THE ALL-PAIRS PIPELINE

An all-pairs problem is a computation on every possible subset consisting of two elements chosen from a set of n elements. N-body simulation and Householder reduction are all-pairs problems. This chapter defines the all-pairs problem concisely by means of precedence matrices and derives a parallel algorithm. The algorithm is presented in both coarse-grain and medium-grain form. The all-pairs paradigm is illustrated by a pipeline for Householder reduction of a matrix to triangular form.

1 INTRODUCTION

Successful exploitation of parallel computers depends to a large extent on the development of useful concepts which enable programmers to view different applications as variations of a common theme. Our most fundamental concepts, such as parallel processes and message communication, are embedded in programming languages. In other cases, we discover programming paradigms which can be used to solve a class of applications.

An *all-pairs problem* is a computation on every possible subset consisting of two elements chosen from a set of n elements. N-body simulation is an all-pairs problem (Fox 1988, Brinch Hansen 1991). Householder reduction of a matrix to triangular form is a less obvious example (Press 1989, Brinch Hansen 1990a). This paper develops the all-pairs paradigm discussed by Shih (1987) and Cosnard (1988). I define the problem concisely by means of precedence matrices and derive a parallel algorithm. The algorithm is presented in both coarse-grain and medium-grain form. The all-pairs paradigm is illustrated by a pipeline for Householder reduction.

Pipeline algorithms for matrix reduction have already been developed based on a detailed understanding of various reduction methods, such as Gaussian elimination, Givens rotations, and Householder reduction (Ortega 1988).

P. Brinch Hansen, *Studies in Computational Science*, Chapter 4: The all-pairs pipeline. Prentice Hall, Englewood Cliffs, NJ, (March 1995), 73–89.

I will take a different approach. I am convinced that the emphasis on paradigms is the appropriate way to study parallel algorithms. I will illustrate the benefits of this approach by developing a parallel algorithm for Householder reduction from a sequential algorithm. The program transformation is completely mechanical.

2 THE ALL-PROBLEMS PROBLEM

Let A be a set of n elements:

$$A = \{a_1, a_2, \ldots, a_n\}$$

There are $(n-1)n/2$ ways to select a subset of A consisting of two elements:

$$\begin{array}{llllll}
\{a_2, a_1\} & & & & & \\
\{a_3, a_1\} & \{a_3, a_2\} & & & & \\
\{a_4, a_1\} & \{a_4, a_2\} & \{a_4, a_3\} & & & \\
\quad \cdots & \quad \cdots & \quad \cdots & \cdots & & \\
\{a_n, a_1\} & \{a_n, a_2\} & \{a_n, a_3\} & \cdots & \{a_n, a_{n-1}\}
\end{array}$$

Each subset $\{a_i, a_j\}$ can be represented by an ordered pair (a_i, a_j), where a_i and a_j are elements of A, and $1 \leq j < i \leq n$.

An all-pairs computation performs an operation $Q(a_i, a_j)$ on every pair (a_i, a_j). This operation transforms a_i and a_j without involving any other elements of A. Inspired by the n-body problem, I will say that the operation defines an "interaction" between a pair of elements.

I will consider the all-pairs computation defined by Fig. 1. In this *precedence graph*, an arrow from one operation to another indicates that the former operation must be performed before the latter in any solution to the problem. The figure shows that control flows from top to bottom and left to right.

Element a_1 interacts with $a_2, a_3, \ldots, a_n$ in that order. Element a_2 interacts with $a_1, a_3, \ldots, a_n$, and so on. Finally, element a_n interacts with $a_1, a_2, \ldots, a_{n-1}$. All operations on a particular element a_i take place strictly one at a time. There is no possibility of race conditions when the all-pairs computation is performed in parallel.

Figure 2 is a more compact representation of the precedence graph in the form of a triangular *precedence matrix*. The elements of the precedence matrix are operations. Each operation is preceded by the operations (if any) immediately above and to the left of it, and is followed by the operations (if

$$
\begin{array}{ccccccccc}
Q(a_2,a_1) & & & & & & & & \\
\downarrow & & & & & & & & \\
Q(a_3,a_1) & \rightarrow & Q(a_3,a_2) & & & & & & \\
\downarrow & & \downarrow & & & & & & \\
Q(a_4,a_1) & \rightarrow & Q(a_4,a_2) & \rightarrow & Q(a_4,a_3) & & & & \\
\downarrow & & \downarrow & & \downarrow & & & & \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & & \\
\downarrow & & \downarrow & & \downarrow & & & & \\
Q(a_n,a_1) & \rightarrow & Q(a_n,a_2) & \rightarrow & Q(a_n,a_3) & \rightarrow & \cdots & \rightarrow & Q(a_n,a_{n-1})
\end{array}
$$

Figure 1: All-pairs precedence graph.

any) immediately below and to the right of it. In other words, $Q(a_i, a_j)$ is preceded by $Q(a_{i-1}, a_j)$ and $Q(a_i, a_{j-1})$ and is followed by $Q(a_{i+1}, a_j)$ and $Q(a_i, a_{j+1})$.

$$
\begin{array}{cccccc}
Q(a_2,a_1) & & & & & \\
Q(a_3,a_1) & Q(a_3,a_2) & & & & \\
Q(a_4,a_1) & Q(a_4,a_2) & Q(a_4,a_3) & & & \\
\cdots & \cdots & \cdots & \cdots & & \\
Q(a_n,a_1) & Q(a_n,a_2) & Q(a_n,a_3) & \cdots & Q(a_n,a_{n-1}) &
\end{array}
$$

Figure 2: All-pairs precedence matrix.

3 SEQUENTIAL ALGORITHM

Figure 3 defines a sequential solution of the all-pairs problem for n elements of type T. The correctness of the algorithm is obvious when you compare it with Fig. 2. It defines the same sequence of operations as the precedence matrix, column by column, from left to right.

Example 1:

An *n-body simulation* computes the trajectories of n particles which interact through gravitational forces only. For each time step, the algorithm computes the forces between each pair of particles (a_i, a_j) and adds them

```
type table = array [1..n] of T;
var a: table; i, j: integer;
for i := 1 to n − 1 do
  for j := i + 1 to n do Q(a[j], a[i])
```

Figure 3: All-pairs algorithm.

to the total forces acting on these particles. The main loop of the force summation is programmed as follows:

```
type system = array [1..n] of body;
var a: system; i, j: integer;
for i := 1 to n − 1 do
  for j := i + 1 to n do
    [sic] addforces(a[j], a[i])
```

Force interactions are symmetric, since addforces(a_j, a_i) is equivalent to addforces(a_i, a_j). The example shows that an interaction between a pair of elements may transform both elements. For large n, the $O(n\log n)$ force calculation of Barnes and Hut (1986) is much faster than the all-pairs algorithm.

Example 2:

Gaussian elimination reduces an $n \times n$ real matrix to upper triangular form in $n-1$ steps. In the ith step, the algorithm subtracts row a_i multiplied by a_{ji}/a_{ii} from row a_j. If you ignore the (serious) rounding problems which occur when the pivot element a_{ii} is very small, you have the following loop:

```
type matrix = array [1..n] of row;
var a: matrix; i, j: integer;
for i := 1 to n − 1 do
  for j := i + 1 to n do
    subtract(i, a[j], a[i])
```

The row interactions are asymmetric: subtract(i, a_j, a_i) is not the same as subtract(j, a_i, a_j). Gaussian elimination without pivoting is numerically unstable (Press 1989). I use it only as a simple example of the all-pairs problem. Householder reduction, which will be discussed later, is numerically stable and well-suited for parallel execution.

Another sequential algorithm for the all-pairs problem is obtained by implementing the precedence matrix, row by row, from top to bottom (Fig. 4). For $i = 1$, the inner *for* statement defines an empty operation, so it makes no difference whether the initial value of i is 1 or 2.

```
var a: table; i, j: integer;
for i := 1 to n do
  for j := 1 to i − 1 do Q(a[i], a[j])
```

Figure 4: Equivalent all-pairs algorithm.

4 A COARSE-GRAIN PIPELINE

I will solve the all-pairs problem on a pipeline with p nodes, where $1 \leq p \leq n-1$ (Fig. 5). The nodes communicate by messages only. The first node inputs the original elements of A. The last node outputs the final elements of A. Without loss of generality, I assume that $n-1$ is divisible by p. Each node implements $(n-1)/p$ columns of the precedence matrix (Fig. 2).

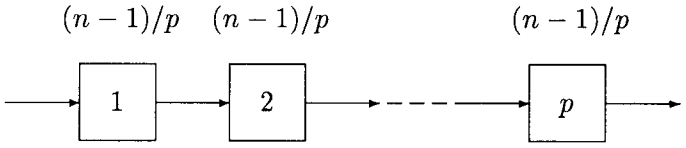

Figure 5: The all-pairs pipeline.

The pipeline can be designed to output the elements in either natural order $a_1, a_2, \ldots, a_n$, or reverse order $a_n, a_{n-1}, \ldots, a_1$. I will use *reverse output* to facilitate back substitution after matrix reduction.

I will program the pipeline nodes in *SuperPascal*. Each node has an input channel, named *left*, and an output channel, named *right*. In program assertions, a channel name denotes the sequence of messages transmitted through the channel up to that point. As an example, the assertion

$$\text{left} = < a_r..a_n > \textbf{rev} < a_1..a_{r-1} >$$

shows that a node has input the elements a_r through a_n, in that order, followed by the elements a_1 through a_{r-1} in reverse order. In other words,

$$\text{left} = < a_r, a_{r+1}, \ldots, a_n, a_{r-1}, a_{r-2}, \ldots, a_1 >$$

Some sequences are, per definition, empty:

$$< a_i..a_j > = <>, \quad \textbf{rev}< a_i..a_j > = <> \quad \text{for } i > j$$

Figure 6 shows how the precedence matrix in Fig. 2 is partitioned for an all-pairs pipeline with 2 nodes and 5 elements. An arrow in row i denotes either input of element a_i by the first node, communication of a_i from the first to the second node, or output of a_i by the second node. At the end of the computation, node 1 holds elements a_1 and a_2, node 2 stores a_3 and a_4, while a_5 has been output. The final task of the nodes is to output the stored elements in reverse order a_4, a_3, a_2, a_1.

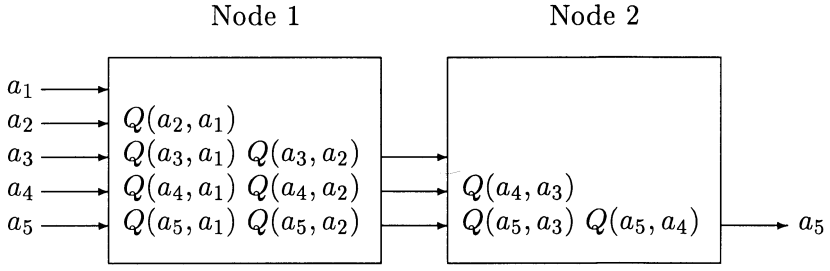

Figure 6: Precedence matrix of a pipeline.

Figure 7 shows the precedence matrix of a pipeline node that implements columns r through s of Fig. 2, where $1 \leq r \leq s \leq n-1$. This matrix enables you to develop an algorithm for a pipeline node.

A pipeline node goes through four phases:

1. *Input phase:* The node inputs elements a_r through a_s and stores them in a local array a. Every input element a_i interacts with each of the previously stored elements a_r through a_{i-1}.

```
{ left = <>, right = <> }
  for i := r to s do
    begin
      receive(left, a[i]);
      for j := r to i − 1 do Q(a[i], a[j])
    end
{ left = < a_r..a_s >, right = <> }
```

$$
\begin{array}{llllllll}
a_r & \rightarrow & & & & & & \\
a_{r+1} & \rightarrow & Q(a_{r+1}, a_r) & & & & & \\
\cdots & \cdots & \cdots & \cdots & & & & \\
a_s & \rightarrow & Q(a_s, a_r) & \cdots & Q(a_s, a_{s-1}) & & & \\
a_{s+1} & \rightarrow & Q(a_{s+1}, a_r) & \cdots & Q(a_{s+1}, a_{s-1}) & Q(a_{s+1}, a_s) & \rightarrow & a_{s+1} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
a_n & \rightarrow & Q(a_n, a_r) & \cdots & Q(a_n, a_{s-1}) & Q(a_n, a_s) & \rightarrow & a_n
\end{array}
$$

Figure 7: Precedence matrix of a pipeline node.

2. *Transfer phase:* The node inputs elements a_{s+1} through a_n. Every transfer element a_j interacts with every local element and is then immediately output to the next node. There is no room for transfer elements in the local array. They are stored temporarily in a local variable a_j. (The last node transfers element a_n only, since $s = n - 1$.) This phase completes the local computation defined by Fig. 7.

{ left = $< a_r..a_s >$, right = <> }
  for j := s + 1 **to** n **do**
    begin
      receive(left, aj);
      for i := r **to** s **do** Q(aj, a[i]);
      send(right, aj)
    end
{ left = $< a_r..a_n >$, right = $< a_{s+1}..a_n >$ }

3. *Output phase:* The node outputs the local elements in reverse order.

{ left = $< a_r..a_n >$, right = $< a_{s+1}..a_n >$ }
  for i := s **downto** r **do**
    send(right, a[i])
{ left = $< a_r..a_n >$,
  right = $< a_{s+1}..a_n >$**rev**$< a_r..a_s >$ }

4. *Copy phase:* The node copies all elements output in reverse order by the previous nodes. (The first node copies no elements since $r = 1$.)

$$
\begin{array}{l}
\{\ \text{left} = < a_r..a_n >, \\
\quad \text{right} = < a_{s+1}..a_n > \mathbf{rev} < a_r..a_s >\ \} \\
\quad \textbf{for j} := \text{r} - 1 \textbf{ downto } 1 \textbf{ do} \\
\quad\quad \textbf{begin} \\
\quad\quad\quad \text{receive(left, aj);} \\
\quad\quad\quad \text{send(right, aj)} \\
\quad\quad \textbf{end} \\
\{\ \text{left} = < a_r..a_n > \mathbf{rev} < a_1..a_{r-1} >, \\
\quad \text{right} = < a_{s+1}..a_n > \mathbf{rev} < a_1..a_s >\ \}
\end{array}
$$

Putting these program pieces together, I obtain the complete algorithm for a pipeline node (Fig. 8). To suppress irrelevant detail, I use an array type with dynamic bounds *r..s* (which does not exist in *SuperPascal*).

The algorithm does not duplicate the whole set A within each node. The first $n-1$ elements of the set are distributed evenly among the nodes of the pipeline. The last element is transferred through the pipeline without being stored.

The postcondition of the last phase shows that the input sequence of a node is a function of its lower bound r, while the output sequence is determined by the upper bound *s:*

$$
\begin{array}{c}
\text{left(r)} = < a_r..a_n > \mathbf{rev} < a_1..a_{r-1} > \\
\text{right(s)} = < a_{s+1}..a_n > \mathbf{rev} < a_1..a_s >
\end{array}
$$

This assertion implies that the first node inputs the numbers in natural order:

$$
\text{left}(1) = < a_1..a_n > \mathbf{rev} < a_1..a_0 > = < a_1..a_n >
$$

while the last node outputs them in reverse order:

$$
\text{right}(\text{n} - 1) = < a_n..a_n > \mathbf{rev} < a_1..a_{n-1} > = \mathbf{rev} < a_1..a_n >
$$

I leave it as an exercise for you to write a modified algorithm which accepts input and produces output in natural order. The key idea is to use the input/output sequences

$$
\begin{array}{c}
\text{left(r)} = < a_r..a_{n-1} >< a_1..a_{r-1} >< a_n > \\
\text{right(s)} = < a_{s+1}..a_{n-1} >< a_1..a_s >< a_n >
\end{array}
$$

```
procedure node(r, s: integer;
  left, right: channel);
type block = array [r..s] of T;
var a: block; aj: T; i, j: integer;
begin
  { 1 <= r <= s <= n − 1 }
  for i := r to s do
    begin
      receive(left, a[i]);
      for j := r to i − 1 do Q(a[i], a[j])
    end;
  for j := s + 1 to n do
    begin
      receive(left, aj);
      for i := r to s do Q(aj, a[i]);
      send(right, aj)
    end;
  for i := s downto r do
    send(right, a[i]);
  for j := r − 1 downto 1 do
    begin
      receive(left, aj);
      send(right, aj)
    end
end;
```

Figure 8: Node algorithm.

The all-pairs paradigm enables a programmer to formulate parallel versions of similar sequential algorithms by trivial substitution.

Example 3:

You can derive a pipelined algorithm for the force summation in *n-body simulation* by performing the following substitutions in Fig. 8:

type body	replaces	type T
addforces(a[i], a[j])	replaces	Q(a[i], a[j])
addforces(aj, a[i])	replaces	Q(aj, a[i])

By setting $r = 1$ and $s = n - 1$ in Fig. 8, you obtain a single-processor version of the all-pairs pipeline which is equivalent to Fig. 4.

5 A MEDIUM-GRAIN PIPELINE

A medium-grain pipeline consists of $n - 1$ nodes, each of which holds only one element of the set A. The medium-grain algorithm is derived from the coarse-grain version by setting $i = r = s$ in Fig. 8. Figure 9 defines a node that implements the ith column of the precedence matrix (Fig. 2).

```
procedure node(i: integer;
  left, right: channel);
var ai, aj: T; j: integer;
begin
  { 1 <= i <= n - 1 }
  receive(left, ai);
  for j := i + 1 to n do
    begin
      receive(left, aj);
      Q(aj, ai);
      send(right, aj)
    end;
  send(right, ai);
  for j := i - 1 downto 1 do
    begin
      receive(left, aj);
      send(right, aj)
    end
end;
```

Figure 9: Medium-grain node algorithm.

Example 4:

From a sequential algorithm for *Gaussian elimination* without pivoting, you can design a pipeline algorithm by making the following substitutions in Fig. 9:

type row	replaces	type T
subtract(i, aj, ai)	replaces	Q(aj, ai)

6 VARIATION ON A THEME

In the all-pairs computation discussed so far, each operation is an interaction between two elements of the same set

$$A = \{a_1, a_2, \ldots, a_n\}$$

In some applications, it is more convenient to use A to compute another set

$$B = \{b_1, b_2, \ldots, b_{n-1}\}$$

and let the elements of A interact with the elements of B. The set B is a temporary data structure which exists during the computation only.

Figure 10 shows the precedence matrix for this variant of the all-pairs computation.

$P(a_1, b_1)$				
$Q(a_2, b_1)$	$P(a_2, b_2)$			
$Q(a_3, b_1)$	$Q(a_3, b_2)$	$P(a_3, b_3)$		
$Q(a_4, b_1)$	$Q(a_4, b_2)$	$Q(a_4, b_3)$		
$\cdots$	$\cdots$	$\cdots$	$\cdots$	
$Q(a_{n-1}, b_1)$	$Q(a_{n-1}, b_2)$	$Q(a_{n-1}, b_3)$	$\cdots$	$P(a_{n-1}, b_{n-1})$
$Q(a_n, b_1)$	$Q(a_n, b_2)$	$Q(a_n, b_3)$	$\cdots$	$Q(a_n, b_{n-1})$

Figure 10: Variant precedence matrix.

The all-pairs variant is a computation on every set $\{a_i, b_j\}$, where a_i is a member of A, b_j is a member of B, and $j \leq i$. For each of these sets, one of two operations is performed:

1. The operation $P(a_i, b_i)$ transforms element a_i and computes the corresponding element b_i, where $1 \leq i \leq n-1$.

2. The operation $Q(a_i, b_j)$ transforms elements a_i and b_j, where $1 \leq j < i \leq n$.

From the precedence matrix, I derive a sequential algorithm (Fig. 11). In this case, each element of B exists only during a single step of the computation. So, the set B is represented by a variable b_i, which holds a single element only. This is a variant of Fig. 3.

```
var a: table; bi: T; i, j: integer;
for i := 1 to n − 1 do
  begin
    P(a[i], bi);
    for j := i + 1 to n do Q(a[j], bi)
  end
```

Figure 11: All-pairs variant.

Example 5:

Householder's method reduces an $n \times n$ real matrix to upper triangular form in $n-1$ steps. The main loop of a sequential Householder reduction is shown below (Brinch Hansen 1990a):

```
type matrix = array [1..n] of column;
var a: matrix; vi: column; i, j: integer;
for i := 1 to n − 1 do
  begin
    eliminate(i, a[i], vi);
    for j := i + 1 to n do
      transform(i, a[j], vi)
  end
```

The matrix is stored by columns, that is, a[i] denotes the ith column of A. In the ith step, the algorithm uses column a[i] to compute a column vector v_i. This vector is then used to transform each remaining column a[j], where $i+1 \leq j \leq n$. The *eliminate* and *transform* operations are defined in Case Study 10. The elements of the set A are matrix columns a_1 through a_n. The elements of the set B are column vectors v_1 through v_{n-1}. For each element a_i of A (except a_n), the algorithm computes the corresponding element v_i of B.

Figure 12 is a variant of Fig. 4 obtained from Fig. 10.

Figure 13 defines a pipeline node for the all-pairs variant. All elements of A and B (except a_n) are distributed evenly among the nodes. The elements of B are temporary local entities which are not transmitted between the nodes.

For $a = b$ and $P = empty$, the algorithm reduces to the algorithm in Fig. 8. A medium-grain version of this pipeline is similar to Fig. 9.

```
var a, b: table; i, j: integer;
for i := 1 to n − 1 do
  begin
    for j := 1 to i − 1 do Q(a[i], b[j]);
    P(a[i], b[i])
  end;
for i := 1 to n − 1 do Q(a[n], b[i])
```

Figure 12: Equivalent all-pairs variant.

7 AN EXAMPLE: HOUSEHOLDER REDUCTION

Many problems in science and engineering involve a system of n linear equations. The equations can be solved in two steps: First, the equations are reduced to triangular form by systematic elimination of unknowns. The triangular equations are then solved by back substitution.

The most time-consuming part of the computation is the reduction of the coefficient matrix to triangular form. The standard Gaussian and Gauss-Jordan eliminations are straightforward reduction algorithms. They do, however, require pivoting, a rearrangement of the rows and columns, which in most cases, prevents numerical instability (Press 1989). On a parallel computer, pivoting complicates these algorithms (Fox 1988).

For a parallel computer, Householder reduction is an attractive method that is numerically stable and does not require pivoting (Press 1989, Brinch Hansen 1990a). In the following, I derive a pipeline algorithm for Householder reduction directly from the all-pairs paradigm.

Example 5 defines the main loop of sequential Householder reduction. The theory behind Householder reduction is explained in Brinch Hansen (1990a) and will not be repeated here.

A comparison of Fig. 11 and Example 5 shows that Householder reduction is an all-pairs variant. So, you can derive a pipeline for Householder reduction by making the following substitutions in Fig. 13:

type column	replaces	type T
variable v	replaces	variable b
eliminate(i, a[i], v[i])	replaces	P(a[i], b[i])
transform(j, a[i], v[j])	replaces	Q(a[i], b[j])
transform(i, aj, v[i])	replaces	Q(aj, b[i])

Figure 14 defines a node of the Householder pipeline which holds columns

```
procedure node(r, s: integer;
  left, right: channel);
type block = array [r..s] of T;
var a, b: block; aj: T; i, j: integer;
begin
  { 1 <= r <= s <= n − 1 }
  for i := r to s do
    begin
      receive(left, a[i]);
      for j := r to i − 1 do Q(a[i], b[j]);
      P(a[i], b[i])
    end;
  for j := s + 1 to n do
    begin
      receive(left, aj);
      for i := r to s do Q(aj, b[i]);
      send(right, aj)
    end;
  for i := s downto r do
    send(right, a[i]);
  for j := r − 1 downto 1 do
    begin
      receive(left, aj);
      send(right, aj)
    end
end;
```

Figure 13: The all-pairs variant.

r through s, where $1 \leq r \leq s \leq n-1$. The pipeline inputs the columns in natural order, reduces the matrix to triangular form, and outputs the final columns in reverse order. The performance of the parallel algorithm has been analyzed and measured on a Computing Surface (Brinch Hansen 1990b).

The parallel Householder algorithm is an ideal algorithm for experimenting with a parallel computer:

1. It is a fundamental algorithm of considerable practical value.

```
procedure node(r, s: integer;
  left, right: channel);
type block = array [r..s] of column;
var a, v: block; aj: column;
  i, j: integer;
begin
  { 1 <= r <= s <= n − 1 }
  for i := r to s do
    begin
      receive(left, a[i]);
      for j := r to i − 1 do
        transform(j, a[i], v[j]);
      eliminate(i, a[i], v[i])
    end;
  for j := s + 1 to n do
    begin
      receive(left, aj);
      for i := r to s do
        transform(i, aj, v[i]);
      send(right, aj)
    end;
  for i := s downto r do
    send(right, a[i]);
  for j := r − 1 downto 1 do
    begin
      receive(left, aj);
      send(right, aj)
    end
end;
```

Figure 14: Householder node algorithm.

2. It demonstrates the use of a general paradigm to transform a sequential algorithm into a parallel one.

3. It illustrates the subtleties of distributing a large computation evenly among parallel processors.

8 FINAL REMARKS

After programming n-body simulation and Householder reduction in occam for the Computing Surface, I was delighted to discover that these seemingly unrelated problems can be solved by refinements of the same abstract program.

I have presented pipeline algorithms for two variants of the all-pairs paradigm. As a non-trivial example, I have used the paradigm to derive a pipeline algorithm for Householder reduction of a real matrix to triangular form. The parallel algorithm was derived from a sequential one by trivial substitution of data types, variables, and procedure statements.

References

Barnes, J. and Hut, P. (1986) A hierarchical $O(NlogN)$ force-calculation algorithm. *Nature 324*, 446–449.

Brinch Hansen, P. (1990a) Householder reduction of linear equations. School of Computer and Information Science, Syracuse University, Syracuse, NY. Also in *ACM Computing Surveys 24*, 185–194, June 1992. Also in Brinch Hansen (1995).

Brinch Hansen, P. (1990b) Balancing a pipeline by folding. School of Computer and Information Science, Syracuse University, Syracuse, NY. Revised version in Brinch Hansen (1995).

Brinch Hansen, P. (1991) The n-body pipeline. School of Computer and Information Science, Syracuse University, Syracuse, NY. Revised version in Brinch Hansen (1995).

Brinch Hansen, P. (1995) *Studies in Computational Science*. Prentice Hall, Englewood cliffs, NJ (March).

Cosnard, M. and Tchuente, M. (1988), Designing systolic algorithms by top-down analysis. *The Third International Conference on Supercomputing*, Vol. 3, International Supercomputing Institute, St. Petersburg, FL, 9–18.

Fox, G.C., Johnson, M.A., Lyzenga, G.A., Otto, S.W., Salmon, J.K. and Walker, D.W. (1988) *Solving Problems on Concurrent Processors*, Vol. I, Prentice-Hall, Englewood Cliffs, NJ.

Ortega, J.M. (1988) *Introduction to Parallel and Vector Solutions of Linear Systems*. Plenum Press, New York.

Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T. (1989) *Numerical Recipes in Pascal: The Art of Scientific Computing*. Cambridge University Press, Cambridge, MA.

Shih, Z., Chen, G. and Lee, R.T.C. (1987) Systolic algorithms to examine all pairs of elements. *Communications of the ACM 30*, 161–167.