University of Maryland College Park

Institute for Advanced Computer Studies TR–92–44
Department of Computer Science TR–2880

Updating URV Decompositions in Parallel*

G. W. Stewart†

April 1992

ABSTRACT

A URV decomposition of a matrix is a factorization of the matrix into the product of a unitary matrix (U), an upper triangular matrix (R), and another unitary matrix (V). In [8] it was shown how to update a URV decomposition in such a way that it reveals the effective rank of the matrix. It was also argued that the updating procedure could be implemented in parallel on a linear array of processors; however, no specific algorithms were given. This paper gives a detailed implementation of the updating procedure.

*This report is available by anonymous ftp from `thales.cs.umd.edu` in the directory `pub/reports`.

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. This work was supported in part by the Air Force Office of Scientific Research under Contract AFOSR-87-0188. Part of it was done while the author was visiting the Istitute for Mathematics and Its Applications, University of Minnesota, Minneapolis, MN 55455.

Updating URV Decompositions
in Parallel

G. W. Stewart

1. Introduction

A matrix is of effective rank k if a small perturbation is sufficient to make it of rank k but a large perturbation is required to make it of rank $k-1$. Matrices that are effectively rank degenerate occur in a number of applications, in which it is required to determine the effective rank and to compute an orthonormal basis for its effective null space of the matrix (e.g., see [9]). Of course, the notions of "large" and "small", which define the effective rank, depend on the application.

If the $n \times p$ matrix X is of effective rank k, there are orthogonal matrices U and V such that

$$U^{\mathrm{T}} X V = \begin{pmatrix} R & F \\ 0 & G \\ 0 & 0 \end{pmatrix}, \tag{1.1}$$

where R and G are upper triangular matrices of orders k and $p-k$ and F and G are small. Such a decomposition is called a *rank-revealing URV decomposition.* If $V = (V_1\ V_2)$ is partitioned conformally with (1.1), then

$$U^{\mathrm{T}} X V_2 = \begin{pmatrix} F \\ G \end{pmatrix}$$

is small, so that the columns of V_2 furnish an orthonormal basis for the effective null space of X.

An example of a rank-revealing URV decomposition is the singular value decomposition [1], in which R and G are diagonal and F is zero. However, the singular value decomposition is expensive to compute and often furnishes more information than is needed to solve the problem at hand. By stepping back from diagonality we obtain a triangular decomposition that is cheaper to compute and easy to update.

The problem of updating occurs when rows are added to X. If z^{T} is a new row, then the updating problem is to determine a rank revealing URV decomposition of

$$\begin{pmatrix} X \\ z^{\mathrm{T}} \end{pmatrix}$$

from that of X. Since

$$\begin{pmatrix} U^{\mathrm{T}} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ z^{\mathrm{T}} \end{pmatrix} V = \begin{pmatrix} R & F \\ 0 & G \\ 0 & 0 \\ x^{\mathrm{T}} & y \end{pmatrix},$$

where $(x^{\mathrm{T}}\ y^{\mathrm{T}}) = z^{\mathrm{T}}(V_1\ V_2)$, the problem reduces to one of computing a rank-revealing URV factorization of

$$\begin{pmatrix} R & F \\ 0 & G \\ x^{\mathrm{T}} & y \end{pmatrix} \tag{1.2}$$

from the matrix

$$\begin{pmatrix} R & F \\ 0 & G \end{pmatrix}.$$

In [8] the author described an algorithm requiring $O(p^2)$ time for updating (1.2). The algorithm alternates between two steps:

Update: Reduce (1.2) to upper triangular form, preserving as far as possible the small elements of F and G.

Deflate: If the resulting matrix R is defective in rank, find its rank-revealing URV decomposition.

In the same paper it was shown that the algorithm could be made to run on a linear array of p processors in $O(p)$ time; however, no specific algorithms were given. One purpose of this paper is to provide the missing algorithms.

Another purpose is to illustrate a style of programming. Our model of computation will be fine-grain MIMD; that is, short messages interleaved with short computations without global control. Fine-grain SIMD algorithms have an extensive literature, as do coarse-grain MIMD algorithms (e.g., see [5, 6]). Less attention has been paid to fine-grain MIMD algorithms, chiefly because machines to run them have not been widely available. Now that such machines as the IWARP are in production, it is appropriate to illustrate coding techniques with a completely new and fairly complex set of algorithms.

The paper is organized as follows. In the next section we will describe the model of computation we will use. In Section 3 we show how to use precedence diagrams to pass from sequential to parallel algorithms. The parallel updating algorithm will be described piece by piece in the remaining sections. Although we will give brief sketches of the sequential algorithms, the reader should look to [8] for background and details.

2. Architectural Details

The way a parallel algorithm is implemented depends on the architecture of the system for which it is intended. In this paper we will work with a linear MIMD array that is capable of fine-grained communication. The number of processors, p, will be equal to the order of the matrix in question, and the matrices will be stored row-wise, one row to each processor. These are not the only possible choices, but they are well suited to matrices of moderate order, say thirty, such as arise in applications like signal processing.

The programming style will be SPMD; that is, each processor will run the same program with different data. A processor is identified by its position in the array, or equivalently by the row of the matrix it contains. This identifier is denoted by `I` in the programs to follow.

Communications is effected by bidirectional communication channels between processors. In the programs we will use the following conventions We will imagine the processors as being arranged in a vertical line, north to south. The function call (all code is in C)

```
fget(NORTH, &x);
```

assigns the variable `x` of type `float` the value in the north input register. If no input is available, the program is blocked until input appears. Similarly the statement

```
fget(SOUTH, &x);
```

gets input from the channel to the south. The statements

```
fput(NORTH, x);
fput(SOUTH, x);
```

send output to the north and south respectively. If there is already output in the channel, the program is blocked until the channel becomes free. We make no explicit assumptions about the speed of communication; but if the programs are to remain balanced, the time taken for two processors to exchange a floating-point number should be commensurate with the time required for a floating-point operation. This style of computing has the flavor of computations on a systolic array; however, it differs from it in the fact that no global synchronization of the processors is required. To give in a name we will call it *quasi-systolic*.

Instead of presenting entire programs, we will break the code into more manageable fragments. The fragments are from programs that have been debugged on a simulator that forks a process for each processor in the array. The communications functions are implemented by manipulating queues in shared memory. It has been shown [4] that if a program works on such a simulator it will work on a truly parallel system.

3. Precedence Diagrams and Parallel Code

It is not a trivial matter to pass from sequential code for a matrix algorithm to the corresponding quasi-systolic code. One obvious reason is that a quasi-systolic program must take on the additional responsibility of passing data between processors. However, a more subtle reason is that the two kinds of code look at the computations from different points of view. The sequential program follows the computations as they pass through the matrix, a global view analogous to the Lagrangian approach to fluid flow. The

$$
\begin{array}{ccccccccc}
\div & \xleftarrow{8} & \ast & \xleftarrow{7} & \ast & \xleftarrow{6} & \ast & \xleftarrow{5} & \ast \\
 & & 7\Uparrow & & 6\Uparrow & & 5\Uparrow & & 4\Uparrow \\
 & & \div & \xleftarrow{6} & \ast & \xleftarrow{5} & \ast & \xleftarrow{4} & \ast \\
 & & & & 5\Uparrow & & 4\Uparrow & & 3\Uparrow \\
 & & & & \div & \xleftarrow{4} & \ast & \xleftarrow{3} & \ast \\
 & & & & & & 3\Uparrow & & 2\Uparrow \\
 & & & & & & \div & \xleftarrow{2} & \ast \\
 & & & & & & & & 1\Uparrow \\
 & & & & & & & & \div
\end{array}
$$

$$
\begin{array}{lcl}
\ast & : & \texttt{w[I]} = \texttt{w[I]} + \texttt{r[I][j]} \ast \texttt{w[j]} \\
\div & : & \texttt{w[I]} = (\texttt{b[I]} - \texttt{w[I]})/\texttt{a[I][I]}
\end{array}
$$

Figure 3.1: Precedence Diagram for a Triangular System

quasi-systolic program takes the Eulerian approach; it sits at one part in the matrix and manipulates data as it flows by.

Precedence diagrams are a useful intermediary between sequential and parallel code. Consider, for example, the following sequential code for solving the $p \times p$ upper triangular system $Rw = b$.

Fragment 3.1. Sequential solution of $Rw = b$.

```
for (i=p; i>=1; i--){
   w[i] = 0.
   for (j=p; j>i; j--)
      w[i] = w[i] + r[i][j]*w[j];
   w[i] = (b[i] - w[i])/a[i][i];
}
```

A transition diagram for this code is given in Figure 3.1.

The matrix is stored row-wise on the linear array. The nodes of the diagram are associated with matrix elements and they represent a computation involving nearby elements. The computation is specified by a symbol, whose meaning is given in the legend below the diagram. Thus the nodes labeled $\ast$ are part of the computation of the inner product in the sequential program Fragment 3.1, while the nodes labeled $\div$ produce the components of the solution.

The arrows specify the order in which the calculation must occur. Thus the $\div$ operations in the diagram must proceed the $*$ operations above them. The numbers represent times at which the calculation can occur, and they must be distinct for each node on a processor. The vertical arrows are doubled to indicate that they involve communication across processors — in this case the passing of the components `w[j]` of the solution. To keep the numbers in the diagram to a reasonable size, a computation shares a number with its input; but the output from a computation has a higher number.

The easy part of the code implementing a precedence diagram like the one in Figure 3.1 is a loop that follows the arrows across a row — gathering data, computing, and communicating the results. The hard part is handling edge conditions, which invariably have special communication requirements. For example, if the direction of communication is generally north, then the first processor must not send output, and the last must not request input. Again, each row in a diagram will usually have a special node — the $\div$ node in Figure 3.1 — that must set things up for the next processor in line.

In the following code the `I`th processor contains the `I`th row of the matrix R in a singly subscripted array named `r`.

Fragment 3.2. Quasi-systolic solution of $Rw = b$.

```
w[I] = 0.;
for (j=p; j>I; j--){
   fget(SOUTH, &w[j]);
   w[I] = w[I] + r[j]*w[j];
   if (I != 1)
      fput(NORTH, w[j]);
}
w[I] = (b - w[I])/r[I];
if (I != 1)
   fput(NORTH, w[I]);
```

The `I`th right-hand side b is contained in `b`, while the components of the solution are contained in an array `w`. At the end of the program the entire solution is on the first processor, ready for distribution to the other processors, if it is needed.

It is worth noting that by replacing the horizontal arrows with double arrows, we obtain a precedence diagram suitable for a square array of processors, in which each processor is responsible for one matrix element. This will be true of many of the precedence diagrams to follow. The additional processors do not result in a corresponding savings in time since the longest path through the precedence graph is $O(p)$. However, on a square array it is often possible to pipeline such computations, something that is not possible on a linear array.

BEFORE

1	2	3	4	5	6	7	8
X	X	X	O	O	X	E	O
X	$\check{\text{X}}$	X	X	O	E	E	E

AFTER

1	2	3	4	5	6	7	8
X	X	X	X	O	X	E	E
X	O	X	X	O	X	E	E

Figure 4.1: Application of a Plane Rotation

4. Plane Rotations and Simple Updating [1]

Most of the computation in URV updating involves the use of plane rotations to introduce zeros into the matrix R. Since treatments of plane rotations are widely available (e.g., see [1]), we will not go into the numerical details here. Instead we will sketch the few basic facts needed to understand our algorithms and introduce some conventions for describing reductions based on plane rotations.

Figure 4.1 shows two rows of a matrix before and after the application of a plane rotation. The X's represent nonzero elements, the O's represent zero elements, and the E's represent small elements. The plane rotation has been chosen to introduce a zero into the position occupied by the checked X in column 2. When the rotation is applied the following rules hold.

1. A pair of X's remains a pair of X's (columns 1 and 3).
2. An X and an O are replaced by a pair of X's (column 4).
3. A pair of O's remains a pair of O's (column 5).
4. An X and an E are replaced by a pair of X's (column 6).
5. A pair of E's remains a pair of E's (column 7).
6. An E and an O are replaced by a pair of E's (column 8).

The fact that a pair of small elements remains small (columns 7 and 8) follows from the fact that a plane rotation is orthogonal and cannot change the norm of any vector to which it is applied. This is a key observations, since the point of the updating algorithm is to keep small elements small.

[1] This section draws heavily on a similar section in [8].

$$
\begin{array}{ccccc}
\rightarrow & r & r & r & r \\
 & 0 & r & r & r \\
 & 0 & 0 & r & r \\
 & 0 & 0 & 0 & r \\
\rightarrow & \check{x} & x & x & x
\end{array}
\Longrightarrow
\begin{array}{ccccc}
 & r & r & r & r \\
\rightarrow & 0 & r & r & r \\
 & 0 & 0 & r & r \\
 & 0 & 0 & 0 & r \\
\rightarrow & 0 & \check{x} & x & x
\end{array}
\Longrightarrow
\begin{array}{ccccc}
 & r & r & r & r \\
 & 0 & r & r & r \\
\rightarrow & 0 & 0 & r & r \\
 & 0 & 0 & 0 & r \\
\rightarrow & 0 & 0 & \check{x} & x
\end{array}
\Longrightarrow
$$

$$
\begin{array}{ccccc}
 & r & r & r & r \\
 & 0 & r & r & r \\
 & 0 & 0 & r & r \\
\rightarrow & 0 & 0 & 0 & r \\
\rightarrow & 0 & 0 & 0 & \check{x}
\end{array}
\Longrightarrow
\begin{array}{cccc}
r & r & r & r \\
0 & r & r & r \\
0 & 0 & r & r \\
0 & 0 & 0 & r \\
0 & 0 & 0 & 0
\end{array}
$$

Figure 4.2: Simple Updating

Premultiplication by a plane rotation operates on the rows of the matrix. We will call such rotations *left rotations*. Postmultiplication by *right rotations* operates on the columns. Analogous rules hold for the application of a right rotation to two columns of a matrix.

When rotations are used to update a URV decomposition, the right rotations must be multiplied into V. To get a complete update of the decomposition, we must also multiply the left rotations into U. However, in many applications U is not needed, and this step can be omitted. In this paper we will not give code for updating U.

Algorithms that use plane rotations are best described by pictures. To fix our conventions, we will show how the matrix

$$\begin{pmatrix} R \\ x^{\mathrm{T}} \end{pmatrix},$$

where R is upper triangular, can be reduced to upper triangular form by left rotations. This updating procedure occurs twice in the following algorithms and will be called *simple updating.*

The reduction is illustrated in Figure 4.2. The elements of R and x^{T} are represented generically by r's and x's. The first step in the reduction is to eliminate the first element of x^{T} by a rotation that acts on x^{T} and the first row of R. The element to be eliminated has a check over it and the two rows that are being combined are indicated by the arrows to the left of the array.

According to this notation, the second step combines the second row of R with x^{T} to eliminate the second element of the latter. Note that r_{21}, which is zero, forms a pair

$$
\begin{array}{ccccccccc}
\bullet & \xrightarrow{1} & \circ & \xrightarrow{2} & \circ & \xrightarrow{3} & \circ & \xrightarrow{4} & \circ \\
 & & 2\Downarrow & & 3\Downarrow & & 4\Downarrow & & 5\Downarrow \\
 & & \bullet & \xrightarrow{3} & \circ & \xrightarrow{4} & \circ & \xrightarrow{5} & \circ \\
 & & & & 4\Downarrow & & 5\Downarrow & & 6\Downarrow \\
 & & & & \bullet & \xrightarrow{5} & \circ & \xrightarrow{6} & \circ \\
 & & & & & & 6\Downarrow & & 7\Downarrow \\
 & & & & & & \bullet & \xrightarrow{7} & \circ \\
 & & & & & & & & 8\Downarrow \\
 & & & & & & & & \bullet
\end{array}
$$

$\bullet$: Generate rotation for `r[I][I]` and `x[I]`
$\circ$: Apply rotation

Figure 5.1: Precedence Diagram for Simple Updating

of zeros with the first component of x^{T}, so that the zero we introduced in the first step is not destroyed in the second step. The third and fourth steps of the reduction are similar.

For column operations with right rotations we will use an analogous notation. The main difference is that the arrows will point down to the columns being combined.

5. Simple Updating in Parallel

Figure 5.1 contains a precedence diagram for simple updating. The diagram assumes that the vector x^{T} lies in the first processor. After a component of x^{T} has been combined with the row on one processor, it drops to the next to be combined with the next row.

The following fragment implements the algorithm.

Fragment 5.1. Simple updating.

```
for (j=I; j<=p; j++){
   if (I == 1)
      xx = x[j];
   else
     fget(NORTH, &xx);
   if (j == I)
      rotgen(&r[j], &xx, &c, &s);
```

```
      else{
         rotapp(&r[j], &xx, &c, &s);
         if (I != p)
            fput(SOUTH, xx);
      }
   }
```

The function `rotgen(&r, &x, &c, &s)` generates a rotation from `r` and `x`, returning $\sqrt{\mathtt{r}^2 + \mathtt{x}^2}$ in `r` and zero in `x`. The function `rotapp(&r, &x, &c, &s)` applies the rotation whose cosine and sine are `c` and `s` to `r` and `s`, returning the altered values of `r` and `x`.

6. Updating

We now turn to the problem of updating the matrix (1.2). The first step is to compute $z^{\mathrm{T}}V$. We will assume that z^{T} is situated in the topmost processor. We begin by passing the `I`th component down the array until it reaches processor `I`, where it is stored in `zi`. This passing of data is simple enough not to require a precedence diagram, and we give only the code.

Fragment 6.1. Distribution of z^{T}.

```
for (j=p; j>I; j--){
   if (I == 1)
      temp = z[j];
   else
      fget(NORTH, &temp);
   fput(SOUTH, temp);
}
if (I == 1)
   zi = z[1];
else
   fget(NORTH, &zi);
```

Once the components of z^{T} are in place, they are are multiplied into the components of the rows of V, the products being passed down and summed on the way. The precedence diagram is given in Figure 6.1. The corresponding fragment is

Fragment 6.2. Formation of $z^{\mathrm{T}}V$.

```
for (j=1; j<=p; j++){
   if (I == 1)
      sum = 0;
   else
      fget(NORTH, &sum);
```

$$\begin{array}{ccccccc}
* & \xrightarrow{2} & * & \xrightarrow{3} & * & \xrightarrow{4} & * \\
2\Downarrow & & 3\Downarrow & & 4\Downarrow & & 5\Downarrow \\
* & \xrightarrow{3} & * & \xrightarrow{4} & * & \xrightarrow{5} & * \\
3\Downarrow & & 4\Downarrow & & 5\Downarrow & & 6\Downarrow \\
* & \xrightarrow{4} & * & \xrightarrow{5} & * & \xrightarrow{6} & * \\
4\Downarrow & & 5\Downarrow & & 6\Downarrow & & 7\Downarrow \\
* & \xrightarrow{5} & * & \xrightarrow{6} & * & \xrightarrow{7} & *
\end{array}$$

$$* \quad : \quad \mathtt{sum} = \mathtt{sum} + \mathtt{zi} * \mathtt{v}[\mathtt{j}];$$

Figure 6.1: Precedence Diagram for $z^{\mathrm{T}}V$

```
      sum = sum + zi*v[j];
      if (I != p)
         fput(SOUTH, sum);
      else
         x[j] = sum;
   }
```

Finally the vector x^{T}, which is now on the pth processor, is distributed to all the processors:

Fragment 6.3. Distribution of x^{T}

```
   for (j=1; j<=p; j++){
      if (I != p)
         fget(SOUTH, &x[j]);
      if (I != 1)
         fput(NORTH, x[j]);
   }
```

The next step in the updating algorithm depends on the size of the vector y^{T} in (1.2); that is, the vector whose components are `x[k+1]`, `x[k+2]`, ..., `x[p]`. If simple updating is performed on (1.2), the result will be a new matrix

$$\begin{pmatrix} \hat{R} & \hat{F} \\ 0 & \hat{G} \end{pmatrix},$$

where $\|\hat{F}\|_{\mathrm{F}}^2 + \|\hat{G}\|_{\mathrm{F}}^2 = \|F\|_{\mathrm{F}}^2 + \|G\|_{\mathrm{F}}^2 + \|y^{\mathrm{T}}\|_{\mathrm{F}}^2$. Consequently, if y^{T} is small enough, the updated matrix will be of effective rank not greater than k. In this case, simple

$$
\begin{array}{cccc}
 & & \downarrow & \downarrow \\
f & f & f & f \\
g & g & g & g \\
0 & g & g & g \\
0 & 0 & g & g \\
0 & 0 & 0 & g \\
y & y & y & \check{y}
\end{array}
\Longrightarrow
\begin{array}{c|cccc}
 & f & f & f & f \\
 & g & g & g & g \\
 & 0 & g & g & g \\
\rightarrow & 0 & 0 & g & g \\
\rightarrow & 0 & 0 & \check{g} & g \\
 & y & y & y & 0
\end{array}
\Longrightarrow
\begin{array}{cccc}
 & \downarrow & \downarrow & \\
f & f & f & f \\
g & g & g & g \\
0 & g & g & g \\
0 & 0 & g & g \\
0 & 0 & 0 & g \\
y & y & \check{y} & 0
\end{array}
\Longrightarrow
\begin{array}{c|cccc}
 & f & f & f & f \\
 & g & g & g & g \\
\rightarrow & 0 & g & g & g \\
\rightarrow & 0 & \check{g} & g & g \\
 & 0 & 0 & 0 & g \\
 & y & y & 0 & 0
\end{array}
\Longrightarrow
$$

$$
\begin{array}{cccc}
\downarrow & \downarrow & & \\
f & f & f & f \\
g & g & g & g \\
0 & g & g & g \\
0 & 0 & g & g \\
0 & 0 & 0 & g \\
y & \check{y} & 0 & 0
\end{array}
\Longrightarrow
\begin{array}{c|cccc}
 & f & f & f & f \\
\rightarrow & g & g & g & g \\
\rightarrow & \check{g} & g & g & g \\
 & 0 & 0 & g & g \\
 & 0 & 0 & 0 & g \\
 & y & 0 & 0 & 0
\end{array}
\Longrightarrow
\begin{array}{cccc}
f & f & f & f \\
g & g & g & g \\
0 & g & g & g \\
0 & 0 & g & g \\
0 & 0 & 0 & g \\
y & 0 & 0 & 0
\end{array}
$$

Figure 6.2: Reduction of y^{T}

updating followed by the deflation procedure to be described later is what we want to do.

However, if y^{T} is sufficiently large, we must take into the account a possible increase in effective rank. Here simple updating will not do, since it will fold the large elements of y^{T} into F and G and destroy the rank-revealing character of the decomposition. The cure is to preprocess the matrix so that only the first component of y^{T} is nonzero while G remains upper triangular; for then simple updating will make at most the $(k+1)$th column large.

The reduction is illustrated in Figure 6.2. Left rotations introduce zeros into the component of y. In the process each rotation introduces a nonzero element below the diagonal of G, which is removed by a right rotation.

The parallel implementation of this reduction is the trickiest part of the algorithm. The reason is that the left and right rotations must be interleaved. We will proceed in two stages.

Figure 6.3 gives a precedence diagram for part of the reduction. The array begins with the $(k+1, k+1)$-element. The right rotations come from the south and are applied to R and V before being passed north. The application of the rotations continues through the matrix F up to row one (not shown in the diagram). Here is the code.

Fragment 6.4. Reduction of y^{T}, Part 1.

$$
\begin{array}{ccccccccc}
\bullet & \overset{9}{\leftarrow} & \circ & \overset{8}{\leftarrow} & \circ & \overset{7}{\leftarrow} & \circ & \overset{6}{\leftarrow} & \circ \\
9\Uparrow & & 8\Uparrow & & 7\Uparrow & & 6\Uparrow & & 5\Uparrow \\
 & & \bullet & \overset{7}{\leftarrow} & \circ & \overset{6}{\leftarrow} & \circ & \overset{5}{\leftarrow} & \circ \\
 & & & & 6\Uparrow & & 5\Uparrow & & 4\Uparrow \\
 & & & & \bullet & \overset{5}{\leftarrow} & \circ & \overset{4}{\leftarrow} & \circ \\
 & & & & & & 4\Uparrow & & 3\Uparrow \\
 & & & & & & \bullet & \overset{3}{\leftarrow} & \circ \\
 & & & & & & & & 2\Uparrow \\
 & & & & & & & & \bullet
\end{array}
$$

$\bullet$: 1. Generate left rotation for rows `I` and `I+1`
2. Generate right rotation for columns `I-1` and `I`

$\circ$: Apply right rotation to R and V

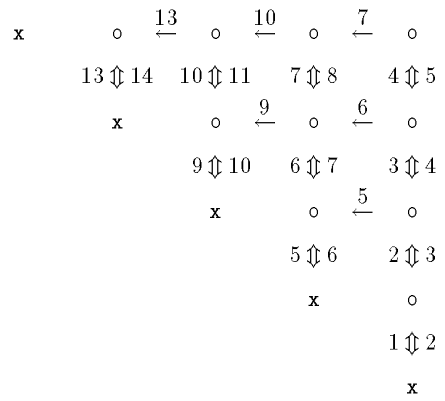
Figure 6.3: Precedence Diagram for the Reduction of y^{T}, Part 1

```
for (j=p; j>=(k+2>I+1?k+2:I+1); j--){
   fget(SOUTH, &cr[j]);
   fget(SOUTH, &sr[j]);
   rotapp(&r[j-1], &r[j], &cr[j], &sr[j]);
   rotapp(&v[j-1], &v[j], &cr[j], &sr[j]);
   if (I != 1){
      fput(NORTH, cr[j]);
      fput(NORTH, sr[j]);
   }
}
```

To generate the left rotation, processor `I` requires the element $r_{\mathtt{I},\mathtt{I}+1}$ from the south. To generate the right rotation, processor `I` requires the current value of $y_{\mathtt{I}}$ from the south. It is important that the corresponding information be sent north in the proper order.

Fragment 6.5. Reduction of y^{T}, Part 1 continued.

```
if (I!=p && I>k){
   fget(SOUTH, &rinf);
```



o : Apply left rotation to rows I and $\mathtt{I}+1$
x : No computation

Figure 6.4: Reduction of y^{T}, Part 2

```
      rotgen(&r[I], &rinf, &cl, &sl);
   }
   if (I > k+1){
      if (I != p)
         fget(SOUTH, &x[I]);
      rotgen(&x[I-1], &x[I], &cr[I], &sr[I]);
      fput(NORTH, cr[I]);
      fput(NORTH, sr[I]);
      rotapp(&r[I-1], &r[I], &cr[I], &sr[I]);
      fput(NORTH, r[I-1]);
      r[I-1] = 0.;
      if (I != k+2)
         fput(NORTH, x[I-1]);
   }
```

We now turn to the application of the left rotations. Since a rotation can be applied to a pair of elements only after rotations have been applied to all inferior pairs, some care must be taken to get an $O(p)$ algorithm. The trick is to apply a rotation and allow the superior processor to apply the same rotation before going on to the next rotation. Figure 6.4 contains a precedence diagram for this process. The array begins at the $(k+1,k+1)$-element. The double arrows indicate that an element must be transmitted and then returned. Code implementing this diagram follows.

Fragment 6.6. Reduction of y^{T}, Part 2.

```
if (I > k){
   for (j=p; j>=I; j--){
      if (j != I){
         fget(SOUTH, &rinf);
         rotapp(&r[j], &rinf, &cl, &sl);
         fput(SOUTH, rinf);
      }
      if (I != k+1){
         fput(NORTH, r[j]);
         fget(NORTH, &r[j]);
      }
   }
}
```

It remains to apply the right rotations to the part of V lying on and below the diagonal. The code is simple and requires no particular explanation.

Fragment 6.7. Reduction of y^{T}, Part 2 continued.

```
for (j=I; j>k+1; j--){
   if (j != I){
      fget(NORTH, &cr[j]);
      fget(NORTH, &sr[j]);
   }
   rotapp(&v[j-1], &v[j], &cr[j], &sr[j]);
   if (I != p){
      fput(SOUTH, cr[j]);
      fput(SOUTH, sr[j]);
   }
}
```

The updating process is completed with a simple update. Consequently, we must move the reduced vector to the topmost processor. Note that only nonzero (i.e., the first $k+1$) components have to be moved. The follow code accomplishes this task.

Fragment 6.8. Moving x^{T} to the top.

```
if (I==1)
   for (j=k+2; j<=p; j++)
      x[j] = 0;
if (I < k+2 && I!=p)
   fget(SOUTH, &x[k+1]);
if (I<=k+2 && I!=1)
   fput(NORTH, x[k+1]);
```

7. Deflation

An increase in effective rank will show up in the updating process. A decrease in effective rank is also possible; however, it is not as easy to detect. This section is devoted to algorithms to detect a decrease in effective rank and make the URV decomposition reveal it, a process we will call *deflation*.

The procedure is as follows. First determine a vector w of norm one such that $b = Rw$ is as small as possible, or at least nearly so. If $\|b\|$ is greater than an (application dependent) tolerance, declare R to be of full rank and stop. Otherwise, determine orthogonal matrices P and Q such that $Q^{\mathrm{T}}w = \mathbf{e}_k$, where $\mathbf{e}_k$ is the kth unit vector, and $P^{\mathrm{T}}RQ$ is upper triangular. It then follows that

$$\|b\| = \|Rw\| = \|(P^{\mathrm{T}}RQ)Q^{\mathrm{T}}w\| = \|(P^{\mathrm{T}}RQ)\mathbf{e}_k\|,$$

or equivalently that the norm of the last column of the upper triangular matrix $P^{\mathrm{T}}RQ$ is equal to $\|b\|$, which is small. Thus, $P^{\mathrm{T}}RQ$ is a rank-revealing URV decomposition of R.

The determination of the vector w falls under the rubric of condition estimation (for a survey see [3]). Here we give a very simple condition estimator [2]. The idea is to solve the system $R\hat{w} = b$, where the components of b are ± 1, the sign being chosen to maximize the growth of $\hat{w}$. The vector w is given by $\hat{w}/\|\hat{w}\|$. The following code is adapted from Fragment 3.2.

Fragment 7.1. Condition estimation.

```
if (I <= k){
   w[I] = 0;
   for (j=k; j>I; j--){
      fget(SOUTH, &w[j]);
      w[I] = w[I] + r[j]*w[j];
      if (I != 1)
         fput(NORTH, w[j]);
   }
   w[I] = ((w[I]<0 ? 1 : -1) - w[I])/r[I];
   if (I != 1)
      fput(NORTH, w[I]);
}
```

The next step is to compute the reciprocal of the norm of $\hat{w}$ and distribute it through the array, so that each processor can decide whether deflation is necessary. The function `nrm2` computes the norm.

Fragment 7.2. Distribution of $\|\hat{w}\|$.

$$
\begin{array}{cccc|ccc}
\downarrow & \downarrow & & & & \\
r & r & r & r & \rightarrow & w \\
0 & r & r & r & \rightarrow & \breve{w} \\
0 & 0 & r & r & & w \\
0 & 0 & 0 & r & & w
\end{array}
\Longrightarrow
\begin{array}{cccc|ccc}
 & \downarrow & \downarrow & & & \\
r & r & r & r & & w \\
r & r & r & r & \rightarrow & w \\
0 & 0 & r & r & \rightarrow & \breve{w} \\
0 & 0 & 0 & r & & w
\end{array}
\Longrightarrow
$$

$$
\begin{array}{cccc|ccc}
 & & \downarrow & \downarrow & & \\
r & r & r & r & & 0 \\
r & r & r & r & & 0 \\
0 & r & r & r & \rightarrow & w \\
0 & 0 & 0 & r & \rightarrow & \breve{w}
\end{array}
\Longrightarrow
\begin{array}{cccc|c}
r & r & r & r & 0 \\
r & r & r & r & 0 \\
0 & r & r & r & 0 \\
0 & 0 & r & r & 1
\end{array}
$$

Figure 7.1: Reduction of $\hat{w}$

```
if (I == 1){
   *rnormw = sqrt((double) k)/nrm2(k, w);
else
   fget(NORTH, rnormw);
if (I != p)
   fput(SOUTH, *rnormw);
```

Assuming a deflation is necessary, the next step is to reduce w to a multiple of $\mathbf{e}_k$ by plane rotations. Since w and $\hat{w}$ produce the same sequence of rotations, we work with the latter. The rotations are post multiplied into R to produce RQ, which is upper Hessenberg. The process is illustrated in Figure 7.1. A precedence diagram is given in Figure 7.2.

We assume that $\hat{w}$ is in the first row, where the rotations are generated. They are then passed south, where they are applied to R and V, as in the following code.

Fragment 7.3. Reduction of $\hat{w}$.

```
for (j=2; j<=k; j++){
   if (I == 1)
      rotgen(&w[j], &w[j-1], &c[j], &s[j]);
   else{
      fget(NORTH, &c[j]);
      fget(NORTH, &s[j]);
   }
   if (I <= k)
      rotapp(&r[j], &r[j-1], &c[j], &s[j]);
   rotapp(&v[j], &v[j-1], &c[j], &s[j]);
```

$$
\begin{array}{cccccccc}
\mathtt{x} & \bullet & \xrightarrow{2} & \bullet & \xrightarrow{3} & \bullet & \xrightarrow{4} & \bullet \\
 & 2\Downarrow & & 3\Downarrow & & 4\Downarrow & & 5\Downarrow \\
\mathtt{x} & \circ & \xrightarrow{3} & \circ & \xrightarrow{4} & \circ & \xrightarrow{5} & \circ \\
 & 3\Downarrow & & 4\Downarrow & & 5\Downarrow & & 6\Downarrow \\
\mathtt{x} & \mathtt{v} & \xrightarrow{4} & \circ & \xrightarrow{5} & \circ & \xrightarrow{6} & \circ \\
 & 4\Downarrow & & 4\Downarrow & & 6\Downarrow & & 7\Downarrow \\
\mathtt{x} & \mathtt{v} & \xrightarrow{5} & \mathtt{v} & \xrightarrow{6} & \circ & \xrightarrow{7} & \circ \\
 & 5\Downarrow & & 6\Downarrow & & 7\Downarrow & & 8\Downarrow \\
\mathtt{x} & \mathtt{v} & \xrightarrow{6} & \mathtt{v} & \xrightarrow{7} & \mathtt{v} & \xrightarrow{8} & \circ
\end{array}
$$

$\bullet$: Generate and apply rotation, for `w[j-1]` and `w[j]`
$\circ$: Apply rotation to `r` and `v`
`v` : Apply rotation to `v`
`x` : No computation

Figure 7.2: Precedence Diagram for Reduction of $\hat{w}$

$$
\begin{array}{ccccc}
\rightarrow & r & r & r & r \\
\rightarrow & \check{r} & r & r & r \\
 & 0 & r & r & r \\
 & 0 & 0 & r & r
\end{array}
\Longrightarrow
\begin{array}{ccccc}
 & r & r & r & r \\
\rightarrow & 0 & r & r & r \\
\rightarrow & 0 & \check{r} & r & r \\
 & 0 & 0 & r & r
\end{array}
\Longrightarrow
\begin{array}{ccccc}
 & r & r & r & r \\
 & 0 & r & r & r \\
\rightarrow & 0 & 0 & r & r \\
\rightarrow & 0 & 0 & \check{r} & r
\end{array}
\Longrightarrow
\begin{array}{cccc}
r & r & r & r \\
0 & r & r & r \\
0 & 0 & r & r \\
0 & 0 & 0 & r
\end{array}
$$

Figure 7.3: Reduction of Hessenberg R

```
    if (I != p){
       fput(SOUTH, c[j]);
       fput(SOUTH, s[j]);
    }
 }
```

The matrix R is now upper Hessenberg, and the final step of the deflation is to reduce the subdiagonal elements to zero. This process is shown in Figure 7.3.

A precedence diagram is given in Figure 7.4. As always with rotations that cross processor boundaries, it is important for a processor feed the next one before continuing

$$
\begin{array}{ccccccccc}
\mathtt{x} & & \mathtt{x} & & \mathtt{x} & & \mathtt{x} & & \mathtt{x} \\
1 \Updownarrow 2 & & 3 \Updownarrow 4 & & 6 \Updownarrow 7 & & 9 \Updownarrow 10 & & 12 \Updownarrow 13 \\
\bullet & \xrightarrow{3} & \circ & \xrightarrow{6} & \circ & \xrightarrow{9} & \circ & \xrightarrow{12} & \circ \\
 & & 4 \Updownarrow 5 & & 7 \Updownarrow 8 & & 10 \Updownarrow 11 & & 13 \Updownarrow 14 \\
 & & \bullet & \xrightarrow{7} & \circ & \xrightarrow{10} & \circ & \xrightarrow{13} & \circ \\
 & & & & 8 \Updownarrow 9 & & 11 \Updownarrow 12 & & 14 \Updownarrow 15 \\
 & & & & \bullet & \xrightarrow{11} & \circ & \xrightarrow{14} & \circ \\
 & & & & & & 12 \Updownarrow 13 & & 15 \Updownarrow 16 \\
 & & & & & & \bullet & \xrightarrow{15} & \circ
\end{array}
$$

•	:	Generate rotation for rows `I-1` and `I`
∘	:	Apply rotation
`x`	:	No computation

Figure 7.4: Precedence Diagram for Reduction of Hessenberg R

to do its own thing.

Fragment 7.4. Reduction of Hessenberg R.

```
if (I > k) return;
for (j=I-1; j<=p; j++){
   if (I != 1){
      fget(NORTH, &rsup);

      if (j == I-1)
         rotgen(&rsup, &r[j], &c, &s);
      else
         rotapp(&rsup, &r[j], &c, &s);
      fput(NORTH, rsup);
   }

   if (j!=I-1 && I!=k){
      fput(SOUTH, r[j]);
      fget(SOUTH, &r[j]);
   }
}
```

$$
\begin{array}{cccc}
 & & \downarrow & \downarrow \\
r & r & r & e \\
0 & r & r & e \\
0 & 0 & r & \check{e} \\
0 & 0 & 0 & e
\end{array}
\Longrightarrow
\begin{array}{cccc}
 & \downarrow & & \downarrow \\
r & r & r & e \\
0 & r & r & \check{e} \\
0 & 0 & r & 0 \\
0 & 0 & e & e
\end{array}
\Longrightarrow
\begin{array}{cccc}
\downarrow & & & \downarrow \\
r & r & r & \check{e} \\
0 & r & r & 0 \\
0 & 0 & r & 0 \\
0 & e & e & e
\end{array}
\Longrightarrow
\begin{array}{cccc}
 & & & \\
r & r & r & 0 \\
0 & r & r & 0 \\
0 & 0 & r & 0 \\
e & e & e & e
\end{array}
$$

Figure 8.1: Reducing the Last Column

This completes one step of the deflation. Since it is possible for the rank to decrease by more than one, the deflation must be repeated, with k reduced by one at each repetition, until R is declared to be of full rank. However, after each iteration, one may choose to to apply an optional refinement step, which can further reduce the small elements above r_{kk}.

8. Refinement

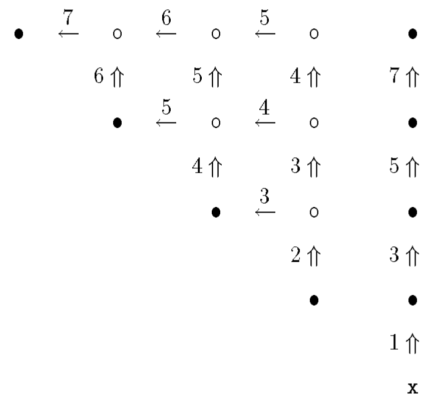
The refinement step is essentially a block QR iteration and consists of two parts. In the first part, the elements above r_{kk} are reduced to zero by left rotations. This process introduces small elements in the last row, which are then eliminated in the second part by simple updating. For a detailed analysis of this step, see [7].

The reduction of the last column of R is shown in Figure 8.1. A precedence diagram is given in Figure 8.2. An element in the last column and its corresponding diagonal element form a single computational node in which $r_{\mathrm{I}k}$ is zeroed by combining it with r_{II}. The last row of R is passed north with the rotations.

The code in Fragment 8.1 begins by starting the kth row of R north, where it will be accumulated in the array `rk` in the topmost processor. The process continues with the application of previously generated rotations, followed by the generation of a new rotation. The new nonzero elements introduced into the kth row of R are also passed up. Finally, elements $k+1$ to p of the kth row of R are moved to the top.

Fragment 8.1. Reducing the last column.

```
if (I == k)
   for (j=k; j<=p; j++)
      fput(NORTH, r[j]);
if (I < k){
   for (j=k-1; j>I; j--){
      fget(SOUTH, &c);
      fget(SOUTH, &s);
```



$\bullet$: Generate rotation for `r[I][I]`, `r[I][k]`.
$\circ$: Apply rotation.
x : Pass `r[k][k]` north.

Figure 8.2: Precedence Diagram for Reducing the Last Column

```
      fget(SOUTH, &rk[j]);
      if (I != 1){
         fput(NORTH, c);
         fput(NORTH, s);
         fput(NORTH, rk[j]);
      }
      rotapp(&r[j], &r[k], &c, &s);
      rotapp(&v[j], &v[k], &c, &s);
   }
   rotgen(&r[I], &r[k], &c, &s);
   rotapp(&v[I], &v[k], &c, &s);
   fget(SOUTH, &rk[k]);
   rk[I] = 0.;
   rotapp(&rk[I], &rk[k], &c, &s);
   if (I != 1){
      fput(NORTH, c);
      fput(NORTH, s);
      fput(NORTH, rk[I]);
      fput(NORTH, rk[k]);
   }
   for (j=k+1; j<=p; j++){
```

```
            fget(SOUTH, &rk[j]);
            if (I != 1)
               fput(NORTH, rk[j]);
         }
         if (I != p){
            fput (SOUTH, c);
            fput (SOUTH, s);
         }
      }
```

After the reduction, the right rotations, which are in the first processor, are applied to V as in Fragment 6.7.

The last row of R, which is now nonzero is in the first processor. This row is reduced to zero by a variant of the simple updating algorithm Fragment 5.1. This restores R to upper triangular form, after which k is reduced by one and the new, smaller R is checked for effective deficiency in rank.

9. Observations and Conclusions

The code presented in this paper has a finality that belies the work involved in actually writing. In this section I would like to make some informal observations on coding and debugging quasi-systolic algorithms.

Unless the hardware and software on your target system are completely reliable, it is a good idea to do preliminary debugging on a simulator. A simulator is easy to write using standard system routines. Since it runs on a single machine, it is easy to insert debugging statements and direct output to wherever you wish. Moreover, if a `put-get` discipline, such as the one in this paper, is used to synchronize computations, successful execution on a simulator guarantees that the code will run in parallel.

In most cases the starting point for parallel code is a sequential algorithm. If you do not understand the latter there is little hope of your producing the former. Begin by coding and debugging the sequential version (unless of course you already understand it well).

I found precedence diagrams to be extremely useful in passing from the sequential algorithm to parallel code. However, there is a learning curve here, and toward the end of the project my diagrams degenerated into impressionistic sketches.

Unless you are careful, you will find you are spending more time debugging your test cases than your code. It is worth your while to put hard thought into devising tests whose correctness can be recognized without elaborate post-processing. In running test cases, all the usual caveats about testing boundary conditions apply. In particular, it is easy to write code that performs adequately when communication is well buffered but fails when the buffer queues have length one.

Most of the standard debugging tools do not work well with simulators, and you will probably be reduced to inserting explicit debugging output in your code. If you understand the sequential algorithm and your tests are well constructed, the main problems will be with the flow of your algorithm. The vast majority of my debugging output consisted of statements like

```
<Proc id>: I am here <position id>
```

Although the code given here assumes only one row is allocated to each processor, it would not be a difficult task to extend it to the case where several consecutive rows are assigned to each processor.

I will conclude by expressing my belief that quasi-systolic algorithms implemented on linear arrays will become increasingly important in the solution of medium size, dense matrix problems. Such problems arise regularly in real-time control and signal processing applications, where they must be solved with dispatch. A linear array insures that the hardware can be both manageably small and inexpensive. The MIMD implementation of quasi-systolic algorithms means that the same system can be adapted to solve a wide variety of problems.

References

[1] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, Maryland, 2nd edition, 1989.

[2] W. B. Gragg and G. W. Stewart. A stable variant of the secant method for solving nonlinear equations. *SIAM Journal on Numerical Analysis*, 13:880–903, 1976.

[3] N. J. Higham. A survey of condition number estimation for triangular matrices. *SIAM Review*, 29:575–596, 1987.

[4] D. P. O'Leary and G. W. Stewart. From determinacy to systaltic arrays. *IEEE Transactions on Computers*, C-36:1355–1359, 1987.

[5] J. M. Ortega, R. G. Voigt, and C. H. Romine. *A Bibliography on Parallel and Vector Numerical Algorithms*. Technical Report TM-10998, Oak Ridge National Laboratory, 1989.

[6] Y. Robert. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*. Halsted Press, New York, 1990.

[7] G. W. Stewart. *On an Algorithm for Refining a Rank-Revealing URV Decomposition and a Perturbation Theorem for Singular Values*. Technical Report CS-TR 2626, Department of Computer Science, University of Maryland, 1991. To appear in *Linear Algebra and Its Applications.*.

[8] G. W. Stewart. *An Updating Algorithm for Subspace Tracking.* Technical Report CS-TR 2494, Department of Computer Science, University of Maryland, 1990. To appear in IEEE Transactions on Signal Processing.

[9] R. J. Vaccaro, editor. *SVD and Signal Processing, II*, Elsevier Science Publishers, Amsterdam, 1990.