THE ALL-PAIRS PIPELINE

Per Brinch Hansen

Syracuse University, School of Computer and Information Science, pbh@top.cis.syr.edu
THE ALL-PAIRS PIPELINE

PER BRINCH HANSEN

December 1990

School of Computer and Information Science
Suite 4-116
Center for Science and Technology
Syracuse, New York 13244-4100

(315) 443-2368
THE ALL-PAIRS PIPELINE¹

PER BRINCH HANSEN

School of Computer and Information Science
Syracuse University
Syracuse, New York 13244

December 1990

Abstract—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. The paper 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.

Index Terms—All-pairs paradigm, Householder reduction, Precedence matrices, Pipelined computers.

I. 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 [1]. Householder reduction of a matrix to triangular form is a less obvious example [2], [3]. This paper develops the all-pairs paradigm discussed in [4], [5]. We 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 reduction, and Householder reduction [6].

¹Copyright©1990 Per Brinch Hansen
We will take a different approach. We are convinced that the emphasis on paradigms is the appropriate way to study parallel algorithms. We will illustrate the benefits of this approach by deriving a parallel algorithm for Householder reduction from a sequential algorithm. The program transformation is completely mechanical and requires no understanding of Householder’s method.

II. THE ALL-PAIRS 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:

$$\begin{align*}
\{a_2, a_1\} \\
\{a_3, a_1\} & \{a_3, a_2\} \\
\{a_4, a_1\} & \{a_4, a_2\} & \{a_4, a_3\} \\
\vdots & \vdots & \vdots & \vdots \\
\{a_n, a_1\} & \{a_n, a_2\} & \{a_n, a_3\} \ldots & \{a_n, a_{n-1}\}
\end{align*}$$

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 we will say that the operation defines an “interaction” between a pair of elements.

We 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 racing conditions when the all-pairs computation is performed in parallel.
Fig. 1 All-pairs precedence graph.

Fig. 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 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 \( Q(a_i, a_{j+1}) \).

III. SEQUENTIAL ALGORITHMS

Algorithm 1 defines a sequential solution of the all-pairs problem for \( n \) elements of type \( T \).

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

Algorithm 1
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 to the total forces acting on these particles. The main loop of the force summation is programmed as follows

```plaintext
var a: array [1..n] of body;
i, j integer;
for i := 1 to n - 1 do
  for j := i + 1 to n do
    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 [7] is much faster than the all-pairs algorithm. (*End of example.*)

**Example 2.**

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

```plaintext
var a: array [1..n] of row;
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 [2]. We 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. (*End of example.*)

Another sequential algorithm for the all-pairs problem is obtained by implementing the precedence matrix, row by row, from top to bottom (Algorithm 2). 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.
THE ALL-PAIRS PIPELINE

var a: array [1..n] of T; i, j: integer;
for i := 1 to n do
    for j := 1 to i - 1 do Q(a[i], a[j])

Algorithm 2

IV. A COARSE-GRAIN PIPELINE

We will solve the all-pairs problem on a pipeline with p nodes, where 1 ≤ p ≤ n - 1 (Fig. 3). The nodes communicate by messages only. The first node inputs the original elements of A. The last node outputs the final elements of A.

![Fig. 3 The all-pairs pipeline.](image)

Without loss of generality we assume that n - 1 is divisible by p. Each node implements (n - 1)/p columns of the matrix (Fig. 2).

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

We will program the pipeline nodes in Pascal extended with statements for message communication. Each node has an input channel and an output channel. The input and output of an element $a_i$ are denoted

inp?$a_i$  out!$a_i$

In program assertions, a channel name denotes the sequence of elements transmitted through the channel so far. As an example, the assertion

inp = < $a_r..a_n$ > 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,

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

Some sequences are empty

< $a_i..a_j$ > = < >,  rev < $a_i..a_j$ > = < >  for $i > j$
Fig. 4 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 $a_4, a_3, a_2, a_1$.

Fig. 4 Precedence matrix of a pipeline.

Fig. 5 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 us to develop an algorithm for a pipeline node.

```
in?a_r    Q(a_r+1, a_r)
inp?a_{r+1}    Q(a_{r+1}, a_r)
   . . .
inp?a_s    Q(a_s, a_r) . . . Q(a_s, a_{s-1})
inp?a_{s+1}    Q(a_{s+1}, a_r) . . . Q(a_{s+1}, a_{s-1}) Q(a_{s+1}, a_s) out!a_{s+1}
   . . .
inp?a_n    Q(a_n, a_r) . . . Q(a_n, a_{s-1}) Q(a_n, a_s) out!a_n
```

Fig. 5 Precedence matrix of 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 $a_{i-1}$.
{ inp = <>, out = <> }  
for i := r to s do 
begin 
inp?a[i]; 
for j := r to i - 1 do Q(a[i], a[j]) 
end  
{ inp = < a_r..a_s >, out = <> } 

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. 5.

{ inp = < a_r..a_s >, out = <> }  
for j := s + 1 to n do 
begin 
inp?aj; 
for i := r to s do Q(aj, a[i]); 
out!aj 
end  
{ inp = < a_r..a_n >, out =< a_{s+1}..a_n > } 

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

{ inp = < a_r..a_n >, out = < a_{s+1}..a_n > }  
for i := s downto r do out!a[i] 
{ inp =< a_r..a_n >, 
out =< 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 $r = 1$.)

{ inp = < a_r..a_n >, 
out =< a_{s+1}..a_n > rev < a_r..a_s > }  
for j := r - 1 downto 1 do 
begin inp?aj; out!aj end  
{ inp =< a_r..a_n > rev< a_1..a_{r-1} >, 
out =< a_{s+1}..a_n >rev< a_1..a_s > } 

Putting these program pieces together we obtain the complete algorithm for a pipeline node (Algorithm 3). To suppress irrelevant detail we use an array with dynamic bounds $r..s$ (which does not exist in Pascal).
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.

```
procedure node(r, s: integer; inp, out: channel);
  var a: array [r..s] of T; aj: T;
  i, j: integer;
  begin { 1 ≤ r ≤ s ≤ n - 1 }
    for i := r to s - 1 do
      begin
        inp?a[i];
        for j := r to i - 1 do Q(a[i], a[j])
      end;
    for j := s + 1 to n do
      begin
        inp?aj;
        for i := r to s do Q(aj, a[i]);
        out!aj
      end;
    for i := s downto r do out!a[i];
    for j := r - 1 downto 1 do
      begin
        inp?aj; out!aj
      end
  end
```

Algorithm 3

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 $s$

- $\text{inp}(r) = \langle a_r..a_n \rangle \text{rev} \langle a_1..a_{r-1} \rangle$
- $\text{out}(s) = \langle a_{s+1}..a_n \rangle \text{rev} \langle a_1..a_s \rangle$

This assertion implies that the first node inputs the elements in natural order

- $\text{inp}(1) = \langle a_1..a_n \rangle \text{rev} \langle a_1..a_0 \rangle = \langle a_1..a_n \rangle$

while the last node outputs them in reverse order

- $\text{out}(n - 1) = \langle a_n..a_n \rangle \text{rev} \langle a_1..a_{n-1} \rangle = \text{rev} \langle a_1..a_n \rangle$

We leave it as an exercise for the reader to write a modified algorithm which accepts input and produces output in natural order. The key idea is to use the input/output sequences
The all-pairs paradigm enables a programmer to formulate parallel versions of similar sequential algorithms by trivial substitution.

**Example 3.**

We can derive a pipelined algorithm for the force summation in \(N\)-body simulation by performing the following substitutions in Algorithm 3

- \(\text{type \ body} \quad \text{replaces} \quad \text{type \ T} \)
- \(\text{addforces}(a[i], a[j]) \quad \text{replaces} \quad Q(a[i], a[j]) \)
- \(\text{addforces}(a[j], a[i]) \quad \text{replaces} \quad Q(a[j], a[i]) \)

(End of example.)

By setting \(r = 1\) and \(s = n-1\) in Algorithm 3 we obtain a single-processor version of the all-pairs pipeline which is equivalent to Algorithm 2.

**V. A MEDIUM-GRAIN PIPELINE**

A medium-grain pipeline consists of \(n - 1\) nodes, each of which holds one element only of the set \(A\). The medium-grain algorithm is derived from the coarse-grain version by setting \(i = r = s\) in Algorithm 3. Algorithm 4 defines a node that implements the \(i^{th}\) column of the precedence matrix (Fig. 2).

```plaintext
procedure node(i: integer; inp, out: channel);
var ai, aj: T; j: integer;
begin { 1 \leq i \leq n - 1 }
inp?ai;
for j := i + 1 to n do
begin
inp?aj; Q(aj, ai); out!aj
end;
out!ai;
for j := i - 1 downto 1 do
begin inp?aj; out!aj end
end
```

Algorithm 4
**Example 4.**

From a sequential algorithm for Gaussian elimination without pivoting, we can design a pipeline algorithm by making the following substitutions in Algorithm 4.

\[
\begin{align*}
\text{type row} & \quad \text{replaces} \quad \text{type } T \\
\text{subtract}(i, a_j, a_i) & \quad \text{replaces} \quad Q(a_j, a_i)
\end{align*}
\]

*(End of example.)*

**VI. 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 6 shows the precedence matrix for this variant of the all-pairs computation.

\[
\begin{array}{cccc}
P(a_1, b_1) & P(a_2, b_2) & & \\
Q(a_2, b_1) & Q(a_3, b_1) & P(a_3, b_3) & \\
Q(a_3, b_1) & Q(a_4, b_1) & Q(a_4, b_3) & \\
& & & \\
& \cdots & \cdots & \cdots \\
& Q(a_{n-1}, b_1) & Q(a_{n-1}, b_2) & Q(a_{n-1}, b_3) \cdots P(a_{n-1}, b_{n-1}) \\
& Q(a_n, b_1) & Q(a_n, b_2) & Q(a_n, b_3) \cdots Q(a_n, b_{n-1}) \\
\end{array}
\]

Fig. 6 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 we derive the sequential Algorithm 5. 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 Algorithm 1.

```pascal
var a: array [1..n] of T; 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
```

Algorithm 5

**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 [3]. The matrix is stored by columns, that is, $a[i]$ denotes the $i^{th}$ column of $A$. In the $i^{th}$ step the algorithm uses column $a[i]$ to compute a column vector $v_i$. This vector is then used to transform each remaining columns $a[j]$, where $i + 1 \leq j \leq n$. The eliminate and transform operations will be defined later.

```pascal
var a: array [1..n] of column;
  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 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 $B$. *End of example.*
Algorithm 6 is a variant of Algorithm 2 obtained from Fig. 6.

\begin{verbatim}
var a: array [1..n] of T;
b: array [1..n-1] of T;
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[i], b[i])
\end{verbatim}

Algorithm 6

Algorithm 7 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 entities which are not transmitted between nodes.

\begin{verbatim}
procedure node(r, s: integer; inp, out: channel);
var a, b: array [r..s] of T; aj: T;
i, j: integer;
begin { 1 \leq r \leq s \leq n - 1 }
  for i := r to s do
  begin
    inp?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
    inp?aj;
    for i := r to s do Q(aj, b[i]);
    out!aj
  end;
  for i := s downto r do out!a[i];
  for j := r - 1 downto 1 do
  begin inp?aj; out!aj end
end
\end{verbatim}

Algorithm 7

For $a = b$ and $P =$ empty, the algorithm reduces to Algorithm 3. A medium-grain version of this pipeline is similar to Algorithm 4.
VII. 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 a 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 [2]. On a parallel computer pivoting complicates these algorithms [1].

For a parallel computer, Householder reduction is an attractive method which is numerically stable and does not require pivoting [2], [3]. In the following we derive a pipeline algorithm for Householder reduction directly from the all-pairs paradigm.

Example 5 defines the main loop of sequential Householder reduction. Since this is a fundamental numerical method, we will present the complete algorithm. The theory behind Householder reduction is explained in [3] and will not be repeated here.

The algorithm performs two kinds of operations on columns, where

\[
\text{type column} = \text{array [1..n] of real}
\]

The eliminate operation derives a column vector \( v_i \) from a column \( a_i \) and reduces \( a_i \) to a form that has all zeros below the diagonal element \( a_{ii} \) (Algorithm 8).
procedure eliminate(i: integer;
  var ai, vi: column);
var anorm, dii, fi, wii: real;
  k: integer;
begin
  anorm :=
    sqrt(product(i, ai, ai));
  if ai[i] > 0.0
    then dii := -anorm
    else dii := anorm;
  wii := ai[i] - dii;
  fi := sqrt(-2.0*wii*dii);
  vi[i] := wii/fi;
  ai[i] := dii;
  for k := i + 1 to n do
    begin
      vi[k] := ai[k]/fi;
      ai[k] := 0.0
    end
end

Algorithm 8

The transform operation uses a column vector \( v_i \) to transform a column \( a_j \) (Algorithm 9).

procedure transform(i: integer;
  var aj, vi: column);
var fi: real; k: integer;
begin
  fi := 2.0*product(i, vi, aj);
  for k := i to n do
    aj[k] := aj[k] - fi*vi[k]
end

Algorithm 9
Algorithm 10 computes the scalar product of two column vectors \( a \) and \( b \) of length \( n - i + 1 \).

\[
\text{function } \text{product}(i: \text{integer}; \\
\quad \text{var } a, b: \text{column}): \text{real}; \\
\quad \text{var } ab: \text{real}; k: \text{integer}; \\
\begin{align*}
\text{begin} \\
\quad ab := 0.0; \\
\quad \text{for } k := i \text{ to } n \text{ do} \\
\quad \quad ab := ab + a[k] * b[k]; \\
\quad \text{product} := ab \\
\end{align*}
\]

Algorithm 10

A comparison of Algorithm 5 and Example 5 shows that Householder reduction is an all-pairs variant. So we can derive a pipeline for Householder reduction by making the following substitutions in Algorithm 7:

<table>
<thead>
<tr>
<th>Type Operation</th>
<th>Original</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type column</td>
<td>replaces type ( T )</td>
<td></td>
</tr>
<tr>
<td>Variable ( v )</td>
<td>replaces variable ( b )</td>
<td></td>
</tr>
<tr>
<td>Eliminate ((i, a[i], v[i]))</td>
<td>replaces ( P(a[i], b[i]) )</td>
<td></td>
</tr>
<tr>
<td>Transform ((j, a[i], v[j]))</td>
<td>replaces ( Q(a[i], b[j]) )</td>
<td></td>
</tr>
<tr>
<td>Transform ((i, a[j], v[i]))</td>
<td>replaces ( Q(a[j], b[i]) )</td>
<td></td>
</tr>
</tbody>
</table>

Algorithm 11 defines a node of the Householder pipeline which holds columns \( r \) 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 [8].

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

1) It is a fundamental algorithm of considerable practical value.
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].
procedure node(r, s: integer; inp, out: channel);
var a, v: array [r..s] of column;
aj: column; i, j: integer;
begin { 1 ≤ r ≤ s ≤ n - 1 }
for i := r to s do
begin
inp?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
inp?aj;
for i := r to s do
transform(i, aj, v[i]);
out!aj
end;
for i := s downto r do out!a[i];
for j := r - 1 downto 1 do
begin inp?aj; out!aj end
end

Algorithm 11

VIII. FINAL REMARKS

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

We have presented pipeline algorithms for two variants of the all-pairs paradigm. As a non-trivial example we 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.

ACKNOWLEDGEMENT

I am grateful to Nawal Copty and Jonathan Greenfield for helpful comments on previous drafts of this paper.
REFERENCES


