UNIVERSITY OF CALIFORNIA

Santa Barbara

Parallel Algorithms for Interpolation and Approximation

A Dissertation submitted in partial satisfaction

of the requirements for the degree of

Doctor of Philosopy

in

Electrical and Computer Engineering

by

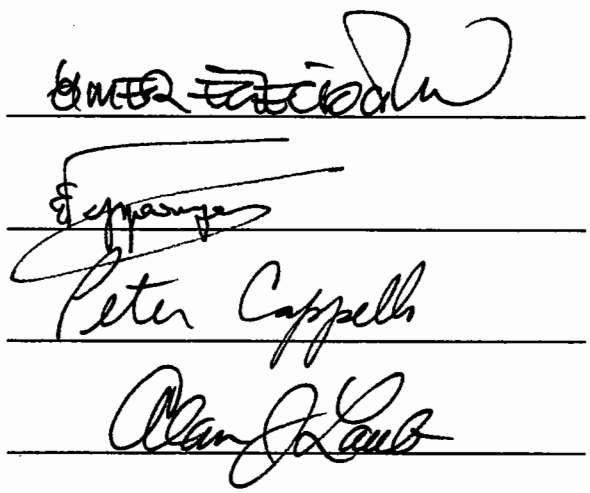
Çetin Kaya Koç

Committee in charge :

Professor	Alan J. Laub, Chairman
Professor	Ömer Eğecioğlu
Professor	Efstratios Gallopoulos
Professor	Peter R. Cappello

June 1988

The dissertation of Çetin Kaya Koç
is approved



Committee Chairperson

June 1988

Gördüğün herhangi birşey
*Görülecek bir başka şeyi gösterir olsa olsa**

Armalar, Edip Cansever, 1984.

Acknowledgements

I would like to thank Stratis Gallopoulos for his criticism, his time, and his friendship, and also for his and Toula Gallopoulos' hospitality in Santa Barbara and Urbana-Champaign.

I am indebted to Alan Laub and Peter Cappello for providing financial support and equipment. As committee members they were more than helpful, and devoted a lot of their time to refine the ideas presented in this dissertation.

Ömer Eğecioğlu has been the *sine qua non* person in my research. As I spent hours in Espresso Roma trying to follow his reasoning, I have come to appreciate and enjoy the process of *doing* science rather than its end-products, more than ever before.

I would also like to thank all members of the Scientific Computation Laboratory for their friendship and support. I am indebted to Bob Brandt and Charles Kenney for reading parts of this dissertation.

Laurie Collins, my dear friend, has also read some parts of it and provided valuable criticism.

I dedicate this dissertation to my family for their love, patience, and tolerance made it possible.

Çetin K. Koç

* *What is seen merely points out that which shall be seen.*

Vita

January 23, 1957	Born in Karaköse, Ağrı, Turkey
9/1980	*Bachelor of Science* Department of Electrical Engineering Istanbul Technical University, Istanbul, Turkey
9/1982	*Master of Science* Department of Electrical Engineering Istanbul Technical University, Istanbul, Turkey
1/1982 - 1/1983	Design Engineer Northern Telecom, Istanbul, Turkey
4/1984 - 6/1988	Teaching Assistant & Associate, and Research Assistant Department of Electrical and Computer Engineering University of California, Santa Barbara
12/1985	*Master of Science* Department of Electrical and Computer Engineering University of California, Santa Barbara
1/1988 - 4/1988	Visiting Lecturer Department of Computer Science University of California, Santa Barbara
6/1988	*Doctor of Philosophy* Department of Electrical and Computer Engineering University of California, Santa Barbara

Publications

Ö. Eğecioğlu, Ç. K. Koç, and A. J. Laub, "A Recursive Doubling Algorithm for Solution of Tridiagonal Systems on Hypercube Multiprocessors," to appear in *Journal of Computational and Applied Mathematics.*

Ö. Eğecioğlu and Ç. K. Koç, "A Fast Algorithm for Rational Interpolation via Orthogonal Polynomials," to appear in *Mathematics of Computation.*

Ö. Eğecioğlu, E. Gallopoulos, and Ç. K. Koç, "Fast and Practical Parallel Polynomial Interpolation," to appear in *Journal of Parallel and Distributed Processing.*

Ö. Eğecioğlu, Ç. K. Koç, and A. J. Laub, "Prefix Algorithms for Tridiagonal Systems on Hypercube Multiprocessors," to appear in *Proceedings of The Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, California, January 19 - 20, 1988.

P. Cappello, G. Davidson, A. Gersho, Ç. K. Koç, and V. Somayazulu, "A Systolic Vector Quantization Processor for Real-Time Speech Coding," *Proceedings of the IEEE International Conference on Acoustic, Speech, and Signal Processing*, Tokyo, Japan, April 8 - 11, 1986, Vol. 3, pp. 2143 - 2146.

P. R. Cappello, E. Gallopoulos, and Ç. K. Koç, "Systolic Computation of Interpolating Polynomials," unpublished report, Department of Computer Science, University of California, Santa Barbara, June 1988.

Ç. K. Koç and P. F. Ordung, "Schwarz-Christoffel Transformation for Simulation of Two-Dimensional Capacitance," Department of Electrical and Computer Engineering, University of California, Santa Barbara, Technical Report No. 88-03, February 1988.

Ö. Eğecioğlu and Ç. K. Koç, "An $O(N \log N)$ Parallel Algorithm for Rational Interpolation," Department of Computer Science, University of California, Santa Barbara, Technical Report No. TRCS88-2, January 1988.

Ö. Eğecioğlu, E. Gallopoulos, and Ç. K. Koç, "Parallel Hermite Interpolation: An Algebraic Approach," Department of Computer Science, University of California, Santa Barbara, Technical Report No. TRCS86-27, November 1986.

Ö. Eğecioğlu, E. Gallopoulos, and Ç. K. Koç, "Parallel Hermite Interpolation: An Analytical Approach," unpublished report, November 1986.

Fields of Study

Major Field : Electrical Engineering and Computer Science
Studies in Parallel Algorithms and Architectures
Studies in Scientific Computing
Studies in Applied Mathematics

ABSTRACT

Parallel Algorithms for Interpolation and Approximation

by

Çetin Kaya Koç

A collection of practical parallel algorithms for interpolation and approximation of discrete data by polynomials and rational functions is introduced. The algorithms are suitable for implementation in systolic computing systems, on general purpose parallel computers, and in VLSI.

First we show that, by suitably embedding the process dependence graph of the Aitken algorithm in space-time and by using non-linear transformations, time-optimal and spacetime-optimal systolic arrays can be obtained to compute the divided differences which are the coefficients of the Newton and Hermite interpolating polynomials. Two-dimensional versions of the systolic arrays allow us to compute the multivariate divided differences optimally in spacetime. For the case where the size of the systolic computing system is smaller than the size of the problem, we describe an efficient algorithm based on the partitioning of the Newton-Vandermonde matrix. The techniques easily apply to the Neville algorithm producing similar results.

Next we show that parallel prefix algorithms can be applied to solve a broad class of interpolation problems in logarithmic time. This approach constitutes an alternative to the use of the FFT, and yields a class of algorithms to compute the coefficients of the Newton and Hermite interpolating polynomials, and to solve tridiagonal and banded linear systems which appear in the construction of cubic or higher degree splines. For the Newton and Hermite interpolation, the algorithms rely on linear expansions of the divided differences in terms of the given function and derivative values. The special cases where up to first and second order derivative information is available are treated

in detail. The proof for the case of arbitrarily high order derivative information involves combinatorial arguments that make use of the theory of symmetric functions. The parallel Newton interpolation algorithm is shown to be numerically stable. Experiments indicate that a floating-point implementation results in error accumulation similar to that of the widely used serial algorithms. Efficient implementation of the parallel prefix algorithms on the hypercube makes the techniques especially attractive.

Finally, a new rational interpolation algorithm (and its parallel version) based on orthogonal polynomials and the Newton interpolation is described.

Table of Contents

0

Introduction

This dissertation is concerned with *parallel* and *practical* algorithms for interpolation and approximation of discrete data. With the advances in VLSI design and the commercialization of parallel computing systems, a considerable amount of parallel computing capabilities have become accessible in university and industrial environments, making the design and implementation of parallel algorithms an essential step. Two broad classes of commercially available parallel computing systems are of interest (1) general-purpose parallel processors (e.g. *Intel hypercube*, *Alliant*, and *Sequent*), and (2) special-purpose co-processors (e.g. *The Function Machine* by *Electronic Associates*). In the coming years, the users of general-purpose parallel processors will inevitably demand practical parallel algorithms in all areas of scientific computing. The development of special-purpose co-processors, which consist of a

collection of function boxes clustered around the main bus of a very fast serial computer, requires that mathematical software library procedures be replaced by *mathematical hardware* procedures. The idea of creating a *Mathematical Hardware Library* (MHL) [Hell85] consisting of special-purpose chips (hardwired equivalents of the library subroutines) has been the essence of many research efforts in the area of VLSI computing. Several expositions have appeared in the literature describing the implementation of algorithms in special-purpose computing environments. Examples are special units to perform matrix computations, convolutions, and other computations of interest in signal processing applications. Nevertheless, there is a great potential for algorithms for other areas of numerical computing (e.g. curve and surface fitting, solution of non-linear equations, numerical integration and differentiation, etc.). A complete hardware library should also be capable of coping with these areas.

Regarding *practicality* of the parallel algorithms we note that this issue is crucially important, since in the quest for fast algorithms there is little value in studying those which are numerically unstable [Mill75]. In this dissertation, we approach the subject from the numerical point of view rather than the computational complexity point of view, and focus on the stability aspects of parallel algorithms.

Polynomials are the most widely used class of functions to approximate discrete data, and in this work we consider polynomials and rational functions as approximating functions. We are interested in approximation for the following reasons.

i) It is widely used for curve or surface fitting or, in general, the fitting of multidimensional curves to scattered data. Computer graphics and finite element methods are some areas where approximation is widely and intensively used. In many applications of surface-fitting techniques, the ultimate aim is to use the data to construct a *contour map*

of the unknown function. Since the function is known only at the data points, one must construct a contour map for one of the fitted surfaces. Contour map construction has applications in the oil industry (petroleum explorations), geological maps and cardiology (heart potentials) [Schu76].

ii) A "hard" function can be replaced by an "easy" function via approximation or interpolation. This "easy" function can then be manipulated (i.e. integrated or differentiated) much more easily and effectively [BeZh65].

iii) Inverse interpolation algorithms can be used for the solution of non-linear equations [BeZh65], [Lark81], [NeSc85], [Trau82].

The interaction between computing sciences and approximation theory, and the impact of computers on approximation techniques have created the need for approximation, and provided the means for fulfilling this need [Davi65]. We believe that the design and implementation of parallel algorithms for approximation techniques will further advance this interaction.

Some additional background information may be helpful to the reader before we continue. The simplest case of approximation is when a set of pairs of points $(x_i, f_i) \in F \times F$ for $0 \le i \le n$ is approximated by a polynomial of degree n, $p_n(x) \in F[x]$ where F is field. If the points f_i are the values of a function $f(x)$ at the points x_i for $0 \le i \le n$, then $p_n(x)$ approximates the value of $f(x)$ for some x, i.e.

$$f(x) \approx \sum_{k=0}^{n} a_k x^k$$

The error associated with this approximation will be

$$e(x) = f(x) - p_n(x)$$

The different criteria of choosing the constants a_k lead to three types of approximation (of

major importance) :

I. *Interpolation* : The constants, a_k for $0 \le k \le n$, are chosen so that for a fixed set of points x_i $0 \le i \le n$, the value of the approximating polynomial is equal to the value of the function

$$p_n(x_i) = f(x_i) \quad , \quad 0 \le i \le n \ .$$

II. *Least-Squares Approximation* : Here we try to choose the a_k 's such that

$$\|e(x)\|_2^2 = \sum_{i=0}^{n} e^2(x_i)$$

is minimized.

III. *Uniform or Chebyshev Approximation* : Here we try to minimize

$$\|e(x)\|_\infty = \max(|e(x_i)|) \quad , \quad 0 \le i \le n$$

by choosing the a_k 's.

In this dissertation we introduce a collection of parallel algorithms for interpolation of discrete data by polynomials and rational functions. Algorithms for constructing a polynomial (or a rational function) to fit the given data in the Chebyshev sense use interpolation at each step of iteration. These types of algorithms, termed as *exchange algorithms*, start with a subset of the node points and construct the function interpolating this subset. The algorithm proceeds by exchanging a point from the subset with another not in the subset until the best fit is achieved [Stie56], [DeMa74], [RaRa78]. At each step polynomial or rational *interpolation* is used to construct the approximating function. Thus, parallel interpolation algorithms are essential for parallel implementation of Chebyshev approximation algorithms.

In Chapter 1, we show that the divided differences, which are the coefficients of the Newton and Hermite interpolating polynomials, can be computed optimally on systolic computing systems. The classical nature of the Aitken and the Neville algorithms to compute the

divided differences makes this well-justified. These algorithms have been thoroughly studied and widely implemented. Much is known about their numerical properties, and they are widely preferred to other types of interpolation algorithms, especially those which use the Fast Fourier Transform. It is shown that by suitably embedding the process dependence graph of the Aitken algorithm in space-time, time-optimal and spacetime-optimal systolic arrays can be obtained. Variations of spacetime-optimal arrays are produced by using different nonlinear transformations. The spacetime optimal linear arrays use $\lceil n/2 \rceil$ processors to compute the interpolating Newton polynomial of degree n in $2n-1$ steps. We also show that these arrays can be used to compute the generalized divided differences which are the coefficients of the Hermite interpolating polynomial. Two-dimensional versions of the systolic arrays allow us to compute the multivariate generalized divided differences optimally in spacetime. The techniques easily apply to the Neville algorithm producing similar results. For the case where the size of systolic computing system is smaller than the size of the problem, we describe an efficient algorithm based on the partitioning of the Newton-Vandermonde matrix. We also show that by re-programming the processors the interpolating polynomials can be implicitly evaluated.

The results of Chapter 1 will appear as a technical report at the Department of Computer Science, University of California, Santa Barbara.

In Chapter 2, a new parallel algorithm for Newton interpolation is presented. The algorithm makes use of 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 $2\lceil \log(n+1) \rceil + 2$ parallel arithmetic steps, and circuit size $O(n^2)$. The interpolation algorithm is shown to be numerically stable. Experiments indicate that a floating-point implementation results in error accumulation similar to

that of the widely used serial algorithms. An efficient implementation of the algorithm on a hypercube parallel computer is also exhibited. Further advantages of the algorithm are that it does not require equidistant points, preconditioning, or use of the Fast Fourier Transform.

The results of Chapter 2 appeared as "Fast and Practical Parallel Polynomial Interpolation," Technical Report No. 646, January 1987, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, and has been accepted for publication in *Journal of Parallel and Distributed Computing*.

Given $n+1$ distinct points and arbitrary order derivative information at these points, a parallel algorithm to compute the coefficients of the corresponding Hermite interpolating polynomial in $O(\log n)$ time using $O(n^2)$ processors is presented in Chapter 3. The algorithm relies on a novel closed formula that yields the coefficients when the generalized divided differences are expressed linearly in terms of the given function and derivative values. We show that each one of these coefficients and the required linear combinations can be evaluated efficiently. The particular cases where up to first and second order derivative information is available are treated in detail. The proof of the general case, where arbitrarily high order derivative information is available, involves combinatorial arguments that make use of the theory of symmetric functions.

The results of Chapter 3 appeared as "Parallel Hermite Interpolation: An Algebraic Approach," Technical Report No. TRCS86-27, November 1986, Department of Computer Science, University of California, Santa Barbara. It was presented at *The First International Conference on Industrial and Applied Mathematics*, Paris, France, June 29 - July 3 1987.

Due to the Runge effect, polynomials of higher degree can not be used for approximation of certain functions [Davi75]. In this case the most effective approach is to use cubic

splines or splines of higher degree. A spline curve is one which is a piecewise polynomial and which has an appropriate amount of continuity where the pieces join. The algorithms for construction of cubic splines to fit the given data set solve a set of tridiagonal equations [AhNW67]. In Chapter 4, we show that the recursive doubling algorithm as developed by Stone can be used to solve a tridiagonal linear system of size n in $O(\log n)$ steps on a parallel computer with n processors. Here, we give a limited processor version of the recursive doubling algorithm for the solution of tridiagonal linear systems using $O(\frac{n}{p} + \log p)$ parallel arithmetic steps on a parallel computer with $p < n$ processors. The main technique relies on parallel prefix algorithms, which can be efficiently mapped on the hypercube architecture using the binary-reflected Gray code. For $p << n$ this algorithm achieves linear speed-up and constant efficiency over its sequential implementation as well as over the sequential LU decomposition algorithm. These results are confirmed by numerical experiments obtained on an Intel iPSC/d5† hypercube multiprocessor.

The results of Chapter 4 appeared as "A Recursive Doubling Algorithm for Solution of Tridiagonal Systems on Hypercube Multiprocessors," Technical Report No. TRCS88-1, January 1988, Department of Computer Science, University of California, Santa Barbara. It was presented at *The Third Conference on Hypercube Concurrent Computers and Applications*, January 19-20 1988, Pasadena, California, and *The Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1-4 1987, Los Angeles, California, and has been accepted for publication in *Journal of Computational and Applied Mathematics*.

In Chapter 5, a new algorithm for rational interpolation is proposed. Given the data set, the algorithm generates a set of orthogonal polynomials by the classical three-term recurrence

† iPSC is a trademark of Intel Corporation.

relation and then uses Newton interpolation to find the numerator and the denominator polynomials of the rational interpolating function. The number of arithmetic operations required by the algorithm to find a particular rational interpolant is $O(N^2)$, where $N+1$ is the number of data points. A variant of this algorithm that avoids Newton interpolation can be used to construct all rational interpolants using only $O(N^2)$ arithmetic operations. The parallel versions of the algorithms require $N+1$ processors to construct a particular rational interpolant in $O(n \log N)$ arithmetic steps, or all rational interpolants in $O(N \log N)$ arithmetic steps.

The results of Chapter 5 appeared as "A Fast Algorithm for Rational Interpolation via Orthogonal Polynomials," Technical Report No. TRCS87-6, May 1987, and "An $O(N \log N)$ Parallel Algorithm for Rational Interpolation," Technical Report No. TRCS88-2, January 1988, Department of Computer Science, University of California, Santa Barbara. The first technical report has been accepted for publication in *Mathematics of Computation*. These results were presented at *SIAM 35th Anniversary Meeting*, October 12-15 1987, Denver, Colorado.

1

Systolic Computation of Interpolating Polynomials

It is shown that by suitably embedding the process dependence graph of the Aitken algorithm in spacetime, time-optimal and spacetime-optimal systolic arrays can be obtained. Variations of spacetime-optimal arrays are produced by using different nonlinear transformations. We also show that these arrays can be used to compute the generalized divided differences which are the coefficients of the Hermite interpolating polynomial. Two-dimensional versions of the systolic arrays allow us to compute the multivariate generalized divided differences very efficiently. The techniques apply to the Neville algorithm, producing similar results. For the case where the size of systolic computing system is smaller than the size of the problem, we describe an efficient algorithm based on the partitioning of the Newton-Vandermonde matrix.

1.1. INTRODUCTION

McKeown [McKe86] establishes a rationale for performing iterated interpolation using a systolic array:

> *"Another consequence of developments in VLSI technology is that interpolation in a table as a means of function approximation could well rehabilitate itself, at least for certain 'difficult' functions. . . . the continually falling cost and increasing capacity of semiconductor memory chips could soon make it feasible to consider storing on such a chip a table of values for a function requiring complex direct computation. The set of 'chip tables' together with a systolic array for iterated linear interpolation could thus have attractive possibilities.*

He presents a systolic implementation of Aitken's method of iterated interpolation. In this Chapter, we consider a systolic version of Newton and Hermite polynomial interpolation using the algorithms of Aitken and Neville. This work builds on that of McKeown by:

i) presenting several alternative systolic implementations of Aitken's algorithm, some of which are optimal;

ii) applying these implementations to Neville's algorithm;

iii) presenting 2-level systolic arrays for computing generalized divided differences;

iv) presenting a systolic scheme for the multivariate case;

v) presenting a scheme, based on the mathematics of the problem, for 'partitioning' a large interpolation problem onto smaller systolic arrays;

vi) showing that the interpolating polynomials can be implicitly evaluated by only re-programming the processors.

1.2. NEWTON AND HERMITE POLYNOMIAL INTERPOLATION

The polynomial interpolation problem is defined as follows:

Input:

$n+1$ pairs of points $(x_i, f_i) \in F \times F$, $0 \le i \le n$, where F is a field.

Problem:

Find a polynomial $p_n(x) \in F[x]$ of degree n such that $p_n(x_i) = f_i$, $0 \le i \le n$.

The interpolating polynomial of degree n exists and is unique, provided that $x_i \neq x_j$ for $i \neq j$. Polynomial interpolation is applied to the problem of function approximation. If the points f_i are the values of a function $f(x)$ at the points x_i for $0 \le i \le n$, then the interpolating polynomial $p_n(\bar{x})$ is used to approximate the value of $f(\bar{x})$ for some $\bar{x} \neq x_i$ for $0 \le i \le n$, but which usually is in the same interval as the x_i 's.

The polynomial of degree n passing through the points (x_i, f_i) for $0 \le i \le n$ is given in the Newton form as

$$p_n(x) = f_0 + f_{01}(x - x_0) + f_{012}(x - x_0)(x - x_1) + \cdots + f_{012..n}(x - x_0)(x - x_1) \cdots (x - x_{n-1}), \tag{1.1}$$

where the coefficients $f_{012..i}$ are called the *divided differences*. The coefficients of the Newton polynomial interpolating the function $f(x)$ at the node points x_i for $0 \le i \le n$ can be computed using the well-known algorithms of Neville and Aitken [Hild56], [Krog70]. These algorithms use recursions to compute what is known as the *divided difference table* whose diagonal entries are the desired coefficients of the Newton interpolating polynomial. The Aitken algorithm uses the recursion:

$$f_{012..i,k} = \frac{f_{012..i} - f_{012..i-1,k}}{x_i - x_k} \tag{1.2}$$

for $0 \le i \le n-1$ and $i+1 \le k \le n$. The Neville algorithm uses the recursion:

$$f_{i,i+1,..,k} = \frac{f_{i,i+1,..,k-1} - f_{i+1,i+2,..,k}}{x_i - x_k} \tag{1.3}$$

for $0 \le i \le n$ and $i+1 \le k \le n$.

For Hermite interpolation (the confluent case for Newton interpolation), we construct a polynomial that passes through the values of $f(x)$, as well as its derivatives, at the node points x_i. Let $\mathbf{m}$ be an ordered collection of integers such that

$$\mathbf{m} = (m_0, m_1, \cdots, m_n) \text{ with } 1 \le m_i, \text{ for } 0 \le i \le n.$$

We denote the kth derivative of $f(x)$ evaluated at x_i by $f^{(k)}(x_i)$, with $f^{(0)}(x_i) = f_i$. If we are given

$$x_i, f_i, f^{(1)}(x_i), f^{(2)}(x_i), \cdots, f^{(m_i)}(x_i)$$

for all $0 \le i \le n$, then we can construct the Hermite polynomial to interpolate this data set. That is, we can construct

$$p_N^{(k_i)}(x_i) = f^{(k_i)}(x_i), \text{ for } 0 \le i \le n, \ 1 \le k_i \le m_i.$$

The Hermite interpolating polynomial also is unique, provided $x_i \ne x_j$ for $i \ne j$ as is the case for Newton interpolation [BeZh65]. The degree of the Hermite interpolating polynomial is

$$N = m_0 + m_1 + \cdots + m_{n-1} + m_n - 1 \ . \tag{1.4}$$

The Hermite interpolating polynomial can be given as

$$\begin{aligned} p_N(x) = {} & f_{0^1} + f_{0^2}(x-x_0) + f_{0^3}(x-x_0)^2 + \cdots + f_{0^{m_0}}(x-x_0)^{m_0-1} + \\ & f_{0^{m_0}1}(x-x_0)^{m_0} + f_{0^{m_0}1^2}(x-x_0)^{m_0}(x-x_1) + \cdots + f_{0^{m_0}1^{m_1}}(x-x_0)^{m_0}(x-x_1)^{m_1-1} + \\ & f_{0^{m_0}1^{m_1}2}(x-x_0)^{m_0}(x-x_1)^{m_1} + f_{0^{m_0}1^{m_1}2^2}(x-x_0)^{m_0}(x-x_1)^{m_1}(x-x_2) + \cdots + \cdots \\ & f_{0^{m_0}1^{m_1}2^{m_2}\cdots n^{m_n}}(x-x_0)^{m_0}(x-x_1)^{m_1}(x-x_2)^{m_2}\cdots(x-x_n)^{m_n-1}. \end{aligned} \tag{1.5}$$

The simplest instance of the Hermite interpolating polynomial is when $\mathbf{m} = (1, 1, \cdots, 1)$, which corresponds to the Newton interpolating polynomial. The coefficients of this polynomial are referred to as the *generalized divided differences* [Hild56], [TsPr78], [Krog70]. The Aitken and Neville recursions can be used to compute the generalized divided differences, provided that we avoid division by zero in the recursion formulae (1.2) and (1.3) by defining

$$f_{ii\ldots i} = \frac{1}{(k-1)!} f^{(k-1)}(x_i), \tag{1.6}$$

where the subscript i is repeated k times [BeZh65]. We also denote the generalized divided difference (1.6) by f_{i^k}.

The Aitken algorithm for generalized divided differences uses the recursion:

$$f_{0^{a_0}\cdots i^{a_i},k^{a_k}} = \frac{f_{0^{a_0}\cdots i^{a_i},k^{a_k-1}} - f_{0^{a_0}\cdots i^{a_i-1},k^{a_k}}}{x_i - x_k} \tag{1.7}$$

for $0 \le i \le n-1$ and $i+1 \le k \le n$, and the Neville algorithm for generalized divided differences uses the recursion

$$f_{i^{a_i}\cdots k^{a_k}} = \frac{f_{i^{a_i}\cdots k^{a_k-1}} - f_{i^{a_i-1}\cdots k^{a_k}}}{x_i - x_k} \tag{1.8}$$

for $0 \le i \le n-1$ and $i+1 \le k \le n$ where $a_i < m_i$ for $0 \le i \le n$. The recursion formulae (1.7) and (1.8) specialize to (1.2) and (1.3) respectively when $\mathbf{m} = (1, 1, \cdots, 1)$.

1.3. AITKEN ALGORITHM

In this section we give the Aitken algorithm in a Pascal-like notation which will allow us to construct its process dependence graph. To illustrate, let $n = 3$ and $\mathbf{m} = (2, 2, 2, 2)$. The input data, then, consists of the function and derivative values x_i, f_i, f_{ii} for $0 \le i \le 3$.

The following matrices, $A_{ij} \in F^{m_i \times m_j}$, are assumed to contain the generalized divided differences for $0 \le i \le 2$ and $1 \le j \le 3$ in the prescribed fashion:

$$A_{03} = \begin{bmatrix} f_{033} & f_{0033} \\ f_{03} & f_{003} \end{bmatrix} , \quad A_{13} = \begin{bmatrix} f_{00133} & f_{001133} \\ f_{0013} & f_{00113} \end{bmatrix} , \quad A_{23} = \begin{bmatrix} f_{0011233} & f_{00112233} \\ f_{001123} & f_{0011223} \end{bmatrix} ,$$

$$A_{02} = \begin{bmatrix} f_{022} & f_{0022} \\ f_{02} & f_{002} \end{bmatrix} , \quad A_{12} = \begin{bmatrix} f_{00122} & f_{001122} \\ f_{0012} & f_{00112} \end{bmatrix} ,$$

$$A_{01} = \begin{bmatrix} f_{011} & f_{0011} \\ f_{01} & f_{001} \end{bmatrix} .$$

We denote an entry of the above matrices with $A_{ij}(p, q)$ where $1 \le p \le m_i$ and $1 \le q \le m_j$. For notational convenience we assume that the index p increases from left to right, and the index q increases from bottom to top. The entries of these matrices (e.g. the generalized divided differences) are computed with the Aitken algorithm using (1.2) as

$$\begin{array}{lll} A_{03} & A_{13} & A_{23} \\ A_{02} & A_{12} & \\ A_{01} & & \end{array}$$

In most the general case, the input data is given as (x_i, f_{ik}) for $0 \le i \le n$ and $1 \le k \le m_i$. The following procedure given in a Pascal-like notation computes $A_{ij}(p, q)$ for all $0 \le i \le n-1$, $1 \le j \le n$, and $1 \le p \le m_i$, $1 \le q \le m_j$ using recursion (1.7).

Procedure_Aitken

Input : (x_i, f_{ik}) for $0 \le i \le n$, $1 \le k \le m_i$.

Output : $A_{ij}(p, q)$ for $0 \le i \le n-1$, $1 \le j \le n$, $1 \le p \le m_i$, $1 \le q \le m_j$.

BEGIN

FOR $i = 0$ TO $n - 1$ DO

FOR $j = i + 1$ TO n DO

FOR $p = 1$ TO m_i DO

FOR $q = 1$ TO m_j DO

$$A_{ij}(p, q) = \frac{A_{ij}(p, q-1) - A_{ij}(p-1, q)}{x_i - x_j}$$

END PROCEDURE

We note that in the above procedure p and q may take the value 0, corresponding to boundary and interface values of the Aitken table. Whenever this happens, then that value of the matrix entry should be replaced with one of the following:

$$A_{0j}(p, 0) = f_{0^p}, \quad 1 \le p \le m_0,$$
$$A_{ij}(p, 0) = A_{i-1,i}(m_{i-1}, p), \quad 1 \le i \le n-1, \; 2 \le j \le n, \; 1 \le p \le m_i,$$
$$A_{0j}(0, q) = f_{j^q}, \quad 1 \le j \le n, \; 1 \le q \le m_j,$$
$$A_{ij}(0, q) = A_{i-1,j}(m_{i-1}, q), \quad 1 \le i \le n-1, \; 2 \le j \le n, \; 1 \le q \le m_j.$$

The outputs of this procedure are the generalized divided differences which are

$$A_{ij}(p, q) = f_{0^{m_0} 1^{m_1} 2^{m_2} \cdots i^p j^q}$$

for $0 \le i \le n-1$, $1 \le j \le n$, and $1 \le p \le m_i$, $1 \le q \le m_j$. The coefficients of the Hermite interpolating polynomial are the entries of the matrices $A_{i-1,i}$ for $1 \le i \le n$, and expressed in this notation as

$$f_{0^{m_0} 1^q} = A_{01}(m_0, q), \quad 1 \le q \le m_1,$$
$$f_{0^{m_0} 1^{m_1} 2^q} = A_{12}(m_1, q), \quad 1 \le q \le m_2,$$

$$\ldots$$

$$f_{0^{m_0}1^{m_1}2^{m_2}\ldots n^q} = A_{n-1,n}(m_{n-1}, q) \quad , \quad 1 \le q \le m_n \quad .$$

1.4. PROCESS DEPENDENCE GRAPH OF THE AITKEN ALGORITHM

The data dependences among the entries in the divided difference tables computed by the Aitken and Neville algorithms lend themselves to systolic implementation. In order to exploit the properties of these tables we will take a closer look at the algorithms to compute the entries. This will allow us to form the *process dependence graph* of each algorithm to compute the divided differences.

We start with $n = 4$ and $\mathbf{m} = (1, 1, 1, 1, 1)$. In this case the matrices A_{ij} are simply scalars. In order to see the functional dependence of each entry on the previously computed entries and on the node points x_i in the table, we use the recursion (2.3) and fill the Aitken table as follows:

$$A_{04} = \frac{f_0 - f_4}{x_0 - x_4} \quad A_{14} = \frac{A_{01} - A_{04}}{x_1 - x_4} \quad A_{24} = \frac{A_{12} - A_{14}}{x_2 - x_4} \quad A_{34} = \frac{A_{23} - A_{24}}{x_3 - x_4}$$

$$A_{03} = \frac{f_0 - f_3}{x_0 - x_3} \quad A_{13} = \frac{A_{01} - A_{03}}{x_1 - x_3} \quad A_{23} = \frac{A_{12} - A_{13}}{x_2 - x_3}$$

$$A_{02} = \frac{f_0 - f_2}{x_0 - x_2} \quad A_{12} = \frac{A_{01} - A_{02}}{x_1 - x_2}$$

$$A_{01} = \frac{f_0 - f_1}{x_0 - x_1}$$

Here we notice that in the denominator terms, $x_i - x_j$, the node point x_i is repeated along a column whereas the node point x_j is repeated along a row. The positions of the numerator terms, (e.g. the divided difference terms) are similar. First, a divided difference term of the form $A_{i-1,i}$ is computed on the diagonal, then this term is used in every

operation along the *ith* column. Based on these observations we illustrate the process dependence graph of the Aitken algorithm for $n=4$ and $\mathbf{m}=(1,1,\cdots,1)$ in Figure 1.1. The graph is drawn on the (i,j) coordinate system. The nodes of this acyclic directed graph represent the operations, and the oriented edges correspond the dependence among the variables used in the operations. The node at point (i,j) computes A_{ij} by performing the operation

$$A_{ij}=\frac{A_{i-1,i}-A_{i-1,j}}{x_i-x_j} \quad . \tag{1.9}$$

A comparison of Figure 1.1 with the above table reveals the fact that the necessary operands are present for the operations performed in the nodes.

These observations can be used to compute the generalized divided differences. In this case we have $\mathbf{m}=(m_0,m_1,\cdots,m_n)$ with $m_i>1$ and A_{ij} are matrices of dimension $m_i\times m_j$. If we allow the nodes to represent the operations to compute the entries of the matrix A_{ij}, then the process dependence graph for the generalized divided difference tables will be the same as the Newton case which is given in Figure 1.1. Let $m_i=3$ and $m_j=4$, by using Procedure_Aitken we observe that these entries are computed in the following fashion

$$A_{ij}(1,4)=\frac{A_{ij}(1,3)-A_{i-1,j}(m_{i-1},4)}{x_i-x_j} \qquad A_{ij}(2,4)=\frac{A_{ij}(2,3)-A_{ij}(1,4)}{x_i-x_j} \qquad A_{ij}(3,4)=\frac{A_{ij}(3,3)-A_{ij}(2,2)}{x_i-x_j}$$

$$A_{ij}(1,3)=\frac{A_{ij}(1,2)-A_{i-1,j}(m_{i-1},3)}{x_i-x_j} \qquad A_{ij}(2,3)=\frac{A_{ij}(2,2)-A_{ij}(1,3)}{x_i-x_j} \qquad A_{ij}(3,3)=\frac{A_{ij}(3,2)-A_{ij}(2,3)}{x_i-x_j}$$

$$A_{ij}(1,2)=\frac{A_{ij}(1,1)-A_{i-1,j}(m_{i-1},2)}{x_i-x_j} \qquad A_{ij}(2,2)=\frac{A_{ij}(2,1)-A_{ij}(1,2)}{x_i-x_j} \qquad A_{ij}(3,2)=\frac{A_{ij}(3,1)-A_{ij}(2,2)}{x_i-x_j}$$

$$A_{ij}(1,1)=\frac{A_{i-1,j}(m_{i-1},1)-A_{i-1,j}(m_{i-1},1)}{x_i-x_j} \qquad A_{ij}(2,1)=\frac{A_{i-1,j}(m_{i-1},2)-A_{ij}(1,1)}{x_i-x_j} \qquad A_{ij}(3,1)=\frac{A_{i-1,j}(m_{i-1},3)-A_{ij}(2,1)}{x_i-x_j}$$

The process dependence graph of the above operations is illustrated in Figure 1.2. The

node at the point (i, j) now represents $m_i \times m_j$ smaller nodes connected as a rectangular mesh where the entries of the matrix A_{ij} are computed. If $m_i = 1$ for all $0 \le i \le n$, then this graph consists of only one node and the process dependence graph of the generalized Aitken table reduces to the simple case depicted in Figure 1.1. Thus we can view Figure 1.1 as the process dependence graph of the most general case where the nodes are considered as rectangular meshes of smaller nodes. This hierarchical view simplifies the treatment of each case, and allows us embed the process dependence graph in spacetime to produce various systolic arrays. In the following sections, the process dependence graph, G_A, for Aitken's algorithm is embedded in spacetime, obtaining several different systolic arrays (for spacetime embedding techniques, see [CaSt84]).

1.5. THE MCKEOWN ARRAY

In this section, we present McKeown's array [McKe86]. We embed the process dependence graph for Aitken's algorithm in space and time. The abscissa is interpreted as time (t); the ordinate as space (s). The linear embedding, E_1, is as follows:

$$\begin{bmatrix} t \\ s \end{bmatrix} := T_1 \begin{bmatrix} i \\ j \end{bmatrix} \quad , \text{ where } \quad T_1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

The result, depicted in Figure 1.3 for $n = 6$, is McKeown's array. Data that flows south $\rightarrow$ north in Figure 1.1 will flow in the direction of time, (perpendicular to space) in the McKeown design: It is in the processors' memory. Data that flows west $\rightarrow$ east in Figure 1.1 will flow up through the McKeown array. Data that flows south $\rightarrow$ east in Figure 1.1 also will flow up through the McKeown array, but at half the speed of the west $\rightarrow$ east data.

Process (i, j) is executed at time $i + j$ in processor j. By inspection, we see that the array uses n processors, finishing the computation in $2n - 1$ time steps. The number of

vertices (processes) in a longest directed path in any process dependence graph is a lower bound on the number of time steps of any schedule for computing the processes. In our graph, the number of vertices in a longest path is $2n-1$. McKeown's array thus uses a spacetime embedding that is optimal with respect to the number of time steps used. Such an embedding is referred to as *time-optimal.*

1.6. A SPACETIME-OPTIMAL VERSION OF THE MCKEOWN ARRAY

Definition: A graph's embedding is *spacetime-optimal* when it is space-minimal among those embeddings that are time-optimal.

We now make a slight modification to the McKeown array, producing a *spacetime-optimal* array. There is unused time on the lower numbered processors. We reschedule the computation done on the upper processors onto these lower processors. More formally, we embed the process dependence graph as follows:

$$\begin{bmatrix} t \\ s \end{bmatrix} := T_1 \begin{bmatrix} i \\ j \end{bmatrix} \quad , \text{ for } \; i \le n-j \; ;$$

$$\begin{bmatrix} t \\ s \end{bmatrix} := T_1 \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ n-j \end{bmatrix} \quad , \text{ for } \; i > n-j \quad .$$

This embedding, E_2, is illustrated, for $n=6$, in Figure 1.3. This design has 2 phases of data movement. In the 1st phase, data moves as in the McKeown design. As the 1st phase ends, and the 2nd begins, there is a transition: When n is even (as depicted in Figure 1.3), there are 2 time steps in which the south $\rightarrow$ east data will flow in the direction of time: It is in the uppermost processor's memory for these 2 steps. When n is odd, the south $\rightarrow$ east data always will move through the array.

In the 2nd phase, data moves as follows. Data that flows south $\rightarrow$ north in Figure 1.1 will flow down through the array. Data that flows west $\rightarrow$ east in Figure 1.1 will flow in the direction of time: It is remembered in this phase. Data that flows south $\rightarrow$ east in Figure 1.1 also will flow down through this array, but at half the speed of the south $\rightarrow$ north data.

The following theorem asserts that, of those spacetime embeddings that are time-optimal, this embedding also is space-minimal.

Theorem 1.1

Embedding E_2 of G_A is spacetime-optimal.

Proof :

The embedding E_2 is identical to E_1 with respect to time: it too is time-optimal. We argue space minimality as follows. To reduce space, we must reschedule a processor's processes to other processors. Let us focus on the points in time where all processors are used. In Figure 1.3, they occur for $t = 5, 6, 7$. During these points all 3 processors are used. We refer to such points in time as the *space-maximal* points of the embedding. In order to reduce the spatial extent of this embedding, it is necessary that the processes from some processor's space-maximal points be rescheduled onto other processors. We can reduce the spatial extent of the embedding depicted in Figure 1.3, for example, if processes embedded at coordinates are (5,3), (6,3), and (7,3) can be rescheduled onto other processors. Notice that these processes are on a longest directed path in the process dependence graph. This means that none can be rescheduled for earlier completion without violating an order constraint. Neither can they be scheduled for later completion without either violating an order constraint or extending the overall completion time, violating the time-optimality property. In fact, in this embedding every process is on some longest path, and hence can be

rescheduled onto neither an earlier nor a later cycle. For this process dependence graph, processes occurring during the space-maximal points, in particular, cannot be rescheduled. Therefore the number of processors used cannot be reduced without violating time-optimality. We conclude that the embedding is spacetime-optimal: Any spacetime embedding of this process dependence graph that completes in $2n-1$ cycles, must use at least $\lceil n/2 \rceil$ processors. ●

Moreover, the nonlinearity of our spacetime transformation is necessary: There does not exist a linear embedding of the initial indices that is spacetime-optimal.

1.7. SOME OTHER SPACETIME-OPTIMAL ARRAYS

We now present 2 other spacetime-optimal embeddings of the process dependence graph of Figure 1.1. The first is another variation of McKeown's array. We again reschedule the computation done on the upper processors onto the lower processors. To do this, we connect the endpoints of the linear array, making a *ring* of processors. More formally, we nonlinearly embed the process dependence graph as follows:

$$t := i + j \;;$$
$$s := j \bmod \lfloor n/2 \rfloor \;;$$

This embedding, E_3, is illustrated, for $n = 6$, in Figure 1.5. This design has data flow characteristics that are identical to the McKeown array, except that the upper processor is attached to the lower processor, and data movement wraps around.

Since this embedding results in a computation of the process dependence graph that uses $2n-1$ steps and $\lceil n/2 \rceil$ processors, it too is spacetime-optimal.

Finally, we present a bilateral array in which the south $\rightarrow$ north data of Figure 1.1

moves up through the array, while the west $\rightarrow$ data moves down through the array. Such a data movement scheme may be useful, depending on the larger context of which this computation is a part. The spacetime embedding, E_4, is presented in 2 steps:

1. First, we embed the process dependence graph, illustrated in Figure 1.6, as follows:

$$\begin{bmatrix} t \\ s \end{bmatrix} := T_2 \begin{bmatrix} i \\ j-1 \end{bmatrix} \quad , \text{ where } \quad T_2 = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$$

   In this spacetime embedding, when processors are used, they are used only every other time step.

2. We now compress the spatial extent of this embedding with the following nonlinear transformation:

$$T_2 = \lfloor C \rfloor \quad ,$$

   where

$$C = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \quad , \text{ and } y = \lfloor C \rfloor x = \lfloor C x \rfloor \quad , \text{ where } \left\lfloor \begin{bmatrix} i \\ j \end{bmatrix} \right\rfloor = \begin{bmatrix} \lfloor i \rfloor \\ \lfloor j \rfloor \end{bmatrix}$$

   Processor efficiency (i.e., the percentage of time that a processor is used) is doubled asymptotically by this nonlinear transformation. Figure 1.7 illustrates the result.

This design has 2 phases of data movement which alternate with each time step. Regardless of the phase, data that flows south $\rightarrow$ east in Figure 1.1 will flow in the direction of time: It is remembered.

In phase A, data that flows south $\rightarrow$ north in Figure 1.1 will flow up through the array; data that flows west $\rightarrow$ east in Figure 1.1 will in the direction of time.

In phase B, data that flows south $\rightarrow$ north in Figure 1.1 will flow in the direction of time; data that flows west $\rightarrow$ east in Figure 1.1 will flow up through the array.

Since this second transformation results in an embedding that uses $2n-1$ steps and $\lceil n/2 \rceil$ processors, it too is spacetime-optimal.

1.8 TWO-DIMENSIONAL ARRAY

We now present a 2–d array for computing the process dependence graph of Figure 1.1. This is done by embedding the process dependence graph into a 3–d space. One way to do this with a linear embedding, E_5, is as follows:

$$\begin{bmatrix} t \\ s_1 \\ s_2 \end{bmatrix} := T_3 \begin{bmatrix} i \\ j \end{bmatrix} \quad , \text{ where } \quad T_3 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figure 1.8 illustrates the result. In this array, there is a processor for every process (in the process dependence graph). The flow of data between processors corresponds to the arcs in the process dependence graph. Every processor whose corresponding vertex in Figure 1.1 has indices whose sum is k will execute its process at time step k. Execution completes after $2n-1$ time steps. This embedding has the property that each processor is used exactly once per execution of the process dependence graph. The array can start executing a new process dependence graph every time step. Figure 1.9 is intended to illustrate the pipeline quality of this array; it shows 2 process dependence graphs embedded in spacetime such that execution of the 2nd starts 1 time step after the 1st. Consequently, executing k such process dependence graphs uses $2n+k-2$ time steps.

The 2-d systolic array is useful when one faces the problem of computing many interpolation polynomials with possibly differents set of points. Such is the case for multivariate interpolation which we consider in Section 1.11.

1.9. COMPUTING GENERALIZED DIVIDED DIFFERENCES

In this section, we incorporate extensions to Aitken's algorithm, enabling it to be used to compute generalized divided differences. As mentioned in Section 1.4, the process dependence graph for the generalized divided differences has a two-level structure. At the top level, its structure is that of the divided differences process dependence graph, illustrated in Figure 1.1. Each node in this top level structure comprises a rectangular mesh of nodes that perform scalar operations, illustrated in Figure 1.2. Figure 1.10 illustrates how these rectangular meshes are interconnected to form the 2-level process dependence graph G_{GA} . The spacetime embeddings used for computing the divided differences (see Sections 1.5-1.8) can be applied to the process dependence graph for computing generalized divided differences. The McKeown embedding for the generalized case is illustrated in Figure 1.11. Table 1.1 contains the time and space complexities of these embeddings. Embeddings E_2 , E_3 , and E_4 also are spacetime-optimal for the process dependence graph, G_{GA} , that computes generalized divided differences.

There are several other ways to embed this 2-level process dependence graph. Some of these design alternatives are touched upon in the Conclusion.

1.10. NEVILLE ALGORITHM AND PROCESS DATA DEPENDENCE GRAPH

In this section we describe the Neville algorithm and its process dependence graph. For simplicity of the illustration, we assume that $n = 3$ and $\mathbf{m} = (2, 2, 2, 2)$. Similar to the Aitken algorithm, we define the matrices $N_{ij} \in F^{m_i \times m_j}$ for $0 \le i \le 2$ and $1 \le j \le 3$ as:

$$N_{03} = \begin{bmatrix} f_{00112233} & f_{0112233} \\ f_{0011223} & f_{011223} \end{bmatrix} , \quad N_{13} = \begin{bmatrix} f_{112233} & f_{12233} \\ f_{11223} & f_{1223} \end{bmatrix} , \quad N_{23} = \begin{bmatrix} f_{2233} & f_{233} \\ f_{223} & f_{23} \end{bmatrix} ,$$

$$N_{02}=\begin{bmatrix} f_{001122} & f_{01122} \\ f_{00122} & f_{0112} \end{bmatrix} \quad , \quad N_{12}=\begin{bmatrix} f_{1122} & f_{122} \\ f_{112} & f_{12} \end{bmatrix} \quad ,$$

$$N_{01}=\begin{bmatrix} f_{0011} & f_{011} \\ f_{001} & f_{01} \end{bmatrix} \quad .$$

In general we have

$$N_{ij}(p\ ,q\ )=f_{i^p\,(i+1)^{m_{i+1}}\cdots\,(j-1)^{m_{j-1}}j^q}$$

where index p is assumed to increase from right to left and index q from bottom to top. The entries of the matrices N_{ij} are computed by the Neville algorithm using recursion (1.10).

$$\begin{array}{lll} N_{03} & N_{13} & N_{23} \\ N_{02} & N_{12} & \\ N_{01} & & \end{array}$$

Here, the diagonal entries, N_{01}, N_{12}, and N_{23}, are computed first, followed by the entries above the main diagonal, N_{02}, and N_{13}, and so on. This process is formalized in the following procedure.

Procedure_Neville

Input : $(x_i\ ,f_{i^k})$ for $0\le i\le n\ ,\ 1\le k\le m_i$.

Output : $N_{ij}(p\ ,q\ )$ for $0\le i\le n-1\ ,\ 1\le j\le n\ ,\ 1\le p\le m_i\ ,\ 1\le q\le m_j$.

BEGIN

FOR $j=1$ TO n DO

FOR $i = 0$ TO $n - j$ DO

FOR $p = 1$ TO m_i DO

FOR $q = 1$ TO m_j DO

$$N_{i,i+j}(p,q) = \frac{N_{i,i+j}(p,q-1) - N_{i,i+j}(p-1,q)}{x_i - x_{i+j}}$$

END PROCEDURE

The boundary and interface conditions for the Neville_Procedure are found to be

$$N_{i,i+1}(p,0) = f_{ip} \ , \ 0 \le i \le n-1 \ , \ 1 \le p \le m_0 \ ,$$
$$N_{i,i+j}(p,0) = N_{i,i+j-1}(p, m_{i+j-1}) \ , \ 2 \le j \le n \ , \ 0 \le i \le n-j \ , \ 1 \le p \le m_i \ ,$$
$$N_{i,i+1}(0,q) = f_{(i+1)q} \ , \ 0 \le i \le n-1 \ , \ 1 \le q \le m_{i+1} \ ,$$
$$N_{i,i+j}(0,q) = N_{i,i+j+1}(m_{i+j+1}, q) \ , \ 2 \le j \le n \ , \ 0 \le i \le n-j \ , \ 1 \le q \le m_{i+j+1} \ .$$

To illustrate we take $n = 4$ and $\mathbf{m} = (1,1,1,1,1)$.

$$N_{04} = \frac{N_{03} - N_{14}}{x_0 - x_4} \quad N_{14} = \frac{N_{13} - N_{24}}{x_1 - x_4} \quad N_{24} = \frac{N_{23} - N_{34}}{x_2 - x_4} \quad N_{34} = \frac{f_3 - f_4}{x_3 - x_4}$$

$$N_{03} = \frac{N_{02} - N_{13}}{x_0 - x_3} \quad N_{13} = \frac{N_{12} - N_{23}}{x_1 - x_3} \quad N_{23} = \frac{f_2 - f_3}{x_2 - x_3}$$

$$N_{02} = \frac{N_{01} - N_{12}}{x_0 - x_2} \quad N_{12} = \frac{f_1 - f_2}{x_1 - x_2}$$

$$N_{01} = \frac{f_0 - f_1}{x_0 - x_1}$$

For the denominator terms $x_i - x_j$, the point x_i is repeated along a column whereas the point x_j is repeated along a row. Also when a divided difference term, N_{ij}, is computed, then this term is used to compute $N_{i-1,j}$ and $N_{i,j+1}$. The process dependence graph

of the Neville algorithm, G_N, is given in Figure 1.12 drawn on the (i, j) coordinate system where the node at the point (i, j) computes N_{ij} by performing the operation

$$N_{ij} = \frac{N_{i,j-1} - N_{i+1,j}}{x_i - x_j} \quad . \tag{1.10}$$

For the generalized divided differences, similar to the Aitken case, we have $\mathbf{m} = (m_0, m_1, \cdots, m_n)$ with $m_i > 1$ and N_{ij} are matrices of dimension $m_i \times m_j$. Now we let the nodes in Figure 1.12 represent the set operations to compute the entries of the matrix N_{ij}. Figure 1.12, then, represents the process dependence graph for the generalized divided difference table as well. If $m_i = 3$ and $m_j = 4$, then by using Procedure_Neville we compute the entries of N_{ij} as

$$N_{ij}(3,4) = \frac{N_{ij}(3,3) - N_{ij}(2,4)}{x_i - x_j} \qquad N_{ij}(2,4) = \frac{N_{ij}(2,3) - N_{ij}(1,4)}{x_i - x_j} \qquad N_{ij}(1,4) = \frac{N_{ij}(1,3) - N_{i+1,j}(m_{i+1},4)}{x_i - x_j}$$

$$N_{ij}(3,3) = \frac{N_{ij}(3,2) - N_{ij}(2,3)}{x_i - x_j} \qquad N_{ij}(2,3) = \frac{N_{ij}(2,2) - N_{ij}(1,3)}{x_i - x_j} \qquad N_{ij}(1,3) = \frac{N_{ij}(1,2) - N_{i+1,j}(m_{i+1},3)}{x_i - x_j}$$

$$N_{ij}(3,2) = \frac{N_{ij}(3,1) - N_{ij}(2,2)}{x_i - x_j} \qquad N_{ij}(2,2) = \frac{N_{ij}(2,1) - N_{ij}(1,2)}{x_i - x_j} \qquad N_{ij}(1,2) = \frac{N_{ij}(1,1) - N_{i+1,j}(m_{i+1},2)}{x_i - x_j}$$

$$N_{ij}(3,1) = \frac{N_{i,j-1}(3,m_{j-1}) - N_{ij}(2,1)}{x_i - x_j} \qquad N_{ij}(2,1) = \frac{N_{i,j-1}(2,m_{j-1}) - N_{ij}(1,1)}{x_i - x_j} \qquad N_{ij}(1,1) = \frac{N_{i,j-1}(1,m_{j-1}) - N_{i+1,j}(m_{i+1},1)}{x_i - x_j}$$

The process dependence graph of the above operations is illustrated in Figure 1.13. Similar to the Aitken algorithm the node at the point (i, j) represents $m_i \times m_j$ smaller nodes connected as a rectangular mesh.

Process dependence graph of the Neville algorithm, G_N, can be embedded in spacetime similar to the embeddings of the Aitken algorithm as we have described in Sections 1.5-1.8. Here we describe one such embedding which is a variation of McKeown's array. The spacetime embedding of the process dependence graph given in Figure 1.12 is as follows

$$\begin{bmatrix} t \\ s \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

This embedding illustrated in Figure 1.14 for $n = 6$. Data that flows south $\rightarrow$ north in Figure 1.12 will flow in the direction of time and space. It flows up through the array. Data that flows east $\rightarrow$ west in Figure 1.12 will flow in the direction of time: it stays in the processors' memory. We see that the array uses n processors, finishing the computation in n time.

1.11. COMPUTATION OF MULTIVARIATE DIVIDED DIFFERENCES

In this section we consider the problem of multivariate interpolation. We illustrate the underlying algorithms and systolic arrays for computation of the bivariate divided differences. Extension to more than two variables is straightforward.

The general problem of multivariate interpolation of an arbitrary set of points is difficult, and the algorithms are complicated. The main reason for this is that *unisolvence* can not be satisfied for completely arbitrary spaced data [BeZh65]. In one dimension, the distribution of the points is restricted enough to satisfy unisolvence [Thac60]. One usually needs to make assumptions regarding the distribution of the points; see, for example, [Salz56], [GuRo70], [GaMa82]. Otherwise it is required that certain determinants which appear in constructing the interpolating polynomials do not vanish [Salz64]. We consider bivariate interpolation on a two-dimensional grid only. This is a common case which also offers a natural generalization of the divided difference algorithms to higher dimensions. The machinery developed so far for computation of the univariate divided differences suits well to this case. We assume that there exists a bivariate function $f(x, y)$ evaluated at $(n+1)(n+1)$ points on the Euclidean plane (x_i, y_j) for $0 \le i, j \le n$. Denote $f(x_i, x_j)$ with $f_{i,j}$; then the points are arranged as

$$f_{0,n} \quad f_{1,n} \quad f_{2,n} \quad \cdot \quad f_{n,n}$$

$$\begin{array}{ccccc} . & . & . & . & . \\ f_{0,2} & f_{1,2} & f_{2,2} & . & f_{n,2} \\ f_{0,1} & f_{1,1} & f_{2,1} & . & f_{n,1} \\ f_{0,0} & f_{1,0} & f_{2,0} & . & f_{n,0} \end{array}$$

The bivariate Newton polynomial interpolating this data set can be given as

$$P(x,y) = \sum_{i=0}^{n} \sum_{j=0}^{n} (x-x_0)..(x-x_{i-1})\,(y-y_0)..(y-y_{j-1}) f_{01..i,01..j}$$

where the coefficients $f_{01..i,01..j}$ are bivariate divided differences which are found by applying the univariate divided difference algorithms repeatedly in the x and y directions. First we simultaneously apply the Aitken (or the Neville) Algorithm in the x direction for all $0 \le j \le n$, and find

$$\begin{array}{ccccc} f_{0,n} & f_{01,n} & f_{012,n} & . & f_{012..n,n} \\ . & . & . & . & . \\ f_{0,2} & f_{01,2} & f_{012,2} & . & f_{012..n,2} \\ f_{0,1} & f_{01,1} & f_{012,1} & . & f_{012..n,1} \\ f_{0,0} & f_{01,0} & f_{012,0} & . & f_{012..n,0} \end{array}$$

The application of the Aitken (or the Neville) Algorithms in y direction for all $0 \le i \le n$ yields the bivariate divided differences.

$$\begin{array}{ccccc} f_{0,012..n} & f_{01,012..n} & f_{012,012..n} & . & f_{012..n,012..n} \\ . & . & . & . & . \\ f_{0,012} & f_{01,012} & f_{012,012} & . & f_{012..n,012} \\ f_{0,01} & f_{01,01} & f_{012,01} & . & f_{012..n,01} \\ f_{0,0} & f_{01,0} & f_{012,0} & . & f_{012..n,0} \end{array}$$

The 2-d arrays developed in Section 1.8 can be used to perform the above computations efficiently. Assume that initially $B_{ij} = f_{i,j}$ for $0 \le i, j \le n$. The following procedure computes the bivariate divided differences assuming a 2-d Aitken array is available to a host computer.

Procedure Bivariate $(x_i, y_j, B_{ij} \; ; \; 0 \le i, j \le n)$

FOR $j = 0$ TO n

$B_{ij} =$ **Univariate** $(x_i, B_{ij} \; ; \; 0 \le i \le n)$

END FOR

FOR $i = 0$ TO n

$B_{ij} =$ **Univariate** $(y_j, B_{ij} \; ; \; 0 \le j \le n)$

END FOR

END PROCEDURE

At the end of all computations the arrays B_{ij} holds the bivariate divided differences, i.e. $B_{ij} = f_{01..i,01..j}$ for $0 \le i, j \le n$.

The execution of k processes on a 2-d Aitken array takes $2n + k - 2$ time steps where each step is a divided difference operation (1.9). Thus, the above procedure will take $3n - 1$ steps to execute the first loop, and the same number of steps for the second loop; which gives the total parallel time as $T_{par} = 6n - 2$. Since the sequential computation of the bivariate divided differences takes $T_{seq} = n(n-1)(n+1)$, and 2-d array contains $p = n(n-1)/2$ processing elements, the efficiency of this scheme will be

$$E = \frac{T_{seq}}{p\ T_{par}} = \frac{n+1}{3\,n-1} \approx 0.33$$

which is asymptotically optimal. Computation of bivariate (or *multivariate*) generalized divided differences can be done by straightforward extension of the above algorithm.

1.12. LARGE INTERPOLATION ON SMALLER ARRAYS

One of the well-known constraints in the VLSI implementation of algorithms is what we will call the *dimensionality problem.* Processor arrays come with fixed dimensions and we must try to use them effectively. A parallel can be drawn with the situation in strongly typed languages like Pascal, where procedures and functions handling array type arguments cannot handle arrays of different dimensions, even if the dimension bounds are not exceeded. The software version of the problem is solved by making suitable improvements to the language-compiler capabilities. It is obvious that none of these methods can handle the hardware version of the problem, which will require a direct modification of the algorithm. As the dimension of the problem will only rarely be fixed beforehand, one of the following may happen:

1) The dimension of the problem fits exactly the size of the processor array; In this case there should be no difficulty in using the processor array effectively.

2) The dimension of the problem is smaller than the size of the processor array; In this case, much depends on the structure of the algorithm. In most cases the situation is easily handled, by disabling some units (assuming this is possible). Hence a less efficient use of the array will result. There are situations however in which this is not possible. An example is the case of a processor array used in a recirculating fashion or making use of wrap-around connections, e.g. like the use of a processor array to handle the multiplication of two matrices by means of Cannon's algorithm [Cann69],

[DeNS81]. Again, the situation is handled by suitably increasing the size of the problem with data elements which do not affect the result (e.g. identity elements with respect to the size of the problem.) In any case, the array is used with less than full efficiency.

3) The dimension of the problem is larger than the size of the processor array; This is the most interesting situation and we have to consider it in some detail. The difficulty under which this will be handled is directly proportional to the amount of decomposability inherent in the algorithm. At best, the algorithm would be decomposed into blocks of size equal to the dimension of the processor array. Subsequently, partial results would be composed together using a processor array of the same size (it could be the same array for example).

In this section we show that by formulating the Newton and Hermite polynomial interpolation problems as solution of linear systems of equations, it is possible to decompose a polynomial interpolation problem to polynomial interpolation and matrix-vector product problems of smaller size. The idea is based on the observation that a polynomial interpolation problem can be formulated as solution of a linear system of equations of the form

$$\mathbf{N}\,\mathbf{c} = \mathbf{f} \tag{1.11}$$

where $\mathbf{N} \in F^{(n+1)\times(n+1)}$ and $\mathbf{c}, \mathbf{f} \in F^{n+1}$. If the standard representation of a polynomial is used then the matrix $\mathbf{N}$ is transpose of a *Vandermonde* matrix of dimension $n+1$. The polynomial interpolation problem for the standard representation of the polynomial can thus be solved by solving a Vandermonde system of linear equations [BjPe70].

Traditionally the Newton and Hermite interpolation problems are solved by using the Aitken and Neville recursions rather than introducing the problem as a linear algebra prob-

lem. This is based on the fact that the recursions are easy to program, and the memory requirements are quite modest. It takes only $O(n^2)$ arithmetic operations and $O(n)$ memory space to compute the coefficients of a n-degree Newton interpolating polynomial. We will show that a Newton polynomial interpolation problem of size $n+1$ can be decomposed into $q+1$ polynomial interpolation and matrix-vector product problems of size $p+1$ each if $n+1=(p+1)(q+1)$ if it is formulated as a linear algebra problem of the form (1.11).

This formulation and decomposition is naturally generalized to the Hermite interpolation. We illustrate the decomposition for the case of the Newton interpolation. Here, $\mathbf{c}$ and $\mathbf{f}$ are vectors containing the divided differences, and the function values, respectively. That is $\mathbf{f}=[f_0, f_1, \cdots, f_n]^T$ and $\mathbf{c}=[c_0, c_1, \cdots, c_n]^T$ where $c_i = f_{012..i}$ for $0 \le i \le n$. The matrix $\mathbf{N}$ is a lower-triangular matrix containing the prefix products of $x_i - x_j$ for $0 \le i \ne j \le n$, and it is conveniently named as the *Newton-Vandermonde* matrix. This can be illustrated as follows: if we write the interpolating polynomial in the Newton form

$$p_n(x) = c_0 + c_1(x-x_0) + c_2(x-x_0)(x-x_1) + \cdots$$
$$c_n(x-x_0)(x-x_1)..(x-x_{n-1}) \quad .$$

then the coefficients of this polynomial can be found by solving

$$p_n(x_i) = f_i \quad , \quad 0 \le i \le n \tag{1.12}$$

which yields the following set of linear equations:

$$c_0 = f_0$$
$$c_0 + (x_1 - x_0)c_1 = f_1$$
$$c_0 + (x_2 - x_0)c_1 + (x_2 - x_0)(x_2 - x_1)c_2 = f_2$$
$$\cdots$$

$$c_0+(x_n-x_0)\,c_1+(x_n-x_0)(x_n-x_1)\,c_2+\cdots+(x_n-x_0)\,..\,(x_n-x_{n-1})\,c_n=f_n$$

For simplicity of the notation we make the following definition

$$\pi_{ij}^{k}=\prod_{r=j}^{k}(x_i-x_r) \tag{1.13}$$

for $1\le i\le n$ and $0\le j<k<i$. The above linear system of equations, then, can be written in a matrix form as $\mathbf{N}\,\mathbf{c}=\mathbf{f}$, or more explicitly

$$\begin{bmatrix} 1 & 0 & 0 & . & . & 0 \\ \pi_{10}^{0} & 1 & 0 & . & . & 0 \\ \pi_{20}^{0} & \pi_{20}^{1} & 1 & . & . & 0 \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ \pi_{n0}^{0} & \pi_{n0}^{1} & \pi_{n0}^{2} & . & . & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ . \\ . \\ c_n \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ . \\ . \\ f_n \end{bmatrix} \tag{1.14}$$

Assuming $n+1=(p+1)(q+1)$, we partition the matrix $\mathbf{N}$ into the matrices $\mathbf{N}_{ij}\in F^{(p+1)\times(p+1)}$ for $0\le i,j\le q$, and also the vectors $\mathbf{c}$ and $\mathbf{f}$ to the vectors $\mathbf{c}_i$, $\mathbf{f}_i\in F^{p+1}$ for $0\le i\le q$. Thus we obtain

$$\begin{bmatrix} \mathbf{N}_{00} & \mathbf{0} & \mathbf{0} & . & . & \mathbf{0} \\ \mathbf{N}_{10} & \mathbf{N}_{11} & \mathbf{0} & . & . & \mathbf{0} \\ \mathbf{N}_{20} & \mathbf{N}_{21} & \mathbf{N}_{22} & . & . & \mathbf{0} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ \mathbf{N}_{q0} & \mathbf{N}_{q1} & \mathbf{N}_{q2} & . & . & \mathbf{N}_{qq} \end{bmatrix} \begin{bmatrix} \mathbf{c}_0 \\ \mathbf{c}_1 \\ \mathbf{c}_2 \\ . \\ . \\ \mathbf{c}_q \end{bmatrix} = \begin{bmatrix} \mathbf{f}_0 \\ \mathbf{f}_1 \\ \mathbf{f}_2 \\ . \\ . \\ \mathbf{f}_q \end{bmatrix} \tag{1.15}$$

where

$$\mathbf{N}_{ij} = \begin{bmatrix} \pi_{i(p+1),0}^{j(p+1)} & \pi_{i(p+1),0}^{j(p+1)+1} & \cdot & \pi_{i(p+1),0}^{j(p+1)+p} \\ \pi_{i(p+1)+1,0}^{j(p+1)} & \pi_{i(p+1)+1,0}^{j(p+1)+1} & \cdot & \pi_{i(p+1)+1,0}^{j(p+1)+p} \\ \cdot & \cdot & \cdot & \cdot \\ \pi_{i(p+1)+p,0}^{j(p+1)} & \pi_{i(p+1)+p,0}^{j(p+1)+1} & \cdot & \pi_{i(p+1)+p,0}^{j(p+1)+p} \end{bmatrix}, \tag{1.16}$$

and

$$\mathbf{c}_i = \begin{bmatrix} c_{i(p+1)} \\ c_{i(p+1)+1} \\ \cdot \\ c_{i(p+1)+p} \end{bmatrix}, \quad \mathbf{f}_i = \begin{bmatrix} f_{i(p+1)} \\ f_{i(p+1)+1} \\ \cdot \\ f_{i(p+1)+p} \end{bmatrix}. \tag{1.17}$$

Note that $\mathbf{N}_{ii} \in F^{(p+1)\times(p+1)}$ are lower-triangular matrices with 1's on the diagonal. and the zero matrices are the same dimension as $\mathbf{N}_{ij}$ matrices. The system (1.15) can be solved via forward substitution, i.e. first solving for $\mathbf{c}_0$, and then performing successive substitutions to find all $\mathbf{c}_i$ for $1 \le i \le q$.

Procedure Partition_and_Solve ($\mathbf{N}\,\mathbf{c} = \mathbf{f}$)

BEGIN

$\mathbf{c}_0 = \mathbf{N}_{00}^{-1}\,\mathbf{f}_0$

FOR $i = 1$ TO q DO

BEGIN

$\mathbf{e}_i = \mathbf{f}_i$

FOR $j = 0$ TO $i - 1$ DO

$\mathbf{e}_i = \mathbf{e}_i + \mathbf{N}_{ij}\,\mathbf{c}_j$

$\mathbf{c}_i = \mathbf{N}_{ii}^{-1}\,\mathbf{e}_i$

END

END PROCEDURE

The most important point in this algorithm is that it is not actually required to solve an arbitrary linear system. A linear system involving $\mathbf{N}_{ii}$ is equivalent to the Newton interpolation for some pairs of points. To explain this we start with the solution of the system $\mathbf{N}_{00}\,\mathbf{c}_0 = \mathbf{e}_0$. Here we observe that the solution of the system of equations

$$\begin{bmatrix} 1 & 0 & 0 & . & . & 0 \\ \pi_{10}^{0} & 1 & 0 & . & . & 0 \\ \pi_{20}^{0} & \pi_{20}^{1} & 1 & . & . & 0 \\ . & . & . & . & . & \\ . & . & . & . & . & \\ \pi_{p0}^{0} & \pi_{p0}^{1} & \pi_{p0}^{2} & . & . & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ . \\ . \\ c_p \end{bmatrix} = \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ . \\ . \\ e_p \end{bmatrix} \tag{1.18}$$

is equivalent to performing Newton interpolation with the pairs of the points $(x_j\,,e_j)$ to compute the divided differences $c_j = f_{012..j}$ for $0 \le j \le p$. Thus we have

$$\mathbf{c}_0 = \mathbf{N}_{00}^{-1}\,\mathbf{e}_0 \quad \leftrightarrow \quad \mathbf{c}_0 = \mathbf{Newton\_Interpolation}\,(x_j\,,e_j\;;\;0 \le j \le p)\;.$$

For solution of the system $\mathbf{N}_{ii}\,\mathbf{c}_i = \mathbf{e}_i$ for $1 \le i \le q$ first we observe that since

$$\mathbf{N}_{ii} = \begin{bmatrix} 1 & 0 & 0 & . & . & 0 \\ \pi_{i(p+1)+1,0}^{i(p+1)} & 1 & 0 & . & . & 0 \\ \pi_{i(p+1)+2,0}^{i(p+1)} & \pi_{i(p+1)+2,0}^{i(p+1)+1} & 1 & . & . & 0 \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ \pi_{i(p+1)+p,0}^{i(p+1)} & \pi_{i(p+1)+p,0}^{i(p+1)+1} & \pi_{i(p+1)+p,0}^{i(p+1)+2} & . & . & 1 \end{bmatrix}. \tag{1.19}$$

an element of $\mathbf{N}_{ii}$ can be written as

$$\pi^{i(p+1)+s}_{i(p+1)+r,0} = \prod_{j=0}^{i(p+1)+s} (x_{i(p+1)+r} - x_j) = \prod_{j=0}^{i(p+1)-1} (x_{i(p+1)+r} - x_j) \times \prod_{j=i(p+1)}^{i(p+1)+s} (x_{i(p+1)+r} - x_j)$$

for $0 \le r, s \le p$. Using definition (1.13) we have

$$\pi^{i(p+1)+s}_{i(p+1)+r,0} = \pi^{i(p+1)-1}_{i(p+1)+r,0} \times \pi^{i(p+1)+s}_{i(p+1)+r,i(p+1)} .$$

from definition (1.13). Thus N_{ii} can be written as multiplication of a diagonal and a triangular matrix as

$$\mathbf{N}_{ii} = \begin{bmatrix} 1 & 0 & . & . & 0 \\ \pi^{i(p+1)-1}_{i(p+1)+1,0} \times \pi^{i(p+1)}_{i(p+1)+1,i(p+1)} & 1 & . & . & 0 \\ \pi^{i(p+1)-1}_{i(p+1)+2,0} \times \pi^{i(p+1)}_{i(p+1)+2,i(p+1)} & \pi^{i(p+1)-1}_{i(p+1)+2,0} \times \pi^{i(p+1)+1}_{i(p+1)+2,i(p+1)} & . & . & 0 \\ . & . & . & . & . \\ . & . & . & . & . \\ \pi^{i(p+1)-1}_{i(p+1)+p,0} \times \pi^{i(p+1)}_{i(p+1)+p,i(p+1)} & \pi^{i(p+1)-1}_{i(p+1)+p,0} \times \pi^{i(p+1)+1}_{i(p+1)+p,i(p+1)} & . & . & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & & & & \\ & \pi^{i(p+1)-1}_{i(p+1)+1,0} & & & \\ & & \pi^{i(p+1)-1}_{i(p+1)+2,0} & & \\ & & & . & \\ & & & & \pi^{i(p+1)-1}_{i(p+1)+p,0} \end{bmatrix} \begin{bmatrix} 1 & 0 & . & . & 0 \\ \pi^{i(p+1)}_{i(p+1)+1,i(p+1)} & 1 & . & . & 0 \\ \pi^{i(p+1)}_{i(p+1)+2,i(p+1)} & \pi^{i(p+1)+1}_{i(p+1)+2,i(p+1)} & . & . & 0 \\ . & . & . & . & . \\ . & . & . & . & . \\ \pi^{i(p+1)}_{i(p+1)+p,i(p+1)} & \pi^{i(p+1)+1}_{i(p+1)+p,i(p+1)} & . & . & 1 \end{bmatrix} \Leftarrow$$

We write this equation as

$$\mathbf{N}_{ii} = \mathbf{D}_i \ \mathbf{M}_{ii} \tag{1.20}$$

An inspection of the matrix $\mathbf{M}_{ii}$ reveals that solution of $\mathbf{M}_{ii}\ \mathbf{c}_i = \mathbf{d}_i$ is equivalent to Newton interpolation at the pairs of points (x_j, d_j) for $i(p+1) \le j \le i(p+1)+p$. Thus we have

$$\mathbf{c}_i = \mathbf{N}_{ii}^{-1}\ \mathbf{e}_i = \mathbf{M}_{ii}^{-1}\ \mathbf{D}_i^{-1}\ \mathbf{e}_i = \mathbf{M}_{ii}^{-1}\ \mathbf{d}_i$$

$$\mathbf{c}_i = \mathbf{M}_{ii}^{-1}\ \mathbf{d}_i \quad \leftrightarrow \quad \mathbf{c}_i = \mathbf{Newton\_Interpolation}\ (x_j, d_j\ ;\ i(p+1) \le j \le (p+1)+p) .$$

The next point which needs clarification is the construction of the matrices $\mathbf{N}_{ij}$ and $\mathbf{D}_i$ for $0 \le j < i \le q$ and $1 \le i \le q$, respectively. Regarding the former we note that

$$\mathbf{N}_{ij}(r, s) = \pi^{j(p+1)+s}_{i(p+1)+r,0}$$

for $0 \le r, s \le p$. Thus the first column of $\mathbf{N}_{ij}$ is written as

$$\begin{aligned}\mathbf{N}_{ij}(r, 0) = \pi^{j(p+1)}_{i(p+1)+r,0} &= \prod_{k=0}^{j(p+1)} (x_{i(p+1)+r} - x_k) \\ &= \prod_{k=0}^{(j-1)(p+1)+p} (x_{i(p+1)+r} - x_k) \times (x_{i(p+1)+r} - x_{j(p+1)}) \\ &= \mathbf{N}_{i,j-1}(r, p) \times (x_{i(p+1)+r} - x_{j(p+1)}) \end{aligned} \tag{1.21}$$

For $1 \le s \le p$ we have

$$\begin{aligned}\mathbf{N}_{ij}(r, s) &= \prod_{k=0}^{j(p+1)+s} (x_{i(p+1)+r} - x_k) \\ &= \prod_{k=0}^{j(p+1)+s-1} (x_{i(p+1)+r} - x_k) \times (x_{i(p+1)+r} - x_{j(p+1)+s}) \\ &= \mathbf{N}_{ij}(r, s-1) \times (x_{i(p+1)+r} - x_{j(p+1)+s}) \end{aligned} \tag{1.22}$$

Using (1.21) and (1.22) we can obtain the process dependence graph for the computation of the entries of $\mathbf{N}_{ij}$. The resulting graph is a $(p+1) \times (p+1)$ size mesh as depicted in Figure 1.15

To compute the entries of the diagonal matrix $\mathbf{D}_i$ for $1 \le i \le q$ first we note that $\mathbf{D}_i(0, 0) = 1$, and for $1 \le r \le p$ we write

$$\mathbf{D}_i(r, r) = \prod_{k=0}^{i(p+1)-1} (x_{i(p+1)+r} - x_k)$$

$$= \prod_{k=0}^{(p+1)-1} (x_{i(p+1)+r} - x_k) \times \prod_{k=(p+1)}^{2(p+1)-1} (x_{i(p+1)+r} - x_k) \times \cdots \times \prod_{k=(i-1)(p+1)-1}^{i(p+1)-1} (x_{i(p+1)+r} - x_k).$$

Process dependence graph of one instance of this product consists of a 2-d mesh of size

$p \times p$ similar to the one in Figure 1.15. We depict the process dependence graph for computation of

$$\prod_{k=j(p+1)}^{(j+1)(p+1)-1} (x_{i(p+1)+r} - x_k)$$

in Figure 1.16.

Hence we see that the partitioning of the Newton-Vandermonde matrix leads three types of computations

i) polynomial interpolation of size p

ii) construction of matrices $\mathbf{N}_{ij}$, which has a $(p+1)\times(p+1)$ mesh as its process dependence graph.

iii) construction of diagonal matrices $\mathbf{D}_i$, which has a $p \times p$ mesh as its process dependence graph.

iv) inner-product operation $\mathbf{N}_{ij}\ \mathbf{c}_j$ which also has a $(p+1)\times(p+1)$ mesh as its process dependence

The partitioning, thus, yields process dependence graphs which may be embedded in spacetime to produce linear arrays of size $p+1$

1.13. DISCUSSION AND CONCLUSION

The Aitken and the Neville algorithms explained in Section 1.2 compute the coefficients of the interpolating polynomials. Their recursions can be modified to evaluate the interpolating polynomial at a point $\bar{x}$ without actually computing its coefficients. This implicit evaluation of the interpolating polynomial is named as *iterated interpolation*. In this case recursions (1.9) and (1.10) become

$$A_{ij} = \frac{(\bar{x} - x_j)\, A_{i-1,i} - (\bar{x} - x_i)\, A_{i-1,j}}{x_i - x_j} \tag{1.21}$$

$$N_{ij} = \frac{(\bar{x} - x_j)\, N_{i,j-1} - (\bar{x} - x_i)\, N_{i+1,j}}{x_i - x_j} \tag{1.22}$$

This observation allows us to use the systolic arrays explained here for iterated interpolation by *re-programming* the processors to use the above recursions rather than (1.9) and (1.10).

There are several extensions of the material that we have presented. We sketch some of them here. A presentation of details is tedious but straightforward, hence omitted. First, the 2-level process dependence graph for computing generalized divided differences admits composite embeddings, each of which possesses its own data flow and spacetime tradeoffs. For example, the 2-level process dependence graph for computing generalized divided differences may be embedded in spacetime such that the top level graph has a spatial projection that is a 1-D array of processing elements, while the bottom level graph — the rectangular mesh process dependence graph — is embedded so that its spatial projection is a 2-D array of processing elements. Its dual embedding, a 2-D triangular array of 1-D arrays, also is possible.

Second, one can 'compose' our solutions for 1) generalized divided differences, 2) multivariate computation, and 3) problem 'partitioning'. That is, the schemes and arrays presented can be implemented hierarchically. For example, we can use the same problem partitioning approach to decompose a large multivariate computation of generalized divided differences.

All of these design options have a place, indicating the potential usefulness of software implementations on a programmable systolic/wavefront array. Examples of such software-oriented systolic computing systems include 1) an array of Transputers† [INMO86] 2) the

† Transputer is a trademark of INMOS, Ltd.

Warp [AAGK87] and 3) the Matrix-1 [FoSc87]. The Warp and Matrix-1 currently use a linear (ring) array of processing elements. Transputers can be connected into a two-dimensional mesh of processors. The processing elements of these machines provide 32-bit floating point hardware.

Table 1.1: Complexities for various embeddings of G_{GA} , a 2-level process dependence graph for computing generalized divided differences.

Embedding	Time complexity	Space complexity
McKeown (E_1)	$m_0 + m_n - 1 + 2\sum_{i=1}^{n-1} m_i$	$\sum_{i=0}^{n-1} m_i$
Optimal McKeown (E_2)	$m_0 + m_n - 1 + 2\sum_{i=1}^{n-1} m_i$	$\left\lceil \frac{1}{2}\sum_{i=0}^{n-1} m_i \right\rceil$
Optimal ring (E_3)	$m_0 + m_n - 1 + 2\sum_{i=1}^{n-1} m_i$	$\left\lceil \frac{1}{2}\sum_{i=0}^{n-1} m_i \right\rceil$
Optimal bilateral (E_4)	$m_0 + m_n - 1\,2\sum_{i=1}^{n-1} m_i$	$\left\lceil \frac{1}{2}\left(\sum_{i=0}^{n} m_i - 1 \right) \right\rceil$
2-D (E_5)	$m_0 + m_n - 1 + 2\sum_{i=1}^{n-1} m_i$	$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} m_i m_j$

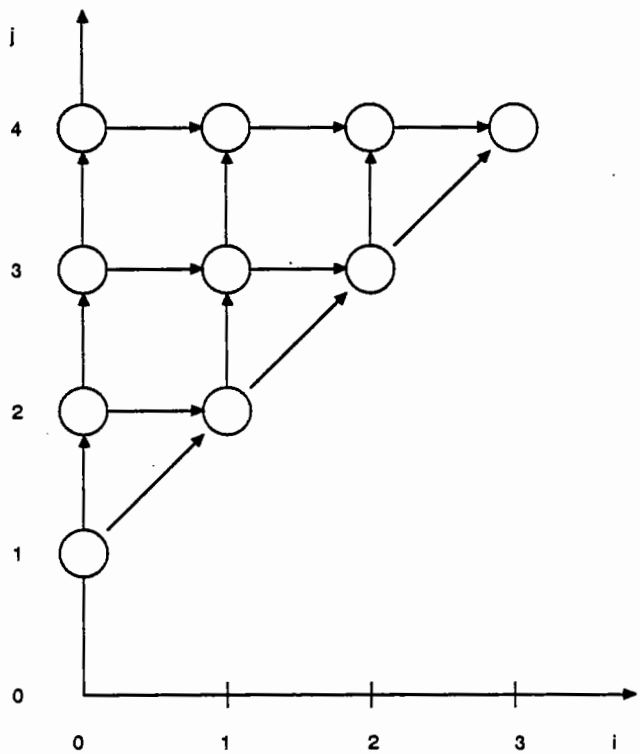


Figure 1.1. Process dependence graph of the Aitken algorithm, G_A , for $n = 4$.

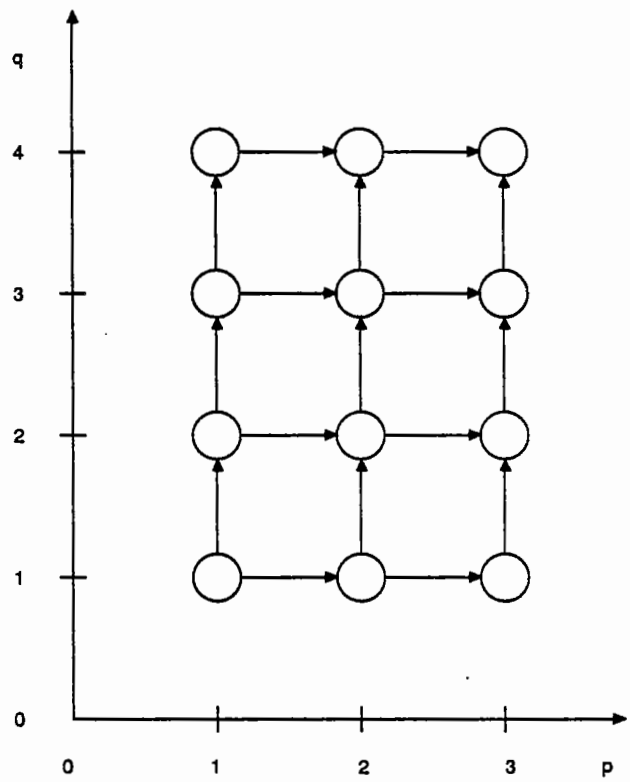


Figure 1.2. Process dependence graph of the Aitken algorithm to compute the generalized divided differences for $m_i = 3$ and $m_j = 4$.

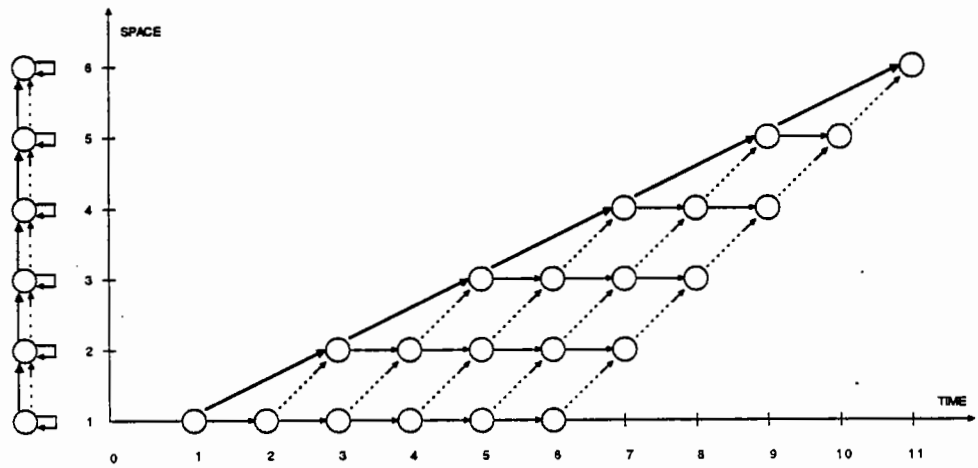


Figure 1.3. A spacetime representation of McKeown's array, for $n = 6$. Its spatial projection is a linear array of n processors.

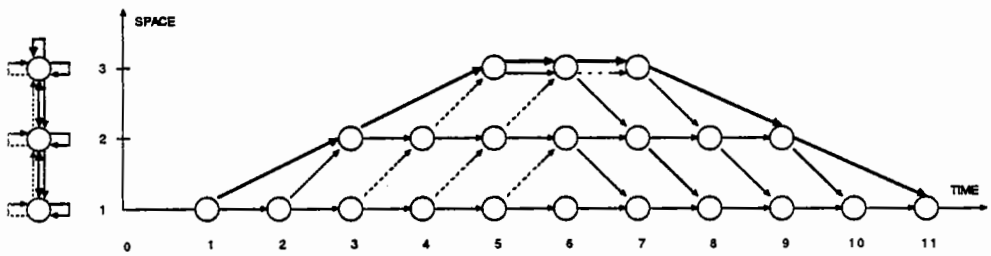


Figure 1.4. Spacetime-optimal embedding E_2 of G_A, a process dependence graph for Aitken's algorithm. Its spatial projection is a linear array of $\lceil n/2 \rceil$ processors.

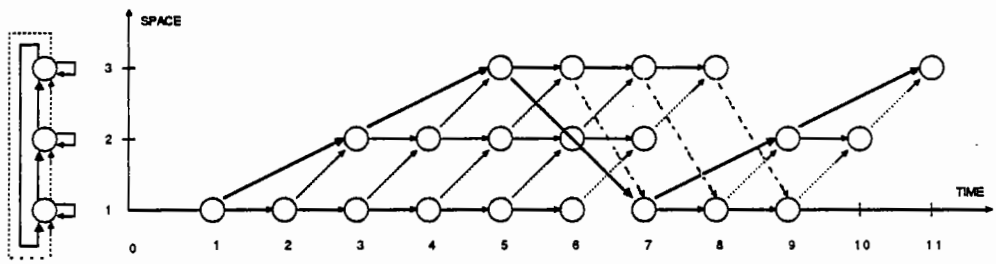


Figure 1.5. Spacetime-optimal embedding E_3 of G_A, a process dependence graph for Aitken's algorithm. Its spatial projection is a ring of $\lceil n/2 \rceil$ processors.

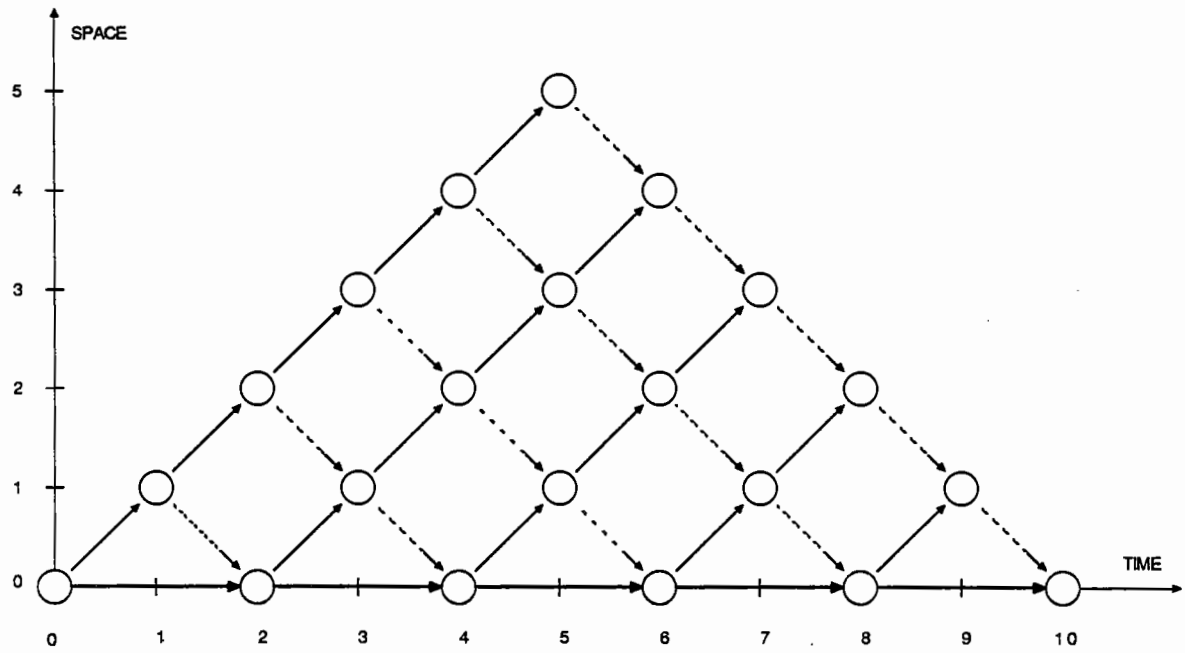


Figure 1.6. An embedding of a process dependence graph for Aitken's algorithm. Its spatial projection is a linear array of n processors.

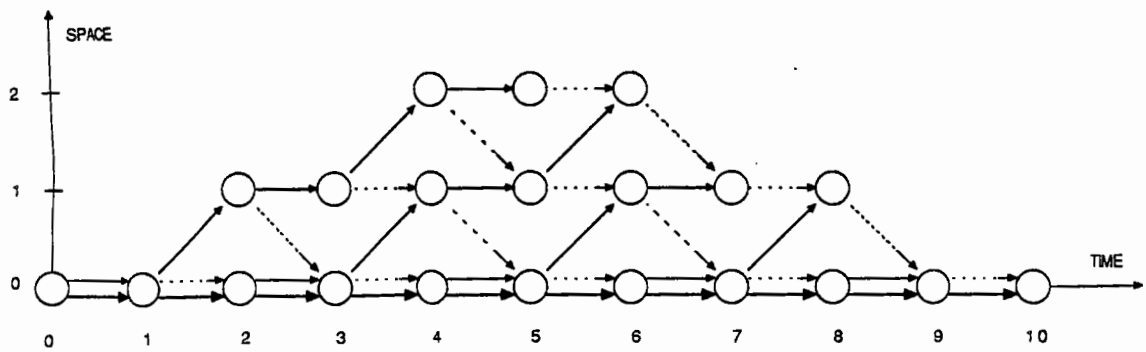


Figure 1.7. A spacetime-optimal embedding of G_A, a process dependence graph for Aitken's algorithm. Its spatial projection is a linear array of $\lceil n/2 \rceil$ processors.

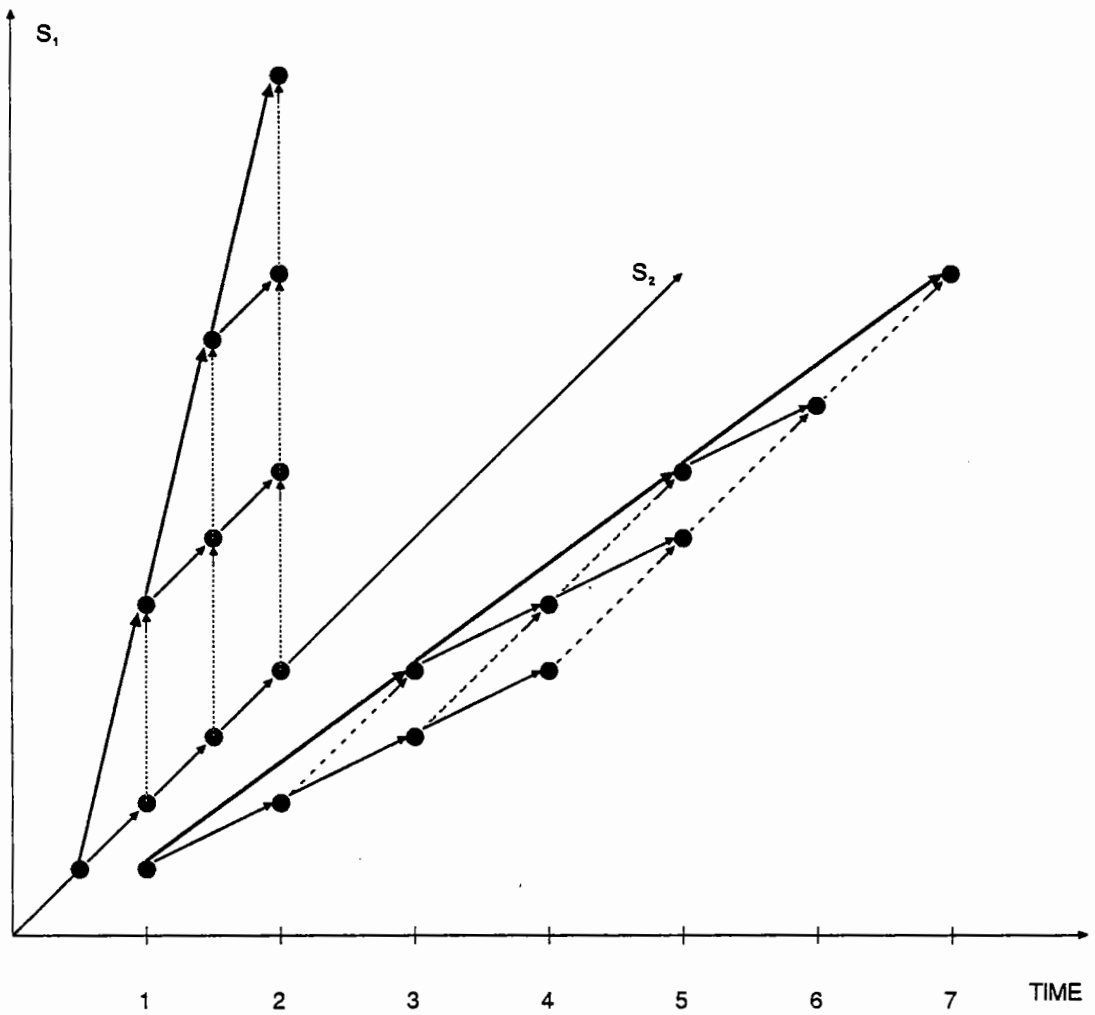


Figure 1.8. An embedding of G_A whose spatial projection is a triangular array of processors.

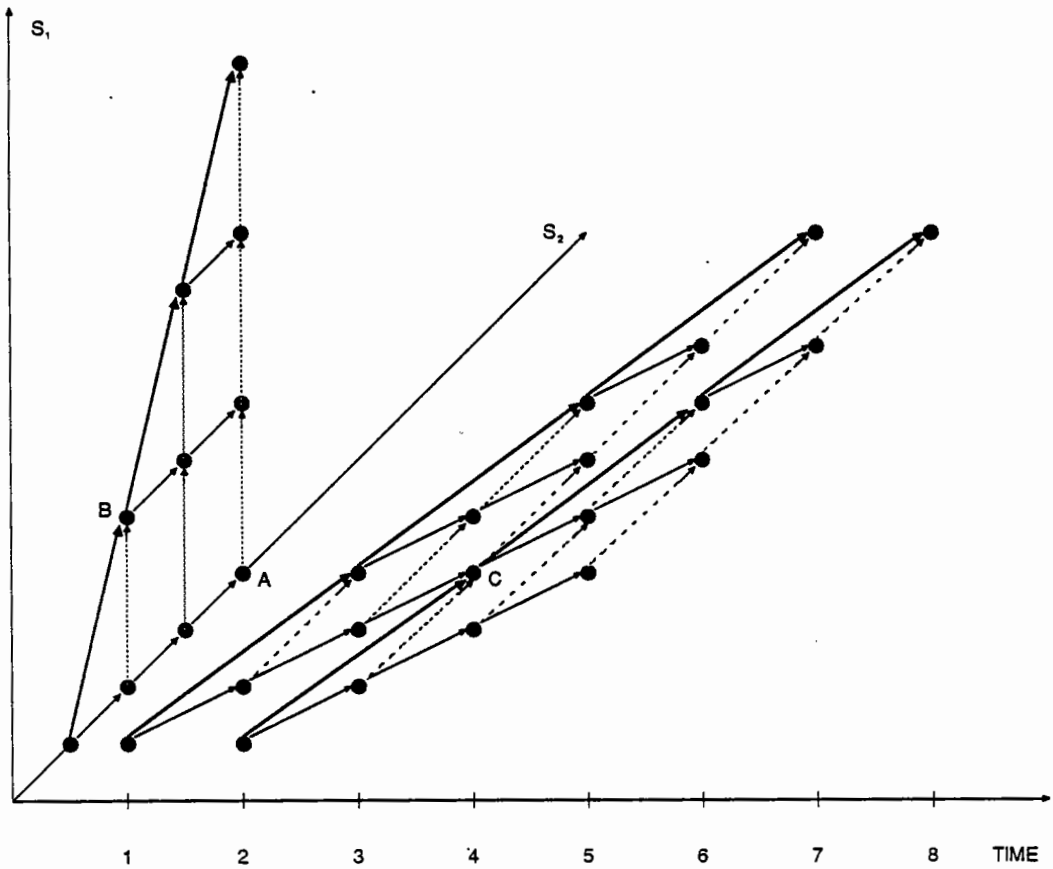


Figure 1.9. An embedding of 2 copies of G_A . There are 2 distinct processes that appear as though they are embedded in the same point (C) of spacetime. They, in fact, are embedded in distinct spatial coordinates. The process from the 1st copy of G_A executes on processor A while the process from the 2nd copy of G_A executes on processor B.

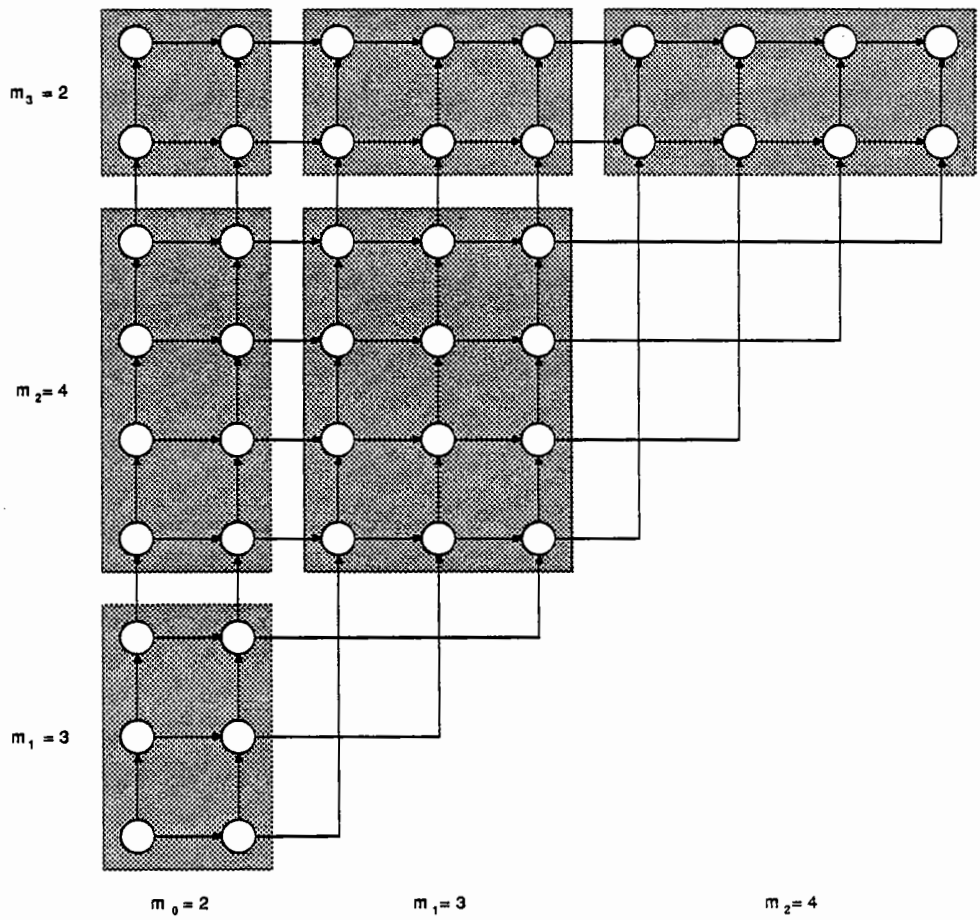


Figure 1.10. A 2-level process dependence graph, G_{GA} , for computing generalized divided differences.

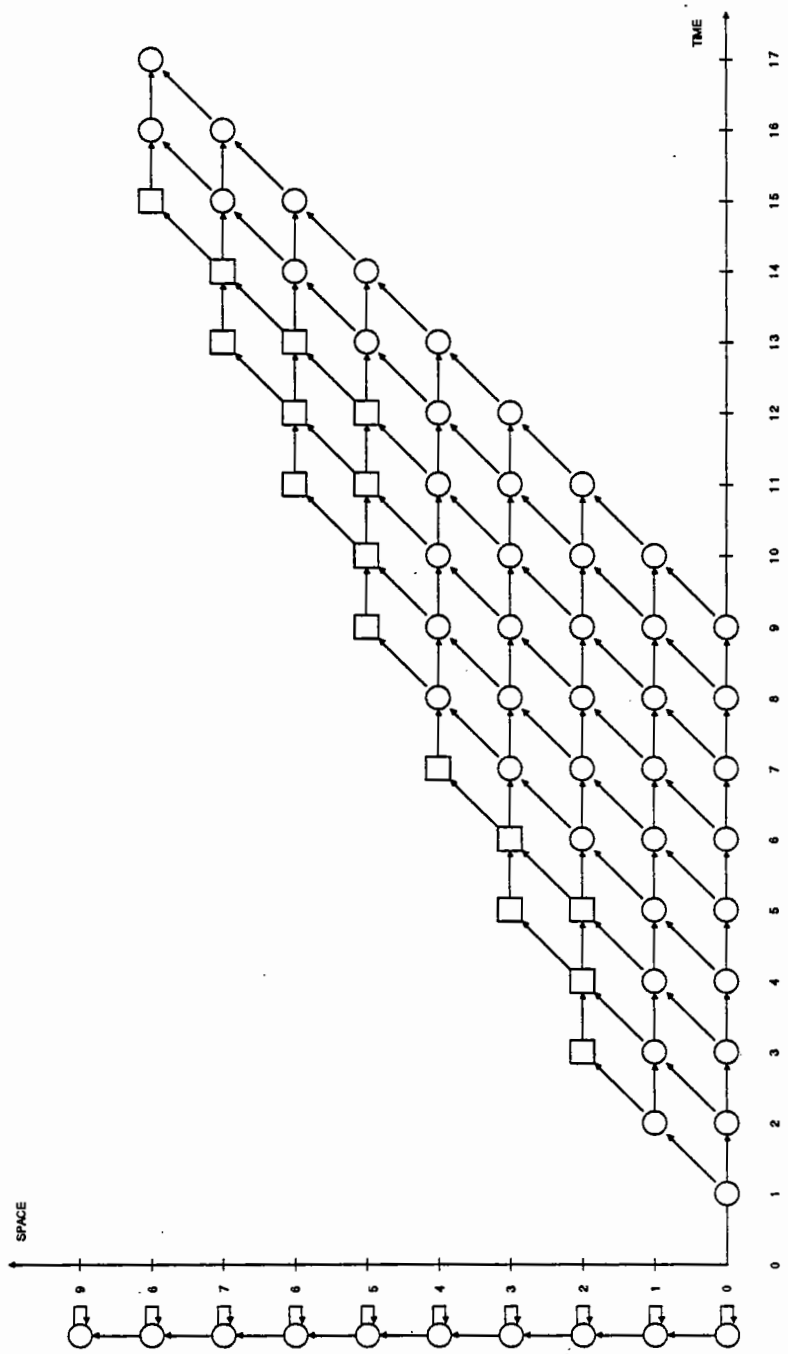


Figure 1.11. The McKeown embedding, E_1, applied to G_{GA}, a 2-level process dependence graph for computing generalized divided differences.

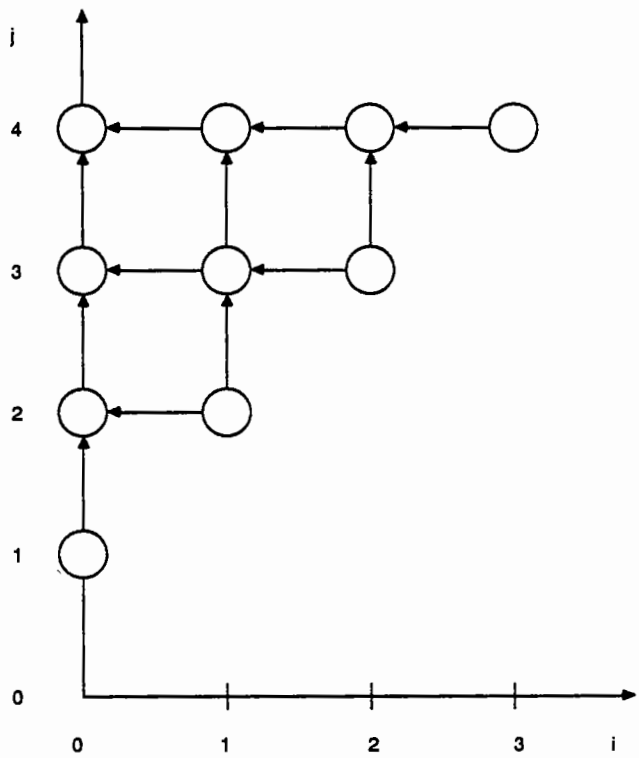


Figure 1.12. Process dependence graph of the Neville algorithm, G_N , for $n = 4$.

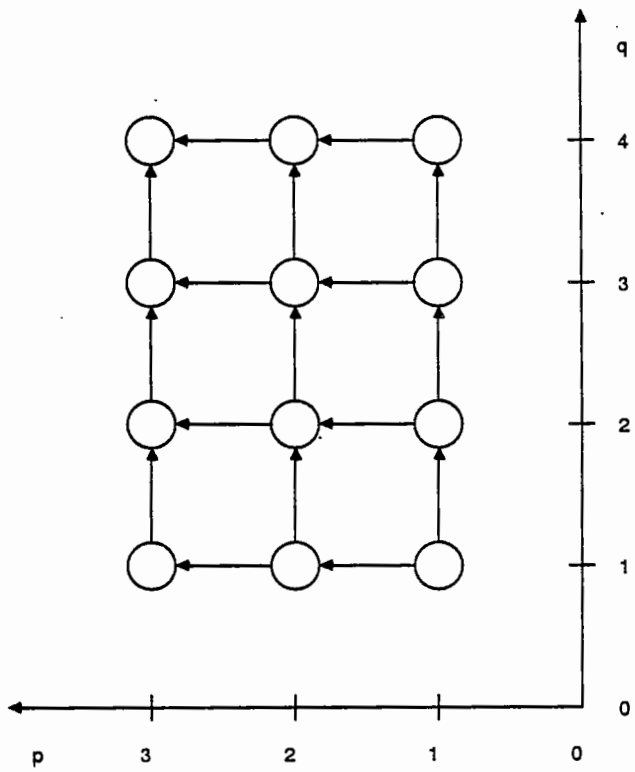


Figure 1.13. Process dependence graph of the Neville algorithm to compute the generalized divided differences for $m_i = 3$ and $m_j = 4$.

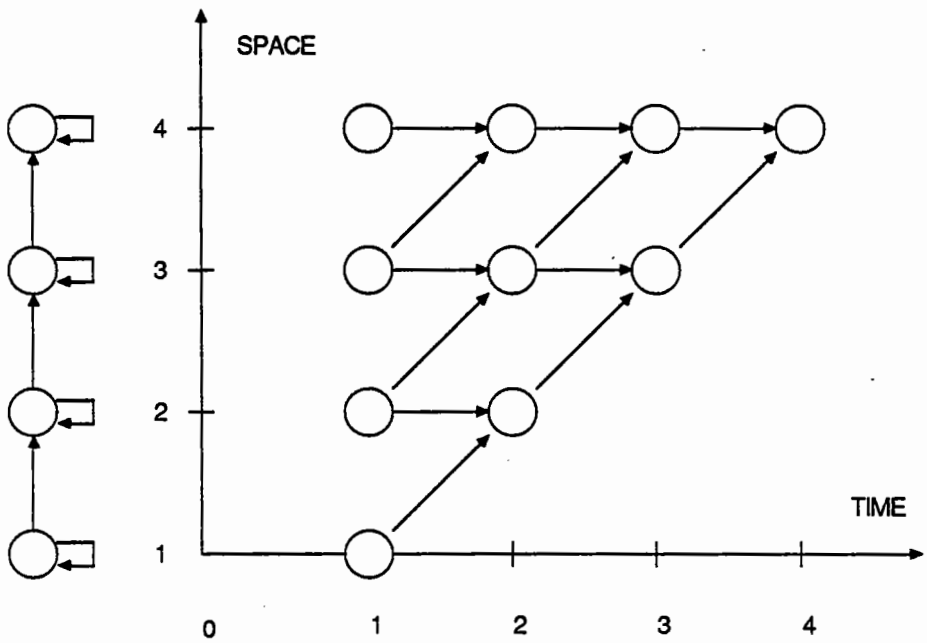


Figure 1.14. An embedding of a process dependence graph of the Neville algorithm, G_N, for $n = 4$. Its spatial projection is a linear array of n processors.

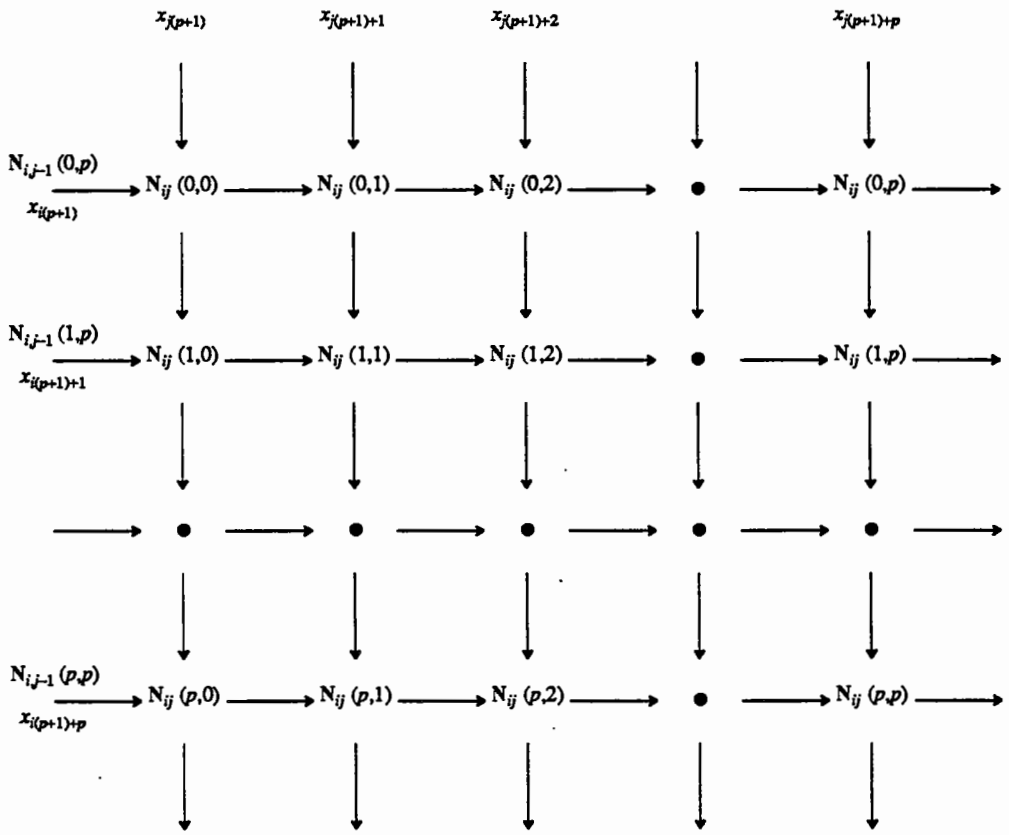


Figure 1.15. Process dependence graph for computation of of $\mathbf{N}_{ij}(r, s)$ for $0 \leq r, s \leq p$.

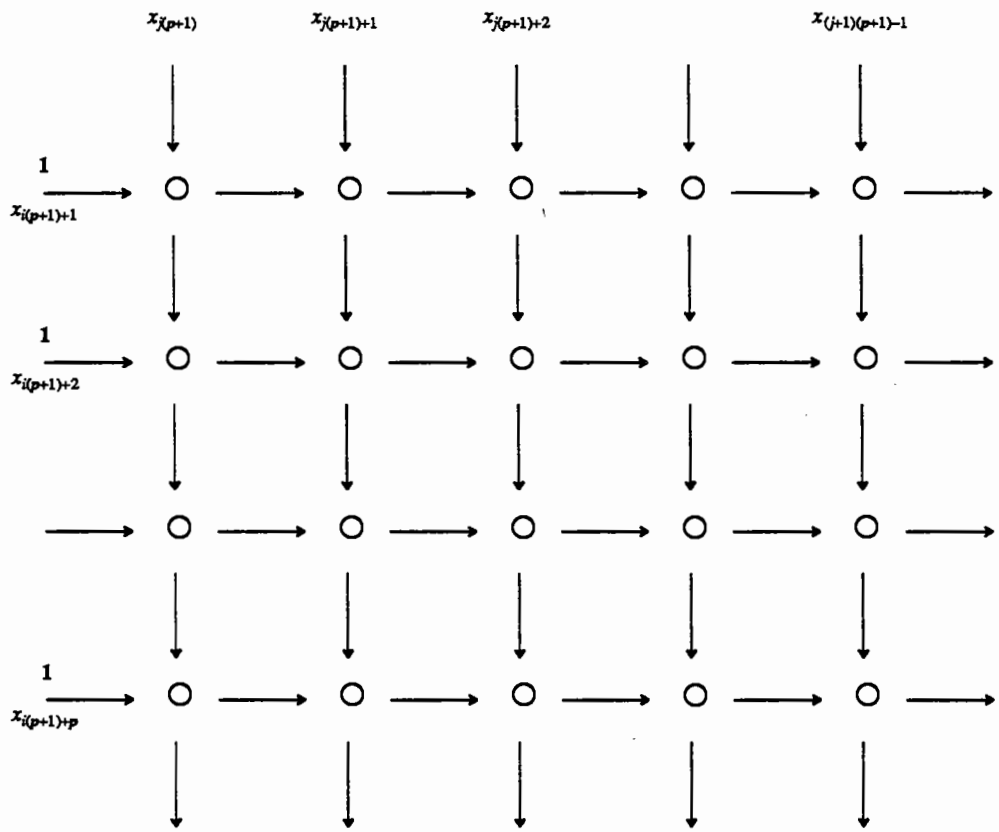


Figure 1.16. Process dependence graph for computation of $\prod_{k=j(p+1)}^{(j+1)(p+1)-1} (x_{i(p+1)+r} - x_k)$ for $1 \le r \le p$.

2

Parallel Newton Interpolation

A new parallel algorithm for Newton interpolation is presented. The algorithm makes use of 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 $2\lceil \log(n+1) \rceil + 2$ parallel arithmetic steps, and circuit size $O(n^2)$. The interpolation algorithm is shown to be numerically stable. Experiments indicate that a floating-point implementation results in error accumulation similar to that of the widely used serial algorithms. An efficient implementation of the algorithm on a hypercube parallel computer is also exhibited. Further advantages of the algorithm are that it does not require equidistant points, preconditioning, or use of the Fast Fourier Transform.

2.1. INTRODUCTION

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. [Kung73], [AhHU74], [Horo72], [Chin76], and [Reif86]), numerical algorithms are still based on the slow serial schemes or small improvements thereof, which require $O(n^2)$ operations ([Krog70], [SiVl73], and [Macl82]). In view of the increasing availability of parallel systems, the impracticability of the existing approaches can be attributed mainly to the following two causes:

(i) constant of proportionality of the order tends to be relatively 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 is self-defeating however, as polynomial interpolation is not recommended for large numbers of points. As for (ii), we may echo the remark by Miller in [Mill75], 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 $O(\log n)$. In fact the time is $2\lceil \log(n+1) \rceil + 2$ with $n(n+1)$ processors.† By *practical* we mean that the proposed algorithm is stable and may be implemented in

† All logarithms are base 2.

floating-point with resulting error accumulation similar to the stable and widely used serial algorithms.

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 $O(n^2 \log n)$ [Reif86] (cf. the definitions in Section 2.2). 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 is greater than 4 as the algorithm requires fast implementations (e.g. by means of FFT) of the forward and the inverse Discrete Fourier Transform (see Appendix 2).

The algorithm presented here makes use of the Newton representation of the interpolating polynomial. It turns out that 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 2.2) for the evaluation of the DD coefficients. The expressions obtained for the expansion of the DDs 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 (see also Chapter 3). The algorithms described may be implemented either on distributed or shared-memory machines. Efficient mapping of the parallel prefix algorithms to the cube-connected architecture makes the implementation of these algorithms on hypercube computers particularly attractive.

2.2. DEFINITIONS

An *arithmetic circuit* α over a field is a labeled finite directed acyclic graph such that: Each node v has a type

$$r(v) \in \{x_1, x_2, \cdots x_N\} \cup \{y_1, y_2, \cdots, y_M\} \cup \{+, -, \times, /\} \cup \Delta$$

(i) A node $r(v) = x_i$ must have indegree 0, and is called an *input* node.

(ii) A node $r(v) = y_i$ must have outdegree 0, and is called an *output* node.

(iii) A node $r(v) \in \{+, -, \times, /\}$ must have indegree 2, and is called an *operational* node.

(iv) A node $r(v) \in \Delta$ must have indegree 0, and is called a *constant* node, and is labeled with a constant from the field.

The *depth* of α, denoted $d(\alpha)$, is the length of the longest path in α from an input node to an output node. The *size* of α, denoted $s(\alpha)$, is the number of nodes in α. 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. Each circuit α is an abstract parallel machine which can be described by its circuit diagram. This description can be thought of as a parallel program executed by a universal circuit simulator, or as a blueprint that is read by a circuit constructor [HoRu85]. Hence $d(\alpha)$ is the time required to run the corresponding algorithm in parallel, and $s(\alpha)$ is the number of necessary circuit modules to build α in hardware.

Let D be a set closed under an associative operation $*$. The *prefix computation* problem is defined as follows: Given the elements $y_1, y_2, \cdots y_n \in D$ compute all n initial products (prefixes) $y_1 * y_2 * y_3 * \cdots * y_i$ for $i = 1, 2, \cdots, n$. The parallel algorithms for this computation are defined as parallel prefix algorithms. The following results are well known and essential for the remaining discussion [KrRS85], [LaFi80]:

Lemma 2.1

i) *The n input parallel prefix computation can be achieved in at most $\lceil \log n \rceil$ time if $p \geq n$ processors are available.*

ii) *There exists a family of arithmetic circuits $P_k(n)$ for the n input prefix computation, such that*

$$s(P_k(n)) \leq 2(1+\frac{1}{2^k})n - 4$$
$$d(P_k(n)) \leq k + \lceil \log n \rceil$$

Part (ii) of Lemma 2.1 for $k=0$ shows that n input parallel prefix can be computed in $\lceil \log n \rceil$ depth and size less than $4n$. There also exist parallel prefix circuits of size less than $4n$ with small increases to its depth [LaFi80], [Fich83], [BiGa86], [Snir86], [LaYD87]. Parallel prefix circuits first appeared in the construction of circuits to add two binary numbers [BrKu82]. It has also been shown that any sequential circuit can be efficiently transformed into a combinational circuit using the parallel prefix circuit [LaFi80]. Prefix computations also occur in the solution of linear recurrences, tridiagonal system of linear equations [KoSt73].

2.3. PARALLEL NEWTON INTERPOLATION ALGORITHM

The interpolation polynomial is completely defined once the divided difference (DD) coefficients are evaluated. In a serial setting two methods for their derivation are the Neville and Aitken methods both of sequential complexity $O(n^2)$. For example the Neville procedure uses

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

to calculate the terms in the following triangular table

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

The te diagonal are the required DDs. Note that the entries in a given column can be cal pendently of one another, and they depend only on the entries in the previous co he x_i 's. This gives a straightforward parallel algorithm for the DDs, where each omputed in constant time using as many processors as there are entries in that lus this approach requires $O(n)$ time to calculate all the DDs using $O($ prs. This approach was used in Chapter 1, and in [McKe86] for the construction rrays for parallel interpolation.

el Newton interpolation algorithm is now presented:

The

wton interpolating polynomial for $n+1$ points can be computed in $2\lceil \rceil + 2$ parallel arithmetic steps using $n(n+1)$ processors and can be imple- me arithmetic circuit having the same depth, and size $O(n^2)$.

Pr

d of the Neville or the Aitken recurrence formulae, our point of departure is an a formulation for the DDs given in [AbSt65], [Davi75], and elsewhere. This can be s ollows: Let $y_{ij} = (x_i - x_j)$ for $i \neq j$ and $i, j = 0, 1, \cdots, n$. Then the k^{th} I can be expressed as a linear combination of the given values $f_0, f_1, \cdots, f_k$

with coefficients that are inverses of products of the y_{ij} 's in the form

$$f_{012\cdots k} = \frac{f_0}{y_{01}y_{02}\cdot\cdot y_{0k}} + \frac{f_1}{y_{10}y_{12}\cdot\cdot y_{1k}} + \cdots + \frac{f_k}{y_{k0}y_{k1}\cdot\cdot y_{k,k-1}} \qquad (2.1)$$

where k ranges from 0 to n . Denoting the inverse of the coefficient of f_i in the linear expansion of $f_{012\ldots k}$ ($i \le k$) as $f_{012\ldots k}\,|_{f_i}$, the coefficients in the expansion (2.1) can be written as

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

Hence

$$f_{01}\,|_{f_0} = y_{01}$$

$$f_{012}\,|_{f_0} = y_{01}y_{02}$$

$$f_{0123}\,|_{f_0} = y_{01}y_{02}y_{03}$$

$$\cdots$$

$$f_{012\ldots n}\,|_{f_0} = y_{01}y_{02}y_{03}\cdots y_{0n}$$

and therefore the computation of this sequence of coefficients amounts to the calculation of the prefixes of the quantities ($y_{01}, y_{02}, \cdots, y_{0n}$). From Lemma 2.1 this can be done by using the parallel prefix algorithm in $\lceil \log n \rceil$ time with n processors. Since $n+1$ concurrent instances of a parallel prefix algorithm are needed to compute the prefixes of the terms ($y_{i0}, y_{i1}, \cdots, y_{i,i-1}, y_{i,i+1}, \cdots, y_{in}$) for i ranging from 0 to n , $O(n^2)$ processors suffice. Note that if all $f_{012\ldots k}\,|_{f_i}$'s are known for $i \le k \le n$, then the DDs $\{f_0, f_{01}, f_{012}, \cdots, f_{012\ldots n}\}$ 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 the y_{ij} 's can

be calculated in a single step using $\frac{n(n+1)}{2}$ processors for all $i, j = 0, 1, \cdots, n$ 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 $k = 1, 2, \cdots, n$ 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 $\frac{n(n+1)}{2}$ 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 another $\lceil \log(n+1) \rceil$ arithmetic steps using $n(n+1)$ processors. Thus $n(n+1)$ processors suffice to compute all $f_{012\ldots k}$ for $k = 0, 1, \cdots, n$ in $2\lceil \log(n+1) \rceil + 2$ parallel time, proving the first part of the theorem.

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 $4(n+1)$. Consider each of the steps in the algorithm: At first all of the y_{ij}'s are calculated. This can be done in one step with $\frac{n(n+1)}{2}$ 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 $\frac{n(n+1)}{2}$ 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 $O(n^2)$. Adding these contributions we have the second part of the theorem. ●

By making use of further results from [LaFi80], [BiGa86], [Snir86], and [LaYD87] tighter bounds may be found for the circuit size by only small increases to its depth.

Figures 2.1 and 2.2 illustrate the structure of the algorithm. The parallel prefix circuit in Figure 2.2 is a pictorial representation of the parallel prefix algorithm for $n = 7$ given in [LaFi80].

The following observation will be important in the discussions for the algorithm's implementation in the limited processor and hypercube cases: 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 basic steps of the interpolation can be implemented using the same type of algorithm and circuit. The algorithm is expressed as follows:

Parallel Newton Interpolation Algorithm

Step 1. Compute all of $y_{ij} = x_i - x_j$ for $0 \le i \ne j \le n$ setting $y_{ii} = 1$.

Step 2. For all $i = 0, 1, \cdots, n$ compute in parallel the prefixes $Y_{ij} = y_{i0} y_{i1} \cdots y_{ij}$ of the terms $(y_{i0}, y_{i1}, y_{i2}, \cdots, y_{in})$,

Step 3. For all $i = 0, 1, \cdots, n$ and for all $j = 0, 1, \cdots, n$ set $Z_{ij} = \frac{f_i}{Y_{ij}}$ for $i \le j$,

Step 4. Compute in parallel the sum of the terms $(Z_{0j}, Z_{1j}, Z_{2j}, \cdots, Z_{jj})$ for all $j = 0, 1, \cdots, n$; at the end of this last step, the j^{th} DD is the term $Z_{0j} + \cdots + Z_{jj}$. According to the previous comments, the summation can be performed by parallel prefix.

2.4. LIMITED PROCESSORS CASE

In the case of a limited number of processors p, in particular when $p = m(n+1)$ where $1 < m < n+1$ is an integer, the parallel Newton interpolation can still be efficiently performed. For simplicity it is assumed that the parallel additions in the last step are done using the parallel prefix algorithm. It is shown that one instance of parallel prefix for $n+1$ elements can be computed in

$$2\left\lceil \frac{n+1}{m} \right\rceil + \lceil \log m \rceil - 2$$

parallel arithmetic steps when m processors are available [KrRS85]. Since the parallel Newton interpolation algorithm requires $n+1$ concurrent instances of parallel prefix, we distribute $p = m(n+1)$ processors among concurrent parallel prefix operations equally such that each of them is allocated to m processors, where $1 < m < n+1$. One can then determine the time for the parallel Newton algorithm in this case by applying limited processor parallel prefix for the product and summation parts of the algorithm and adding the times needed for the parallel subtraction and division of about $(n+1)^2$ elements by means of $m(n+1)$ processors. It can be easily shown that:

Theorem 2.2

The parallel Newton algorithm for $n+1$ data points can be done in less than

$$6\left\lceil \frac{n+1}{m} \right\rceil + 2\lceil \log m \rceil - 4$$

arithmetic steps when $p = m(n+1)$ processors are available.

Proof :

The Steps 1 and 3 of the algorithm can be done using no more than $\frac{(n+1)^2}{p}$ opera-

tions each with p processors. Since $p = m(n+1)$ these two steps will take $2\left\lceil \frac{n+1}{m} \right\rceil$ time together.

To compute $n+1$ concurrent instances of the parallel prefix operations we distribute $m(n+1)$ processors such that each parallel prefix instance is computed by m processors. It follows from the above discussion that the Steps 2 and 4 of the algorithm will take $2\left\lceil \frac{n+1}{m} \right\rceil + \lceil \log m \rceil - 2$ time each. Thus the total result follows. ●

If the number of available processors p is less than $n+1$ then the above algorithm 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. For 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 $\frac{n(n+1)}{2}$ arithmetic steps.

2.5. PERMANENCE PROPERTY FOR PARALLEL INTERPOLATION

One important advantage in using the Newton representation for polynomial interpolation as opposed to alternative forms (e.g. Lagrangian) is the *permanence property*. Let the Newton interpolating polynomial be based upon the set of values $\{x_i, f_i\}$ for $i = 0, 1, \cdots, n$. Permanence simply means the ability to add a new pair of interpolation points $\{x_{n+1}, f_{n+1}\}$ at minimal cost. The complexity of finding the new divided difference $f_{012..n,n+1}$ in the single processor case amounts to calculating a new row of the divided-

difference table.

If the partial values are saved, then the computation of the new row (thus the new divided difference $f_{012..n,n+1}$) amounts to $3(n+1)$ arithmetic operations. The computation of the whole table for $n+2$ input pairs would amount to $\frac{3}{2}(n+1)(n+2)$ arithmetic operations. Hence there is considerable economy involved in this process. Now we show that the parallel Newton interpolation algorithm has permanence property as well:

Theorem 2.3

The divided difference $f_{012..n,n+1}$ *can be computed in* $2\lceil \log(n+2) \rceil + 2$ *arithmetic steps with* $n+2$ *processors if the partial results of the parallel Newton interpolation algorithm with the input* $\{ x_i, f_i \}$ *for* $0 \le i \le n$ *are reused.*

Proof :

The following algorithm computes $f_{012..n,n+1}$ in $2\lceil \log(n+2) \rceil + 2$ arithmetic steps using $n+2$ processors:

Step 1. Compute $y_{i,n+1} = x_i - x_{n+1}$ for $i = 0, 1, \cdots, n$ using $n+1$ processors. This step takes 1 parallel subtraction.

Step 2. Compute $Y_{n+1,n} = y_{n+1,0}\, y_{n+1,1} \cdots y_{n+1,n}$ with $n+1$ processors by either using a binary tree multiplication or the parallel prefix algorithm. This step takes $\lceil \log(n+1) \rceil$ parallel multiplications with $n+1$ processors.

Step 3. Compute $W_i = \frac{Z_{in}}{y_{i,n+1}}$ for $i = 0, 1, \cdots, n$ and $W_{n+1} = \frac{f_{n+1}}{Y_{n+1,n}}$. This step takes 1 parallel division with $n+2$ processors. The terms Z_{in} are computed in Step 3 of the

parallel Newton interpolation algorithm.

Step 4. Compute $f_{012..n,n+1} = W_0 + W_1 + \cdots + W_{n+1}$ using $n+2$ processors by either the parallel prefix or a binary tree addition algorithm. This step takes $\lceil \log(n+2) \rceil$ parallel additions.

Thus $n+2$ processors will suffice to compute $f_{012..n,n+1}$ in $2\lceil \log(n+2) \rceil + 2$ parallel arithmetic steps. ●

Thus for parallel computation the permanence property allows a saving in terms of the number of processors to compute the new divided difference. The computation of the new set of coefficients of the interpolating polynomial with the introduction of the new point requires $O(n)$ processors for the parallel Newton interpolation algorithm whereas the Lagrange interpolation by using FFT [Reif86] requires $O(n^2)$ processors since this algorithm has to be applied from the beginning.

2.6. EVALUATION OF THE INTERPOLATING POLYNOMIAL

Whenever polynomial interpolation is used, it is also required to 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 [Maru73], [MuPa73]. Given n processors, a polynomial of degree n can be evaluated in $O(\log n)$ arithmetic steps. 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 can be used to evaluate the Newton interpolating polynomial $p_n(x)$ at x in $2\lceil \log(n+1) \rceil + 2$ arithmetic steps with n processors. It is based on the same idea as the parallel Newton interpolation algorithm given in Section 2.3. Briefly, it consists of the following steps:

Step 1. Compute all of the differences $y_i = x - x_i$ for $0 \le i \le n-1$,

Step 2. Compute the prefixes $Y_i = y_0 y_1 \cdots y_i$ of the terms $(y_0, y_1, y_2, \cdots, y_{n-1})$,

Step 3. Compute the quantities $f_{012..i}\, Y_{i-1}$ for $0 < i \le n$,

Step 4. Add the terms computed in Step 3 to obtain $p_n(x)$.

Theorem 2.4

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 2.1, and will be omitted. ●

By a straightforward extension it can also be shown that the evaluation can be done at n points with a circuit of the same depth and size $O(n^2)$, which are the same values obtained for the interpolation.

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 Newton interpolation algorithm: the same parallel prefix algorithm is used to carry out Step 2, and a (possibly prefix based) summation algorithm is necessary in Step 4. This uniformity may be desirable for VLSI implementation of the algorithms.

2.7 CONVERSION FROM NEWTON FORM TO STANDARD FORM

Given a polynomial in the Newton form, sometimes it is desirable or necessary to bring it to the standard form. That is we need to find the coefficients $a_0, a_1, \cdots, a_n$ such that

$$\sum_{k=0}^{n} a_k x^k = \sum_{j=0}^{n} c_j \prod_{i=0}^{k-1} (x - x_i)$$

where c_j 's are the DD coefficients. By repeated application of the Horner algorithm for the Newton polynomial we can find a_k for $0 \le k \le n$ in $O(n^2)$ time sequentially [BjPe70]. Here we will give an algorithm to achieve this in $O(\log n)$ time using $O(n^2)$ processors. The technique is based on parallel prefix and Fast Fourier Transform (FFT) algorithms.

We denote the coefficients of the monomial $x - x_i$ with $\alpha_i \in R^{n+1}$ such that $\alpha_i = [-x_i, 1, 0, \cdots, 0]$. Let $P_j(x)$ be the product of the monomials up to $j-1$, e.g. $P_j(x) = (x - x_0)(x - x_1) \cdots (x - x_{j-1})$ with $P_0(x) = 1$. Also suppose we are given $n+1$ vectors $p_j \in R^{n+1}$ for $j = 0, 1, \cdots, n$. Each vector $p_j = [p_{j,0}, p_{j,1}, \cdots, p_{j,n}]$ gives the coefficients of the polynomial $P_j(x)$ in the standard form. Clearly $p_{j,k} = 0$ for $j < k$. With this notation the Newton polynomial can be written as

$$\begin{aligned} \sum_{k=0}^{n} a_k x^k &= \sum_{j=0}^{n} c_j P_j(x) \\ &= \sum_{j=0}^{n} c_j \sum_{k=0}^{n} p_{j,k} x^k \\ &= \sum_{k=0}^{n} x^k \sum_{j=0}^{n} c_j p_{j,k} \end{aligned}$$

Thus $a_0, a_1, \cdots, a_n$ are found in terms of $p_{j,k}$ as

$$a_k = \sum_{j=0}^{n} c_j\, p_{j,k} = \sum_{j=k}^{n} c_j\, p_{j,k} \qquad \text{for} \quad k = 0, 1, \cdots, n$$

Since $P_0(x) = 1$ and $P_1(x) = x - x_0$ we have

$$[p_{0,0}, p_{0,1}, \cdots, p_{0,n}]^T = [1, 0, \cdots, 0]^T$$
$$[p_{1,0}, p_{1,1}, \cdots, p_{1,n}]^T = [-x_0, 1, 0, \cdots, 0]^T = \alpha_0$$

Now we will show that the coefficients $p_{j,k}$ for $j \geq 1$ can be found by application of the FFT and parallel prefix algorithms. Using the convolution theorem we can write [AhHU74]

$$[p_{2,0}, p_{2,1}, \cdots, p_{2,n}]^T = \alpha_0 \otimes \alpha_1$$
$$= DFT_{n+1}^{-1}\,[\,DFT_n(\alpha_0) \circ DFT_{n+1}(\alpha_1)\,]$$

where DFT_{n+1} and DFT_{n+1}^{-1} denote the Discrete Fourier Transform operator and its inverse, and $\circ$ denotes pairwise (Hadamard) product. Let $\beta_i = DFT_{n+1}[\,\alpha_i\,]$, then we have

$$[p_{2,0}, p_{2,1}, \cdots, p_{2,n}]^T = DFT_{n+1}^{-1}\,[\,\beta_0 \circ \beta_1\,]$$

In the general case

$$[p_{j,0}, p_{j,1}, \cdots, p_{j,n}]^T = \alpha_0 \otimes \alpha_1 \otimes \cdots \otimes \alpha_{j-1}$$
$$= DFT_{n+1}^{-1}\,[\,\beta_0 \circ \beta_1 \circ \cdots \circ \beta_{j-1}\,]$$

for $j = 1, 2, \cdots, n$. To compute the coefficients of the polynomials $P_j(x)$ for all $j \geq 1$ we make the following observation

$$[p_{1,0}, p_{1,1}, \cdots, p_{1,n}]^T = DFT_{n+1}^{-1}\,[\,\beta_0\,]$$
$$[p_{2,0}, p_{2,1}, \cdots, p_{2,n}]^T = DFT_{n+1}^{-1}\,[\,\beta_0 \circ \beta_1\,]$$
$$[p_{3,0}, p_{3,1}, \cdots, p_{3,n}]^T = DFT_{n+1}^{-1}\,[\,\beta_0 \circ \beta_1 \circ \beta_2\,]$$
$$[p_{4,0}, p_{4,1}, \cdots, p_{4,n}]^T = DFT_{n+1}^{-1}\,[\,\beta_0 \circ \beta_1 \circ \beta_2 \circ \beta_3\,]$$
$$\cdots$$
$$[p_{n,0}, p_{n,1}, \cdots, p_{n,n}]^T = DFT_{n+1}^{-1}\,[\,\beta_0 \circ \beta_1 \circ \beta_2 \circ \cdots \beta_{n-1}\,]$$

Thus we see that the parallel prefix algorithm can be used here to compute the prefix (Hadamard) products of the vectors $\beta_0, \beta_1, \cdots, \beta_{n-1}$. Then we need to apply the inverse DFT operator to find all $p_{j,k}$ for $0 \le j, k \le n$.

The parallel algorithm to compute the coefficients of the Newton polynomial in the standard form proceeds as follows:

Step 1. Compute $\beta_i = DFT_{n+1}[\alpha_i]$ for all $i = 0, 1, n-1$. This can be done by running n concurrent instances of the FFT algorithm in parallel. Thus the corresponding circuit has depth $O(\log n)$ and size $O(n^2 \log n)$ [Reif86].

Step 2. Compute prefix (Hadamard) products $(\gamma_0, \gamma_1, \cdots, \gamma_{n-1})$ of the vectors $(\beta_0, \beta_1, \cdots, \beta_{n-1})$ using the parallel prefix algorithm. This step can be done in $\log n$ time with a circuit of size $O(n^2)$ due to Lemma 2.1.

Step 3. Define $p_{0,0} = 1$ and compute $p_i = DFT_n^{-1}[\gamma_i]$ for all $i = 1, 2, \cdots, n$. This step, similar to (1), can be achieved in depth $O(\log n)$ and size $O(n^2 \log n)$.

Step 4. Compute the coefficients a_k for all $k = 0, 1, \cdots, n$ using

$$a_k = \sum_{j=k}^{n} c_k \, p_{j,k}$$

which can be done in $O(\log n)$ time using $O(n^2)$ processors.

Thus we see that by using the FFT and parallel prefix algorithms, a Newton polynomial can be brought to its standard form in $O(\log n)$ time with $O(n^2)$ processors. Due to the FFT algorithm the corresponding circuit is of size $O(n^2 \log n)$.

This algorithm can be used to solve the Vandermonde systems. Suppose $x_0, x_1, \cdots, x_n \in R$ are given. The Vandermonde matrix $V \in R^{(n+1)\times(n+1)}$ is defined as

$$V(x_0, x_1, \cdots, x_n) = \begin{bmatrix} 1 & 1 & . & . & 1 \\ x_0 & x_1 & . & . & x_n \\ . & . & . & . & . \\ . & . & . & . & . \\ x_0^n & x_1^n & . & . & x_n^n \end{bmatrix}.$$

A linear system of equations of the form $V^T a = f$ involving the transpose of the matrix V and two vectors $f, a \in R^{n+1}$ is equivalent to a polynomial interpolation problem. Thus a solution exists if $x_i \neq x_j$ for $i \neq j$. The sequential algorithm explained in [GoVa83] solves the above system in $O(n^2)$ time first by computing c_k for $0 \leq k \leq n$ using the Neville algorithm, and then by computing a_k for $0 \leq k \leq n$ using repeated application of the Horner algorithm.

The Vandermonde systems can be solved in $O(\log n)$ time using the parallel Lagrange interpolation algorithm in [Reif86]. This algorithm directly computes a_k for $0 \leq k \leq n$ (see Appendix 2). An alternative approach would be to use the parallel Newton interpolation algorithm to find c_k for $0 \leq k \leq n$, and then compute a_k for $0 \leq k \leq n$ using the above algorithm. Whether we use the Lagrange interpolation approach, or the Newton interpolation approach the corresponding circuits have the same depth, $O(\log n)$, and the sime size, $O(n^2 \log n)$.

2.8. MAPPING ON A HYPERCUBE

Although the described algorithms are suitable for the shared memory environments, it is also interesting to examine their implementation on distributed memory systems. It is well

known that unless great care is taken in mapping algorithms to such systems, the communication cost involved could greatly degrade the performance. It will be shown that the properties of Gray code mapping and the communication requirements of parallel prefix allow the mapping of the parallel Newton interpolation algorithm on a hypercube at a very small cost. This architecture has been studied extensively [Seit85], and many versions are commercially available (intel iPSC series, Ncube).

In a d dimensional hypercube connected parallel computer there exist 2^d nodes connected in a boolean-cube fashion. Thus in a d dimensional cube every node has d neighbors. Data transfers are performed by passing messages between neighbors. Thus a data packet which must travel from node A to node B must cross a sequence of nodes starting at node A and ending at node B. The length of the shortest path between any two nodes A and B in a d dimensional cube is at most d. It is assumed that it takes 1 unit time to move a data item from a node to any of its neighbors [SaSc85a], [SaSc85b].

First we give a definition of a hypercube connected parallel computer:

Definition: *Hypercube connected parallel computer:* If $p = 2^d$ and $b_d \cdots b_1$ is the binary representation of b for $b \in [0, .., p-1]$ and $b^{(i)}$ is the number whose binary representation is $b_d ... b_{i+1} \bar{b_i} b_{i-1} \cdots b_1$, where $\bar{b_i}$ is the complement of b_i and $1 \le i \le d$ then in a hypercube connected computer, processing element b is connected to processing element $b^{(i)}$, for $1 \le i \le d$.

Now we give the definition of the binary-reflected Gray code and a lemma related to the mapping of the parallel prefix algorithm on the cube:

Definition: *Binary-reflected Gray code:* $G(b) = g_d g_{d-1} \cdots g_1$ of a d-bit binary number $b = b_d b_{d-1} \cdots b_1$ is defined by setting [ReND77]

$$g_i = b_i + b_{i+1} \ mod\ 2\ , \quad \text{for } i = 1, 2, \ldots, d-1\ , \quad g_d = b_d\ .$$

Lemma 2.2

If b and c are two d-bit binary numbers such that $0 \le b \le 2^d - 1 - 2^{k-1}$ and $c = b + 2^{k-1}$ then the Hamming distance between $G(b)$ and $G(c)$ is 1 if $k = 1$ and 2 if $2 \le k \le d$. Furthermore the communication paths are disjoint.

(For proof see Lemma 5.1 in [John87a].)

Binary-reflected Gray code allows efficient implementation of the parallel prefix algorithm on a hypercube.

Lemma 2.3

The prefixes of the terms $y_0, y_1, \cdots, y_n$ can be computed in $\log(n+1)$ arithmetic and $2\log(n+1) - 1$ routing steps on a hypercube with $n + 1 = 2^r$ nodes.

Proof :

Partition the element y_i to the node $G(i)$ for all $0 \le i \le n$. In the k^{th} step of the parallel prefix algorithm, the node which has the element y_i needs to communicate with the node which has the element y_{i+2^k} for $k = 0, 1, \cdots, \log(n+1) - 1$ and $i = 0, 1, \cdots, n + 1 - 2^k$. It follows from the properties of the binary-reflected Gray code that the first step takes 1 arithmetic and 1 routing step, and the steps $1, 2, \cdots, \log(n+1) - 1$ takes 1 arithmetic and 2 routing steps each. ●

Now we give a theorem regarding the implementation of the parallel Newton interpolation algorithm on a hypercube:

Theorem 2.5

The parallel Newton interpolation algorithm with $n+1$ *inputs computes the interpolating polynomial in* $2\log(n+1)+2$ *arithmetic and* $4\log(n+1)-2$ *routing steps on a hypercube with* $2^d=(n+1)^2$ *nodes.*

Proof :

The cube interpolation algorithm proceeds as follows: For the case of $n+1=2^r$ data elements we assume a hypercube with $N=(n+1)^2$ nodes. Define $H = b_{2r} \cdots b_{r+1}$ and $L = b_r \cdots b_1$. Then the address of each node can be written as $[H\ L]=b_{2r} \cdots b_{r+1}\, b_r \cdots b_1$. This implies that we partition the $(n+1)^2$ processors into $n+1$ groups of $n+1$ nodes each so that the r most significant bits H of the processor address give the group number and the r least significant bits L give the node number in the group [Capp87]. Initially node $G(j)$ in group $G(i)$ contains x_j, x_i and F_i, where $F_i=f_i$ if $i \le j$ and 0 otherwise (in this discussion no account is taken of the time required to load operands in the nodes). In the first step all nodes perform a subtraction using locally available data: $y_{ij}=x_i-x_j$. Processors with identical group and node numbers are masked and store 1 in the result locations y_{ii}. The first step does not involve any routing of the data. At the start of the first parallel prefix pass, y_{ij} and F_i are in node $G(j)$ of group $G(i)$. Thus by applying Lemma 2.3, we observe that after $2\log(n+1)-1$ routing and $\log(n+1)$ arithmetic steps each group of nodes returns the required constituent elements of the DD. The third step is the parallel division, which is done in a single arithmetic step with local data, since the partial result $y_{i0}\, y_{i1} \cdots y_{ij}$ and F_i are now in node $G(j)$ of group $G(i)$. From the definition of the F_i it is seen that the (redundant) partial summands for the next step are set to zero. For the last step of the algorithm, we use a partition orthogonal to the previous one, namely group and node addresses are given by L and H

respectively. Then a parallel prefix (summation) algorithm is applied to each group, and with $2\log(n+1)-1$ routing and $\log(n+1)$ arithmetic steps this is achieved. In the end, the k^{th} DD is found in node address $[H\ L]$ where $H = 10\cdots0 = G(11\cdots1)$ and $L = G(k)$.

Thus we see that by making use of the properties of Gray code and applying parallel prefix at appropriate subcubes in parallel, we compute all divided differences in $2\log(n+1)+2$ arithmetic and $4\log(n+1)-2$ routing steps. ●

The hypercube implementation proposed here requires at most twice the number of routing instructions as there are arithmetic steps in the parallel prefix calculation section of the code. This appears to be is the best possible, given the assumptions of a $2r$ dimensional hypercube and $n+1=2^r$ data points. This approach was tested by means of an implementation on the Intel iPSC hypercube system. Nevertheless, that particular implementation is not well suited for this computation as the communication time far exceeds the floating-point computation time. Moreover, an SIMD hypercube with appropriate processor masking capabilities would be enough. Some recent machines, the Thinking Machines Connection Machine for example, implement parallel prefix as a library routine making the implementation of the parallel Newton interpolation algorithm very easy.

2.9. ERROR ANALYSIS

It is quite natural to expect that the fastest algorithms in terms of parallel computational complexity will behave worse or at best as well as the best serial algorithms from the point of view of numerical accuracy. Hence showing for a new parallel algorithm that it is as stable as the stable serial algorithms is highly desirable. Analysis as well as experimental evidence

show that the parallel Newton polynomial interpolation algorithm has error properties similar to the ones of the traditional serial algorithms,.

The papers by [Brue84], [Macl82] present analytical and experimental results for some of the classical algorithms of interpolation. Nowhere in the literature however is there an attempt to analyze the numerical error in the fast serial and parallel interpolation algorithms. Only [Horo72] contains a negative comment regarding the numerical properties of the fast interpolation algorithms which use fast convolution and FFT.

It will shown that the parallel Newton interpolation algorithm is very suitable for a parallel processing environment by not only being the fastest algorithm available but most important with numerical properties similar to those of the widely used serial algorithms.

The principle behind forward error analysis consists of the analysis of the consequences of perturbed input data and rounding errors at every arithmetic operation referred to the floating-point result. The methodology has been developed in [Stum80], [Stum81] and [Rons84]. The method was also followed in [Gao_87] to demonstrate the stability of pipe-lined recurrence solvers.

The final result of an arithmetic expression produced using floating point arithmetic is affected by a relative (or absolute) error which can be traced back to two sources:

(i) the error caused by perturbed input data

(ii) the error caused by rounding errors at each arithmetic operation.

The machine floating point arithmetic is required to satisfy the following conditions:

(i) The exponent range is unlimited in both directions.

(ii) The relative input error α_{in} , caused by conversion of α to an N digit floating point

number α', is bounded by

$$\alpha_{in} = \left| \frac{\alpha - \alpha'}{\alpha} \right| \leq \varepsilon$$

where $\varepsilon = 0.5\, b^{-N+1}$ if symmetric rounding is used, and $\varepsilon = b^{-N+1}$ if the mantissa is chopped to N digits. Here b denotes the basis of the number representation.

(iii) For each of the arithmetic operations $* \in \{+, -, \times, /\}$ the relative error is bounded by

$$\left| \frac{\alpha \circ \beta - \alpha * \beta}{\alpha * \beta} \right| \leq \varepsilon$$

for all machine numbers satisfying $\alpha * \beta \neq 0$. Here $\circ$ denotes the corresponding machine operation of $*$ (i.e. $\alpha \circ \beta$ is again a machine number).

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 readily 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 less than another representation of an arithmetic circuit as defined in Section 2.2 Each of the graph's arcs is labeled with *relative error magnification factor* (or *absolute error magnification factor*) as given in Figure 2.3.

It is possible to determine the influence of data and rounding errors on the final floating-point results by attaching magnification factors to every arc of the computational graph. Although in general an algorithm will have many outputs (e.g. the $n+1$ DD 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. Let each of the different paths from the node ζ (input or operational node except the output node) to the output node be labeled from an index set J_ζ. For every 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. For the relative case define the sum of these products

$$S_\zeta^{rel} = \sum_{j \in J_\zeta} P_j \quad .$$

and S_ζ^{abs} is defined similarly using absolute magnification factors. For the output node $S_{output}^{rel} = S_{output}^{abs} = 1$. Now we define four different condition numbers:

$$\rho_R = \sum_{\zeta \text{ operational node}} |S_\zeta| \qquad \text{relative rounding condition number} \tag{2.2}$$

$$\rho_D = \sum_{\zeta \text{ input node}} |S_\zeta| \qquad \text{relative data condition number} \tag{2.3}$$

$$\sigma_R = \sum_{\zeta \text{ operational node}} |S_\zeta| \, |\text{value}(\zeta)| \qquad \text{absolute rounding condition number} \tag{2.4}$$

$$\sigma_D = \sum_{\zeta \text{ data node}} |S_\zeta| \, |\text{value}(\zeta)| \qquad \text{absolute data condition number} \tag{2.5}$$

Using the relative and absolute condition numbers one obtains a priori estimates for the relative and absolute error, respectively:

$$F^{rel} \leq (\rho_D + \rho_R)\,\varepsilon + O(\varepsilon^2) \quad ,$$
$$F^{abs} \leq (\sigma_D + \sigma_R)\,\varepsilon + O(\varepsilon^2) \quad .$$

An algorithm for the evaluation of an expression is called *numerically stable* if and only if there exists a positive A dependent on the structure of the algorithm such that for all input data for which ρ_R and ρ_D (or σ_R and σ_D respectively we have

$$\rho_R \leq A\,\rho_D \quad (\,\sigma_R \leq A\,\sigma_D\,) \, .$$

The factor A can be taken as a measure of the stability of the algorithm. In [Rons84], an algorithm is classified as *unconditionally* stable if A is independent of the input

variables, n, while it is *conditionally* stable if A depends on n. An even finer classification was recently proposed in [Gao_87], by observing how A varies with n. We say that an algorithm is stable of order $f(n)$ if $A(n) = O(f(n))$.

We give the forward error analysis for the parallel Newton interpolation algorithm and demonstrate its stability.

Theorem 2.6

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:

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

and

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

where

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

Proof :

The computational graph of the parallel Newton interpolation algorithm is given in Figure 2.4 for the value of $k = 3$. The parallel prefix computations considered in this graph correspond to a larger than minimum size circuit (of [KrRS85] which is of size $O(n \log n)$) it is easy to show that the same discussion applies to the better circuits of [LaFi80], [BiGa86], [Snir86], and [LaYD876]. 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 for the corresponding operation at the node 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 to the output node. Out of these, k paths correspond to x_i 's contribution in $x_i - x_j$ for $j = 0, 1,, \cdots, k$ and $j \neq i$. Each contribution can be seen to be:

$$-\frac{f_i}{\phi_i^{(k)}(x_i)} \frac{1}{x_i - x_j} \quad j = 0, 1, \cdots, i\text{–}1, i\text{+}1, \cdots, k. \tag{2.6}$$

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

$$\frac{f_j}{\phi_j^{(k)}(x_j)} \frac{1}{x_j - x_i} \quad j = 0, 1, \cdots, i\text{–}1, i\text{+}1, \cdots, k. \tag{2.7}$$

Hence from (2.6-7), for a single x_i , the sum of products of the magnification factors of all paths to the output is equal to

$$S_{x_i} = -\sum_{\substack{j=0 \\ j \neq i}}^{k} \frac{f_i}{\phi_i^{(k)}(x_i)} \frac{1}{x_i - x_j} + \sum_{\substack{j=0 \\ j \neq i}}^{k} \frac{f_j}{\phi_j^{(k)}(x_j)} \frac{1}{x_j - x_i} \quad . \tag{2.8}$$

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

$$S_{f_i} = \frac{1}{\phi_i^{(k)}(x_i)} \tag{2.9}$$

From the definition (2.5):

$$\sigma_{k,D} = \sum_{i=0}^{k} \left[|S_{f_i}||f_i| + |S_{x_i}||x_i| \right]$$

hence multiplying each of the terms in (2.8-9) 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 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 consist of a single term, namely

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

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

$$S_{(-)} = -\frac{f_i}{\phi_i^{(k)}(x_i)}\,\frac{1}{x_i - x_j}$$

which leads to

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

$$= k \sum_{i=0}^{k} \frac{|f_i|}{|\phi_i^{(k)}(x_i)|} \tag{2.10}$$

and similarly

$$\sum_{\text{all } (\times) \text{ nodes}} |S_{(\times)}|\,|\text{value}(\,(\times)\text{ node}\,)| = (k-1) \sum_{i=0}^{k} \frac{|f_i|}{|\phi_i^{(k)}(x_i)|} \tag{2.11}$$

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

$$\sum_{\text{all } (/) \text{ nodes}} |S_{(/)}|\,|\text{value}(\,(/)\text{ node}\,)| = \sum_{i=0}^{k} \frac{|f_i|}{|\phi_i^{(k)}(x_i)|} \tag{2.12}$$

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

$$\sum_{\substack{\text{all (+) nodes} \\ \text{at level t}}} |S_{(+)}| \, |\text{value ((+) node)}| \leq \sum_{i=0}^{k} \frac{|f_i|}{|\phi_i^{(k)}(x_i)|}$$

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

$$\sum_{\text{all (+) nodes}} |S_{(+)}| \, |\text{value [(+) node]}| \leq \lceil \log(k+1) \rceil \sum_{i=0}^{k} \frac{|f_i|}{|\phi_i^{(k)}(x_i)|} \qquad (2.13)$$

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

$$\begin{aligned}
\sum_{\text{all (+) nodes}} |S_{(+)}| \, |\text{value [(+) node]}| = {} & \left|\frac{f_0}{\phi_0^{(k)}(x_0)} + \frac{f_1}{\phi_1^{(k)}(x_1)}\right| + \left|\frac{f_2}{\phi_2^{(k)}(x_2)} + \frac{f_3}{\phi_3^{(k)}(x_3)}\right| + \cdots + \\
& + \left|\frac{f_{k-1}}{\phi_{k-1}^{(k)}(x_{k-1})} + \frac{f_k}{\phi_k^{(k)}(x_k)}\right| + \left|\frac{f_0}{\phi_0^{(k)}(x_0)} + \frac{f_1}{\phi_1^{(k)}(x_1)} + \frac{f_2}{\phi_2^{(k)}(x_2)} + \frac{f_3}{\phi_3^{(k)}(x_3)}\right| + \cdots + \\
& + \left|\frac{f_{k-3}}{\phi_{k-3}^{(k)}(x_{k-3})} + \frac{f_{k-2}}{\phi_{k-2}^{(k)}(x_{k-2})} + \frac{f_{k-1}}{\phi_{k-1}^{(k)}(x_{k-1})} + \frac{f_k}{\phi_k^{(k)}(x_k)}\right| + \cdots \cdots + \\
& + \left|\frac{f_0}{\phi_0^{(k)}(x_0)} + \frac{f_1}{\phi_1^{(k)}(x_1)} + \frac{f_2}{\phi_2^{(k)}(x_2)} + \frac{f_3}{\phi_3^{(k)}(x_3)} + \cdots + \frac{f_k}{\phi_k^{(k)}(x_k)}\right|
\end{aligned}$$

Adding the terms in (2.10), (2.11), (2.12) and (2.13)

$$\begin{aligned}
\sigma_{k,R} = {} & \sum_{\text{all (–) nodes}} |S_{(-)}| \, |\text{value ((–) node)}| + \sum_{\text{all (×) nodes}} |S_{(\times)}| \, |\text{value ((×) node)}| + \\
& \sum_{\text{all (/) nodes}} |S_{(/)}| \, |\text{value ((/) node)}| + \sum_{\text{all (+) nodes}} |S_{(+)}| \, |\text{value ((+) node)}|
\end{aligned}$$

and thus

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

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

Theorem 2.7

The parallel Newton algorithm is numerically stable.

Proof :

From Theorem 2.6 it immediately follows that

$$\sigma_{k,D} \geq \sum_{i=0}^{k} \frac{|f_i|}{|\phi_i^{(k)}(x_i)|}$$

Comparing $\sigma_{k,D}$ with $\sigma_{k,R}$

$$\sigma_{k,R} \leq A(k) \cdot \sigma_{k,D}$$

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

Using the classification proposed by Gao in [Gao_87], we observe that $A(k) = O(k)$, thus the parallel Newton interpolation is *linearly* stable.

We emphasize that what we have shown is the numerical stability of the parallel *algorithm*. It does not salvage the overall numerical behavior of the polynomial interpolation *problem*, which can be badly conditioned when n is large.

2.10. NUMERICAL EXPERIMENTS

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

The first experiment is produced as follows: First we choose points x_k and coefficients c_k to define a Newton polynomial

$$\sum_{k=0}^{n} c_k \cdot (x-x_0) \cdots (x-x_{k-1})$$

and use Horner's algorithm to evaluate it at $x_k = -1 + \frac{2k}{(n-1)}$ for $k = 0, \cdots, n-1$ in double

precision. These values are in turn input to each of the interpolation algorithms (Aitken, Neville and parallel Newton) from which approximations $c_k^{(computed)}$ for the DDs 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 $|c_k^{(computed)} - c_k|$ is then a measure of the error for the corresponding algorithm. Figure 2.6a corresponds to $n = 16$ and coefficients $c_k = \frac{(-1)^k}{k+1}$. It depicts $\log_{10}|c_k - c_k^{(computed)}|$ as a function of k. Since c_0 is available directly, results are shown only for $k > 0$. The same is repeated in Figures 2.6b, 2.6c, 2.6d, 2.6e, and 2.6f for the divided difference coefficients $c_k = \frac{1}{k+1}$, $c_k = \frac{(-1)^k}{(k+1)^2}$, $c_k = \frac{1}{(k+1)^2}$, $c_k = (-1)^k\,(k+1)$, and $c_k = (k+1)$, respectively. From these examples, 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.

Figures 2.7 and 2.8 show the effectiveness of polynomial interpolation to approximate a given known function. As inputs to each algorithm, we use the values of a known function at $n + 1$ points computed in double precision. The rest of the computation is in single precision. Interpolating polynomials are then produced by means of each of the i) Newton based algorithms (parallel, Aitken, Neville), and ii) the parallel algorithm of [Reif86] which utilizes the Lagrange representation and FFT. The interpolating polynomials are evaluated 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 poly-

nomials constructed using the sequential algorithms are evaluated by means of Horner's scheme, whereas those constructed using the parallel algorithms are evaluated as described in Section 2.6. Figures (2.7a-c) show

$$\log_{10}|f(x_i+\frac{h}{2})-P_n(x_i+\frac{h}{2})|$$

as a function of x where $f(x)=\log_e(1+x)$. The interpolation nodes are $x_i=i\,h$ for $0\le i\le n-1$, $h=\frac{1}{(n-1)}$ and $n=8, 16, 32$. For ease of representation, any differences smaller than 10^{-20} were set equal to that value. Figures (2.8a-c) show the same experiment for the function $f(x)=\frac{1}{1+x^2}$ in the interval $[-5, 5]$. This function was used as an example by Runge [Davi75] for which the interpolating polynomial diverges from f outside the interval $|x|\le 3.63..$ 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 $x_i=-5+i\,h$ for $i=0, 2, .., n-2$ and $x_i=i\,h$ for $i=1, 3, .., n-1$ with $h=\frac{5}{n-1}$ (n is assumed to be even). It is easy to see from the definition that Newton interpolation is sensitive to the order of the interpolation nodes. The above interleaved order was chosen to reduce the error. Lagrange interpolation on the other hand is mostly independent of the order. An additional benefit in this interleaved ordering is that for $n=32$ and single precision arithmetic, the evaluation of the denominators in parallel Newton is less likely to produce values close to the underflow region.

The experiments indicate that the parallel Newton algorithm is almost as good as the serial algorithms, although as n grows all algorithms demonstrate the expected weakness of equidistant polynomial interpolation. The experiments also indicate that 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 [Reif86]. Nevertheless, a rigorous analysis of the FFT based interpolation algorithms remains to be done.

2.11. CONCLUSIONS

The presented parallel algorithm constructs the interpolating polynomial in its Newton form by fast evaluation of divided differences through 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 $O(n^2)$. The computation of the divided differences using this algorithm is shown to be numerically stable.

The parallel computational complexity is an improvement over other parallel implementations of the Neville and Aitken algorithms and the FFT based, Lagrange interpolation algorithms described in the literature. No preconditioning of the data is assumed and the algorithm does not require equidistant interpolation points. As it will be seen in the next chapter the techniques are applicable to the Hermite interpolation as well. In addition, the interpolating polynomial in its Newton form can be evaluated by means of a fast algorithm having the same characteristics. A natural consequence is circuit similarity in VLSI implementation.

It is worth noting that the evaluation of the products may cause overflow or underflow. Therefore, some care may be required in the implementation.

APPENDIX 1. PARALLEL INTERPOLATION ON A HYPERCUBE

The example is for the interpolation of $n+1=4$ points on a hypercube of $(n+1)^2=16$ processors. Initially partition the nodes of the cube into $n+1=4$ subcubes of $n+1=4$ nodes each. Figure 2.5 shows the labeled cube nodes, connected in groups of four, dashed lines delineating the first grouping, continuous lines for the second grouping.

Let each node be represented by the $2r$ bits $[H\ L]$. Let G denote the Gray code map defined on $\log(n+1)=r=2$ bit natural numbers. It is easy to verify that $G(0)=0, G(1)=1, G(2)=3, G(3)=2$. The initial assumed distribution for each element is: $x_j, x_i, F_i \in node\ [G(i)\,G(j)]$.

Table 2.1 shows the initial distribution and the application of Step 1. The application of the parallel prefix (product) algorithm (Step 2) and the parallel division (Step 3) result in the distributions shown in Table 2.2. Finally, applying the parallel prefix (summation) algorithm in each subcube, consisting of those nodes with their r least significant bits identical, and grouping in that manner, one obtains the results depicted in Table 2.3.

As expected, the results $f_0, f_{01}, f_{012}, f_{0123}$ are found in nodes 1000, 1001, 1011 and 1010 respectively.

Table 2.1. Data distribution in the CCC nodes for Steps 0 and 1.

[H L]	Step 0	Step 1
0000	x_0, x_0, f_0	$1, f_0$
0001	x_1, x_0, f_0	y_{01}, f_0
0011	x_2, x_0, f_0	y_{02}, f_0
0010	x_3, x_0, f_0	y_{03}, f_0
0110	x_3, x_1, f_1	y_{13}, f_1
0111	x_2, x_1, f_1	y_{12}, f_1
0101	x_1, x_1, f_1	$1, f_1$
0100	$x_0, x_1, 0$	$y_{10}, 0$
1100	$x_0, x_2, 0$	$y_{20}, 0$
1101	$x_1, x_2, 0$	$y_{21}, 0$
1111	x_2, x_2, f_2	$1, f_2$
1110	x_3, x_2, f_2	y_{23}, f_2
1010	x_3, x_3, f_3	$1, f_3$
1011	$x_2, x_3, 0$	$y_{32}, 0$
1001	$x_1, x_3, 0$	$y_{31}, 0$
1000	$x_0, x_3, 0$	$y_{30}, 0$

Table 2.2. Data distribution in the CCC nodes for Steps 2 and 3.

[H L]	Step 2	Step 3
0000	$1, f_0$	f_0
0001	y_{01}, f_0	$\frac{f_0}{y_{01}}$
0011	$y_{01}y_{02}, f_0$	$\frac{f_0}{y_{01}y_{02}}$
0010	$y_{01}y_{02}y_{03}, f_0$	$\frac{f_0}{y_{01}y_{02}y_{03}}$
0110	$y_{10}y_{12}y_{13}, f_1$	$\frac{f_1}{y_{10}y_{12}y_{13}}$
0111	$y_{10}y_{12}, f_1$	$\frac{f_1}{y_{10}y_{12}}$
0101	y_{10}, f_1	$\frac{f_1}{y_{10}}$
0100	$y_{10}, 0$	0
1100	$y_{20}, 0$	0
1101	$y_{20}y_{21}, 0$	0
1111	$y_{20}y_{21}, f_2$	$\frac{f_2}{y_{20}y_{21}}$
1110	$y_{20}y_{21}y_{23}, f_2$	$\frac{f_2}{y_{20}y_{21}y_{23}}$
1010	$y_{30}y_{31}y_{32}, f_3$	$\frac{f_3}{y_{30}y_{31}y_{32}}$
1011	$y_{30}y_{31}y_{32}, 0$	0
1001	$y_{30}y_{31}, 0$	0
1000	$y_{30}, 0$	0

Table 2.3. Results in the CCC nodes after Step 4.

[H L]	Step 4
0000	f_0
0100	f_0
1100	f_0
1000	f_0
0001	$\frac{f_0}{y_{01}}$
0101	$f_{01} = \frac{f_0}{y_{01}} + \frac{f_1}{y_{10}}$
1101	$f_{01} = \frac{f_0}{y_{01}} + \frac{f_1}{y_{10}}$
1001	$f_{01} = \frac{f_0}{y_{01}} + \frac{f_1}{y_{10}}$
0011	$\frac{f_0}{y_{01}y_{02}}$
0111	$\frac{f_0}{y_{01}y_{02}} + \frac{f_1}{y_{10}y_{12}}$
1111	$f_{012} = \frac{f_0}{y_{01}y_{02}} + \frac{f_1}{y_{10}y_{12}} + \frac{f_2}{y_{20}y_{21}}$
1011	$f_{012} = \frac{f_0}{y_{01}y_{02}} + \frac{f_1}{y_{10}y_{12}} + \frac{f_2}{y_{20}y_{21}}$
0010	$\frac{f_0}{y_{01}y_{02}y_{03}}$
0110	$\frac{f_0}{y_{01}y_{02}y_{03}} + \frac{f_1}{y_{10}y_{12}y_{13}}$
1110	$\frac{f_0}{y_{01}y_{02}y_{03}} + \frac{f_1}{y_{10}y_{12}y_{13}} + \frac{f_2}{y_{20}y_{21}y_{23}}$
1010	$f_{0123} = \frac{f_0}{y_{01}y_{02}y_{03}} + \frac{f_1}{y_{10}y_{12}y_{13}} + \frac{f_2}{y_{20}y_{21}y_{23}} + \frac{f_3}{y_{30}y_{31}y_{32}}$

APPENDIX 2. FFT BASED PARALLEL INTERPOLATION ALGORITHM

For comparison purposes we describe the parallel interpolation algorithm based on a $O(\log n)$ depth circuit. For the proofs and details we refer the reader to the source [Reif86]. The algorithm utilizes the Lagrange representation and the Fast Fourier Transform. As before, we assume that the $n+1$ pairs (x_i, f_i) are given. The notation is the same as in Section 2.7. Let $\alpha_i \in R^{n+1}$ be the coefficient vectors for the monomials $x - x_i$ for $i = 0, 1, \cdots, n$. Hence $\alpha_i = [-x_i, 1, 0, \cdots, 0]^T$. We assume that $\alpha_i \in R^{n+1}$ as in Section 2.7. Let $\beta_i = DFT_n[\alpha_i]$, and let also the Hadamard product be denoted by $\circ$. The Lagrange interpolating polynomial is given by

$$P(x) = \sum_{i=0}^{n} \frac{\phi_i^{(n)}(x)}{\phi_i^{(n)}(x_i)} f_i \quad \text{where} \quad \phi_i^{(n)}(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} (x - x_j)$$

The algorithm is as follows:

Step 1. Compute the coefficients the polynomial

$$L(x) = (x - x_0)(x - x_1) \cdots (x - x_n) = \sum_{i=0}^{n} l_i x^i$$

using the convolution theorem as

$$[l_0, l_1, \cdots, l_n]^T = DFT_{n+1}^{-1}[\beta_0 \circ \beta_1 \circ \cdots \circ \beta_n]$$

where $\beta_i = DFT_{n+1}[\alpha_i]$ for $0 \le i \le n$.

Step 2. For all $0 \le i \le n$ compute

$$\phi_i^{(n)}(x) = \frac{L(x)}{x - x_i}.$$

Step 3. For all $0 \le i \le n$ evaluate $\phi_i^{(n)}(x_i)$.

Step 4. Multiply and add to obtain the interpolating polynomial

$$p_n(x) = \sum_{i=0}^{n} \frac{\phi_i^{(n)}(x)}{\phi_i^{(n)}(x_i)} f_i$$

Step 1 is the one making this algorithm competitive as opposed to the more straightforward $O(\log^2 n)$. With this technique, first all DFT_{n+1}'s are computed by means of some FFT algorithm resulting in $n+1$ vectors of dimension $n+1$ each. Next the $n+1$ vectors are multiplied componentwise and finally a fast inverse DFT_{n+1} is applied to the result vector of the previous operation. In Step 2 the division of $L(x)$ by $(x - x_i)$ induces a three term recursion on the coefficients of the polynomial $\phi_i^{(n)}(x)$. As is shown in [Reif86], the Lagrangian interpolation polynomial can be computed with the above algorithm in depth $O(\log n)$ and circuit size $O(n^2 \log n)$.

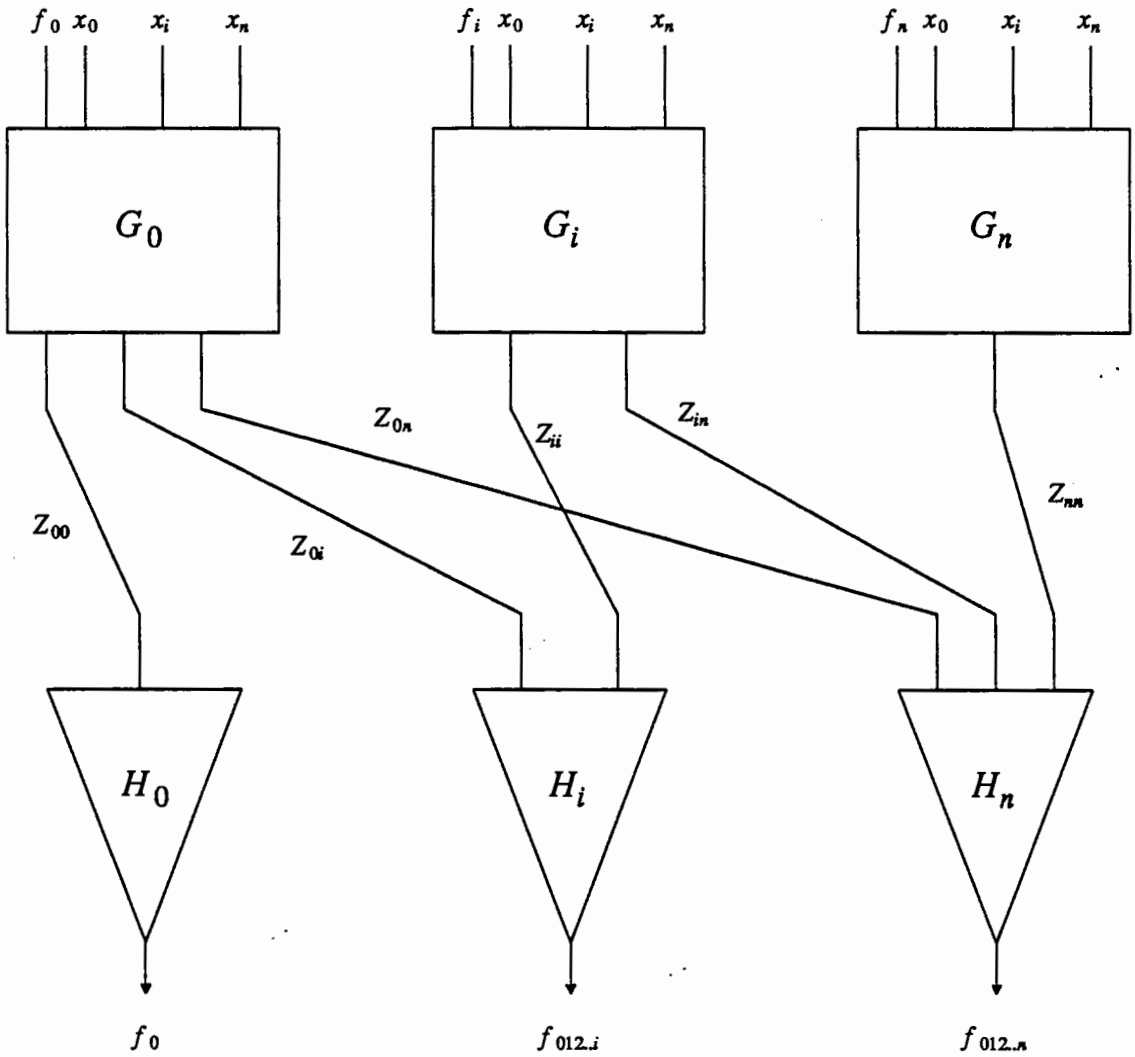


Figure 2.1. Parallel Newton interpolation circuit, here H_i is a binary tree of depth $\lceil \log i \rceil$ with H_0 being trivially empty.

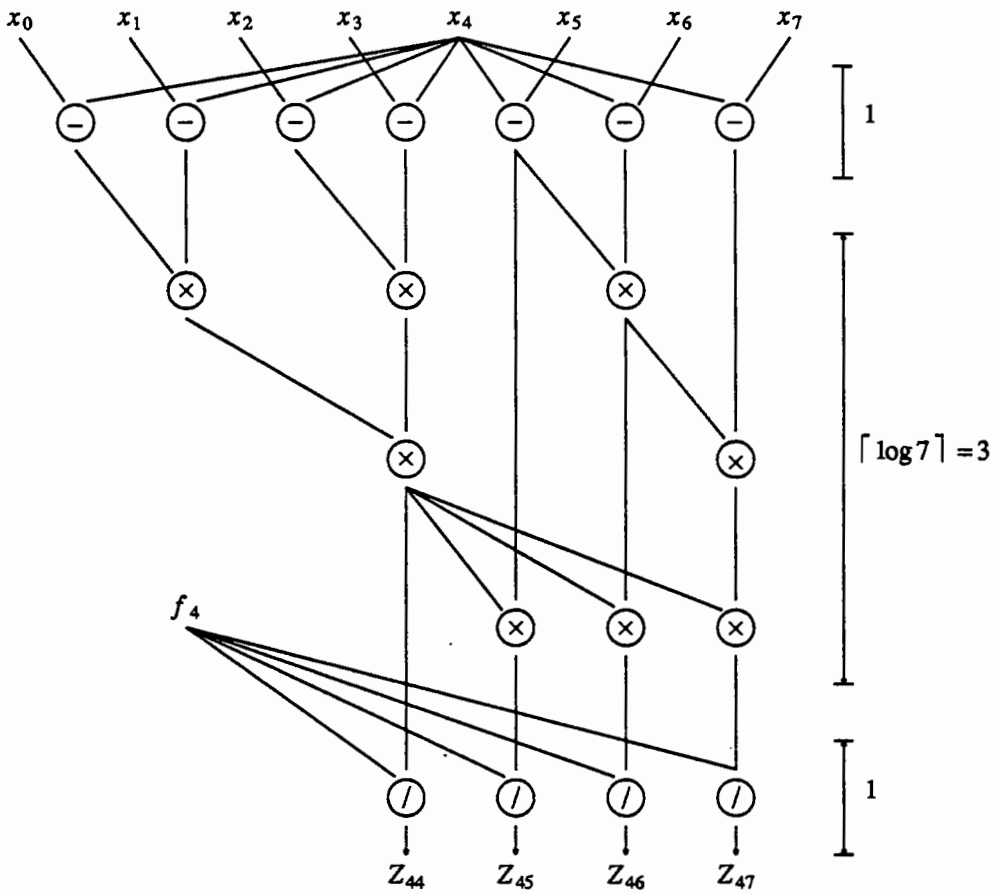
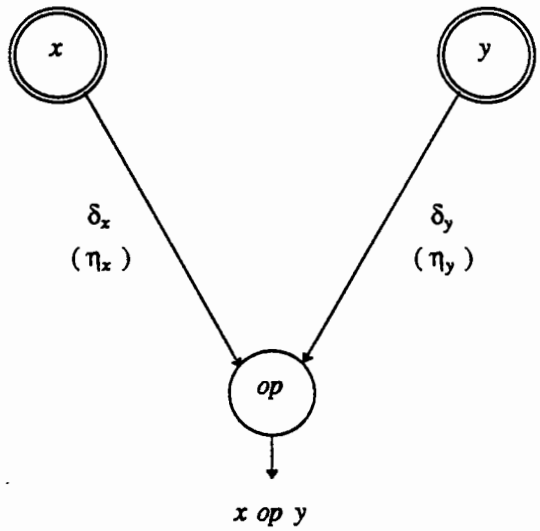


Figure 2.2. Expansion of G_4.



Operation	δ_x	δ_y	η_x	η_y
$x \times y$	y	x	1	1
$\frac{x}{y}$ †	$\frac{1}{y}$	$-\frac{x}{y^2}$	1	−1
$x+y$ †	1	1	$\frac{x}{x+y}$	$\frac{y}{x+y}$
$x-y$ †	1	−1	$\frac{x}{x-y}$	$\frac{-y}{x-y}$

Figure 2.3. Absolute (δ) and relative (η) error magnification factors.

† Assume $y \neq 0$, $x+y \neq 0$ and $x-y \neq 0$.

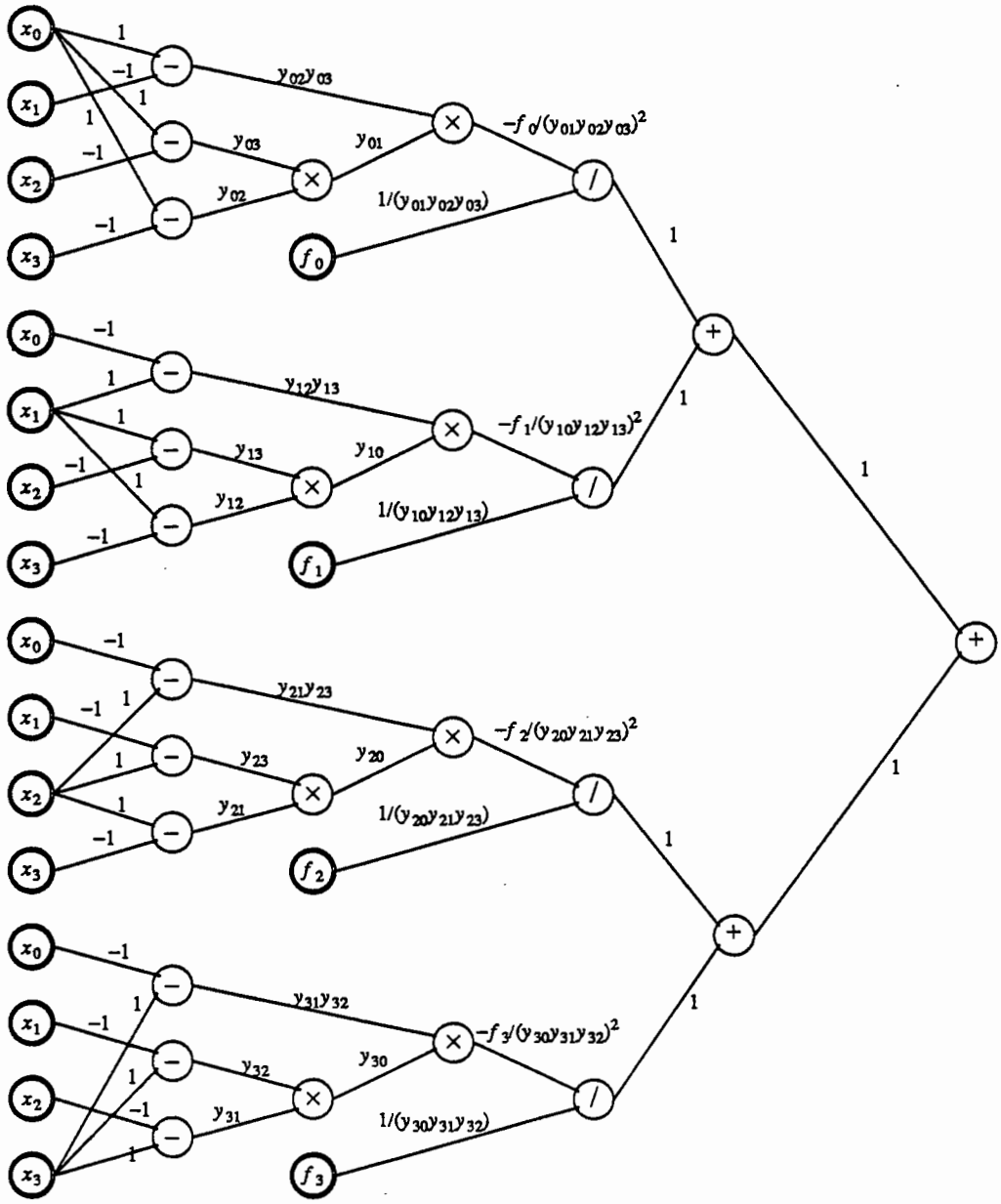


Figure 2.4. Absolute error magnifications factors of the parallel Newton interpolation algorithm for $n = 3$.

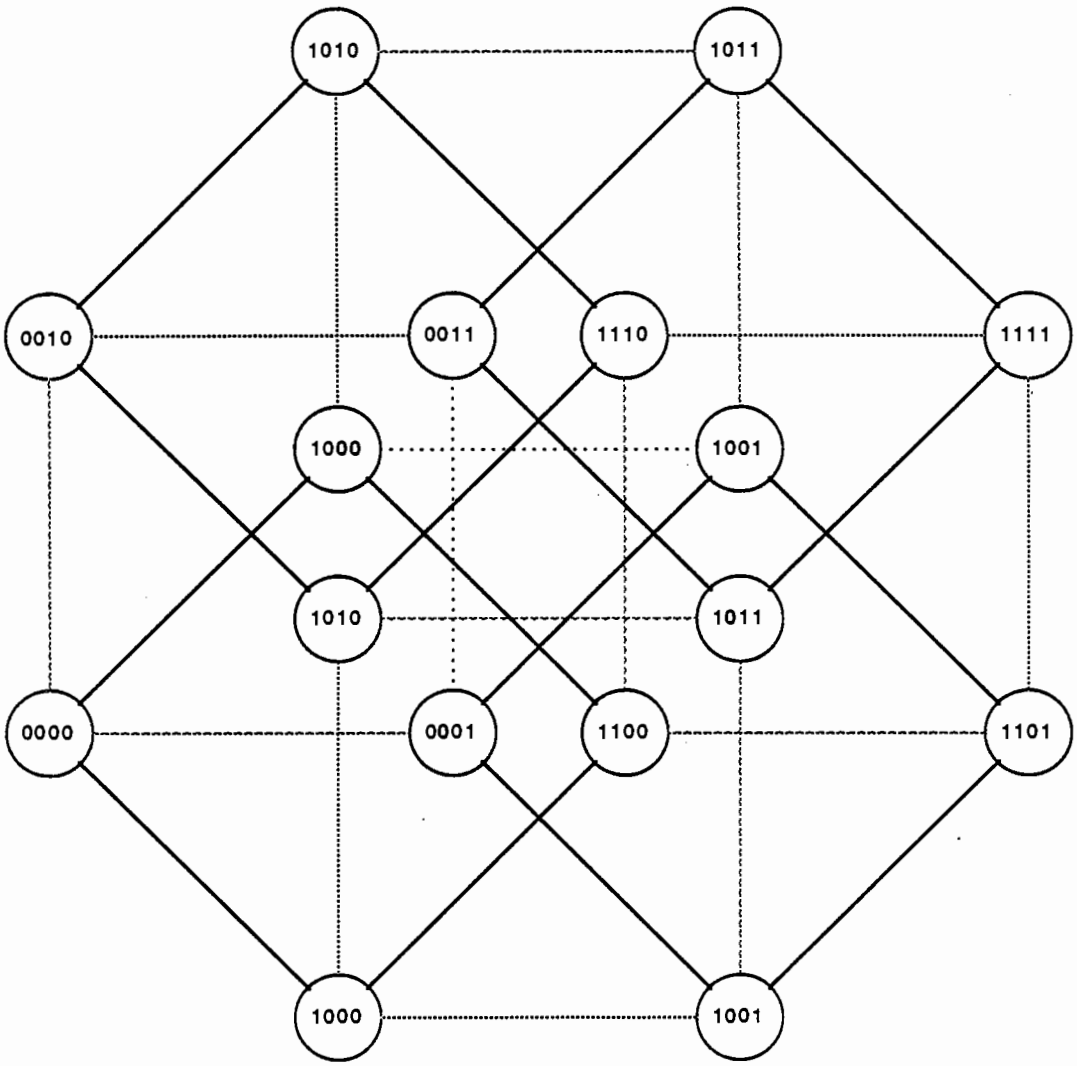


Figure 2.5. Sixteen node hypercube.

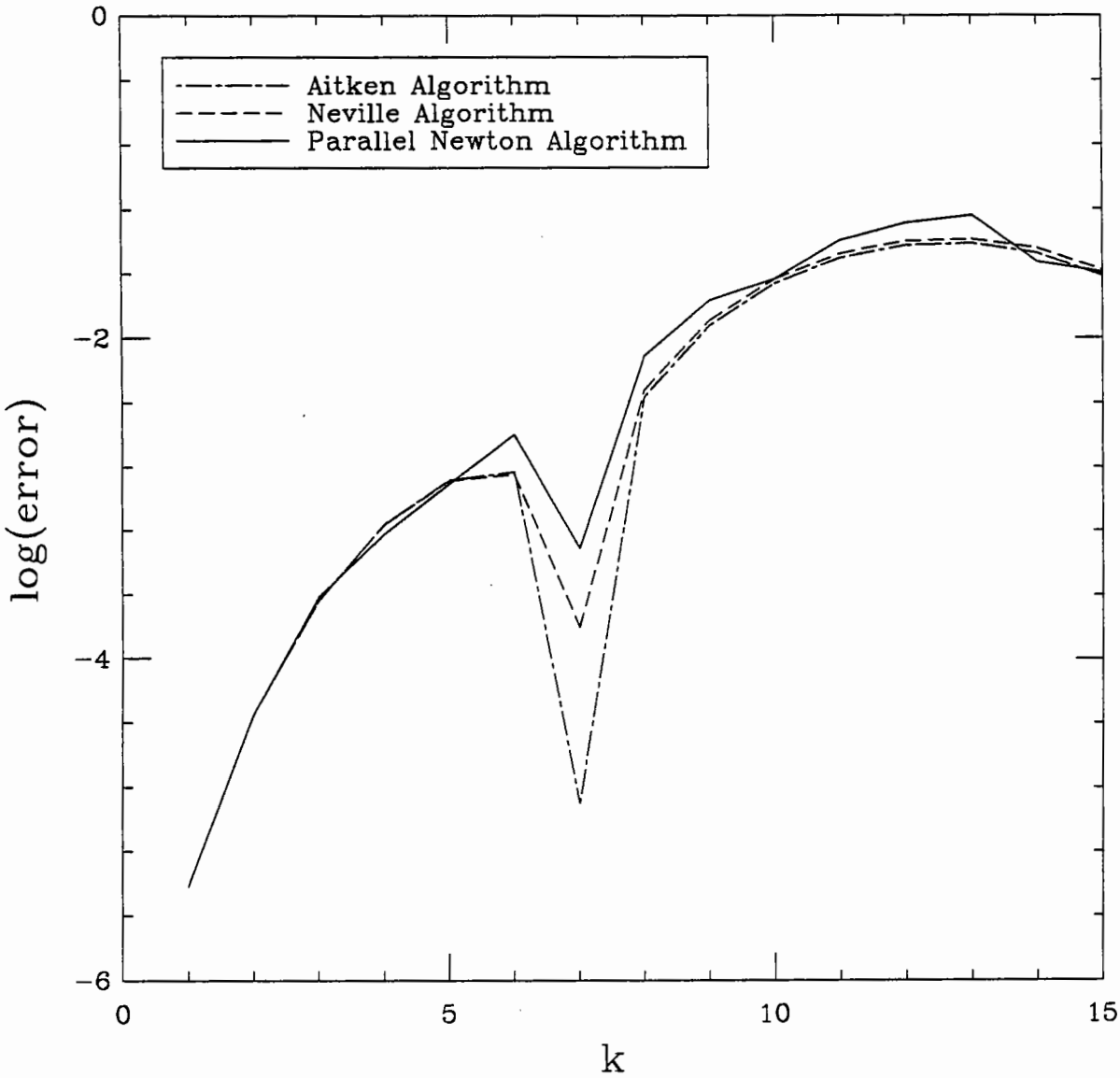


Figure 2.6a. $\log_{10} \left| c_k - c_k^{(computed)} \right|$ $k = 0, 1, \cdots, 15$ and $c_k = \frac{(-1)^k}{k+1}$.

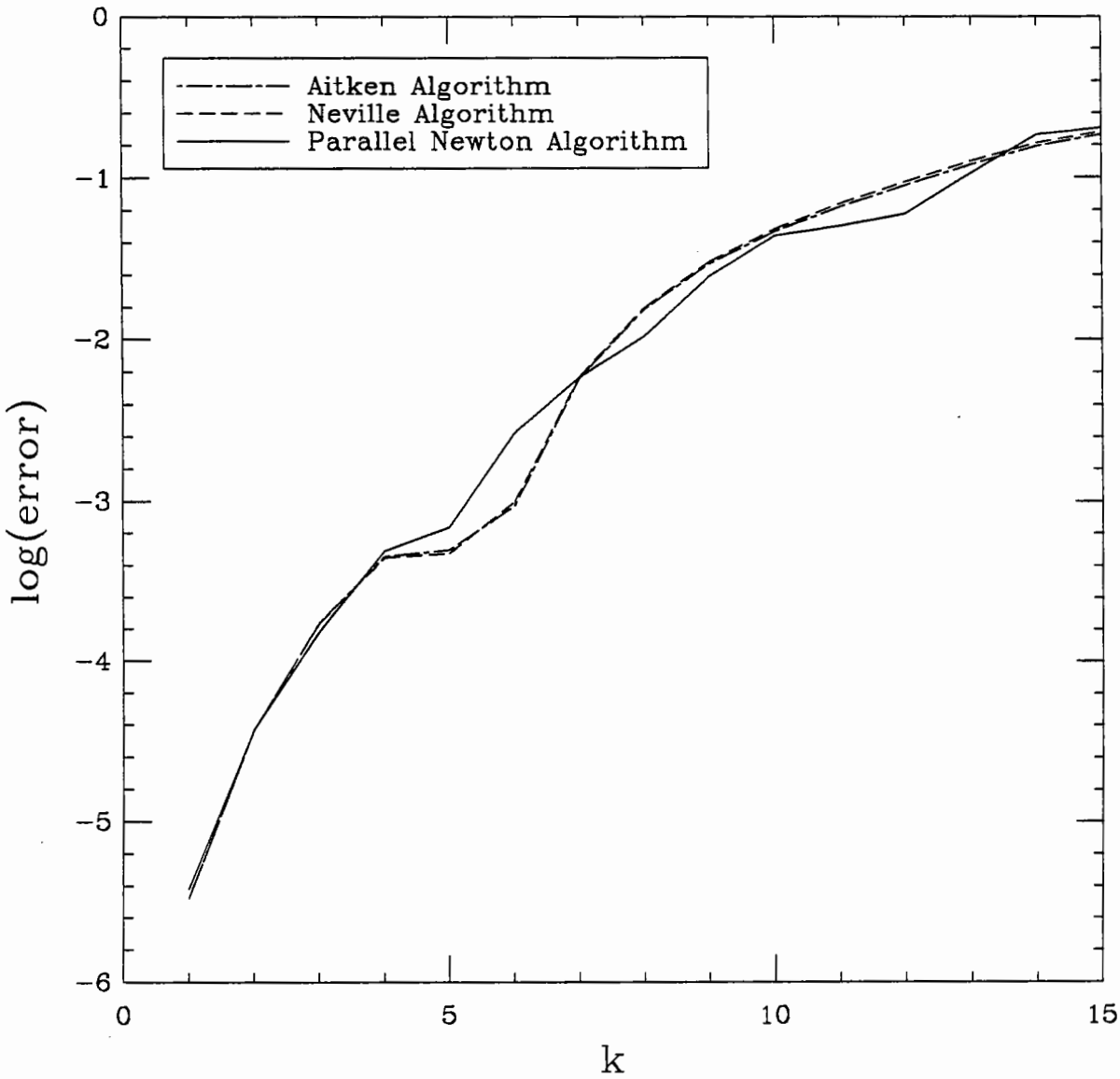


Figure 2.6b. $\log_{10} \left| c_k - c_k^{(computed)} \right|$ $k = 0, 1, \cdots, 15$ and $c_k = \frac{1}{k+1}$.

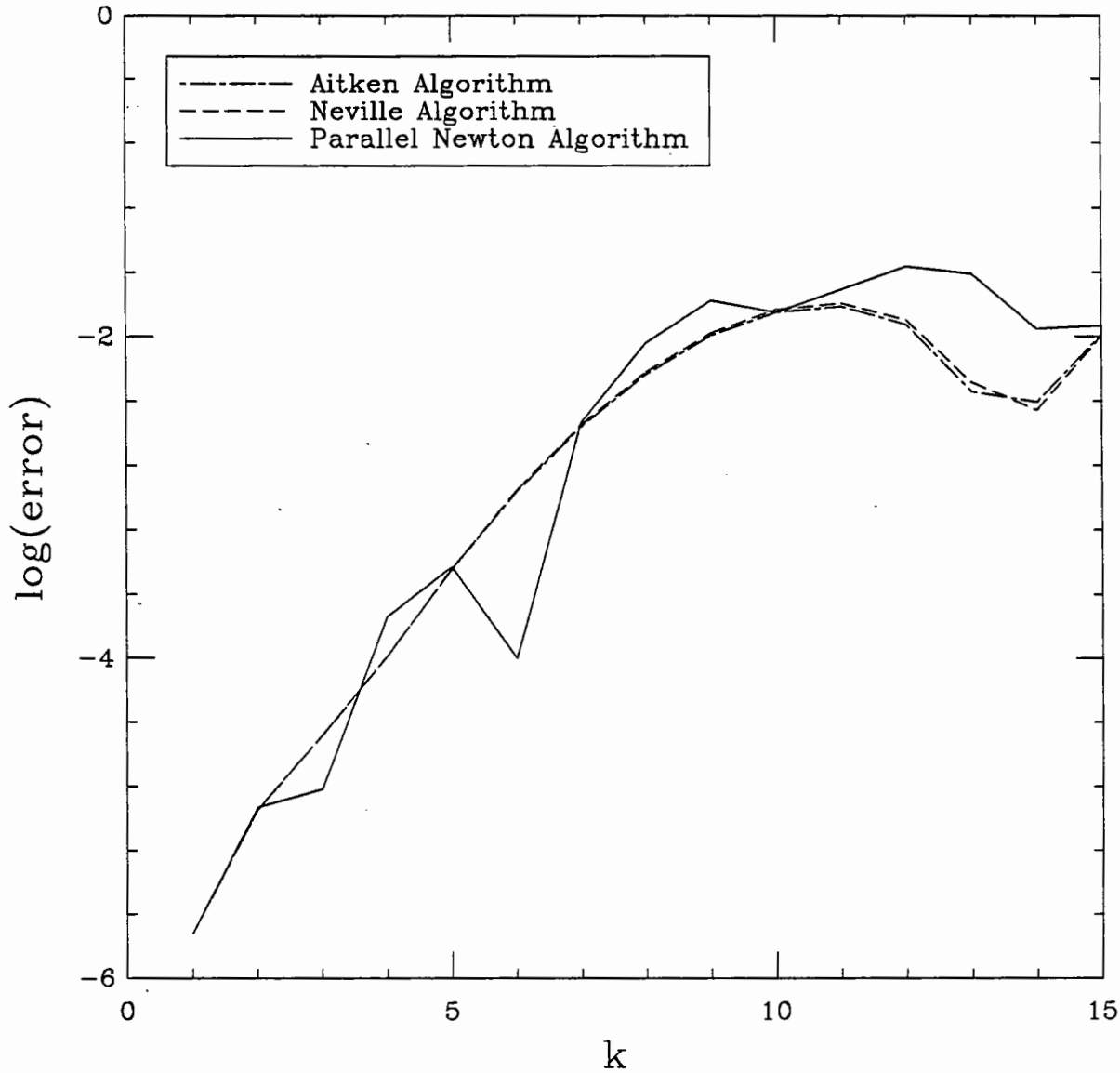


Figure 2.6c. $\log_{10} \left| c_k - c_k^{(computed)} \right|$ $k = 0, 1, \cdots, 15$ and $c_k = \dfrac{(-1)^k}{(k+1)^2}$.

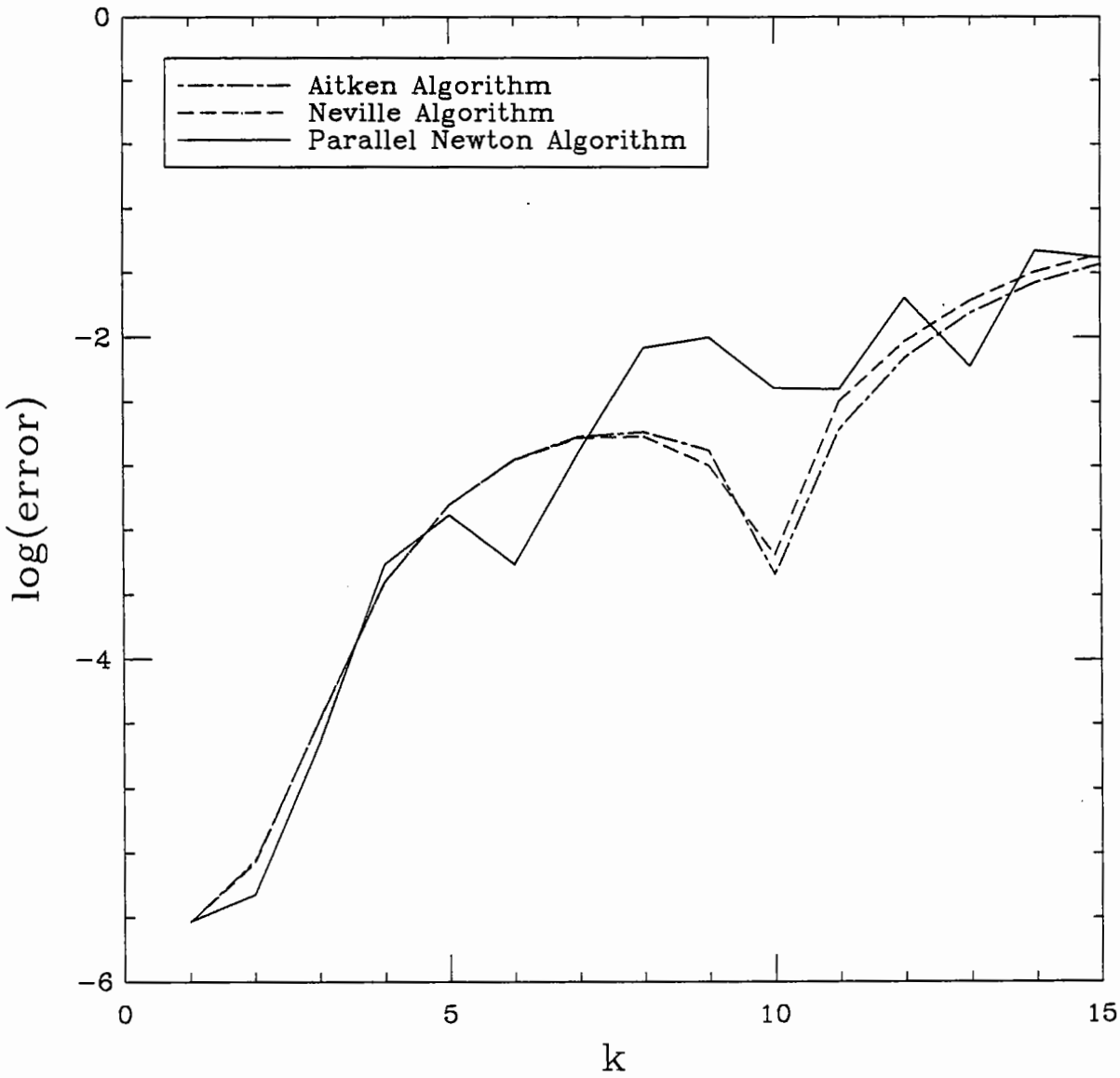


Figure 2.6d. $\log_{10} \left| c_k - c_k^{(computed)} \right| \quad k = 0, 1, \cdots, 15$ and $c_k = \frac{1}{(k+1)^2}$.

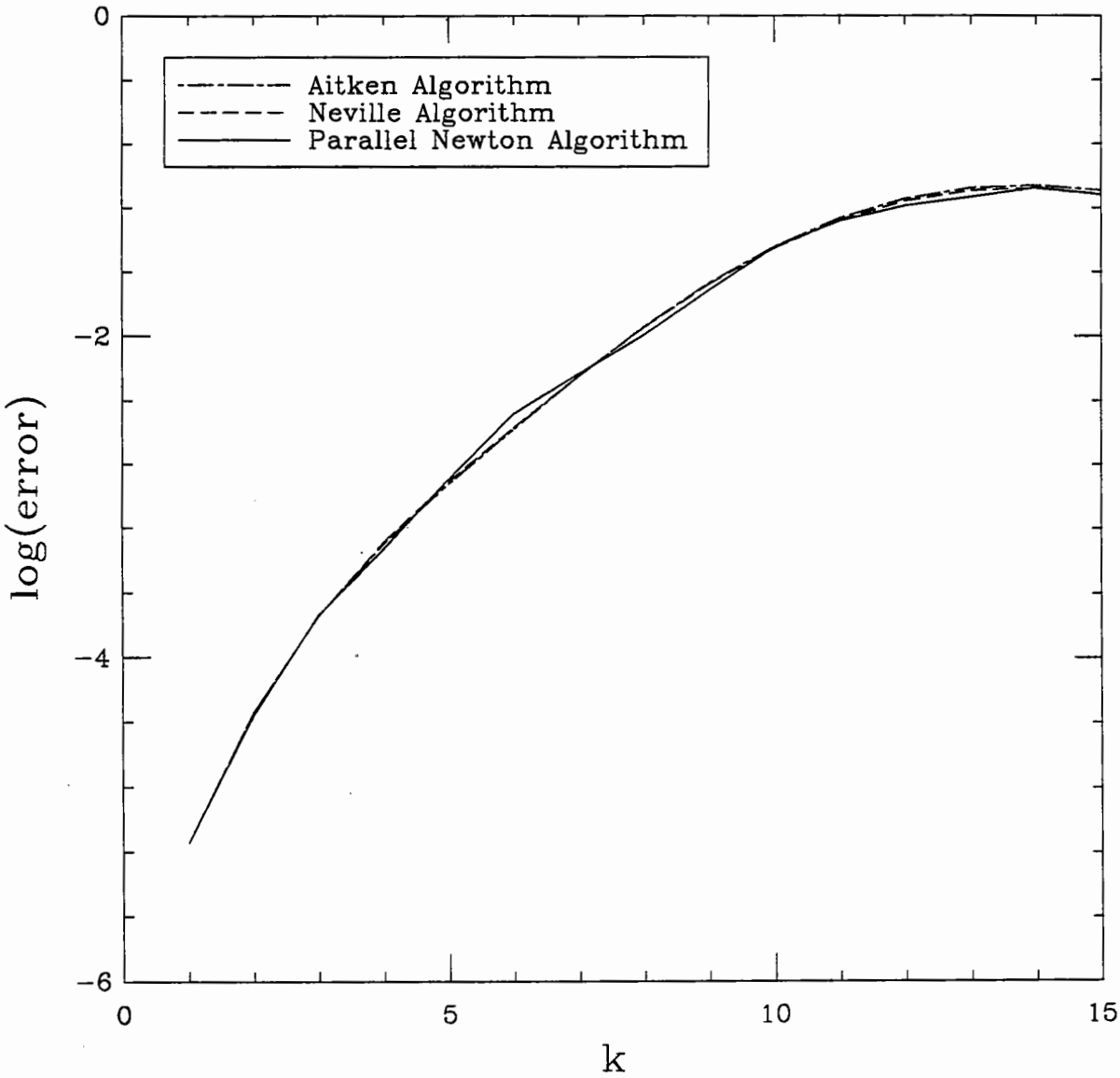


Figure 2.6e. $\log_{10} \left| c_k - c_k^{(computed)} \right|$ $k = 0, 1, \cdots, 15$ and $c_k = (-1)^k (k+1)$.

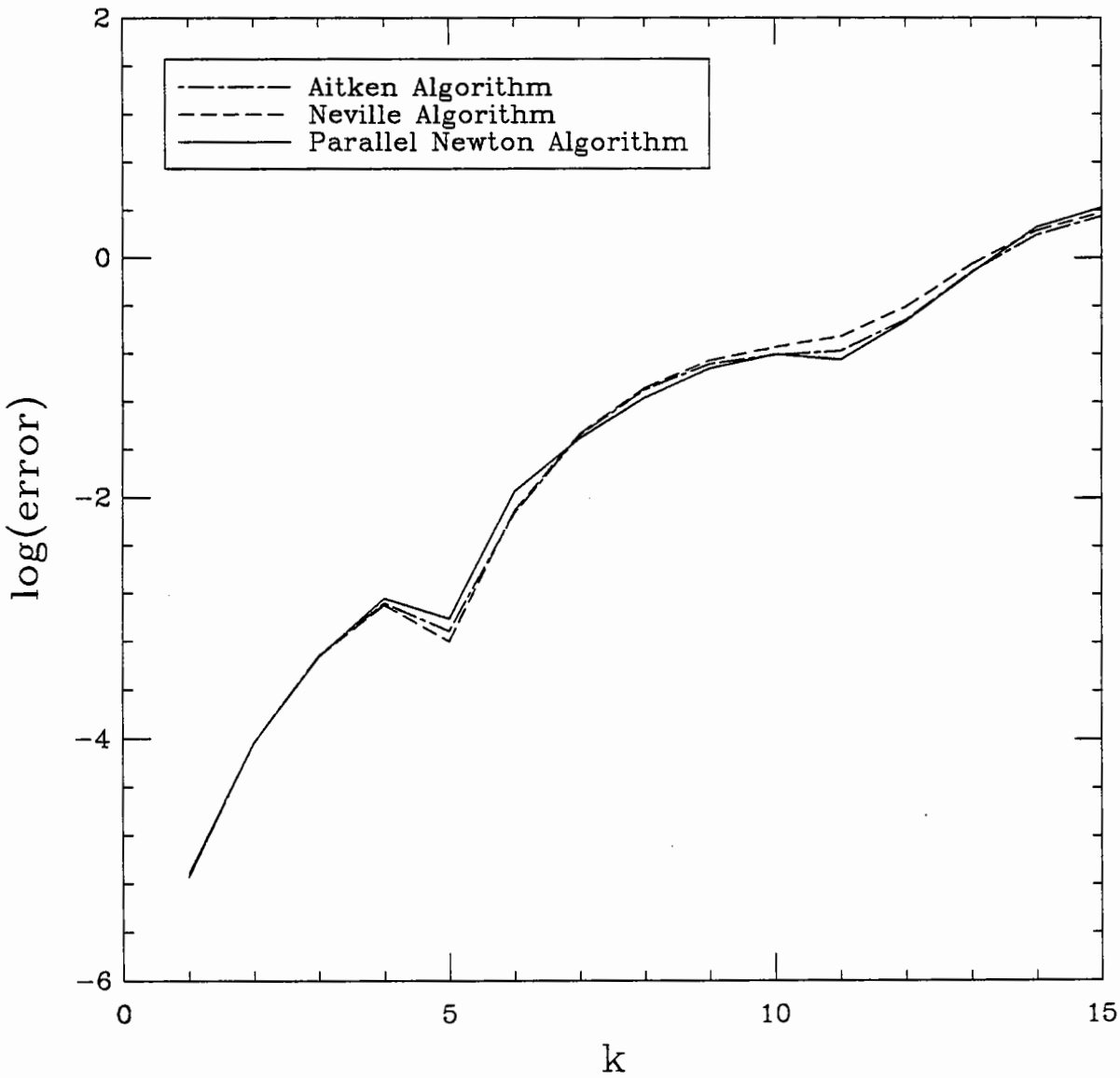


Figure 2.6f. $\log_{10} \left| c_k - c_k^{(computed)} \right|$ $k = 0, 1, \cdots, 15$ and $c_k = (k+1)$.

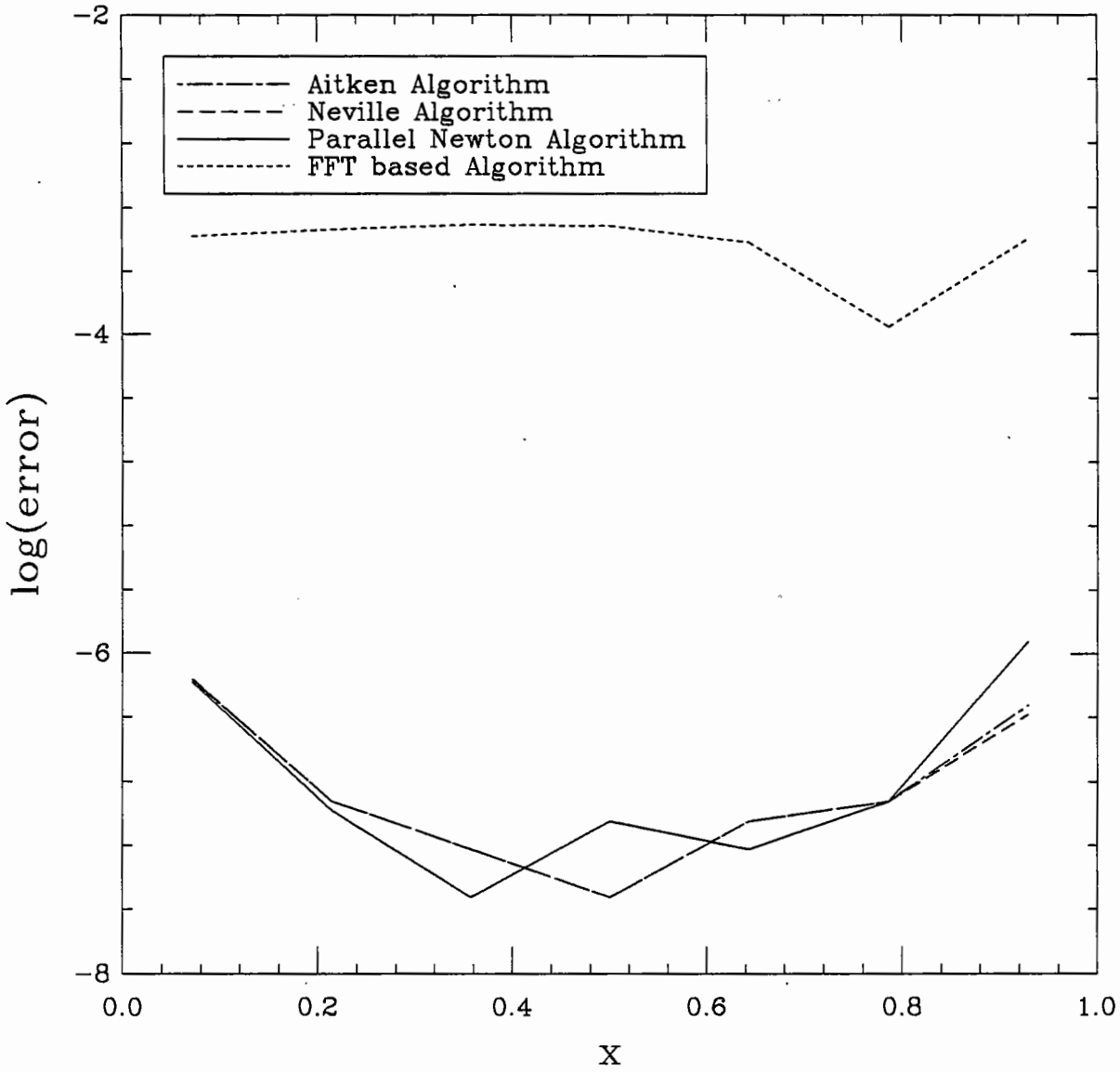


Figure 2.7a. $\log_{10} \left| \ln(1 + x_i + \frac{h}{2}) - P_7(x_i + \frac{h}{2}) \right|$, $i = 0, 1, \cdots, 6$, interpolation is based on $x_i = i\ h$ for $i = 0, 1, \cdots, 7$ and $h = \frac{1}{7}$.

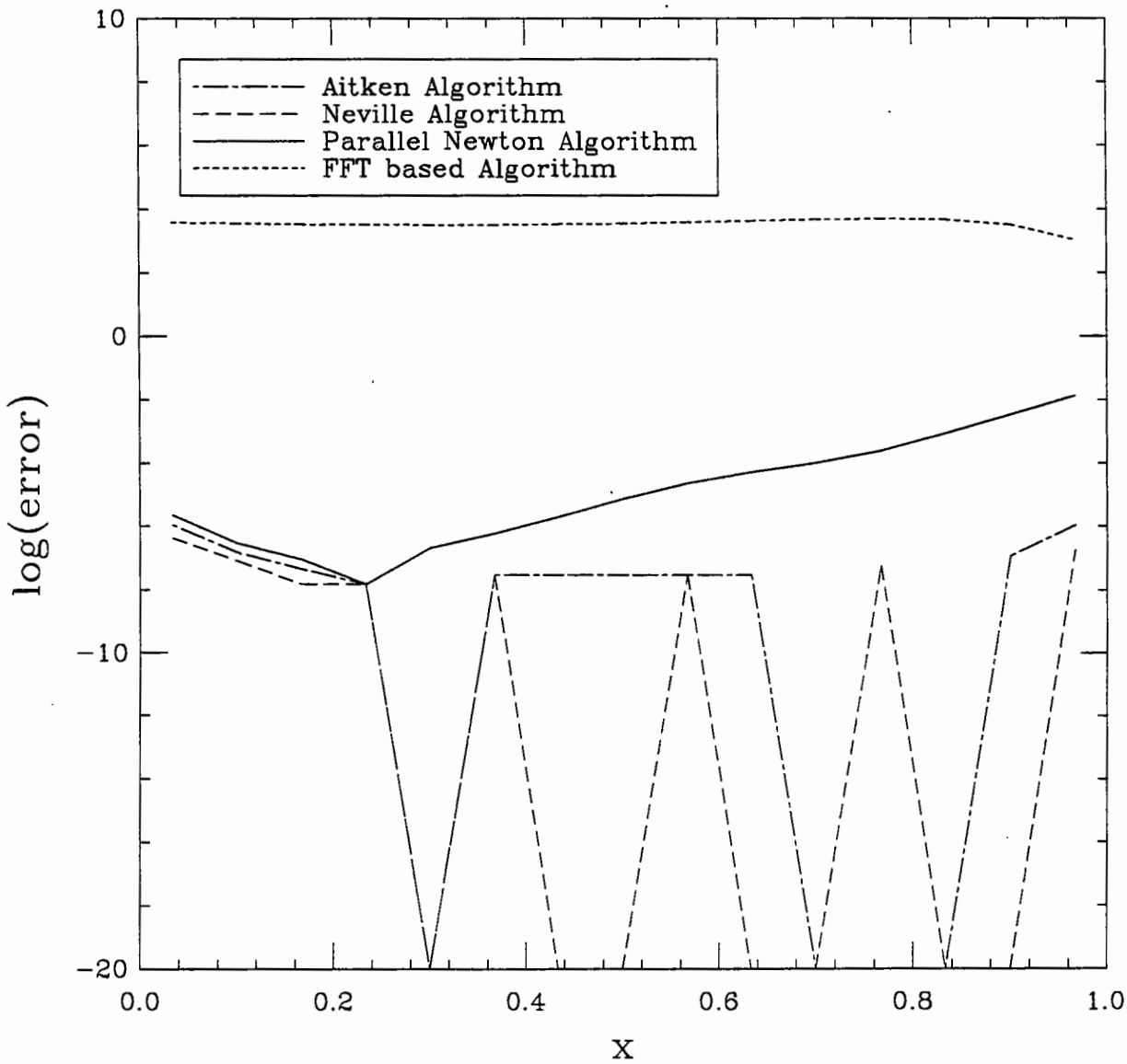


Figure 2.7b. $\log_{10} \left| \ln(1 + x_i + \frac{h}{2}) - P_{15}(x_i + \frac{h}{2}) \right|$, $i = 0 , 1 , \cdots , 14$, interpolation is based on $x_i = i\, h$ for $i = 0 , 1 , \cdots , 15$ and $h = \frac{1}{15}$.

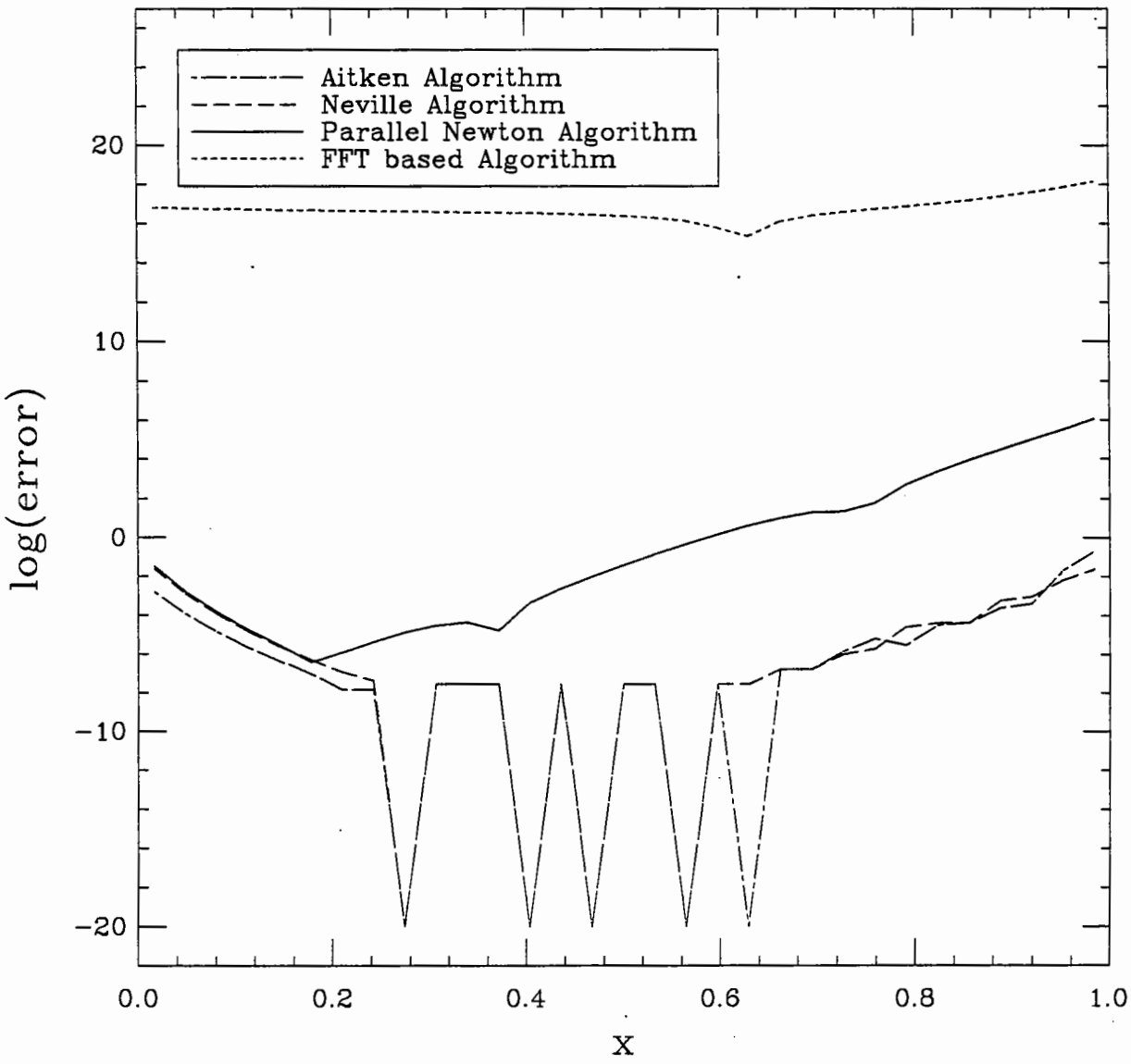


Figure 2.7c. $\log_{10} \left| \ln(1+x_i+\frac{h}{2}) - P_{31}(x_i+\frac{h}{2}) \right|$, $i = 0, 1, \cdots, 30$, interpolation is based on $x_i = i\,h$ for $i = 0, 1, \cdots, 31$ and $h = \frac{1}{31}$.

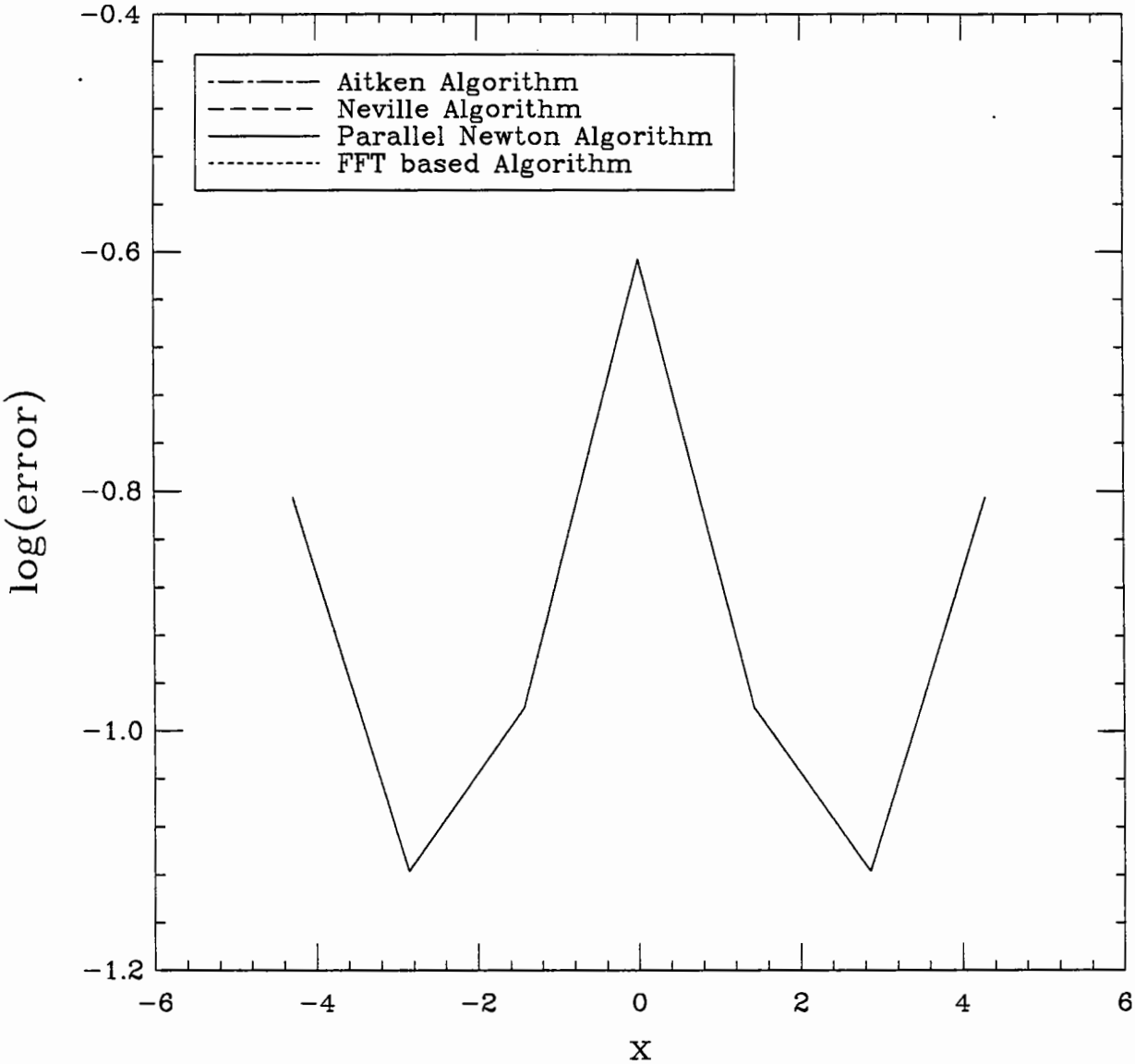


Figure 2.8a. $\log_{10} \left| \frac{1}{1+(x_i+\frac{h}{2})^2} - P_7(x_i+\frac{h}{2}) \right|$, $i = 0, 1, \cdots, 6$, interpolation is based on $x_i = -5 + i\,h$ for $i = 0, 2, \cdots, 6$ and $x_i = i\,h$ for $i = 1, 3, \cdots, 7$ with $h = \frac{5}{7}$.

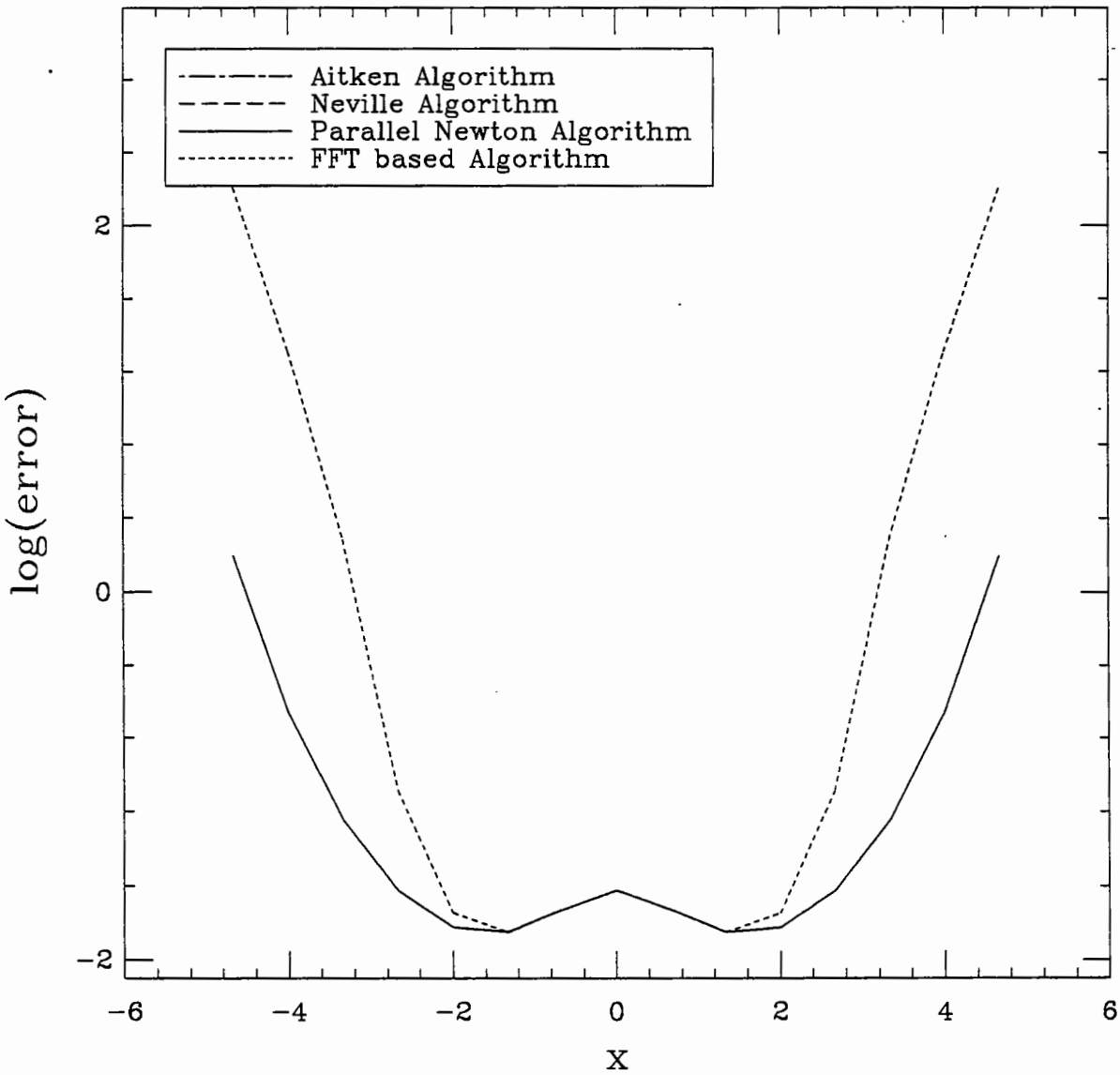


Figure 2.8b. $\log_{10} \left| \frac{1}{1+(x_i+\frac{h}{2})^2} - P_{15}(x_i+\frac{h}{2}) \right|$, $i = 0, 1, \cdots, 14$, interpolation is based on $x_i = -5 + i\,h$ for $i = 0, 2, \cdots, 14$ and $x_i = i\,h$ for $i = 1, 3, \cdots, 15$ with $h = \frac{5}{15}$.

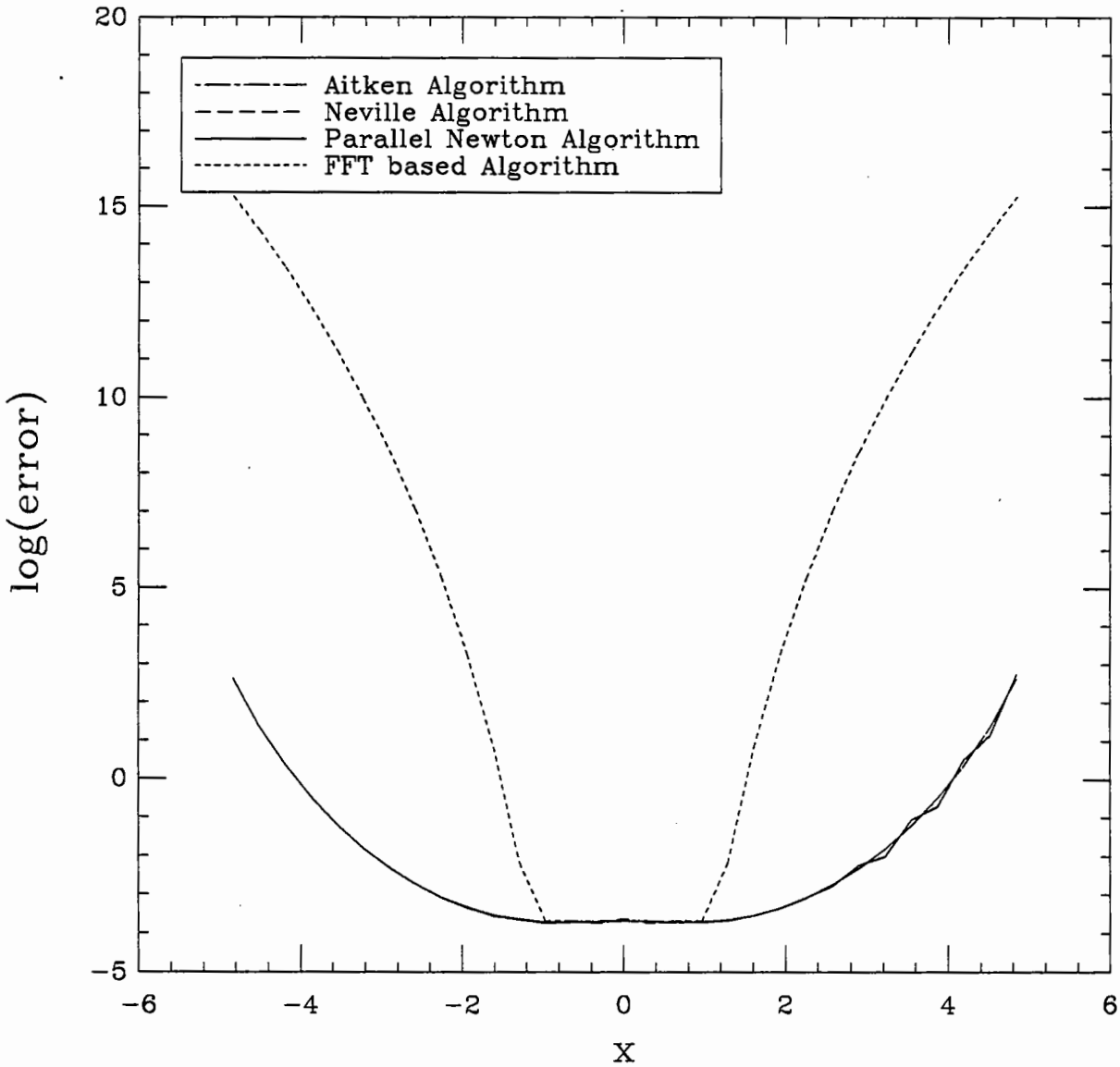


Figure 2.8c. $\log_{10}\left|\frac{1}{1+(x_i+\frac{h}{2})^2}-P_{31}(x_i+\frac{h}{2})\right|$, $i=0,1,\cdots,30$, interpolation is based on $x_i=-5+i\,h$ for $i=0,2,\cdots,30$ and $x_i=i\,h$ for $i=1,3,\cdots,31$ with $h=\frac{5}{31}$.

3

Parallel Hermite Interpolation Algorithm

Given $n+1$ distinct points and arbitrary order derivative information at these points, a parallel algorithm to compute the coefficients of the corresponding Hermite interpolating polynomial in $O(\log n)$ time using $O(n^2)$ processors is presented. The algorithm relies on a novel closed formula that yields the coefficients when the generalized divided differences are expressed linearly in terms of the given function and derivative values. We show that each one of these coefficients and the required linear combinations can be evaluated efficiently. The particular cases where up to first and second order derivative information is available are treated in detail. The proof of the general case, where arbitrarily high order derivative information is available, involves combinatorial arguments that make use of the theory of symmetric functions.

3.1. INTRODUCTION

In this chapter we construct a parallel algorithm for Hermite interpolation that computes the coefficients of the Hermite interpolating polynomial in $O(\log n)$ time using $O(n^2)$ processors for a fixed number of derivatives.

In Chapter 2 we showed that the parallel Newton interpolation algorithm computes the divided differences (DDs) in $O(\log n)$ time using $O(n^2)$ processors. Let †

$$y_{ij} = \frac{1}{x_i - x_j} \tag{3.1}$$

for $i \neq j$, then the p^{th} divided difference of f can be expressed as a linear combination of the given values $f_0, f_1, \cdots, f_p$ with coefficients that are products of the y_{ij}'s in the form

$$f_{012..p} = (y_{01} y_{02} \cdots y_{0p}) f_0 + (y_{10} y_{12} \cdots y_{1p}) f_1 + \cdots + (y_{p0} y_{p1} \cdots y_{p,p-1}) f_p \tag{3.2}$$

where p ranges from 0 to n. The coefficients in the expansion in (3.2) can be written concisely as

$$f_{012..p}\,\big|_{f_i} = y_{i0} y_{i1} \cdots y_{i,i-1} y_{i,i+1} \cdots y_{ip} \quad . \tag{3.3}$$

If all $f_{012..p}\,\big|_{f_i}$'s are known for $(i \leq p \leq n)$ then all of the DDs $(f_0, f_{01}, f_{012}, \cdots, f_{012..n})$ of f that are required for the interpolating Newton polynomial can be calculated in $O(\log n)$ time using $O(n^2)$ processors. In Chapter 2 we showed that the computation of the coefficients $f_{012..p}\,\big|_{f_i}$ can also be done in $O(\log n)$ time with $O(n^2)$ processors by application of the parallel prefix algorithms. Here we will show that the linear expansion and the parallel prefix techniques can also be extended to

† We change the definition of y_{ij} in this chapter for notational convenience.

compute the Hermite interpolating polynomials.

In most practical instances the Hermite interpolating polynomials are constructed for the data

(1) $f(x_i)$ and $f'(x_i)$ for $i = 0, 1, \cdots, n$, or

(2) $f(x_i)$, $f'(x_i)$ and $f''(x_i)$ for $i = 0, 1, \cdots, n$.

In the most general case, it is assumed that for a set of $n+1$ distinct x_i 's the following information about f is available

$$f(x_i), f^{(1)}(x_i), \cdots, f^{(m_i-1)}(x_i) \quad \text{for} \quad i = 0, 1, \cdots, n \quad .$$

Denote $f(x_i)$ by f_i, and $\dfrac{f^{(k-1)}(x_i)}{(k-1)!}$ by f_{ik} then the interpolating Hermite polynomial can be written as

$$\begin{aligned} P(x) = {} & f_0 + f_{0^2}(x-x_0) + f_{0^3}(x-x_0)^2 + \cdots + f_{0^{m_0}}(x-x_0)^{m_0-1} + \\ & f_{0^{m_0}1}(x-x_0)^{m_0} + f_{0^{m_0}1^2}(x-x_0)^{m_0}(x-x_1) + \cdots + f_{0^{m_0}1^{m_1}}(x-x_0)^{m_0}(x-x_1)^{m_1-1} + \\ & f_{0^{m_0}1^{m_1}2}(x-x_0)^{m_0}(x-x_1)^{m_1} + f_{0^{m_0}1^{m_1}2^2}(x-x_0)^{m_0}(x-x_1)^{m_1}(x-x_2) + \cdots + \cdots \\ & f_{0^{m_0}1^{m_1}2^{m_2}\cdots n^{m_n}}(x-x_0)^{m_0}(x-x_1)^{m_1}(x-x_2)^{m_2}\cdots(x-x_n)^{m_n-1} \end{aligned}$$

The terms $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}$ for $1 \le a_j \le m_j$ and $0 \le j \le n$, and $a_0 \ge a_1 \ge \cdots \ge a_p$ for $1 \le p \le n$ are called the *generalized divided differences* (GDDs). The GDDs can be calculated in the same way as the DDs by using the Neville or the Aitken recursion formulae. For instance, the Neville recursion in this case takes the form

$$f_{i^{a_i}\cdots j^{a_j}} = \frac{f_{i^{a_i}\cdots j^{a_j-1}} - f_{i^{a_i-1}\cdots j^{a_j}}}{x_i - x_j} = y_{ij}\,(f_{i^{a_i}\cdots j^{a_j}} - f_{i^{a_i-1}\cdots j^{a_j}}) \quad . \qquad (3.4)$$

Here the terms of the form f_{ik} ; $1<=k\le m_i$ are to be interpreted as $\frac{1}{(k-1)!}f^{(k-1)}(x_i)$. The correctness of this process can be verified by noting that

$$\frac{d^{k-1}}{dx^{k-1}}P(x_i)=(k-1)!\,f_{ik}=f^{(k-1)}(x_i) \quad \text{for} \quad 1\le k\le m_i \quad \text{and} \quad 0\le i\le n$$

holds. As an example, given two points x_0 and x_1 with $m_0=3$, $m_1=2$, the Neville procedure computes the entries in the following triangle

$$\begin{array}{lllll} f_0 & & & & \\ f_0 & f_{0^2} & & & \\ f_0 & f_{0^2} & f_{0^3} & & \\ f_1 & f_{01} & f_{0^21} & f_{0^31} & \\ f_1 & f_{1^2} & f_{01^2} & f_{0^21^2} & f_{0^31^2} \end{array}$$

using the recursion in (3.4).

As in the case for calculating the DDs, the Neville algorithm (also the Aitken algorithm) takes $O(n^2)$ sequential time to calculate all GDDs for fixed values of m_i for $0\le i\le n$. In Chapter 1 we have developed systolic arrays for the Hermite interpolation by observing that the entries 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. The systolic arrays compute the GDDs in $O(n)$ time using $O(n)$ processors for fixed derivative information, e.g. $1\le m_j\le k$ for all $0\le j\le n$ a fixed value of k.

Since we are interested in developing a parallel algorithm to calculate all GDDs, we will seek a linear formula of the form (3.2) for $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}$ in terms of f_{ji} ($0\le j\le n$; $1\le i\le a_j$) for $1\le a_j\le m_j$ and $a_0\ge a_1\ge\cdots\ge a_p$, $1\le p\le n$. It turns out that the coefficient $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}\,|_{f_{ji}}$ can be expressed in a closed form, which reduces to

the expressions for the coefficients that appear in the Newton polynomial (3.3) when $a_0 = a_1 = \cdots = a_n = 1$.

First we give a theorem regarding the computation of all GDDs if the coefficients $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}|_{f_{ji}}$ are known.

Theorem 3.1

Let the coefficients $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}|_{f_{ji}}$ *in the linear expansion of* $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}$ *be known for all* $0 \le j \le n$, $1 \le i \le m_j$, $1 \le a_j \le m_j$, *and* $a_0 \ge a_1 \ge \cdots \ge a_p$, $1 \le p \le n$. *If we have arbitrary but fixed derivative information, i.e.* $m_j \le k$ *for all* $0 \le j \le n$ *for a fixed value of* k, *then all GDDs can computed in* $O(\log n)$ *time using* $O(n^2)$ *processors.*

Proof :

For simplicity assume that m_j take their maximum value for each $0 \le j \le n$, e.g. $m_j = k$ for $0 \le j \le n$. The maximum number of terms appear in the expansion of $f_{0^{m_0}1^{m_1}\cdots n^{m_n}}$ which is the highest order GDD. Now denote $f_{0^{m_0}1^{m_1}\cdots n^{m_n}}|_{f_{ji}}$ by c_{ji} then we have a linear expansion of the form

$$\begin{aligned} f_{0^{m_0}1^{m_1}\cdots n^{m_n}} = {} & c_{01} f_0 + c_{02} f_{0^2} + \cdots + c_{0k} f_{0^k} + \\ & c_{11} f_1 + c_{12} f_{1^2} + \cdots + c_{1k} f_{1^k} + \\ & c_{n1} f_n + c_{n2} f_{n^2} + \cdots + c_{nk} f_{n^k} \end{aligned}$$

which amounts to innerproduct of two vectors of length $k(n+1)$. As the set $a_0, a_1, \cdots, a_n$ spans its all possible values, we have to perform $k\,n$ such innerproduct operations. This is because we have $k\,n$ GDDs to compute, and they are simply

$$f_{0^k 1}\ ,\ f_{0^k 1^2}\ ,\ \cdots\ ,\ f_{0^k 1^k}$$

$$f_{0^k 1^k 2}\ ,\ f_{0^k 1^k 2^2}\ ,\ \cdots\ ,\ f_{0^k 1^k 2^k}$$
$$\cdots$$
$$f_{0^k 1^k 2^k \cdots n}\ ,\ f_{0^k 1^k 2^k \cdots n^2}\ ,\ \cdots\ ,\ f_{0^k 1^k 2^k \cdots n^k}$$

Thus the innerproduct of $k\,n$ vectors each of which has a length less than $k(n+1)$ can be done in $\lceil \log(k\,n+k) \rceil$ parallel arithmetic steps given $k^2 n\,(n+1)$ processors. For a fixed k this amounts $O(\log n)$ time with $O(n^2)$ processors. ●

Thus once we have computed the coefficients $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}\big|_{f_{j^i}}$ then all GDDs can be computed in $O(\log n)$ time with $O(n^2)$ processors. It turns out that these coefficients can also be computed in $O(\log n)$ time by application of the parallel prefix algorithms as it was the case for the Newton interpolation.

3.2. LINEAR EXPANSION OF GENERALIZED DIVIDED DIFFERENCES

In this section, for brevity of notation we will denote y_{0i} by y_i and represent the GDD $f_{0^r 1^s}$ simply by the string $0^r 1^s$ whenever necessary. The repeated application of (3.4) for this particular case with two given points x_0 and x_1 can be represented as a signed and weighted binary tree, where the weight associated with each node at level p is y_1^p. The leftson of each node is obtained by dropping a 1, and the rightson by dropping a 0. The leaves correspond to strings consisting of 0's or 1's only. All the right branches carry a negative sign and the left branches a positive sign in accordance with the signs produced by repeated application of (3.4). The sign of a given node is defined to be the product of all the signs on the path from the node to the root. As an example, when $r=3$ and $s=2$, the representation of the expansion of f_{00011} is shown in Figure 3.1.

From this representation, we see for instance, that $f_{00011}\big|_{f_0}$ is equal to the signed sum

of all the weights of the leaves labeled 0 . Since each 0 omitted on the path from the root to a given node introduces a negative sign, we see that in this case the sign of each leaf labeled 0 is positive. This gives

$$f_{00011}\big|_{f_0} = +3y_1^4 \quad .$$

In general, we have that in the expansion of $f_{0^r1^s}$, the sign of each leaf labeled 0 will be $(-1)^{r-1}$ and that all these leaves are at level $r+s-1$. Furthermore, in the expansion

$$f_{0^r1^s}\big|_{f_0} = (-1)^{r-1} C_o\, y_1^{r+s-1} \quad ,$$

the coefficient C_o is simply the number of leaves labeled 0.

Next, we count the number of leaves labeled 0 in such a tree in the general case. It is not difficult to see that C_o is the number of ways of parenthesizing the string 0^r1^s starting from $0^{r-1}(01)1^{s-1}$ in such a way that each new pair of parentheses introduced contains one more symbol than the previous. For example, with this coding, the leftmost leaf in Figure 3.1 corresponds to the parenthesization $((0(0(01)))1)$, and the rightmost one to $(0(0((01)1)))$.

Let $a_{r,s}$ denote the number of such parenthesizations of the string 0^r1^s . Using this interpretation, or proceeding directly from the recursive structure of a binary tree, we have

$$a_{r,s} = a_{r-1,s} + a_{r,s-1}$$

with $a_{r,1} = a_{1,s} = 1$. By a simple induction, this gives

$$a_{r,s} = \binom{r+s-2}{s-1} \quad .$$

Thus

$$f_{0^r1^s}\big|_{f_0} = (-1)^{r-1} \binom{r+s-2}{s-1} y_1^{r+s-1} \quad . \tag{3.5}$$

Note that treating the string 00 as a single symbol, the derivation of (3.5) also yields that the sign of each leaf labeled 00 is $(-1)^{r-2}$ and that there are $\begin{bmatrix} r+s-3 \\ s-1 \end{bmatrix}$ of these. Thus

$$f_{0^r 1^s}\big|_{f_\infty} = (-1)^{r-2}\begin{bmatrix} r+s-3 \\ s-1 \end{bmatrix} y_1^{r+s-2}$$

and in general we have for $i \le r$

$$f_{0^r 1^s}\big|_{f_i} = (-1)^{r-i}\begin{bmatrix} r+s-i-1 \\ s-1 \end{bmatrix} y_1^{r+s-i} \tag{3.6}$$

Combining all these observations, f_{00011} has the expansion

$$f_{00011} = 3y_1^4 f_0 - 2y_1^3 f_{00} + y_1^2 f_{000} - 3y_1^4 f_1 - y_1^3 f_{11} \ . \tag{3.7}$$

Equivalently, using the notation introduced in (3.1), the expansion in (3.7) takes the form

$$f_{00011} = (3y_{01}^4)f_0 + (-2y_{01}^3)f_{00} + (y_{01}^2)f_{000} + (-3y_{10}^4)f_1 + (y_{10}^3)f_{11} \ .$$

Now we turn to the general case of computing the coefficients f_{0^i} in the expansion of $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}$. Even though the combinatorial treatment of the derivation of (3.6) can be extended to this case, we will prove the general case by induction. First, we give a closed formula for the coefficient of f_0 in the expansion of $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}$.

Theorem 3.2

$$f_{0^{a_0}1^{a_1}\cdots p^{a_p}}\big|_{f_0} =$$

$$(-1)^{a_0-1}\sum_{\lambda_1+\lambda_2+\cdots+\lambda_p=a_0-1}\begin{bmatrix} \lambda_1+a_1-1 \\ \lambda_1 \end{bmatrix}\begin{bmatrix} \lambda_2+a_2-1 \\ \lambda_2 \end{bmatrix}\cdots\begin{bmatrix} \lambda_p+a_p-1 \\ \lambda_p \end{bmatrix} y_1^{\lambda_1+a_1} y_2^{\lambda_2+a_2}\cdots y_p^{\lambda_p+a_p}$$

where the summation is over all nonnegative ordered partitions of $a_0 - 1$.

Proof:

Note first of all that $a_0 = r$ and $a_1 = s$ gives

$$f_{0^r 1^s}\,|_{f_0} = (-1)^{r-1} \sum_{\lambda_1 = r-1} \binom{\lambda_1+s-1}{\lambda_1} y_1^{\lambda_1+s} = (-1)^{r-1} \binom{r+s-2}{s-1} y_1^{r+s-1} \quad ,$$

in agreement with (3.5). Furthermore, in the special case where $a_0 = a_1 = \cdots = a_p = 1$, we obtain the coefficient of the Newton interpolating polynomial

$$f_{01\ldots p}\,|_{f_0} = y_1 y_2 \cdots y_p \quad ,$$

as given in (3.3).

For convenience, set

$$C_{\mathbf{a},m} = \sum_{\lambda_1+\lambda_2+\cdots+\lambda_p = m} \binom{\lambda_1+a_1-1}{\lambda_1} \binom{\lambda_2+a_2-1}{\lambda_2} \cdots \binom{\lambda_p+a_p-1}{\lambda_p} y_1^{\lambda_1+a_1} y_2^{\lambda_2+a_2} \cdots y_p^{\lambda_p+a_p}$$

for $\mathbf{a} = (a_1, a_2, \ldots, a_p)$ and $m \geq 0$. Thus we claim that

$$f_{0^{a_0} 1^{a_1} \cdots p^{a_p}}\,|_{f_0} = (-1)^{a_0-1} C_{\mathbf{a},a_0-1}$$

First, we construct the generating function $F_{\mathbf{a}}$ of the sequence of numbers $C_{\mathbf{a},m}$. Note that by Newton's theorem we have for every i, the formal power series expansion

$$\frac{1}{(1-y_i)^{a_i}} = \sum_{\lambda_i \geq 0} \binom{\lambda_i+a_i-1}{\lambda_i} y_i^{\lambda_i} \quad , \tag{3.8}$$

so that multiplying these expansions for $i = 1, 2, \ldots, p$ gives

$$\frac{y_1^{a_1} y_2^{a_2} \cdots y_p^{a_p}}{(1-y_1)^{a_1} (1-y_2)^{a_2} \cdots (1-y_p)^{a_p}} =$$

$$\sum_{\lambda_1, \lambda_2, \ldots, \lambda_p \geq 0} \binom{\lambda_1+a_1-1}{\lambda_1} \binom{\lambda_2+a_2-1}{\lambda_2} \cdots \binom{\lambda_p+a_p-1}{\lambda_p} y_1^{\lambda_1+a_1} y_2^{\lambda_2+a_2} \cdots y_p^{\lambda_p+a_p} \quad .$$

It follows that

$$F_{\mathbf{a}} = \sum_{m \geq 0} C_{\mathbf{a},m}\, t^m = \frac{y_1^{a_1}\, y_2^{a_2} \cdots y_p^{a_p}}{(1-t\, y_1)^{a_1} (1-t\, y_2)^{a_2} \cdots (1-t\, y_p)^{a_p}} \quad . \tag{3.9}$$

From the recursive definition given in (3.4) for the GDDs, we have

$$f_{0^{a_0}1^{a_1} \cdots p^{a_p}} = y_p \left[f_{0^{a_0}1^{a_1} \cdots p^{a_p-1}} - f_{0^{a_0-1}1^{a_1} \cdots p^{a_p}} \right] \quad .$$

Therefore,

$$f_{0^{a_0}1^{a_1} \cdots p^{a_p}} \Big|_{f_0} = y_p\, f_{0^{a_0}1^{a_1} \cdots p^{a_p-1}} \Big|_{f_0} - y_p\, f_{0^{a_0-1}1^{a_1} \cdots p^{a_p}} \Big|_{f_0} \quad . \tag{3.10}$$

By induction on $\sum_{i=0}^{p} a_i$, we may assume that the two coefficients on the right hand side of (3.10) are given by

$$y_p\, (-1)^{a_0-1}\, F_{(a_1, a_2, \ldots, a_p-1)} \Big|_{t^{a_0-1}} \quad \text{and} \quad y_p\, (-1)^{a_0-2}\, F_{(a_1, a_2, \ldots, a_p)} \Big|_{t^{a_0-2}} \quad ,$$

respectively. Thus to prove the theorem, it suffices to show that

$$F_{(a_1, a_2, \ldots, a_p)} \Big|_{t^{a_0-1}} = y_p\, F_{(a_1, a_2, \ldots, a_p-1)} \Big|_{t^{a_0-1}} + y_p\, F_{(a_1, a_2, \ldots, a_p)} \Big|_{t^{a_0-2}}$$

$$= y_p\, F_{(a_1, a_2, \ldots, a_p-1)} \Big|_{t^{a_0-1}} + y_p\, t\, F_{(a_1, a_2, \ldots, a_p)} \Big|_{t^{a_0-1}} \quad .$$

But by (3.9), this reduces to the verification of the functional identity

$$\frac{y_1^{a_1}\, y_2^{a_2} \cdots y_p^{a_p}}{(1-t\, y_1)^{a_1} (1-t\, y_2)^{a_2} \cdots (1-t\, y_p)^{a_p}} = \tag{3.11}$$

$$= y_p\, \frac{y_1^{a_1}\, y_2^{a_2} \cdots y_p^{a_p-1}}{(1-t\, y_1)^{a_1} (1-t\, y_2)^{a_2} \cdots (1-t\, y_p)^{a_p-1}} + y_p\, t\, \frac{y_1^{a_1}\, y_2^{a_2} \cdots y_p^{a_p}}{(1-t\, y_1)^{a_1} (1-t\, y_2)^{a_2} \cdots (1-t\, y_p)^{a_p}}$$

which is immediate. ●

The general formula for the coefficient $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}\,|f_{0^i}$ can be stated as follows:

Theorem 3.3

$$f_{0^{a_0}1^{a_1}\cdots p^{a_p}}\,|f_{0^i} =$$

$$(-1)^{a_0-i}\sum_{\lambda_1+\lambda_2+\cdots+\lambda_p=a_0-i}\binom{\lambda_1+a_1-1}{\lambda_1}\binom{\lambda_2+a_2-1}{\lambda_2}\cdots\binom{\lambda_p+a_p-1}{\lambda_p}y_1^{\lambda_1+a_1}y_2^{\lambda_2+a_2}\cdots y_p^{\lambda_p+a_p}$$

for every $i \le a_0$, *where the summation is over all nonnegative ordered partitions of* a_0-i.

Proof:

The proof is similar to the proof of Theorem 3.2 and will be omitted. ●

Note that using the notation for the y_{ij}'s given in (3.1), the formula for $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}\,|f_{j^i}$ in the general case takes the form

$$(-1)^{a_j-i}\sum\binom{\lambda_o+a_o-1}{\lambda_o}\cdots\binom{\lambda_{j-1}+a_{j-1}-1}{\lambda_{j-1}}\binom{\lambda_{j+1}+a_{j+1}-1}{\lambda_{j+1}}\cdots\binom{\lambda_p+a_p-1}{\lambda_p}y_{j,0}^{\lambda_o+a_o}\cdots y_{j,j-1}^{\lambda_{j-1}+a_{j-1}}y_{j,j+1}^{\lambda_{j+1}+a_{j+1}}\cdots y_{j,p}^{\lambda_p+a_p}$$

where the summation is carried over all nonnegative ordered partitions $\lambda_o+\cdots+\lambda_{j-1}+\lambda_{j+1}+\cdots+\lambda_p$ of a_j-i.

3.3. SPECIAL HERMITE INTERPOLATING POLYNOMIALS

In this section we will give algorithms for the parallel computation of the GDDs for the following important cases

(1) $f(x_i)$ and $f'(x_i)$ given for $0 \le i \le n$,

(2) $f(x_i)$, $f'(x_i)$ and $f''(x_i)$ given for $0 \le i \le n$.

As it follows from Theorem 3.1 that, if the coefficients of f_{j^i} are known for all $0 \le j \le n$ and $i \le m_j$ then all the necessary GDDs, namely the terms of the form $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}$; $k \ge a_0 \ge a_1 \ge .. \ge a_p$, $1 \le p \le n$, can be calculated in $O(\log n)$ time using $O(n^2)$ processors for a fixed value of k . Hence the problem reduces to calculating the coefficients $f_{0^{a_0}1^{a_1}\cdots p^{a_p}}|_{f_{j^i}}$ for $1 \le i \le k$ and $0 \le j \le p \le n$. Now we will show that for $k = 2$ and $k = 3$ corresponding to the cases (1) and (2) above, these coefficients can also be calculated in $O(\log n)$ time with $O(n^2)$ processors.

CASE (1) $(k = 2)$

From Theorem 3.2 we know that

$$f_{0^2 1^{a_1}\cdots p^{a_p}}|_{f_0} =$$

$$(-1) \sum_{\lambda_1+\lambda_2+\cdots+\lambda_p=1} \begin{bmatrix} \lambda_1+a_1-1 \\ \lambda_1 \end{bmatrix} \begin{bmatrix} \lambda_2+a_2-1 \\ \lambda_2 \end{bmatrix} \cdots \begin{bmatrix} \lambda_p+a_p-1 \\ \lambda_p \end{bmatrix} y_1^{\lambda_1+a_1} y_2^{\lambda_2+a_2} \cdots y_p^{\lambda_p+a_p}$$

for $1 \le p \le n$. This equation simplifies as

$$f_{0^2 1^{a_1}\cdots p^{a_p}}|_{f_0} = -y_1^{a_1} y_2^{a_2} \cdots y_p^{a_p} \sum_{i=1}^{p} a_i y_i \quad .$$

Similarly, for the coefficient of f_{0^2} , we obtain from Theorem 3.3 that

$$f_{0^2 1^{a_1}\cdots p^{a_p}}|_{f_{0^2}} = y_1^{a_1} y_2^{a_2} \cdots y_p^{a_p}$$

Since $2 \ge a_1 \ge a_2 \ge \cdots \ge a_n \ge 1$, as p ranges from 1 to n we obtain the following table for the coefficients of f_{0^2}

$$f_{0^2 1}|_{f_{0^2}} = y_1$$

$$f_{0^2 1^2}\,|_{f_{0^2}} = y_1^2$$

$$f_{0^2 1^2 2}\,|_{f_{0^2}} = y_1^2\, y_2$$

$$f_{0^2 1^2 2^2}\,|_{f_{0^2}} = y_1^2\, y_2^2$$

$$f_{0^2 1^2 2^2 3}\,|_{f_{0^2}} = y_1^2\, y_2^2\, y_3$$

$$f_{0^2 1^2 2^2 3^2}\,|_{f_{0^2}} = y_1^2\, y_2^2\, y_3^2$$

$$\cdots$$

$$f_{0^2 1^2 2^2 3^2 \cdots n}\,|_{f_{0^2}} = y_1^2\, y_2^2\, y_3^2 \cdots y_n$$

$$f_{0^2 1^2 2^2 3^2 \cdots n^2}\,|_{f_{0^2}} = y_1^2\, y_2^2\, y_3^2 \cdots y_n^2$$

Clearly these are the prefixes of the quantities $(y_1, y_1, y_2, y_2, y_3, y_3, \cdots, y_n, y_n)$, and hence they can be calculated in $\log 2n$ time using $2n$ processors. Due to symmetry of the GDDs, all of the terms

$$f_{0^2 1^2 2^2 \cdots (p-1)^2 p}\,|_{f_{j^2}} \quad , \quad f_{0^2 1^2 2^2 \cdots (p-1)^2 p^2}\,|_{f_{j^2}} \qquad 0 \le j \le p \quad , \quad 1 \le p \le n$$

can be calculated using $n+1$ instances of the parallel prefix algorithm. Hence $O(n^2)$ processors suffice to calculate all of them in $O(\log n)$ time.

For the terms $f_{0^2 1^2 2^2 \cdots (p-1)^2 p}\,|_{f_0}$ and $f_{0^2 1^2 2^2 \cdots (p-1)^2 p^2}\,|_{f_0}$ we obtain the following triangular table

$$f_{0^2 1}\,|_{f_0} = -y_1 (y_1)$$

$$f_{0^2 1^2}\,|_{f_0} = -{y_1}^2 (2y_1)$$

$$f_{0^2 1^2 2}\,|_{f_0} = -{y_1}^2 y_2 (2y_1 + y_2)$$

$$f_{0^2 1^2 2^2}\big|_{f_0} = -y_1^2 y_2^2 (2y_1 + 2y_2)$$

$$f_{0^2 1^2 2^2 3}\big|_{f_0} = -y_1^2 y_2^2 y_3 (2y_1 + 2y_2 + y_3)$$

$$f_{0^2 1^2 2^2 3^2}\big|_{f_0} = -y_1^2 y_2^2 y_3^2 (2y_1 + 2y_2 + 2y_3)$$

$$\cdots$$

$$f_{0^2 1^2 2^2 3^2 \cdots n}\big|_{f_0} = -y_1^2 y_2^2 y_3^2 \cdots y_n (2y_1 + 2y_2 + 2y_3 + \cdots + y_n)$$

$$f_{0^2 1^2 2^2 3^2 \cdots n^2}\big|_{f_0} = -y_1^2 y_2^2 y_3^2 \cdots y_n^2 (2y_1 + 2y_2 + 2y_3 + \cdots + 2y_n) .$$

The terms outside the parentheses are the prefixes of the elements

$$(y_1, y_1, y_2, y_2, \cdots, y_n, y_n)$$

with multiplication operation. This can be computed in $\log 2n$ time with $2n$ processors. For the terms inside the parentheses notice that if we apply the prefix algorithm to the same elements we will get terms

$$y_1, 2y_1, 2y_1 + y_2, 2y_1 + 2y_2, \cdots$$

Hence $O(n)$ processors suffice to calculate all the coefficients of f_0 in the linear expansion of $f_{0^2 1^{a_1} \cdots p^{a_p}}$ for $2 \geq a_1 \geq \cdots \geq a_p \geq 1$ and $1 \leq p \leq n$ in $O(\log n)$ time. The coefficients of f_j for j ranging from 0 to n are found by $n+1$ concurrent applications of the procedure explained above.. Hence the coefficients of all f_j and f_{j^2} in the expansion of $f02pa$ for $1 \leq p \leq n$ and $2 \geq a_1 \geq a_2 \geq \cdots \geq a_n \geq 1$ can be found in $O(\log n)$ time using $O(n^2)$ processors.

CASE (2) $(k = 3)$

Now we are interested the coefficients of f_0, f_{0^2} and f_{0^3} in the linear expansion of $f_{0^3 1^{a_1} \cdots p^{a_p}}$ for $3 \geq a_1 \geq \cdots \geq a_p \geq 1$. We will start from the simplest one

$$f_{0^3 1^{a_1} \cdots p^{a_p}} \mid f_{0^3} = y_1^{a_1} y_2^{a_2} \cdots y_p^{a_p} ,$$

which gives us the following triangular table

$$f_{0^3 1} \mid f_{0^3} = y_1$$

$$f_{0^3 1^2} \mid f_{0^3} = y_1^2$$

$$f_{0^3 1^3} \mid f_{0^3} = y_1^3$$

$$f_{0^3 1^3 2} \mid f_{0^3} = y_1^3 y_2$$

$$f_{0^3 1^3 2^2} \mid f_{0^3} = y_1^3 y_2^2$$

$$f_{0^3 1^3 2^3} \mid f_{0^3} = y_1^3 y_2^3$$

$$\cdots$$

$$f_{0^3 1^3 2^3 \cdots n} \mid f_{0^3} = y_1^3 y_2^3 \cdots y_{n-1}^3 y_n$$

$$f_{0^3 1^3 2^3 \cdots n^2} \mid f_{0^3} = y_1^3 y_2^3 \cdots y_{n-1}^3 y_n^2$$

$$f_{0^3 1^3 2^3 \cdots n^3} \mid f_{0^3} = y_1^3 y_2^3 \cdots y_{n-1}^3 y_n^3 .$$

It is apparent that these quantities are the prefix product of the terms

$$(y_1, y_1, y_1, \cdots, y_n, y_n, y_n)$$

and, thus, can be calculated in $\log 3n$ time using $3n$ processors. By applying $n+1$ concurrent instances of the parallel prefix algorithm we find all $f_{0^3 1^{a_1} \cdots p^{a_p}} \mid f_{j^3}$ in $O(\log n)$ time using $O(n^2)$ processors for $3 \geq a_1 \geq a_2 \geq \cdots \geq a_p \geq 1$ and $1 \leq p \leq n$.

For the coefficient of f_{0^2} in the expansion of $f_{0^3 1^{a_1} \cdots p^{a_p}}$, from Theorem 3.3 we obtain the following formula

$$f_{0^3 1^{a_1} \cdots p^{a_p}} \mid f_{0^2} = -y_1^{a_1} y_2^{a_2} \cdots y_p^{a_p} \sum_{i=1}^{p} a_i y_i \ .$$

which can be given explicitly as

$$f_{0^3 1} \mid f_{0^2} = -y_1 (y_1)$$

$$f_{0^3 1^2} \mid f_{0^2} = -y_1^2 (2y_1)$$

$$f_{0^3 1^3} \mid f_{0^2} = -y_1^3 (3y_1)$$

$$f_{0^3 1^3 2} \mid f_{0^2} = -y_1^3 y_2 (3y_1 + y_2)$$

$$f_{0^3 1^3 2^2} \mid f_{0^2} = -y_1^3 y_2^2 (3y_1 + 2y_2)$$

$$f_{0^3 1^3 2^3} \mid f_{0^2} = -y_1^3 y_2^3 (3y_1 + 3y_2)$$

$$\cdots$$

$$f_{0^3 1^3 2^3 \cdots n} \mid f_{0^2} = -y_1^3 y_2^3 \cdots y_{n-1}^3 y_n (3y_1 + 3y_2 + \cdots + y_n)$$

$$f_{0^3 1^3 2^3 \cdots n^2} \mid f_{0^2} = -y_1^3 y_2^3 \cdots y_{n-1}^3 y_n^2 (3y_1 + 3y_2 + \cdots + 2y_n)$$

$$f_{0^3 1^3 2^3 \cdots n^3} \mid f_{0^2} = -y_1^3 y_2^3 \cdots y_{n-1}^3 y_n^3 (3y_1 + 3y_2 + \cdots + 3y_n) \ .$$

As before the terms outside the parentheses are the prefix products of the elements

$$(y_1, y_1, y_1, \cdots, y_n, y_n, y_n)$$

and, the terms inside the parentheses are the prefixes of the same elements with addition operation. It follows that $O(n^2)$ processors are sufficient to calculate in $O(\log n)$ time, all the coefficients $f_{0^3 1^{a_1} \cdots p^{a_p}} \mid f_{j^2}$ for $3 \geq a_1 \geq a_2 \geq \cdots \geq a_p \geq 1$, $1 \leq p \leq n$ and $0 \leq j \leq n$.

To calculate the coefficients of f_0 in the expansion of $f_{0^3 1^{a_1} \cdots p^{a_p}}$ for all

$1 \le p \le n$ we again use Theorem 3.2.

$$f_{0^3 1^{a_1} \cdots p^{a_p}} \Big|_{f_0} =$$

$$\sum_{\lambda_1+\lambda_2+\cdots+\lambda_p=2} \begin{bmatrix} \lambda_1+a_1-1 \\ \lambda_1 \end{bmatrix} \begin{bmatrix} \lambda_2+a_2-1 \\ \lambda_2 \end{bmatrix} \cdots \begin{bmatrix} \lambda_p+a_p-1 \\ \lambda_p \end{bmatrix} y_1^{\lambda_1+a_1} y_2^{\lambda_2+a_2} \cdots y_p^{\lambda_p+a_p}$$

The sum $\lambda_1 + \lambda_2 + \cdots + \lambda_p$ can be equal to 2 only in two ways :

(i) $\lambda_i = 2$ for $1 \le i \le p$,

(ii) $\lambda_i = \lambda_j = 1$ for $1 \le i, j \le p$ and $i \ne j$.

By separating these two cases in the sum operation we obtain

$$f_{0^3 1^{a_1} \cdots p^{a_p}} \Big|_{f_0} =$$

$$= y_1^{a_1} y_2^{a_2} \cdots y_p^{a_p} \left[\sum_{1 \le i \le p} \begin{bmatrix} a_i+1 \\ 2 \end{bmatrix} y_i^2 + \sum_{\substack{1 \le i,j \le p \\ i \ne j}} \begin{bmatrix} a_i \\ 1 \end{bmatrix} \begin{bmatrix} a_j \\ 1 \end{bmatrix} y_i y_j \right]$$

$$= y_1^{a_1} y_2^{a_2} \cdots y_p^{a_p} \left[\sum_{1 \le i \le p} \frac{a_i (a_i+1)}{2} y_i^2 + \sum_{\substack{1 \le i,j \le p \\ i \ne j}} a_i a_j y_i y_j \right]$$

$$= y_1^{a_1} y_2^{a_2} \cdots y_p^{a_p} \left[\sum_{1 \le i \le p} \frac{a_i (a_i+1)}{2} y_i^2 + \frac{1}{2} \left[\sum_{1 \le i \le p} a_i y_i \right]^2 - \frac{1}{2} \left[\sum_{1 \le i \le p} a_i^2 y_i^2 \right] \right]$$

$$= \frac{1}{2} y_1^{a_1} y_2^{a_2} \cdots y_p^{a_p} \left[\sum_{1 \le i \le p} a_i y_i^2 + \left[\sum_{1 \le i \le p} a_i y_i \right]^2 \right]$$

These quantities can be arranged in a triangular table as in the previous cases

$$f_{0^3 1} \Big|_{f_0} = \frac{1}{2} y_1 [y_1^2 + (y_1)^2]$$

$$f_{0^3 1^2} \Big|_{f_0} = \frac{1}{2} y_1^2 [2y_1^2 + (2y_1)^2]$$

$$f_{0^3 1^3}\big|_{f_\bullet} = \frac{1}{2} y_1{}^3 [3y_1{}^2 + (3y_1)^2]$$

$$f_{0^3 1^3 2}\big|_{f_\bullet} = \frac{1}{2} y_1{}^3 y_2 [3y_1{}^2 + y_2{}^2 + (3y_1 + y_2)^2]$$

$$f_{0^3 1^3 2^2}\big|_{f_\bullet} = \frac{1}{2} y_1{}^3 y_2{}^2 [3y_1{}^2 + 2y_2{}^2 + (3y_1 + 2y_2)^2]$$

$$f_{0^3 1^3 2^3}\big|_{f_\bullet} = \frac{1}{2} y_1{}^3 y_2{}^3 [3y_1{}^2 + 3y_2{}^2 + (3y_1 + 3y_2)^2]$$

$$\cdots$$

$$f_{0^3 1^3 2^3 \cdots (n-1)^3 n}\big|_{f_\bullet} =$$

$$\frac{1}{2} y_1{}^3 y_2{}^3 \cdots y_{n-1}^3 y_n [3y_1{}^2 + 3y_2{}^2 + \cdots + 3y_{n-1}^2 + y_n^2 + (3y_1 + 3y_2 + \cdots + 3y_{n-1} + y_n)^2]$$

$$f_{0^3 1^3 2^3 \cdots (n-1)^3 n^2}\big|_{f_\bullet} =$$

$$\frac{1}{2} y_1{}^3 y_2{}^3 \cdots y_{n-1}^3 y_n{}^2 [3y_1{}^2 + 3y_2{}^2 + \cdots + 3y_{n-1}^2 + 2y_n{}^2 + (3y_1 + 3y_2 + \cdots + 3y_{n-1} + 2y_n)^2]$$

$$f_{0^3 1^3 2^3 \cdots (n-1)^3 n^3}\big|_{f_\bullet} =$$

$$\frac{1}{2} y_1{}^3 y_2{}^3 \cdots y_{n-1}^3 y_n{}^3 [3y_1{}^2 + 3y_2{}^2 + \cdots + 3y_{n-1}^2 + 3y_n{}^2 + (3y_1 + 3y_2 + \cdots + 3y_{n-1} + 3y_n)^2]$$

It is not difficult to see that these quantities can also be calculated in $O(\log n)$ time using only $O(n)$ processors by application of the parallel prefix algorithms. We conclude that the coefficients of f_0 , f_{0^2} and f_{0^3} can be computed in $O(\log n)$ time using $O(n^2)$ processors. Hence all GDDs of the form $f_{0^3 1^{a_1} \cdots p^{a_p}}$ can be calculated in $O(\log n)$ time using $O(n^2)$ processors for $3 \geq a_1 \geq a_2 \geq \cdots \geq a_p \geq 1$ and $1 \leq p \leq n$.

3.4. THE GENERAL CASE

Next, we will show that in the most general case where up to k^{th} order derivatives are given (i.e. each $a_i \leq k$), the computation of the coefficients of the Hermite interpolating polynomial still can be computed in $O(\log n)$ parallel time using $O(n^2)$ processors.

Theorem 3.4

The coefficients of an arbitrary order Hermite interpolating polynomial can be computed in $O(\log n)$ *parallel time using* $O(n^2)$ *processors.*

Proof:

The idea of the proof rests on the theory of symmetric functions, and we give a sketch of the basic ideas involved.

Given an infinite set of variables $y_1, y_2, \cdots$, denote by Λ the ring of symmetric functions in these variables with rational coefficients. The m^{th} homogeneous symmetric function $h_m(y_1, y_2, \cdots)$ and the m^{th} power symmetric function $\psi_m(y_1, y_2, \cdots)$ are defined by setting

$$h_m(y_1, y_2, \cdots) = \sum_{i_1 \le i_2 \le \cdots \le i_m} y_{i_1} y_{i_2} \cdots y_{i_m} \tag{3.12}$$

$$\psi_m(y_1, y_2, \cdots) = \sum_{i \ge 1} y_i^m \ . \tag{3.13}$$

Furthermore, for any partition $\lambda = (\lambda_1 \ge \lambda_2 \ge \cdots \ge \lambda_r \ge 0)$, put

$$h_\lambda = h_{\lambda_1} h_{\lambda_2} \cdots h_{\lambda_r} \ , \qquad \psi_\lambda = \psi_{\lambda_1} \psi_{\lambda_2} \cdots \psi_{\lambda_r} \ .$$

It is well known [MacD50] that the collections $\{h_\lambda\}$ and $\{\psi_\lambda\}$, where λ runs through all partitions, form bases for Λ. Homogeneous and power symmetric functions are related by the determinantal formula

$$m!\, h_m = \det \begin{bmatrix} \psi_1 & -1 & 0 & \cdots & 0 \\ \psi_2 & \psi_1 & -2 & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \psi_{m-1} & \psi_{m-2} & \cdot & \cdots & -m+1 \\ \psi_m & \psi_{m-1} & \cdot & \cdots & \psi_1 \end{bmatrix} \tag{3.14}$$

For finitely many variables $y_1, y_2, \cdots, y_n$, the power series h_λ and ψ_λ become symmetric polynomials by setting $y_i = 0$ for $i > n$ in (3.12) and (3.13). In this case $\{ h_\lambda \}$ and $\{ \psi_\lambda \}$, where λ is a partition of d, form rational bases for the symmetric functions in $y_1, y_2, \cdots, y_n$ that are homogeneous of degree d. In particular, for any partition λ of d

$$h_\lambda(y_1, y_2, \cdots, y_n) = \sum_\mu c_\mu \psi_\mu(y_1, y_2, \cdots, y_n) \ .$$

for some rational numbers c_μ where μ runs through all partitions of d. Note that by (3.14), the coefficients c_μ are independent of n.

From (3.12), it easily follows that the generating function of the h_m's is given by

$$\sum_{m \geq 0} h_m(y_1, y_2, \cdots, y_n)\, t^m = \frac{1}{(1-t\,y_1)(1-t\,y_2)\cdots(1-t\,y_n)} \ . \tag{3.15}$$

From the generating function in (3.15), we have that for any positive integer a,

$$\frac{1}{(1-t\,y_1)^a\,(1-t\,y_2)^a\,\cdots(1-t\,y_n)^a} = \sum_{m \geq 0} C_{a,m}(\psi_1, \psi_2, \cdots, \psi_m)\, t^m \tag{3.16}$$

where $C_{a,m}$ is a homogeneous form of degree m.

Now we consider the expansion of the generating function $F_{\mathbf{a}}$ in terms of the power symmetric functions. Note that by the symmetry of the computation of the GDDs, we may assume that $a_0 \geq a_1 \geq a_2 \geq \cdots \geq a_n$, and all of these numbers are bounded above by k. The

expansion

we need for the $F_{\mathbf{a}}$ is best communicated by an example. Suppose $n=4$ and $\mathbf{a}=(4,4,3,2)$. Then

$$F_{\mathbf{a}}=\frac{y_1^4 y_2^4 y_3^3 y_4^2}{(1-t\,y_1)^4(1-t\,y_2)^4(1-t\,y_3)^3(1-t\,y_4)^2}$$

can be written in the form

$$(y_1^4 y_2^4 y_3^3 y_4^2)\times\frac{1}{(1-t\,y_1)^2(1-t\,y_2)^2(1-t\,y_3)^2(1-t\,y_4)^2}\times\frac{1}{(1-t\,y_1)(1-t\,y_2)(1-t\,y_3)}\times\frac{1}{(1-t\,y_1)(1-t\,y_2)}$$

Clearly, in such an expansion, the number of products of the form

$$\frac{1}{(1-t\,y_1)^a\,(1-t\,y_2)^a\,\cdots(1-t\,y_n)^a}$$

depends only on k.

From the expansion in (3.16), it follows that in general

$$\frac{F_{\mathbf{a}}}{y_1^{a_1}y_2^{a_2}\cdots y_n^{a_n}}\Big|_{t^m}$$

is a linear combination of power symmetric functions $\psi_\lambda(y_1,y_2,\cdots,y_i)$ for various $i\le n$, where λ is a partition of m.

Note that the determinantal expansion of h_m in terms of the power symmetric functions given in (3.14) is valid in the ring Λ, and therefore does not depend on the number of variables $y_1,y_2,\cdots,y_n$. Hence in the expansion (3.16), the number of terms of the form ψ_λ that appear as summands of the functions $C_{a,m}$ is independent of n. This means in return that the number of terms of the form ψ_λ that appear in

$$\frac{F_{\mathbf{a}}}{y_1^{a_1}y_2^{a_2}\cdots y_n^{a_n}}\Big|_{t^m}$$

is independent of n . Note further that for each power symmetric function ψ_λ that is such a summand, λ is a partition of m . Since we are interested in the coefficients of t^m in F_a for the values $m = 0, 1, \cdots, a_i - 1$ and $a_i \leq k$ for every i , the number of individual power symmetric functions ψ_r that enters as a product in ψ_λ only depends on k and not on n .

It follows that the coefficients of the Hermite interpolating polynomial can be calculated in $O(\log n)$ parallel time using $O(n^2)$ processors, provided each $\psi_r(y_1, y_2, \cdots, y_i)$ and each fixed product of the form $y_1^{a_1} y_2^{a_2} \cdots y_n^{a_n}$ can be computed in $O(\log n)$ time using $O(n^2)$ processors. But for each such computation, the analogue of the parallel prefix algorithm that was demonstrated in the computation of the cases $k = 2$ and $k = 3$ is applicable. Our claim follows from this observation. ●

We remark that even though the running time is guaranteed to be logarithmic in n with $O(n^2)$ processors in the general case, the constants hidden in the big O notation are necessarily exponential in k if no other shortcuts are taken into account. This is not surprising in view of the formula for the coefficient of f_0 given in Theorem 3.2, since the summation involved is over all ordered partitions, and there are exponentially many ordered partitions of k into n parts in general. As an example, the expansion of

$$f_{0^3 1^3 2^3 \cdots (n-1)^3 n^2}\Big|_{f_0}$$

in terms of the power symmetric functions is obtained by taking the coefficient of t^2 in

$$\frac{y_1^3 y_2^3 \cdots y_{n-1}^3 y_n^2}{(1-t\,y_1)^3 (1-t\,y_2)^3 \cdots (1-t\,y_{n-1})^3 (1-t\,y_n)^2} =$$

$$y_1^3 y_2^3 \cdots y_{n-1}^3 y_n^2 \left[\sum_{m \geq 0} h_m(y_1, y_2, .., y_n)\, t^m \right]^2 \left[\sum_{m \geq 0} h_m(y_1, y_2, .., y_{n-1})\, t^m \right]$$

and then expressing each homogeneous symmetric function in terms of the power symmetric functions by the determinantal expansion (3.14). For this particular case, we obtain the expression

$$f_{0^3 1^3 2^3 \cdots (n-1)^3 n^2}\Big|_{f_0} = y_1^3 y_2^3 \cdots y_{n-1}^3 y_n^2 \times$$
$$\Big[\psi_1^2(y_1,..,y_n) + \psi_2(y_1,..,y_n) + \frac{1}{2}\psi_2(y_1,..,y_{n-1}) + \frac{1}{2}\psi_2(y_1,..,y_{n-1}) +$$
$$\psi_1^2(y_1,..,y_n) + 2\psi_1(y_1,..,y_n)\psi_1(y_1,..,y_{n-1}) \Big]$$

as opposed to the much simpler expansion obtained in Section 3.3.

Thus for particular cases involving small values of k, it seems possible to cut down the constants in question by considering special expansions with smaller number of terms than the above treatment would produce. Thus the algorithm implied by the above proof for arbitrary k has more of an existential flavor, and special instances (e.g. $k = 4, 5$) can be made more efficient by judicious grouping of the terms involved in the expansion of the formula in Theorem 3.3.

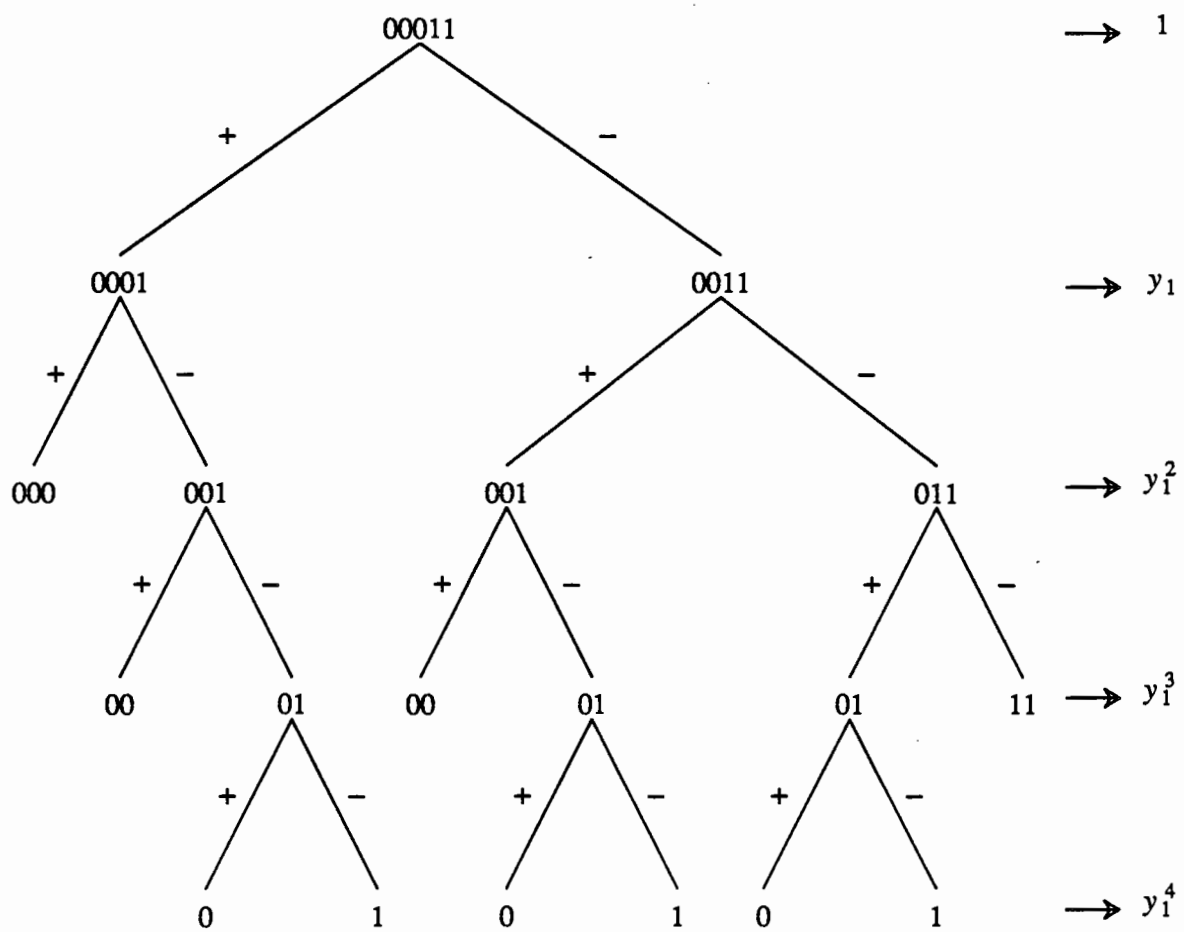


Figure 3.1. The Representation of the Expansion of f_{00011} ($r = 3$ and $s = 2$).

4

Parallel Prefix Algorithms for Tridiagonal Systems

The recursive doubling algorithm as developed by Stone can be used to solve a tridiagonal linear system of size n in $O(\log n)$ steps on a parallel computer with n processors. Here, we give a limited processor version of the recursive doubling algorithm for the solution of tridiagonal linear systems using $O(\frac{n}{p} + \log p)$ parallel arithmetic steps on a parallel computer with $p < n$ processors. The main technique relies on parallel prefix algorithms, which can be efficiently mapped on the hypercube architecture using the binary-reflected Gray code. For $p << n$ this algorithm achieves linear speed-up and constant efficiency over its sequential implementation as well as over the sequential LU decomposition algorithm. These results are confirmed by numerical experiments obtained on an Intel iPSC/d5 hypercube multiprocessor.

4.1. INTRODUCTION

We are interested in solving the following system of linear equations

$$\mathbf{A}\,\mathbf{x} = \mathbf{d} \tag{4.1}$$

where $\mathbf{A}$ is a (nonsymmetric) tridiagonal matrix of order n

$$\mathbf{A} = \begin{bmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_1 & & & \\ & a_2 & b_2 & c_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-2} & b_{n-2} & c_{n-2} \\ & & & & a_{n-1} & b_{n-1} \end{bmatrix}$$

and $\mathbf{x}$ and $\mathbf{d}$ are vectors of dimension n

$$\mathbf{x} = (x_0, x_1, .., x_{n-2}, x_{n-1})^T$$
$$\mathbf{d} = (d_0, d_1, .., d_{n-2}, d_{n-1})^T \ .$$

We shall assume that $\mathbf{A}$, $\mathbf{x}$, and $\mathbf{d}$ have real coefficients. Extension to the complex case is straightforward.

Tridiagonal systems of equations appear frequently in the solution of partial differential equations, cubic spline interpolation, and in numerous other areas of science and engineering. There has been a considerable amount of work to solve (4.1) on parallel computers; see, for example, the review articles [Hell78], [OrVo85], and [Ston75]. More recently Johnsson, et al. have developed algorithms to solve such systems on ensemble architectures [John86], [John87a], [John87b], [JoHo87]. The recursive doubling algorithm is one of the first algorithms that has resulted from considering parallelism in computation. This approach relates the LDU decomposition of $\mathbf{A}$ to first and second order linear recurrences. The well known relationship between (4.1) and linear recurrences was utilized by Stone to develop an algo-

rithm to solve (4.1) in $O(\log n)$ parallel arithmetic steps with n processors [Ston73]. This algorithm can be generalized to solve banded linear systems as well [KoSt73].

The recursive doubling algorithm is suitable when a large number of processing elements are available, such as on the Connection Machine. In this chapter we give a limited processor version of the recursive doubling algorithm on hypercube multiprocessor architectures with $p < n$ processors. This algorithm is more suitable for hypercubes of smaller dimension such as the Caltech Hypercube, the Intel iPSC series, and the NCUBE. We show that the limited processor version recursive doubling algorithm solves a tridiagonal system of size n with arithmetic complexity $O(\frac{n}{p} + \log p)$ and communication complexity $O(\log p)$ on a hypercube multiprocessor with p processors. The algorithm becomes more efficient if $p << n$. The main techniques rely on fast parallel prefix algorithms for which we describe an efficient mapping using the binary-reflected Gray code. These techniques can also be extended to solve banded or block tridiagonal linear systems.

We compare the algorithm proposed here to the LU decomposition algorithm and to a sequential version of the recursive doubling algorithm. The theoretical estimates for speed-up and efficiency, as well as the experimental results on an Intel iPSC/d5 hypercube multiprocessor indicate that the limited processor recursive doubling algorithm achieves linear speed-up and its efficiency is more than 0.5.

4.2. THE LU DECOMPOSITION ALGORITHM

One of the most efficient existing sequential algorithms for solving (4.1) relies on the LU decomposition of $\mathbf{A}$; see, for example, [DBMS79]. Here $\mathbf{A}$ is decomposed into a product of two bidiagonal matrices $\mathbf{L}$ and $\mathbf{U}$ as follows :

$$\mathbf{A} = \mathbf{L}\,\mathbf{U} = \begin{bmatrix} 1 & & & \\ e_1 & 1 & & \\ & \ddots & \ddots & \\ & & e_{n-2} & 1 \\ & & & e_{n-1} \; 1 \end{bmatrix} \begin{bmatrix} f_0 & c_0 & & \\ & f_1 & c_1 & \\ & & \ddots & \ddots \\ & & & f_{n-2} \; c_{n-2} \\ & & & f_{n-1} \end{bmatrix}$$

The algorithm then proceeds to solve for $\mathbf{y}$ from $\mathbf{L}\,\mathbf{y} = \mathbf{d}$ and then finds $\mathbf{x}$ by solving $\mathbf{U}\,\mathbf{x} = \mathbf{y}$. More precisely, the LU decomposition algorithm (the LU Algorithm) to solve the system (4.1) consists of the following steps:

The LU Algorithm

Step 1. Compute LU decomposition of $\mathbf{A}$ given by

$$\begin{aligned} f_0 &= b_0 \\ e_i &= a_i / f_{i-1} \qquad 1 \le i \le n-1 \\ f_i &= b_i - e_i * c_{i-1} \qquad 1 \le i \le n-1 \; . \end{aligned}$$

Step 2. Solve for $\mathbf{y}$ from $\mathbf{L}\,\mathbf{y} = \mathbf{d}$ using

$$\begin{aligned} y_0 &= d_0 \\ y_i &= d_i - e_i * y_{i-1} \qquad 1 \le i \le n-1 \; . \end{aligned}$$

Step 3. Compute $\mathbf{x}$ by solving $\mathbf{U}\,\mathbf{x} = \mathbf{y}$ using

$$\begin{aligned} x_{n-1} &= y_{n-1} / f_{n-1} \\ x_i &= (y_i - c_i * x_{i+1}) / f_i \qquad 0 \le i \le n-2 \; . \end{aligned}$$

We record the number of arithmetic operations required by the algorithm as

Theorem 4.1

The LU Algorithm solves the tridiagonal linear system of size n *using* $8n - 7$ *arithmetic operations.*

Proof :

The proof is straightforward counting of the number of multiplication, division, and subtraction operations performed in Steps 1, 2, and 3 above. ●

4.3. APPLICATION OF PREFIX ALGORITHMS

The equation (4.1) can be represented as a three-term recurrence relation

$$a_i\, x_{i-1} + b_i\, x_i + c_i\, x_{i+1} = d_i \tag{4.2}$$

for $1 \le i \le n-2$ with

$$b_0 x_0 + c_0 x_1 = d_0$$
$$a_{n-1} x_{n-2} + b_{n-1} x_{n-1} = d_{n-1} .$$

Define $a_0 = c_{n-1} = 1$ and $x_{-1} = x_n = 0$. Then with this convention, the relation in (4.2) holds for $0 \le i \le n-1$.

Solving for x_{i+1} in equation (4.2) we get

$$x_{i+1} = -\frac{b_i}{c_i}\, x_i - \frac{a_i}{c_i}\, x_{i-1} + \frac{d_i}{c_i} \quad . \tag{4.3}$$

Here we assume that all c_i 's are nonzero, since otherwise the system of equations can be broken into two decoupled tridiagonal systems which can then be treated separately. Setting

$$\alpha_i = -\frac{b_i}{c_i} \qquad \beta_i = -\frac{a_i}{c_i} \qquad \gamma_i = \frac{d_i}{c_i}$$

(4.3) can be rewritten as

$$x_{i+1} = \alpha_i\, x_i + \beta_i\, x_{i-1} + \gamma_i$$

for $0 \le i \le n-1$. This recurrence formula can be put in a matrix form neatly as

$$\begin{bmatrix} x_{i+1} \\ x_i \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix}$$

which is essentially the same idea developed in [Ston73]. Now define

$$X_i = \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix} \qquad \text{and} \qquad B_i = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} .$$

Then we may write

$$X_{i+1} = B_i X_i \qquad 0 \le i \le n-1 \tag{4.4}$$

This matrix recursion formula allows us to calculate all X_i for $1 \le i \le n-1$ provided that the initial vector X_0 is available. Since

$$X_0 = \begin{bmatrix} x_0 \\ x_{-1} \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ 0 \\ 1 \end{bmatrix}$$

all we need is to calculate x_0 to start the computation. Now note that by repeated application of (4.4) we obtain

$$\begin{aligned} X_1 &= B_0 X_0 \\ X_2 &= B_1 X_1 = B_1 B_0 X_0 \\ &\cdots \\ X_n &= B_{n-1} B_{n-2} \cdots B_1 B_0 X_0 \ . \end{aligned}$$

Now let

$$C_i = B_i B_{i-1} \cdots B_1 B_0 \qquad 0 \le i \le n-1 \ .$$

Then $X_n = C_{n-1} X_0$, or more explicitly

$$\begin{bmatrix} x_n \\ x_{n-1} \\ 1 \end{bmatrix} = \begin{bmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_{-1} \\ 1 \end{bmatrix} ,$$

where

$$C_{n-1} = \begin{bmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ 0 & 0 & 1 \end{bmatrix}$$

and the g_{ij} depend on α_i , β_i , γ_i for $0 \le i \le n-1$. Since $x_n = x_{-1} = 0$, by multiplying the first row of C_{n-1} with X_0 we obtain

$$0 = g_{00} x_0 + g_{02} ,$$

which gives us x_0 as

$$x_0 = -\frac{g_{02}}{g_{00}} . \tag{4.5}$$

Once X_0 is available in this manner, we can calculate all X_i for $1 \le i \le n-1$ by using the matrix recursion formula $X_i = C_{i-1} X_0$.

The sequential prefix algorithm (The SP Algorithm) to solve the tridiagonal system (4.1) thus proceeds as follows :

The SP Algorithm

Step 1. Form the matrices B_i for $0 \le i \le n-1$ using

$$\alpha_i = -\frac{b_i}{c_i} \qquad \beta_i = -\frac{a_i}{c_i} \qquad \gamma_i = \frac{d_i}{c_i}$$

and

$$B_i = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} .$$

Step 2. Compute the chain products C_i by

$$C_0 = B_0$$
$$C_i = B_i \, C_{i-1} \qquad 1 \le i \le n-1$$

Step 3. Denote C_{n-1} computed in Step 2 by

$$C_{n-1} = \begin{bmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ 0 & 0 & 1 \end{bmatrix} .$$

Compute x_0 and hence X_0 using

$$x_0 = -\frac{g_{02}}{g_{00}} \quad \text{and} \quad X_0 = \begin{bmatrix} x_0 \\ x_{-1} \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ 0 \\ 1 \end{bmatrix} .$$

Step 4. Compute X_i and hence x_i using

$$X_i = \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix} = C_{i-1} X_0 \qquad 1 \le i \le n-1 .$$

Step 2 of this algorithm essentially calculates prefixes of the matrices $(B_0, B_1, B_2, \cdots, B_{n-1})$ (here we imagine that the matrix products are performed in reverse order). If this algorithm is used to solve a tridiagonal system of dimension n sequentially, then $O(n)$ arithmetic operations suffice, but the algorithm turns out to be slightly less efficient then the LU Algorithm. Nevertheless it is more suitable for efficient implementation on a parallel machine than the LU Algorithm.

Theorem 4.2

The SP Algorithm for the solution of the tridiagonal linear system of equations (4.1) requires $15n - 11$ *arithmetic operations.*

Proof :

Step 1 requires $3n$ divisions to form the B_i matrices. In Step 2 we perform $n-1$ matrix multiplications to compute the C_i matrices, but because of the special structure of the matrices each matrix multiplication can be performed using 6 floating-point multiplications and 4 floating-point additions. Hence Step 2 requires $6(n-1)$ multiplications and $4(n-1)$ additions. Step 3 is a single division. In Step 4 to compute all x_i for $1 \le i \le n-1$

we perform $n-1$ multiplications and $n-1$ additions. Thus the total number of arithmetic operations sums to $15n - 11$. ●

4.4. PARALLEL PREFIX ON HYPERCUBE MULTIPROCESSORS

In this section we show that the prefix algorithm for the solution of a tridiagonal linear system of equations can be implemented efficiently on hypercube multiprocessors.

Step 2 of the SP Algorithm where the prefixes of the matrices $(B_0, B_1, \cdots, B_{n-1})$ are computed is the bottleneck point in the algorithm. An efficient parallel implementation of the recursive doubling algorithm depends on how efficiently this computation can be performed. Various parallel algorithms have been developed for prefix computation [KrRS85], [LaFi80]. The prefixes of the quantities $(q_0, q_1, \cdots, q_{n-1})$ can be computed in $\log n$ steps† given n processors. Here each step consists of a suitably defined binary operation performed in any of the identical processors. For $n = 8$ the parallel prefix algorithm is given in Figure 4.1. This algorithm is the same as the algorithms given in [KrRS85] and [Ston73]. For simplicity we denote the product block $q_j\, q_{j-1} .. q_{i+1}\, q_i$ as $j\ i$. For example $q_7 q_6 q_5 q_4$ is denoted by the pair $7\,4$.

If the element q_i is initially allocated to processor p_i then at step k, for $1 \le k \le \log n$, processor p_i sends its data to processor p_j where $j = i + 2^{k-1}$. Processor p_j receives this data and multiplies with its own and writes the result where its data resides.

The implementation of this algorithm on a hypercube multiprocessor will be efficient only if the communication requirements of the algorithm are minimal. This requires that we map the parallel prefix algorithm efficiently on the cube. Using the results of Lemma 2.2 and

† All logarithms are base 2.

Lemma 2.3 of Chapter 2, we allocate the element q_i to processor $G(i)$ where $G(i)$ is the *binary-reflected Gray code of the integer* i. The parallel prefix algorithm requires that at step k for $1 \le k \le \log n$, the node to which element q_i is allocated should communicate with the node to which element $q_{i+2^{k-1}}$ is allocated. The distance between nodes $G(i)$ and $G(i+2^{k-1})$ is 1 if $k=1$ and 2 if $2 \le k \le \log n$. Hence we see that by making use of the properties of a Gray code, locality is achieved at the sole expense of slightly increasing the number of routing instructions. The hypercube implementation of the parallel prefix algorithm proposed here requires at most twice the number of routing instructions of a fully-connected system implementation.

The following pseudo-code shows the required computations. This code runs in all nodes concurrently. The binary address of each node is returned when the subroutine *node_id* ( ) is called. The subroutine $G^{-1}(.)$ converts from Gray code to binary code. For example $G^{-1}(110) = 100$. Initially the node $G(i)$ contains the element q_i. This element, which is local to node $G(i)$, is denoted by Q. At the end of the computation node $G(i)$ contains the product $q_i\, q_{i-1} \cdots q_0$. Without loss of generality we assume that $n = 2^d$.

```
PROCEDURE Parallel_Prefix ( n , Q )
    i = G^{-1} ( node_id ( ) )
    FOR k = 1 TO log n DO BEGIN
        IF i ∈ { 0 , .. , n − 1 − 2^{k-1} } THEN
            SEND Q TO PROCESSOR G ( i + 2^{k-1} )
        IF i ∈ { 2^{k-1} , .. , n − 1 } THEN
```

RECEIVE *temp_Q*

$Q = temp\_Q * Q$

END FOR

END PROCEDURE.

Thus we have the following lemma:

Lemma 4.1

The prefixes of n *elements can be computed in* $\log n$ *arithmetic and in* $2\log n - 1$ *communication steps on a hypercube with* n *nodes.*

Proof :

It follows from Lemma 2.2 in Chapter 2 that the first step will cost 1 arithmetic and 1 communication step. The remaining steps cost $\log n - 1$ arithmetic and $2(\log n - 1)$ communication steps. ●

Now we suppose that we have p processors with $p < n$ and $m\,p = n$. Then the prefixes of n elements are computed as follows: we allocate m elements to each processor and perform sequential prefix at each processor to find prefixes of these elements. Then we find prefixes of the p product blocks by performing the parallel prefix algorithm. Processor i sends this product to processor $i+1$ for $0 \le i \le n-2$ and this element is multiplied with each element in the processor except the last one. Initially we allocate the elements $q_{(i+1)m-1}, q_{(i+1)m-2}, \cdots, q_{im}$ to node $G(i)$. These elements, which are local to node $G(i)$, are denoted $Q_m, Q_{m-1}, .., Q_1$. After the sequential prefix at each node we obtain a product block at each node. This result

$$Q_m\, Q_{m-1} \cdots Q_1 = q_{(i+1)m-1}\, q_{(i+1)m-2} \cdots q_{im}$$

also resides in node $G(i)$. At the end of all computations the node $G(i)$ contains the products

$$q_{im} \cdot\cdot q_1\, q_0$$
$$\cdots$$
$$q_{(i+1)m-2} \cdot\cdot q_{im} \cdot\cdot q_1\, q_0$$
$$q_{(i+1)m-1}\, q_{(i+1)m-2} \cdot\cdot q_{im} \cdot\cdot q_1\, q_0$$

The following code shows the required computations:

PROCEDURE Parallel_Prefix $(n, p, Q_1, Q_2, .., Q_m)$ { *limited processor case*; $n = m\,p$ }

$i = G^{-1}(node\_id())$

FOR $k = 2$ TO m DO BEGIN

$Q_k = Q_k * Q_{k-1}$

END FOR

FOR $k = 1$ TO $\log n$ DO BEGIN

IF $i \in \{0, .., n-1-2^{k-1}\}$ THEN

SEND Q_m TO PROCESSOR $G(i + 2^{k-1})$

IF $i \in \{2^{k-1}, .., n-1\}$ THEN

RECEIVE $temp\_Q_m$

$Q_m = temp\_Q_m * Q_m$

END FOR

IF $i \in \{0, .., n-1-2^{k-1}\}$ THEN

SEND Q_m TO PROCESSOR $G(i+1)$

IF $i \in \{ 1, .., n-1 \}$ THEN

RECEIVE $temp\_Q_m$

FOR $k = 1$ TO $m - 1$ DO BEGIN

$Q_k = temp\_Q_m * Q_k$

END FOR

END PROCEDURE.

Lemma 4.2

The prefixes of $n = m\,p$ *elements can be performed in* $2\frac{n}{p} + \log p - 2$ *arithmetic and* $2 \log p$ *communication steps on a hypercube with* p *nodes.*

Proof :

First we perform sequential prefix computation which costs $m - 1$ arithmetic steps. The parallel prefix costs $\log p$ arithmetic and $2 \log p - 1$ communication steps according to Lemma 4.1. The transfer of the last element of each block to the next processor will take 1 communication step. Then we multiply this element with each element in the processor except the last one which will take $m - 1$ arithmetic steps. Thus the total number of arithmetic and communication steps become $2m + \log p - 2$ and $2 \log p$, respectively. ●

In Figure 4.2 we illustrate the limited processor parallel prefix algorithm for the values of $n = 12$ and $p = 4$. Thus it takes $2 \log 4 = 4$ communication steps and $2\frac{12}{4} + \log 4 - 2 = 6$ arithmetic steps to compute prefixes of 12 terms with 4 processors.

For parallel implementation of the SP Algorithm (henceforth called the PP Algorithm) we allocate m matrices to each processor and perform the limited processor parallel prefix

algorithm with these matrices. Considering all four steps of the SP Algorithm for the solution of (4.1) we have the following theorem:

Theorem 4.3

The PP Algorithm solves (4.1) with $n = m\,p$ *in* $35\frac{n}{p} + 20\log p - 29$ *parallel arithmetic and* $13\log p$ *communication steps on a hypercube with* p *nodes.*

Proof :

Step 1 is performed in $3m$ divisions since there are m matrices allocated to each processor.

Step 2 has 3 substeps. In the first we perform sequential prefix at each processor. Because of the special structure of the matrices each matrix multiplication is performed with 6 multiplications and 4 additions. Hence the first substep costs $10(m-1)$ arithmetic operations. In the second substep of Step 2 we perform parallel prefix using these product blocks. We lose some of the structure in the matrices involved and perform matrix multiplication using 12 multiplications and 8 additions. Thus the parallel prefix step will take $20\log p$ arithmetic steps. Since only the first two rows of the matrices need to be communicated, the parallel prefix step will take $6(2\log p - 1)$ communication steps. In the third substep of Step 2 we first send the product block in processor $G(i)$ to processor $G(i+1)$ which will cost 6 communication steps. Then we multiply this element with all the elements in the processor except the last one. This substep costs $20(m-1)$ arithmetic steps since the matrices are multiplied with 12 floating-point multiplications and 8 floating-point additions.

In Step 3 processor $p-1$, which holds the matrix C_{n-1}, calculates x_0 by performing

a single division, and then x_0 is broadcast to all other processors. This operation can be performed in $\log p$ communication steps by embedding a suitable tree of depth $\log p$ [SaSc85a], [SaSc85b]. In Step 4 we calculate all x_i by performing m multiplications and m additions per processor. The total result follows by summing the number of arithmetic operations and communication steps. ●

Step	Arithmetic Complexity	Communication Complexity
1	$3m$	-
2	$30(m-1)+20\log p$	$12\log p$
3	1	$\log p$
4	$2m$	-
Total	$35m+20\log p-29$	$13\log p$

Finally it is interesting to observe that an SIMD system with processor masking capability is adequate for the algorithm although in actual experiments we used the Intel iPSC/d5 which is an MIMD system.

4.5. ESTIMATED SPEED-UP AND EFFICIENCY

The speed-up and efficiency of the PP Algorithm with respect to the LU and the SP Algorithms can be estimated using the arithmetic and communication complexity figures found previously. We have performed experiments, similar to those mentioned in [McVa87], on on the Intel iPSC/d5 hypercube running XENIX 286 R3.4 and iPSC Software R3.1 to measure the time it takes to perform a floating-point operation (τ_{comp}), and the time it takes to transfer a floating-point number to an adjacent node (τ_{comm}). The experiments indicated

that $\tau_{comm} \simeq 1.48$ milliseconds, and if the floating-point operation is taken to be multiplication, addition, or subtraction then $\tau_{comp} \simeq 0.058$ milliseconds. Division takes a little longer (around 0.072 milliseconds). Using these we can estimate the speed-up of the PP Algorithm with respect to the LU and SP Algorithms as

$$S_{PP/LU} = \frac{T_{LU}}{T_{PP}} = \frac{(8n-7)\tau_{comp}}{(35\frac{n}{p} + 20\log p - 29)\tau_{comp} + (13\log p)\tau_{comm}},$$

$$S_{PP/SP} = \frac{T_{SP}}{T_{PP}} = \frac{(15n-11)\tau_{comp}}{(35\frac{n}{p} + 20\log p - 29)\tau_{comp} + (13\log p)\tau_{comm}}.$$

Similarly the efficiency of the PP Algorithm with respect to the LU and the SP Algorithms is found as

$$E_{PP/LU} = \frac{S_{PP/LU}}{p} = \frac{(8n-7)\tau_{comp}}{(35n + 20p\log p - 29p)\tau_{comp} + (13p\log p)\tau_{comm}},$$

$$E_{PP/SP} = \frac{S_{PP/SP}}{p} = \frac{(15n-11)\tau_{comp}}{(35n + 20p\log p - 29p)\tau_{comp} + (13p\log p)\tau_{comm}}.$$

The results are shown in Table 4.1 for the value of $p = 32$ for the values of $\tau_{comp} = 0.058$ and $\tau_{comm} = 1.48$. The efficiency of the PP algorithm with respect to its sequential counterparts is a function of the ratio, $\tau = \frac{\tau_{comm}}{\tau_{comp}}$, for fixed values of n and p. For the Intel cube we have $\tau = 25.51$. Given $n = 8192$ and $p = 32$, we see that $E_{PP/SP}$ takes the value of 0.359. $E_{PP/SP}$ will take values between 0.422 and 0.247 as τ takes values between 1 and 100. This ratio is a crucial parameter in message-passing parallel computers, and its value changes between 2 and 1000 for different hypercubes (see "Gordon Bell on the Future of Computers," *SIAM News*, Vol. 20, No.2, March 1987).

4.6. EXPERIMENTAL RESULTS AND CONCLUSIONS

We have experimented on an Intel iPSC/d5 hypercube system for values of n between 32 and 8192. The LU and SP Algorithms were run on a single node and the PP Algorithm was run on 1, 2, 3, 4, and 5 dimensional subcubes. The initial loading of the data was not taken into account for any of these algorithms. The experiments were done to compute the cubic spline approximation of some random data. The types of tridiagonal matrices that arise in cubic (or higher degree) spline approximation are diagonally dominant and mostly symmetric [AhNW67].

The computation and communication times were measured using the *clock( )* routine at the beginning and end of each program. The timings of the LU, SP, and PP Algorithms are given in Table 2 in milliseconds. Using these data we can compute the measured speed-up and efficiency of the PP Algorithm with respect to its sequential counterparts. These are shown in Table 3 for the value of $p = 32$ (compare Table 1 to Table 3). Also, in Figures 3a, and 3b we show the estimated and measured efficiency of the PP Algorithm with respect to the LU Algorithm as a function of dimension of the cube for values of $n = 4096$ and $n = 8192$, respectively. Similarly, the PP Algorithm is compared to the SP Algorithm in Figures 4a and 4b. The small differences between the estimated and measured values are due to the fact that we assumed all floating-point operations take the same amount of time, and also overhead factors, such as loop control, memory fetch, etc. were not taken into account. The experimental results have shown the proposed algorithm achieves linear speed-up and its efficiency is somewhere between 0.50 and 0.60.

It has been observed that some numerical stability problems can arise in the use of the recursive doubling algorithm for certain classes of problems when the size of the system is

large [DuRo77]. Since memory size on the Intel iPSC/d5 is about 300 kilobytes/node, experimentation was kept to tridiagonal systems of size no more than 8192. In attaching high importance to speed, numerical stability problems involved in the use of parallel algorithms are occasionally ignored. As pointed out in [Mill75], a parallel algorithm may become completely useless if its numerical stability properties are undesirable. Methods for analyzing the numerical stability of parallel algorithms have been developed in [Rons84] and classification schemes have been proposed in [Gao_87] based on the theoretical foundations of *forward error analysis* [Stum81]. Solution of tridiagonal or banded systems has been done using (block) Gaussian elimination, (block) cyclic reduction [John86], and the recursive doubling algorithm. In [Gao_87] Gao has applied the forward error analysis technique to pipelined algorithms for the solution of first and second order recurrences. We are currently implementing parallel algorithms for the solutions of general recurrence relations, tridiagonal, block tridiagonal, and banded linear systems on the Intel hypercube, and investigating their numerical stability properties.

Table 4.1. Estimated speed-up and efficiency for $p = 32$.

n	$S_{PP/LU}$	$S_{PP/SP}$	$E_{PP/LU}$	$E_{PP/SP}$
32	0.14	0.27	0.004	0.008
64	0.28	0.53	0.009	0.017
128	0.54	1.02	0.017	0.032
256	1.02	1.91	0.032	0.060
512	1.79	3.35	0.056	0.105
1024	2.87	5.39	0.090	0.168
2048	4.13	7.74	0.129	0.242
4096	5.28	9.89	0.165	0.309
8192	6.13	11.49	0.192	0.359

Table 4.2. The timings of the LU, SP, and PP Algorithms (in milliseconds).

n	LU	SP	PP $p=2$	PP $p=4$	PP $p=8$	PP $p=16$	PP $p=32$
32	15	40	40	30	25	25	75
64	30	75	80	45	35	60	85
128	60	155	155	85	55	65	90
256	120	315	310	160	95	80	100
512	235	625	615	315	165	125	120
1024	480	1250	1225	620	320	210	150
2048	960	2495	2445	1230	625	370	230
4096	1920	4990	4885	2450	1235	655	400
8192	3840	9990	9775	4895	2455	1260	685

Table 4.3. Measured speed-up and efficiency for $p = 32$.

n	$S_{PP/LU}$	$S_{PP/SP}$	$E_{PP/LU}$	$E_{PP/SP}$
32	0.20	0.53	0.006	0.017
64	0.35	0.88	0.011	0.028
128	0.67	1.72	0.021	0.054
256	1.20	3.15	0.038	0.098
512	1.96	5.21	0.061	0.163
1024	3.20	8.33	0.100	0.260
2048	4.17	10.85	0.130	0.339
4096	4.80	12.47	0.150	0.390
8192	5.61	14.58	0.175	0.456

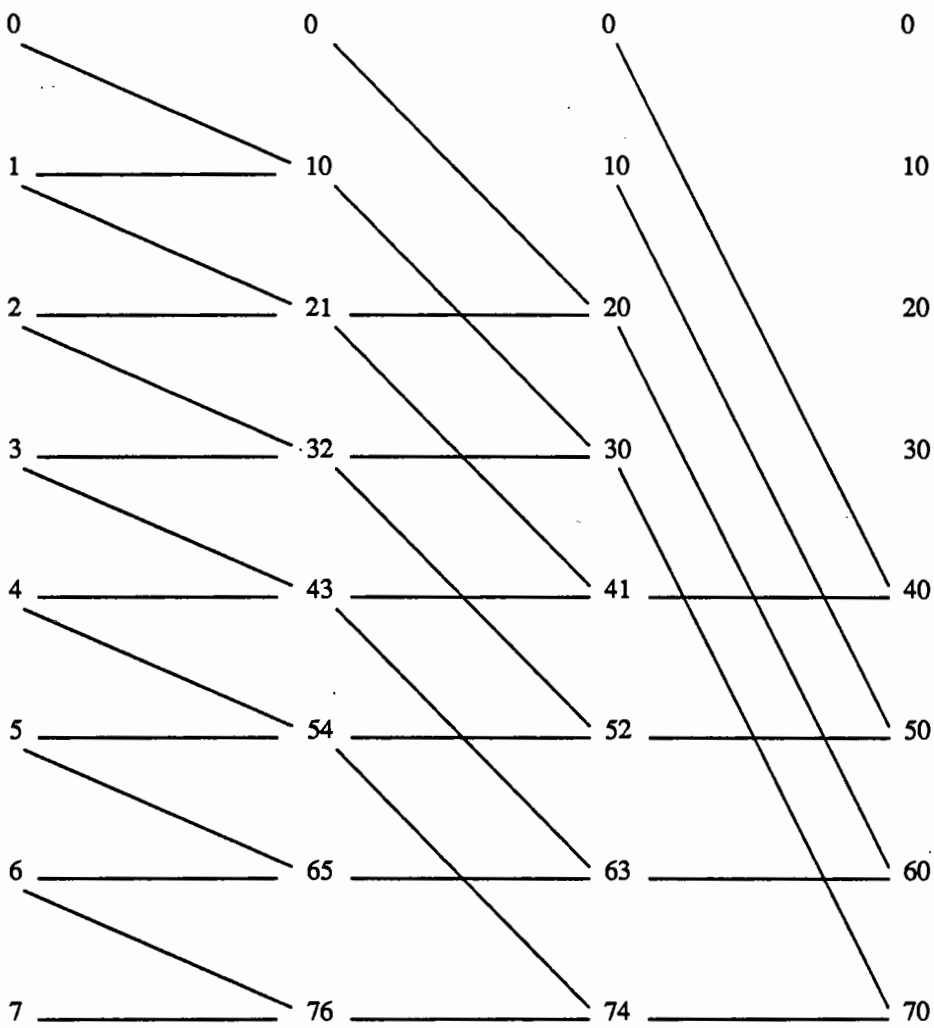


Figure 4.1. The parallel prefix algorithm for $n = 8$.

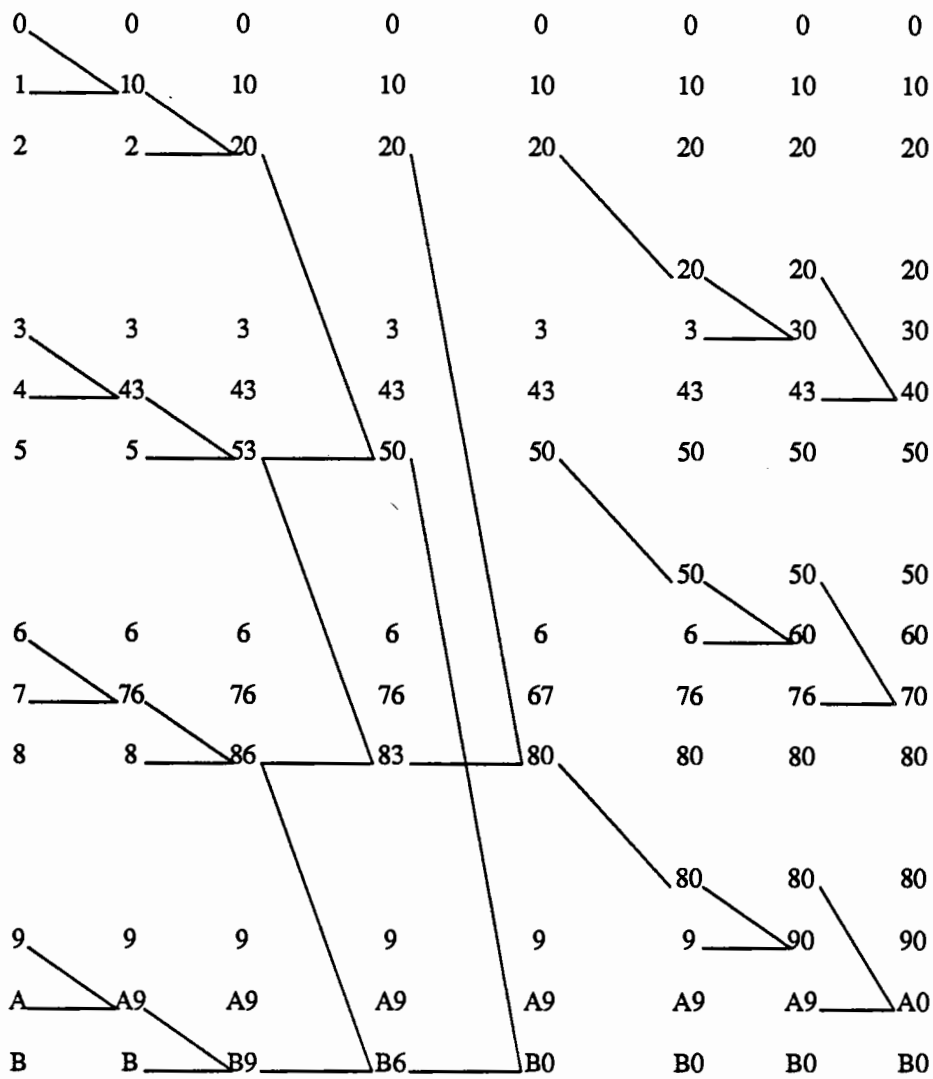


Figure 4.2. The limited processor version of parallel prefix algorithm for $n = 12$ and $p = 4$.

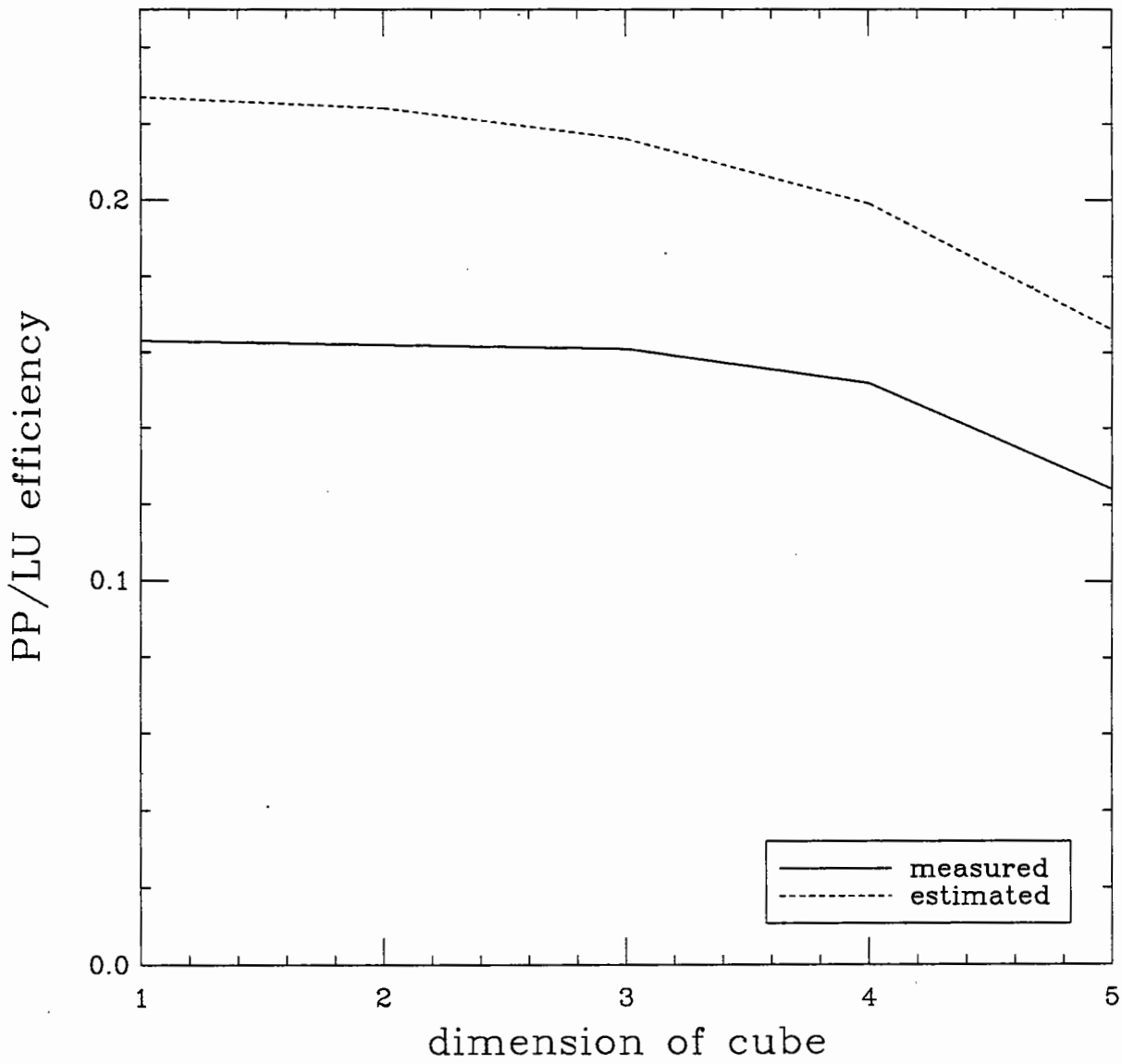


Figure 4.3a. Efficiency of the PP Algorithm with respect to the LU Algorithm as a function of cube dimension for $n = 4096$.

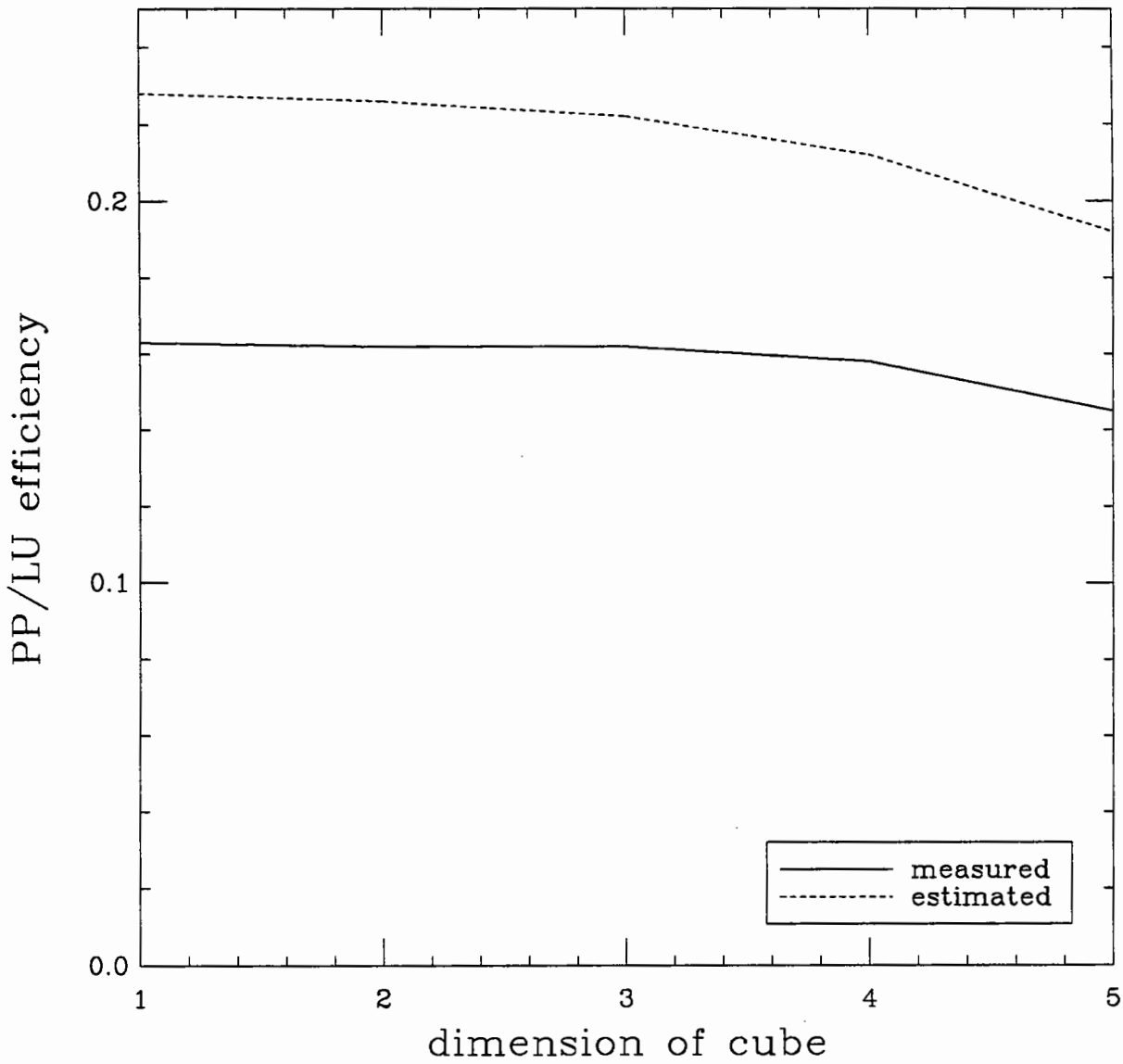


Figure 4.3b. Efficiency of the PP Algorithm with respect to the LU Algorithm as a function of cube dimension for $n = 8192$.

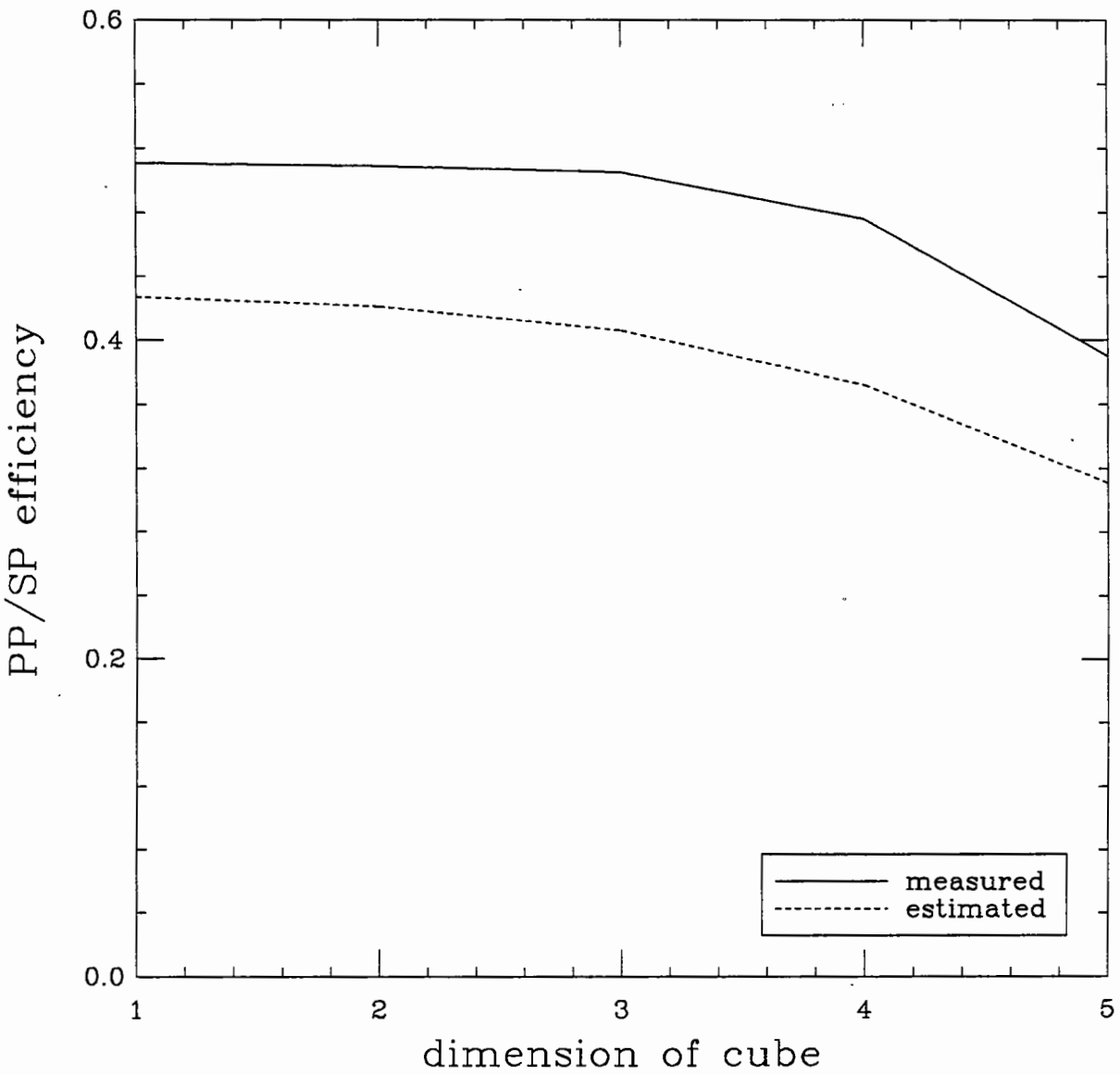


Figure 4.4a. Efficiency of the PP Algorithm with respect to the SP Algorithm as a function of cube dimension for $n = 4096$.

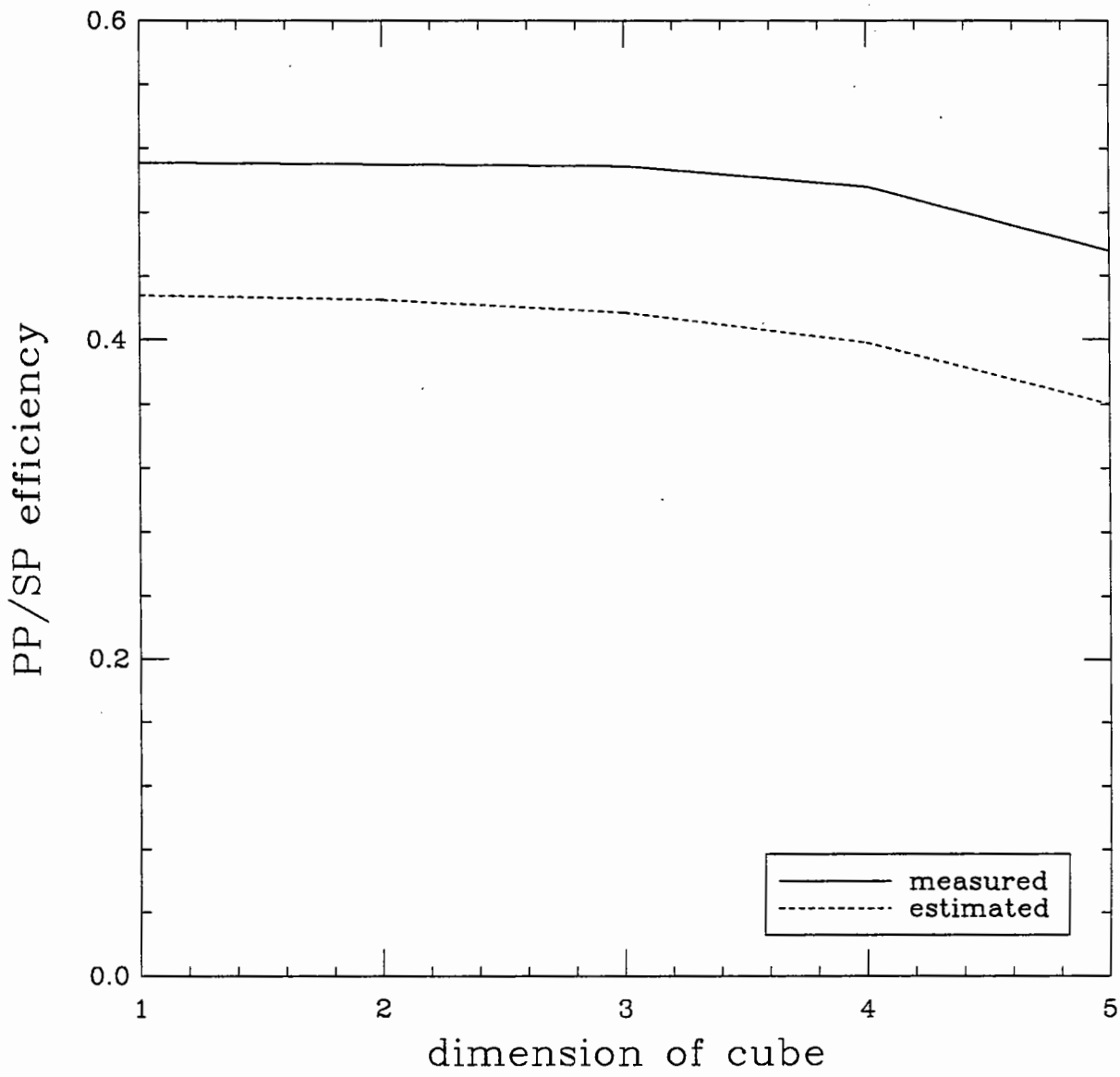


Figure 4.4b. Efficiency of the PP Algorithm with respect to the SP Algorithm as a function of cube dimension for $n = 8192$.

5

Rational Interpolation via Orthogonal Polynomials

A new algorithm for rational interpolation is proposed. Given the data set, the algorithm generates a set of orthogonal polynomials by the classical three-term recurrence relation and then uses Newton interpolation to find the numerator and the denominator polynomials of the rational interpolating function. The number of arithmetic operations required by the algorithm to find a particular rational interpolant is $O(N^2)$, where $N+1$ is the number of data points. A variant of this algorithm that avoids Newton interpolation can be used to construct all rational interpolants using only $O(N^2)$ arithmetic operations. The parallel versions of the algorithms require $N+1$ processors to construct a particular rational interpolant in $O(n \log N)$ arithmetic steps, or all rational interpolants in $O(N \log N)$ arithmetic steps.

5.1. INTRODUCTION

Let $R(m,n)$ be the set of rational functions of the form $r_{m,n}(x)=\frac{p_m(x)}{q_n(x)}$ where $p_m(x)$ and $q_n(x)$ are polynomials of degree m and n respectively. If a set of $N+1=m+n+1$ pairs of points (x_i,f_i) for $0\le i\le N$ is given, then the *rational interpolation* problem is defined as the task of determining a rational function $r_{m,n}(x)\in R(m,n)$ such that

$$r_{m,n}(x_i)=f_i \qquad \text{for} \qquad 0\le i\le N \tag{5.1}$$

where x_i 's are distinct elements. We also assume that f_i 's are the values of a function $f(x)$ at the nodes x_i for all i.

In the case of polynomial interpolation (i.e. $n=0$) it is always possible to construct a unique polynomial satisfying (5.1), but this is not true for rational interpolation. For fixed m and n a rational interpolant that satisfies (5.1) at all points may not exist, even though it may be possible to satisfy (5.1) on a subset of the given point set by means of an $r_{\mu,\nu}(x)\in R(\mu,\nu)$ with $\mu<m$ and $\nu<n$. For this particular pair m and n, the point set becomes a *degenerate configuration.* The points (x_j,f_j) for which

$$r_{\mu,\nu}(x_j)\neq f_j$$

are called the *unattainable* points. Roughly speaking the existence of an unattainable point (x_j,f_j) for $0\le j\le N$ means that x_j is a common zero of the numerator and the denominator polynomials. Thus a degenerate point set contains one or more unattainable points.

A direct algorithm for rational interpolation solves a set of linear equations

$$p_m(x_i)-f_i\,q_n(x_i)=0 \qquad 0\le i\le N \tag{5.2}$$

to compute the coefficients of the polynomials $p_m(x)$ and $q_n(x)$, the oldest method being an elegant elimination process due to Jacobi [Muir60], [Rivl69]. Jacobi's method

yields exact determinantal formulae for the coefficients of the denominator and numerator polynomials. A more recent algorithm by Schneider and Werner [ScWe86] makes use of the barycentric representation of the rational interpolant. Here a set of linear equations of size n is solved by applying Gaussian elimination to compute the coefficients of $q_n(x)$.

Direct methods can be computationally inefficient because the solution of a general set of linear equations may require as much as $O(n^3)$ arithmetic operations. The Jacobi algorithm, on the other hand, results in a Hankel system of linear equations, which in turn can be solved with $O(n^2)$ arithmetic operations.

We remark that alongside the direct algorithms there exist an array of iterative methods to construct $r_{m,n}(x)$, and the reader is referred to [GrHo81] for a review of these algorithms.

In this Chapter, a fast direct algorithm for rational interpolation is proposed. Given the data set (x_i, f_i) for $0 \le i \le N$ the algorithm first generates a sequence $q_0(x), q_1(x), \ldots, q_n(x)$ of orthogonal polynomials on the discrete set $\{x_0, x_1, \ldots, x_N\}$ with respect to certain weights. The values of these polynomials at the nodes x_i for $0 \le i \le N$ are computed by using the classical three-term recurrence relation satisfied by orthogonal polynomials. Once the values of a particular polynomial are known, its coefficients can be computed easily via Newton interpolation. The last orthogonal polynomial $q_n(x)$ generated by this algorithm is the required denominator polynomial of the rational interpolant. Once the denominator polynomial $q_n(x)$ is determined, the construction of the numerator $p_m(x)$ becomes a polynomial interpolation problem. We prove that the number of arithmetic operations of this algorithm to find a particular rational interpolant for given values of m and n is $O(N^2)$.

The coefficients of the orthogonal polynomials can also be found without the application of the Newton interpolation algorithm. This is again achieved by using the three-term recurrence formula. The resulting algorithm requires $O(N^2)$ arithmetic operations to generate the polynomials $q_0(x), q_1(x), \ldots, q_N(x)$. If the same reasoning is applied to the data (x_i, f_i^{-1}) for $0 \le i \le N$, the coefficients as well as the values of the numerator polynomials $p_0(x), p_1(x), \ldots, p_N(x)$ can be computed. We show that in this way all rational interpolants $r_{m,n}(x)$ with $m+n=N$ and $0 \le n \le N$ can be computed using a total of only $O(N^2)$ arithmetic operations provided that $f_i \neq 0$ for all $0 \le i \le N$, and the orthogonal polynomials exist.

5.2. JACOBI RATIONAL INTERPOLATION ALGORITHM

Let $[g(x)]_{0..N}$ denote the *Nth* divided difference of a function $g(x)$ with respect to the node values x_i for $0 \le i \le N$. Put

$$w(x) = (x - x_0)(x - x_1) \ldots (x - x_N) \tag{5.3}$$

with

$$w_i = w'(x_i) = \prod_{\substack{j=0 \\ j \neq i}}^{N} (x_i - x_j) \quad . \tag{5.4}$$

By direct application of the Lagrange interpolation formula one has

$$[g(x)]_{0..N} = \sum_{i=0}^{N} \frac{g_i}{w_i} \tag{5.5}$$

where $g_i = g(x_i)$. Note that (5.5) gives the leading coefficient of the polynomial that takes the values g_i at the node points x_i for $0 \le i \le N$. Clearly, if $g(x)$ is a polynomial of degree strictly less than N then $[g(x)]_{0..N}$ vanishes.

From (5.2) we can write for any $j \ge 0$

$$\frac{x_i^j\, p_m(x_i)}{w_i} = \frac{x_i^j\, f_i\, q_n(x_i)}{w_i} \qquad \text{for } 0 \le i \le N \tag{5.6}$$

and by summing these terms over i we obtain

$$\sum_{i=0}^{N} \frac{x_i^j\, p_m(x_i)}{w_i} = \sum_{i=0}^{N} \frac{x_i^j\, f_i\, q_n(x_i)}{w_i}$$

Using the definition (5.5) this can be simplified as

$$[\, x^j\, p_m(x)\,]_{0..N} = [\, x^j\, f(x)\, q_n(x)\,]_{0..N} \tag{5.7}$$

The left hand side of (5.7) is equal to the *Nth* divided difference of the polynomial $x^j\, p_m(x)$. Thus if j is in the range $0 \le j \le n-1$ we have

$$deg\,(x^j\, p_m(x)) = j + m \le n - 1 + m = N - 1$$

Hence

$$[\, x^j\, p_m(x)\,]_{0..N} = 0 \qquad \text{for} \quad 0 \le j \le n-1 \tag{5.8}$$

and therefore

$$[\, x^j\, f(x)\, q_n(x)\,]_{0..N} = 0 \qquad \text{for} \quad 0 \le j \le n-1 \tag{5.9}$$

The equation (5.9) allows us to compute the coefficients of $q_n(x)$ by solving a set of linear equations. Since there are n equations and $n+1$ unknowns, one of the parameters can be chosen arbitrarily.

If $q_n(x)$ is represented in the standard power basis

$$q_n(x) = \sum_{k=0}^{n} b_k\, x^k$$

then from (5.9) the b_k 's satisfy the following linear system of equations

$$\sum_{i=0}^{N} \frac{x_i^j\, f_i\, q_n(x_i)}{w_i} = \sum_{i=0}^{N} \frac{x_i^j\, f_i}{w_i} \sum_{k=0}^{n} b_k\, x_i^k = 0 \qquad 0 \le j \le n-1$$

which can be put in the form

$$\sum_{k=0}^{n} h_{j+k}\, b_k = 0 \qquad 0 \le j \le n-1 \tag{5.10}$$

where

$$h_s = \sum_{i=0}^{N} \frac{f_i\, x_i^{\,s}}{w_i} \qquad 0 \le s \le 2n-1 \quad . \tag{5.11}$$

The system of equations given in (5.10) is a Hankel system. As indicated before, we have one degree of freedom in choosing the coefficients of $q_n(x)$ in any rational interpolation. From now on we will assume that $q_n(x)$ is a *monic* polynomial of degree n, that is $b_n = 1$. With this choice (5.10) can be rewritten as the matrix equation

$$\begin{bmatrix} h_0 & h_1 & h_2 & \cdot\cdot & h_{n-1} \\ h_1 & h_2 & h_3 & \cdot\cdot & h_n \\ h_2 & h_3 & h_4 & \cdot\cdot & h_{n+1} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ h_{n-1} & h_n & h_{n+1} & \cdot\cdot & h_{2n-2} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \cdot \\ \cdot \\ b_{n-1} \end{bmatrix} = - \begin{bmatrix} h_n \\ h_{n+1} \\ h_{n+2} \\ \cdot \\ \cdot \\ h_{2n-1} \end{bmatrix} \tag{5.12}$$

Therefore,

$$q_n(x) = \frac{1}{\det(H_{n-1})} \det \begin{bmatrix} h_0 & h_1 & \cdot\cdot & h_{n-1} & h_n \\ h_1 & h_2 & \cdot\cdot & h_n & h_{n+1} \\ \cdot & \cdot & \cdot\cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot\cdot & \cdot & \cdot \\ h_{n-1} & h_n & \cdot\cdot & h_{2n-2} & h_{2n-1} \\ 1 & x & \cdot\cdot & x^{n-1} & x^n \end{bmatrix} \tag{5.13}$$

where

$$H_j = \begin{bmatrix} h_0 & h_1 & \cdot\cdot & h_j \\ h_1 & h_2 & \cdot\cdot & h_{j+1} \\ \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot\cdot & \cdot \\ h_j & h_{j+1} & \cdot\cdot & h_{2j} \end{bmatrix}$$

As it stands, the determinantal formula (5.13) is computationally inefficient to construct the denominator polynomial $q_n(x)$. However, Trench [Tren65] has provided a fast algorithm to solve the Hankel system (5.12) directly. If all of the matrices $H_0, H_1, \ldots, H_{n-1}$ are non-singular then the number of arithmetic operations to invert H_{n-1} is proportional to n^2. Thus the system in (5.12) can be solved in $O(n^2)$ arithmetic operations given the non-singularity condition on the principle minors of H_{n-1}.

Note that the existence of the denominator of degree n for $r_{m,n}(x)$ depends only on the nonsingularity of the matrix H_{n-1} by (5.13). The nonsingularity of the all the matrices $H_0, H_1, \ldots, H_{n-1}$ is thus equivalent to the existence of all of the denominator polynomials $q_0(x), q_1(x), \ldots, q_n(x)$. It turns out that this is precisely the assumption required on the matrices $H_0, H_1, \ldots, H_{n-1}$ for the construction of orthogonal polynomials outlined in the next section.

Once the coefficients of $q_n(x)$ are found, we can evaluate $q_n(x)$ at all of the node points to check if the data set is degenerate for the given values of m and n and thus locate unattainable points. The coefficients of $p_m(x)$ can then be computed by performing a polynomial interpolation for the data set $(x_i, f_i\, q_n(x_i))$ for $0 \le i \le m$.

To summarize, Jacobi algorithm for rational interpolation can be carried out in the following manner:

Jacobi Algorithm

Input: (x_i, f_i) for $0 \le i \le N$.

Output: Coefficients of $p_m(x)$ and $q_n(x)$ of the rational interpolant $r_{m,n}(x)$

Step 1. Compute w_i for $0 \le i \le N$ using (5.4).

Step 2. Compute h_s for $0 \le s \le 2n-1$ using (5.11).

Step 3. Solve the resulting Hankel system by using the Trench algorithm to find the coefficients of $q_n(x)$ in the standard form.

Step 4. Check whether or not the data set is degenerate for the given values of m and n, i.e. check if $q_n(x_i)=0$ for any $0 \le i \le N$ by evaluating $q_n(x)$ at all x_i and thus locating the unattainable points.

Step 5. Interpolate the data set $(x_i, f_i\, q_n(x_i))$ for $0 \le i \le m$ using the Newton interpolation algorithm to find $p_m(x)$ in the Newton form.

Thus we have

Theorem 5.1

Given the data set (x_i, f_i) *for* $0 \le i \le N$, *the Jacobi algorithm computes the coefficients of the rational interpolant* $r_{m,n}(x)$ *which satisfies (5.1) using* $O(N^2)$ *arithmetic operations.*

Proof :

We will count the number of arithmetic operations at each step of the algorithm.

In Step 1, w_i can be calculated with N subtractions and $N-1$ multiplications for any fixed i. For all $i = 0, 1, \ldots, N$ this clearly takes $O(N^2)$ arithmetic operations. In Step 2 first x^s is computed for all $s = 0, 1, \ldots, 2n-1$ with $2n-2$ multiplications. To compute h_s we perform $N+1$ multiplications, $N+1$ divisions and N additions. For all $s = 0, 1, \ldots, 2n-1$ this takes $O(Nn)$ operations. Thus Step 2 takes $O(N^2)$ arith-

metic operations.

The solution of the Hankel system takes $O(n^2)$ arithmetic operations as stated earlier. Step 4 consists of evaluating a polynomial of degree n at $N+1$ points. Hence up to Step 5, the number of operations required is no more than $O(N^2)$. Finally, in Step 5 we apply the Newton interpolation to construct the polynomial $p_m(x)$ which takes $\frac{3}{2}m(m+1) = O(m^2) = O(N^2)$ operations [Hild56]. Thus the number of operations of the Jacobi algorithm add up to $O(N^2)$. ●

It should be noted that Step 3 of the Jacobi algorithm can be carried out with $O(n^2)$ arithmetic operations without the assumption that the principal minors of H_{n-1} be non-zero (except $\det(H_{n-1})$) by applying an algorithm of Rissanen to solve Hankel and Toeplitz systems [Riss74]. Nevertheless, if one is interested in constructing all $r_{m,n}(x)$ for $m+n=N$ and $0 \le n \le N$ we see that the construction of all of the denominator polynomials $q_0(x), q_1(x), \ldots, q_n(x)$ would require the solution of n such systems, resulting in an $O(n^3)$ algorithm. Given the nonsingularity assumption on the matrices $H_0, H_1, \ldots, H_{n-1}$, we can compute *all* of the polynomials $q_0(x), q_1(x), \ldots, q_n(x)$ with a total of $O(n^2)$ arithmetic operations by using orthogonal polynomials. This is the subject matter of the next section.

Alternatively, the process that yielded the coefficients of $q_n(x)$ can be repeated with the data set (x_i, f_i^{-1}) for $0 \le i \le N$ to construct the coefficients of $p_m(x)$. To this end, note that (5.2) can be written as

$$q_n(x_i) - \frac{1}{f_i} p_m(x_i) = 0 \qquad 0 \le i \le N$$

provided $f_i \neq 0$ for all $0 \le i \le N$. By representing $p_m(x)$ in its standard form

$$p_m(x) = \sum_{k=0}^{m} a_k x^k$$

and applying the same reasoning as in the equations (5.6) through (5.10) we obtain

$$\sum_{k=0}^{m} h'_{j+k} a_k = 0 \qquad 0 \le j \le m-1 \tag{5.14}$$

where

$$h'_s = \sum_{i=0}^{N} \frac{x_i^s}{f_i w_i} \qquad 0 \le s \le 2m-1 \tag{5.15}$$

Again the system of equations in (5.14) becomes a Hankel system. By assuming $p_m(x)$ to be a monic polynomial for the moment, we obtain a system of linear equations of size m with the unknowns $a_0, a_1, \ldots, a_{m-1}$ similar to (5.12).

$$\begin{bmatrix} h'_0 & h'_1 & h'_2 & \cdots & h'_{m-1} \\ h'_1 & h'_2 & h'_3 & \cdots & h'_m \\ h'_2 & h'_3 & h'_4 & \cdots & h'_{m+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h'_{m-1} & h'_m & h'_{m+1} & \cdots & h'_{2m-2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{m-1} \end{bmatrix} = \begin{bmatrix} h'_m \\ h'_{m+1} \\ h'_{m+2} \\ \vdots \\ h'_{2m-1} \end{bmatrix} \tag{5.16}$$

Since $p_m(x)$ and $q_n(x)$ cannot be forced to be monic at the same time, we need to multiply one of these polynomials with a nonzero constant to make them consistent with (5.1). Thus if $q_n(x)$ and $p_m(x)$ are monic polynomials as solutions of the equations (5.12) and (5.16) the corresponding rational interpolant is simply

$$r_{m,n}(x) = \frac{a_m p_m(x)}{q_n(x)} \tag{5.17}$$

where

$$a_m = \frac{f_0 q_n(x_0)}{p_m(x_0)}$$

assuming that (x_0, f_0) is an attainable point. If we put

$$H'_j = \begin{bmatrix} h'_0 & h'_1 & \cdot\cdot & h'_j \\ h'_1 & h'_2 & \cdot\cdot & h'_{j+1} \\ \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot\cdot & \cdot \\ h'_j & h'_{j+1} & \cdot\cdot & h'_{2j} \end{bmatrix} \tag{5.18}$$

and use the Trench algorithm to solve (5.16), we see that the matrices $H'_0, H'_1, \ldots, H'_{m-1}$ should be nonsingular.

Thus the Newton interpolation can be avoided in Step 5 of the Jacobi Algorithm by solving another Hankel system to determine $p_m(x)$. Since the number of arithmetic operations to solve a Hankel system or to perform a Newton interpolation is the same, $O(m^2)$, we see that this approach doesn't provide any gains in terms of number of arithmetic operations. However we will make use of this idea in section 4, to devise an algorithm to construct all rational interpolants using only $O(N^2)$ arithmetic operations.

Jacobi algorithm may have undesirable numerical properties because of the stability issues involved in the way the divided differences, (i.e. the individual h_s 's) are calculated. The algorithm proposed by Schneider and Werner in [ScWe86] makes use of the barycentric representation of the rational interpolant, and seems to have better numerical properties. On the other hand, there is a trade off in terms of time complexity, since the linear system of equations that arise in this particular algorithm has no additional structure to facilitate a reduction in the overall time complexity. In particular, the use of Gaussian elimination to solve a general system of linear equations, as proposed, requires $O(n^3)$ operations. Thus the number of arithmetic operations required by the Schneider-Werner algorithm to compute a particular rational interpolant is $O(N^3)$ because of this apparent bottleneck as opposed to $O(N^2)$ achieved by the Jacobi algorithm.

5.3. RATIONAL INTERPOLATION USING ORTHOGONAL POLYNOMIALS

Our point of departure will be the Jacobi algorithm and we will show that it is possible to compute the denominator polynomial $q_n(x)$ without solving a system of linear equations.

We start by noting that any polynomial of degree j in x_i can be used as a multiplier in Equation (5.6). It follows that (5.7) and (5.8) hold for an arbitrary polynomial $t_j(x)$ degree j in place of x^j. Hence we have

$$[\, t_j(x) f(x) q_n(x) \,]_{0..N} = \sum_{i=0}^{N} \frac{f_i}{w_i} t_j(x_i) q_n(x_i) = 0 \qquad (5.19)$$

for $0 \le j \le n-1$. Set $c_i = \frac{f_i}{w_i}$ for $0 \le i \le N$ and define a discrete symmetric bilinear form $<.,.>$ on the space of polynomials of degree less than or equal to n by setting

$$< t_j(x), t_k(x) > := \sum_{i=0}^{N} c_i\, t_j(x_i)\, t_k(x_i) \quad . \qquad (5.20)$$

Using (5.20), the linear system (5.19) can be written as

$$\sum_{i=0}^{N} c_i\, t_j(x_i) q_n(x_i) = < t_j(x), q_n(x) > = 0 \qquad 0 \le j \le n-1 \quad . \qquad (5.21)$$

Note that the bilinear form (5.20) does not necessarily define an inner product since it is possible to have $< t_j(x), t_j(x) > \le 0$. For our purposes, however, it suffices to assume non-degeneracy, that is $< t_j(x), t_j(x) > \neq 0$ for $0 \le j \le n$. If we require the set $\{ t_0(x), t_1(x), \ldots, t_n(x) \}$ to be orthogonal with respect to this discrete bilinear form, then the non-degeneracy condition together with (5.21) imply that $q_j(x)$ is a constant multiple of $t_j(x)$ for $j = 0, 1, \ldots, n$. In particular the denominator polynomials $q_0(x), q_1(x), \ldots, q_n(x)$ are orthogonal with respect to (5.20).

The nondegeneracy of the the bilinear form (5.20) is in turn guaranteed by the non-singularity of the matrices $H_0, H_1, \ldots, H_{n-1}$.

Theorem 5.2

There exists a sequence of orthogonal polynomials $\{ t_0(x), t_1(x), \ldots, t_n(x) \}$ *on the set* $\{ x_0, x_1, \ldots, x_N \}$ *with respect to the symmetric bilinear form*

$$< t_j(x), t_k(x) > = \sum_{i=0}^{N} \frac{f_i}{w_i} t_j(x_i)\, t_k(x_i)$$

if and only if the matrices $H_0, H_1, \ldots, H_{n-1}$ *are all nonsingular.*

Proof :

From the general theory of orthogonal polynomials [Davi75] the polynomial $t_j(x)$ is given explicitly (up to a constant multiple) by the determinant

$$t_j(x) = \det \begin{bmatrix} <1,1> & <1,x> & \ldots & <1,x^j> \\ <x,1> & <x,x> & \ldots & <x,x^j> \\ \vdots & \vdots & \ddots & \vdots \\ <x^{j-1},1> & <x^{j-1},x> & \ldots & <x^{j-1},x^j> \\ 1 & x & \ldots & x^j \end{bmatrix} \tag{5.22}$$

Note that

$$< x^k, x^l > = \sum_{i=0}^{N} \frac{f_i}{w_i} x_i^{k+l} = h_{k+l} \quad .$$

Thus the coefficient of x^j in $t_j(x)$ is precisely the determinant of H_{j-1}, which has to be nonzero. ●

Since we have assumed that $q_n(x)$ is monic, by making use of (5.22) we arrive at Jacobi's explicit formula for the denominator polynomials given in (5.13).

The conventional method for generating orthogonal polynomials is the Gram-Schmidt orthogonalization process. However in our case, this approach would require $O(n^2)$ inner product operations to generate $q_n(x)$. A more efficient technique is to use the three-term recurrence relation for orthogonal polynomials to generate the values of the polynomials $q_j(x)$ directly. With this approach, the total number of inner-product operations required to compute the values $q_0(x_i), q_1(x_i), \ldots, q_{n-1}(x_i)$ for any given point x_i becomes only $O(n)$ [Fors57], [RaRa78], [Szeg59].

More precisely, let $\{t_0(x), t_1(x), \ldots, t_n(x)\}$ be a set of polynomials satisfying the orthogonality relationship with respect to the weights c_i and the sequence of data points x_i for $0 \le i \le N$. It is well known that

$$t_{-1}(x) = 0, \quad t_0(x) = 1$$
$$t_{j+1}(x) = (x - \alpha_j)\, t_j(x) - \beta_j\, t_{j-1}(x) \qquad j = 0, 1, \ldots \tag{5.23}$$

where α_j and β_j are constants determined as

$$\alpha_j = \frac{< x\, t_j(x), t_j(x) >}{< t_j(x), t_j(x) >} \qquad \beta_j = \frac{< t_j(x), t_j(x) >}{< t_{j-1}(x), t_{j-1}(x) >} \tag{5.24}$$

Note that the relation (5.23) generates monic orthogonal polynomials, and requires only the nondegeneracy of the underlying symmetric bilinear form.

Let $T = [T_{ji}]$ be the $(N+1) \times (N+1)$ matrix where

$$T_{ji} = t_j(x_i) \qquad 0 \le i, j \le N \quad . \tag{5.25}$$

In other words, the *jth* row of T consists of the vector of values of the *jth* orthogonal polynomial $t_j(x)$ at the nodes $x_0, x_1, \ldots, x_N$. The following procedure generates the first $n+1$ rows of the matrix T using three-term recursion (TTR) in (5.23).

Procedure TTR

Input: n and (x_i, f_i) for $0 \le i \le N$

Output: $T = [T_{ji}]$ for $0 \le j \le n$ and $0 \le i \le N$

Step 1. Compute w_i and c_i for $0 \le i \le N$ using

$$w_i = \prod_{\substack{j=0 \\ j \neq i}}^{N} (x_i - x_j) \quad \text{and} \quad c_i = \frac{f_i}{w_i} \quad .$$

Step 2. Set $T_{0i} = 1$ for $0 \le i \le N$, $\beta_0 = 0$, and compute

$$\gamma_0 = \sum_{i=0}^{N} c_i x_i \quad , \quad \theta_0 = \sum_{i=0}^{N} c_i \quad , \quad \text{and} \quad \alpha_0 = \frac{\gamma_0}{\theta_0} \quad .$$

Step 3. Set $T_{1i} = x_i - \alpha_0$ for $0 \le i \le N$, and compute

$$\gamma_1 = \sum_{i=0}^{N} c_i x_i T_{1i}^2 \quad , \quad \theta_1 = \sum_{i=0}^{N} c_i T_{1i}^2 \quad , \quad \text{and} \quad \alpha_1 = \frac{\gamma_1}{\theta_1} \quad , \quad \beta_1 = \frac{\theta_1}{\theta_0} \quad .$$

Step 4. For $1 \le j \le n-1$ compute

$$T_{j+1,i} = (x_i - \alpha_j) T_{ji} - \beta_j T_{j-1,i} \qquad 0 \le i \le N$$

$$\gamma_{j+1} = \sum_{i=0}^{N} c_i x_i T_{j+1,i}^2 \quad , \quad \theta_{j+1} = \sum_{i=0}^{N} c_i T_{j+1,i}^2$$

$$\alpha_{j+1} = \frac{\gamma_{j+1}}{\theta_{j+1}} \quad , \quad \beta_{j+1} = \frac{\theta_{j+1}}{\theta_j} \quad .$$

We will denote by $T := \mathrm{TTR}(n, x_i, f_i)$ the $(n+1)\times(N+1)$ matrix produced by TTR from the input data n and (x_i, f_i), $0 \le i \le N$.

Lemma 5.1

Given the data set (x_i, f_i) *for* $0 \le i \le N$, *and* $n \le N$, *the Procedure TTR computes the first* $n+1$ *rows of the matrix* T *using*

$$2N^2 + 9Nn + 2N + 9n - 1 = O(N^2) \tag{5.26}$$

arithmetic operations. In particular, all rows of T *can be computed with*

$$11N^2 + 11N - 1 = O(N^2) \tag{5.27}$$

arithmetic operations.

Proof :

In Step 1 the computation of w_i takes $2N-1$ operations for a particular value of i. All $0 \le i \le N$ takes $(N+1)(2N-1)$ operations. The c_i 's can be computed in $N+1$ steps. Hence Step 1 of the procedure takes $2N^2+2N$ arithmetic operations. The computation of α_0 in Step 2 takes $3N+2$ arithmetic operations. In Step 3 T_{1i} and T_{1i}^2 are computed with $2(N+1)$ arithmetic operations. Then to compute α_1 and β_1 we need to perform $4N+4$ arithmetic operations. Thus Step 3 takes $6N+6$ operations. In Step 4 for a particular value of j the computation of $T_{j+1,i}$ and $T_{j+1,i}^2$ take $4N+4$ arithmetic operations. Then to compute α_{j+1} and β_{j+1} $5N+5$ operations is needed. For all $1 \le j \le n-1$ Step 4 takes $(n-1)(9N+9)$ operations. The result in (5.26) thus follows. To find (5.27) we replace n with N in (5.26). ●

We are now in a position to describe the rational interpolation algorithm via orthogonal polynomials and to prove that rational interpolation can be done using $O(N^2)$ arithmetic operations. The following algorithm first generates the first $n+1$ rows of the matrix $Q = [Q_{ji}]$ where $Q_{ji} = q_j(x_i)$ and then applies the Newton interpolation algorithm to find $q_n(x)$ and $p_m(x)$.

Algorithm 1

Input: (x_i, f_i) for $0 \le i \le N$

Output: Coefficients of $p_m(x)$ and $q_n(x)$ of the rational interpolant $r_{m,n}(x)$

Step 1. $Q := \mathrm{TTR}(n, x_i, f_i)$

Step 2. Interpolate the data set (x_i, Q_{ni}) for $0 \le i \le n$ to compute the coefficients of $q_n(x)$ in the Newton form by using the Newton interpolation algorithm.

Step 3. Compute $f_i Q_{ni}$ for all $0 \le i \le N$ and then interpolate the data set $(x_i, f_i Q_{ni})$ for $0 \le i \le m$ to compute the coefficients of $p_m(x)$ in the Newton form by using the Newton interpolation algorithm.

Theorem 5.3

Given the data set (x_i, f_i) for $0 \le i \le N$, Algorithm 1 computes the coefficients of the rational interpolant $r_{m,n}(x)$ which satisfies (5.1) using $O(N^2)$ arithmetic operations.

Proof :

By Lemma 5.1 Step 1 takes $2N^2 + 9Nn + 2N + 9n - 1$ arithmetic operations. Step 2 is an interpolation process with $n+1$ points. This requires $\frac{3}{2} n(n+1)$ operations [Hild56]. Similarly in Step 3 first we compute $f_i Q_{ni}$ for all $0 \le i \le m$ and then perform a polynomial interpolation with $m+1$ points hence $m + 1 + \frac{3}{2} m(m+1)$ arithmetic operations need to be performed in this step. If we sum the number operations at each step we conclude that Algorithm 1 takes

$$2N^2+\frac{3}{2}(n^2+m^2)+9N\,n+2N+\frac{21}{2}n+\frac{5}{2}m = O(N^2) \qquad (5.28)$$

arithmetic operations. ●

One remarkable property of Algorithm 1 is that it makes it trivial to check whether the given data set is degenerate for any values of m and n with $m+n=N$. To check for degeneracy, we compute the values of Q_{ji} for all $0 \le j \le N$ in Step 1. This implies that the input set for the Procedure TTR is N and (x_i, f_i) for $0 \le i \le N$. The number of arithmetic operations to compute all rows Q is $12N^2+12N-1$ as given in (5.27). Thus the total number of arithmetic operations of Algorithm 1 increases slightly to

$$12N^2+\frac{3}{2}(n^2+m^2)+12N+\frac{3}{2}n+\frac{5}{2}m \qquad (5.29)$$

which is still $O(N^2)$. This allows us to select a particular value of m and n for which the data set is not degenerate while keeping the total number of operations within $O(N^2)$.

5.4. FAST COMPUTATION OF ALL RATIONAL INTERPOLANTS

Note that it is possible to compute the coefficients of the polynomial $q_n(x)$ recursively using the three-term recurrence formula in (5.23) and thus avoid the Newton interpolation in Step 2 of Algorithm 1. This can be done by applying the three term recursion directly to the coefficients of the denominator polynomials. More precisely, let $B=[B_{jk}]$ be the $(N+1)\times(N+1)$ matrix in which the jth row consists of the coefficients of the polynomial $q_j(x)$, i.e.

$$q_j(x)=\sum_{k=0}^{j} B_{jk}\,x^k \quad . \qquad (5.30)$$

Then (5.23) defines a recursion on the elements of B

$$B_{j+1,k}=B_{j,k-1}-\alpha_j B_{jk}-\beta_j B_{j-1,k} \qquad 0 \le j \le k \le N \qquad (5.31)$$

with the boundary conditions

$$B_{00}=1 \ , \ B_{0k}=0 \ \text{ for } \ 1\le k\le N \ , \ B_{j,-1}=0 \ \text{ for } \ 0\le j\le N$$
$$B_{-1,k}=0 \ \text{ for } \ 0\le k\le N \ , \ B_{jk}=0 \ \text{ for } \ 0\le j<k\le N \ . \qquad (5.32)$$

Thus using the this recursion formula we can generate the coefficients of the polynomials $q_j(x)$ for $0\le j\le n$. The values of the polynomials $q_j(x)$ at the node points x_i (i.e. Q_{ji}) are computed at each step to calculate α_j and β_j, but this also helps to locate the unattainable points if the point set happens to degenerate for the particular values of m and n.

If only one rational interpolant is needed then, in Step 2 of Algorithm 1, the choice between the Newton interpolation algorithm or the application of recurrence formula in (5.31) is somewhat an arbitrary one, since both algorithms will require $O(n^2)$ arithmetic operations. For the Newton algorithm, however, the constant in front of the order is smaller.

More importantly the generation of the coefficients of the polynomials $q_j(x)$ in $O(n^2)$ arithmetic operations suggests a drastic cut-down on the number of arithmetic operations when all rational interpolants $r_{m,n}(x)$ for $0\le n\le N$ with $m=N-n$ are computed. We note that for $0\le j\le N$ the coefficients of the polynomials $p_j(x)$ can also be computed similarly by applying the recursion in (5.31) to the data (x_i, f_i^{-1}) for $0\le i\le N$ as we already remarked at the end of section 2. This is possible provided $f_i\neq 0$ for $0\le i\le N$ and $H'_0, H'_1, \ldots, H'_{N-1}$ are nonsingular. This given, define $A=[A_{jk}]$ and $P=[P_{ij}]$ to be $(N+1)\times(N+1)$ matrices where the jth row of A contains the coefficients of the polynomial $p_j(x)$

$$p_j(x)=\sum_{k=0}^{j} A_{jk}\,x^k \qquad (5.33)$$

and jth row of P contains the values of the polynomial $p_j(x)$ at the nodes x_i

$$P_{ji}=p_j(x_i) \qquad 0\le i,j\le N$$

similar to the matrices B and Q .

The following algorithm first generates the values and the coefficients of the polynomials $q_j(x)$ for all $0 \le j \le N$. The algorithm then proceeds to generate the values and the coefficients of the polynomials $p_j(x)$ for $0 \le j \le N$ by applying the same technique to the data (x_i, f_i^{-1}) for $0 \le i \le N$.

Algorithm 2

Input: (x_i, f_i) for $0 \le i \le N$

Output: Coefficients of $p_m(x)$ and $q_n(x)$ for $0 \le n \le N$ and $m = N - n$

Step 1. $Q := \mathrm{TTR}(N, x_i, f_i)$

Step 2. Set $B_{00} = 1$ and $B_{0k} = 0$ for $1 \le k \le N$ and $B_{10} = -\alpha_0$, $B_{11} = 1$ and $B_{1k} = 0$ for $2 \le k \le N$. For all $1 \le k \le j \le N-1$ compute

$$B_{j+1,k} = B_{j,k-1} - \alpha_j B_{jk} - \beta_j B_{j-1,k}$$

Step 3. Compute f_i^{-1} for $0 \le i \le N$ and $P := \mathrm{TTR}(N, x_i, f_i^{-1})$

Step 4. Set $A_{00} = 1$ and $A_{0k} = 0$ for $1 \le k \le N$, and $A_{10} = -\alpha_0$, $A_{11} = 1$ and $A_{1k} = 0$ for $2 \le k \le N$. For all $1 \le k \le j \le N-1$ compute

$$A_{j+1,k} = A_{j,k-1} - \alpha_j A_{jk} - \beta_j A_{j-1,k}$$

Step 5. Update the coefficients of $p_j(x)$ according to (5.17)

$$A_{jk} = f_0 \frac{Q_{N-j,0}}{P_{j0}} A_{jk} \qquad 0 \le k \le j \le N \quad .$$

At the end of Algorithm 2, the coefficients of the polynomials $p_m(x)$ and $q_n(x)$ are A_{mk} and B_{nk} respectively for $0 \le k \le N$. Note that $A_{mk} = 0$ for $k > m$ and $B_{nk} = 0$ for

$n > k$.

Theorem 5.4

Algorithm 2 computes all rational interpolants $r_{m,n}(x)$ *for* $n+m=N$ *and* $n=0,1,\ldots,N$ *using* $O(N^2)$ *arithmetic operations provided the matrices* $H_0, H_1, \ldots, H_{N-1}$ *and* $H'_0, H'_1, \ldots, H'_{N-1}$ *are nonsingular.*

Proof :

As we showed in Theorem 5.2, the orthogonal polynomials $q_0(x), q_1(x), \ldots, q_N(x)$ can be constructed iff the matrices $H_0, H_1, \ldots, H_{N-1}$ are all nonsingular. By applying the same technique to the data (x_i, f_i^{-1}) for $0 \le i \le N$ we conclude that the polynomials $p_0(x), p_1(x), \ldots, p_N(x)$ can be constructed if the matrices $H'_0, H'_1, \ldots, H'_{N-1}$ are all nonsingular and if $f_i \neq 0$ for all $0 \le i \le N$.

To compute the number of arithmetic operations required for Algorithm 2 we count the operations at each step. Step 1 takes $11N^2 + 11N - 1$ arithmetic operations due to Lemma 1. Also Step 3 will take $N + 1 + 11N^2 + 11N - 1$ arithmetic operations. For each of Step 2 and 4 notice that the recursion in (4.2) requires $\sum_{j=1}^{N-1} \sum_{k=0}^{j} 4 = 2N^2 + 2N - 4$ arithmetic operations. Also in Step 5 we first compute $\frac{f_0 Q_{N-j,0}}{P_{j0}}$ for $0 \le j \le N$ using $N+1$ operations, then the lower triangular part of the matrix A is updated which results in $\sum_{j=0}^{N} \sum_{k=0}^{j} 1 = \frac{1}{2}N^2 + \frac{5}{2}N + 2$ arithmetic operations. Thus Algorithm 2 requires a total of

$$\frac{53}{2}N^2 + \frac{59}{2}N - 7 = O(N^2) \tag{5.34}$$

arithmetic operations to compute all rational interpolants. ●

5.5. EXAMPLES

The algorithms have been implemented on a VAX-11/780 computer running Unix 4.2 BSD using *Pascal* programming language. Even though we have not yet conducted a detailed experimental study of the numerical properties of these algorithms, we present computer generated solutions to two simple interpolation problems.

Example 1 (Algorithm 1)

Given $f(x)=|x|$, find $r_{2,2}(x)$ which interpolates $f(x)$ at the node points $-1, -0.5, 0, 0.5, 1$.

We apply Step 1 of Algorithm 1 to the data and obtain the following Q matrix of size (5×5)

$$Q = \begin{bmatrix} 1.00 & 1.00 & 1.00 & 1.00 & 1.00 \\ -1.00 & -0.50 & 0.00 & 0.50 & 1.00 \\ 1.50 & 0.75 & 0.50 & 0.75 & 0.75 \\ 0.75 & 0.75 & 0.00 & -0.75 & -0.75 \\ 0.00 & 0.00 & 0.25 & 0.00 & 0.00 \end{bmatrix}$$

The rows of the matrix Q are the values of the denominator polynomials $q_j(x)$ at the nodes x_i for $0\le i, j\le 4$. As mentioned before, if $Q_{ni}=0$ for an i then the point (x_i, f_i) is an unattainable point for the (m, n) pair. An inspection of all entries in Q gives the following result:

m	n	unattainable points
4	0	none
3	1	(0,0)
2	2	none
1	3	(0,0)
0	4	all points except (0,0)

The rational interpolant $r_{2,2}(x)$ interpolates $f(x)$ at all points. In order to determine $q_2(x)$ we use the Newton interpolation algorithm. The values of the polynomial $q_2(x)$ at the node points are seen from the third row:

$q_2(-1)=1.50$, $q_2(-0.5)=0.75$, $q_2(0)=0.50$, $q_2(0.5)=0.75$, $q_2(1)=0.75$

Since 3 points are necessary and sufficient to determine the coefficients of $q_2(x)$ we choose the first three points: $(-1, 1.50)$ $(-0.5, 0.75)$, $(0.0, 0.5)$, and find $q_2(x)=x^2+0.5$. In order to determine the coefficients of the numerator polynomial, first we compute

$$f_0 Q_{20}=1.50 \ , \ f_1 Q_{21}=0.375 \ , \ f_2 Q_{22}=0.00 \ .$$

and apply Newton interpolation to the data $(-1, 1.50)$, $(-0.5, 0.375)$, $(0, 0.00)$, and find the numerator polynomial as $p_2(x)=1.50x^2$. Thus

$$r_{2,2}(x)=\frac{p_2(x)}{q_2(x)}=\frac{1.50x^2}{x^2+0.5}$$

Here we note that since $f_2=0$, Algorithm 2 cannot be applied to find all rational interpolants for this problem.

Example 2 (Algorithm 2)

Given $f(x)=2^x$, find all rational interpolants of $f(x)$ at the node points $-2, -1, 0, 1, 2$.

The application of Step 1 and Step 2 of Algorithm 2 to this data set produces the following Q and B matrices:

$$Q = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ -8 & -7 & -6 & -5 & -4 \\ 48 & 36 & 26 & 18 & 12 \\ -192 & -120 & -72 & -42 & -24 \\ 384 & 192 & 96 & 48 & 24 \end{bmatrix} \quad , \quad B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -6 & 1 & 0 & 0 & 0 \\ 26 & -9 & 1 & 0 & 0 \\ -72 & 38 & -9 & 1 & 0 \\ 96 & -66 & 23 & -6 & 1 \end{bmatrix}$$

Thus we observe that for this interpolation problem all points are attainable for all (m, n) pairs. The elements of B are the coefficients of the denominator polynomials $q_j(x)$ for $0 \le j \le 4$. In other words the entries in the first column are the coefficient of x^0 in $q_j(x)$, the entries in the next column are the coefficient of x and so on. The unit diagonal entries correspond to the coefficient of the leading term x^j in $q_j(x)$.

Similarly the application of Steps 3, 4 and 5 of Algorithm 2 produces the coefficients of the numerator polynomials, namely the A matrix:

$$A = \begin{bmatrix} 96 & 0 & 0 & 0 & 0 \\ -72 & -12 & 0 & 0 & 0 \\ 26 & 9 & 1 & 0 & 0 \\ -6 & -\frac{19}{6} & -\frac{3}{4} & -\frac{1}{12} & 0 \\ 1 & \frac{11}{16} & \frac{23}{96} & \frac{1}{16} & \frac{1}{96} \end{bmatrix}$$

Hence the rational interpolants are found to be

$$r_{0,4}(x) = \frac{p_0(x)}{q_4(x)} = \frac{96}{x^4 - 6x^3 + 23x^2 - 66x + 96}$$

$$r_{1,3}(x) = \frac{p_1(x)}{q_3(x)} = \frac{12x + 72}{-x^3 + 9x^2 - 38x + 72}$$

$$r_{2,2}(x) = \frac{p_2(x)}{q_2(x)} = \frac{x^2 + 9x + 26}{x^2 - 9x + 26}$$

$$r_{3,1}(x) = \frac{p_3(x)}{q_1(x)} = \frac{x^3 + 9x^2 + 38x + 72}{-12x + 72}$$

$$r_{4,0}(x) = \frac{p_4(x)}{q_0(x)} = \frac{x^4 + 6x^3 + 23x^2 + 66x + 96}{96}$$

5.6. PARALLEL COMPUTATION OF THE RATIONAL INTERPOLANTS

In this section we assume that we have a shared-memory multiprocessor with $N+1$ identical processors. Each processor reads its operands from the memory, performs floating point operations and writes the result back to the prescribed memory location. We also assume that read, write and wait times are negligible with respect to floating point operations and a floating point operation takes 1 unit time. The algorithms we now describe are suitable for a shared-memory PRAM model [FoWy78]. First we observe the Procedure TTR can be parallelized.

Lemma 5.2

Given $N+1$ processors Procedure TTR computes the first $n+1$ rows of the matrix T in $O(n \log N)$ time. Also all rows of the matrix T can be computed in $O(N \log N)$ time with the same number of processors.

Proof :

In Step 1 of Procedure TTR the *ith* processor first computes w_i and then c_i. The computation of w_i takes $2N-1$ steps and c_i is computed on a single step. Since $N+1$ processors operate simultaneously, Step 1 takes $2N$ parallel steps.

For the computation of α_0 in Step 2, the $N+1$ processors first compute $c_i x_i$ in a single step. The computation of γ_0 involves the sum of $N+1$ scalars. This can be done in $\log(N+1)$ parallel steps using $N+1$ processors by implementing a binary tree addition algorithm. Similarly the computation of θ_0 will take $\log(N+1)$ parallel steps. Then we perform a single division to compute α_0. Thus Step 2 contributes $2\log(N+1)+2$ parallel arithmetic steps.

In Step 3 first T_{1i} is computed by the *ith* processor in a single step. Then $c_i T_{1i}^2$ and $c_i x_i T_{1i}^2$ are computed in three parallel steps by all processors. To compute γ_1 all processors perform a binary tree addition algorithm which will take. $\log(N+1)$ parallel operations. Similarly the computation of θ_1 will take $\log(N+1)$ parallel arithmetic operations. Then we compute α_1 and β_1 in a single step using 2 processors. Thus Step 3 will take $2\log(N+1)+5$ arithmetic steps.

In Step 4, $T_{j+1,i}$ is computed by the *ith* processor using 4 steps. Then α_{j+1} and β_{j+1} are computed in $2\log(N+1)+4$ steps. For all $j = 1, 2, \ldots, n-1$ Step 4 of Procedure TTR requires $(n-1)[2\log(N+1)+8]$ parallel arithmetic steps. If we compute all rows of the matrix T then this step takes $(N-1)[2\log(N+1)+8]$ parallel arithmetic steps.

Thus by summing the number of parallel arithmetic operations at each step we obtain

$$2n\log(N+1)+2N+8n+2\log(N+1)-1=O(n\log N) \tag{5.35}$$

if the first $n+1$ rows of the matrix T is computed, and

$$2N\log(N+1)+10N+2\log(N+1)-1=O(N\log N) \tag{5.36}$$

if all entries of the matrix T is computed. ●

Also we note that given n processors a Newton polynomial interpolating the data set (x_i, f_i) can be computed in $3n$ parallel arithmetic steps. Using this result and Lemma 5.2 we now show that given $N+1$ processors Algorithm 1 computes a rational interpolant in $O(n\log N)$ parallel arithmetic steps, and all rational interpolants can be computed in $O(N\log N)$ parallel arithmetic steps using Algorithm 2.

Theorem 5.5

Given $N+1$ processors, a particular rational interpolating function $r_{m,n}(x)$ can be computed in $O(n \log N)$ parallel steps by Algorithm 1. Furthermore Algorithm 2 computes all rational interpolants $r_{m,n}(x)$ for $m+n=N$ and $n=0,1,\ldots,N$ in $O(N \log N)$ parallel steps with the same number of processors.

Proof :

Step 1 of Algorithm 1 is a call to Procedure TTR with the data set n and (x_i, f_i) for $0 \le i \le N$. This will take

$$2n \log(N+1) + 2N + 8n + 2\log(N+1) - 1$$

steps according to Lemma 5.2. The coefficients of a Newton polynomial interpolating a set of $n+1$ points can be computed $3n$ time with n processors. For Step 2 and 3 we first compute $f_i Q_{ni}$ for $0 \le i \le m$ in a single step using $m+1$ processors. Then m processors perform Newton interpolation on the data $(x_i, f_i Q_{ni})$ for $0 \le i \le m$, and the remaining processors perform Newton interpolation on the data (x_i, Q_{ni}) for $0 \le i \le n$ simultaneously. The former takes $3m$ parallel steps while the latter $3n$ takes parallel steps. Thus Step 2 and Step 3 will take $1+3m+3n = 3N+1$ parallel arithmetic operations.

Thus the total number of parallel arithmetic operations for Algorithm 1 is found as

$$2n \log(N+1) + 5N + 8n + 2\log(N+1) = O(n \log N) \qquad (5.37)$$

The parallel implementation Algorithm 2 is along the same lines as Algorithm 1 with the exception of Steps 2, 4, and 5. Also note that Procedure TTR computes all rows of the matrix T.

In Step 2 the *kth* processor computes $B_{j+1,k}$ in 4 parallel steps. For all $j = 1, 2, \ldots, N-1$ Step 2 takes $4(N-1)$ steps. The same count applies also to Step 4.

In Step 5 the jth processor computes $f_0 \frac{Q_{N-j,0}}{P_{j0}}$ in 2 steps. The lower triangular half of the matrix A is then updated in $N+1$ steps using all processors. Thus Step 5 takes exactly $N+3$ parallel arithmetic steps. Thus we can summarize the number of parallel operations for each step of Algorithm 2 as follows:

Step 1 : $2N \log(N+1) + 10N + 2\log(N+1) - 1$

Step 2 : $4(N-1)$

Step 3 : $1 + 2N \log(N+1) + 10N + 2\log(N+1) - 1$

Step 4 : $4(N-1)$

Step 5 : $N+3$

Thus we conclude that Algorithm 2 computes all rational interpolants in

$$4N \log(N+1) + 29N + 4\log(N+1) - 4 = O(N \log N) \tag{5.38}$$

parallel arithmetic steps with $N+1$ processors. ●

5.7. SUMMARY AND CONCLUSIONS

We have described two algorithms for rational interpolation. Given that the Hankel matrices $H_0, H_1, \ldots, H_{N-1}$ are all nonsingular, the first one of these algorithms (Algorithm 1) generates orthogonal polynomials $q_0(x), q_1(x), \ldots, q_N(x)$ which are the monic denominator polynomials of the associated rational interpolants. If the computation of the corresponding numerator polynomials are then carried out by Newton interpolation, then the number of arithmetic operations required becomes $O(N^3)$. However, if the corresponding matrices $H'_0, H'_1, \ldots, H'_{N-1}$ for the numerators are also nonsingular, then Newton interpolation can be avoided, and all rational interpolants can be found with only $O(N^2)$ arithmetic operations. This is the content of Algorithm 2.

Note that in case a particular matrix H_{j-1} happens to be singular, then it is no longer possible to continue the generation of the denominator polynomials $q_j(x), q_{j+1}(x), \ldots, q_N(x)$ by using the three term recursion, as the necessary condition for the existence of the desired orthogonal family of polynomials is not satisfied. Thus, if the discrete symmetric bilinear form we are interested in happens to be degenerate, it is not clear how to proceed with the method of orthogonal polynomials to generate the rest of the rational interpolants. At this point, however, the remaining denominator polynomials can be computed individually by repeated application of Rissanen's algorithm at a cost of $O(N^2)$ operations each. Similarly, the singularity of the a matrix H'_{j-1} associated with a numerator polynomial puts a limit on the applicability and the operating range of Algorithm 2. Nevertheless, it is possible to combine these two approaches for a hybrid algorithm whose running time is guaranteed to be no worse than the existing algorithms for rational interpolation. In addition, both Algorithm 1 and Algorithm 2 provide simple provisions to check for degeneracy of the interpolation problem at hand.

Although the proposed algorithms are fast, an extensive analysis of their practicality in terms of numerical stability remains to be done. In a recent manuscript, Higham proposed fast algorithms for the solution of Vandermonde-like systems using orthogonal polynomials [High86]. The analysis shows that addition of just one step iterative refinement in single precision is enough to make his algorithms numerically stable.

REFERENCES

[AbSt65]
M. Abramowitz and I. A. Stegun, **Handbook of Mathematical Functions**, pp. 877, Dover, 1972.

[AhHU74]
A. Aho, J. E. Hopcroft, and J. D. Ullman, **The Design and Analysis of Computer Algorithms**, Addison-Wesley, 1974.

[AhNW67]
J. H. Ahlberg, E. N. Nilson, and J. L. Walsh, **The Theory of Splines and their Applications**, Academic Press, 1967.

[AAGK87]
M. E. Annarotone, E. Arnold, T. Gross, H.-T. Kung, M. Lam, O. Menzilcioglu, and J. Webb, "The WARP Computer: Architecture, Implementation, and Performance," *IEEE Transactions on Computers*, Vol. C-36, No. 12, pp. 1523-1538, December 1987.

[BeZh65]
I. S. Berezin and N. P. Zhidkov, **Computing Methods**, Vol. 1, Addison-Wesley, 1965.

[BiGa86]
A. Bilgory and D. Gajski, "A Heuristic for Suffix Solutions," *IEEE Transactions on Computers*, Vol. C-35, No. 1, pp. 34-42, January 1986.

[BjPe70]
A. Bjorck and V. Pereyra, "Solution of Vandermonde Systems of Equation," *Mathematics of Computation*, Vol. 24, No. 112, pp. 893-903, October 1970.

[BrKu82]
R. P. Brent and H. T. Kung, "A Regular Layout for Parallel Adders," *IEEE Transactions on Computers*, Vol. C-31, No. 3, pp. 260-264, March 1982.

[Brue84]
G. Bruegner, "Rounding Error Analysis of Interpolation Procedures," *Computing*, Vol. 33, No. 1, pp. 83-87, 1984.

[Cann69]
L. E. Cannon, **A Cellular Computer to Implement the Kalman Filtering Algorithm**, Ph. D. Thesis, Montana State University, 1969.

[Capp87]
P. R. Cappello, "Gaussian Elimination on a Hypercube Automaton," *Journal of Parallel and Distributed Computing*, Vol. 4, No. 3, pp. 288-308, June 1987.

[CaSt84]
P. R. Cappello and K. Steiglitz, "Unifying VLSI Array Designs with Linear Transformations of Space-Time," in **Advances in Computer Research**, edited by F. P. Preparata, Vol. 2, pp. 23-65, JAI Press, 1984.

[Chin76]
F. Y. Chin, "A Generalized Upper Bound on Fast Polynomial Evaluation and Interpolation," *SIAM Journal on Computing*, Vol. 5, No. 4, pp. 682-690, December 1976.

[Davi65]
P. J. Davis, "Approximation Theory in the First Two Decades of Electronic Computers," in **Approximation of Functions**, edited by H. L. Garabedian, pp. 152-163, Elsevier Publishing Company, 1965.

[Davi75]
P. J. Davis, **Interpolation and Approximation**, Dover, 1975.

[DeNS81]
E. Dekel, D. Nassimi, and S. Sahni, "Parallel Matrix and Graph Algorithms," *SIAM Journal on Computing*, Vol. 10, No. 4, November 1981.

[DeMa74]
V. F. Dem'yanov and V. N. Malozemov, **Introduction to Minimax**, John-Wiley and Sons, 1974.

[Dew_84]
P. M. Dew, "VLSI Architectures for Problems in Numerical Computation," in **Supercomputers and Parallel Computation**, edited by D. J. Paddon, pp. 1-24, Oxford University Press, 1984.

[DBMS79]
J.J. Dongarra, J. R. Bunch, C. B. Moler, and G.W. Stewart, **Linpack Users' Guide**, SIAM, Philadelphia, 1979.

[DuRo77]
P. Dubois and G. Rodrigue, "An Analysis of the Recursive Doubling Algorithm," in **High Speed Computer and Algorithm Organization**, edited by D. J. Kuck, D. H. Lawrie and A. H. Sameh, pp. 299-305, Academic Press, 1977.

[Fich83]
F. E. Fich, "New Bounds for Parallel Prefix Circuits," *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pp. 100-109, Boston, MA., April 1983.

[FiKu85]
A. L. Fisher and H. T. Kung, "Special-Purpose VLSI Architectures: General Discussions and a Case Study," in **VLSI and Modern Signal Processing**, edited by S. Y. Kung, H. J. Whitehouse and T. Kailath, pp. 153-169, Prentice-Hall, 1985.

[FoWy78]
S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," *Proceedings of the 10th ACM Annual Symposium on Theory of Computing*, pp. 114-118, San Diego, CA., May 1978.

[Fors57]
G. E. Forsythe, "Generation and Use of Orthogonal Polynomials for Data Fitting with a Digital Computer," *Journal of SIAM*, Vol. 5, No. 2, pp. 74-88, June 1957.

[FoSc87]
D. E. Foulser and R. Schreiber, "The Saxpy Matrix-1: A General Purpose Systolic Computer," *IEEE Computer*, Vol. 20, No. 7, pp. 35-43, July 1987.

[GaMa82]
M. Gasca and J. I. Maeztu, "On Lagrange and Hermite Interpolation in R^k," *Numerische Mathematik*, Vol. 39, No. 1, pp. 1-14, 1982.

[Gao_87]
G. R. Gao, "A Stability Classification Method and its Application to Pipelined Solution of Linear Recurrences," *Parallel Computing*, Vol. 4, No. 3, pp. 305-321, June 1987.

[GoVa83]
G. Golub and C. F. Van Loan, **Matrix Computations**, The Johns Hopkins University Press, 1983.

[GrHo81]
P. R. Graves-Morris and T. R. Hopkins, "Reliable Rational Interpolation," *Numerische Mathematik*, Vol 36, No. 2, pp. 111-128, 1981.

[GuRo70]
R. B. Guenther and E. L. Roetman, "Some Observations on Interpolation in Higher Dimensions," *Mathematics of Computation*, Vol. 24, No. 111, pp. 517-527, July 1970.

[Hako82]
H. A. Hakopian, "Multivariate Divided Differences and Multivariate Interpolation of Lagrange and Hermite Type," *Journal of Approximation Theory*, Vol. 34, No. 3, pp. 286-305, March 1982.

[Hell78]
D. Heller, "A Survey of Parallel Algorithms in Numerical Algebra," *SIAM Review*, Vol. 20, No. 4, pp. 740-777, 1978.

[Hell85a]
D. Heller, "Mathematical Hardware - Design and Responsibilities," in **Algorithmically Specialized Parallel Computers**, edited by L. Snyder, L. H. Jamieson, D. B. Gannon, and H. J. Siegel, pp. 233-241, Academic Press, 1985.

[Hell85b]
D. Heller, "Partitioning Big Matrices for Small Systolic Arrays," in **VLSI and Modern Signal Processing**, edited by S. Y. Kung, H. J. Whitehouse and T. Kailath, pp. 185-199, Prentice-Hall, 1985.

[High86]
N. J. Higham, "Fast Solution of Vandermonde-like Systems Involving Orthogonal Polynomials," Numerical Analysis Report No. 118, Department of Mathematics, University of Manchester, June 1986.

[Hild56]
F. B. Hildebrand, **Introduction to Numerical Analysis**, McGraw-Hill, 1956.

[HoRu85]
H. J. Hoover and W. L. Ruzzo, "A Compendium of Problems Complete for P," Technical Report, Department of Computer Science, University of Washington, December 1985.

[Horo72]
E. Horowitz, "A Fast Method for Interpolation using Preconditioning," *Information Processing Letters*, Vol. 1, No. 4, pp. 157-163, June 1972.

[INMO86]
INMOS Ltd., "IMS Transputer," 72 TRN 117 01, INMOS Ltd., Almondsbury, Bristol, UK, November 1986.

[John86]
S. L. Johnsson, "Band Matrix Systems Solvers on Ensemble Architecture," in **Supercomputers: Algorithms, Architectures, and Scientific Computation**, edited by F. A. Matsen and T. Tajima, pp. 196-216, University of Texas Press, 1986.

[John87a]
S. L. Johnsson, "Solving Tridiagonal Systems on Ensemble Architectures," *SIAM Journal on Scientific and Statistical Computing*, Vol. 8, No. 3, pp. 354-392, May 1987.

[John87b]
S. L. Johnsson, "Communication Efficient Basic Linear Algebra Computations on Hypercube Multiprocessors," *Journal of Parallel and Distributed Computing*, No. 4, pp. 133-172, 1987.

[JoHo87]
S. L. Johnsson and C. T. Ho, "Multiple Tridiagonal Systems, the Alternating Direction Methods and Boolean Cube Configured Multiprocessors," Technical Report No. YALEU/DCS/RR-532, Yale University, June 1987.

[KAGB82]
S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and, D. V. Bhaskar-Rao, and R. Gal-Ezer, "Wavefront Array Processor: Language, Architecture and Applications," *IEEE Transactions on Computers*, Vol. C-31, No. 11, pp. 1054-1065, November 1982.

[Kinc48]
W. M. Kincaid, "Solution of Equations by Interpolation," *Annals of Mathematical Statistics*, Vol. 19, No. 2, pp. 207-219, June 1948.

[KoSt73]
P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers*, Vol. C-22, No. 8, pp. 786-793, August 1973.

[KrRS85]
C. P. Kruskal, L. Rudolph, and M. Snir, "The Power of Parallel Prefix," *IEEE Transactions on Computers*, Vol. C-34, No. 10, pp. 965-968, October 1985.

[Krog70]
F. Krogh, "Efficient Algorithms for Polynomial Interpolation and Divided Differences," *Mathematics of Computation*, Vol. 24, No. 109, pp. 185-190, January 1970.

[Kung73]
H. T. Kung, "Fast Evaluation and Interpolation," Technical Report, Department of Computer Science, Carnegie-Mellon University, January 1973.

[KuLe80]
H. T. Kung and C. E. Leiserson "Algorithms for VLSI Processor Arrays," in **Introduction to VLSI Systems**, by C. Mead and L. Conway, pp. 271-292, Addison-Wesley, 1980

[LaFi80]
R. Ladner and M. Fischer, "Parallel Prefix Computation," *Journal of ACM*, Vol. 27, No. 4, pp. 831-838, October 1980.

[Lark81]
F. M. Larkin, "Root Finding By Divided Differences," *Numerische Mathematik*, Vol. 37, No. 1, pp. 93-104, 1981.

[LaYD87]
S. Lakshmivarahan, C. Yang, and S. K. Dhall, "On a New Class of Optimal Parallel Prefix Circuits with $(SIZE + DEPTH) = 2n - 2$ and $\lceil \log n \rceil \leq DEPTH \leq (2\lceil \log n \rceil - 3)$," *Proceedings of the International Conference on Parallel Processing*, pp. 58-65, 1987.

[Leis82]
C. E. Leiserson, **Area-Efficient VLSI Computation**, The MIT Press, 1982.

[MacD79]
I. G. MacDonald, **Symmetric Functions and Hall Polynomials**, Oxford University Press, London, 1979.

[Macl82]
A. J. Macleod, "A Comparison of Algorithms for Polynomial Interpolation," *Journal of Computational and Applied Mathematics*, Vol. 8, No. 4, pp. 275-277, 1982.

[Maru73]
K. Maruyama, "On the Parallel Evaluation of Polynomials," *IEEE Transactions on Computers*, Vol. C-22, No. 1, pp. 2-5, January 1973.

[McKe86]
G. P. McKeown, "Iterated Interpolation using a Systolic Array," *ACM Transactions on Mathematical Software*, Vol. 12, No. 2, pp. 162-170, June 1986.

[McVa87]
O. A. McBryan and E. F. Van de Velde, "Hypercube Algorithms and Implementations," *SIAM Journal on Scientific and Statistical Computing*, Vol. 8, No. 2, pp. s227-s287, March 1987.

[Mill75]
W. Miller, "Computational Complexity and Numerical Stability," *SIAM Journal on Computing*, Vol. 4, No. 2, pp. 97-107, June 1975.

[MoFo86]
D. I. Moldovan and J. A. B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Transactions on Computers*, Vol. C-35, No. 1, pp. 1-12, January 1986.

[MuPa73]
I. Munro and M. Paterson, "Optimal Algorithms for Parallel Polynomial Evaluation," *Journal of Computer and System Sciences*, Vol. 7, No. 2, pp. 189-198, April 1973.

[Muir60]
T. Muir, **The Theory of Determinants**, Vol. 2, pp. 326-330, Dover Publications, 1960.

[NeSc85]
A Neumaier and A. Schafer, "Divided Differences, Shift Transformations and Larkin's Root Finding Method," *Mathematics of Computation*, Vol. 45, No. 171, pp. 181-196, July 1985.

[OrVo85]
J. Ortega and R. Voigt, "Partial Differential Equations on Vector and Parallel Computers," *SIAM Review*, Vol. 27, No. 2, pp. 149-240, June 1985.

[Powe67]
M. J. D. Powell, "On the Maximum Errors of Polynomial Approximations Defined by Interpolation and by Least Squares Criteria," *Computer Journal*, Vol. 9, No. 4, pp. 404-407, February 1967.

[RaRa78]
A. Ralston and P. Rabinowitz, **A First Course in Numerical Analysis**, McGraw-Hill, 1978.

[ReND77]
E. M. Reingold, J. Nievergelt, and N. Deo, **Combinatorial Algorithms: Theory and Practice**, Prentice-Hall, 1977.

[Reif86]
J. Reif, "Logarithmic Depth Circuits for Algebraic Functions," *SIAM Journal on Computing*, Vol. 15, No. 1, pp. 231-242, February 1986.

[Rice70]
J. R. Rice, "General Purpose Curve Fitting," in **Approximation Theory**, pp. 191-204, edited by A. Talbot, Academic Press, 1970.

[Riss74]
J. Rissanen, "Solution of Linear equations with Hankel and Toeplitz Matrices," *Numerische Mathematik*, Vol. 22, No. 5, pp. 361-366, 1974.

[Rivl69]
T. J. Rivlin, **An Introduction to the Approximation of Functions**, Dover, 1969.

[Rons84]
W. Ronsch, "Stability Aspects in using Parallel Algorithms," *Parallel Computing*, Vol. 1, No. 1, pp. 75-98, August 1984.

[Salz56]
H. E. Salzer, "Some New Divided Difference Algorithms," in **On Numerical Approximation**, edited by R. E. Langer, pp. 61-98, The University of Wisconsin Press, 1956.

[Salz64]
H. E. Salzer, "Divided Differences for Functions of Two Variables for Irregularly Spaced Arguments," *Numerische Mathematik*, Vol. 6, No. 2, pp. 68-77, 1964.

[SaSc85a]
Y. Saad and M. H. Schultz, "Data Communication in Hypercubes," Technical Report No. YALEU/DSC/RR-428, Department of Computer Science, Yale University, October 1985.

[SaSc85b]
Y. Saad and M. H. Schultz, "Topological Properties of Hypercubes," Technical Report No. YALEU/DSC/RR-389, Department of Computer Science, Yale University, June 1985.

[ScWe86]
C. Schneider and W. Werner, "Some New Aspects of Rational Interpolation," *Mathematics of Computation*, Vol. 47, No. 175, pp. 285-299, July 1986.

[Schu76]
L. L. Schumaker, "Fitting Surfaces to Scattered Data," in **Approximation Theory**, Vol. II, edited by G. G. Lorentz, C. K. Chui, and L. L. Schumaker, pp. 203-268, Academic Press, 1976.

[Seit85]
C. L. Seitz, "The Cosmic Cube," *Communications of the ACM*, Vol. 28, No. 1, pp. 22-33, January 1985.

[SiVl73]
K. Singhal and J. Vlach, "Accuracy and Speed of Real and Complex Interpolation," *Computing*, Vol. 11, No. 2 pp. 147-158, 1973.

[Snir86]
M. Snir, "Depth-Size Trade-Off for Parallel Prefix Computation," *Journal of Algorithms*, Vol. 7, No. 2, pp. 185-201, 1986.

[Stef27]
J. F. Steffensen, **Interpolation**, The Williams & Wilkins Company, 1927.

[Stie56]
E. L. Stiefel, "Numerical Methods for Tchebycheff Approximation," in **On Numerical Approximation**, edited by R. E. Langer, pp. 217-232, The University of Wisconsin Press, 1956.

[Ston73]
H. S. Stone, "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations," *Journal of ACM*, Vol. 20, No. 1, pp. 27-38, January 1973.

[Ston75]
H. S. Stone, "Parallel Tridiagonal Equation Solvers," *ACM Transactions on Mathematical Software*, Vol. 1, No. 4, pp. 289-307, December 1975.

[Stum80]
F. Stummel, "Rounding Error Analysis of Elementary Numerical Algorithms," *Proceedings of Conference on Fundamentals of Numerical Computation*, Berlin, 1979, edited by G. Alefeld and R. d. Grigorieff, Springer-Verlag, *Computing*, Vol. Suppl. 2, pp. 169-195, 1980.

[Stum81]
F. Stummel, "Perturbation Theory for Evaluation Algorithms of Arithmetic Expressions," *Mathematics of Computation*, Vol. 37, No. 156, pp. 435-473, October 1981.

[Szeg59]
G. Szego, **Orthogonal Polynomials**, American Mathematical Society Colloquium Publications Vol. 23, 1959.

[TaGo81]
W. P. Tang and G. H. Golub, "The Block Decomposition of a Vandermonde Matrix and its Applications," *BIT*, Vol. 21, No. 4, pp. 505-517, 1981.

[Thac60]
H. C. Thacher, Jr., "Derivation of Interpolation Formulas in Several Independent Variables," *Annals of New York Academy of Sciences*, Vol. 86, No. 3, pp. 758-775, May 1960.

[Trau82]
J. F. Traub, **Iterative Methods for the Solution of Equations**, Chelsea Publishing Company, 1982.

[Tren65]
W. F. Trench, "An Algorithm for the Inversion of Finite Hankel Matrices," *Journal of SIAM*, Vol. 13, No. 4, pp. 1102-1107, December 1965.

[TsPr78]
N. K. Tsao and R. Prior, "On Multipoint Numerical Interpolation," *ACM Transactions on Mathematical Software*, Vol. 4, No. 1, pp. 51-56, March 1978.