Fast Parallel Algorithm for Polynomial Interpolation

P. K. JANA
Department of Computer Engineering
Birla Institute of Technology
Mesra, Ranchi—835 215, India

B. P. SINHA*
Electronics Unit
Indian Statistical Institute
203, Barrackpore Trunk Road
Calcutta—700 035, India

(Received and accepted August 1993)

Abstract—This paper presents a parallel algorithm for polynomial interpolation implemented on a mesh of trees. The algorithm is based on the Lagrange's interpolation formula. It requires $O(\log n)$ time using n^2 processors where n is the number of input data points at which the values of the function will be specified. We have also shown how the algorithm can be extended to the case when only p^2 processors ($p < n$) will be available. The algorithm mapped on p^2 processors has a time complexity of $O((n^2/p^2)\log n)$.

Keywords—Interpolation, Lagrange's interpolation formula, Mesh of trees.

1. INTRODUCTION

In many real-time applications involving numerical techniques, we need fast evaluation of a function, say $f(x)$, at some given value of x, just from the knowledge of the values of $f(x)$ at some finite number of discrete points around x. Conversely, from the given set of values of the function $f(x)$ at some discrete values of x, it may also be necessary to know the value of x corresponding to a given value of the function $f(x)$. The former is well known as interpolation, while the latter is known as inverse interpolation. Both of these can be performed by Lagrange's interpolation technique [1].

In recent years, different parallel algorithms [2–4] have been developed for the polynomial interpolation. Schroeder, Murthy and Krishnamurthy [2] have developed a systolic algorithm for polynomial interpolation based on Newton's divided difference scheme which has $O(n)$ time complexity, where n is the number of input data points at which the values of the function $f(x)$ are specified. Murthy, Krishnamurthy and Chen [3] have developed a parallel algorithm for rational interpolation based on Thiele's differences and continued fraction approximation which has the time complexity of $O(n+1)$ using $(n+1)$ processors. The parallel algorithm described in [4] has the time complexity of $O(\log^3 n)$ on the EREW-PRAM model.

In this paper, we propose a parallel algorithm for polynomial interpolation based on Lagrange's interpolation technique on a mesh of trees. The algorithm has been implemented on n^2 processors

with $O(\log n)$ time complexity. We have next shown how the algorithm can be mapped on p^2 processors, where $n = kp$, k being an integer, greater than 1. The algorithm on p^2 processors requires $O((n^2/p^2)\log n)$ time for interpolation.

The paper is organized as follows. In Section 2, we discuss the sequential algorithm for polynomial interpolation. In Section 3, the proposed parallel algorithm using n^2 processors has been discussed. Section 4 describes an example. In Section 5, we have extended the basic idea to develop the interpolation algorithm on p^2 processors, $(p < n)$. Throughout the later discussions, we will assume the base of logarithm as 2.

2. SEQUENTIAL ALGORITHM FOR INTERPOLATION

The n point Lagrange's interpolation formula is as follows [3]:

$$P(x) = \pi(x) \sum_i \left[\frac{y_i}{(x - x_i)\pi'(x_i)} \right],$$

where $y_i = f(x_i)$,

$$\pi(x) = (x - x_0)(x - x_1)(x - x_2) \cdots (x - x_{n-1}),$$
$$\pi'(x_i) = (x_i - x_0)(x_i - x_1)(x_i - x_2) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_{n-1}).$$

ALGORITHM.
Input: $x, x_0, x_1, x_2, \ldots, x_{n-1}, y_0, y_1, y_2, \ldots, y_{n-1}$
Output : $P(x)$

```
1. P(x) ← 0;
2. for i = 0 to n − 1 do
   begin
    prod ← 1;
    for j = 0 to n − 1 do
    begin
     if i≠ j then
       prod ← prod · (x − x_j)/(x_i − x_j)
    end;
    P(x) ← P(x) + prod · y_i;
   end.
```

3. PARALLEL ALGORITHM

In this section, we describe the parallel algorithm for polynomial interpolation using n^2 processors. The computational model used for this purpose is described below.

MODEL DESCRIPTION. We use n^2 processors which are interconnected as follows.

1. n^2 processors are organized in a square array consisting of n rows and n columns; P(i, j) denotes the processor placed in the i^{th} row and j^{th} column.
2. The processors in row i, $1 \le i \le n$, are interconnected in the form of a binary tree rooted at P(i, 1). That is, for j = 1 to $\lfloor n/2 \rfloor$, processor P(i, j) is directly linked to processor P(i, 2j) and P(i, 2j+1), whenever they exist. Similarly, all the processors in column j, $1 \le j \le n$, are interconnected to form a binary tree rooted at P(1, j).
3. Every processor P(i, j) has three local registers A(i, j), B(i, j) and D(i, j). Registers A(i, j)'s are used for data communication along the rows and the B(i, j)'s for data communication along the columns.
4. Data inputting / outputting can be done only through the processors P(i, 1)'s and P(1, j)'s, $\forall i, j,\ 1 \le i, j \le n$.

The parallel algorithm is now described below, assuming that n is a power of 2.

Algorithm A:

Step 1:

1.1 $\forall$ i, $1 \leq i \leq n$, P(i, 1) receives x (in parallel) and stores it in A(i, 1).

1.2 $\forall$ i, $1 \leq i \leq n$, the data in A(i, 1) is broadcast to A(i, j)'s of all the processors in row i, $1 \leq j \leq n$.

1.3 $\forall$ i, $1 \leq i \leq n$, D(i, i) $\leftarrow$ A(i, i).

Step 2: Do the steps 2.1 and 2.2 in parallel.

2.1 $\forall$ i, $1 \leq i \leq n$, P(i, 1) receives x_{i-1} which is stored in A(i, 1).

2.2 $\forall$ j, $1 \leq j \leq n$, P(1, j) receives x_{j-1} which is stored in B(1, j).

Step 3: Do the steps 3.1 and 3.2 in parallel.

3.1 $\forall$ i, $1 \leq i \leq n$, broadcast the value of A(i, 1) to A(i, j)'s of all the processors in row i, $1 \leq j \leq n$.

3.2 $\forall$ j, $1 \leq j \leq n$, broadcast the value of B(1, j) to B(i, j)'s of all the processors in column j, $1 \leq i \leq n$.

3.3 $\forall$ i, $1 \leq i \leq n$, do in parallel A(i, i) $\leftarrow$ D(i, i).

[**Note:** The code that can be used for broadcasting has been shown in the Appendix.]

Step 4:

```
begin
  ∀ i, j, 1 ≤ i, j ≤ n, do in parallel
    A(i, j) ← A(i, j) − B(i, j);
  ∀ i, 1 ≤ i ≤ n, do in parallel
    B(i, i) ← A(i, i);
end.
```

Step 5:

5.1 Do the steps 5.1.1 and 5.1.2 in parallel.

5.1.1 $\forall$ i, $1 \leq i \leq n$, compute the product of the contents of A(i, j)'s $1 \leq j \leq n$ and put it in A(i, 1), $1 \leq i \leq n$.

5.1.2 $\forall$ j, $1 \leq j \leq n$, move the contents of B(j, j) to B(1, j).

5.2 Compute the product of the contents of B(1, j)'s, $\forall$ j, $1 \leq j \leq n$, and put the result in B(1, 1).

5.3 D(1, 1) $\leftarrow$ B(1, 1).

[**Note:** The code that can be used for the steps 5.1.1 and 5.1.2 has been shown in the Appendix.]

Step 6:

```
begin
  ∀ i, 1 ≤ i ≤ n, do in parallel
  begin
    B(i, 1) ← A(i, 1);
    A(i, 1) ← y_{i−1};
    B(i, 1) ← A(i, 1) / B(i, 1);
  end;
end.
```

Step 7: /* sum up all B(i, 1)'s and store the result in B(1, 1) */

```
begin
  for k = log n downto 1 do
  begin
    for i = 2^{k−1} to 2^k − 1 do in parallel
```

```
      begin
       if 2i ≤ n then
        begin
         A(i, 1) ← B(i, 1);
         B(i, 1) ← B(2i, 1);
         B(i, 1) ← A(i, 1) + B(i, 1);
        end;
       if (2i+1) ≤ n then
        begin
         A(i, 1) ← B(i, 1);
         B(i, 1) ← B(2i+1, 1);
         B(i, 1) ← A(i, 1) + B(i, 1);
        end;
      end;
     end;
  end.
```

Step 8: /* Compute the interpolated value at x and store it in B(1,1) */

$B(1, 1) \leftarrow B(1, 1) \cdot D(1, 1)$.

COMPLEXITY OF ALGORITHM A. Each of the steps 1.2, 3, 5.1, 5.2 and 7 in Algorithm A requires $\log n$ time. All the remaining steps require a constant amount of time. Hence, Algorithm A requires $5 \log n + O(1)$ time.

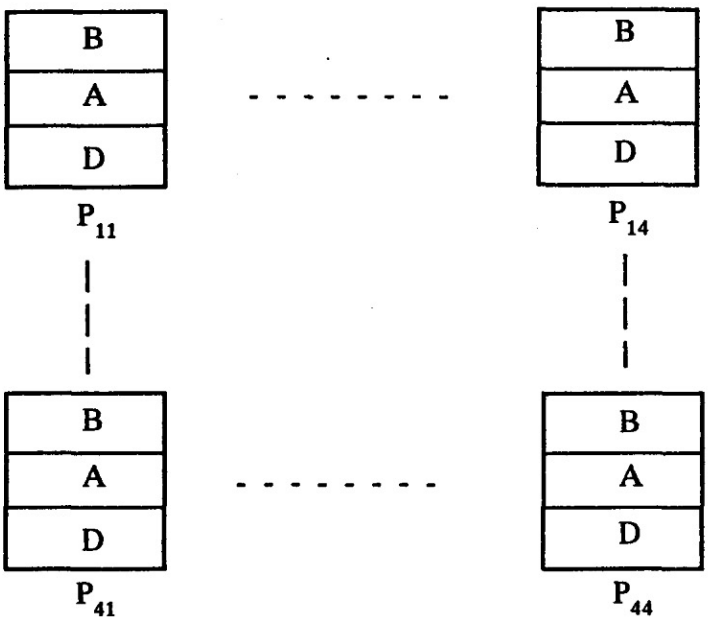

Figure 1. Arrangement of processors for $n = 4$.

4. AN EXAMPLE

We illustrate the above algorithm with an example for $n = 4$. The arrangement of the 16 processors without the interconnections has been shown in Figure 1. The A, B and D registers of the processors have also been shown in the figure.

After executing step 3, we get the situation as shown in the Figure 2, where the values at any processor position correspond to the contents of the registers B, A and D in order from top to bottom, and a 'dash' (-) entry means a don't care value. The situation after executing the step 4 is shown in Figure 3.

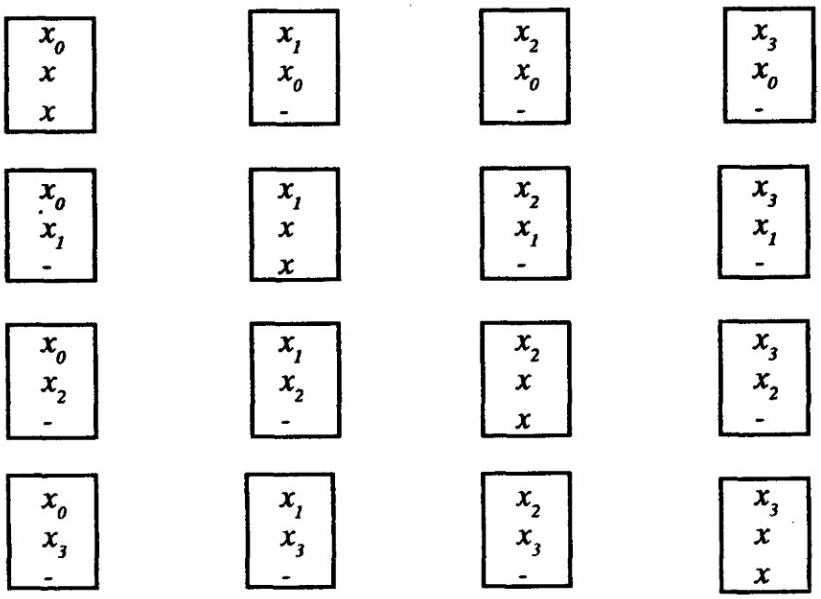

Figure 2. The contents of B, A and D registers of different processors.

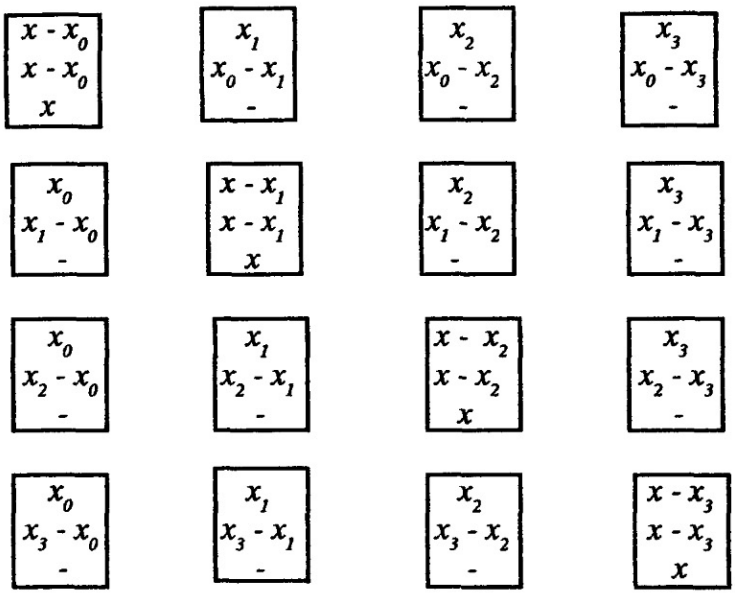

Figure 3. The contents of B, A and D registers after step 4 of Algorithm A.

At the end of step 5 of the algorithm, we get the contents of B, A and D registers as shown in Figure 4.

5. PARALLEL ALGORITHM ON P^2 PROCESSORS

We would now modify Algorithm A for the case when only p^2 processors are available, $p < n$. We will assume, for the sake of simplicity, that $n = kp$, where k is an integer. The basic idea is as follows.

The n input values are grouped in $k = n/p$ sets: $\{x_0, x_1, \ldots, x_{p-1}\}$, $\{x_p, x_{p+1}, \ldots, x_{2p-1}\}$, $\ldots$, $\{x_{(k-1)p}, x_{(k-1)p+1}, \ldots, x_{kp-1}\}$. For a given input set to the p rows, the columns are successively fed with possible input sets and each time the required product terms are evaluated. Then the input sets to the rows are successively changed and the above procedure is repeated to generate the final interpolated value at some result register R(1, 1) of the processor P(1, 1). The algorithm

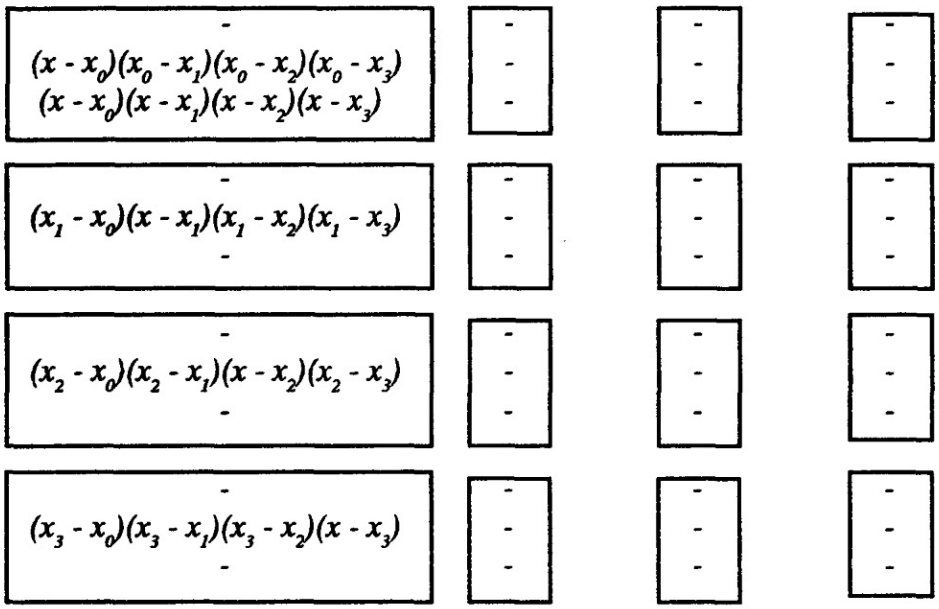

Figure 4. The contents of B, A and D registers after step 5 of Algorithm A.

is described below stepwise. We use two additional temporary registers TEMP1 and TEMP2, along with a result register R per processor for the description of the algorithm.

ALGORITHM B.

Step 1: Initialize TEMP1(1, 1) to 0.

Step 2: The value of x is given as input to all the rows and stored in the D register of only the diagonal processors.

Step 3:

3.1 Do the steps 3.1.1 and 3.1.2 in parallel.

3.1.1 The inputs $x_0, x_1, \ldots, x_{p-1}$ are given to the rows 1, 2, ..., p, respectively.

3.1.2 The inputs $x_0, x_1, \ldots, x_{p-1}$ are given to the columns $1, 2, \ldots, p$, respectively.

3.2 Do the steps 3.2.1 and 3.2.2 in parallel.

3.2.1 Proceeding in the same way as in Algorithm A, the product $(x - x_0)(x - x_1) \cdots (x - x_{p-1})$ is computed and stored in the register R(1, 1) of P(1, 1).

3.2.2 The products $(x - x_i)(x_i - x_0)(x_i - x_1) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_{p-1})$, $0 \leq \mathrm{i} \leq \mathrm{p} - 1$, are computed and stored in the temporary registers TEMP2 (i+1, 1) of the processors P(i+1, 1).

Step 4: Now the input set to the columns 1, 2, ..., p is changed to $\{x_p, x_{p+1}, \ldots, x_{2p-1}\}$ to do the steps 4.1 and 4.2 in parallel.

4.1 Generate the product $(x - x_p)(x - x_{p+1}) \cdots (x - x_{2p-1})$ and then multiply it by R(1, 1) to store the result in R(1, 1). R(1, 1) now contains $(x - x_0)(x - x_1) \cdots (x - x_{2p-1})$.

4.2 The products $(x_i - x_p)(x_i - x_{p+1}) \cdots (x_i - x_{2p-1})$ for $0 \leq \mathrm{i} \leq p - 1$, are computed to multiply the contents of TEMP2(i+1, 1). Thus, TEMP2(i+1, 1) now contains $(x - x_i)(x_i - x_0)(x_i - x_1) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_{2p-1})$.

Step 5: Step 4 is now repeated $(k-2)$ times with the successive input sets $\{x_{2p}, x_{2p+1}, \ldots, x_{3p-1}\}$, $\{x_{3p}, x_{3p+1}, \ldots, x_{4p-1}\}, \ldots$, to the columns 1, 2, ..., p. After this, R(1, 1) will contain the value of $(x-x_0)(x-x_1)\cdots(x-x_n)$ and TEMP2(i+1, 1) will contain the value of $(x-x_i)(x_i-x_0)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_{n-1})$, for $0 \leq \mathrm{i} \leq p-1$.

Step 6: The values $y_i = f(x_i)$, $0 \leq \mathrm{i} \leq p-1$, of the function are now given as inputs to the $(\mathrm{i}+1)^{\text{th}}$ row. y_i is then divided by TEMP2(i+1, 1) and the result is stored in B(i+1, 1). The values of all B(i+1, 1), $0 \leq \mathrm{i} \leq p-1$, are then summed up to store the result in B(1, 1). B(1, 1) is then added to the temporary register TEMP1(1, 1).

Step 7: Steps 3.1.2 through 6 are repeated for the successive input sets $\{x_p, x_{p+1}, \ldots, x_{2p-1}\}$, $\{x_{2p}, x_{2p+1}, \ldots, x_{3p-1}\}$, ..., $\{x_{(k-1)p}, x_{(k-1)p+1}, \ldots, x_{kp-1}\}$, given to the rows 1, 2, ..., p.

Step 8: Multiply R(1, 1) by TEMP1(1, 1) to produce the final interpolated value at R(1, 1) and then stop.

The detailed codes for the above steps can very easily be developed. To find the time complexity of this algorithm, we proceed as follows.

Step 1 requires constant time. Step 2 requires $\log p$ time. Each of steps 3 and 4 requires $2\log p$ time. Step 5 requires $(k-2)\cdot 2\log p$ time. Step 6 requires constant time. Hence, steps 1 through 6 require $(2k+1)\log p + O(1)$ time. Step 7 needs $(k-1)[2k\log p + O(1)]$ time. Step 8 requires constant time. Hence, the total time required by Algorithm B is $(2k^2+1)\log p + O(k) = O((n^2/p^2)\log n)$.

6. CONCLUSION

A parallel algorithm for polynomial interpolation has been developed on a mesh of trees with n^2 processors and $O(\log n)$ time complexity, where n is the number of data points at which the values of the function will be specified. It has also been shown how the underlying idea can be adapted to the situation when only p^2 processors ($p < n$) will be available. The corresponding implementation has been shown to have $O((n^2/p^2)\log n)$ time complexity.

APPENDIX

We would present here the detailed code for some of the steps in Algorithm A. We can use the following code to broadcast the value in A(i, 1) to all the processors in row i in log n time.

```
begin
  for k = 1 to log n do
  begin
   for j = 2^(k-1) to 2^k - 1 do in parallel
   begin
     if 2j ≤ n then A(i, 2j) ← A(i, j);
     if (2j + 1) ≤ n then A(i, 2j+1) ← A(i, j);
   end;
  end;
end.
```

The code for broadcasting the value in B(1, j) can similarly be developed. We can use the following code for the step 5.1.1 in Algorithm A.

```
begin
  for k = log n downto 1 do
  begin
   for j = 2^(k-1) to 2^k - 1 do in parallel
```

```
      begin
       if 2j ≤ n then
         begin
          D(i, j) ← A(i, j);
          A(i, j) ← A(i, 2j);
          A(i, j) ← A(i, j) · D(i, j);
         end;
        if (2j + 1) ≤ n then
         begin
          D(i, j) ← A(i, j);
          A(i, j) ← A(i, 2j+1);
          A(i, j) ← A(i, j) · D(i, j);
         end;
      end;
  end;
end.
```

The following code can be used for the step 5.1.2 in Algorithm A.

```
begin
  for j = 2 to n do in parallel
  begin
   i ← j;
   repeat
     B( ⌊ i/2 ⌋, j) ← B(i, j);
     i ← ⌊ i/2 ⌋;
   until (i = 0);
  end;
end.
```

REFERENCES

1. F.B. Hildebrand, *Introduction to Numerical Analysis*, McGraw-Hill, New York, (1956).
2. H. Schroeder, V.K. Murthy and E.V. Krishnamurthy, Systolic algorithm for polynomial interpolation and related problems, *Parallel Computing*, 493–503 (July 1991).
3. V.K. Murthy, E.V. Krishnamurthy and P. Chen, Systolic algorithm for rational interpolation and Pade approximation, *Parallel Computing*, 75–83 (January 1992).
4. J. Já Já, *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, Reading, MA, (1992).