Automatically Pipelining Loop Iterations on Distributed-Memory Machines

Carl D. Offner

Abstract

This paper really has two parts:

1. We start with an expository discussion of LU decomposition, including a motivation as well as a clean derivation of the standard algorithm.
2. Then we show how a clever HPF compiler could infer—simply from the HPF data mapping directives—that the outer loop can be pipelined. This inference is made possible by an extension of the ordinary notion of dependence to include what are here called *spatial dependences*.

1 Introduction

Parallel constructs can be expressed in Fortran 90 by means of array syntax, or more generally by use of the FORALL construct. Fortran 77 DO loops that are completely parallelizable can be identified by a compiler by the use of standard vectorization techniques, and so can be transformed internally into parallel form.

There are, however, significant computations that are partially but not completely parallelizable. They are expressed in terms of DO loops that contain code that cannot be written in terms of Fortran 90 parallel constructs, yet in which the loop iterations do not have to be executed strictly sequentially—one iteration may be able to begin execution before a previous one is finished. This pipelining of iterations of a DO loop has also been called *doacross* behavior.

The problem we address here is how the compiler can identify such potential for pipelining and generate good code for it.

Our machine model is a distributed-memory multi-processor. Our compiler will accept a source program written in Fortran 90/HPF—i.e., a single-threaded program with parallel constructs allowed—and will generate explicit SPMD code that will run on each processor. The generated code is parametrized by the processor number, and contains explicit message-passing. Such message-passing is not visible in the source code. Since the startup cost for a message is larger by several orders of magnitude than the cost of a local memory fetch, the most significant thing the compiler can do to generate good code is to generate messages with the right granularity. In general, this means that we want the compiler to generate messages that are as large as possible.

Our message-passing (i.e., interprocessor communication) is carried out by means of SEND/RECEIVE calls. The RECEIVE calls are blocking; that is, they do not return until the data has actually been received. This has an important consequence: No explicit synchronizations need to be introduced by the compiler. If a processor needs some non-local data, the RECEIVE will cause it to wait until the data is available. That in turn cannot happen until the corresponding SEND has occurred, and that will not occur until the sending processor has computed the data to be sent. So in particular, there is no need to introduce barrier synchronizations around subroutine calls, or anywhere else in the program.

Another way of looking at this is that there is no notion of universal time shared by all the processors—each processor in effect has its own local clock. Any necessary synchronization between two such local clocks is carried out automatically by the SEND/RECEIVE mechanism.

There are two tasks that have to be performed in order to generate good code for DO loops:

1. Generate communication with the correct granularity. A naive generation of code for a do loop would cause a separate message to be sent for each element needing to be moved. To the extent possible, we want to package up these communications so that between each two processors there is at most one message for each purely parallel construct. This packaging process is called *message vectorization*.

2. Place the communication correctly. In order to reduce the time that a processor has to wait when a RECEIVE is being executed, we need to place each SEND as early as possible. RECEIVE placement is more problematical, since RECEIVEs are blocking. Our current idea is to place each RECEIVE as late as possible. The correct placement of communication is what will implement pipelining in scheduling loop iterations.

To demonstrate the processing that the compiler needs to perform, we will use LU decomposition of a matrix as our main example. This is a typical example of a matrix computation used in numerical linear algebra which is amenable to pipelining the loop iterations.

2 The idea behind LU decomposition

2.1 Solving systems of linear equations

Say we have a system of n simultaneous linear equations in n variables:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= y_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= y_2 \\
a_{31}x_1 + a_{32}x_2 + \cdots + a_{3n}x_n &= y_3 \\
&\cdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= y_1
\end{aligned}$$

We can write this in matrix notation as

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix}$$

or more simply, as

$$\mathbf{A}\vec{x} = \vec{y}$$

Let us assume that the matrix $\mathbf{A}$ is non-singular so that we know that the system of equations has a unique solution $\vec{x}$ for each $\vec{y}$. Of course this solution can be represented formally as

$$\vec{x} = \mathbf{A}^{-1}\vec{y}$$

where $\mathbf{A}^{-1}$ is the inverse of the matrix $\mathbf{A}$. However, finding the inverse of a matrix is relatively expensive. It is more efficient, and in general more accurate, to find the solution $\vec{x}$ by the familiar technique of Gaussian elimination.

We will illustrate this technique with the equation

$$\begin{pmatrix} 3 & 1 & 6 \\ 2 & 1 & 3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \\ 4 \end{pmatrix} \tag{1}$$

or, writing this as a system of equations,

$$\begin{aligned} 3x_1 + x_2 + 6x_3 &= 2 \\ 2x_1 + x_2 + 3x_3 &= 7 \\ x_1 + x_2 + x_3 &= 4 \end{aligned}$$

2.2 Solving equations by eliminating variables

In high school, students are taught to solve systems of equations like these by "eliminating variables": First we eliminate the variable x_1 from the second and third equations by subtracting $2/3$ of the first equation from the second and $1/3$ of the first equation from the third:

$$\begin{aligned} 3x_1 + \phantom{\tfrac{1}{3}}x_2 + 6x_3 &= 2 \\ \tfrac{1}{3}x_2 - x_3 &= \tfrac{17}{3} \\ \tfrac{2}{3}x_2 - x_3 &= \tfrac{10}{3} \end{aligned}$$

(Of course a real high school student would not have done it precisely this way: she would more likely have subtracted multiples of the third equation from the first and second, because that way the computations are simpler—there are no fractions involved. However, we are not concerned here with how messy the numbers look—we are going to program a computer to do these computations—so we want a method that is *algorithmically* as simple as possible.)

At this stage, the second and third equations have become two equations in two variables. We can perform a similar procedure of eliminating the variable x_2 from the third equation by subtracting a multiple of the second equation from the third. In this case the multiple is 2, and we get

$$\begin{aligned} 3x_1 + \quad x_2 + 6x_3 &= 2 \\ \tfrac{1}{3}x_2 - \quad x_3 &= \tfrac{17}{3} \\ x_3 &= -8 \end{aligned}$$

The solution is now virtually at hand: The third equation has been reduced to 1 equation in 1 variable, which is easily solved—in this case, it is already solved:

$$x_3 = -8$$

We can then substitute the value of x_3 into the second equation, yielding

$$\tfrac{1}{3}x_2 + 8 = \tfrac{17}{3}$$

which gives us

$$x_2 = -7$$

Then -7 and -8 are substituted for x_2 and x_3 in the first equation, yielding

$$3x_1 - 7 - 48 = 2$$

and this produces

$$x_1 = 19$$

so we have arrived at the solution

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 19 \\ -7 \\ -8 \end{pmatrix}$$

The process of eliminating variables as we have done here is called *forward elimination*. The subsequent process of solving for x_3, then using its value to solve for x_2, and so on, is called *back substitution*. (The terms "forward" and "back" refer to the order in which we process the equations; from first to last, or from last to first.)

The reason we are able to perform back substitution is that the processing performed during forward elimination has changed the matrix of coefficients so that it is 0 under the main diagonal—we say that it has been transformed into an *upper-triangular* matrix. That is, solving a system of equations $\mathbf{T}\vec{x} = \vec{y}$ amounts to back substitution if $\mathbf{T}$ is upper-triangular. If $\mathbf{T}$ is lower-triangular, the solution would be carried out by means of *forward substitution*.

There is an assumption we have made in outlining this algorithm: we have assumed that at the j^{th} step, when we are about to subtract multiples of the j^{th} row from the succeeding rows, that the j^{th} coefficient in the j^{th} row is non-zero. (Otherwise, we would not be eliminating the terms in x_j from the succeeding rows.) Although this does not have to be true—for instance, it is not true for the nice non-singular matrix $\left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right)$—it is nevertheless quite often true, and there are commonly arising classes of matrices for which it is guaranteed to be true.

But beyond this, a real numerical analyst would not do precisely what we have done in any case: Rather than subtracting a multiple of the second equation from the third during forward elimination, the numerical analyst would be more likely to subtract a multiple of the third equation from the second. The reason for this is that the coefficient of x_2 in the third equation ($\frac{2}{3}$) is larger than that in the second equation ($\frac{1}{3}$), and choosing the equation with the largest coefficient of the variable to be eliminated generally leads to greater numerical stability. The coefficient thus chosen is called the *pivot element* of its column, and choosing a pivot element in this manner is called *partial pivoting*.[1] So a numerical analyst would choose the largest coefficient as a pivot element, while a high school student, working under different constraints, would probably choose the smallest coefficient as a pivot element. For our purposes, however, we will not perform pivoting at all; we will assume that our matrix of coefficients is sufficiently numerically stable that this is not needed.

This process of solving a system of linear equations by elimination of variables is usually called *Gaussian elimination*.

2.3 Elementary row operations

Now let us look at what we are doing a little more closely. The operations we are performing during forward elimination are composed of what are usually called *elementary row operations*. There are two such operations that we use:

- multiplying a row by a constant, and
- adding (or subtracting) one row from another row.

When we apply an elementary row operation to the matrix $\mathbf{A}$, we apply the same operation to the vector $\vec{y}$. To see what is going on, let us rewrite the matrix equation in the original form:

$$\begin{aligned} 3x_1 + x_2 + 6x_3 &= 2 \\ 2x_1 + x_2 + 3x_3 &= 7 \\ x_1 + x_2 + x_3 &= 4 \end{aligned}$$

Applying the elementary row operation of multiplying by 7 (say) to the first row of the matrix and to the first element of $\vec{y}$ (which in this case is 2) just multiplies the entire first row of the first equation by 7. Similarly, applying the elementary row operation of adding the first row of the matrix to the second (i.e., replacing the second row by the sum of the first and second rows), and also replacing y_2 by $y_1 + y_2$ (i.e., in this case, replacing the 7 on the right by 9) just amounts to adding the first equation elementwise to the second. In either case, equality is preserved: the transformed set of equations is satisfied by a vector $\vec{x}$ if and only if the original set of equations is satisfied.

In practice (as we saw above), we usually combine these row operations: we add or subtract a multiple of a row from another row.

[1] *Complete pivoting* would be similar except that at each step we would also choose the column to be processed, rather than automatically working on the next column.

The significant thing about these row operations is that they are naturally expressible as matrix transformations. For instance, say as before we start with

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix}$$

To make things come out most simply in the end, we are going to adopt the convention that we will *subtract* multiples of the first row from each other row in such a way that the whole first column under a_{11} becomes 0. Let us say that we multiply the first row by m_{j1} and then subtract it from the j^{th} row. Then we must have

$$m_{j1} = a_{j1}/a_{11}$$

Then the operation of multiplying the first row by all the m_{j1} multipliers and then subtracting from the appropriate rows can be represented by the following computation:

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ -m_{21} & 1 & 0 & \dots & 0 \\ -m_{31} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -m_{n1} & 0 & 0 & \dots & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ -m_{21} & 1 & 0 & \dots & 0 \\ -m_{31} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -m_{n1} & 0 & 0 & \dots & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix}$$

That is, letting $\mathbf{M}_1$ denote the matrix

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ -m_{21} & 1 & 0 & \dots & 0 \\ -m_{31} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -m_{n1} & 0 & 0 & \dots & 1 \end{pmatrix}$$

we have for the first step in forward elimination

$$\mathbf{M}_1\mathbf{A}\vec{x} = \mathbf{M}_1\vec{y}$$

The second step in forward elimination then consists in multiplying the second row *of the new matrix* $\mathbf{M}_1\mathbf{A}$ by appropriate multipliers and adding the results elementwise to rows $3 \dots n$. This can be represented by

$$\mathbf{M}_2\mathbf{M}_1\mathbf{A}\vec{x} = \mathbf{M}_2\mathbf{M}_1\vec{y}$$

where

$$\mathbf{M_2} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & -m_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & -m_{n2} & 0 & \dots & 1 \end{pmatrix}$$

and where $m_{j2} = a'_{j2}/a'_{22}$ (where the elements a'_{jk} are elements of the matrix $\mathbf{M}_1\mathbf{A}$).

Continuing in this fashion, we see that the entire process of forward elimination can be represented as follows:

$$\mathbf{M}_{n-1}\cdots\mathbf{M}_2\mathbf{M}_1\mathbf{A}\vec{x} = \mathbf{M}_{n-1}\cdots\mathbf{M}_2\mathbf{M}_1\vec{y}$$

Thus, writing $\mathbf{M} = \mathbf{M}_{n-1}\cdots\mathbf{M}_2\mathbf{M}_1$, we have transformed $\mathbf{A}$ into an upper-triangular matrix

$$\mathbf{U} = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & u_{nn} \end{pmatrix} = \mathbf{MA} \tag{2}$$

2.4 LU decomposition

Since all the matrices $\mathbf{M_j}$ are invertible, so is their product $\mathbf{M}$, and so from equation 2 we immediately derive the equivalent equation

$$\mathbf{M}^{-1}\mathbf{U} = \mathbf{A} \tag{3}$$

and so solving $\mathbf{A}\vec{x} = \vec{y}$ is equivalent to solving $\mathbf{M}^{-1}\mathbf{U}\vec{x} = \vec{y}$.

But this would be useful only if $\mathbf{M}^{-1}$ could be computed easily and if it turned out to have a form that was easy to compute with. Remarkably, both these hopes are fulfilled: $\mathbf{M}^{-1}$ is easy to compute, and it turns out to be a lower-triangular matrix, which we will denote below by $\mathbf{L}$.

To see this, we look more closely at the matrix product $\mathbf{M} = \mathbf{M}_{n-1}\cdots\mathbf{M}_2\mathbf{M}_1$. Let us write each matrix factor $\mathbf{M}_j$ as a difference of two matrices:

$$\mathbf{M}_j = \mathbf{I} - \boldsymbol{\mu}_j$$

where

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

is the identity matrix, and

$$\boldsymbol{\mu}_j = \begin{pmatrix} 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & 0 & \dots & 0 \\ 0 & \dots & m_{j,j+1} & \dots & 0 \\ 0 & \dots & m_{j,j+2} & \dots & 0 \\ \vdots & & \vdots & & \vdots \\ 0 & \dots & m_{j,n} & \dots & 0 \end{pmatrix}$$

is the matrix where every column is 0 except the j^{th} column, which is 0 except for the multipliers in rows $j+1$ to n.

We are going to regard $\mathbf{R}^n$ as a vector space and consider some of its subspaces:

Let us use the notation $[i \dots j]$ to refer to the subspace of $\mathbf{R}^n$ consisting of vectors whose components are restricted to be 0 for all dimensions other than those between i and j. Thus, (if $n = 5$, say), $[2 \dots 4]$ contains the vectors $(0, 1, 0, 3, 0)$ and $(0, 1, -11, 3, 0)$, but does not contain $(-5, 1, 0, 3, 0)$ or $(0, 1, 0, 3, 12)$.

If A and B are two disjoint subspaces of $\mathbf{R}^n$ (that is, their only element in common is $\vec{0}$), then $A \oplus B$ refers to the *span* of A and B—the vectors that are expressible as sums $\vec{a} + \vec{b}$ with $\vec{a} \in A$ and $\vec{b} \in B$. In three-dimensional "xyz-space" $\mathbf{R}^3$, for example, if A is the x-axis and B is the y-axis, then $A \oplus B$ is the xy-plane. If on the other hand B is the yz-plane, then $A \oplus B$ is the whole space $\mathbf{R}^3$.

We regard any $n \times n$ matrix $\mathbf{T}$ as a linear operator from $\mathbf{R}^n$ to $\mathbf{R}^n$, and we use the notations

$$\begin{aligned} \ker \mathbf{T} &= \text{the kernel of } \mathbf{T} = \text{the subspace of vectors } \vec{x} \text{ such that } \mathbf{T}\vec{x} = 0 \\ \operatorname{ran} \mathbf{T} &= \text{the range of } \mathbf{T} = \begin{array}{l}\text{the subspace of vectors } \vec{y} \text{ that are in the} \\ \text{image of } \mathbf{T}\text{; i.e., for which there exists a} \\ \text{vector } \vec{x} \text{ with } \vec{y} = \mathbf{T}\vec{x}\end{array} \end{aligned}$$

Everything we need will then follow from the following two inclusions, which follow immediately by looking at the form of the matrix $\boldsymbol{\mu}_j$:

$$\begin{aligned} \ker \boldsymbol{\mu}_j &\supset [1 \dots j{-}1] \oplus [j{+}1 \dots n] \\ \operatorname{ran} \boldsymbol{\mu}_j &\subset [j{+}1 \dots n] \end{aligned}$$

From these two inclusions it follows at once that for $i \le j$,

$$\operatorname{ran} \boldsymbol{\mu}_j \subset [j{+}1 \dots n] \subset [i{+}1 \dots n] \subset \ker \boldsymbol{\mu}_i$$

and hence

$$\boldsymbol{\mu}_i \boldsymbol{\mu}_j = 0 \qquad \text{for } i \le j \tag{4}$$

Therefore,

$$(\mathbf{I} + \boldsymbol{\mu}_j)(\mathbf{I} - \boldsymbol{\mu}_j) = \mathbf{I} - \boldsymbol{\mu}_j^2 = \mathbf{I}$$

and so $\mathbf{M}_j^{-1} = \mathbf{I} + \boldsymbol{\mu}_j$. That is, the inverse of the matrix

$$\begin{pmatrix} 1 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & 1 & \dots & 0 \\ 0 & \dots & -m_{j,j+1} & \dots & 0 \\ 0 & \dots & -m_{j,j+2} & \dots & 0 \\ \vdots & & \vdots & & \vdots \\ 0 & \dots & -m_{j,n} & \dots & 1 \end{pmatrix}$$

is the matrix

$$\begin{pmatrix} 1 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & 1 & \dots & 0 \\ 0 & \dots & m_{j,j+1} & \dots & 0 \\ 0 & \dots & m_{j,j+2} & \dots & 0 \\ \vdots & & \vdots & & \vdots \\ 0 & \dots & m_{j,n} & \dots & 1 \end{pmatrix}$$

This identity can be understood at an intuitive level—it just says that the operation of subtracting certain multiples of the j^{th} row from rows $j{+}1 \dots n$ has an inverse which is just the operation of *adding* the corresponding multiples of the j^{th} row to rows $j{+}1 \dots n$. So equation (4) is really not needed to see this identity.

On the other hand, we can now compute the inverse of the matrix $\mathbf{M}$; and for this, equation (4) is really helpful:

$$\begin{aligned} \mathbf{M}^{-1} &= \mathbf{M}_1^{-1}\mathbf{M}_2^{-1} \cdots \mathbf{M}_{n-1}^{-1} \\ &= (\mathbf{I} + \boldsymbol{\mu}_1)(\mathbf{I} + \boldsymbol{\mu}_2) \cdots (\mathbf{I} + \boldsymbol{\mu}_{n-1}) \\ &= \mathbf{I} + \boldsymbol{\mu}_1 + \boldsymbol{\mu}_2 + \cdots + \boldsymbol{\mu}_{n-1} + \text{products of the form } \boldsymbol{\mu}_{i_1}\boldsymbol{\mu}_{i_2} \dots \boldsymbol{\mu}_{i_j} \end{aligned}$$

where $i_1 < i_2 < \cdots < i_j$. By equation (4), each such product $\boldsymbol{\mu}_{i_1}\boldsymbol{\mu}_{i_2} \dots \boldsymbol{\mu}_{i_j}$ is zero. Thus, we have

$$\mathbf{M}^{-1} = \mathbf{I} + \boldsymbol{\mu}_1 + \boldsymbol{\mu}_2 + \cdots + \boldsymbol{\mu}_{n-1}$$

In other words,

$$\mathbf{M}^{-1} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ m_{21} & 1 & 0 & \dots & 0 \\ m_{31} & m_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & \dots & 1 \end{pmatrix}$$

and we have shown that the matrix $\mathbf{M}^{-1}$ is just the matrix of multipliers m_{ij} added to the identity matrix.

Thus, $\mathbf{M}^{-1}$ is a lower triangular matrix which we will refer to as $\mathbf{L}$. Since $\mathbf{MA} = \mathbf{U}$, we have $\mathbf{A} = \mathbf{M}^{-1}\mathbf{U} = \mathbf{LU}$, and we see that we have factored the matrix $\mathbf{A}$ into a product of a lower-triangular and an upper-triangular matrix. This factorization is known as the *LU-decomposition* of $\mathbf{A}$.

Thus, the solution of the matrix equation $\mathbf{A}\vec{x} = \vec{y}$ can be carried out as follows:

1. Compute the LU-decomposition of $\mathbf{A}$.
2. Solve $\mathbf{L}\vec{b} = \vec{y}$ by forward substitution.
3. Solve $\mathbf{U}\vec{x} = \vec{b}$ by back substitution.

If we have to solve the equation $\mathbf{A}\vec{x} = \vec{y}$ for a number of different vectors $\vec{y}$, there is an advantage to performing the computation in this manner: the LU decomposition only has to be computed once, and since the diagonal elements of $\mathbf{L}$ are all 1, the forward substitution step involves no division operations, as it would if we performed forward elimination each time starting with the matrix $\mathbf{A}$.

2.5 The example again, worked out

Let us consider again our original example (1):

$$\begin{pmatrix} 3 & 1 & 6 \\ 2 & 1 & 3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \\ 4 \end{pmatrix}$$

and let us see how the computations described above proceed. The matrix $\mathbf{A}$ is $\left(\begin{smallmatrix} 3 & 1 & 6 \\ 2 & 1 & 3 \\ 1 & 1 & 1 \end{smallmatrix}\right)$. The multipliers in the first column are $2/3$ and $1/3$, and so we have

$$\begin{array}{cccc} \mathbf{M_1} & \mathbf{A} & & \mathbf{M_1A} \\ \begin{pmatrix} 1 & 0 & 0 \\ -\frac{2}{3} & 1 & 0 \\ -\frac{1}{3} & 0 & 1 \end{pmatrix} & \begin{pmatrix} 3 & 1 & 6 \\ 2 & 1 & 3 \\ 1 & 1 & 1 \end{pmatrix} & = & \begin{pmatrix} 3 & 1 & 6 \\ 0 & \frac{1}{3} & -1 \\ 0 & \frac{2}{3} & -1 \end{pmatrix} \end{array}$$

Then $\mathbf{M_1A}$ has one multiplier (corresponding to the third element in the second column); it is 2. This give us

$$\begin{array}{cccc} \mathbf{M_2} & \mathbf{M_1A} & & \mathbf{M_2M_1A} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix} & \begin{pmatrix} 3 & 1 & 6 \\ 0 & \frac{1}{3} & -1 \\ 0 & \frac{2}{3} & -1 \end{pmatrix} & = & \begin{pmatrix} 3 & 1 & 6 \\ 0 & \frac{1}{3} & -1 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

Thus, we have

$$\mathbf{L} = (\mathbf{M_2M_1})^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & 2 & 1 \end{pmatrix}$$

(Remember that we don't have to compute this—we know that $\mathbf{L}$ is lower-triangular with all its diagonal elements being 1, and its elements below the diagonal being just the multipliers which we already discovered.) And we also have

$$\mathbf{U} = \mathbf{M_2 M_1 A} = \begin{pmatrix} 3 & 1 & 6 \\ 0 & \frac{1}{3} & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

and it is easy to check (if we really want to) that

$$\mathbf{LU} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & 2 & 1 \end{pmatrix} \begin{pmatrix} 3 & 1 & 6 \\ 0 & \frac{1}{3} & -1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 1 & 6 \\ 2 & 1 & 3 \\ 1 & 1 & 1 \end{pmatrix} = \mathbf{A}$$

Now recall that we want to solve $\mathbf{LU}\vec{x} = \vec{y}$ for $\vec{x}$, where $\vec{y} = \left(\begin{smallmatrix} 2 \\ 7 \\ 4 \end{smallmatrix}\right)$. This is equivalent to first solving

$$\mathbf{L}\vec{b} = \vec{y} \tag{5}$$

for $\vec{b}$, and then solving

$$\mathbf{U}\vec{x} = \vec{b} \tag{6}$$

for $\vec{x}$.

So let us do this. Equation (5) is

$$\begin{pmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & 2 & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \\ 4 \end{pmatrix}$$

And performing a forward elimination on this system of equations (that is, solving them from the top down), we get successively

$$\begin{aligned} b_1 &= 2 & b_1 &= 2 & b_1 &= 2 \\ \tfrac{2}{3} \cdot b_1 + b_2 &= 7 & \tfrac{2}{3} \cdot 2 + b_2 &= 7 & b_2 &= \tfrac{17}{3} \\ \tfrac{1}{3} \cdot b_1 + 2 \cdot b_2 + b_3 &= 4 & \tfrac{1}{3} \cdot 2 + 2 \cdot \tfrac{17}{3} + b_3 &= 4 & b_3 &= -8 \end{aligned}$$

So $\vec{b} = \left(\begin{smallmatrix} 2 \\ \frac{17}{3} \\ -8 \end{smallmatrix}\right)$. Using this value for $\vec{b}$, the second equation (6) is then

$$\begin{pmatrix} 3 & 1 & 6 \\ 0 & \frac{1}{3} & -1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ \frac{17}{3} \\ -8 \end{pmatrix}$$

Performing back substitution on this system of equations (that is, solving them from the bottom up), we get successively

$$\begin{aligned} x_3 &= -8 & x_3 &= -8 & x_3 &= -8 \\ \tfrac{1}{3} \cdot x_2 - x_3 &= \tfrac{17}{3} & \tfrac{1}{3} \cdot x_2 + 8 &= \tfrac{17}{3} & x_2 &= -7 \\ 3 \cdot x_1 + x_2 + 6 \cdot x_3 &= 2 & 3 \cdot x_1 - 7 - 48 &= 2 & x_1 &= 19 \end{aligned}$$

So we finally have our solution $\vec{x} = \left(\begin{smallmatrix}-8\\-7\\19\end{smallmatrix}\right)$. That is,

$$\begin{pmatrix}3 & 1 & 6\\2 & 1 & 3\\1 & 1 & 1\end{pmatrix}\begin{pmatrix}-8\\-7\\19\end{pmatrix} = \begin{pmatrix}2\\7\\4\end{pmatrix}$$

3 Computing LU decompositions

3.1 Computing an LU decomposition in place

Now we look at how to compute an LU decomposition in such a way as to use as little of a computer's memory as possible. For very large arrays, this can be significant.

First, since we know a priori that the diagonal elements of **L** are all 1, we do not need to store them in memory. Therefore, **L** and **U** can really be stored together in the same $n \times n$ array in memory—the non-zero elements of **U** will be the elements on and above the diagonal, the diagonal elements of **L**, which are all 1, will not appear, and the remainder of the non-zero elements of **L** will be the elements below the diagonal. In other words, the matrix we compute looks like this:

$$\begin{pmatrix}u_{11} & u_{12} & u_{13} & \dots & u_{1n}\\ m_{21} & u_{22} & u_{23} & \dots & u_{2n}\\ m_{31} & m_{32} & u_{33} & \dots & u_{3n}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ m_{n1} & m_{m2} & m_{n3} & \dots & u_{nn}\end{pmatrix}$$

To see how this can be done, let us write some Fortran code to compute **L** and **U**. Remember that **U** is actually computed by transforming the elements of **A** in place, so as we transform the elements of **A** we are actually computing **U**. The computation is shown in Figure 1.

```
do k = 1, n
   do i = k+1, n
      l(i,k) = a(i,k)/a(k,k)
   end do
   do i = k+1, n
      do j = k+1, n
         a(i,j) = a(i,j) - l(i,k) * a(k,j)
      end do
   end do
end do
```

Figure 1: Computation of **L** and **U**, expressed in Fortran

Next, we notice that once the value $l(i,k)$ has been computed, the corresponding value $a(i,k)$ is never needed again. Hence the value $l(i,k)$ can be stored in the location holding $a(i,k)$. That is, instead of assigning to $l(i,k)$, we just assign to $a(i,k)$ and use the value of $a(i,k)$ where before we would have used $l(i,k)$. In this way **A** is overwritten by the new matrix consisting of **L** and **U**; the computation becomes that of Figure 2.

```
do k = 1, n
   do i = k+1, n
      a(i,k) = a(i,k)/a(k,k)
   end do
   do i = k+1, n
      do j = k+1, n
         a(i,j) = a(i,j) - a(i,k) * a(k,j)
      end do
   end do
end do
```

Figure 2: LU decomposition in place, expressed in Fortran

3.2 Introducing parallelism

Parallelism is a large subject, and we only touch on it here. We just want to point out that this computation exhibits some big opportunities for parallel execution:

The two loop nests rooted at "**do** i" can each be executed in parallel—that is, each iteration of each loop nest is independent of every other iteration of that loop nest, and so the iterations of each loop nest can be executed in any order, or all at once if there are enough computing resources available. (This is something a compiler can see by looking at the dependence graph.) Thus these loop nests could be written as explicitly parallel constructs, as in Figure 3. Note that although a FORALL construct looks syntactically similar to a DO loop nest, its meaning is quite different—it is a parallel construct which must be computed as if the entire right-hand side of the assignment is evaluated before any assignment is made to the left-hand side.

```
do k = 1, n
   a(k+1:n, k) = a(k+1:n, k)/a(k, k)
   forall (i = k+1:n, j = k+1:n)
      a(i,j) = a(i,j) - a(i,k) * a(k,j)
   end forall
end do
```

Figure 3: LU decomposition in Fortran 90, using parallel constructions

The computations in the k^{th} iteration of the outer loop are illustrated in Figure 4.

At the k^{th} iteration, the lightly shaded column of elements below $a(k,k)$ is normalized by dividing each element by $a(k,k)$. This is the operation performed by the first vector statement in the k loop. Next, the darker shaded square of elements below and to the right of $a(k,k)$ is updated by taking each element $a(i,j)$ and subtracting the product of the elements $a(i,k)$ and $a(k,j)$.

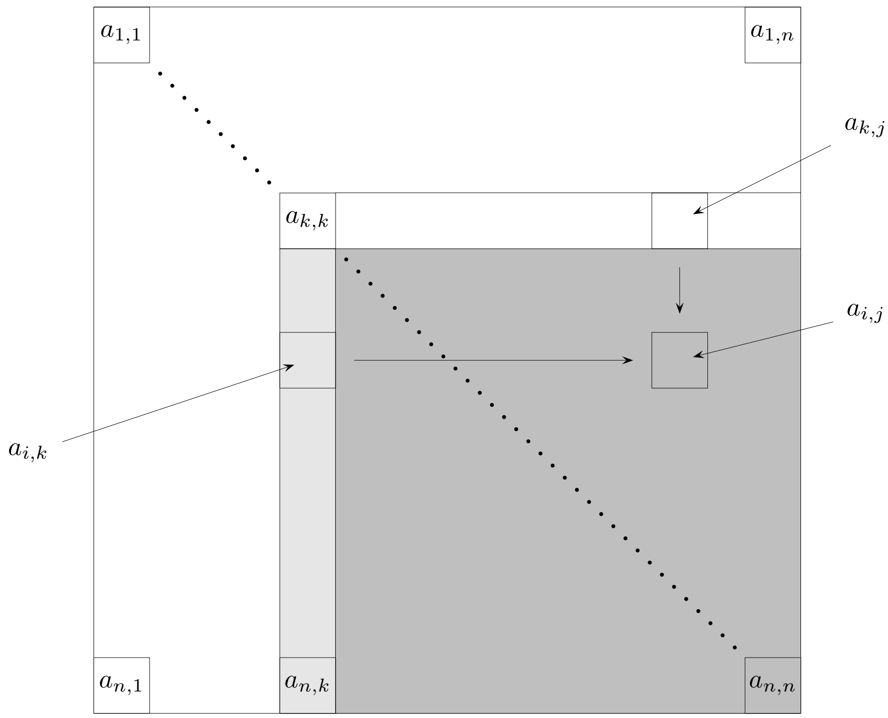

Figure 4: Computations in the k^{th} iteration of the outer loop.

4 Data Distributions

Of the two parallel constructs in Figure 3, the FORALL accesses data symmetrically across both dimensions of the matrix. The vector assignment, on the other hand, accesses data down a single column of the matrix. For this reason, it seems most efficient to allocate the data so that each column lives entirely on one processor—that way, the vector assignment can be carried out as a purely local pipelined computation. Parallelism will come about by distributing the columns over distinct processors.

4.1 A Very Simple Case

Suppose first that we have as many processors as there are columns in the matrix. Then we can see that the most efficient way to perform the computation is as in Figure 5, where the code is written in explicit SPMD form—that is, it is parametrized by the processor number and contains explicit message-passing.

In this figure, each processor receives in order the previous normalized columns under the elements $a_{k,k}$ (letting k run from 1 to myproc−1), and performs the update corresponding to the FORALL computation for all its elements as appropriate. When it has finished doing this, it normalizes its

```
do k = 1, myproc−1
    receive a(k+1:n, k)
    a(k+1:n, myproc) = a(k+1:n, myproc) − a(k+1:n, k) ∗ a(k, myproc)
end do

a(myproc+1:n, myproc) = a(myproc+1:n, myproc)/a(myproc, myproc)
send a(myproc+1:n, myproc) to processors myproc+1:n
```

Figure 5: Ideal code with 1 column per processor

column and broadcasts it to the processors holding the following columns.

Note that we are *not* distributing iterations of the k loop over the processors. In fact, the processor holding column myproc executes iterations $1 \ldots$ myproc of the original k loop.

Figure 6 shows the code a present-day compiler might generate. It is very close to the ideal code. The only real difference is that in effect k is allowed to run from 1 all the way to n, and guards have to be inserted to restrict the iterations on which computations take place. The difference in time between this code and the ideal code in Figure 5 is probably not even measurable.

```
do k = 1, n−1
    if k = myproc
        a(k+1:n, k) = a(k+1:n, k)/a(k, k)
    end if

    if k = myproc
        send a(k+1:n, k) to processors myproc+1:n
    end if

    if k < myproc
        receive a(k+1:n, k)
    end if

    if k < myproc
        a(k+1:n, myproc) = a(k+1:n, myproc) − a(k+1:n, k) ∗ a(k, myproc)
    end if
end do
```

Figure 6: Actual code with 1 column per processor

4.2 Practical Distributions

In practice, there are many more columns than there are processors, and each processor must hold a large number of columns. The manner in which the columns are allocated to the processors can have a noticeable effect on the efficiency of the generated code.

Consider first the case when the columns are allocated to the processors in a BLOCK fashion. This just means that (if there are p processors) the first block of n/p columns is allocated to the first processor, the second block of n/p columns is allocated to the second processor, and so on. This is illustrated in Figure 7, and is expressed by the HPF directive

!hpf$ distribute $(*, \textbf{block})$ :: a

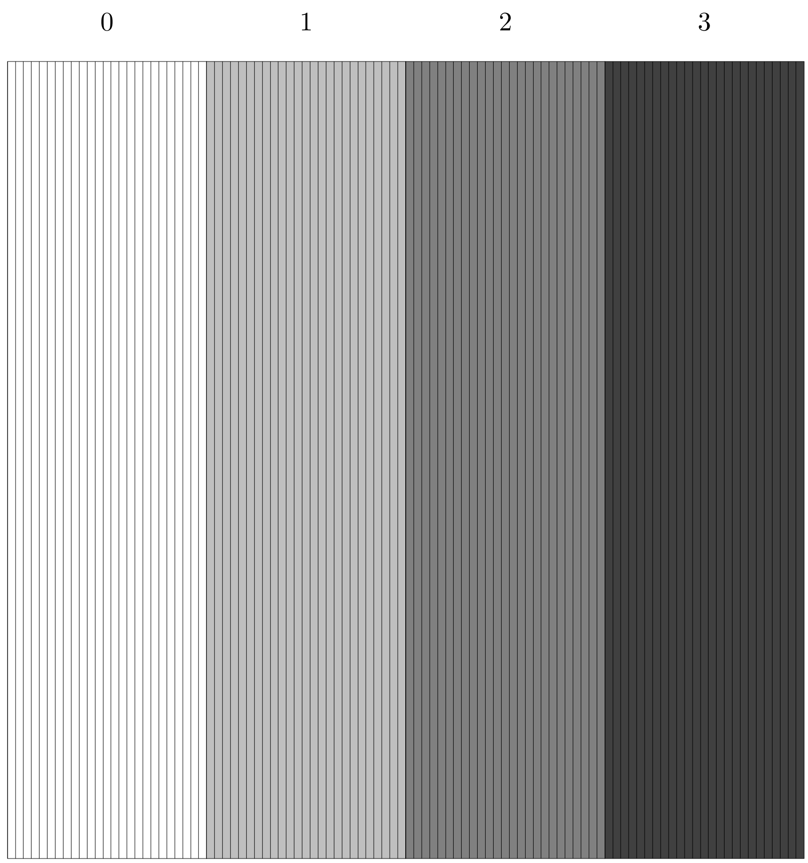

Figure 7: A BLOCK distribution of the columns. The processors owning different blocks of columns of the array are indicated by different shadings. The processor numbers are at the top. The array is 100×100, and each processor owns 25 consecutive columns.

Although BLOCK distributions are widely used, in this case such a distribution of a is inefficient for load balancing reasons—when the computation is halfway through the outermost k loop, half the processors will be completely finished, and will have no work to do, while the remaining processors are left to finish up the computation on their own.

For this reason, a CYCLIC distribution gives noticeably better results. This distribution is illustrated in Figure 8 and is described in HPF as follows:

!hpf$ distribute $(*, \textbf{cyclic})$ :: a

This directive causes the columns to be distributed cyclically to the processors, much as one might deal out a pack of cards: the first column goes on processor 1, the second on processor 2, and so on until all the processors have been given one column, after which the process starts over again at

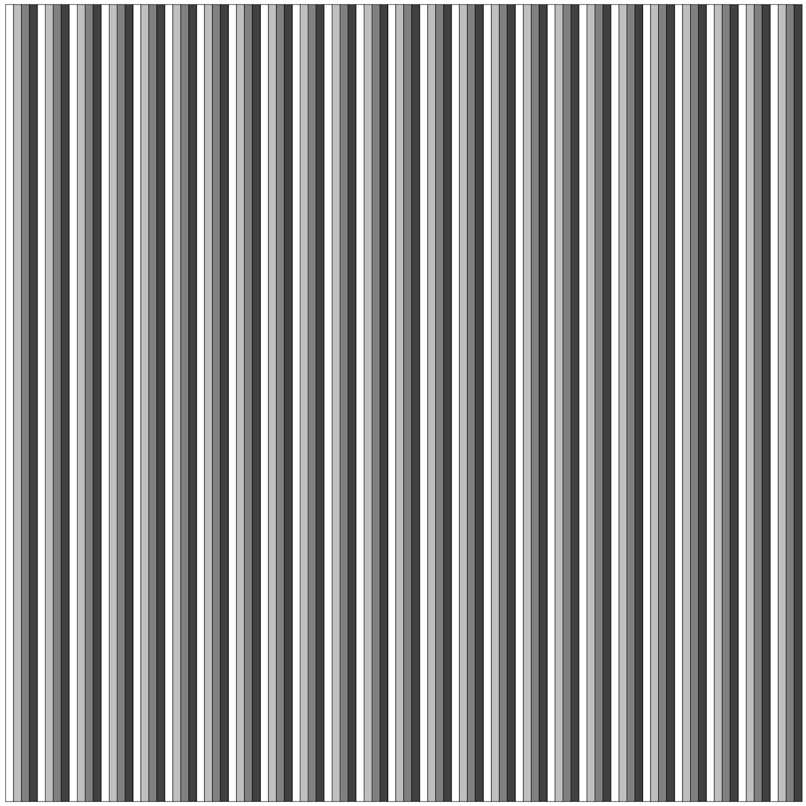

Figure 8: A CYCLIC distribution of the columns. Again, the different shadings indicate the different processors. The array is 100×100, and the columns are distributed over the processors as a deck of cards is dealt out.

processor 1 and keeps on going in this fashion until all the columns have been dealt out.

Until the very final few iterations of the k loop, such a CYCLIC distribution keeps all the processors busy, and so the entire computation finishes somewhat earlier than with a BLOCK distribution.

As in the example above, a present-day implementation will typically handle the communication for these cases in a virtually optimal fashion.

The weakness of these distributions is that they cause too much communication to be generated—each time a column is normalized, it is sent to the processors holding the later columns. It is better if we can block these messages into somewhat larger chunks. To do this, we distribute the columns of a in what is often called a "block-cyclic" manner. In HPF, this is called a CYCLIC(b) distribution. It is illustrated in Figure 9, and is specified like this:

!hpf$ distribute $(*, \textbf{cyclic}(b)) :: a$

The way this works is that the first b columns are allocated to the first processor, the next b columns to the second processor, and so on. So far this is similar to a BLOCK distribution, But in general, b

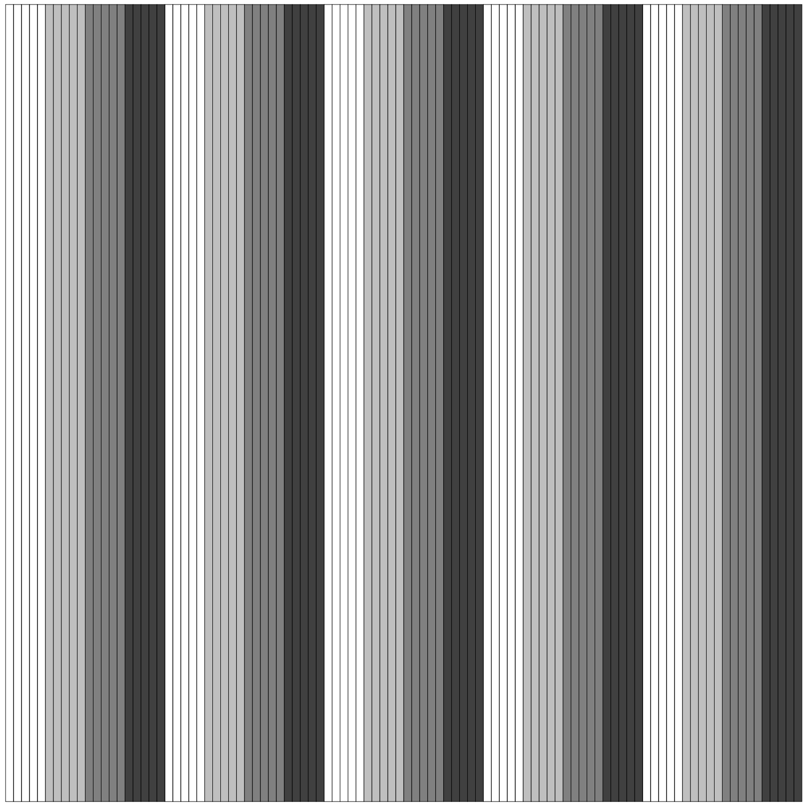

Figure 9: A CYCLIC(5) distribution of the columns. Again, the different shadings indicate the different processors. The array is 100×100; consecutive groups of 5 columns are dealt out cyclically, like a deck of cards, to the processors.

is much smaller that n/p, and so the process wraps around, just as in a CYCLIC distribution—after a block of b columns is allocated to processor p, the next block of b columns is allocated to processor 1 again, and so on. So the columns are divided up into consecutive blocks of b columns, and those blocks are allocated cyclically to the processors.

Now suppose the programmer writes the source code in Figure 10, with the intent that the compiler should generate the code to perform the computations as in Figure 11. (Of course, it would be preferable to have the user simply write the code in Figure 3 or Figure 2, and have the compiler discover the blocking of the outer k loop. We will address this matter at the end of this paper.)

In the comments in Figure 11, we refer to each block of b columns as a "thick column", or more simply, as a "block". num_blocks is the number of these thick columns; in this case, it is 20. $lo(k)$ and $hi(k)$ are just the first and last actual column numbers corresponding to the thick column k. In this case, $lo(k) = 5 * k - 4$ and $hi(k) = 5 * k$.

The idea of the computation is this: if a processor owns the thick column being processed—this

$$
\begin{array}{l}
\textbf{!hpf\$ distribute } (*, \textbf{cyclic}(b)) :: a \\
\textbf{do } k = 1, \text{num\_blocks} \\
\quad \textbf{do } kk = lo(k), hi(k) \\
\quad\quad a(kk{+}1{:}n,\ kk) = a(kk{+}1{:}n,\ kk)/a(kk,\ kk) \\
\quad\quad \textbf{forall } (i = kk{+}1{:}n,\ j = kk{+}1{:}n) \\
\quad\quad\quad a(i,\ j) = a(i,\ j) - a(i,\ kk) * a(kk,\ j) \\
\quad\quad \textbf{end forall} \\
\quad \textbf{end do} \\
\textbf{end do}
\end{array}
$$

Figure 10: Columns distributed CYCLIC(b): source code

is determined by the iteration of the k loop—then it successively normalizes each column in that thick column (this is what happens in the first kk loop) and updates *only the entries to the right of that column that are in the thick column being normalized*. Then it sends the thick column—which now contains the multipliers—to all remaining processors that need it, following which it updates the remaining entries to the right of the thick column being processed. (These remaining entries lie in other thick columns also owned by that processor.) Breaking the update processing into two separated computations in this way is what enables subsequent processors to get started on their updating for iteration k before the processor owning the thick column is finished its updating. This is the loop iteration pipelining we are interested in generating.

Since each processor owns a subset of the thick columns, it performs the updating indicated by the last FORALL only for those values of $a(i,\ j)$ which lie in those thick columns. That is, each processor executes only a subset of the FORALL iterations—this is what is meant by the comment in Figure 11 about "localized bounds". The compiler generates the loops that run over these localized iteration ranges automatically. The bounds of the compiler-generated loops are expressions involving my_processor(), and so they have the effect of having each processor execute only those iterations that belong to it. For simplicity, we will not write out these compiler-generated loops in what follows, since the expressions become messy and do not bear on the issue at hand.

A common present-day implementation would actually generate the code in Figure 12. This code is correct but not good: we are sending each column separately and we are not recognizing the pipelining possibilities in this code. We have to find ways of resolving these two problems.

5 Vectorizing the Communication

Our first task is to find a way to have the compiler block the communication so that it occurs once per k iteration, rather than on each kk iteration.

We can use ordinary vectorization techniques to do this blocking, as follows: We identify the following dependences[2] and their corresponding direction vectors with respect to the (k, kk) loops:

[2]Actually, there is one more dependence. Nevertheless, the analysis presented here is correct. We'll give the complete explanation in the next section.

do $k = 1$, num_blocks

    ! First do the "thick column" (all on one processor). This normalizes the
    ! column and updates the following entries in each row in the thick column.

    if I hold block k
        do $kk = lo(k), hi(k)$
            $a(kk{+}1\!:\!n,\ kk) = a(kk{+}1\!:\!n,\ kk)/a(kk,\ kk)$
            forall $(i = kk{+}1\!:\!n,\ j = kk{+}1\!:\!hi(k))$
                $a(i,\ j) = a(i,\ j) - a(i,\ kk) * a(kk,\ j)$
            end forall
        end do
    end if

    ! Perform the communications:

    if I hold block k
        send $a(lo(k)\!:\!n,\ lo(k)\!:\!hi(k))$ to processors holding blocks
            $k{+}1$:num_blocks
    end if
    if I am one of the processors holding a block in $k{+}1$:num_blocks
        receive $a(lo(k)\!:\!n,\ lo(k)\!:\!hi(k))$
    end if

    ! Now do the "interior square". This updates the rest of the entries to the
    ! right of the "thick column".

    do $kk = lo(k), hi(k)$
        forall $(i = kk{+}1\!:\!n,\ j = lo(k{+}1)\!:\!n)$! with localized bounds for j
            $a(i,\ j) = a(i,\ j) - a(i,\ kk) * a(kk,\ j)$
        end forall
    end do

end do

Figure 11: Columns distributed CYCLIC(b): Intended order of computation

$$
\begin{array}{ll}
N \to S & (=,=) \\
N \to U & (=,=) \\
R \to U & (=,=) \\
U \to N & (<,<) \\
U \to N & (=,<) \\
U \to U & (<,<) \\
U \to U & (=,<)
\end{array}
$$

```
    do k = 1, num_blocks
        do kk = lo(k), hi(k)
            if a(:, kk) is local    ! "I hold column kk"
N:              a(kk+1:n, kk) = a(kk+1:n, kk)/a(kk, kk)
            end if

            if a(:, kk) is local    ! "I hold column kk"
S:              send a(kk+1:n, kk) to remote processors holding columns kk+1:n.
            end if

            ! Actually, no guard is generated for the following receive because the
            ! generated do loop bounds enforce the necessary constraints.

            if a(:, kk) is not local and I hold one of the columns kk+1:n
R:              receive a(kk+1:n, kk)
            end if

            forall (i = kk+1:n, j = kk+1:n)
U:              a(i, j) = a(i, j) - a(i, kk) * a(kk, j)
            end forall
        end do

    end do
```

Figure 12: Columns distributed CYCLIC(b): This is what a typical compiler might generate now. The (non-Fortran) labels are used in the article to identify the four statements: Normalize, Send, Receive, and Update.

This gives rise to the dependence graph shown in Figure 13.

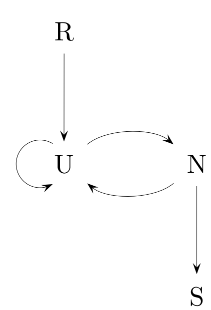

Figure 13: Dependence graph for the statements in Figure 12.

The only strongly connected region in this graph is the region composed of the two nodes N and U. Thus, the kk loop can be distributed over the three regions $\{R\}$, $\{N, U\}$, and $\{S\}$ (in that order, gotten by topologically sorting the strongly connected regions). In this way we get the code in Figure 14.

```
    do k = 1, num_blocks
       ! The next kk loop is parallel. This is key for what follows.
       do kk = lo(k), hi(k)
          ! Actually, no guard is generated for the following receive because the
          ! generated do loop bounds enforce the necessary constraints.
          if a(:, kk) is not local and I hold one of the columns kk+1:n
R:           receive a(kk+1:n, kk)
          end if
       end do
       do kk = lo(k), hi(k)
          if a(:, kk) is local    ! "I hold column kk"
N:           a(kk+1:n, kk) = a(kk+1:n, kk)/a(kk, kk)
          end if
          forall (i = kk+1:n, j = kk+1:n)
U:           a(i, j) = a(i, j) - a(i, kk) * a(kk, j)
          end forall
       end do
       do kk = lo(k), hi(k)
          if a(:, kk) is local    ! "I hold column kk"
S:           send a(kk+1:n, kk) to remote processors holding columns kk+1:n.
          end if
       end do
    end do
```

Figure 14: The kk loop has been distributed.

Now we can use the fact that the SEND and RECEIVE statements lie in parallel loop nests. They can therefore be vectorized, and we get the code in Figure 15.

6 The Missing Dependence

There are two problems with the analysis in the last section, which we clear up now:

- If we can distribute the kk loop over the three regions, what is stopping us from also distributing the k loop over those regions? On the other hand, if we did, the generated code would clearly be incorrect—a RECEIVE would be generated before anything was sent. In fact, the k loop is inherently sequential. Our analysis has to reflect this fact.

- There is a dependence, or at any rate something that seems like a dependence, that we left

```
    do k = 1, num_blocks

       ! Actually, no guard is generated for the following receive because the
       ! generated do loop bounds enforce the necessary constraints.

       if a(:, lo(k)) is not local and I hold one of the columns lo(k+1):n
R:        receive a(:, lo(k):hi(k))
       end if

       do kk = lo(k), hi(k)

          if a(:, kk) is local    ! "I hold column kk"
N:           a(kk+1:n, kk) = a(kk+1:n, kk)/a(kk, kk)
          end if

          forall (i = kk+1:n, j = kk+1:n)
U:           a(i, j) = a(i, j) − a(i, kk) ∗ a(kk, j)
          end forall
       end do

       if a(:, lo(k)) is local    ! "I hold column lo(k)"
S:        send a(:, lo(k):hi(k)) to remote processors holding columns lo(k+1):n.
       end if

    end do
```

Figure 15: The messages have been vectorized.

out—from the SEND to the RECEIVE:

$$S \to R \qquad (=,=)$$

Including this dependence in the graph of Figure 13 would force all the nodes in that graph to be in one strongly connected region, and would inhibit *any* loop distribution. And yet we know that the kk loop can in fact be distributed.

Both of these problems can be resolved by looking closer at this new dependence.

There are several kinds of dependences. Each dependence corresponds to a kind of constraint that must be honored by a program transformation if the semantics of the program is to be preserved:

6.1 Data Dependences

Data dependences fall into two classes:

loop independent dependences These are dependences relating data accessed on the same iteration of a loop.

loop carried dependences These are dependences relating data accessed on different iterations of a loop.

In either case, the two data references in a data dependence are accessed at different times: in a loop carried dependence, they are accessed in different iterations of a loop, and in a loop independent dependence, they are accessed at different times in the same iteration of a loop. In both cases, the data dependence specifies a constraint that must be honored by any program transformation if the semantics of the program is to be preserved, the constraint being that the two data references continue to be accessed in the same order. So we can think of data dependences as relating data that is *temporally separated*—they relate data at the same location but accessed at different times.

If a data dependence is ignored (say by a vectorizer) and parallel code is erroneously produced, the code will still execute, but will in general yield the wrong results. This is because failing to enforce a data dependence amounts to changing the order in which a memory location is accessed by two statements, or by one statement on two different iterations.

6.2 Spatial Dependences

The SEND/RECEIVE dependence that we have just found is a dependence of a different sort. We will refer to it as a *spatial dependence*, because it relates data accessed on *different* processors. In our SEND/RECEIVE model, spatial dependences are trivial to identify—they occur only between corresponding SEND and RECEIVE nodes, and every pair of such nodes has a spatial dependence associated with it.

Spatial dependences are rather different in nature from data dependences: A spatial dependence, in and of itself, does not place a constraint on program transformations. This is because a spatial dependence is just an assertion that a SEND must execute before its corresponding RECEIVE. Now if a RECEIVE is posted by a processor, it will simply wait until its message arrives—the corresponding SEND (on a different processor) may in fact be posted later. Thus, spatial dependences are in some sense self-enforcing. Another way to think about this is that a spatial dependence says that a reference on one processor must occur before a reference on another processor. But processors share no common clock; the mechanism that enforces this is just the SEND/RECEIVE mechanism, and not any other property of the program.

However, there is one case in which spatial dependences contribute to a constraint that must be honored by any program transformation: if a path composed of ordinary and spatial dependences starts and ends *on the same processor*, then it really does mandate an order of execution on that processor. If this order is not maintained in the generated code, then the code will not execute at all—execution will be stalled. This is because we would be generating on a single processor a RECEIVE that cannot return until a later SEND on the same processor is executed. (Specifically: a path in the dependence graph that leaves one processor and then returns to it contains a SEND followed by a later RECEIVE. If the RECEIVE is scheduled *before* the SEND, execution will stall.)

An example of this occurs in our LU decomposition: if we ignore the spatial dependence $S \to R$ and distribute the k loop, then all the RECEIVEs will be scheduled at the start of computation, and execution will stall immediately.

6.3 The Augmented Dependence Graph

To analyse spatial dependences correctly, we really have to augment the dependence graph so that the set of nodes is not just the set {statements}, but is the product set {statements} $\times$ {processors}. To put it another way, we need a separate dependence graph for each processor. Ordinary dependences are arcs between statements on the same processor (and refer either to the same or different iterations in the iteration space), while spatial dependences are arcs between statements on different processors (but correspond to the same iteration on each).

The key fact in analyzing such an augmented dependence graph is this: **A cycle in such a graph is a path that begins and ends at the same statement *on the same processor*.** (We have phrased this to include both the cases in which the cycle contains or does not contain spatial dependences.)

Now in general, it is hard to determine whether cycles exist in these graphs.[3] The reason for this is that the distribution of the data over the processors causes each processor to own data from widely different parts of an array. In this case, however, there is an easy way to see what is going on:

As indicated above, let us imagine creating a separate set of dependence graph nodes for the code on each processor. Having done this, we look at the spatial dependence on a particular (fixed) processor:

The guard on the SEND:

if $a(:, kk)$ is local

is disjoint from the guard on the RECEIVE:

if $a(:, kk)$ is not local and I hold one of the columns $kk{+}1:n$

Therefore, on any particular iteration of the kk loop, the dependence has a head or a tail, but not both. And further, each of these guards is kk-invariant; it is a function only of k and of my_processor(). (It is only really necessary that this be true for one of the guards.) Hence for a particular value of k (i.e., holding k fixed) and for each processor, the SEND/RECEIVE dependence is either incoming or outgoing, but not both: it therefore cannot appear in a cycle in the augmented dependence graph (there is no other spatial dependence that could be used to form such a cycle), and so can be ignored.

That is, in the special case in which within a loop the following three conditions are satisfied:

- There is only one SEND/RECEIVE pair,
- The SEND and the RECEIVE are guarded by disjoint expressions, and
- Either the SEND or the RECEIVE (or both) is guarded by an expression that is loop-invariant,

the spatial dependence corresponding to the SEND/RECEIVE pair cannot possibly be part of a cycle in the dependence graph, and so can be ignored in deciding whether the loop can be distributed.

So that is why the kk loop can be distributed as we did above.

On the other hand, on different k iterations, the dependence has both a head and a tail on each processor (see Figures 19, 20, and 21, in which the spatial dependences are indicated by dotted arrows), and so is a true obstacle to vectorization.

This resolves the two problems stated above: it explains why distributing the kk loop was legal, but why we could not distribute the k loop.

[3] If all the distributions are BLOCK or serial, the analysis is much easier, since the actual processors correspond to consecutive sequences of the virtual processors. However, pipelining of loop iterations does not typically occur in such cases. BLOCK distributions are characteristic of nearest-neighbor computations, which are already handled by a different mechanism.

7 Improving Communication Placement

Now this is much better, but we still are not finished. Comparing the code in Figure 11 to that in Figure 15, we see that we really only want to perform the FORALL loop for the range $j = kk{+}1 : hi(k)$, *then* perform the SEND, and finally, perform the rest of the FORALL loop. (This implements the pipelining we are looking for.)

Here is how the compiler can discover this: The dependence

$$U \to N \qquad (=,<)$$

only occurs for values of j in the range $lo(k) : hi(k)$. Similarly, the dependence

$$U \to N \qquad (<,<)$$

only occurs for values of j in the range $hi(k) + 1 : n$. Therefore, we can split the FORALL construct into two, with different ranges of j: $j \leq hi(k)$ and $j > hi(k)$. Figure 16 shows the dependence graph that results.

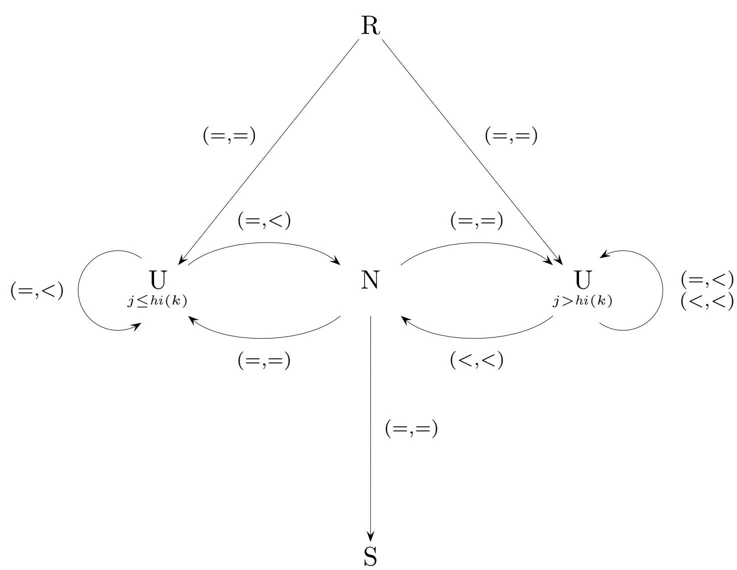

Figure 16: Dependence graph after splitting the j loop.

We saw above that the k loop must be serialized to preserve the SEND/RECEIVE semantics. After serializing this loop, we can ignore the dependences that are carried by the k loop, since they will automatically be satisfied. Thus, the only dependences that remain to be satisfied are represented in Figure 17.

In topologically sorting the strongly connected regions in this dependence graph, the only choice that has to be made is whether S should come last or not. Since S is the SEND node, and since we want to move each SEND as far forward as possible, we use the order

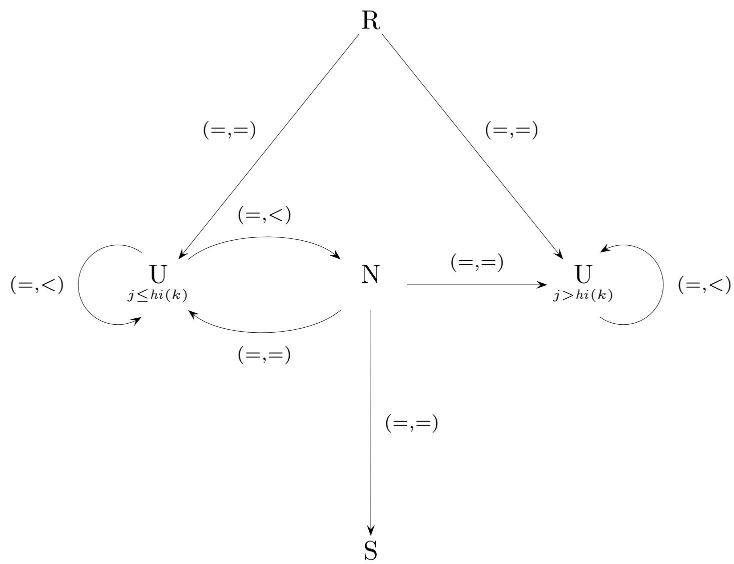

Figure 17: Dependence graph after splitting the j loop, ignoring dependences carried by the k loop.

R
N, U ($j \leq hi(k)$)
S
U ($j > hi(k)$)

This results in the code in Figure 18, which exhibits exactly the pipelining we have been looking for.

A more detailed way to picture what is going on is illustrated in Figure 22, which shows a still more augmented dependence graph. Here there is a separate graph for each iteration of the k loop. This graph makes it clear that the RECEIVE on processor 1 in iteration $k = 2$ can only happen after the SEND on the same processor in the previous iteration $k = 1$. This is what stops us from distributing the k loop over the SEND and RECEIVE nodes—it would schedule all the RECEIVEs at the beginning.

8 Blocking the Outer Loop

Finally, as promised, we indicate how the user can simply write code as in Figure 2 or Figure 3, and have the compiler discover the blocking of the outer k loop, as in Figure 10.

The matter is really quite simple. A loop is either blocked or it isn't. Blocking a loop is only useful if there is something profitable that can be done with the inner loop but not the outer loop. For example, there may be a computation that is invariant in the inner loop but not the outer, so it can be moved out of the inner loop. Or (as in this case) the inner loop can be distributed, leading to greater parallelism; the outer loop can't. This blocking in turn is only likely to happen if the loop

do $k = 1$, num_blocks

    ! Actually, no guard is generated for the following **receive** because the
    ! generated **do** loop bounds enforce the necessary constraints.

    if $a(:,\ lo(k))$ is not local and I hold one of the columns $lo(k{+}1):n$
        receive $a(:,\ lo(k):hi(k))$
    end if

    do $kk = lo(k), hi(k)$

        if $a(:,\ kk)$ is local   ! "I hold column kk"
            $a(kk{+}1:n,\ kk) = a(kk{+}1:n,\ kk)/a(kk,\ kk)$
        end if

        forall $(i = kk{+}1:n,\ j = kk{+}1:hi(k))$
            $a(i,\ j) = a(i,\ j) - a(i,\ kk) * a(kk,\ j)$
        end forall
    end do

    if $a(:,\ lo(k))$ is local   ! "I hold column $lo(k)$"
        send $a(:,\ lo(k):hi(k))$ to remote processors holding columns $lo(k{+}1):n$.
    end if

    do $kk = lo(k), hi(k)$

        forall $(i = kk{+}1:n,\ j = hi(k){+}1:n)$
            $a(i,\ j) = a(i,\ j) - a(i,\ kk) * a(kk,\ j)$
        end forall
    end do

end do

Figure 18: Final code, with pipelined loop iterations exposed.

index corresponds to a dimension of some array that is distributed CYCLIC(N).

So for each user-written loop, the compiler checks to see if the loop index corresponds to any CYCLIC(N) dimensions of arrays in the loop. If so, and if the values of N for all such dimensions are the same, then it may pay to block the loop with block size N. A further check can be made so see if this is actually a profitable transformation, by seeing if there is actually an invariant computation that can be moved out of the inner loop.

Actually, this is a bit oversimplified. The real constraints also involve how the loop index is used in the array expressions—in particular, if there is a multiplier or an offset (i.e. if the index k appears in a subscript of the form $a * k + b$). We won't bother writing down the actual constraints, which are not really very complicated.

In the LU example we have been considering, the loop index k is used to address both dimensions of the array a. Only one of those dimensions is distributed CYCLIC(N), and as we have seen, it is profitable to block the loop.

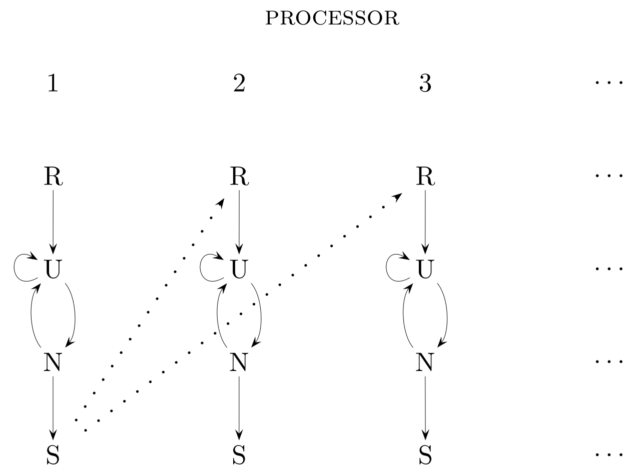

Figure 19: Dependence graph for k held constant. Here $k = 1$; processor 1 sends and the others receive. No cycle includes a spatial dependence.

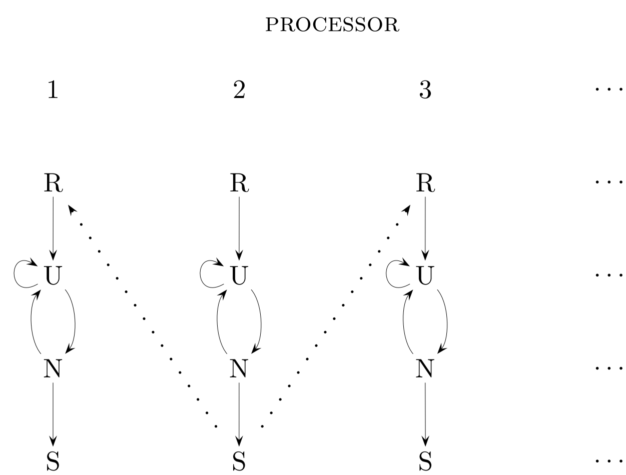

Figure 20: Dependence graph for k held constant. Here $k = 2$; processor 2 sends and the others receive. No cycle includes a spatial dependence.

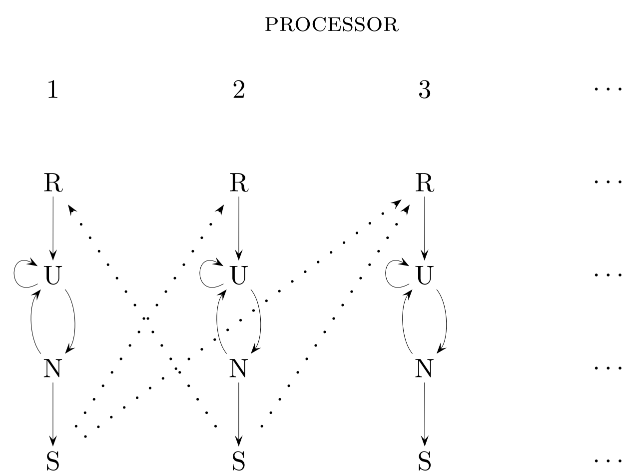

Figure 21: Dependence graph when k is allowed to vary. Dependences for $k = 1$ and $k = 2$ are shown here. Now there is a cycle in the graph including processors 1 and 2. This cycle prevents the k loop from being distributed.

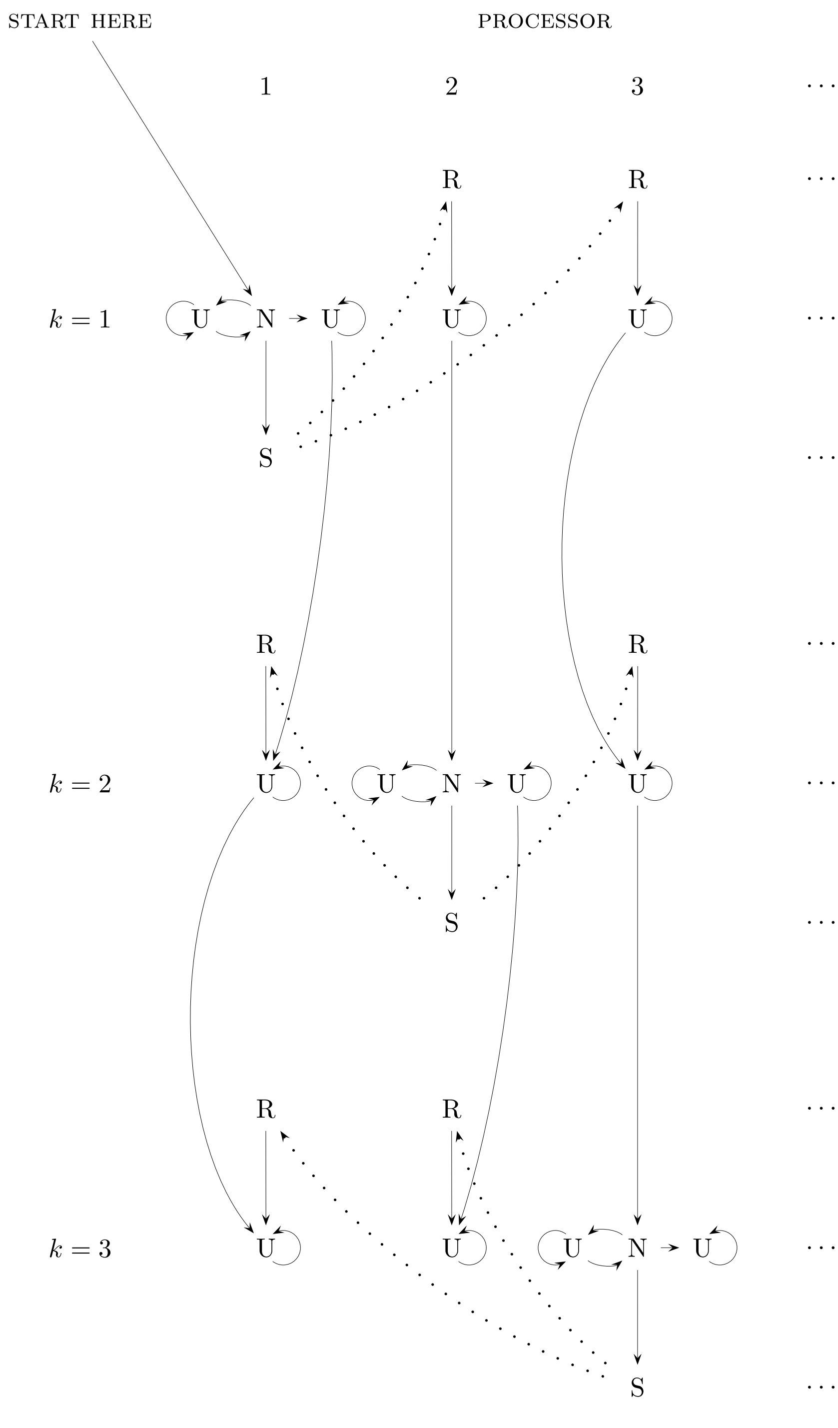

Figure 22: A still more augmented dependence graph. Here there is a separate graph for each iteration of the k loop and for each processor.