-wise. The simplest reduce operation sums the elements. Let us examine four vectors of length 8 each:

$$x^{(0)} = \begin{pmatrix} 0.0 \\ 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \\ 5.0 \\ 6.0 \\ 7.0 \end{pmatrix}, x^{(1)} = \begin{pmatrix} 0.1 \\ 1.1 \\ 2.1 \\ 3.1 \\ 4.1 \\ 5.1 \\ 6.1 \\ 7.1 \end{pmatrix}, x^{(2)} = \begin{pmatrix} 0.2 \\ 1.2 \\ 2.2 \\ 3.2 \\ 4.2 \\ 5.2 \\ 6.2 \\ 7.2 \end{pmatrix}, \text{ and } x^{(3)} = \begin{pmatrix} 0.3 \\ 1.3 \\ 2.3 \\ 3.3 \\ 4.3 \\ 5.3 \\ 6.3 \\ 7.3 \end{pmatrix}$$

then

$$\begin{aligned} \textstyle\sum_j x^{(j)} &= x^{(0)} + x^{(1)} + x^{(2)} + x^{(3)} \\ &= \begin{pmatrix} 0.0 \\ 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \\ 5.0 \\ 6.0 \\ 7.0 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 1.1 \\ 2.1 \\ 3.1 \\ 4.1 \\ 5.1 \\ 6.1 \\ 7.1 \end{pmatrix} + \begin{pmatrix} 0.2 \\ 1.2 \\ 2.2 \\ 3.2 \\ 4.2 \\ 5.2 \\ 6.2 \\ 7.2 \end{pmatrix} + \begin{pmatrix} 0.3 \\ 1.3 \\ 2.3 \\ 3.3 \\ 4.3 \\ 5.3 \\ 6.3 \\ 7.3 \end{pmatrix} = \begin{pmatrix} 0.0+0.1+0.2+0.3 \\ 1.0+1.1+1.2+1.3 \\ 2.0+2.1+2.2+2.3 \\ 3.0+3.1+3.2+3.3 \\ 4.0+4.1+4.2+4.3 \\ 5.0+5.1+5.2+5.3 \\ 6.0+6.1+6.2+6.3 \\ 7.0+7.1+7.2+7.3 \end{pmatrix}. \end{aligned}$$

The reduce(-to-one) operation performs this operation across processes, leaving the result only at an indicated root node:

Operation	Before				After			
	Process 0	Process 1	Process 2	Process 3	Process 0	Process 1	Process 2	Process 3
Reduce(-to-one)	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$			

Reduce-scatter. The reduce-scatter operation similarly computes an element-wise reduction of vectors, but this time leaves the result scattered among the processes.

In out previous example, the operation my be illustrated by

	Process 0	Process 1	Process 2	Process 3
Before	0.0	0.1	0.2	0.3
	1.0	1.1	1.2	1.3
	2.0	2.1	2.2	2.3
	3.0	3.1	3.2	3.3
	4.0	4.1	4.2	4.3
	5.0	5.1	5.2	5.3
	6.0	6.1	6.2	6.3
	7.0	7.1	7.2	7.3
After	$0.0+0.1+0.2+0.3$			
	$1.0+1.1+1.2+1.3$			
		$2.0+2.1+2.2+2.3$		
		$3.0+3.1+3.2+3.3$		
			$4.0+4.1+4.2+4.3$	
			$5.0+5.1+5.2+5.3$	
				$6.0+6.1+6.2+6.3$
				$7.0+7.1+7.2+7.3$

Thus, each process i starts with an equally sized vector $x^{(i)}$ which is partitioned as

$$x^{(i)} = \begin{pmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \vdots \\ x_{p-1}^{(i)} \end{pmatrix}.$$

Upon completion, each process i ends up with the result of computing $\sum_{j=0}^{p-1} x_i^{(j)}$ as illustrated by

Operation	Before				After			
	Process 0	Process 1	Process 2	Process 3	Process 0	Process 1	Process 2	Process 3
Reduce-scatter	$x_0^{(0)}$	$x_0^{(1)}$	$x_0^{(2)}$	$x_0^{(3)}$	$\sum_j x_0^{(j)}$			
	$x_1^{(0)}$	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(3)}$		$\sum_j x_1^{(j)}$		
	$x_2^{(0)}$	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(3)}$			$\sum_j x_2^{(j)}$	
	$x_3^{(0)}$	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(3)}$				$\sum_j x_3^{(j)}$

Allreduce. Finally, there is the allreduce operation, which element-wise reduces vectors, leaving the result duplicated on all processes.

Operation	Before				After			
	Process 0	Process 1	Process 2	Process 3	Process 0	Process 1	Process 2	Process 3
Allreduce	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$

4.4 More Algorithms for Collective Communication

4.4.1 Minimum Spanning Tree scatter and gather algorithms

We describe what is often referred to as a *minimum spanning tree* algorithm for the scatter operation. As for the MST broadcast, the algorithm is broken down into steps, during each of which we can do a round of communication where each process can send and receive at most one message. When it starts, only one process, the root, has the message. In the first step, it partitions the processes into two subsets (components) and sends from the root only the part of the data that needs to end up in the "other component" to some process in that component. At this point, the set of processes is partitioned into two roughly equal components (disjoint subsets), where the root and the destination of this first message are in separate components. These two processes now become the roots for two separate scatter operations, one in each component.

Let us illustrate how to implement a scatter operation by simplifying the operation slightly. We will implement

```
int MST_Scatterx( int *buf, int *displs,
                  int root, MPI_Comm comm )
```

This routine communicates items of type `int`.

- On the root process, `buf` is the address where the data to be scattered is stored. On the other processes, it is a temporary buffer of the same size that can be used as part of the implementation *and* the part that is meant for process `i` is left starting at `&buf[ displ[ i ] ]`.

- `displs` is an array of $p+1$ integers that are displacements into the send buffer that indicate where the part that ends up at each process starts. `buf[ i ]` through `buf[ displs[ i+1 ]-1 ]` will end up on process i. All processes start with a copy of this array.

- `root` is the rank of sending process.

- `comm` is the communicator.

Notice in particular that the displacement has one extra entry, `displs[ nprocs]` that indicates where the element *after* the part to be sent to process `nprocs-1` starts. Usually, we would expect `displs[ 0 ]` to equal 0. This makes calculations of what to send simpler. An implementation of `MST_Scatterx` closely mirrors that of `MST_Bcast` and is given in Figure 4.7.

```c
#include <mpi.h>
int MST_Scatterx_aux( int *, int *, int, MPI_Comm, int, int );

int MST_Scatterx( int *buf, int *displs, int root, MPI_Comm comm)
{
  int nprocs;

  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
  MST_Scatterx_aux( buf, displs, root, comm, 0, nprocs-1 );
}

int MST_Scatterx_aux( int *buf, int *displs,
          int root, MPI_Comm comm, int left, int right )
{
  int me, mid, new_root, scat_tag=9999;
  MPI_Status status;

  if ( left != right ){
    MPI_Comm_rank( MPI_COMM_WORLD, &me );

    mid = ( left + right ) / 2;
    new_root = ( root <= mid ? right : left );

    if ( me == root )
      if ( me <= mid )
        MPI_Send( &buf[ displs[ mid+1 ] ], displs[ right+1 ] - displs[  mid+1 ],
                  MPI_INT, new_root, scat_tag, comm );
      else
        MPI_Send( &buf[ displs[ left ] ], displs[ mid+1 ] - displs[ left ],
                  MPI_INT, new_root, scat_tag, comm );

    if ( me == new_root )
      if ( me > mid )
  MPI_Recv( &buf[ displs[ mid+1 ] ], displs[ right+1 ] - displs[ mid+1 ],
                  MPI_INT, root, scat_tag, comm, &status );
      else
  MPI_Recv( &buf[ displs[ left ] ], displs[ mid+1 ] - displs[ left ],
                  MPI_INT, root, scat_tag, comm, &status );

    if ( me <= mid )  //me is in left component
      MST_Scatterx_aux( buf, displs,
                        ( root <= mid ? root : new_root ), comm, left, mid );
    else              // me is in right component
      MST_Scatterx_aux( buf, displs,
                        ( root <= mid ? new_root : root ), comm, mid+1, right );
  }
}
```

Figure 4.7: Minimum Spanning Tree scatter.

Homework 4.4.1.1 Examine the implementation in `MST_Scatterx.c` and test it with the driver in `test_scatter.c` by executing

```
mpicc test_scatter.c MST_Scatterx.c -o test_scatter.x
mpirun -np 4 test_scatter.x
```

☛ SEE ANSWER

So what is the cost of the MST scatter algorithm under out model? Let's make the simplification that the number of processes, p, is a power of two, so that $\log_2 p$ is an integer. If n data items are to be scattered, and we assume that each process in the end receives n/p data items, then

- In the first communication step, $n/2$ items are sent by the root.
- In the second communication step, $n/4$ items are sent by each of the two roots.
- In the kth communication step, $n/(2^k)$ items are sent by each of the 2^{k-1} roots.

Thus, the total cost is given by

$$\sum_{k=1}^{\log_2 p} \left(\alpha + \frac{n}{2^k} \beta \right) = \log_2 p \alpha + \frac{p-1}{p} n \beta.$$

What is the lower bound on the cost? Under the same assumptions, one can argue that the lower bound on the α term is given by

$$\log_2 p \alpha$$

while the lower bound on the β term is

$$\frac{p-1}{p} n \beta.$$

Why? Because there are at least $\log_2 p$ communication steps, each of which incurs a startup cost of α. Also, all but n/p items must leave the root proces and hence the beta term is at least $(n - n/p)\beta$, which is equal to $(p-1)/p \, n\beta$. We conclude that the MST scatter algorithm is optimal in both the α term and β term.

We now turn to the gather operation, which reverses the communication performed by the scatter operation. We leave it as an exercise to implement this as

```
int MPI_Gatherx( int *buf, int *displs,
int root, MPI_Comm comm )
```

This routine communicates items of type `int`.

- On the root process, `buf` is the address where the data is returned upon completion. On the other processes, it is a temporary buffer of the same size that can be used as part of the implementation *and* the part that is contributed by process `i` starts at `&buf[ displ[ i ] ]`.
- `displs` is an array of $p+1$ integers that are displacements into the buffer that indicate where the part that starts at each process exists. `buf[ i ]` through `buf[ displs[ i+1 ]-1 ]` are contributed by process i. All processes start with a copy of this array.

- `root` is the rank of the root of the gather.
- `comm` is the communicator.

Homework 4.4.1.2 Copy `MST_Scatterx.c` into `MST_Gatherx.c` and modify it to implement a gather to the root. You can test this with the driver in `test_gather.c` by executing

```
mpicc test_gather.c MST_Gatherx.c -o test_gather.x
mpirun -np 4 test_gather.x
```

☛ SEE ANSWER

Homework 4.4.1.3 Give the time required for gathering a vector with n items using the MST gather algorithm. Give the best lower bound that you can justify.

☛ SEE ANSWER

What we notice about the scatter and gather operations is that they are inverses of each other. If one has an algorithm for one, the communication can be reversed to yield an algorithm for the other. We are going to call these two operations *duals* of each other.

4.4.2 MST broadcast and reduce(-to-one)

We have already discussed the MST broadcast algorithm. We now discuss a MST algorithm for its dual operation: the reduce(-to-one). While in the broadcast, we start with a vector of size n items on the root and finish with vectors of size n items on each of the processes (where these vectors are duplicates of the original vector), The reduce-to-one inverts this operation: it starts with vectors of size n items on each of the processes and reduces these element-wise to one vector, leaving the result at the root. What is important is that the reduction operator commutes, as is the case for summation.

The broadcast algorithm partitions the processes into two components, sends the message from the root to the "other component" and then proceeds recursively with broadcasts in each of the two components. A MST reduce-to-one can be implemented by partitioning the processes into two components, identifying a new root in the "other component," recursively performing a reduce-to-one within each of the two components, and finishing by sending the result from the new root to the root, reducing the result upon receipt.

Consider now the routine

```
int MST_Sum_to_One( int *buf, int count, int *tempbuf,
int root, MPI_Comm comm )
```

that sums arrays of double precision numbers. Here

- `buf` is the address where the contribution of the calling process is stored. On the root, upon completion of the operation, it will hold the result of element-wise adding the vectors from the different processes. It should not be changed on the other processes.

- `count` equals the *maximum* number of items in the vectors.
- `tempbuf` is a temporary buffer of size `count` that can be used as part of the communication/-computation.
- `root` is the rank of the root process, relative to the communicator given in `comm`.
- `comm` is the communicator that describes the group of processes involved in the communication, of which the sending receiving processes are part.

Homework 4.4.2.1 Copy `MST_Bcast.c` into `MST_Sum.c` and modify it to implement a MST algorithm for summing to the root. You can test this with the driver in `test_mst_sum.c`.
☛ SEE ANSWER

Homework 4.4.2.2 Give the time required for summing a vector with n items using the MST sum algorithm. Assume that adding two floating point numbers requires time γ. Give the best lower bound that you can justify.
☛ SEE ANSWER

4.4.3 Bucket allgather and reduce-scatter

We now describe a pair of algorithms that we call "bucket" algorithms. For the allgather, the idea is to view the processes as a ring, with process $(i+1) \bmod p$ to the right of process i. Each process starts with a bucket that contains its contribution to the allgather, and in each step these buckets are shifted to the right like a circular conveyer belt. We illustrate the approach with five processes in Figure 4.8.

The bucket allgather requires $p-1$ communication steps, each of which takes $\alpha + n/p\beta$ time if each subvector is of size n/p. The total cost is then

$$(p-1)\alpha + \frac{p-1}{p} n\beta.$$

As before, we can easily argue that the lower bound on the α term is $\log_2 p\alpha$. Thus the bucket algorithm is far from optimal in that respect. We will fix this later. Like for the gather operation, it can be argued that each process needs to receive $p-1$ packets of size n/p and hence the beta term achieves the lower bound under our model.

An implementation of the discussed algorithm for is given in Figure 4.9. For that routine,

- `buf` is the address where upon completion the data is returned upon completion. When the routine is called, it also holds on each process i the part contributed by that process, starting at `&buf[ displ[ i ] ]`.
- `displs` is an array of $p+1$ integers that are displacements into the buffer that indicate where the part that starts at each process exists. `buf[ i ]` through `buf[ displs[ i+1 ]-1 ]` are contributed by process i. All processes start with a copy of this array.

	Process 0	Process 1	Process 2	Process 3	Process 4
	$x_0 \longrightarrow$				
		$x_1 \longrightarrow$			
Step 1			$x_2 \longrightarrow$		
				$x_3 \longrightarrow$	
					$x_4 \longrightarrow$
	x_0	$x_0 \longrightarrow$			
		x_1	$x_1 \longrightarrow$		
Step 2			x_2	$x_2 \longrightarrow$	
				x_3	$x_3 \longrightarrow$
	$x_4 \longrightarrow$				x_4
	x_0	x_0	$x_0 \longrightarrow$		
		x_1	x_1	$x_1 \longrightarrow$	
Step 3			x_2	x_2	$x_2 \longrightarrow$
	$x_3 \longrightarrow$			x_3	x_3
	x_4	$x_4 \longrightarrow$			x_4
	x_0	x_0	x_0	$x_0 \longrightarrow$	
		x_1	x_1	x_1	$x_1 \longrightarrow$
Step 4	$x_2 \longrightarrow$		x_2	x_2	x_2
	x_3	$x_3 \longrightarrow$		x_3	x_3
	x_4	x_4	$x_4 \longrightarrow$		x_4
	x_0	x_0	x_0	x_0	x_0
	x_1	x_1	x_1	x_1	x_1
	x_2	x_2	x_2	x_2	x_2
	x_3	x_3	x_3	x_3	x_3
	x_4	x_4	x_4	x_4	x_4

Figure 4.8: Bucket algorithm for allgather on five processes.

```c
#include <mpi.h>

int Bucket_Allgatherx( int *buf, int *displs, MPI_Comm comm)
{
  int me, nprocs, left, right, inbucket, outbucket, allgather_tag=9999;
  MPI_Status status;
  MPI_Request request;

  MPI_Comm_rank( comm, &me );
  MPI_Comm_size( comm, &nprocs );

  left  = ( me - 1 + nprocs ) % nprocs;
  right = ( me + 1 )          % nprocs;

  inbucket  = left;
  outbucket = me;

  for ( int step=1; step<nprocs; step++ ){
    // Post receive for incoming bucket
    MPI_Irecv( &buf[ displs[ inbucket ] ], displs[ inbucket+1 ] - displs[ inbucket ],
         MPI_INT, left, allgather_tag, comm, &request );

    // Send outgoing bucket
    MPI_Send( &buf[ displs[ outbucket ] ],  displs[ inbucket+1 ] - displs[ inbucket ],
        MPI_INT, right, allgather_tag, comm );

    //
    MPI_Wait( &request, &status );

    outbucket = inbucket;
    inbucket = ( inbucket-1+nprocs ) % nprocs;
  }
}
```

Figure 4.9: Bucket allgather.

- `comm` is the communicator.

Homework 4.4.3.1 If a send occurs before the corresponding receive, the message is typically buffered, which means that an extra memory-to-memory copy happens when the receive is matched to the message. If it is guaranteed that the receive was posted before the send is called, a special routine known as a ”ready” send can be called, `MPI_Rsend`. Read up on this send routine. Then copy `Bucket_Allgatherx.c` into `Bucket_Allgatherx_opt.c` and modify it so that a call to `MPI_Rsend` can be used. If you are careful, each process should only have to send one message to the neighbor to its left to guarantee all receives will be posted before the corresponding send commenses.

☛ SEE ANSWER

The reduce-scatter is the dual of the allgather. In Figure 4.10, we illustrate a bucket algorithm for reduce-scatter. While it is possible to instead send message to the right at each step, as we did for the allgather, we want to emphasize that *if* one has an algorithm for one of the two duals, then the algorithm for the other one can be obtained by reversing the communication (and, possibly, adding a reduction operation). For this reason, all communication is to the left in our example.

Homework 4.4.3.2 Use the illustration in Figure 4.10 to implement a sum-scatter algorithm. You may want to start by copying `Bucket-Allgather.c` to `Bucket-Sum-scatter.c`. Assume the data type is double precision float.

☛ SEE ANSWER

Homework 4.4.3.3 Analyze the cost of the bucket reduce-scatter as well as the lower bound.

☛ SEE ANSWER

4.5 A Calculus of Collective Communication

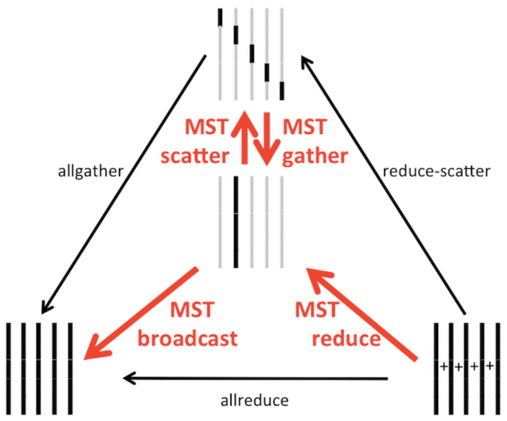
We have discussed a number of algorithms: four MST algorithms, for broadcast, reduce, scatter, and gather, and two bucket algorithms, for allgather and reduce-scatter. We are now going to discuss a family of algorithms that build upon these. The idea is that some algorithms are appropriate for the case where relatively little data is involved, while others are better suited for the case where a large amount of data is communicated. Eventually, we discuss how to then cover the range in between.

4.5.1 The short vector case

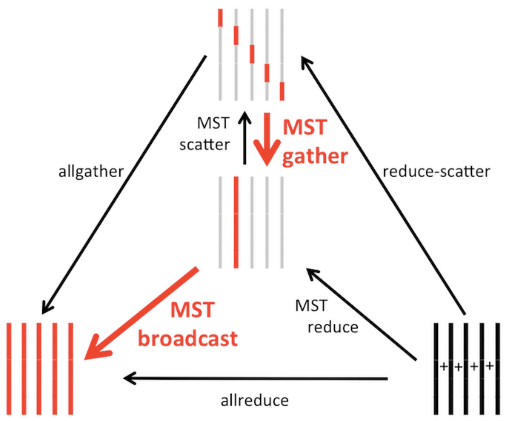
When the vector of data being communication is short, it is the α term of the cost that dominates. It is easy to reason, under our model of communication, that for all seven collectives we have discussed a lower bound on the α term is given by $\log_2(p)\alpha$. The reasoning is simple: All operations require all processes to communication with each other. It takes $\log_2(p)$ communication steps for this to happen. Hence the $\log_2(p)\alpha$ lower bound.

	Process 0		Process 1		Process 2		Process 3		Process 4	
	$x_0^{(0)}$		$x_0^{(1)}$		$x_0^{(2)}$		$x_0^{(3)}$	$\longleftarrow$	$x_0^{(4)}$	
	$x_1^{(0)}$		$x_1^{(1)}$		$x_1^{(2)}$		$x_1^{(3)}$		$x_1^{(4)}$	$\longleftarrow$
Step 1	$x_2^{(0)}$	$\longleftarrow$	$x_2^{(1)}$		$x_2^{(2)}$		$x_2^{(3)}$		$x_2^{(4)}$	
	$x_3^{(0)}$		$x_3^{(1)}$	$\longleftarrow$	$x_3^{(2)}$		$x_3^{(3)}$		$x_3^{(4)}$	
	$x_4^{(0)}$		$x_4^{(1)}$		$x_4^{(2)}$	$\longleftarrow$	$x_4^{(3)}$		$x_4^{(4)}$	
	$x_0^{(0)}$		$x_0^{(1)}$		$x_0^{(2)}$	$\longleftarrow$	$x_0^{(3:4)}$			
			$x_1^{(1)}$		$x_1^{(2)}$		$x_1^{(3)}$	$\longleftarrow$	$x_1^{(4:0)}$	
Step 2	$x_2^{(0:1)}$				$x_2^{(2)}$		$x_2^{(3)}$		$x_2^{(4)}$	$\longleftarrow$
	$x_3^{(0)}$	$\longleftarrow$	$x_3^{(1:2)}$				$x_3^{(3)}$		$x_3^{(4)}$	
	$x_4^{(0)}$		$x_4^{(1)}$	$\longleftarrow$	$x_4^{(2:3)}$				$x_4^{(4)}$	
	$x_0^{(0)}$		$x_0^{(1)}$	$\longleftarrow$	$x_0^{(2:4)}$					
			$x_1^{(1)}$		$x_1^{(2)}$	$\longleftarrow$	$x_1^{(3:0)}$			
Step 3					$x_2^{(2)}$		$x_2^{(3)}$	$\longleftarrow$	$x_2^{(4:1)}$	
	$x_3^{(0:2)}$						$x_3^{(3)}$		$x_3^{(4)}$	$\longleftarrow$
	$x_4^{(0)}$	$\longleftarrow$	$x_4^{(1:3)}$						$x_4^{(4)}$	
	$x_0^{(0)}$	$\longleftarrow$	$x_0^{(1:4)}$							
			$x_1^{(1)}$	$\longleftarrow$	$x_1^{(2:0)}$					
Step 4					$x_2^{(2)}$	$\longleftarrow$	$x_2^{(3:1)}$			
							$x_3^{(3)}$	$\longleftarrow$	$x_3^{(4:2)}$	
	$x_4^{(0:3)}$								$x_4^{(4)}$	$\longleftarrow$
	$x_0^{(0:4)}$									
			$x_1^{(1:0)}$							
					$x_2^{(2:1)}$					
							$x_3^{(3:2)}$			
									$x_4^{(4:3)}$	

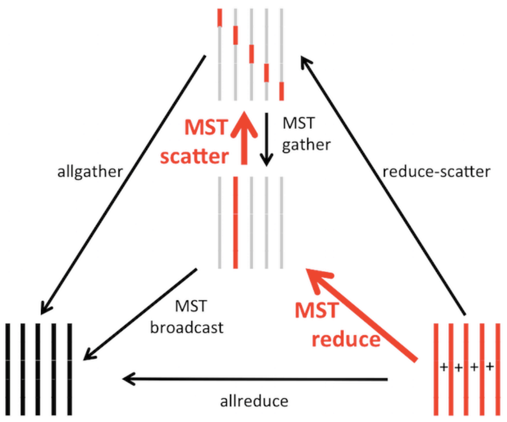
Figure 4.10: Bucket algorithm for sum-scatter on five processes. Here $x_i^{(j:k)}$ stands for $x_i^{(j)} + \cdots + x_i^{(k)}$ if $j < k$ and $x_i^{(j)} + \cdots + x_i^{(4)} + x_i^{(0)} + \cdots + x_i^{(k)}$ if $j > k$.



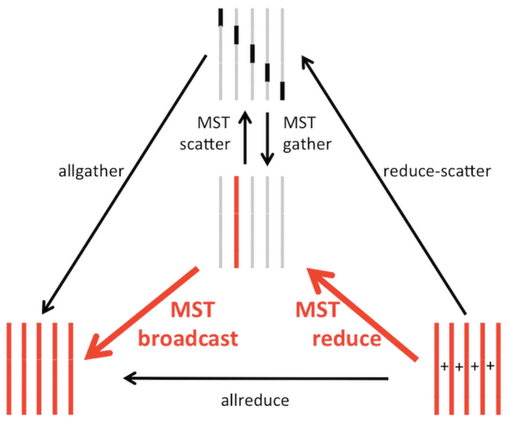
(a) MST algorithms.



(b) Gather/broadcast algorithm for allgather.



(c) Reduce/scatter algorithm for reduce-scatter.



(d) Reduce/broadcast algorithm for allreduce.

Figure 4.11: Algorithms for communicating with short vectors.

We have discussed four algorithms that attain this lower bound: MST broadcast, MST reduce, MST scatter, and MST gather. We highlight these in Figure 4.11 (a). Next, we observe that one can compose these algorithms to achieve algoirthms for the other three operations, as illustrated in Figure 4.11 (b)–(d). All of these derived algorithms have an α term of $2\log_2(p)\alpha$, and hence twice the lower bound under our model.

Homework 4.5.1.1 Which of the following is the result of composing the indicated collective communications (within the same group of processes). (Assume that the subvectors involved are of size n/p.)

1. Scatter followed by gather.
2. Gather followed by scatter.
3. Reduce(-to-one) followed by broadcast.
4. Reduce(-to-one) followed by scatter.
5. Reduce(-to-one) followed by point-to-point.
6. Gather

Notice that there isn't necessarily a single correct answer, or a complete answer. Just think about the options.

☛ SEE ANSWER

4.5.2 Leveraging ready-send for composed short-vector algorithms

A typical implementation of `MPI_Send` uses a "three pass handshake." What we mean by this is that the sending process sends an initial message to the receiving process to check initiate. Upon receipt, the receiving message prepares to receive the message (e.g., checks if space is available to buffer it if the corresponding receive has not been posted) and send an acknowledgment to the sender. The third message then communicates the actual data.

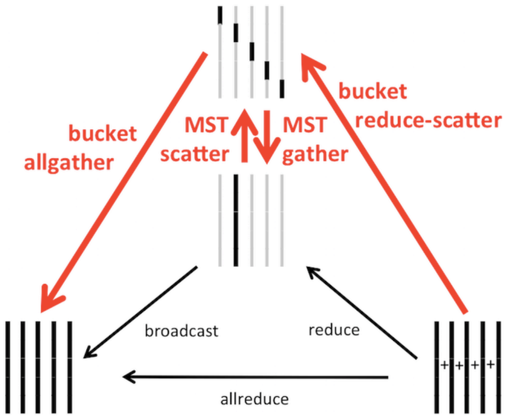
As mentioned before, the message can be sent with a *ready* send (`MPI_Rsend`) if it is known that the receive has already been posted. This variant of sending can use a "one pass handshake." What this means is that the latency is, roughly, one third of that of `MPI_Send`. This, in turn, means that careful implementations of the three derived algorithms (gather/broadcast, reduce/scatter, reduce/broadcast) have an α term of $\frac{4}{3}\alpha$. The idea is to post all the receives for the second MST algorithm up front. The execution of the first MST algorithm also guarantees that ready send can be used during the execution of the second algorithm.

Homework 4.5.2.1 Implement gather/broadcast, reduce/scatter, and reduce/broadcast algorithms with `MPI_Rsend`.

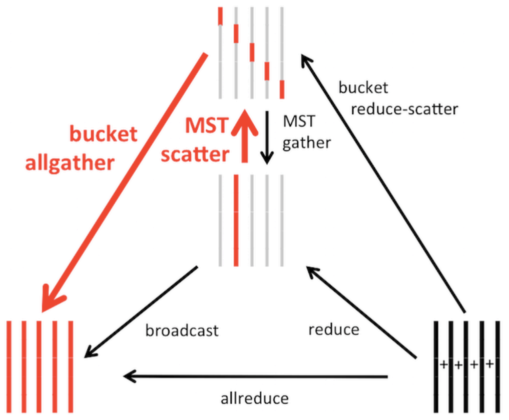
☛ SEE ANSWER

4.5.3 The long vector case

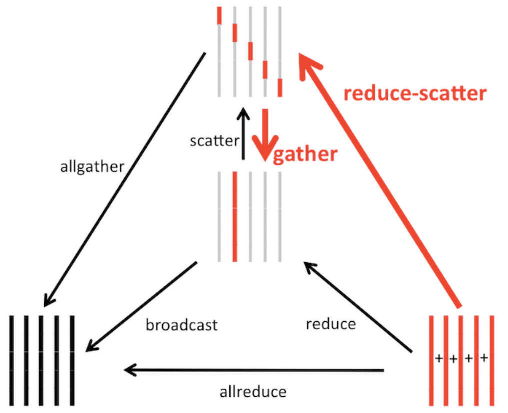
When the vector of data being communication is long, it is the β term of the cost that dominates. Assuming that all processes receive or contribute the same subvector of size n/p, the MST scatter



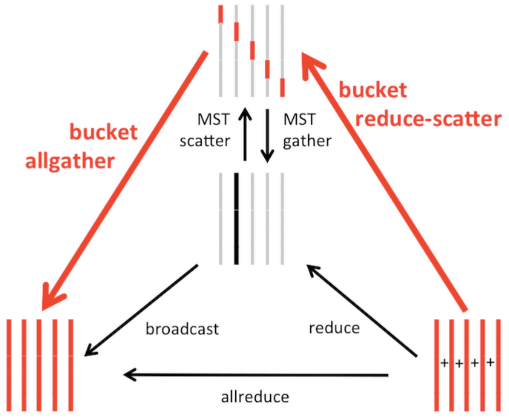
(a) Building blocks for the long vector case.



(b) Scatter/allgather algorithm for broadcast.



(c) Reduce-scatter/gather algorithm for reduce(-to-one).



(d) Reduce-scatter/allgather algorithm for allreduce.

Figure 4.12: Algorithms for communicating with long vectors.

and MST gather algorithms both attain the lower bound for the beta term: $\frac{p-1}{p}n\beta$. Similarly, the bucket allgather algorithm does the same: $\frac{p-1}{p}n\beta$. The bucket reduce-scatter does too: $\frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma$. We highlight these four algorithms in Figure 4.12 (a). We now discuss how these four become the building blocks for the other three collectives when the vectors are long, as illustrated in Figure 4.12 (b)–(d).

Homework 4.5.3.1 Which of the following is the result of composing the indicated collective communications (within the same group of processes). (Assume that the subvectors involved are of size n/p.)

1. Scatter followed by allgather.
2. Reduce-scatter followed by allgather.
3. Reduce-scatter followed by gather.

☛ SEE ANSWER

4.5.4 Viewing the processes as a higher dimensional mesh

The problem with the bucket algorithms is that they have a large α (latency) term. With as large as distributed memory architectures have become, viewing them as a one dimensional array of processes is not feasible, especially considering that α is three to four magnitudes greater than β. We now show how viewing the processes as a logical higher dimensional mesh allows us to reduce the α term without changing the β term.

Let's focus on allgather. View the p processes as an $r \times c$ mesh, where $p = rc$. Then the allgather can be achieved by first performancing an allgather within rows, followed by an allgather of the result within columns. The cost, if bucket algorithms are used in both dimensions and each process starts with n/p items, becomes

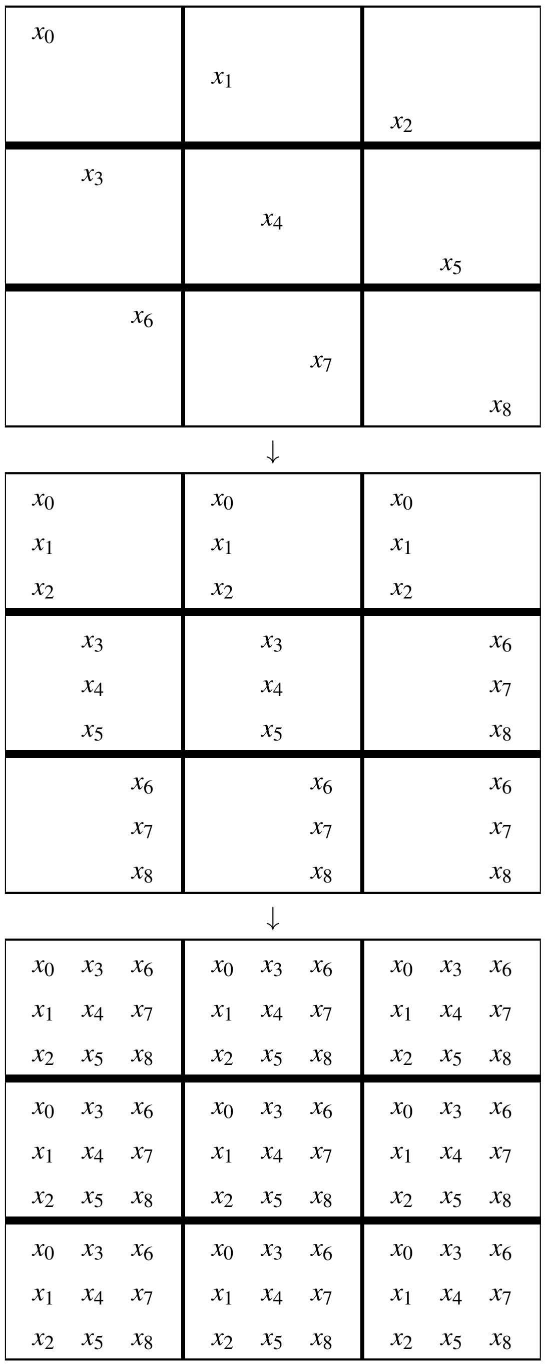
$$(c-1)(\alpha + \frac{n}{p}\beta) + (r-1)(\alpha + c\frac{n}{p}\beta) = (c+r-2)\alpha + \frac{p-1}{p}n\beta.$$

We notice that, relative to the cost of a bucket algorithm that views the processes as a one dimensional array, the cost of the α term is reduced from $(p-1)\alpha$ to $(r+c-2)\alpha$ while the β term remains unchanged (and thus optimal). If $r = c = \sqrt{p}$ then the α term becomes $2(\sqrt{p}-1)$.

Homework 4.5.4.1 Show that

$$(c-1)(\alpha + \frac{n}{p}\beta) + (r-1)(\alpha + c\frac{n}{p}\beta) = (c+r-2)\alpha + \frac{p-1}{p}n\beta.$$

☛ SEE ANSWER



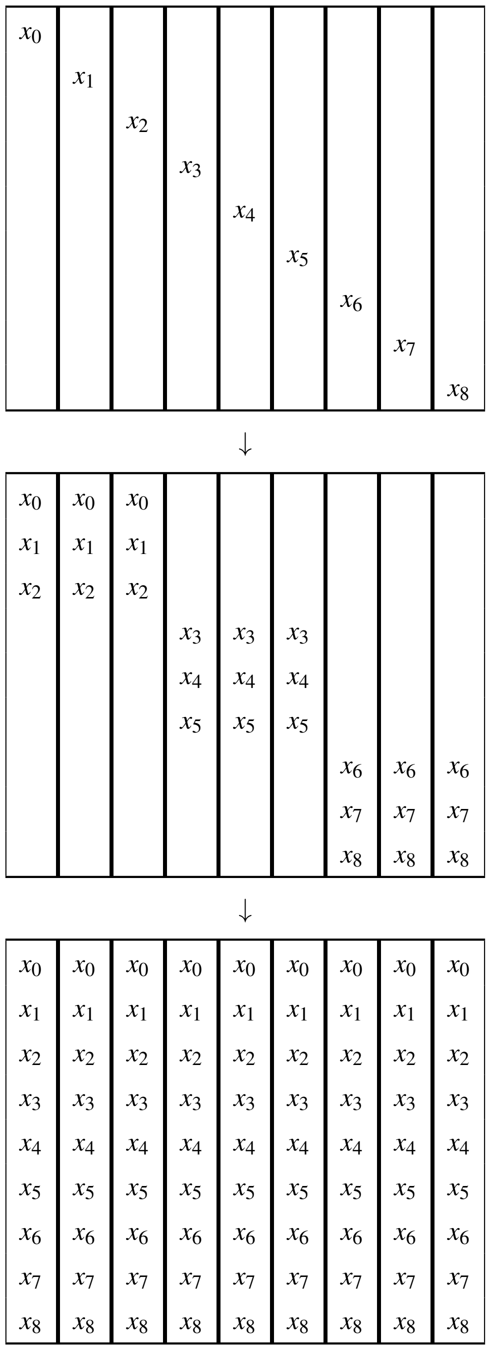


Figure 4.13: Allgather algorithm that views the processes as a 2D mesh, with an allgather within rows followed by an allgather within columns. On the left each box represents a process and the subvectors form a 2D array as well. On the right, we view the processes as a linear array.

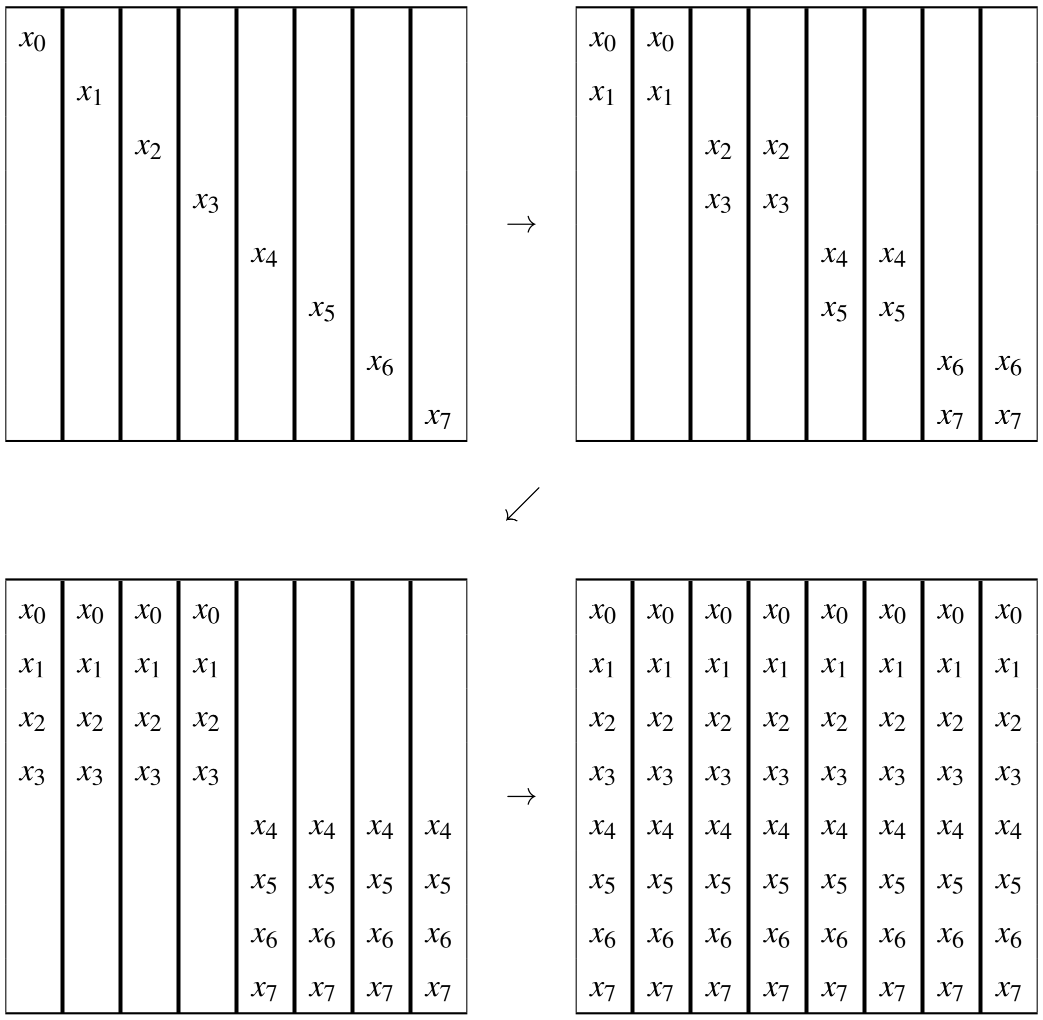


Figure 4.14: Bidirectional exchange algorithm for allgather, illustrated for $p = 8$.

Obviously, the insights can be extended. If the processes are viewed as a three dimensional mesh, $p = p_0 \times p_1 \times p_2$, then one can allgather within each of the dimensions. Under our assumptions, the cost becomes

$$(p_0 + p_1 + p_2 - 3)\alpha + \frac{p-1}{p} n\beta.$$

If $p_0 = p_1 = p_2 = \sqrt[3]{p}$ then this becomes

$$3(\sqrt[3]{p} - 1)\alpha + \frac{p-1}{p} n\beta.$$

In the limit, if $p = 2^d$ (p is a power of two and $d = \log_2(p)$)), the processes can be viewed as a d dimensional mesh, with two processes in each direction. This is illustrated for $p = 8$ in Figure 4.14.

The cost now becomes

$$d(\sqrt[d]{p}-1)\alpha+\frac{p-1}{p}n\beta=\log_2(p)(2-1)\alpha+\frac{p-1}{p}n\beta=\log_2(p)\alpha+\frac{p-1}{p}n\beta.$$

What we notice is that this algorithm is optimal in both the α and β terms. This algorithm is often referred to as an *bidirectional exchange* algorithm since one can view it as a successive pairwise exchange of the data that has accumulated in each dimension.

Homework 4.5.4.2 Show that if $d = \log_2(p)$ then $\sqrt[d]{p} = 2$.

☛ SEE ANSWER

It is important to realize that on modern distributed memory architectures invariably, when the processes are viewed as a higher dimensional mesh, communication will start causing network conflicts that don't impact the α term, but increase the β term. Thus, a bidirectional exchange algorithm will likely cost more in practice. Details go beyond this discussion.

Homework 4.5.4.3 Reduce-scatter is the dual of the allgather operation. Use this insight to propose a multidimensional mesh architecture that is optimal in the α, β, and γ terms when $p = 2^d$ (p is a power of two).

☛ SEE ANSWER

4.5.5 From short vector to long vector

The next question is how to transition form a short vector algorithm to a long vector algorithm. The naive approach would be to simply find the cross over point when one becomes faster than the other. Due to the fact that we can view processes as a multiple dimensional mesh, and the mesh is not unique, there are actually many possible algorithms to be considered. In addition, the physical topology of the underlying network that connects nodes influences the details of each algorithm and/or network conflicts that may occur and may affect the cost of communication. This includes the fact that processes may be on the same node, in which case communication may be much more efficient than between nodes.

For the material in future weeks of this course, it is important to understand how collective communications can be composed to create other collective communications. It is not important to understand all the intricate details of how to achieve near-optimal implementations for the range of vector lengths that may be encountered. For this reason, we refer the reader to a paper that goes into much greater detail, and we believe is quite readable given the background we have given:

> Ernie Chan, Marcel Heimlich, Avi Purkayastha, Robert van de Geijn.
> Collective communication: theory, practice, and experience.
> *Concurrency and Computation: Practice & Experience* , Volume 19 Issue 1, September 2007.

Your institution may have a license to view this paper. If not, you may want to instead read the technical report that preceded the paper:

Ernie Chan, Marcel Heimlich, Avijit Purkayastha, and Robert van de Geijn.
Collective Communication: Theory, Practice, and Experience.
FLAME Working Note #22. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-06-44. September 26, 2006.

available from http://www.cs.utexas.edu/users/flame/web/FLAMEPublications.html.

4.6 Enrichment

4.7 Wrapup

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

Figure 4.15: A set of commonly encountered collective communications.

Week 5

Distributed Memory Parallel Matrix-Vector Operations

5.1 Opener

5.1.1 Launch

5.1.2 Outline Week 4

5.1.3 What you will learn

5.2 Parallel Matrix-Vector Multiplication

5.2.1 Notation

Suppose $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$, where

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

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

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

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

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

(5.1)

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

To elegantly describe the parallelization of operations like matrix-vector multiplication, it is important to understand how to view matrices and vectors by submatrices and subvectors, and how to then multiply so-partitioned matrices and vectors. If we partition

$$A = \begin{pmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,N-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M-1,0} & A_{M-1,1} & \cdots & A_{M-1,N-1} \end{pmatrix}, x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{M-1} \end{pmatrix}, \text{ and } y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{M-1} \end{pmatrix},$$

where $A_{i,j}$ is $m_i \times n_j$, x_j has size n_j, and y_i has size m_i, then

$$\begin{array}{rcccc} y_0 = & A_{0,0}x_0 + & A_{0,1}x_1 + \cdots + & A_{0,N-1}x_{N-1} \\ y_1 = & A_{1,0}x_0 + & A_{1,1}x_1 + \cdots + & A_{1,N-1}x_{N-1} \\ \vdots & \vdots & \vdots & \vdots \\ y_{M-1} = & A_{M-1,0}x_0 + & A_{M-1,1}x_1 + \cdots + & A_{M-1,N-1}x_{N-1}. \end{array} \tag{5.2}$$

5.2.2 A first parallel algorithm

The last unit should make it pretty easy to recognize opportunities for parallelizing matrix-vector multiplication. For simplicity, we are going to assume that the matrix is square ($n \times n$) and $m_i = n_i$. If there are p processes, then in (5.3) one could take $M = p$ and $N = 1$ to find that

$$\begin{array}{rcl} y_0 & = & A_0 x \\ y_1 & = & A_1 x \\ \vdots & & \vdots \\ y_{p-1} & = & A_{p-1} x \end{array}$$

where A_i is $n_i \times n$ and y_i has size n_i. We can then dictate that A_i, x, and y_i all reside on process i, and pick $n_i \approx n/p$. Bingo, we have a very nicely parallelized matrix-vector multiplication.

Let us make the example somewhat more realistic by dictating that however x is initially distributed among processes, that is how y must be distributed upon completion. Why this is a reasonable assumption, or at least a reasonable starting point, will become clear later. What we notice is that after the local matrix-vector multiply $A_i x$ has been computed by process i, the subvectors of y have to be allgathered.

Let us assume that performing a floating point multiply and add each require time γ and the model of communication cost introduced in Week 4 (e.g., a message costs $\alpha + n\beta$). Then, since we chose $m = n$ and n is a multiple of p, performing the local matrix-vector multiplication $y_i := Ax$ costs, approximately, $2n_i n\gamma = 2n^2/p\gamma$ and a reasonable lower bound on the allgather is $\log_2(p)\alpha + (p-1)/p\ n\beta$. Thus, if p is large and hence $(p-1)/p \approx 1$, a reasonable estimate of the cost is given by

$$T_p(n) = 2\frac{n^2}{p}\gamma + \log_2(p)\alpha + n\beta.$$

5.2.3 Speedup and efficiency

When one uses p processes (on different cores), one would hope to compute p times as fast. Usually, that is the best you can hope for[1].

[1] Sometimes, you can achieve better than a factor p, because the p processes collectively have more resources. For example, you can usually make a bed faster than twice as fast when you do it with two people. Why? Because you don't have to walk back and forth from one side to the other. But this is the exception in practice.

Speedup is defined by the time required by (the best) implementation utilizing a single process, T_1 divided by the time when using p processes (on different cores/processors), T_p. Let's revisit the example from the last unit. The parallel time was given by

$$T_p(n) = 2\frac{n^2}{p}\gamma + \log_2(p)\alpha + n\beta.$$

The time it would take by a single process is $T_1(n) = 2n^2\gamma$. Thus, the estimated speedup attained by this algorithm is

$$S_p(n) = \frac{T_1(n)}{T_p(n)} = \frac{2n^2\gamma}{2\frac{n^2}{p}\gamma + \log_2(p)\alpha + n\beta}.$$

We can multiply top and bottom by p and divide top and bottom by $2n^2\gamma$ to find that

$$S_p(n) = \frac{p}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}.$$

The efficiency of the algorithm is given by the speedup divided by p. It gives the fraction of the maximal speedup of p that is attained. For our example, this means that

$$E_p(n) = \frac{S_p(n)}{p} = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}$$

The question we can now ask ourselves is "Is this a good algorithm?" Clearly, this depends on how close the efficiency is to 1 (perfect speedup).

Looking at the efficiency for our example

$$E_p(n) = \frac{S_p(n)}{p} = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}},$$

we can ask that of the algorithm presented in the last unit. The answer is that it depends on whether you are an optimist, a pessimist, or a realist.

The optimist. The optimist will make the following argument: If you fix the number of processes, and you increase the problem size to become (very) large, then for our example

$$\lim_{n\to\infty} E_p(n) = \frac{S_p(n)}{p} = \lim_{n\to\infty} \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} = 1.$$

Thus, in the limit the algorithm will attain 100% efficiency, and hence it is a great parallel algorithm.

The pessimist. The pessimist will make the following counter argument: If you fix the problem size n, and you increase the number of processes that are used, then for our example

$$\lim_{p\to\infty} E_p(n) = \frac{S_p(n)}{p} = \lim_{p\to\infty} \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} = 0.$$

Thus, in the limit the algorithm will attain 0% efficiency, and hence it is a lousy parallel algorithm.

The realist. The realist will argue the following. To the optimist, the realist will explain that one can't store arbitrarily large matrices. To the pessimist, the realist will argue that as the number of processes is increased, so is the number of processes on which they execute, and hence the amount of memory that the processes collectively have access to. So, it is fair to increase the problem size (big machines are made to solve big problems) but it cannot become arbitrarily large.

What does this mean? If you think about $y := Ax$, you realize that it is matrix A that requires n^2 floating point numbers (floats) to be stored. If each process comes with a memory of constant size K floats, then the total memory scales linearly with p. The largest matrix that can fit is (approximately) of size $n_{\max}(p)$ with $n_{\max}(p)^2 = Kp$ or, equivalently, $n_{\max}(p)^2/p = K$. The question is what happens to efficiency when computing with the largest matrix that fits, as the number of processes, p, is increased:

$$\lim_{p\to\infty} E_p(n_{\max}(p)) = \lim_{p\to\infty} \frac{1}{1+\frac{p\log_2(p)}{2n_{\max}(p)^2}\frac{\alpha}{\gamma}+\frac{p}{2n_{\max}(p)}\frac{\beta}{\gamma}} = \lim_{p\to\infty} \frac{1}{1+\frac{\log_2(p)}{2K}\frac{\alpha}{\gamma}+\frac{\sqrt{p}}{2\sqrt{K}}\frac{\beta}{\gamma}}.$$

Now, $\log_2(p)$ grows very slowly, so we wouldn't be particularly concerned about the term

$$\frac{\log_2(p)}{2K}\frac{\alpha}{\gamma}.$$

The term

$$\frac{\sqrt{p}}{2\sqrt{K}}\frac{\beta}{\gamma}$$

is a concern because β is often two to three orders of magnitude greater than γ and $\sqrt{p}$ grows quite quickly with p.

5.2.4 Scalability

An algorithm is considered to be *strongly scalable* if efficiency can be maintained while keeping the problem size constant. Algorithms are never strongly scalable: eventually there isn't enough work to keep all processes usefully busy. But sometimes for a range of probably sizes and range of number of processes, an algorithm may be reasonably scalable.

Algorithms are never strongly scalable: eventually there isn't enough work to keep all processes usefully busy. But sometimes for a range of probably sizes and range of number of processes, an algorithm may be reasonably scalable.

An algorithm is considered to be *weakly scalable* if efficiency can be maintained by growing the problem size so that the memory use per process is kept constant. If the examined algorithm were weakly scalable, then

$$\lim_{p\to\infty} E_p(n_{\max}(p)) = E > 0.$$

where E is some (positive) constant. In this case, a positive efficiency can be maintained as the number of processors increases.

The proposed parallel algorithm for matrix-vector multiplication is neither strongly nor weakly scalable since

$$\lim_{p\to\infty} E_p(n_{\max}(p)) = \lim_{p\to\infty} \frac{1}{1+\frac{p\log_2(p)}{2n_{\max}(p)^2}\frac{\alpha}{\gamma}+\frac{p}{2n_{\max}(p)}\frac{\beta}{\gamma}} = \lim_{p\to\infty} \frac{1}{1+\frac{\log_2(p)}{2K}\frac{\alpha}{\gamma}+\frac{\sqrt{p}}{2\sqrt{K}}\frac{\beta}{\gamma}} = 0$$

even if we consider $\log_2(p)$ to grow slowly enough that we can pretend it is a constant.

5.2.5 A second parallel algorithm

Let's revisit

$$\begin{array}{rcccc} y_0 = & A_{0,0}x_0 + & A_{0,1}x_1 + \cdots + & A_{0,N-1}x_{N-1} \\ y_1 = & A_{1,0}x_0 + & A_{1,1}x_1 + \cdots + & A_{1,N-1}x_{N-1} \\ \vdots & \vdots & \vdots & \vdots \\ y_{M-1} = & A_{M-1,0}x_0 + & A_{M-1,1}x_1 + \cdots + & A_{M-1,N-1}x_{N-1}. \end{array} \tag{5.3}$$

If we again restrict ourselves to a square $n \times n$ matrix A and we choose $M = 1$ and $N = p$ then

$$y = \underbrace{A_0 x_0}_{y^{(0)}} + \underbrace{A_1 x_1}_{y^{(1)}} + \cdots + \underbrace{A_{p-1} x_{p-1}}_{y^{(p-1)}},$$

where A_j is now $n \times n_j$ and x_j has size n_j. We can then dictate that A_j and x_j all reside on process j. This allows to, in parallel, compute $y^{(j)}$ of process j. The final result, y, is then computed by adding these contributions, requiring a reduction (summation) across the processes. The question is where the result is to reside. As we discussed before, we will dictate that the result vector, y, is left distributed like x. Thus, it is a reduce-scatter that is required.

Homework 5.2.5.1 Analyze the cost, speedup, and efficiency of this second algorithm. Is it strongly scalable? Is it weakly scalable? How are the communications encountered in the first and second algorithms related?

☛ SEE ANSWER

5.2.6 A weakly scalable algorithm

The two parallel algorithms that we have discussed so far view the p processes as a one dimensional array, indexed 0 through $p - 1$. We are now going to discuss a data distribution and algorithm that views the p processes as a $r \times c$ mesh, where $p = rc$. To motivated this two dimensional distribution and algorithm, consider the computation $y = Ax$ where A has been partitioned into

9×9 submatrices and the vectors into 9 subvectors, of appropriate sizes:

$$
\begin{array}{lcl}
y_0 & = & A_{0,0}x_0 + A_{0,1}x_1 + A_{0,2}x_2 + A_{0,3}x_3 + A_{0,4}x_4 + A_{0,5}x_5 + A_{0,6}x_6 + A_{0,7}x_7 + A_{0,8}x_8 \\
y_1 & = & A_{1,0}x_0 + A_{1,1}x_1 + A_{1,2}x_2 + A_{1,3}x_3 + A_{1,4}x_4 + A_{1,5}x_5 + A_{1,6}x_6 + A_{1,7}x_7 + A_{1,8}x_8 \\
y_2 & = & A_{2,0}x_0 + A_{2,1}x_1 + A_{2,2}x_2 + A_{2,3}x_3 + A_{2,4}x_4 + A_{2,5}x_5 + A_{2,6}x_6 + A_{2,7}x_7 + A_{2,8}x_8 \\
y_3 & = & A_{3,0}x_0 + A_{3,1}x_1 + A_{3,2}x_2 + A_{3,3}x_3 + A_{3,4}x_4 + A_{3,5}x_5 + A_{3,6}x_6 + A_{3,7}x_7 + A_{3,8}x_8 \\
y_4 & = & A_{4,0}x_0 + A_{4,1}x_1 + A_{4,2}x_2 + A_{4,3}x_3 + A_{4,4}x_4 + A_{4,5}x_5 + A_{4,6}x_6 + A_{4,7}x_7 + A_{4,8}x_8 \\
y_5 & = & A_{5,0}x_0 + A_{5,1}x_1 + A_{5,2}x_2 + A_{5,3}x_3 + A_{5,4}x_4 + A_{5,5}x_5 + A_{5,6}x_6 + A_{5,7}x_7 + A_{5,8}x_8 \\
y_6 & = & A_{6,0}x_0 + A_{6,1}x_1 + A_{6,2}x_2 + A_{6,3}x_3 + A_{6,4}x_4 + A_{6,5}x_5 + A_{6,6}x_6 + A_{6,7}x_7 + A_{6,8}x_8 \\
y_7 & = & A_{7,0}x_0 + A_{7,1}x_1 + A_{7,2}x_2 + A_{7,3}x_3 + A_{7,4}x_4 + A_{7,5}x_5 + A_{7,6}x_6 + A_{7,7}x_7 + A_{7,8}x_8 \\
y_8 & = & A_{8,0}x_0 + A_{8,1}x_1 + A_{8,2}x_2 + A_{8,3}x_3 + A_{8,4}x_4 + A_{8,5}x_5 + A_{8,6}x_6 + A_{8,7}x_7 + A_{8,8}x_8
\end{array}
$$

We can organize these computations as illustrated by

$$
\begin{array}{c}
\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ A_{1,0} & A_{1,1} & A_{1,2} \\ A_{2,0} & A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} A_{0,3} & A_{0,4} & A_{0,5} \\ A_{1,3} & A_{1,4} & A_{1,5} \\ A_{2,3} & A_{2,4} & A_{2,5} \end{pmatrix} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix} + \begin{pmatrix} A_{0,6} & A_{0,7} & A_{0,8} \\ A_{1,6} & A_{1,7} & A_{1,8} \\ A_{2,6} & A_{2,7} & A_{2,8} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \\ x_8 \end{pmatrix} \\
\begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} A_{3,0} & A_{3,1} & A_{3,2} \\ A_{4,0} & A_{4,1} & A_{4,2} \\ A_{5,0} & A_{5,1} & A_{5,2} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} A_{3,3} & A_{3,4} & A_{3,5} \\ A_{4,3} & A_{4,4} & A_{4,5} \\ A_{5,3} & A_{5,4} & A_{5,5} \end{pmatrix} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix} + \begin{pmatrix} A_{3,6} & A_{3,7} & A_{3,8} \\ A_{4,6} & A_{4,7} & A_{4,8} \\ A_{5,6} & A_{5,7} & A_{5,8} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \\ x_8 \end{pmatrix} \\
\begin{pmatrix} y_6 \\ y_7 \\ y_8 \end{pmatrix} = \begin{pmatrix} A_{6,0} & A_{6,1} & A_{6,2} \\ A_{7,0} & A_{7,1} & A_{7,2} \\ A_{8,0} & A_{8,1} & A_{8,2} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} A_{6,3} & A_{6,4} & A_{6,5} \\ A_{7,3} & A_{7,4} & A_{7,5} \\ A_{8,3} & A_{8,4} & A_{8,5} \end{pmatrix} \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix} + \begin{pmatrix} A_{6,6} & A_{6,7} & A_{6,8} \\ A_{7,6} & A_{7,7} & A_{7,8} \\ A_{8,6} & A_{8,7} & A_{8,8} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \\ x_8 \end{pmatrix}
\end{array}
$$

The idea now is that, for this example, the processes can be viewed as a 3×3 mesh. The process indexed (I, J) now holds submatrix

$$
\begin{pmatrix} A_{3I,3J} & A_{3I,3J+1} & A_{3I,3J+2} \\ A_{3I+1,3J} & A_{3I+1,3J+2} & A_{3I+1,3J+2} \\ A_{3I+2,3J} & A_{3I+2,3J+3} & A_{3I+2,3J+2} \end{pmatrix}
$$

and performs the computation

$$
\begin{pmatrix} y^{(J)}_{3I} \\ y^{(J)}_{3I+1} \\ y^{(J)}_{3I+2} \end{pmatrix} = \begin{pmatrix} A_{3I,3J} & A_{3I,3J+1} & A_{3I,3J+2} \\ A_{3I+1,3J} & A_{3I+1,3J+2} & A_{3I+1,3J+2} \\ A_{3I+2,3J} & A_{3I+2,3J+3} & A_{3I+2,3J+2} \end{pmatrix} \begin{pmatrix} x_{3J} \\ x_{3J+1} \\ x_{3J+2} \end{pmatrix}.
$$

After this, within each row i of processes, the contributions

$$\begin{pmatrix} y_{3I}^{(J)} \\ y_{3I+1}^{(J)} \\ y_{3I+2}^{(J)} \end{pmatrix}$$

must be added (reduced) to the final result

$$\begin{pmatrix} y_{3I} \\ y_{3I+1} \\ y_{3I+2} \end{pmatrix}.$$

This leaves us to decide to which processes to initially assign subvectors x_j and to which processes to assign the result subvectors y_j.

Let us now illustrate the distributions of the vectors and matrix for a 3×4 mesh of processes, as illustrated in Figures 5.1-5.4. In these, we use a 3×4 mesh to demonstrate that the method is not dependent upon the mesh being square. Notice that the subvectors of x are distributed among the processes in column-major order while the subvectors of y are to be left distributed in row-major order. The reason why will become clear. The algorithm can now be described as follows:

- Initially the data is distributed as illustrated in Figure 5.1. The best way to understand the distribution is as follows:
  - Vector x is partitioned into p subvectors. These subvectors are distributed to the $r \times c$ mesh of processes in column-major order, one subvector per process.
  - Vector y is partitioned into p subvectors. These subvectors are distributed to the $r \times c$ mesh of processes in row-major order, one subvector per process.
  - Matrix A is partitioned into $p \times p$ submatrices. Notice that $A_{i,j}$ multiplies x_j and contributes to y_i. By assigning $A_{i,j}$ to the same column of processes as is x_j, subvectors of x need only be communicated within that column of processes. By assigning $A_{i,j}$ to the same row of processes as is y_i, subvectors of y need only be reduced (summed) within that row of processes.
- Allgather subvectors x_j within columns of processes at a cost of (approximately)

  $$\log_2(r)\alpha + \frac{r-1}{r}\frac{n}{c}\beta.$$

  (Figure 5.2.)
- Perform the local matrix-vector multiplication at a cost of (approximately)

  $$2\frac{n^2}{p}\gamma.$$

  (Figure 5.3.)

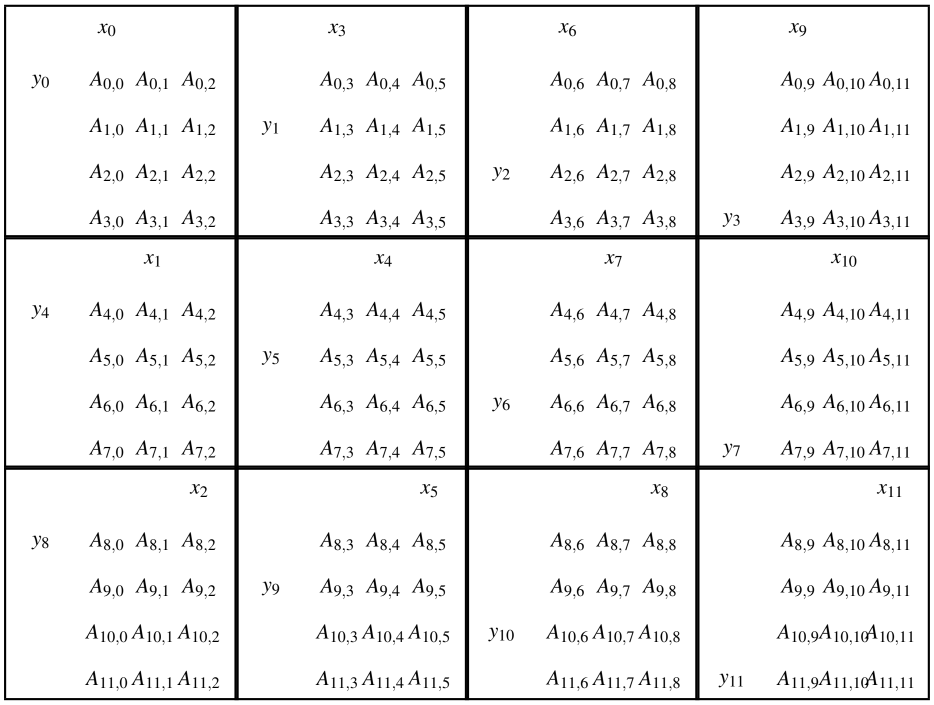

x_0 x_3 x_6 x_9
y_0 $A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{0,4}$ $A_{0,5}$ $A_{0,6}$ $A_{0,7}$ $A_{0,8}$ $A_{0,9}$ $A_{0,10}$ $A_{0,11}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ y_1 $A_{1,3}$ $A_{1,4}$ $A_{1,5}$ $A_{1,6}$ $A_{1,7}$ $A_{1,8}$ $A_{1,9}$ $A_{1,10}$ $A_{1,11}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{2,4}$ $A_{2,5}$ y_2 $A_{2,6}$ $A_{2,7}$ $A_{2,8}$ $A_{2,9}$ $A_{2,10}$ $A_{2,11}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$ $A_{3,4}$ $A_{3,5}$ $A_{3,6}$ $A_{3,7}$ $A_{3,8}$ y_3 $A_{3,9}$ $A_{3,10}$ $A_{3,11}$
x_1 x_4 x_7 x_{10}
y_4 $A_{4,0}$ $A_{4,1}$ $A_{4,2}$ $A_{4,3}$ $A_{4,4}$ $A_{4,5}$ $A_{4,6}$ $A_{4,7}$ $A_{4,8}$ $A_{4,9}$ $A_{4,10}$ $A_{4,11}$
$A_{5,0}$ $A_{5,1}$ $A_{5,2}$ y_5 $A_{5,3}$ $A_{5,4}$ $A_{5,5}$ $A_{5,6}$ $A_{5,7}$ $A_{5,8}$ $A_{5,9}$ $A_{5,10}$ $A_{5,11}$
$A_{6,0}$ $A_{6,1}$ $A_{6,2}$ $A_{6,3}$ $A_{6,4}$ $A_{6,5}$ y_6 $A_{6,6}$ $A_{6,7}$ $A_{6,8}$ $A_{6,9}$ $A_{6,10}$ $A_{6,11}$
$A_{7,0}$ $A_{7,1}$ $A_{7,2}$ $A_{7,3}$ $A_{7,4}$ $A_{7,5}$ $A_{7,6}$ $A_{7,7}$ $A_{7,8}$ y_7 $A_{7,9}$ $A_{7,10}$ $A_{7,11}$
x_2 x_5 x_8 x_{11}
y_8 $A_{8,0}$ $A_{8,1}$ $A_{8,2}$ $A_{8,3}$ $A_{8,4}$ $A_{8,5}$ $A_{8,6}$ $A_{8,7}$ $A_{8,8}$ $A_{8,9}$ $A_{8,10}$ $A_{8,11}$
$A_{9,0}$ $A_{9,1}$ $A_{9,2}$ y_9 $A_{9,3}$ $A_{9,4}$ $A_{9,5}$ $A_{9,6}$ $A_{9,7}$ $A_{9,8}$ $A_{9,9}$ $A_{9,10}$ $A_{9,11}$
$A_{10,0}$ $A_{10,1}$ $A_{10,2}$ $A_{10,3}$ $A_{10,4}$ $A_{10,5}$ y_{10} $A_{10,6}$ $A_{10,7}$ $A_{10,8}$ $A_{10,9}$ $A_{10,10}$ $A_{10,11}$
$A_{11,0}$ $A_{11,1}$ $A_{11,2}$ $A_{11,3}$ $A_{11,4}$ $A_{11,5}$ $A_{11,6}$ $A_{11,7}$ $A_{11,8}$ y_{11} $A_{11,9}$ $A_{11,10}$ $A_{11,11}$

Figure 5.1: Parallel matrix-vector multiplication: Initial distribution of A and x and final distribution of y.

- Reduce-scatter the result within rows of processes at a cost of (approximately)

  $$\log_2(c)\alpha + \frac{c-1}{c}\frac{n}{r}\beta + \frac{c-1}{c}\frac{n}{r}\gamma.$$

  (Figure 5.4.)

The total estimated cost of the algorithm is

$$\begin{aligned} T_{r\times c}(n) &= \log_2(r)\alpha + \frac{r-1}{r}\frac{n}{c}\beta + 2\frac{n^2}{p}\gamma + \log_2(c)\alpha + \frac{c-1}{c}\frac{n}{r}\beta + \frac{c-1}{c}\frac{n}{r}\gamma \\ &= 2\frac{n^2}{p}\gamma + \log_2(p)\alpha + (r-1)\frac{n}{p}\beta + (c-1)\frac{n}{p}\beta + (c-1)\frac{n}{p}\gamma \\ &= 2\frac{n^2}{p}\gamma + \log_2(p)\alpha + (r+c-2)\frac{n}{p}\beta + (c-1)\frac{n}{p}\gamma. \end{aligned}$$

If we assume $r = c = \sqrt{p}$ and we approximate $\sqrt{p}-1$ with $\sqrt{p}$, we get

$$\begin{aligned} T_{\sqrt{p}\times\sqrt{p}}(n) &= 2\frac{n^2}{p}\gamma + \log_2(p)\alpha + 2(\sqrt{p}-1)\frac{n}{p}\beta + (\sqrt{p}-1)\frac{n}{p}\gamma \\ &\approx 2\frac{n^2}{p}\gamma + \log_2(p)\alpha + 2\frac{n}{\sqrt{p}}\beta + \frac{n}{\sqrt{p}}\gamma. \end{aligned}$$

Now, let's analyze the scalability of this algorithm:

$$\begin{aligned} E_{\sqrt{p}\times\sqrt{p}}(n_{\max}(p)) &= \frac{T_1(n)}{pT_{\sqrt{p}\times\sqrt{p}}(n_{\max}(p))} \\ &= \frac{2n_{\max}(p)^2\gamma}{2n_{\max}(p)^2\gamma + p\log_2(p)\alpha + 2\sqrt{p}n_{\max}(p)\beta + \sqrt{p}n_{\max}(p)\gamma} \\ &\approx \frac{1}{1+\frac{1}{2}\frac{p}{n_{\max}(p)^2}\log_2(p)\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{n_{\max}(p)}\frac{\beta}{\gamma} + \frac{1}{2}\frac{\sqrt{p}}{n_{\max}(p)}}. \end{aligned}$$

Recalling that $n_{\max}(p)^2 = Kp$ we find that

$$E_{\sqrt{p}\times\sqrt{p}}(n_{\max}(p)) \approx \frac{1}{1+\frac{1}{2}\frac{1}{K}\log_2(p)\frac{\alpha}{\gamma} + \frac{1}{\sqrt{K}}\frac{\beta}{\gamma} + \frac{1}{2}\frac{1}{\sqrt{K}}}.$$

If we consider $\log_2(p)$ to be constant for practical purposes, we find that $E_{\sqrt{p}\times\sqrt{p}}(n_{\max}(p))$ is constant as p increases, and hence the algorithm is weakly scalable (for practical purposes).

Keep in mind that we are using lower bounds for the cost of allgather and reduce-scatter in our estimated cost. A lot could depend on how efficiently the collective communications are implemented in practice.

Homework 5.2.6.2 Consider how to compute $x = A^T y$ with a partitioned matrix A and partitioned vectors x and y:

$$\begin{array}{rcccc} x_0 = & A_{0,0}^T y_0 + & A_{1,0}^T y_1 + \cdots + & A_{M-1,0}^T y_{M-1} \\ x_1 = & A_{0,1}^T y_0 + & A_{1,1}^T y_1 + \cdots + & A_{M-1,1}^T y_{M-1} \\ \vdots & \vdots & \vdots & \vdots \\ x_{N-1} = & A_{0,N-1}^T y_0 + A_{1,N-1}^T y_1 + \cdots + A_{M-1,N-1}^T y_{M-1}. \end{array} \tag{5.4}$$

Assuming the matrix and vectors are distributed as illustrated in Figure 5.1, propose a parallel algorithm for computing $y = A^T x$.

☛ SEE ANSWER

Homework 5.2.6.3 Assuming the matrix and vectors are distributed as illustrated in Figure 5.1, propose a parallel algorithm for computing $y = Ax$.

☛ SEE ANSWER

5.3 Parallel Rank-1 Update

5.3.1 Notation

Suppose $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$, where

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

Recalling that $A +:= yx^T$ (rank-1 update) is computed as

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

we notice that element $\alpha_{i,j}$ is updated by multiplying ψ_j and χ_i. Thus we can, again, summarize the interactions of the elements of x, y, and A by

$$\begin{array}{c|c|c|c|c} & \chi_0 & \chi_1 & \cdots & \chi_{n-1} \\ \hline \psi_0 & \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \hline \psi_1 & \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \hline \vdots & \vdots & \vdots & \ddots & \vdots \\ \hline \psi_{m-1} & \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{array} . \tag{5.5}$$

If we partition

$$A = \begin{pmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,N-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M-1,0} & A_{M-1,1} & \cdots & A_{M-1,N-1} \end{pmatrix}, x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{M-1} \end{pmatrix}, \text{ and } y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{M-1} \end{pmatrix},$$

where $A_{i,j}$ is $m_i \times n_j$, x_j has size n_j, and y_i has size m_i, then

$$\begin{array}{ccc|ccc|c|ccc} A_{0,0} & +:= & y_0 x_0^T & A_{0,1} & +:= & y_0 x_1^T & \cdots & A_{0,N-1} & +:= & y_0 x_{N-1}^T \\ \hline A_{1,0} & +:= & y_1 x_0^T & A_{1,1} & +:= & y_1 x_1^T & \cdots & A_{1,N-1} & +:= & y_1 x_{N-1}^T \\ \hline & \vdots & & & \vdots & & & & \vdots & \\ \hline A_{M-1,0} & +:= & y_{M-1} x_0^T & A_{M-1,1} & +:= & y_{M-1} x_1^T & \cdots & A_{0,N-1} & +:= & y_{M-1} x_{N-1}^T \end{array}$$

5.3.2 A weakly scalable algorithm

Examining Figure 5.5, the key insight now is that we can organize the update

$$\left(\begin{array}{c|c|c} A_{4I,3J} & A_{4I,3J+1} & A_{4I,3J+2} \\ \hline A_{I+1,3J} & A_{4I+1,3J+2} & A_{4I+1,3J+2} \\ \hline A_{4I+2,3J} & A_{4I+2,3J+3} & A_{4I+2,3J+2} \\ \hline A_{4I+3,3J} & A_{4I+3,3J+3} & A_{4I+3,3J+2} \end{array}\right)$$

with

$$\left(\begin{array}{c|c|c} A_{4I,3J} + y_{4I} x_{3J}^T & A_{4I,3J+1} + y_{4I} x_{3J+1}^T & A_{4I,3J+2} + y_{4I} x_{3J+1}^T \\ \hline A_{I+1,3J} + y_{4I+1} x_{3J}^T & A_{4I+1,3J+2} + y_{4I+1} x_{3J+1}^T & A_{4I+1,3J+2} + y_{4I+1} x_{3J+2}^T \\ \hline A_{4I+2,3J} + y_{4I+2} x_{3J}^T & A_{4I+2,3J+3} + y_{4I+2} x_{3J+1}^T & A_{4I+2,3J+2} + y_{4I+2} x_{3J+2}^T \\ \hline A_{4I+3,3J} + y_{4I+3} x_{3J}^T & A_{4I+3,3J+3} + y_{4I+3} x_{3J+1}^T & A_{4I+3,3J+2} + y_{4I+3} x_{3J+2}^T \end{array}\right)$$

as

$$\left(\begin{array}{c|c|c} A_{4I,3J} & A_{4I,3J+1} & A_{4I,3J+2} \\ \hline A_{I+1,3J} & A_{4I+1,3J+2} & A_{4I+1,3J+2} \\ \hline A_{4I+2,3J} & A_{4I+2,3J+3} & A_{4I+2,3J+2} \\ \hline A_{4I+3,3J} & A_{4I+3,3J+3} & A_{4I+3,3J+2} \end{array}\right) + \left(\begin{array}{c} y_{4I} \\ \hline y_{4I+1} \\ \hline y_{4I+2} \\ \hline y_{4I+3} \end{array}\right) \left(\begin{array}{c|} x_{3J} \\ \hline x_{3J+1} \\ \hline x_{3J+2} \end{array}\right)^T .$$

These insights allow us to illustrate an algorithm in Figures 5.5-5.7:

- Initially the data is again distributed as illustrated in Figure 5.5, just as at the start of the two dimensional matrix-vector multiplication algorithm.

- Allgather subvectors x_j within columns of processes at a cost of (approximately)

  $$\log_2(r)\alpha + \frac{r-1}{r}\frac{n}{c}\beta.$$

  (Figure 5.6.)

- Allgather subvectors y_i within rows of processes at a cost of (approximately)

  $$\log_2(c)\alpha + \frac{c-1}{c}\frac{n}{r}\beta.$$

  (Figure 5.7.)

- Perform the local rank-1 update at a cost of (approximately)

  $$2\frac{n^2}{p}\gamma.$$

The total estimated cost of the algorithm is

$$\begin{array}{rcl} T_{r\times c}(n) & = & \log_2(r)\alpha + \frac{r-1}{r}\frac{n}{c}\beta + 2\frac{n^2}{p}\gamma + \log_2(c)\alpha + \frac{c-1}{c}\frac{n}{r}\beta + \frac{c-1}{c}\frac{n}{r}\gamma \\ & = & 2\frac{n^2}{p}\gamma + \log_2(p)\alpha + (r-1)\frac{n}{p}\beta + (c-1)\frac{n}{p}\beta + (c-1)\frac{n}{p}\gamma \\ & = & 2\frac{n^2}{p}\gamma + \log_2(p)\alpha + (r+c-2)\frac{n}{p}\beta + (c-1)\frac{n}{p}\gamma. \end{array}$$

If we assume $r = c = \sqrt{p}$ and we approximate $\sqrt{p}-1$ with $\sqrt{p}$, we get

$$\begin{array}{rcl} T_{\sqrt{p}\times\sqrt{p}}(n) & = & 2\frac{n^2}{p}\gamma + \log_2(p)\alpha + 2(\sqrt{p}-1)\frac{n}{p}\beta + (\sqrt{p}-1)\frac{n}{p}\gamma \\ & \approx & 2\frac{n^2}{p}\gamma + \log_2(p)\alpha + 2\frac{n}{\sqrt{p}}\beta + \frac{n}{\sqrt{p}}\gamma. \end{array}$$

Homework 5.3.2.4 Analyze the scalability of the discussed parallel algorithm for computing a rank-1 update, for the case where $r \times c = \sqrt{p} \times \sqrt{p}$.

☛ SEE ANSWER

Homework 5.3.2.5 Assuming the matrix and vectors are distributed as illustrated in Figure 5.5, propose a parallel algorithm for computing $A = xy^T + A$.

☛ SEE ANSWER

5.4 Toward Parallel Matrix-Matrix Multiplication

5.4.1 Matrix-matrix multiplication via rank-1 updates, Part 1

Recall that if we wish to compute $C := AB + C$, where C is $m \times n$, A is $m \times k$, and B is $k \times n$, we can partition matrix A by columns and matrix B by rows so that

$$\begin{aligned} C &:= AB + C \\ &= \left(a_0 \,\middle|\, a_1 \,\middle|\, \cdots \,\middle|\, a_{k-1} \right) \left(\begin{array}{c} \widetilde{b}_0^T \\ \hline \widetilde{b}_1^T \\ \hline \vdots \\ \hline \widetilde{b}_{k-1}^T \end{array} \right) + C \\ &= a_0 \widetilde{b}_0^T + a_1 \widetilde{b}_1^T + \cdots + a_{k-1} \widetilde{b}_{k-1}^T + C. \end{aligned}$$

It would seem like we can create a parallel matrix-matrix multiplication algorithm by simply putting a loop around the parallel rank-1 update algorithm that we developed in Unit 5.3.2. The problem is that the parallel rank-1 update for computing $A := yx^T + A$ requires the vector y to be distributed to the mesh of processes with a row-major distribution and the vector x with a column-major distribution. Instead, a typical column a_p is distributed, well, as a column of a matrix and a typical row $\widetilde{b}_p^T$ is distributed as a row of a matrix.

5.4.2 Of vectors, rows, and columns

How does one redistribute columns and rows to and from the vector distributions of x and y?

Column to vector to column. Consider a typical column of matrix A, a_p, as illustrated in Figure 5.8. What we notice is:

- Redistributing this column as vector y requires simultaneous scatters within rows of processes.

- Redistributing vector y to this column requires simultaneous gathers within rows of processes.

Row to vector to row. Consider a typical row of matrix A, $\widetilde{b}_p^T$, as illustrated in Figure 5.9. What we notice is:

- Redistributing this row as vector x requires simultaneous scatters within columns of processes.
- Redistributing vector x to this row requires simultaneous gathers within rows of processes.

Vector to vector. As we have encountered before is some of the homeworks: redistributed x to y or y to x merely requires simultaneous sends between the pairs of processes that "own" x_i and y_i (provided the vectors are of equal size and partitioned conformally). This is a very simple case of permutation among processes.

Column to row to column. What we have uncovered is the fact that the vector distributions are intermediate distributions to and from which one can easily redistribute columns and rows of a matrix. Indeed, a column can be copied into a row by first scattering within rows of processes, leaving it distributed like y, followed by a redistribution from y to x, and finally a gather within columns of processes.

5.4.3 Matrix-matrix multiplication via rank-1 updates, Part 2

With the insights from the last two units, we are ready to propose a parallel matrix-matrix multiplication algorithm based on

$$\begin{aligned} C &:= AB + C \\ &= \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{k-1} \end{array} \right) \left(\begin{array}{c} \widetilde{b}_0^T \\ \hline \widetilde{b}_1^T \\ \hline \vdots \\ \hline \widetilde{b}_{k-1}^T \end{array} \right) + C \\ &= a_0 \widetilde{b}_0^T + a_1 \widetilde{b}_1^T + \cdots + a_{k-1} \widetilde{b}_{k-1}^T + C. \end{aligned}$$

- **for** $p = 0, \ldots, k-1$
  - Redistribute $\widetilde{b}_p^T$ like x. (Scatter within columns of processes.
  - Redistribute a_p like y. (Scatter within rows of processes.)
  - Allgather x within columns of processes.
  - Allgather y within rows of processes.

- Update the local part of C with the local parts of the duplicated vectors x and y.

Let's play with this a bit. We can rearrange the steps as

- **for** $p = 0, \ldots, k-1$
  - Redistribute $\widetilde{b}_p^T$ like x. (Scatter within columns of processes).
  - Allgather x within columns of processes.
  - Redistribute a_p like y. (Scatter within rows of processes.)
  - Allgather y within rows of processes.
  - Update the local part of C with the local parts of the duplicated vectors x and y.

Next, we recognize that a scatter within columns of processes followed by an allgather within columns of processes is the same as a broadcast within columns of processes. Similarly, a scatter within rows of processes followed by an allgather within rows of processes is the same as a broadcast within rows of processes. This leaves us with

- **for** $p = 0, \ldots, k-1$
  - Broadcast $\widetilde{b}_p^T$ within columns of processes.
    (Figure 5.10 to 5.11.)
  - Broadcast a_p within rows of processes.
    (Figure 5.12 to 5.13.)
  - Update the local part of C with the local parts of the duplicated $\widetilde{b}_p^T$ and a_p.
    (Figure 5.14.)

Homework 5.4.3.6 Analyze the cost and scalability of the described parallel matrix-matrix multiplication algorithm, assuming a $\sqrt{p} \times \sqrt{p}$ process mesh and using $\log_2(p)\alpha + n\beta$ as the cost of a broadcast of n items among p processes.

☛ SEE ANSWER

The reason why we like to break down the communications and computations to include the redistributions into those that involve vectors x and y becomes clear when one considers the following exercises:

Homework 5.4.3.7 Consider the computation of $C := AB^T + C$ via rank-1 updates:

$$\begin{array}{rcl} C & := & AB^T + C \\ & = & \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{k-1} \end{array} \right) \left(\begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{k-1} \end{array} \right)^T + C \\ & = & \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{k-1} \end{array} \right) \left(\begin{array}{c} b_0^T \\ \hline b_1^T \\ \hline \vdots \\ \hline b_{k-1}^T \end{array} \right) + C \\ & = & a_0 b_0^T + a_1 b_1^T + \cdots + a_{k-1} b_{k-1}^T + C. \end{array}$$

Propose a parallel algorithm that casts the computation in terms of local rank-1 updates.

☛ SEE ANSWER

Homework 5.4.3.8 Consider the computation of $C := A^T B + C$. Propose a parallel algorithm that casts the computation in terms of local rank-1 updates.

☛ SEE ANSWER

Homework 5.4.3.9 Consider the computation of $C := A^T B + C$. Propose a parallel algorithm that casts the computation in terms of local rank-1 updates.

☛ SEE ANSWER

5.4.4 Matrix-matrix multiplication via matrix-vector multiplications

Recall that, alternatively, if we wish to compute $C := AB + C$, where C is $m \times n$, A is $m \times k$, and B is $k \times n$, we can partition matrices C and B by columns so that

$$\begin{array}{rcl} \left(\begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1} \end{array} \right) & = & A \left(\begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{n-1} \end{array} \right) + \left(\begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1} \end{array} \right) \\ & = & \left(\begin{array}{c|c|c|c} Ab_0 + c_0 & Ab_1 + c_1 & \cdots & Ab_{n-1} + c_{n-1} \end{array} \right). \end{array}$$

We can now implement a parallel matrix-matrix multiplication algorithm by putting a loop around the parallel matrix-vector update algorithm that we developed in Unit **??**. Again, the problem is that parallel matrix-vector multiply requires the vectors x and y to be distributed differently than columns of C and B are. But, by now the solution should be obvious to you.

Homework 5.4.4.10 Consider the computation of $C := AB + C$. Propose a parallel algorithm that casts the computation in terms of local matrix-vector multiplications.

☛ SEE ANSWER

Homework 5.4.4.11 Consider the computation of $C := AB^T + C$. Propose a parallel algorithm that casts the computation in terms of local matrix-vector multiplications.

☛ SEE ANSWER

5.4.5 High-Performance Parallel Matrix-Matrix Multiplication

While the algorithms discussed in the last units are *weakly scalable*, they are *not* high performing. The reason is simple: the local computations are cast in terms of a rank-1 update (for the algorithms in Unit 5.4.3) or matrix-vector multiplications (for the algorithms in Unit 5.4.4). These operations perform $O(1)$ computations for each memory operation, and hence will not attain high performance for the single process computations.

The solution is quite simple. Let us start by focusing on the algorithm that casts $C := AB + C$ in terms of rank-1 updates. If we partition A into blocks of columns and B into blocks of rows, of appropriate size, then we get that

$$\begin{aligned} C &:= AB + C \\ &= \left(A_0 \,\middle|\, A_1 \,\middle|\, \cdots \,\middle|\, A_{K-1} \right) \left(\begin{array}{c} \widetilde{B}_0 \\ \hline \widetilde{B}_1 \\ \hline \vdots \\ \hline \widetilde{B}_{K-1} \end{array} \right) + C \\ &= A_0\widetilde{B}_0 + A_1\widetilde{B}_1 + \cdots + A_{K-1}\widetilde{B}_{K-1} + C. \end{aligned}$$

What we notice is that instead of broadcasting the appropriate parts of a_p within rows of processes and $\widetilde{b}_p^T$ within columns of processes, we instead broadcast the appropriate parts of A_p within rows of processes and $\widetilde{B}_p$ within columns of processes. The width of A_p (and corresponding height of $\widetilde{B}_p$) is dictates by the block size k_c encountered in Week **??**. The important point is that this casts the local computation in terms of a matrix-matrix multiplication.

Similarly, we can take the algorithm that casts the computation in terms of matrix-vector multiplications and cast it in terms of a matrix times a panel of vectors instead:

$$\begin{aligned} \left(C_0 \,\middle|\, C_1 \,\middle|\, \cdots \,\middle|\, C_{N-1} \right) &= A\left(B_0 \,\middle|\, B_1 \,\middle|\, \cdots \,\middle|\, B_{N-1} \right) + \left(C_0 \,\middle|\, C_1 \,\middle|\, \cdots \,\middle|\, C_{N-1} \right) \\ &= \left(AB_0 + C_0 \,\middle|\, AB_1 + C_1 \,\middle|\, \cdots \,\middle|\, AB_{N-1} + C_{N-1} \right). \end{aligned}$$

It should be obvious how to modify the algorithms from Unit 5.4.4 appropriately. Again, the local computation now becomes a matrix-matrix multiplication.

5.5 Enrichment

5.6 Wrapup

x_0 x_1 x_2	x_3 x_4 x_5	x_6 x_7 x_8	x_9 x_{10} x_{11}
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$	$A_{0,3}$ $A_{0,4}$ $A_{0,5}$	$A_{0,6}$ $A_{0,7}$ $A_{0,8}$	$A_{0,9}$ $A_{0,10}$ $A_{0,11}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$	$A_{1,3}$ $A_{1,4}$ $A_{1,5}$	$A_{1,6}$ $A_{1,7}$ $A_{1,8}$	$A_{1,9}$ $A_{1,10}$ $A_{1,11}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$	$A_{2,3}$ $A_{2,4}$ $A_{2,5}$	$A_{2,6}$ $A_{2,7}$ $A_{2,8}$	$A_{2,9}$ $A_{2,10}$ $A_{2,11}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$	$A_{3,3}$ $A_{3,4}$ $A_{3,5}$	$A_{3,6}$ $A_{3,7}$ $A_{3,8}$	$A_{3,9}$ $A_{3,10}$ $A_{3,11}$
x_0 x_1 x_2	x_3 x_4 x_5	x_6 x_7 x_8	x_9 x_{10} x_{11}
$A_{4,0}$ $A_{4,1}$ $A_{4,2}$	$A_{4,3}$ $A_{4,4}$ $A_{4,5}$	$A_{4,6}$ $A_{4,7}$ $A_{4,8}$	$A_{4,9}$ $A_{4,10}$ $A_{4,11}$
$A_{5,0}$ $A_{5,1}$ $A_{5,2}$	$A_{5,3}$ $A_{5,4}$ $A_{5,5}$	$A_{5,6}$ $A_{5,7}$ $A_{5,8}$	$A_{5,9}$ $A_{5,10}$ $A_{5,11}$
$A_{6,0}$ $A_{6,1}$ $A_{6,2}$	$A_{6,3}$ $A_{6,4}$ $A_{6,5}$	$A_{6,6}$ $A_{6,7}$ $A_{6,8}$	$A_{6,9}$ $A_{6,10}$ $A_{6,11}$
$A_{7,0}$ $A_{7,1}$ $A_{7,2}$	$A_{7,3}$ $A_{7,4}$ $A_{7,5}$	$A_{7,6}$ $A_{7,7}$ $A_{7,8}$	$A_{7,9}$ $A_{7,10}$ $A_{7,11}$
x_0 x_1 x_2	x_3 x_4 x_5	x_6 x_7 x_8	x_9 x_{10} x_{11}
$A_{8,0}$ $A_{8,1}$ $A_{8,2}$	$A_{8,3}$ $A_{8,4}$ $A_{8,5}$	$A_{8,6}$ $A_{8,7}$ $A_{8,8}$	$A_{8,9}$ $A_{8,10}$ $A_{8,11}$
$A_{9,0}$ $A_{9,1}$ $A_{9,2}$	$A_{9,3}$ $A_{9,4}$ $A_{9,5}$	$A_{9,6}$ $A_{9,7}$ $A_{9,8}$	$A_{9,9}$ $A_{9,10}$ $A_{9,11}$
$A_{10,0}$ $A_{10,1}$ $A_{10,2}$	$A_{10,3}$ $A_{10,4}$ $A_{10,5}$	$A_{10,6}$ $A_{10,7}$ $A_{10,8}$	$A_{10,9}$ $A_{10,10}$ $A_{10,11}$
$A_{11,0}$ $A_{11,1}$ $A_{11,2}$	$A_{11,3}$ $A_{11,4}$ $A_{11,5}$	$A_{11,6}$ $A_{11,7}$ $A_{11,8}$	$A_{11,9}$ $A_{11,10}$ $A_{11,11}$

Figure 5.2: Parallel matrix-vector multiplication: Allgather x within columns of processes.

$$
\begin{array}{|cccc|cccc|cccc|cccc|}
\hline
 & x_0 & x_1 & x_2 & & x_3 & x_4 & x_5 & & x_6 & x_7 & x_8 & & x_9 & x_{10} & x_{11} \\
y_0^{(0)} & A_{0,0} & A_{0,1} & A_{0,2} & y_0^{(1)} & A_{0,3} & A_{0,4} & A_{0,5} & y_0^{(2)} & A_{0,6} & A_{0,7} & A_{0,8} & y_0^{(3)} & A_{0,9} & A_{0,10} & A_{0,11} \\
y_1^{(0)} & A_{1,0} & A_{1,1} & A_{1,2} & y_1^{(1)} & A_{1,3} & A_{1,4} & A_{1,5} & y_1^{(2)} & A_{1,6} & A_{1,7} & A_{1,8} & y_1^{(3)} & A_{1,9} & A_{1,10} & A_{1,11} \\
y_2^{(0)} & A_{2,0} & A_{2,1} & A_{2,2} & y_2^{(1)} & A_{2,3} & A_{2,4} & A_{2,5} & y_2^{(2)} & A_{2,6} & A_{2,7} & A_{2,8} & y_2^{(3)} & A_{2,9} & A_{2,10} & A_{2,11} \\
y_3^{(0)} & A_{3,0} & A_{3,1} & A_{3,2} & y_3^{(1)} & A_{3,3} & A_{3,4} & A_{3,5} & y_3^{(2)} & A_{3,6} & A_{3,7} & A_{3,8} & y_3^{(3)} & A_{3,9} & A_{3,10} & A_{3,11} \\
\hline
 & x_0 & x_1 & x_2 & & x_3 & x_4 & x_5 & & x_6 & x_7 & x_8 & & x_9 & x_{10} & x_{11} \\
y_4^{(0)} & A_{4,0} & A_{4,1} & A_{4,2} & y_4^{(1)} & A_{4,3} & A_{4,4} & A_{4,5} & y_4^{(2)} & A_{4,6} & A_{4,7} & A_{4,8} & y_4^{(3)} & A_{4,9} & A_{4,10} & A_{4,11} \\
y_5^{(0)} & A_{5,0} & A_{5,1} & A_{5,2} & y_5^{(1)} & A_{5,3} & A_{5,4} & A_{5,5} & y_5^{(2)} & A_{5,6} & A_{5,7} & A_{5,8} & y_5^{(3)} & A_{5,9} & A_{5,10} & A_{5,11} \\
y_6^{(0)} & A_{6,0} & A_{6,1} & A_{6,2} & y_6^{(1)} & A_{6,3} & A_{6,4} & A_{6,5} & y_6^{(2)} & A_{6,6} & A_{6,7} & A_{6,8} & y_6^{(3)} & A_{6,9} & A_{6,10} & A_{6,11} \\
y_7^{(0)} & A_{7,0} & A_{7,1} & A_{7,2} & y_7^{(1)} & A_{7,3} & A_{7,4} & A_{7,5} & y_7^{(2)} & A_{7,6} & A_{7,7} & A_{7,8} & y_7^{(3)} & A_{7,9} & A_{7,10} & A_{7,11} \\
\hline
 & x_0 & x_1 & x_2 & & x_3 & x_4 & x_5 & & x_6 & x_7 & x_8 & & x_9 & x_{10} & x_{11} \\
y_8^{(0)} & A_{8,0} & A_{8,1} & A_{8,2} & y_8^{(1)} & A_{8,3} & A_{8,4} & A_{8,5} & y_8^{(2)} & A_{8,6} & A_{8,7} & A_{8,8} & y_8^{(3)} & A_{8,9} & A_{8,10} & A_{8,11} \\
y_9^{(0)} & A_{9,0} & A_{9,1} & A_{9,2} & y_9^{(1)} & A_{9,3} & A_{9,4} & A_{9,5} & y_9^{(2)} & A_{9,6} & A_{9,7} & A_{9,8} & y_9^{(3)} & A_{9,9} & A_{9,10} & A_{9,11} \\
y_{10}^{(0)} & A_{10,0} & A_{10,1} & A_{10,2} & y_{10}^{(1)} & A_{10,3} & A_{10,4} & A_{10,5} & y_{10}^{(2)} & A_{10,6} & A_{10,7} & A_{10,8} & y_{10}^{(3)} & A_{10,9} & A_{10,10} & A_{10,11} \\
y_{11}^{(0)} & A_{11,0} & A_{11,1} & A_{11,2} & y_{11}^{(1)} & A_{11,3} & A_{11,4} & A_{11,5} & y_{11}^{(2)} & A_{11,6} & A_{11,7} & A_{11,8} & y_{11}^{(3)} & A_{11,9} & A_{11,10} & A_{11,11} \\
\hline
\end{array}
$$

Figure 5.3: Parallel matrix-vector multiplication: Computation of local contributions to y.

y_0	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$		$A_{0,3}$	$A_{0,4}$	$A_{0,5}$		$A_{0,6}$	$A_{0,7}$	$A_{0,8}$		$A_{0,9}$	$A_{0,10}$	$A_{0,11}$
	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	y_1	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$		$A_{1,6}$	$A_{1,7}$	$A_{1,8}$		$A_{1,9}$	$A_{1,10}$	$A_{1,11}$
	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$		$A_{2,3}$	$A_{2,4}$	$A_{2,5}$	y_2	$A_{2,6}$	$A_{2,7}$	$A_{2,8}$		$A_{2,9}$	$A_{2,10}$	$A_{2,11}$
	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$		$A_{3,3}$	$A_{3,4}$	$A_{3,5}$		$A_{3,6}$	$A_{3,7}$	$A_{3,8}$	y_3	$A_{3,9}$	$A_{3,10}$	$A_{3,11}$
y_4	$A_{4,0}$	$A_{4,1}$	$A_{4,2}$		$A_{4,3}$	$A_{4,4}$	$A_{4,5}$		$A_{4,6}$	$A_{4,7}$	$A_{4,8}$		$A_{4,9}$	$A_{4,10}$	$A_{4,11}$
	$A_{5,0}$	$A_{5,1}$	$A_{5,2}$	y_5	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$		$A_{5,6}$	$A_{5,7}$	$A_{5,8}$		$A_{5,9}$	$A_{5,10}$	$A_{5,11}$
	$A_{6,0}$	$A_{6,1}$	$A_{6,2}$		$A_{6,3}$	$A_{6,4}$	$A_{6,5}$	y_6	$A_{6,6}$	$A_{6,7}$	$A_{6,8}$		$A_{6,9}$	$A_{6,10}$	$A_{6,11}$
	$A_{7,0}$	$A_{7,1}$	$A_{7,2}$		$A_{7,3}$	$A_{7,4}$	$A_{7,5}$		$A_{7,6}$	$A_{7,7}$	$A_{7,8}$	y_7	$A_{7,9}$	$A_{7,10}$	$A_{7,11}$
y_8	$A_{8,0}$	$A_{8,1}$	$A_{8,2}$		$A_{8,3}$	$A_{8,4}$	$A_{8,5}$		$A_{8,6}$	$A_{8,7}$	$A_{8,8}$		$A_{8,9}$	$A_{8,10}$	$A_{8,11}$
	$A_{9,0}$	$A_{9,1}$	$A_{9,2}$	y_9	$A_{9,3}$	$A_{9,4}$	$A_{9,5}$		$A_{9,6}$	$A_{9,7}$	$A_{9,8}$		$A_{9,9}$	$A_{9,10}$	$A_{9,11}$
	$A_{10,0}$	$A_{10,1}$	$A_{10,2}$		$A_{10,3}$	$A_{10,4}$	$A_{10,5}$	y_{10}	$A_{10,6}$	$A_{10,7}$	$A_{10,8}$		$A_{10,9}$	$A_{10,10}$	$A_{10,11}$
	$A_{11,0}$	$A_{11,1}$	$A_{11,2}$		$A_{11,3}$	$A_{11,4}$	$A_{11,5}$		$A_{11,6}$	$A_{11,7}$	$A_{11,8}$	y_{11}	$A_{11,9}$	$A_{11,10}$	$A_{11,11}$

Figure 5.4: Parallel matrix-vector multiplication: Result after reduce-scatter within rows of processes.

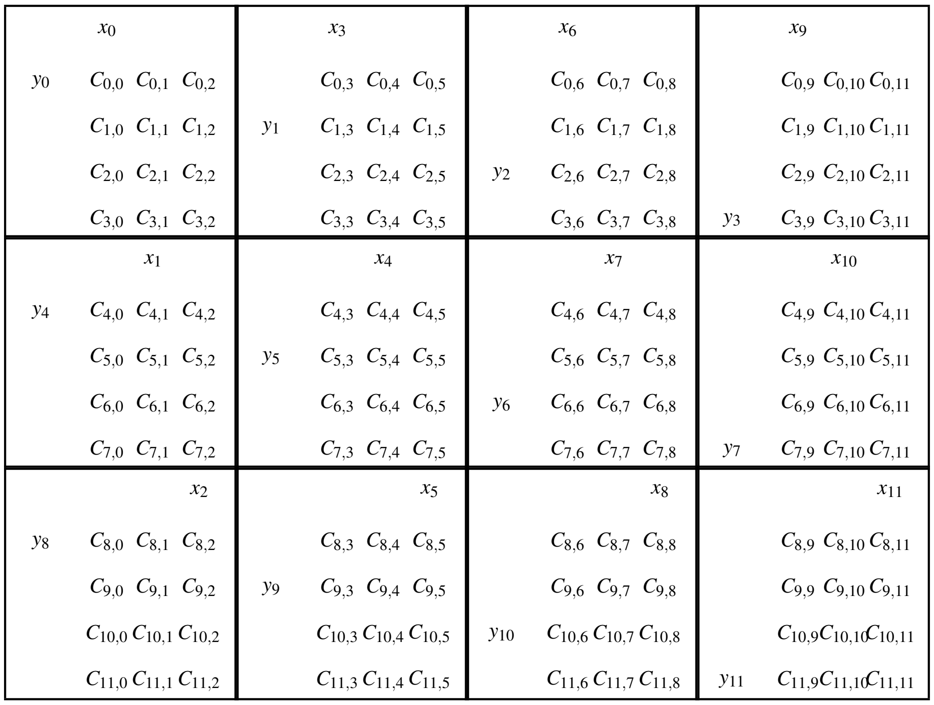

x_0 x_3 x_6 x_9
y_0 $C_{0,0}$ $C_{0,1}$ $C_{0,2}$ $C_{0,3}$ $C_{0,4}$ $C_{0,5}$ $C_{0,6}$ $C_{0,7}$ $C_{0,8}$ $C_{0,9}$ $C_{0,10}$ $C_{0,11}$
$C_{1,0}$ $C_{1,1}$ $C_{1,2}$ y_1 $C_{1,3}$ $C_{1,4}$ $C_{1,5}$ $C_{1,6}$ $C_{1,7}$ $C_{1,8}$ $C_{1,9}$ $C_{1,10}$ $C_{1,11}$
$C_{2,0}$ $C_{2,1}$ $C_{2,2}$ $C_{2,3}$ $C_{2,4}$ $C_{2,5}$ y_2 $C_{2,6}$ $C_{2,7}$ $C_{2,8}$ $C_{2,9}$ $C_{2,10}$ $C_{2,11}$
$C_{3,0}$ $C_{3,1}$ $C_{3,2}$ $C_{3,3}$ $C_{3,4}$ $C_{3,5}$ $C_{3,6}$ $C_{3,7}$ $C_{3,8}$ y_3 $C_{3,9}$ $C_{3,10}$ $C_{3,11}$
x_1 x_4 x_7 x_{10}
y_4 $C_{4,0}$ $C_{4,1}$ $C_{4,2}$ $C_{4,3}$ $C_{4,4}$ $C_{4,5}$ $C_{4,6}$ $C_{4,7}$ $C_{4,8}$ $C_{4,9}$ $C_{4,10}$ $C_{4,11}$
$C_{5,0}$ $C_{5,1}$ $C_{5,2}$ y_5 $C_{5,3}$ $C_{5,4}$ $C_{5,5}$ $C_{5,6}$ $C_{5,7}$ $C_{5,8}$ $C_{5,9}$ $C_{5,10}$ $C_{5,11}$
$C_{6,0}$ $C_{6,1}$ $C_{6,2}$ $C_{6,3}$ $C_{6,4}$ $C_{6,5}$ y_6 $C_{6,6}$ $C_{6,7}$ $C_{6,8}$ $C_{6,9}$ $C_{6,10}$ $C_{6,11}$
$C_{7,0}$ $C_{7,1}$ $C_{7,2}$ $C_{7,3}$ $C_{7,4}$ $C_{7,5}$ $C_{7,6}$ $C_{7,7}$ $C_{7,8}$ y_7 $C_{7,9}$ $C_{7,10}$ $C_{7,11}$
x_2 x_5 x_8 x_{11}
y_8 $C_{8,0}$ $C_{8,1}$ $C_{8,2}$ $C_{8,3}$ $C_{8,4}$ $C_{8,5}$ $C_{8,6}$ $C_{8,7}$ $C_{8,8}$ $C_{8,9}$ $C_{8,10}$ $C_{8,11}$
$C_{9,0}$ $C_{9,1}$ $C_{9,2}$ y_9 $C_{9,3}$ $C_{9,4}$ $C_{9,5}$ $C_{9,6}$ $C_{9,7}$ $C_{9,8}$ $C_{9,9}$ $C_{9,10}$ $C_{9,11}$
$C_{10,0}$ $C_{10,1}$ $C_{10,2}$ $C_{10,3}$ $C_{10,4}$ $C_{10,5}$ y_{10} $C_{10,6}$ $C_{10,7}$ $C_{10,8}$ $C_{10,9}$ $C_{10,10}$ $C_{10,11}$
$C_{11,0}$ $C_{11,1}$ $C_{11,2}$ $C_{11,3}$ $C_{11,4}$ $C_{11,5}$ $C_{11,6}$ $C_{11,7}$ $C_{11,8}$ y_{11} $C_{11,9}$ $C_{11,10}$ $C_{11,11}$

Figure 5.5: Parallel rank-1 update: Initial distribution of matrix and vectors before $C := yx^T + C$.

	x_0	x_1	x_2		x_3	x_4	x_5		x_6	x_7	x_8		x_9	x_{10}	x_{11}
y_0	$C_{0,0}$	$C_{0,1}$	$C_{0,2}$		$C_{0,3}$	$C_{0,4}$	$C_{0,5}$		$C_{0,6}$	$C_{0,7}$	$C_{0,8}$		$C_{0,9}$	$C_{0,10}$	$C_{0,11}$
	$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	y_1	$C_{1,3}$	$C_{1,4}$	$C_{1,5}$		$C_{1,6}$	$C_{1,7}$	$C_{1,8}$		$C_{1,9}$	$C_{1,10}$	$C_{1,11}$
	$C_{2,0}$	$C_{2,1}$	$C_{2,2}$		$C_{2,3}$	$C_{2,4}$	$C_{2,5}$	y_2	$C_{2,6}$	$C_{2,7}$	$C_{2,8}$		$C_{2,9}$	$C_{2,10}$	$C_{2,11}$
	$C_{3,0}$	$C_{3,1}$	$C_{3,2}$		$C_{3,3}$	$C_{3,4}$	$C_{3,5}$		$C_{3,6}$	$C_{3,7}$	$C_{3,8}$	y_3	$C_{3,9}$	$C_{3,10}$	$C_{3,11}$
	x_0	x_1	x_2		x_3	x_4	x_5		x_6	x_7	x_8		x_9	x_{10}	x_{11}
y_4	$C_{4,0}$	$C_{4,1}$	$C_{4,2}$		$C_{4,3}$	$C_{4,4}$	$C_{4,5}$		$C_{4,6}$	$C_{4,7}$	$C_{4,8}$		$C_{4,9}$	$C_{4,10}$	$C_{4,11}$
	$C_{5,0}$	$C_{5,1}$	$C_{5,2}$	y_5	$C_{5,3}$	$C_{5,4}$	$C_{5,5}$		$C_{5,6}$	$C_{5,7}$	$C_{5,8}$		$C_{5,9}$	$C_{5,10}$	$C_{5,11}$
	$C_{6,0}$	$C_{6,1}$	$C_{6,2}$		$C_{6,3}$	$C_{6,4}$	$C_{6,5}$	y_6	$C_{6,6}$	$C_{6,7}$	$C_{6,8}$		$C_{6,9}$	$C_{6,10}$	$C_{6,11}$
	$C_{7,0}$	$C_{7,1}$	$C_{7,2}$		$C_{7,3}$	$C_{7,4}$	$C_{7,5}$		$C_{7,6}$	$C_{7,7}$	$C_{7,8}$	y_7	$C_{7,9}$	$C_{7,10}$	$C_{7,11}$
	x_0	x_1	x_2		x_3	x_4	x_5		x_6	x_7	x_8		x_9	x_{10}	x_{11}
y_8	$C_{8,0}$	$C_{8,1}$	$C_{8,2}$		$C_{8,3}$	$C_{8,4}$	$C_{8,5}$		$C_{8,6}$	$C_{8,7}$	$C_{8,8}$		$C_{8,9}$	$C_{8,10}$	$C_{8,11}$
	$C_{9,0}$	$C_{9,1}$	$C_{9,2}$	y_9	$C_{9,3}$	$C_{9,4}$	$C_{9,5}$		$C_{9,6}$	$C_{9,7}$	$C_{9,8}$		$C_{9,9}$	$C_{9,10}$	$C_{9,11}$
	$C_{10,0}$	$C_{10,1}$	$C_{10,2}$		$C_{10,3}$	$C_{10,4}$	$C_{10,5}$	y_{10}	$C_{10,6}$	$C_{10,7}$	$C_{10,8}$		$C_{10,9}$	$C_{10,10}$	$C_{10,11}$
	$C_{11,0}$	$C_{11,1}$	$C_{11,2}$		$C_{11,3}$	$C_{11,4}$	$C_{11,5}$		$C_{11,6}$	$C_{11,7}$	$C_{11,8}$	y_{11}	$C_{11,9}$	$C_{11,10}$	$C_{11,11}$

Figure 5.6: Parallel rank-1 update: Allgather x within columns of processes.

	x_0	x_1	x_2		x_3	x_4	x_5		x_6	x_7	x_8		x_9	x_{10}	x_{11}
y_0	$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	y_0	$C_{0,3}$	$C_{0,4}$	$C_{0,5}$	y_0	$C_{0,6}$	$C_{0,7}$	$C_{0,8}$	y_0	$C_{0,9}$	$C_{0,10}$	$C_{0,11}$
y_1	$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	y_1	$C_{1,3}$	$C_{1,4}$	$C_{1,5}$	y_1	$C_{1,6}$	$C_{1,7}$	$C_{1,8}$	y_1	$C_{1,9}$	$C_{1,10}$	$C_{1,11}$
y_2	$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	y_2	$C_{2,3}$	$C_{2,4}$	$C_{2,5}$	y_2	$C_{2,6}$	$C_{2,7}$	$C_{2,8}$	y_2	$C_{2,9}$	$C_{2,10}$	$C_{2,11}$
y_3	$C_{3,0}$	$C_{3,1}$	$C_{3,2}$	y_3	$C_{3,3}$	$C_{3,4}$	$C_{3,5}$	y_3	$C_{3,6}$	$C_{3,7}$	$C_{3,8}$	y_3	$C_{3,9}$	$C_{3,10}$	$C_{3,11}$
	x_0	x_1	x_2		x_3	x_4	x_5		x_6	x_7	x_8		x_9	x_{10}	x_{11}
y_4	$C_{4,0}$	$C_{4,1}$	$C_{4,2}$	y_4	$C_{4,3}$	$C_{4,4}$	$C_{4,5}$	y_4	$C_{4,6}$	$C_{4,7}$	$C_{4,8}$	y_4	$C_{4,9}$	$C_{4,10}$	$C_{4,11}$
y_5	$C_{5,0}$	$C_{5,1}$	$C_{5,2}$	y_5	$C_{5,3}$	$C_{5,4}$	$C_{5,5}$	y_5	$C_{5,6}$	$C_{5,7}$	$C_{5,8}$	y_5	$C_{5,9}$	$C_{5,10}$	$C_{5,11}$
y_6	$C_{6,0}$	$C_{6,1}$	$C_{6,2}$	y_6	$C_{6,3}$	$C_{6,4}$	$C_{6,5}$	y_6	$C_{6,6}$	$C_{6,7}$	$C_{6,8}$	y_6	$C_{6,9}$	$C_{6,10}$	$C_{6,11}$
y_7	$C_{7,0}$	$C_{7,1}$	$C_{7,2}$	y_7	$C_{7,3}$	$C_{7,4}$	$C_{7,5}$	y_7	$C_{7,6}$	$C_{7,7}$	$C_{7,8}$	y_7	$C_{7,9}$	$C_{7,10}$	$C_{7,11}$
	x_0	x_1	x_2		x_3	x_4	x_5		x_6	x_7	x_8		x_9	x_{10}	x_{11}
y_8	$C_{8,0}$	$C_{8,1}$	$C_{8,2}$	y_8	$C_{8,3}$	$C_{8,4}$	$C_{8,5}$	y_8	$C_{8,6}$	$C_{8,7}$	$C_{8,8}$	y_8	$C_{8,9}$	$C_{8,10}$	$C_{8,11}$
y_9	$C_{9,0}$	$C_{9,1}$	$C_{9,2}$	y_9	$C_{9,3}$	$C_{9,4}$	$C_{9,5}$	y_9	$C_{9,6}$	$C_{9,7}$	$C_{9,8}$	y_9	$C_{9,9}$	$C_{9,10}$	$C_{9,11}$
y_{10}	$C_{10,0}$	$C_{10,1}$	$C_{10,2}$	y_{10}	$C_{10,3}$	$C_{10,4}$	$C_{10,5}$	y_{10}	$C_{10,6}$	$C_{10,7}$	$C_{10,8}$	y_{10}	$C_{10,9}$	$C_{10,10}$	$C_{10,11}$
y_{11}	$C_{11,0}$	$C_{11,1}$	$C_{11,2}$	y_{11}	$C_{11,3}$	$C_{11,4}$	$C_{11,5}$	y_{11}	$C_{11,6}$	$C_{11,7}$	$C_{11,8}$	y_{11}	$C_{11,9}$	$C_{11,10}$	$C_{11,11}$

Figure 5.7: Parallel rank-1 update: Allgather y within rows of processes and perform local rank-1 update.

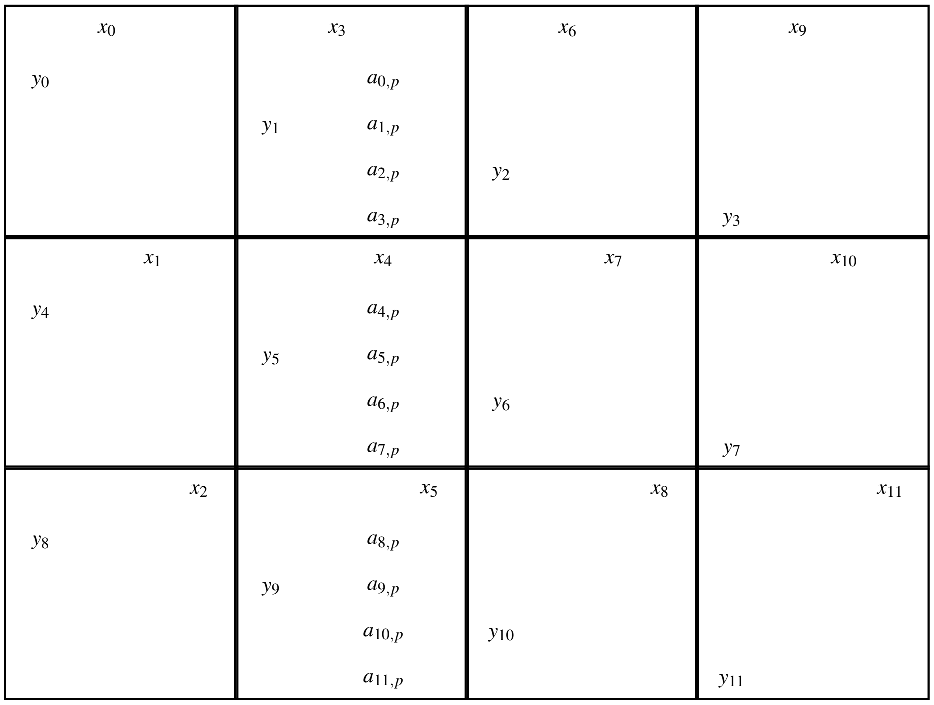


Figure 5.8: Distribution of a typical column in matrix A, a_p.

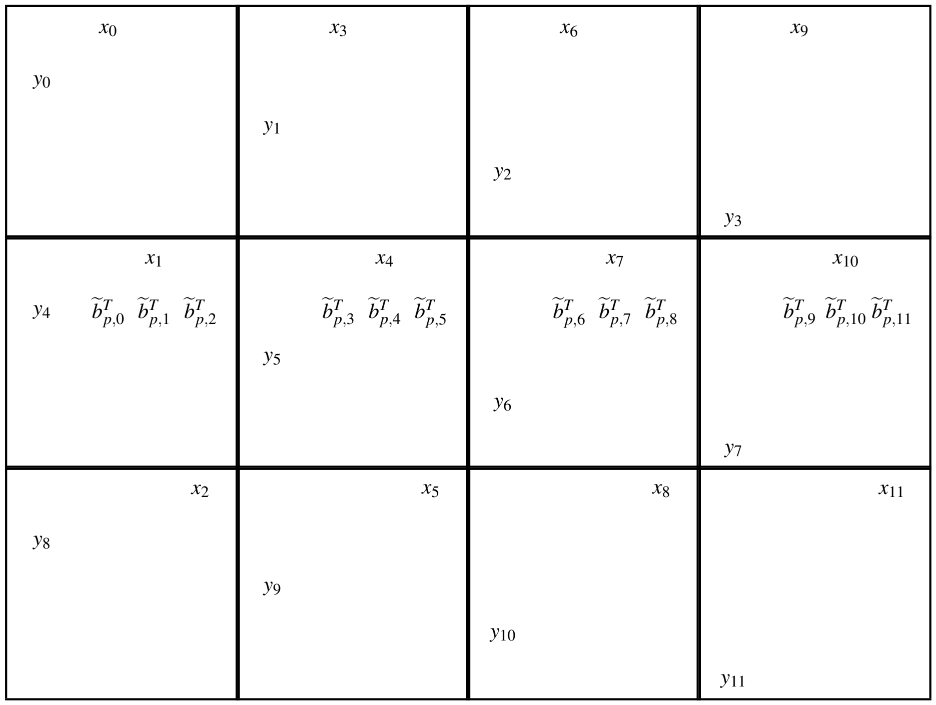

x_0
y_0
x_3
y_1
x_6
y_2
x_9
y_3
x_1
y_4 $\widetilde{b}^T_{p,0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
x_4
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
y_5
x_7
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
y_6
x_{10}
$\widetilde{b}^T_{p,9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$
y_7
x_2
y_8
x_5
y_9
x_8
y_{10}
x_{11}
y_{11}

Figure 5.9: Distribution of a typical row in matrix A, $\widetilde{b}^T_p$.

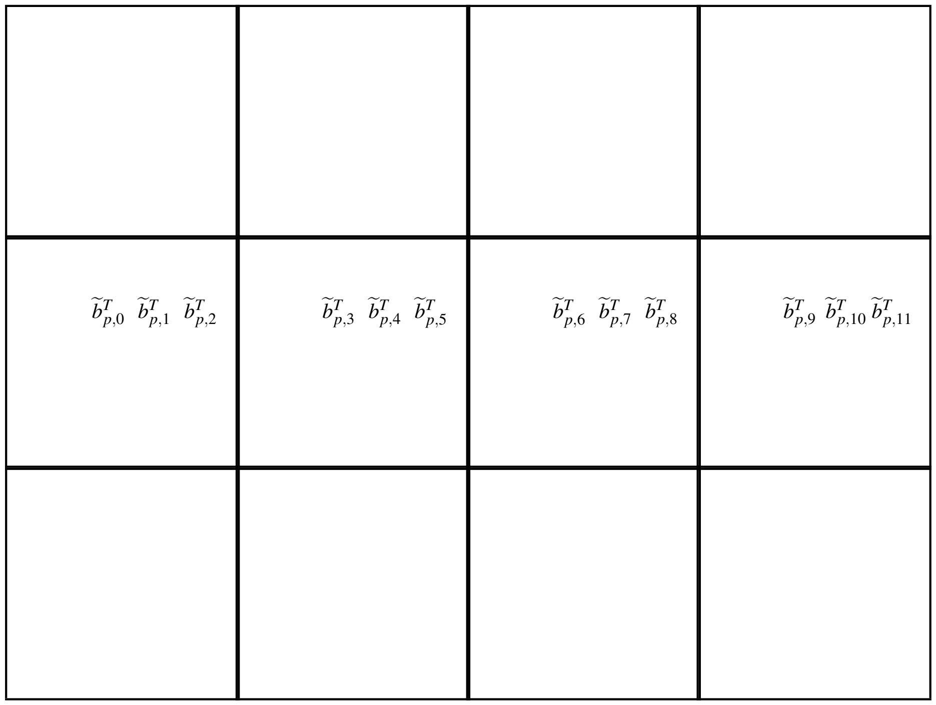

$\widetilde{b}^T_{p,0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
$\widetilde{b}^T_{p,9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$

Figure 5.10: Broadcast $\widehat{b}^T_p$ within rows of processors (before).

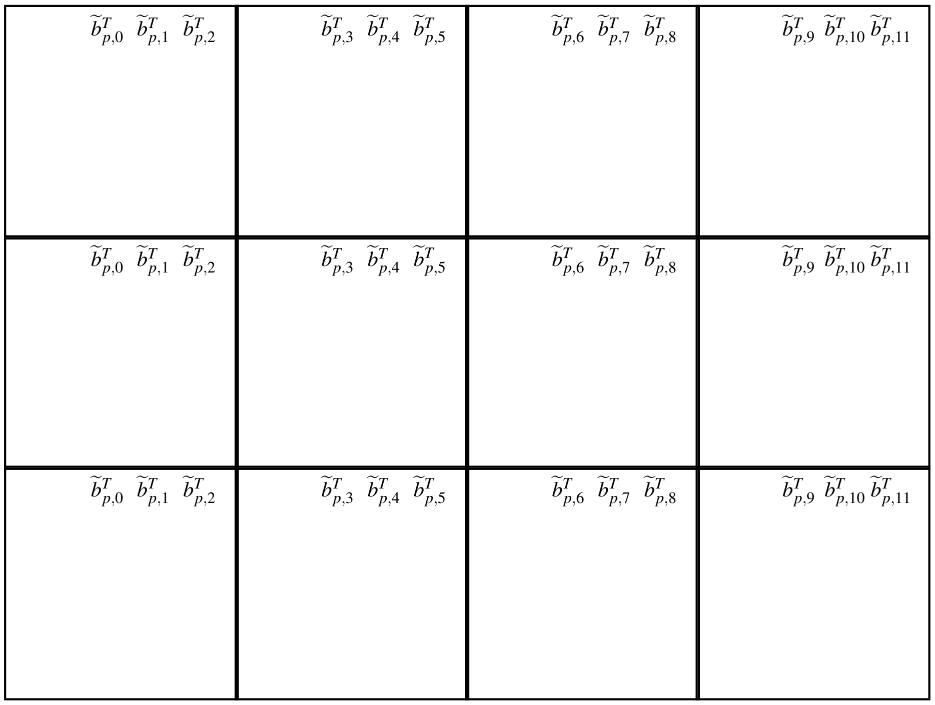

$\widetilde{b}^T_{p,0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
$\widetilde{b}^T_{p,9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$
$\widetilde{b}^T_{p,0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
$\widetilde{b}^T_{p,9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$
$\widetilde{b}^T_{p,0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
$\widetilde{b}^T_{p,9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$

Figure 5.11: Broadcast $\widetilde{b}^T_p$ within rows of processors (after).

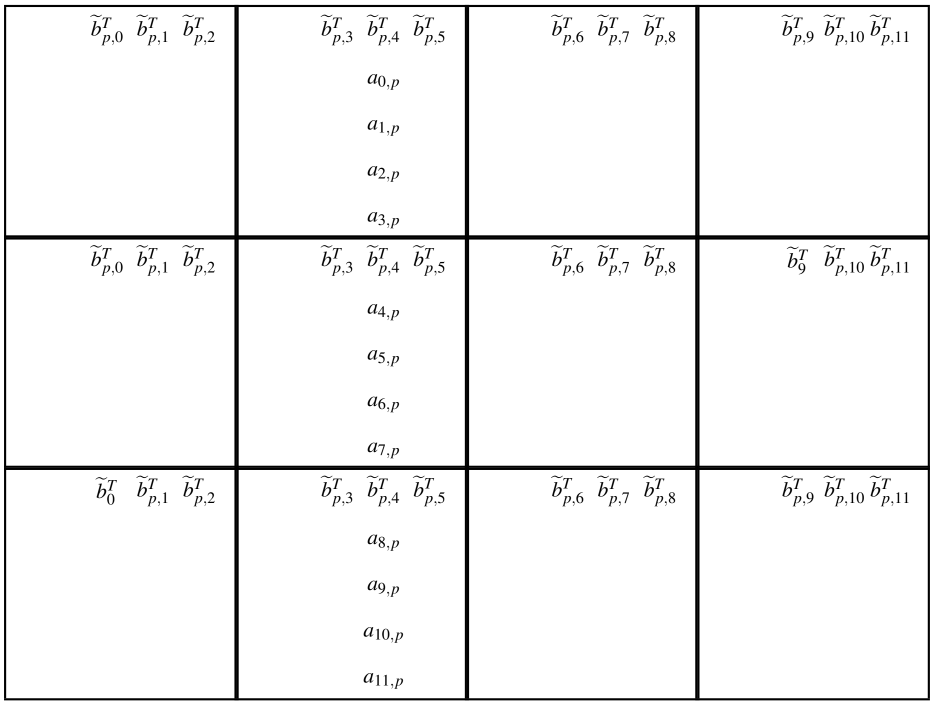

$\widetilde{b}^T_{p,0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
$\widetilde{b}^T_{p,9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$
$a_{0,p}$
$a_{1,p}$
$a_{2,p}$
$a_{3,p}$
$\widetilde{b}^T_{p,0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
$\widetilde{b}^T_{9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$
$a_{4,p}$
$a_{5,p}$
$a_{6,p}$
$a_{7,p}$
$\widetilde{b}^T_{0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
$\widetilde{b}^T_{p,9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$
$a_{8,p}$
$a_{9,p}$
$a_{10,p}$
$a_{11,p}$

Figure 5.12: Broadcast a_p within rows of processors (before).

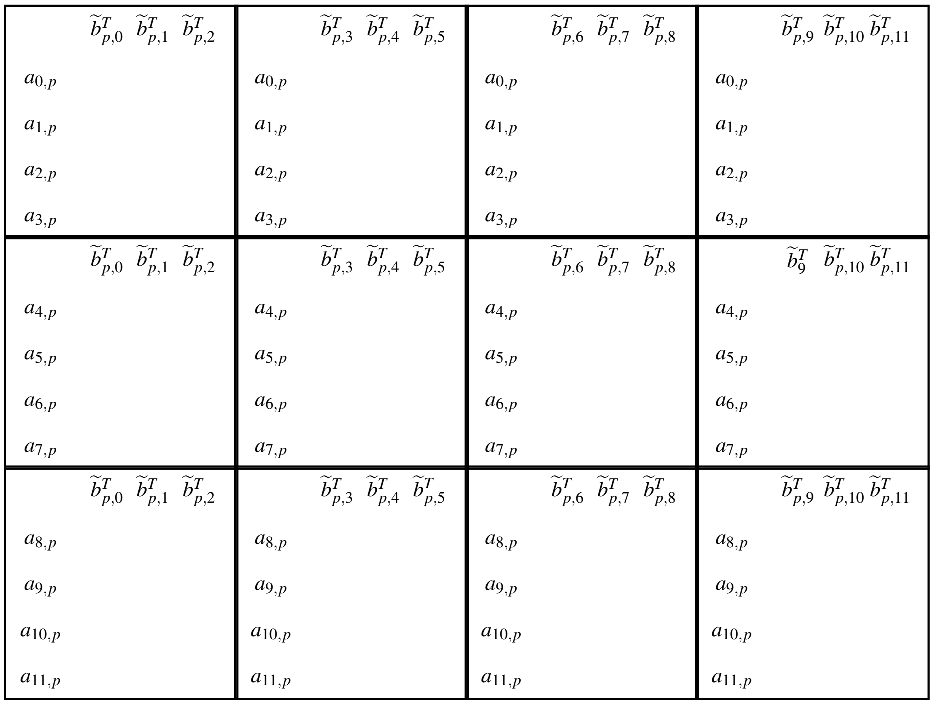

$\widetilde{b}^T_{p,0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
$a_{0,p}$ $a_{1,p}$ $a_{2,p}$ $a_{3,p}$
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
$a_{0,p}$ $a_{1,p}$ $a_{2,p}$ $a_{3,p}$
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
$a_{0,p}$ $a_{1,p}$ $a_{2,p}$ $a_{3,p}$
$\widetilde{b}^T_{p,9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$
$a_{0,p}$ $a_{1,p}$ $a_{2,p}$ $a_{3,p}$
$\widetilde{b}^T_{p,0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
$a_{4,p}$ $a_{5,p}$ $a_{6,p}$ $a_{7,p}$
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
$a_{4,p}$ $a_{5,p}$ $a_{6,p}$ $a_{7,p}$
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
$a_{4,p}$ $a_{5,p}$ $a_{6,p}$ $a_{7,p}$
$\widetilde{b}^T_{9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$
$a_{4,p}$ $a_{5,p}$ $a_{6,p}$ $a_{7,p}$
$\widetilde{b}^T_{p,0}$ $\widetilde{b}^T_{p,1}$ $\widetilde{b}^T_{p,2}$
$a_{8,p}$ $a_{9,p}$ $a_{10,p}$ $a_{11,p}$
$\widetilde{b}^T_{p,3}$ $\widetilde{b}^T_{p,4}$ $\widetilde{b}^T_{p,5}$
$a_{8,p}$ $a_{9,p}$ $a_{10,p}$ $a_{11,p}$
$\widetilde{b}^T_{p,6}$ $\widetilde{b}^T_{p,7}$ $\widetilde{b}^T_{p,8}$
$a_{8,p}$ $a_{9,p}$ $a_{10,p}$ $a_{11,p}$
$\widetilde{b}^T_{p,9}$ $\widetilde{b}^T_{p,10}$ $\widetilde{b}^T_{p,11}$
$a_{8,p}$ $a_{9,p}$ $a_{10,p}$ $a_{11,p}$

Figure 5.13: Broadcast a_p within rows of processors (after).

	$\widetilde{b}^T_{p,0}$	$\widetilde{b}^T_{p,1}$	$\widetilde{b}^T_{p,2}$		$\widetilde{b}^T_{p,3}$	$\widetilde{b}^T_{p,4}$	$\widetilde{b}^T_{p,5}$		$\widetilde{b}^T_{p,6}$	$\widetilde{b}^T_{p,7}$	$\widetilde{b}^T_{p,8}$		$\widetilde{b}^T_{p,9}$	$\widetilde{b}^T_{p,10}$	$\widetilde{b}^T_{p,11}$
$a_{0,p}$	$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	$a_{0,p}$	$C_{0,3}$	$C_{0,4}$	$C_{0,5}$	$a_{0,p}$	$C_{0,6}$	$C_{0,7}$	$C_{0,8}$	$a_{0,p}$	$C_{0,9}$	$C_{0,10}$	$C_{0,11}$
$a_{1,p}$	$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	$a_{1,p}$	$C_{1,3}$	$C_{1,4}$	$C_{1,5}$	$a_{1,p}$	$C_{1,6}$	$C_{1,7}$	$C_{1,8}$	$a_{1,p}$	$C_{1,9}$	$C_{1,10}$	$C_{1,11}$
$a_{2,p}$	$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	$a_{2,p}$	$C_{2,3}$	$C_{2,4}$	$C_{2,5}$	$a_{2,p}$	$C_{2,6}$	$C_{2,7}$	$C_{2,8}$	$a_{2,p}$	$C_{2,9}$	$C_{2,10}$	$C_{2,11}$
$a_{3,p}$	$C_{3,0}$	$C_{3,1}$	$C_{3,2}$	$a_{3,p}$	$C_{3,3}$	$C_{3,4}$	$C_{3,5}$	$a_{3,p}$	$C_{3,6}$	$C_{3,7}$	$C_{3,8}$	$a_{3,p}$	$C_{3,9}$	$C_{3,10}$	$C_{3,11}$
	$\widetilde{b}^T_{p,0}$	$\widetilde{b}^T_{p,1}$	$\widetilde{b}^T_{p,2}$		$\widetilde{b}^T_{p,3}$	$\widetilde{b}^T_{p,4}$	$\widetilde{b}^T_{p,5}$		$\widetilde{b}^T_{p,6}$	$\widetilde{b}^T_{p,7}$	$\widetilde{b}^T_{p,8}$		$\widetilde{b}^T_{9}$	$\widetilde{b}^T_{p,10}$	$\widetilde{b}^T_{p,11}$
$a_{4,p}$	$C_{4,0}$	$C_{4,1}$	$C_{4,2}$	$a_{4,p}$	$C_{4,3}$	$C_{4,4}$	$C_{4.5}$	$a_{4,p}$	$C_{4,6}$	$C_{4,7}$	$C_{4,8}$	$a_{4,p}$	$C_{4,9}$	$C_{4,10}$	$C_{4,11}$
$a_{5,p}$	$C_{5,0}$	$C_{5,1}$	$C_{5,2}$	$a_{5,p}$	$C_{5,3}$	$C_{5,4}$	$C_{5,5}$	$a_{5,p}$	$C_{5,6}$	$C_{5,7}$	$C_{5,8}$	$a_{5,p}$	$C_{5,9}$	$C_{5,10}$	$C_{5,11}$
$a_{6,p}$	$C_{6,0}$	$C_{6,1}$	$C_{6,2}$	$a_{6,p}$	$C_{6,3}$	$C_{6,4}$	$C_{6,5}$	$a_{6,p}$	$C_{6,6}$	$C_{6,7}$	$C_{6,8}$	$a_{6,p}$	$C_{6,9}$	$C_{6,10}$	$C_{6,11}$
$a_{7,p}$	$C_{7,0}$	$C_{7,1}$	$C_{7,2}$	$a_{7,p}$	$C_{7,3}$	$C_{7,4}$	$C_{7,5}$	$a_{7,p}$	$C_{7,6}$	$C_{7,7}$	$C_{7,8}$	$a_{7,p}$	$C_{7,9}$	$C_{7,10}$	$C_{7,11}$
	$\widetilde{b}^T_{p,0}$	$\widetilde{b}^T_{p,1}$	$\widetilde{b}^T_{p,2}$		$\widetilde{b}^T_{p,3}$	$\widetilde{b}^T_{p,4}$	$\widetilde{b}^T_{p,5}$		$\widetilde{b}^T_{p,6}$	$\widetilde{b}^T_{p,7}$	$\widetilde{b}^T_{p,8}$		$\widetilde{b}^T_{p,9}$	$\widetilde{b}^T_{p,10}$	$\widetilde{b}^T_{p,11}$
$a_{8,p}$	$C_{8,0}$	$C_{8,1}$	$C_{8,2}$	$a_{8,p}$	$C_{8,3}$	$C_{8,4}$	$C_{8,5}$	$a_{8,p}$	$C_{8,6}$	$C_{8,7}$	$C_{8,8}$	$a_{8,p}$	$C_{8,9}$	$C_{8,10}$	$C_{8,11}$
$a_{9,p}$	$C_{9,0}$	$C_{9,1}$	$C_{9,2}$	$a_{9,p}$	$C_{9,3}$	$C_{9,4}$	$C_{9,5}$	$a_{9,p}$	$C_{9,6}$	$C_{9,7}$	$C_{9,8}$	$a_{9,p}$	$C_{9,9}$	$C_{9,10}$	$C_{9,11}$
$a_{10,p}$	$C_{10,0}$	$C_{10,1}$	$C_{10,2}$	$a_{10,p}$	$C_{10,3}$	$C_{10,4}$	$C_{10,5}$	$a_{10,p}$	$C_{10,6}$	$C_{10,7}$	$C_{10,8}$	$a_{10,p}$	$C_{10,9}$	$C_{10,10}$	$C_{10,11}$
$a_{11,p}$	$C_{11,0}$	$C_{11,1}$	$C_{11,2}$	$a_{11,p}$	$C_{11,3}$	$C_{11,4}$	$C_{11,5}$	$a_{11,p}$	$C_{11,6}$	$C_{11,7}$	$C_{11,8}$	$a_{11,p}$	$C_{11,9}$	$C_{11,10}$	$C_{11,11}$

Figure 5.14: Local update.