BIT 30 (1990), 268–288

A PARALLEL METHOD FOR FAST AND PRACTICAL HIGH-ORDER NEWTON INTERPOLATION

Ö. EĞECIOĞLU*, E. GALLOPOULOS**

Department of Computer Science, University of California Santa Barbara, Santa Barbara, CA 93106, USA

Center for Supercomputing Research and Development and Department of Computer Science University of Illinois at Urbana-Champaign Urbana, IL 61801, USA

and Ç. K. KOÇ

Department of Electrical Engineering, University of Houston, Houston, Texas 77204, USA

Abstract.

We present parallel algorithms for the computation and evaluation of interpolating polynomials. The algorithms use parallel prefix techniques for the calculation of divided differences in the Newton representation of the interpolating polynomial. For $n + 1$ given input pairs, the proposed interpolation algorithm requires only $2\lceil \log(n + 1)\rceil + 2$ parallel arithmetic steps and circuit size $O(n^2)$, reducing the best known circuit size for parallel interpolation by a factor of $\log n$. The algorithm for the computation of the divided differences is shown to be numerically stable and does not require equidistant points, precomputation, or the fast Fourier transform. We report on numerical experiments comparing this with other serial and parallel algorithms. The experiments indicate that the method can be very useful for very high-order interpolation, which is made possible for special sets of interpolation nodes.

AMS Subject Classifications (1985): 65D05, 65W05, 68C25.

1. Introduction.

Given a set of $n + 1$ pairs of values, (x_i, f_i) for $i = 0, 1, \ldots n$ with x_i distinct, there exists a unique polynomial $p_n(x)$ of degree n such that

$$p_n(x_i) = f_i \quad \text{for } i = 0, 1, \ldots, n.$$

* Supported in part by the National Science Foundation under Grant No. NSF DCR-8603722.

** Supported by the National Science Foundation under Grants No. US NSF MIP-8410110, US NSF DCR85-09970, and US NSF CCR-8717942 and AT&T under Grant AT&T AFFL67Sameh.

Author to whom correspondence should be addressed. stratis@uicsrd.csrd.uiuc.edu

Received June 1989. Revised September 1989.

This interpolating polynomial $p_n(x)$ can be written in the Newton form

$$p_n(x) = \sum_{i=0}^{n} f_{01\ldots i}(x - x_0)(x - x_1)\ldots(x - x_{i-1}) \tag{1}$$

in which the coefficients $f_{01\ldots i}$ are the *divided differences* (*DD*) of f.

Polynomial interpolation in various forms has been studied both in the context of numerical analysis and in that of computational complexity. Despite the advances in the construction of fast interpolation algorithms (of time complexity less than $O(n^2)$) by researchers in the latter group (e.g. Kung in [13], Horowitz in [10], Reif in [21]) numerical algorithms are still based on the slow serial schemes or small improvements thereof, which require $O(n^2)$ operations [11], [23], [15]. In view of the increasing availability of parallel systems, the impracticability of the existing approaches can be attributed mainly to the following two reasons:

(i) constant of proportionally of the order tends to be large,
(ii) the current fast interpolation algorithms are subject to significant roundoff errors when implemented in finite precision arithmetic.

Regarding (i), we note that a large constant multiplier means that the dimensionality of the problem must increase considerably to make an asymptotically fast algorithm competitive. This can be self-defeating as polynomial interpolation may suffer for very large number of points. Recent work, however, on suitable node sequences has caused renewed interest in high-order interpolation ([7, 26]).

As for (ii), we may echo Miller's remark in [18], that in the quest for fast and optimal algorithms there is little practical value in studying those whose numerical stability is unsatisfactory.

As a partial answer to these problems we present a fast and practical parallel algorithm for the computation of interpolating polynomials. By *fast* we mean that the algorithm requires time $O(\log n)$. More explicitly, the parallel time is $2\lceil\log(n + 1)\rceil + 2$ (all algorithms in this paper are base 2) using $n(n + 1)$ processors. By *practical* we mean that the proposed algorithm is stable and may be implemented in floating-point with resulting error accumulation similar to that of the widely used serial algorithms. Some of these results were first reported in [4]. It is fair to say however that it makes little sense using parallel interpolation for low-order problems. We thus used our algorithm for high-order interpolation based on results of recent research ([7,26]). *Experiments with our algorithm indicate that in a parallel environment, high-order polynomial interpolation can be possible, fast, and practical.*

Traditionally, fast algorithms for interpolation have made use of the Lagrange representation of the interpolating polynomial. With this representation it is possible to perform parallel interpolation using an arithmetic circuit of depth $O(\log n)$ and size $O(n^2 \log n)$ [21] (cf. the definitions in Section 2). This approach is based on a novel scheme to multiply more than two polynomials by fast convolution. Here the constant of proportionality in the order turns out to be larger than 4, as the algorithm requires fast implementations (e.g. by means of FFT) of the forward and the inverse Discrete Fourier Transform (DFT).

The algorithm presented here makes use of the Newton representation of the interpolating polynomial. This representation allows for parallel computation of the required coefficients using an arithmetic circuit of depth $2\lceil \log(n+1) \rceil + 2$ and size $O(n^2)$. Central to our approach are well-known fast algorithms and circuits for the *parallel prefix* computation (described in Section 2) for the evaluation of the DD coefficients. The expressions obtained for the expansion of the DD's as a linear combination of the function values turn out to be sufficiently simple in this case, yet promising for similar treatment of other interpolation schemes [6,5]. The algorithms for interpolation and evaluation may by implemented either on distributed or shared-memory machines. In particular, the PRAM model of a shared-memory machine ([8]), in its concurrent read, exclusive write (CREW) version as classified by Snir in [24] is most suitable for the algorithms presented. At the expense of a small constant number of operations, the algorithms are also suitable for the even weaker exclusive read, exclusive write (EREW) model.

The outline of this paper is as follows: Section 2 contains preliminary notions and definitions. In Sections 3 and 4 we present the basic algorithms for interpolation and evaluation, and prove the stated results concerning depth and size. Section 5 is on the error properties of the algorithms. In Section 6 we describe the numerical experiments. Section 7 contains conclusions and suggestions for further research.

2. Preliminaries.

An *arithmetic circuit* α_N over a field [21] is an acyclic labeled digraph with

(i) a list of N distinguished input nodes with fan-in 0;
(ii) constant nodes with fan-in 0 labeled with the elements of the field;
(iii) operational nodes with fan-in 2, each labeled with one of the symbols from $\{+, -, \times, /\}$;
(iv) a list of M distinguished output nodes of fanout 0.

The *depth* of the circuit is the length of its longest path. The *size* of the circuit is the number of operational nodes. The circuit accepts inputs from the N input nodes and produces outputs by evaluation of the operational nodes in topological order. This process is well defined since the graph is acyclic. Hence the depth of an arithmetic circuit is proportional to the time required to run the corresponding algorithm on a parallel machine.

Let $*$ be an associative binary operation on a set T. The *prefix computation* problem is defined as follows: Given an ordered n-tuple $(y_1, y_2, \ldots, y_n)$ of elements of T, compute all n prefixes $y_1 * y_2 * y_3 * \ldots * y_i$ for $i = 1, 2, \ldots, n$. The parallel algorithms for this computation are commonly known as *parallel prefix algorithms*. The following results are well known and essential for the remaining discussion ([12, 14]).

LEMMA 2.1.

i) *The n input parallel prefix computation can be performed in $\lceil \log n \rceil$ time with n processors.*

ii) *There exists a family of arithmetic circuits $P_k(n)$ for the n input prefix computation, such that if S and D denote the arithmetic size and depth then*

$$S(P_k(n)) \leq 2(1 + 2^{-k})n - 4 \tag{2}$$

$$D(P_k(n)) \leq k + \lceil \log n \rceil. \tag{3}$$

Part (ii) of the Lemma for $k = 0$ shows that n input parallel prefixes can be computed in $\lceil \log n \rceil$ depth and size less than $4n$.

The following algorithm summarizes the required computations [12].

```
PROCEDURE Parallel.Prefix($n, y$)
FOR $j := 0$ TO $\lceil \log n \rceil - 1$ DO
    FORALL $i \in \{2^j + 1, \ldots, n\}$ DO IN PARALLEL $newy[i] := y[i] * y[i - 2^j]$
    END FORALL
    FORALL $i \in \{2^j + 1, \ldots, n\}$ DO IN PARALLEL $y[i] := newy[i]$
    END FORALL
END FOR
END PROCEDURE. $\{y[i]$ contains $y_1 * y_2 * \ldots * y_i\}$.
```

3. Parallel Newton interpolation algorithm.

The interpolation polynomial in Newton form is completely determined by the divided difference coefficients. In a serial setting two methods for their derivation are the Neville and Aitken methods, both of sequential complexity $O(n^2)$. These algorithms are described in most introductory numerical analysis textbooks. For example the Neville procedure uses

$$f_{i,i+1,\ldots,i+p} = \frac{f_{i,i+1,\ldots,i+p-1} - f_{i+1,i+2,\ldots,i+p}}{x_i - x_{i+p}}$$

to calculate the terms in the following triangular table

$$\begin{array}{lllll} f_0 & & & & \\ f_1 & f_{01} & & & \\ f_2 & f_{12} & f_{012} & & \\ f_3 & f_{23} & f_{123} & f_{0123} & \\ f_4 & f_{34} & f_{234} & f_{1234} & f_{01234} \end{array}$$

The entries on the diagonal are the required DD's. Note that the terms in a given column can be calculated independently of one another, and they depend only on the entries in the previous column and the x_i's. This gives a straigthforward parallel algorithm for the computation of DD's, where each column is computed in constant time using as many processors as there are entries in the column. Thus this approach requires $O(n)$ time to calculate all the DD's using $O(n)$ processors. Similar techniques can be used for the construction of systolic arrays for parallel interpolation [17].

We now present the parallel Newton interpolation algorithm.

Instead of the Neville or the Aitken recurrence formulae, our point of departure is a classical alternate formulation for the DD's given in [1], [3] and elsewhere. This can be stated as follows: Let $y_{ij} = (x_i - x_j)$ for $i, j = 0, 1, \dots, n$ and $i \neq j$. Then the kth DD of f can be expressed as a linear combination of the given values $f_0, f_1, \dots, f_k$ with coefficients that are inverses of products of the quantities y_{ij}'s in the form

$$f_{012\dots k} = \frac{f_0}{y_{01}y_{02}\cdots y_{0k}} + \frac{f_1}{y_{10}y_{12}\cdots y_{1k}} + \dots + \frac{f_k}{y_{k0}y_{k1}\cdots y_{k,k-1}} \tag{4}$$

where k ranges from 0 to n. Denoting the inverse of the coefficient of f_i in the linear expansion of $f_{012\dots k}$ $(i \leq k)$ by $f_{012\dots k}|_{f_i}$, the coefficients in the expansion of Eq. (4) can be written as

$$f_{012\dots k}|_{f_i} = y_{i0}y_{i1}\cdots y_{i,i-1}y_{i,i+1}\cdots y_{ik}.$$

Hence

$$\begin{aligned} f_{01}|_{f_0} &= y_{01} \\ f_{012}|_{f_0} &= y_{01}y_{02} \\ f_{0123}|_{f_0} &= y_{01}y_{02}y_{03} \\ &\cdots \\ f_{012\dots n}|_{f_0} &= y_{01}y_{02}y_{03}\cdots y_{0n} \end{aligned}$$

and therefore the computation of this sequence of coefficients amounts to the calculation of the prefixes of (y_{01}, y_{02}, y_{0n}). By Lemma 2.1 this requires $\lceil \log n \rceil$ time with n processors. Since $n + 1$ concurrent instances of a parallel prefix algorithm are needed to compute the prefixes of the term $(y_{i0}, y_{i1}, \ldots, y_{i,i-1}, y_{i,i+1}, \ldots, y_{in})$ for i ranging from 0 to n, we need no more than $O(n^2)$ processors for the whole computation. Note that if all $f_{012\ldots k}|_{f_i}$ are known for $i \leq k \leq n$, then the DD's $\{f_0, f_{01}, f_{012}, \ldots f_{012\ldots n}\}$ of f that are required for the interpolating polynomial $p(x)$ can be calculated in $O(\log n)$ time using $O(n^2)$ processors. First all y_{ij} can be calculated in a single step using $n(n+1)/2$ processors for all $i, j = 0, 1, \ldots, n$ and $i \neq j$. Next by using $n + 1$ concurrent instances of the parallel prefix algorithm, all of $f_{012\ldots k}|_{f_i}$ for $k = 1, 2, \ldots, n$ and $i \leq k$ can be computed in at most $\lceil \log n \rceil$ arithmetic steps. This requires $n(n+1)$ processors. Subsequently a parallel division using $n(n+1)/2$ processors is performed. Finally, the application of $n + 1$ concurrent instances of a binary tree addition algorithm (with the one corresponding to f_0 being trivially empty) yields the values in an additional $\lceil \log(n+1) \rceil$ arithmetic steps using $n(n+1)$ processors. Thus $n(n+1)$ processors suffice to compute all $f_{012\ldots k}$ for $k = 0, 1, \ldots, n$ in $2\lceil \log(n+1) \rceil + 2$ parallel time.

The processor count in the argument above was given for a system where each processor is able to do any of the 4 arithmetic operations and where the processors may be reused. From Lemma 2.1 it follows that the solution to the parallel prefix problem can be calculated with a circuit of depth $\lceil \log(n+1) \rceil$ and size less than $4(n+1)$. We claim that this allows for an arithmetic circuit of size $O(n^2)$ and depth $2\lceil \log(n+1) \rceil + 2$ for interpolation. To see this, consider each of the steps in the algorithm: At first all y_{ij} are calculated. This can be done in one step with $n(n+1)/2$ operational nodes. Next the parallel prefix algorithm is applied to form all the $f_{012\ldots k}|_{f_i}$. Hence all these terms may be calculated in parallel with that same depth and with size at most $4n(n+1)$. As with the subtraction, the parallel division also requires $n(n+1)/2$ nodes. Finally the additions of the constituent terms for the calculation of the DD coefficients is done by means of $n + 1$ parallel circuits for binary summation each having depth less than $\lceil \log n \rceil$ and total size at most $O(n^2)$. Thus we have proved

THEOREM 3.1. *The Newton interpolating polynomial for $n + 1$ points can be computed in $2\lceil \log(n+1) \rceil + 2$ parallel arithmetic steps using $n(n+1)$ processors and can be implemented as an arithmetic circuit having the same depth, and size $O(n^2)$.*

We remark that by making use of further results from [14] and [2], tighter bounds may be found for the circuit size by only small increases to its depth.

Figures 1 and 2 illustrate the structure of the algorithm. The parallel prefix circuit in Figure 2 is a pictorial reresentation of the algorithm as given in Section 2. For simplicity of representation we only depict the size $O(n \log n)$ circuit instead of the more efficient $O(n)$ circuit of [14].

We next make an observation which will be important in the following discussions for the algorithm's implementation in the limited processor case: the summation of n elements can be achieved in $\lceil \log n \rceil$ time using the parallel prefix algorithm, with addition as its basic operation. From this it follows that the two

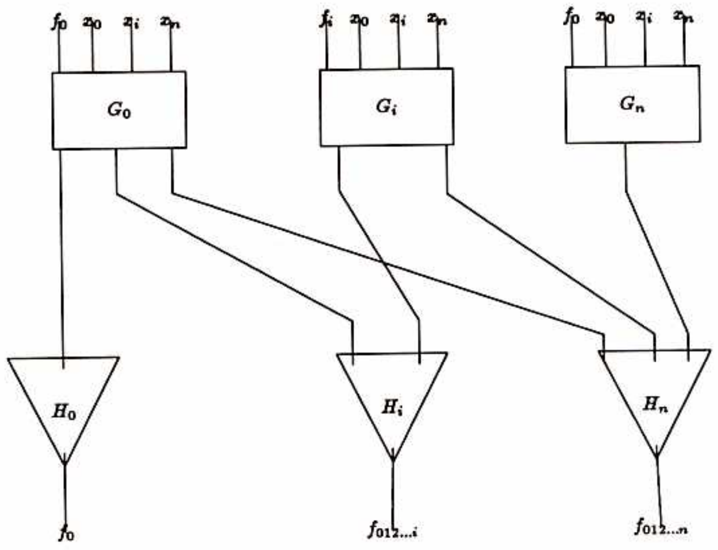

Fig. 1. Parallel Newton interpolation algorithm, H_i is a sum tree of depth $\lceil\log(i+1)\rceil$.

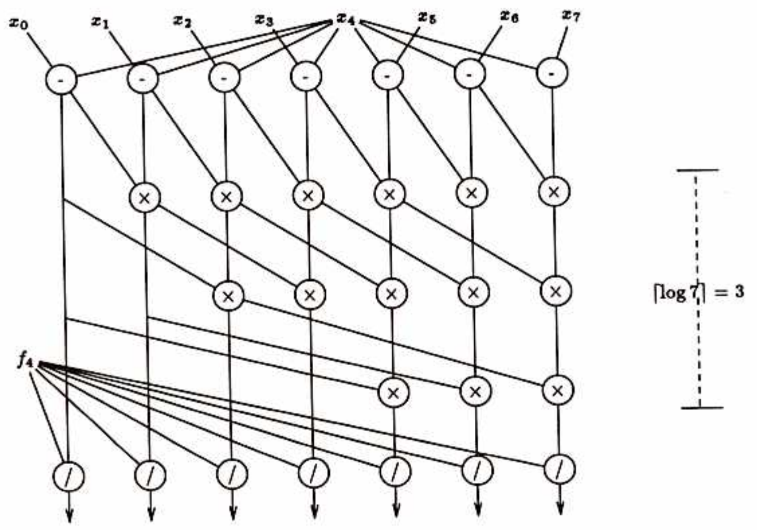

Fig. 2. Expansion of G_i for $i = 4$ and $n = 7$.

basic steps of the interpolation can be implemented using the same type of algorithm and circuit. Thus alternately we have:

I Compute all $y_{ij} = x_i - x_j$ for $0 \le i \ne j \le n$ setting $y_{ii} = 1$.
II For all $i = 0, \ldots, n$ compute in parallel the prefixes $Y_{ij} = y_{i0} \cdots y_{ij}$ of the terms $(y_{i0}, y_{i1}, y_{i2}, \ldots, y_{in})$.
III For all $i = 0, \ldots, n$ and for all $j = 0, \ldots, n$ set $Z_{ij} = f_i / Y_{ij}$ when $i \le j$ and 0 otherwise.

IV For all $j = 0, \ldots, n$ compute in parallel the sum of the terms $(Z_{0j}, Z_{1j}, Z_{2j}, \ldots, Z_{jj})$ using parallel prefix; at the end of this last step, the term $Z_{0j} + \ldots + Z_{jj}$ is the jth DD.

We next consider the case of a limited number of processors p, in particular when $p = m(n + 1)$ where $1 < m < n + 1$ is an integer. For simplicity it is assumed that the parallel additions in the last step are done using the parallel prefix algorithm. It has been shown in [12] that one instance of parallel prefix for $n + 1$ elements can be computed in

$$2\lceil (n + 1)/m \rceil + \lceil \log m \rceil - 2$$

parallel arithmetic steps when m processors are available. Since the parallel Newton interpolation algorithm requires $n + 1$ concurrent instances of parallel prefix, we assign to each group of m processors, one instance of the required $n + 1$ parallel prefix operations. The resulting parallel Newton algorithm uses limited processor parallel prefix both for the product and summation parts of the algorithm, and uses all of the $m(n + 1)$ processors for the required parallel subtractions and divisions. It can be easily shown that:

COROLLARY 3.1. *The parallel Newton algorithm for $n + 1$ data points can be done in $6\lceil (n + 1)/m \rceil + 2\lceil \log m \rceil - 4$ arithmetic steps when $p = m(n + 1)$ processors are available.*

If the number of available processors p is less than $n + 1$, then the algorithm above will become inefficient. A better strategy in this case would be to calculate the entries in the DD table columnwise in parallel using $p \leq n + 1$ processors since the entries in a column are independent of one another and can be calculated in parallel. As an example when $p = n + 1$ then it will take exactly $n + 1$ arithmetic steps to calculate all the entries in the DD table. This amounts to $O(n)$ speedup over the sequential algorithm which requires $n(n + 1)/2$ arithmetic steps.

4. Evaluation of the interpolating polynomial.

Whenever polynomial interpolation is used, it is frequently required to also evaluate the polynomial at a single or many points. Indeed, a fast algorithm for the interpolation would not be very useful unless an algorithm of comparable speed could be designed for the evaluation. Polynomial evaluation has been studied by many authors and fast algorithms have been proposed. It is well known that given n processors, a polynomial of degree n can be evaluated in $O(\log n)$ arithmetic steps [16], [19]. Even though all these algorithms are designed for evaluating a polynomial in its standard form, they can be applied to the Newton polynomial as well.

The following algorithm evaluates the Newton interpolating polynomial p_n at x in

$2\lceil\log(n+1)\rceil + 2$ arithmetic steps using n processors. This is based on the same idea as the computation of the DD's for the interpolating polynomial given in Section 3. Briefly, it consists of the following steps:

I Compute all differences $y_i = x - x_i$ for $0 \leq i \leq n-1$,
II Compute the prefixes of the terms $(y_0, y_1, y_2, \ldots, y_{n-1})$,
III Compute the quantities $f_{012\ldots i} y_0 y_1 \cdots y_{i-1}$ for $0 < i \leq n$,
IV Add the terms computed in step (III) to obtain $p_n(x)$.

Thus we have

THEOREM 4.1. *The Newton interpolating polynomial of degree n can be evaluated in* $2\lceil\log(n+1)\rceil + 2$ *parallel arithmetic steps using n processors and can be implemented as an arithmetic circuit of size* $O(n)$.

PROOF. The proof is essentially a special case of the argument given for the proof of Theorem 3.1, and will be omitted.

By a straigthforward extension it can also be shown that the evaluation can be done at n points with a circuit of depth $O(\log n)$ and size $O(n^2)$.

This evaluation algorithm is suitable in a parallel processing environment where the parallel Newton interpolation algorithm itself is implemented since it follows the same steps as the parallel interpolation algorithm given in Section 3: the same parallel prefix algorithm is used to carry out (II) and a (possibly prefix based) summation algorithm is necessary in (IV). This uniformity may be desirable for VLSI implementation of the algorithms.

5. Error analysis.

We outline here the forward error analysis for the parallel Newton interpolation algorithm and demonstrate its stability. The methodology follows [22] and [25], and [9] (where it was used to demonstrate the stability of pipelined recurrence solvers).

Assuming that symmetric rounding is used and that the mantissa has μ radix β digits, let $\varepsilon = 0.5 \times \beta^{-\mu+1}$. Inputs may be subject to *data errors* whenever they cannot be represented exactly in memory. For real ζ, $\mathrm{fl}(\zeta)$ is the floating-point number closest to ζ, so that

$$|\mathrm{fl}(\zeta) - \zeta| \leq \varepsilon|\zeta|.$$

Arithmetic operations may be subject to *rounding errors* due to finite precision arithmetic: Assuming that η, ζ are exactly representable as floating-point numbers with $\odot$ the machine operation corresponding to operation $*$ (one of $\{+, -, \times, /\}$):

$$|\eta \odot \zeta - \eta * \zeta| \leq \varepsilon|\eta * \zeta|.$$

We are concerned with *absolute error*, the difference between exact and computed results. Data and rounding error effects are considered separately and only to first order. The total error is the sum of the rounding and data errors. The computation of bounds for the errors is facilitated by introducing the notions of a *computational graph* and of *magnification factors*. A straight-line algorithm (an algorithm in which the flow of control is independent of the particular input values, though it may depend on the number of inputs) can be represented by a directed acyclic graph, called a computational graph. This consists of input and operational nodes, with some of them also marked as output nodes. In fact, with the exception of the treatment of the output nodes, the computational graph is nothing more than another representation of an arithmetic circuit as defined in Section 2. Each arc of the graph is labeled with an *absolute error magnification factor* as given in Table 1 for Figure 3. For a given arc, the corresponding magnification factor shows how the absolute error is magnified in that segment of the computation. For example, when computing $z = x + y$ with x and y known only approximately with corresponding absolute errors δ_x and δ_y, Table 1 and Figure 3 tell us that δ_x is magnified by a factor of y and δ_y by a factor of x.

Table 1. *Absolute error magnification factors* (*assume the result and y not equal to* 0).

Operation	δ_x	δ_y
$x \times x$	y	x
x/y	$1/y$	$-x/y^2$
$x + y$	1	1
$x - y$	1	-1

To determine the influence of data and rounding errors on the final floating-point results we attach magnification factors to every arc of the computational graph. Although in general an algorithm will have many outputs (e.g. the $n + 1$ DD's for Newton interpolation), we continue the discussion for a single output. In the multiple output case, the same discussion is applied to each one of them separately. Since not all input and operational nodes necessarily introduce error, we define an indicator variable I_ζ which is equal to 1 if input or operation at node ζ is subject to error and is 0 otherwise. Let each of the different paths from ζ to the output be labeled from an index set J_ζ. For every non-output node ζ of the graph, one forms the products P_j of all magnification factors along all different paths from ζ to the output node.

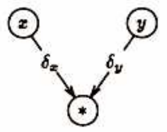

Fig. 3. Absolute magnification factors for $x * y$.

For the output node $S_{\text{output}} = 1$. The absolute data and rounding error for the output are [22]

$$F_D = \sum_{\zeta \text{ input node}} |S_\zeta| \cdot |value[\zeta]| \cdot I_\zeta \cdot \varepsilon$$

$$F_R = \sum_{\zeta \text{ operational node}} |S_\zeta| \cdot |value[\zeta]| \cdot I_\zeta \cdot \varepsilon.$$

Then as shown in Theorem 1 of [22], the total absolute error corresponding to the output satisfies

$$F \le F_R + F_D + O(\varepsilon^2).$$

Since the distribution of I_ζ is unknown, we take the worst case $I_\zeta = 1$ at all ζ, and define absolute data and rounding *condition numbers*:

$$\sigma_D = \sum_{\zeta \text{ input node}} |S_\zeta| \cdot |value[\zeta]| \tag{5}$$

$$\sigma_R = \sum_{\zeta \text{ operational node}} |S_\zeta| \cdot |value[\zeta]|. \tag{6}$$

It follows that $F_D \le \sigma_D \cdot \varepsilon$ and $F_R \le \sigma_R . \varepsilon$ so that ignoring second order terms

$$F \le (\sigma_R + \sigma_D)\,\varepsilon.$$

An algorithm for the evaluation of an expression is called *numerically stable* when the worst effect of the rounding errors (the rounding condition number) is not worse than some positive scalar, possibly dependent only on the structure of the expression (e.g. number of arguments), multiplied with the worst effect of the data errors (the data condition number).

LEMMA 5.1. *If $\sigma_{k,D}$ and $\sigma_{k,R}$ denote the data and rounding condition numbers respectively for the k^{th} divided-difference associated with the parallel Newton interpolation algorithm, then:*

$$\sigma_{k,D} = \sum_{i=0}^{k} \frac{|f_i|}{|\phi_i^{(k)}(x_i)|} + \sum_{i=0}^{k} \left| \sum_{\substack{j=0 \\ j \ne i}}^{k} \left(\frac{f_i}{\phi_i^{(k)}(x_i)} + \frac{f_j}{\phi_j^{(k)}(x_j)} \right) \frac{x_i}{x_i - x_j} \right|$$

and

$$\sigma_{k,R} \le (2k + \lceil \log(k+1) \rceil) \sum_{i=0}^{k} \frac{|f_i|}{|\phi_i^{(k)}(x_i)|}$$

where $\phi_i^{(k)}(x) = \prod_{\substack{v=0 \\ v \neq i}}^{k} (x - x_v)$.

Proof. The computational graph for parallel Newton interpolation when $k = 3$ is given in Figure 4. Although as noted at the end of Section 3, the parallel prefix computations in this graph correspond to a larger than minimum size circuit, it is easy to show that the same discussion applies to the smaller circuits of [14]. Input nodes are given by double circle nodes. Arcs are annotated with the corresponding magnification factors. It is important to note that for each operational node, the left and right operands are determined by the magnification factors on the incoming arcs, and not by the order of these arcs in the figure. To avoid overlapping we inserted x_i as an input node more than once. For each x_i, there are $2k$ paths leading

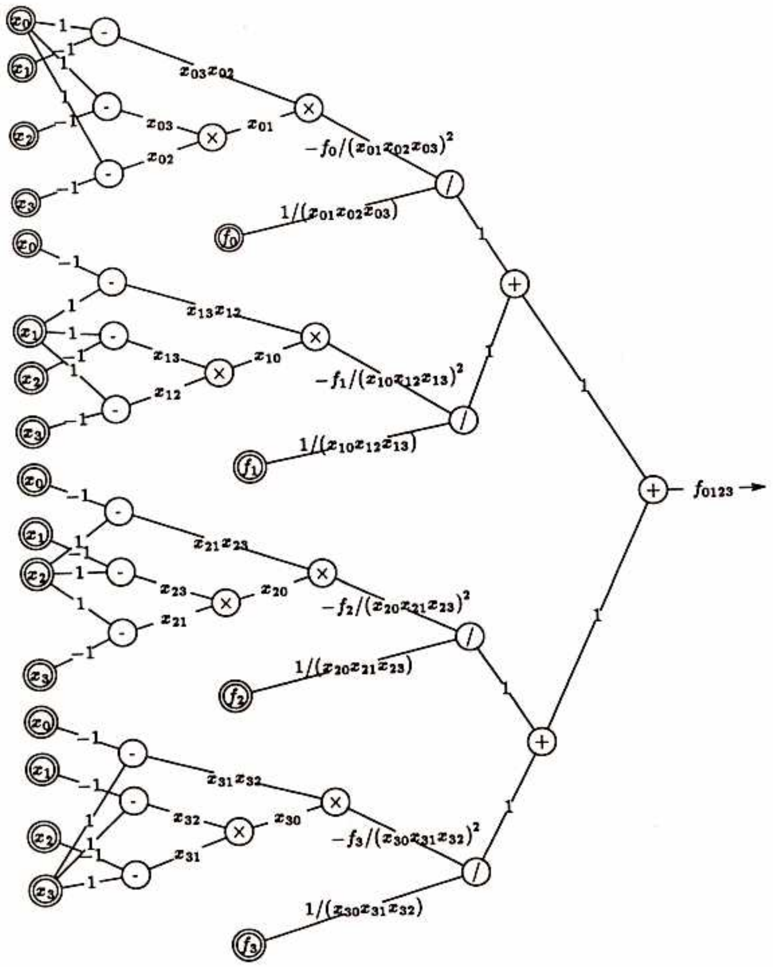

Fig. 4. Computational graph and magnification factors for the calculation of f_{0123}.

to the ouput node. Out of these $2k$ paths, k correspond to x_i's contribution in $x_i - x_j$, $j = 0, \dots, k, j \neq i$. Each contribution can be seen to be:

(7) $$-\frac{f_i}{\phi_i^{(k)}(x_i)} \frac{1}{x_i - x_j} \qquad \text{for } j = 0, \dots, i-1, i+1, \dots, k.$$

The remaining k paths correspond to the x_i contributions in $x_j - x_i$. Each contribution can be seen to be:

(8) $$\frac{f_j}{\phi_j^{(k)}(x_j)} \frac{1}{x_j - x_i} \qquad \text{for } j = 0, \dots, i-1, i+1, \dots, k.$$

Hence from Equations (7, 8) for a single x_i, the sum of products of the magnification factors of all paths to the output is equal to

(9) $$S_{x_i} = -\sum_{\substack{j=0\\ j\neq i}}^{k} \frac{f_i}{\phi_i^{(k)}(x_i)} \frac{1}{x_i - x_j} + \sum_{\substack{j=0\\ j\neq i}}^{k} \frac{f_i}{\phi_j^{(k)}(x_j)} \frac{1}{x_j - x_i}.$$

Finally, there is a single path from each f_i to the output node contributing:

(10) $$S_{f_i} = 1/\phi_i^{(k)}(x_i).$$

From the definitions in Equations (5, 6):

$$\sigma_{k,D} = \sum_{i=0}^{k} (|S_{f_i}| \cdot |f_i| + |S_{x_i}| \cdot |x_i|),$$

hence multiplying each of the terms in Equations 9 and 10 with the input node values x_i and f_i respectively and adding the modulus of all such terms we obtain:

$$\sigma_{k,D} = \sum_{i=0}^{k} \frac{|f_i|}{\phi_i^{(k)}(x_i)|} + \sum_{i=0}^{k} \left| \sum_{\substack{j=0\\ j\neq i}}^{k} \left(\frac{f_i}{\phi_i^{(k)}(x_i)} + \frac{f_j}{\phi_j^{(k)}(x_j)} \right) \frac{x_i}{x_i - x_j} \right|$$

and the equality for the data condition number follows.

To derive the upper bound for the absolute rounding condition number, we must account for each of i) the subtraction nodes in the first step of the algorithm, ii) the multiplication nodes in the parallel prefix computation of the denominators, iii) the division nodes and finally iv) the addition nodes. From each of these nodes, there is only one path leading to the output, and hence each of the corresponding sums of products of the magnification factors consists of a single term, namely

$$S_{(-)} = P_{(-)}, \qquad S_{(\times)} = P_{(\times)}, \qquad S_{(/)} = P_{(/)}, \qquad S_{(+)} = P_{(+)}.$$

For example, for the subtraction node having the value $x_i - x_j$

$$S_{(-)} = -\frac{f_i}{\phi_i^{(k)}(x_i)} \frac{1}{x_i - x_j} \qquad \text{which leads to}$$

$$\sum_{\text{all}(-)\text{nodes}} |S_{(-)}|\,|value[(-)\,\text{node}]| = \sum_{i=0}^{k}\sum_{\substack{j=0\\ j\neq i}}^{k} \left|\frac{f_i}{\phi_i^{(k)}(x_i)}\frac{1}{x_i-x_j}\right|\cdot|x_i-x_j| \tag{11}$$

$$= k\sum_{i=0}^{k}|f_i|/|\phi_i^{(k)}(x_i)|$$

and similarly

$$\sum_{\text{all}(\times)\text{nodes}} |S_{(\times)}|\cdot|value[(\times)\,\text{node}]| = (k-1)\sum_{i=0}^{k}|f_i|/|\phi_i^{(k)}x_i)| \tag{12}$$

since there are $k-1\,(\times)$ nodes per prefix tree, and

$$\sum_{\text{all}(/)\text{nodes}} |S_{(/)}|\cdot|values[(/)\,\text{node}]| = \sum_{i=0}^{k}|f_i|/|\phi_i^{(k)}(x_i)|. \tag{13}$$

For $(+)$ nodes we note that $S_{(+)} = P_{(+)} = 1$ and that the value corresponding to each depends on its position on the addition tree. Next observe that the sum of products at a given level t of the addition tree satisfies

$$\sum_{\substack{\text{all}(+)\text{nodes}\\ \text{at level } t}} |S_{(+)}|\cdot|value[(+)\,\text{node}]| \le \sum_{i=0}^{k}|f_i|/|\phi_i^{(k)}(x_i)|.$$

Since the addition tree has $k+1$ leaves, and hence $\lceil\log(k+1)\rceil$ levels, we obtain

$$\sum_{\text{all}(+)\text{nodes}} |S_{(+)}|\cdot|value[(+)\,\text{node}]| \le \lceil\log(k+1)\rceil\sum_{i=0}^{k}|f_i|/|\phi_i^{(k)}(x_i)|. \tag{14}$$

For example, when $k+1$ is a power of 2

$$\sum_{\text{all}(+)\text{nodes}} |S_{(+)}|\cdot|value[(+)\,\text{node}]| =$$

$$\left|\frac{f_0}{\phi_0^{(k)}(x_0)}+\frac{f_1}{\phi_1^{(k)}(x_1)}\right| + \left|\frac{f_2}{\phi_2^{(k)}(x_2)}+\frac{f_3}{\phi_3^{(k)}(x_3)}\right| + \dots$$

$$+\left|\frac{f_{k-1}}{\phi_{k-1}^{(k)}(x_{k-1})}+\frac{f_k}{\phi_k^{(k)}(x_k)}\right| + \left|\frac{f_0}{\phi_0^{(k)}(x_0)}+\frac{f_1}{\phi_1^{(k)}(x_1)}+\frac{f_2}{\phi_2^{(k)}(x_2)}+\frac{f_3}{\phi_3^{(k)}(x_3)}\right| + \dots$$

$$+\left|\frac{f_{k-3}}{\phi_{k-3}^{(k)}(x_{k-3})}+\frac{f_{k-2}}{\phi_{k-2}^{(k)}(x_{k-2})}+\frac{f_{k-1}}{\phi_{k-1}^{(k)}(x_{k-1})}+\frac{f_k}{\phi_k^{(k)}(x_k)}\right| + \dots$$

$$+\left|\frac{f_0}{\phi_0^{(k)}(x_0)}+\frac{f_1}{\phi_1^{(k)}(x_1)}+\frac{f_2}{\phi_2^{(k)}(x_2)}+\frac{f_3}{\phi_3^{(k)}(x_3)}+\dots+\frac{f_k}{\phi_k^{(k)}(x_k)}\right|.$$

Adding the terms in Equations (11–14) we get

$$\sigma_{k,R} = \sum_{\text{all}(-)\text{nodes}} |S_{(-)}|\cdot|value[(-)\,\text{node}]| + \sum_{\text{all}(\times)\text{nodes}} |S_{(\times)}|\cdot|value[(\times)\,\text{node}]| +$$

$$+ \sum_{\text{all } (/) \text{ nodes}} |S_{(/)}| \cdot |value[(/)\text{ node}]| + \sum_{\text{all } (+) \text{ nodes}} |S_{(+)}| \cdot |value[(+)\text{ node}]|$$

and thus
$$\sigma_{k,R} \le (2k + \lceil \log(k+1) \rceil) \sum_{i=0}^{k} f_i / |\phi_i^{(k)}(x_i)|$$

and the upper bound for the absolute rounding condition number follows.

THEOREM 5.1. *The parallel Newton algorithm is numerically stable.*

PROOF. From Lemma 5.1 it immediately follows that

$$\sum_{i=0}^{k} |f_i| / |\phi_i^{(k)}(x_i)| \le \sigma_{k,D}.$$

Comparing $\sigma_{k,D}$ with $\sigma_{k,R}$ the inequality $\sigma_{k,R} \le A(k)\sigma_{k,D}$ follows with $A(k) = 2k + \lceil \log(k+1) \rceil$. Since this is valid for $k = 1, \ldots, n$, it follows from the definition of numerical stability, that the computation of any DD coefficient by means of parallel Newton is stable.

Intuitively, the factor $A(k)$ can be taken as a measure of the stability of the algorithm. In [22], an algorithm is classified as *unconditionally* stable if A is independent of k, while it is *conditionally* stable if A depends on k. An even finer classification was recently proposed by Gao in [9], by observing how A varies with k: Since $A(k) = O(k)$ parallel Newton interpolation is *linearly* stable. We emphasize that what we have shown is the numerical stability of the parallel algorithm, i.e. the coefficients of the interpolating polynomial in the Newton form can be computed in a stable way. This, however, does not salvage the overall numerical behavior of the polynomial interpolation problem which can be badly conditioned when n is large.

6. Numerical experiments.

The numerical experiments were performed on a Sun 3/50 running Berkely 4.2 Unix. The arithmetic is done in IEEE floating-point with a machine ε approximately equal to 10^{-7} for single precision.

First we choose points x_k and coefficients c_k to define a Newton polynomial

$$\sum_{k=0}^{n} c_k(x - x_0)\ldots(x - x_{k-1}) \tag{15}$$

and use Horner's algorithm to evaluate it at $x_k = -1 + 2k/(n-1)$ for $k = 0, \ldots, n-1$ in double precision. These values are in turn input to each of the interpolation algorithms (Aitken, Neville and parallel Newton) from which approximations $c_k^{(\text{computed})}$ for the DD's c_k are recovered. The interpolation algorithms are run in single precision arithmetic in order to highlight their numerical properties as much as possible. The difference $|c_k - c_k^{(\text{computed})}|$ is then a measure of the error for the corresponding algorithm. Figures 5a and 5b correspond to $n = 16$ and coefficients $c_k = (-1)^k/(k+1)$ and $c_k = 1/(k+1)^2$ respectively. They depict $|c_k - c_k^{(\text{computed})}|$ as a function of k. Since c_0 is available directly, results are shown only for $k > 0$. We see that the errors in the DD coefficients calculated with the new algorithm are very similar to the errors corresponding to the serial Neville and Aitken algorithms.

Figure 6 shows the effectiveness of polynomial interpolation to approximate a known function. As inputs we use the function values at $n + 1$ points computed in double precision. The rest of the computation is in single precision. Interpolating polynomials are produced by means of each of the i) Newton based algorithms (parallel, Aitken, Neville), and ii) the parallel algorithm of [21] which utilizes the Lagrange representation and FFT. The interpolating polynomials are calculated at the midpoints of the interpolation nodes and the error is computed by comparing with the known function values at those points. The algorithms used for evaluation are different for the sequential (Neville, Aitken) and parallel (Lagrange-FFT, parallel Newton) interpolation schemes. This is because the speed advantage of $O(\log n)$ in interpolation is meaningful only in context of an evaluation algorithm of the same complexity. For this reason, the polynomials constructed using the sequential algorithms are evaluated by means of Horner's scheme, whereas those constructed

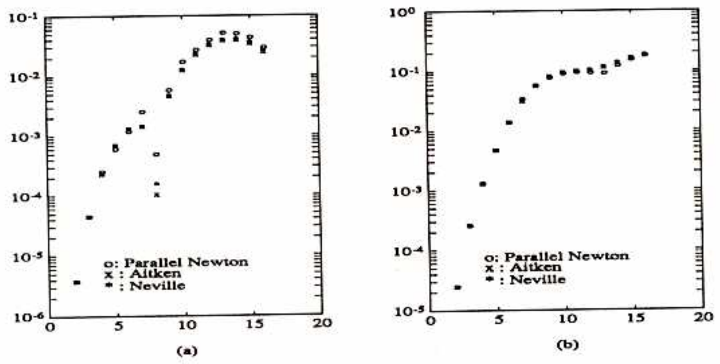

Fig. 5. DD error $|c_k - c_k^{(computed)}|$, $k = 0, 1, \ldots, 15$ for (a) $c_k = \dfrac{(-1)^k}{k+1}$ and (b) $c_k = \dfrac{1}{(k+1)^2}$.

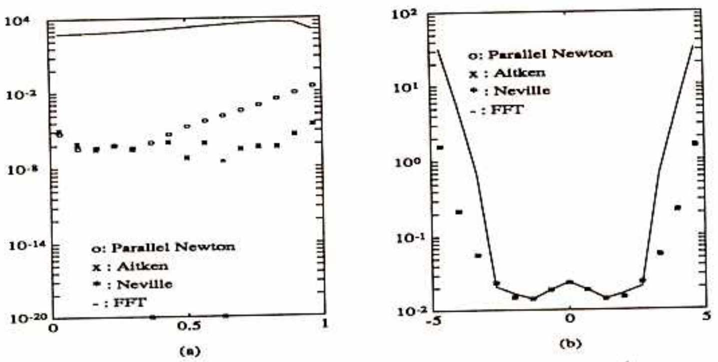

Fig. 6. Midpoint error when $n = 15$ for (a) $\ln(x)$ and (b) $\dfrac{1}{1+x^2}$.

using the parallel algorithms are evaluated as described in Section 4. Figure 6a shows the error at the midpoints, that is

$$|f(x_i + h/2) - P_n(x_i + h/2)| \tag{16}$$

as a function of x where $f(x) = \log_e(1 + x)$. The interpolation nodes are $x_i = ih$ for $0 \leq i \leq n-1$, $h = 1/(n-1)$ and $n = 16$. For ease of representation, any differences smaller than 10^{-20} were set equal to that value. Figure 6b shows the same experiment for the function $f(x) = 1/(1 + x^2)$ in the interval $[-5, 5]$. This function was used as an example by Runge [3] for which the interpolating polynomial diverges from f outside the interval $|x| \leq 3.63..$ for any choice of equidistant interpolation points, (this result is of course independent of our "algorithmic" discussions). In this case, we choose a different order for the interpolation nodes, namely $x_i = -5 + ih$ for $i = 0, 2, \ldots, n-2$ and $x_i = ih$ for $1, 3, \ldots, n-1$ with $h = 5/(n-1)$ (n is taken even). Unlike Lagrange interpolation, Newton interpolation is sensitive to the order of the interpolation nodes and this interleaved order was chosen to reduce the error.

Figure 6a indicates that in that case the combination of the parallel computation of the DD and parallel polynomial evaluation in single precision is slightly inferior for values of x to the right of the middle of the interval. The figures also indicate that, for single precision,these algorithms are superior to the FFT based parallel algorithm for the case of equidistant interpolation. This may be due to the use of the FFT for polynomial multiplication in the latter ([10]). Nevertheless, a rigorous analysis of the FFT based interpolation algorithms remains to be done.

It could be argued that by the time the proposed parallel algorithm becomes effective, polynomial interpolation breaks down due to ill-conditioning. Recent studies have shown that a suitable choice of interpolation nodes would make high-order Newton interpolation numerically practical. In the next experiment we test our parallel algorithm for high-order problems, using the results of [7, 26]. The next set of experiments were conducted using MATLAB (hence double precision) on a Sun 3/50. We interpolated the "shifted" Runge function, $f(x) = 1/(1 + 25x^2/4)$, in the interval $[-2, 2]$ using a polynomial of degree n. We used two node sequences.

1. Nodes x_k^c are constructed as follows: Set $c_0 = 0$, $x_0^c = -2$ and for $k = 1, \ldots, n$ take the binary representation

   $$k = \sum_{j=0}^{\infty} \kappa_j 2^j, \qquad \kappa_j \in \{0, 1\}$$

   and define $$c_k = \sum_{j=0}^{\infty} \kappa_j 2^{-j-1}.$$

   Then $$x_k^c = 2\cos(\pi c_{k-1}).$$

   Obviously c_k is the van der Corput sequence whose favorable properties for Newton interpolation were recently discussed in [7, 20].
2. Nodes x_k^e for $k = 0, \ldots, n$ are equidistant: $x_k^e = -2 + 4k/n$.

We used $n = 70$, and 127 and plotted in each case the magnitude of the DD coefficients (Fig. 7, Fig. 9), and error (16) (Fig. 8, Fig. 10). Both the computation of the DD and the polynomial evaluations were done using the parallel Newton algorithms discussed in this paper. Table 2 also shows the maximum magnitudes of the DD and of the error for each sequence, as well as for $n = 30$ and 128. This

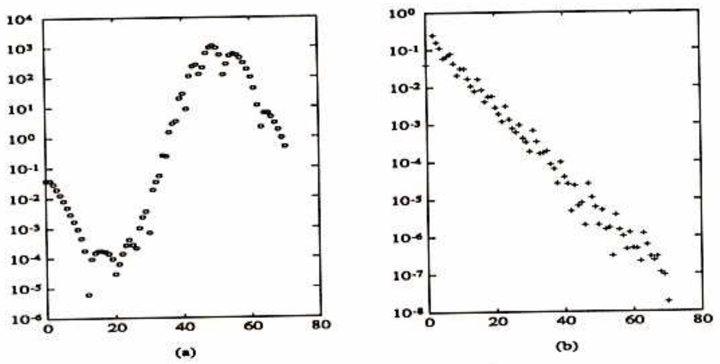

Fig. 7. Magnitude of DD coefficients for interpolation of $\dfrac{1}{1+(2.5x)^2}$ on (a) x^e and (b) x^c nodes, for $n = 70$.

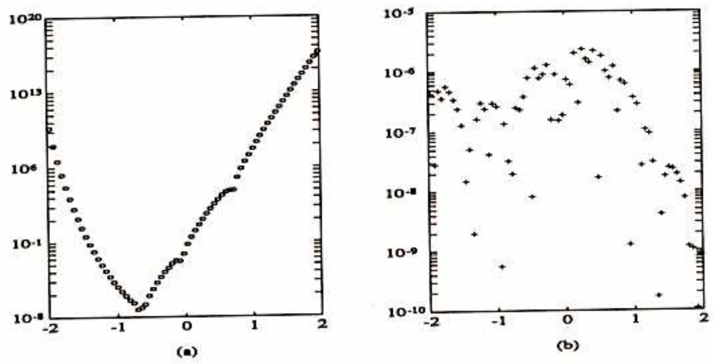

Fig. 8. Midpoint error for parallel Newton interpolation of $\dfrac{1}{1+(2.5x)^2}$ based on (a) x^e and (b) x^c nodes, for $n = 70$.

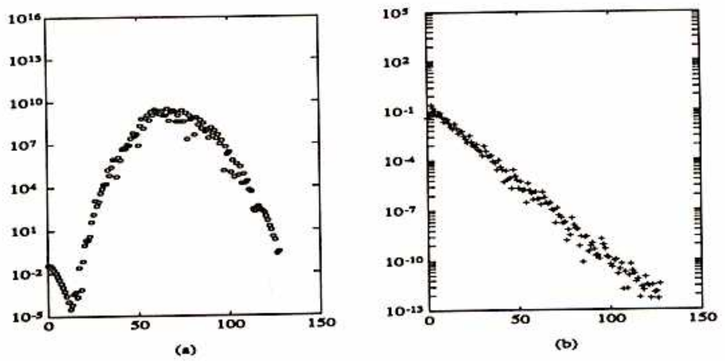

Fig. 9. Magnitude of DD cofficients for interpolation of $\dfrac{1}{1+(2.5x)^2}$ on (a) x^e and (b) x^c nodes, for $n = 127$.

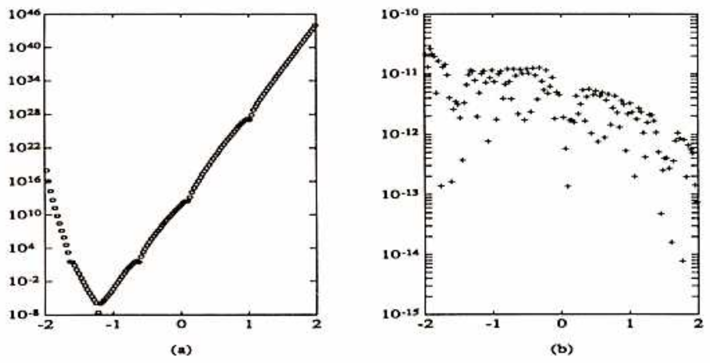

Fig. 10. Midpoint error for parallel Newton interpolation of $\frac{1}{1+(2.5x)^2}$ based on (a) x^e and (b) x^c nodes, for $n = 127$.

confirms the results in [7,26], that $\{x^c\}$ is a much more attractive set to use than $\{x^e\}$. More importantly for this discussion, it demonstrates that high-order interpolation is possible, rendering the use of a parallel algorithm very desirable. When $n = 127$ for example, we see from Theorem 3.1 that given parallel prefix hardware, we can compute the DD using a circuit of depth 16. This contrasts with $2n = 254$ parallel operations in order to go through the $n = 127$ stages of a systolic array designed to compute the DD entries and with approximately $\frac{3}{2}n(n+1) = 24{,}384$ operations for the sequential algorithms.

In experiments not reported here we found that there was no practical difference in the quality of the results when the same experiments were repeated but this time using sequential algorithms for the computation of the DD and polynomial evaluation.

Table 2. *Parallel Newton high-order interpolation*

m	*Max. DD for x^e*	*Max. DD for x^c*	*Max. error for x^e*	*Max. error for x^c*
30	8.842E + 00	2.404E − 01	1.425E + 03	1.970E − 02
70	1.099E + 03	2.404E − 01	4.677E + 16	2.291E − 06
127	3.364E + 09	2.404E − 01	9.882E + 43	2.603E − 11
128	2.943E + 09	2.404E − 01	4.211E + 42	7.861E − 12.

7. Summary and conclusions.

We have presented a parallel algorithm to construct the interpolating polynomial in its Newton form by fast evaluation of divided differences using the parallel prefix algorithm. Given $n + 1$ input pairs, the algorithm requires $2\lceil \log(n+1) \rceil + 2$ parallel arithmetic steps and circuit size $O(n^2)$ This parallel computational complexity is better than the parallel implementations of the Neville and Aitken algorithms and the FFT based, Lagrange interpolation algorithms described in the literature. The computation of the divided differences required by the algorithm for the construction of the interpolating polynomial is then shown to be numerically stable. Furthermore, no precomputation with the data or equidistant interpolation points are required.

The interpolating polynomial in its Newton form can be evaluated by means of a fast parallel algorithm having the same characteristics. A natural consequence is circuit similarity in VLSI implementation. Furthermore, it is shown in [4] that the parallel Newton interpolation algorithm can be implemented on a cube-connected system using only twice the number of parallel routing steps as would be required by a fully connected multiprocessor system. Similar techniques are applicable to the computation of generalized divided differences for parallel Hermite interpolation as well [6,5].

It is worth noting that the evaluation of the products may cause overflow or underflow. Therefore, some care may be required in the implementation. Finally there seems to be no direct carry-over of the permanence property of the Newton representation which is valuable in the serial context: the parallel version as presented here requires $O(\log n)$ time to update the coefficients when a new point is added to the data set.

REFERENCES

[1] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, Dover, 1965.

[2] A. Bilgory and D. Gajski, *A heuristic for suffix solutions*, IEEE Trans. Comput., C-35 (January 1986), pp. 34–42.

[3] P. J. Davis, *Interpolation and Approximation*, Dover, 1975.

[4] O. Eğecioğlu, E. Gallopoulos, and Ç. Koç, *Fast and practical parallel polynomial interpolation*, Tech. Rep. 646, Center for Supercomputing Research and Development, January 1987.

[5] O. Eğecioğlu, E. Gallopoulos, and Ç. Koç, *Fast computation of divided differences and parallel Hermite interpolation*, J. Complexity, (to appear).

[6] O. Eğecioğlu, E. Gallopoulos, and Ç. Koç, *Parallel Hermite interpolation: an algebraic approach*, Computing, (to appear).

[7] B. Fischer and L. Reichel, *Newton interpolation in Fejér and Chebyshev points*, Math. Comp., 53 (1989), pp. 265–278.

[8] S. Fortune and J. Wyllie, *Parallelism in Random Access Machines*, in Proc. 10th ACM Symp. Theory of Computing, San Diego, CA., May 1978.

[9] G. R. Gao, *A stability classification scheme and its application to pipelined solution of linear recurrences*, Parallel Comput., 4 (June 1987), pp. 305–322.

[10] E. Horowitz, *A fast method for interpolating using preconditioning*, IFIP Letters, 1 (1972), pp. 157–163.

[11] F. Krogh, *Efficient algorithms for polynomial interpolation and divided differences*, Math. Comp., 24 (January 1970), pp. 185–190.

[12] C. P. Kruskal, L. Rudolph, and M. Snir, *The power of parallel prefix*, IEEE Trans. Comput., C-34 (October 1985), pp. 965–968.

[13] H. T. Kung, *Fast evaluation and interpolation*, Tech. Rep., Department of Computer Science, Carnegie-Mellon University, January 1973.

[14] R. Ladner and M. Fischer, *Parallel prefix computation*, J. Assoc. Comput. Mach., 27 (October 190), pp. 831–838.

[15] A. Macleod, *A comparison of algorithms for polynomial interpolation*, J. Comput. Appl. Math., 8 (1982), pp. 275–277.

[16] K. Maruyama, *On the parallel evaluation of polynomials*, IEEE Trans. Comput., C-22 (January 1973), p. 25.

[17] G. P. McKeown, *Iterated interpolation using a systolic array*, ACM Trans. Math. Softw., 12 (June 1986), pp. 162–170.

[18] W. Miller, *Computational complexity and numerical stability*, SIAM J. Comput., 4 (June 1975), pp. 97–107.

[19] I. Munro and M. Paterson, *Optimal algorithms for parallel polynomial evaluation*, J. Comput. Sys. Sci., 7 (1973), pp. 189–198.

[20] L. Reichel and G. Opfer, *Chebyshev-Vandermonde systems*, Tech. Rep. 88/48, IBM Bergen Scientific Centre, November 1988.

[21] J. Reif, *Logarithmic depth for algebraic functions*, SIAM J. Comput., 15 (February 1986), pp. 231–242.

[22] W. Rönsch, *Stability aspects in using parallel algorithms*, Parallel Comput., 1 (August 1984), pp. 75–98.

[23] K. Singhal and J. Vlach, *Accuracy and speed of real and complex interpolation*, Computing, 11 (1973), pp. 147–158.

[24] M. Snir, *On parallel search*, in Ottawa Conf. Distr. Comput., August 1982.

[25] F. Stummel, *Perturbation theory for evaluation algorithms of arithmetic expressions*, Math. Comp., 37 (October 1981), pp. 435–473.

[26] H. Tal-Ezer, *High degree polynomial interpolation in Newton form*, Tech. Rep. 88-39, ICASE, 1988.