7-1991

A Generic Multiplication Pipeline

Per Brinch Hansen
Syracuse University, School of Computer and Information Science, pbh@top.cis.syr.edu

SU-CIS-91-21

A Generic Multiplication Pipeline

Per Brinch Hansen

July 1991

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

A GENERIC MULTIPLICATION PIPELINE [1]

PER BRINCH HANSEN

School of Computer and Information Science
Syracuse University
Syracuse, New York 13244-4100

SUMMARY

This paper illustrates the benefits of developing generic algorithms for parallel programming paradigms which can be adapted to different applications. We consider a combinatorial problem called tuple multiplication. This paradigm includes matrix multiplication and the all-pairs shortest paths problem as special cases. We develop a generic pipeline for tuple multiplication. From the generic algorithm we derive pipelines for matrix multiplication and shortest paths computation by making substitutions of data types and functions. The performance of the matrix multiplication pipeline is analyzed and measured on a Computing Surface.

KEY WORDS Parallel algorithms, Programming paradigms, Generic algorithms, Tuple multiplication, Matrix multiplication, All-pairs shortest paths

INTRODUCTION

This paper illustrates the benefits of developing generic algorithms for parallel programming paradigms which can be adapted to different applications [1].

We consider a programming paradigm for a combinatorial problem which we call *tuple multiplication.* The paradigm includes *matrix multiplication* and the *all-pairs shortest paths* problem as special cases.

We develop a generic pipeline algorithm for tuple multiplication. From the generic algorithm we derive pipelines for matrix multiplication and shortest paths computation by making trivial substitutions of data types and functions.

Arthur Cayley is generally credited with having invented matrix multiplication [2]. The origin of the multiplication algorithm for the all-pairs shortest paths problem is uncertain [3]. The analogy between matrix multiplication and path problems is discussed in [4, 5]. Pipelined matrix multiplication is described in [6].

On a Computing Surface with 35 transputers the multiplication pipeline computes the product of two 1400 × 1400 real matrices in 345 s with a processor efficiency of 89%.

TUPLE MULTIPLICATION

Consider two finite tuples a and b. We will simplify the discussion a bit by assuming that a and b are n-tuples with elements of the same type T

$$a = (a_1, a_2, \ldots, a_n)$$
$$b = (b_1, b_2, \ldots, b_n)$$

A product of tuples a and b is an $n \times n$ matrix c obtained by applying the same function f to every ordered pair (a_i, b_j) consisting of an element a_i of tuple a and an element b_j of tuple b

$$c = \begin{bmatrix} f(a_1,b_1) & f(a_1,b_2) & \ldots & f(a_1,b_n) \\ f(a_2,b_1) & f(a_2,b_2) & \ldots & f(a_2,b_n) \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ f(a_n,b_1) & f(a_n,b_2) & \ldots & f(a_n,b_n) \end{bmatrix}$$

Every matrix element $c_{ij} = f(a_i, b_j)$. Without loss of generality we assume that the function f maps two elements of type T into a value of type *real*.

Algorithm 1 defines sequential tuple multiplication in Pascal.

```
type tuple = array [1..n] of T;
  vector = array [1..n] of real;
  matrix = array [1..n] of vector;

procedure multiply(a, b: tuple; var c: matrix);
var i, j: integer;
  function f(ai, bj: T): real;
  begin ... end;
begin
  for i := 1 to n do
    for j := 1 to n do
      c[i,j] := f(a[i], b[j])
end
```

Algorithm 1

PIPELINE ALGORITHM

We will multiply two n-tuples a and b on a pipeline controlled by a master process (Fig. 1).

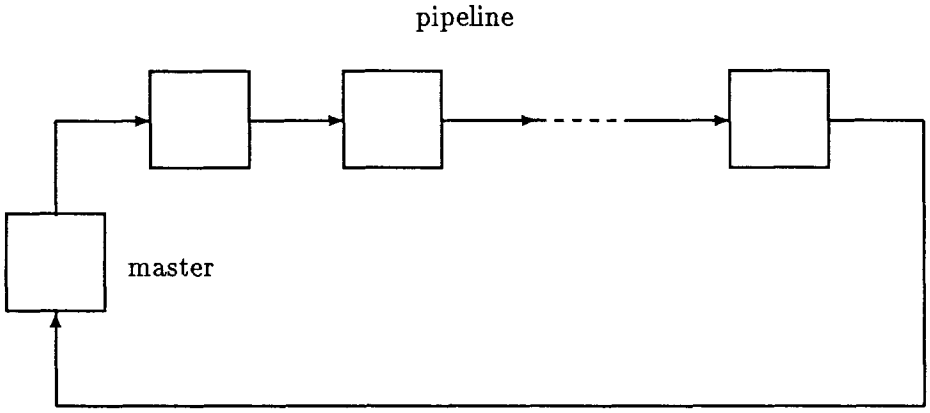

Fig. 1 Master and pipeline.

First, the elements of tuple a are distributed evenly among the nodes of the pipeline. Then the elements of tuple b pass through the pipeline, while each node computes a portion of the tuple product c. Finally, the product matrix is output by the pipeline.

The parallel processes will be defined in Pascal extended with statements for message communication. Each process has an input channel and an output channel. The input and output of an element a_i are denoted

$$\text{inp}?a_i \qquad \text{out}!a_i$$

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

$$\text{inp} = <a_r..a_n><b_1..b_n>$$

shows that a process has input elements a_r through a_n followed by b_1 through b_n, in that order.

Some sequences are empty

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

The master process executes Algorithm 2. The master outputs tuples a and b, one element at a time, to the pipeline and inputs matrix c, one row at a time, from the pipeline.

```
procedure multiply(a, b: tuple; var c: matrix;
  inp, out: channel);
var i, j: integer;
begin
  { inp = <>, out = <> }
  for i := 1 to n do out!a[i];
  for j := 1 to n do out!b[j];
  for i := 1 to n do inp?c[i];
  { inp = < c1..cn >, out =< a1..an >< b1..bn > }
end
```

Algorithm 2

Consider now a pipeline node which holds elements r through s of tuple a and rows r through s of matrix c, where $1 \leq r \leq s \leq n$.

The node goes through five phases:

1) *Input phase*: The node inputs tuple elements a_r through a_s and stores them in a local array a.

```
{ inp = <>, out = <> }
for i := r to s do inp?a[i]
{ inp = <ar..as>, out = <> }
```

2) *Transfer phase*: The node inputs tuple elements a_{s+1} through a_n and outputs them to the next node. There is no room for transfer elements in the local array a. They are stored temporarily in a local variable a_i. This phase completes the distribution of tuple a among the pipeline nodes.

```
{ inp = <ar..as>, out = <> }
for i := s + 1 to n do
begin inp?ai; out!ai end
{ inp = <ar..an>, out = <as+1..an> }
```

3) *Multiplication phase*: The node inputs tuple elements b_1 through b_n and outputs them to the next node (if any). Every input element b_j is combined with each of the local elements a_r through a_s to compute the j^{th} elements of product rows c_r through c_s. These rows are stored in a local matrix c. This phase completes the computation of the tuple product.

```
{ inp = <a_r..a_n>, out = <a_{s+1}..a_n> }
for j := 1 to n do
begin inp?bj;
  if s < n then out!bj;
  for i := r to s do
    c[i,j] := f(a[i], bj)
end
{ inp = <a_r..a_n><b_1..b_n>,
  out = <a_{s+1}..a_n><b_1..b_m> }
```

where

$$m = n \quad \text{for } s < n$$
$$m = 0 \quad \text{for } s = n$$

Since the output sequence $<a_{n+1}..a_n><b_1..b_0>$ is empty, the last node does not output any elements of tuples a and b.

4) *Copy phase*: The node copies all rows of the tuple product output by the previous nodes using a local variable c_i

```
{ inp = <a_r..a_n><b_1..b_n>,
  out = <a_{s+1}..a_n><b_1..b_m> }
for i := 1 to r - 1 do
begin inp?ci; out!ci end
{ inp = <a_r..a_n><b_1..b_n><c_1..c_{r-1}>,
  out = <a_{s+1}..a_n><b_1..b_m><c_1..c_{r-1}> }
```

5) *Output phase*: The node outputs the local portion of the tuple product. This phase completes the output of the product.

```
{ inp = <a_r..a_n><b_1..b_n><c_1..c_{r-1}>,
  out = <a_{s+1}..a_n><b_1..b_m><c_1..c_{r-1}> }
for i := r to s do out!c[i]
{ inp = <a_r..a_n><b_1..b_n><c_1..c_{r-1}>,
  out = <a_{s+1}..a_n><b_1..b_m><c_1..c_s> }
```

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

```
procedure node(r, s: integer; inp, out: channel);
var ai, bj: T; ci: vector; i, j: integer;
  a: array [r..s] of T;
  c: array [r..s] of vector;
  function f(ai, bj: T): real;
  begin ... end;
begin { 1 ≤ r ≤ s ≤ n }
  for i := r to s do inp?a[i];
  for i := s + 1 to n do
  begin inp?ai; out!ai end;
  for j := 1 to n do
  begin inp?bj;
    if s < n then out!bj;
    for i := r to s do
      c[i,j] := f(a[i], bj)
  end;
  for i := 1 to r − 1 do
  begin inp?ci; out!ci end;
  for i := r to s do out!c[i]
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

$$\text{inp(r)} = <a_r..a_n><b_1..b_n><c_1..c_{r-1}>$$
$$\text{out(s)} = <a_{s+1}..a_n><b_1..b_m><c_1..c_s>$$

This assertion implies that the first node inputs the elements of a and b, while the last node outputs the rows of c in their natural order

$$\text{inp(1)} = <a_1..a_n><b_1..b_n><c_1..c_0> = <a_1..a_n><b_1..b_n>$$
$$\text{out(n)} = <a_{n+1}..a_n><b_1..b_0><c_1..c_n> = <c_1..c_n>$$

This matches the final assertion of the master process (see Algorithm 2).

MATRIX MULTIPLICATION

Algorithm 4 defines sequential multiplication

$$c := a \times b$$

of two $n \times n$ real matrices a and b. The algorithm assumes that a and c are stored by rows, while b is stored by columns. The function f computes the dot product of a row a_i and a column b_j.

The multiplication time is $O(n^3)$.

```
procedure multiply(a, b: matrix; var c: matrix);
var i, j: integer;
  function f(ai, bj: vector): real;
  var cij: real; k: integer;
  begin cij := 0.0;
    for k := 1 to n do
      cij := cij + ai[k]*bj[k];
    f := cij
  end;
begin
  for i := 1 to n do
    for j := 1 to n do
      c[i,j] := f(a[i], b[j])
end
```

Algorithm 4

The value parameters a and b denote local copies of actual matrices. It is, of course, impractical to pass large matrices by value. We do it for pedagogical reasons only to ensure that the sequential algorithm has the same semantics as the parallel algorithm which will be described shortly.

The parameter declarations permit us to use the same array as both a value and a variable parameter in the same multiplication. As an example, the multiplication

$$\text{multiply}(d, a, d)$$

of two matrices d and a is equivalent to the assignment

$$d := d * a$$

Matrix multiplication is a tuple multiplication: it can be obtained by making the following type substitutions in Algorithm 1

vector	replaces	T
matrix	replaces	tuple

Consequently, we can derive a pipeline for matrix multiplication by making the same substitutions in Algorithms 2 and 3. This leads to Algorithms 5 and 6.

```
procedure multiply(var a, b, c: matrix;
  inp, out: channel);
var i, j: integer; bj: vector;
begin
  for i := 1 to n do out!a[i];
  for j := 1 to n do
  begin
    for i := 1 to n do bj[i] := b[i,j];
    out!bj
  end;
  for i := 1 to n do inp?c[i]
end
```

Algorithm 5

Two minor refinements have been added to the multiplication procedure:

1) All three matrices are declared as variable parameters. However, the procedure treats a and b as value parameters by making local copies of these matrices in the pipeline before assigning their product to c.

2) All three matrices are stored by rows. While the pipeline processes a column of b, the master process (which runs on a separate processor) simultaneously unpacks the next column.

```
procedure node(r, s: integer; inp, out: channel);
var ai, bj, ci: vector; i, j: integer;
  a, c: array [r..s] of vector;
  function f(ai, bj: vector): real;
  begin ... end;
begin { 1 ≤ r ≤ s ≤ n }
  for i := r to s do inp?a[i];
  for i := s + 1 to n do
  begin inp?ai; out!ai end;
  for j := 1 to n do
  begin inp?bj;
    if s < n then out!bj;
    for i := r to s do
      c[i,j] := f(a[i], bj)
  end;
  for i := 1 to r − 1 do
  begin inp?ci; out!ci end;
  for i := r to s do out!c[i]
end
```

Algorithm 6

ALL-PAIRS SHORTEST PATHS

As a second example of tuple multiplication we will compute the shortest paths between every pair of nodes in a directed graph with n nodes. We assume that every edge has length 1.

The graph is represented by an $n \times n$ adjacency matrix a. A matrix element a_{ij} defines the length of the edge (if any) from node i to node j

$a_{ii} = 0$	for every node i
$a_{ij} = 1$	if there is an edge from node i to node j
$a_{ij} = \infty$	if there is no edge from node i to node j

Figure 2 shows a directed graph with four nodes labeled 1, 2, 3, and 4.

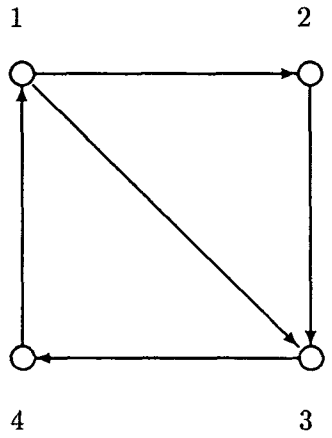

Fig. 2 A directed graph

The adjacency matrix of this graph is

$$a = \begin{bmatrix} 0 & 1 & 1 & \infty \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & 1 \\ 1 & \infty & \infty & 0 \end{bmatrix}$$

Our goal is to compute an $n \times n$ distance matrix d. A matrix element d_{ij} defines the shortest path from node i to node j. If there is no path, then $d_{ij} = \infty$.

If you follow the shortest path from one node to another, you may have to visit each of the other $n - 1$ nodes, but not more than once. (Otherwise, the path is not the shortest one.) Consequently, the shortest path (if any) always has fewer than n edges.

We will compute a sequence of distance matrices

$$d^{(1)},\ d^{(2)}, \ldots, d^{(n-1)}$$

The first matrix defines all shortest paths of 1 or 0 edges. We will express this assertion as follows

$$d_{ij}^{(1)} = \text{shortest}(i, j, 1)$$

The second matrix defines all shortest paths of 2 (or fewer) edges

$$d_{ij}^{(2)} = \text{shortest}(i, j, 2)$$

and so on.

The $(n-1)^{\text{th}}$ matrix defines all shortest paths of $n-1$ (or fewer) edges

$$d_{ij}^{(n-1)} = \text{shortest}(i, j, n-1)$$

The first distance matrix is the adjacency matrix

$$d^{(1)} = a$$

Since every shortest finite path has fewer than n edges, the $(n-1)^{\text{th}}$ matrix is the distance matrix we are looking for.

Suppose we already have computed the m^{th} matrix

$$d = d^{(m)}$$

where $1 \leq m < n-1$. How then can we transform d into the $(m+1)^{\text{st}}$ matrix in the sequence?

For any two nodes i and j we know the shortest path of m (or fewer) edges

$$d_{ij} = \text{shortest}(i, j, m)$$

However, it might be possible to find a shorter path from i to j of $m+1$ (or fewer) edges by going through a third node k (Fig. 3).

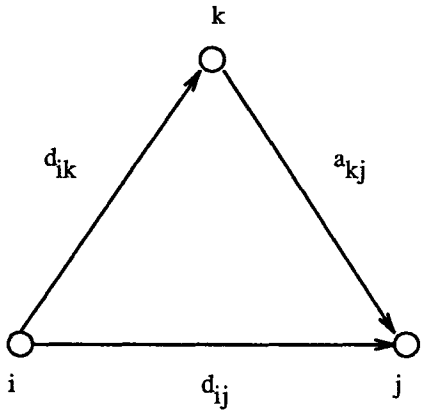

Fig. 3 Three nodes

For any intermediate node k we already know the distances

$$d_{ik} = \text{shortest}(i, k, m)$$
$$a_{kj} = \text{shortest}(k, j, 1)$$

If $d_{ij} > d_{ik} + a_{kj}$ the alternative path through k is shorter than the previous shortest distance from i to j. (Since all edges are of length 1, a shorter alternative path exists only if $d_{ij} = \infty$. Later, we will consider weighted graphs with edges of arbitrary lengths. In that case, it may be possible to replace a *finite* distance d_{ij} with a shorter distance from i to j.)

The following loop attempts to reduce d_{ij} by examining an alternative path through every node

```
for k := 1 to n do
  if d[i,j] > d[i,k] + a[k,j] then
    d[i,j] := d[i,k] + a[k,j]
```

At the end of this loop we have found the shortest path from i to j of $m + 1$ (or fewer) edges

$$d_{ij} = \text{shortest}(i, j, m + 1)$$

For $k = j$ the "alternative" path computed by the loop is the previous shortest distance d_{ij} since

$$d_{ik} + a_{kj} = d_{ij} + a_{jj} = d_{ij} \quad \text{for } k = j$$

Consequently, the loop can be replaced by an equivalent computation of the shortest alternative path from i to j

$$d_{ij} = \min(d_{ik} + a_{kj}) \quad \text{for all } k = 1..n$$

We must perform the same computation for every pair of nodes to find all shortest paths of $m + 1$ (or fewer) edges. We will show that this can be done by a tuple multiplication of the form

$$d := d * a$$

expressed as follows in Pascal

$$\text{multiply}(d, a, d)$$

The multiplication procedure (Algorithm 7) is similar to matrix multiplication. The function f computes the shortest alternative path from node i to node j. The algorithm uses two obvious functions to compute the minimum and the sum of two reals. The sum function ensures that addition handles infinity (represented by a large constant) correctly.

```
procedure multiply(a, b: matrix; var c: matrix);
var i, j: integer;
  function f(ai, bj: vector): real;
  var cij: real; k: integer;
  begin cij := infinity;
    for k := 1 to n do
      cij := min(cij, sum(ai[k], bj[k]));
    f := cij
  end;
begin
  for i := 1 to n do
    for j := 1 to n do
      c[i,j] := f(a[i], b[j])
end
```

Algorithm 7

The algorithm can be derived from Algorithm 4 by making the following substitutions in function f

infinity	replaces	0.0
min	replaces	+
sum	replaces	*

From this analysis we conclude that the all-pairs shortest paths problem is solved by a sequence of tuple multiplications

$$
\begin{aligned}
&d^{(1)} = a \\
&d^{(2)} = d^{(1)} \times a = a^2 \\
&\cdots \\
&d^{(n-1)} = d^{(n-2)} \times a = a^{n-1}
\end{aligned}
$$

The computation is defined by Algorithm 8.

```
procedure allpaths(var a, d: matrix);
var m: integer;
begin d := a;
  for m := 2 to n - 1 do
    multiply(d, a, d)
end
```

Algorithm 8

The computing time is $O(n^4)$.

For the graph in Fig. 2 the computation proceeds as follows

$$d^{(1)} = \begin{bmatrix} 0 & 1 & 1 & \infty \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & 1 \\ 1 & \infty & \infty & 0 \end{bmatrix}$$

$$d^{(2)} = \begin{bmatrix} 0 & 1 & 1 & \infty \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & 1 \\ 1 & \infty & \infty & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & \infty \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & 1 \\ 1 & \infty & \infty & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 2 \\ \infty & 0 & 1 & 2 \\ 2 & \infty & 0 & 1 \\ 1 & 2 & 2 & 0 \end{bmatrix}$$

$$d^{(3)} = \begin{bmatrix} 0 & 1 & 1 & \infty \\ \infty & 0 & 1 & 2 \\ 2 & \infty & 0 & 1 \\ 1 & 2 & 2 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & \infty \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & 1 \\ 1 & \infty & \infty & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 3 & 0 & 1 & 2 \\ 2 & 3 & 0 & 1 \\ 1 & 2 & 2 & 0 \end{bmatrix}$$

If $n-1$ is a power of 2 we can reduce the run time considerably by *repeatedly squaring d*

$$d^{(1)} = a^1$$
$$d^{(2)} = d^{(1)} \times d^{(1)} = a^2$$
$$d^{(4)} = d^{(2)} \times d^{(2)} = a^4$$

and so on. The computing time is now $O(n^3 \log n)$.

Algorithm 9 defines the faster version of the shortest paths computation. The multiplication may be performed sequentially (by Algorithm 7) or in parallel (by Algorithms 5 and 6).

```
procedure allpaths(var a, d: matrix);
var m: integer;
begin d := a; m := 1;
  while m < n - 1 do
  begin multiply(d, d, d); m := 2*m end
end
```

Algorithm 9

It is impossible to find finite shortest paths with more than $n-1$ edges. They do not exist! Consequently, the distance matrix remains unchanged when the exponent of a exceeds $n-1$

$$a^{n-1} = a^n = a^{n+1} \ldots$$

This property ensures that Algorithm 9 also works when $n-1$ is not a power of 2.

For pedagogical reasons we have assumed that every edge has length 1. However, the algorithm assumes only that the elements of the adjacency matrix are reals. So the algorithm also works for a directed, *weighted* graph, where each edge has a weight

(or "length) of type real. Weights may even be *negative* as long as the graph has no cycles of negative lengths. If this constraint is violated, a cyclic path becomes shorter every time it is traversed. Consequently, the shortest paths to all nodes which can be reached from a negative cycle are minus infinity!

PERFORMANCE

A multiplication pipeline divides the computational load evenly among the available processors. During a matrix multiplication, n rows and n columns pass through every pipeline node except the last one. (The latter inputs n rows and n columns, but outputs n rows only.)

On a pipeline with p processors, the parallel run time of a matrix multiplication is

$$T_p = an^3/p + bn^2$$

where a and b are system-dependent constants for computation and communication, respectively.

If the multiplication runs on a single processor only (where $p = 1$), the sequential run time is

$$T_1 = an^3 + bn^2$$

The efficiency of the parallel computation is

$$E_p = T_1/(pT_p)$$

which can be rewritten as follows

$$E_p = \frac{an + b}{an + bp}$$

Although the computational load is balanced, the communication overhead reduces the efficiency. However, the communication overhead is negligible if

$$n/p >> b/a$$

that is, if the problem size n is large compared to the pipe length p.

We reprogrammed the pipeline in occam and ran it on a Computing Surface with T800 transputers using 64-bit arithmetic. Measurements show that

$$a = 3.9\ \mu s \qquad b = 20\ \mu s$$

Table 1 shows measured (and predicted) run times of matrix multiplication. In each experiment the ratio $n/p = 40$. By scaling the problem size n in proportion to the computer size p we maintain an almost constant efficiency of 0.89 to 0.91.

Each node holds n/p rows and n/p columns of $8n$ bytes each. The memory requirement per node M_p is proportional to the pipe length p since

$$M_p = 16n^2/p = 25600p \text{ bytes} \quad \text{for } n/p = 40$$

Our Computing Surface has 1 MB of memory per transputer. This limits the pipeline to a maximum of 35 transputers.

Table 1

p	n	T_p (s)		E_p (est)	M_p (bytes)
1	40	0.3	(0.3)	1.00	25600
5	200	6.9	(7.0)	0.91	128000
10	400	27.7	(28.2)	0.90	256000
20	800	112.2	(112.6)	0.89	512000
30	1200	254.6	(253.4)	0.89	768000
35	1400	345.2	(345.0)	0.89	896000

The shortest paths pipeline repeatedly performs multiplication with similar efficiency.

FINAL REMARKS

We have presented a pipeline algorithm for a programming paradigm known as tuple multiplication. From this algorithm we have derived pipelines for matrix multiplication and the all-pairs shortest paths problem by making substitutions of data types and functions.

The predicted efficiency of the parallel matrix multiplication has been confirmed by experiments on a Computing Surface.

ACKNOWLEDGEMENT

I am grateful to Jonathan Greenfield, Harold F. Mattson, Jr., and Peter O'Hearn for comments on this paper.

REFERENCES

1. P. Brinch Hansen, "The all-pairs pipeline," School of Computer and Information Science, Syracuse University, Syracuse, NY, Dec. 1990.

2. A. Cayley, *The Collected Mathematical Papers*, 13 vols. Cambridge University Press, 1889–97. Johnson Reprint Corp.

3. E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

4. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

6. H. T. Kung, "Systolic communication," *IEEE International Conference on Systolic Arrays*, May 1988, San Diego, CA, 695–703.