Intern. J. Computer Math., Vol. 38, pp. 219–239
Reprints available directly from the publisher
Photocopying permitted by license only

© 1991 Gordon and Breach Science Publishers S.A.
Printed in the United Kingdom

DECOMPOSING POLYNOMIAL INTERPOLATION FOR SYSTOLIC ARRAYS*

ÇETIN K. KOÇ†

Department of Electrical Engineering, University of Houston, Houston, TX 77204

PETER R. CAPPELLO†

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

E. GALLOPOULOS‡

Center for Supercomputing Research and Development, University of Illinois at Urbana Champaign, Urbana, IL 61801

(*Received 18 June 1990*)

This paper (1) summarizes recent work on systolic computation of interpolating polynomials, presenting several time-optimal and spacetime-optimal systolic arrays for computing a process dependence graph for Aitken's algorithm, and (2) presents an efficient decomposition of the computation by partitioning the corresponding so-called *Newton–Vandermonde* matrix. Such a decomposition is used to treat an interpolation problem whose size exceeds the array size.

KEY WORDS: Newton and Hermite interpolation, Aitken's algorithm, systolic array, extent problem, decomposition, Newton–Vandermonde matrix.

AMS SUBJECT CLASSIFICATION: 65D05, 68Q80.

C.R. CATEGORIES: F.1.2, F.2.1, G.1.1.

1. INTRODUCTION

McKeown [15] proposes a systolic array for iterated interpolation. He observes that developments in VLSI technology, continually falling cost, and increasing capacity of semiconductor memory make it feasible to store a table of values for a function requiring complex direct computation. These "chip" tables, coupled with a systolic array for iterated interpolation, thus allow us to efficiently approximate

*A portion of this work was presented at the Fourth SIAM Conference on Parallel Processing for Scientific Computing, Chicago, Illinois, December 11–13, 1989.

†This material is based on work supported by the National Science Foundation under grant MIP89-20598 and the Office of Naval Research under contract N00014-84-K-0664.

‡Research supported by the National Science Foundation under grant NSF CCR-8717942 with additional support from AT&T under grant AT&T AFFL67Sameh, the U.S. Air Force Office for Scientific Research under grant AFOSR-85-0211 and the U.S. Department of Energy under grant DOE DE-FG02-85ER25001.

the function's values. Additionally, when we have access to the values of the function at certain points but not to the function itself, systolic arrays will provide parallelism [5].

The McKeown array is a systolic implementation of Aitken's method of iterated interpolation. A systolic version of Newton and Hermite polynomial interpolation using the algorithms of Aitken and Neville is presented by Cappello, Gallopoulos, and Koç [5]. They present several alternative systolic implementations of Aitken's algorithm, some of which are spacetime-optimal, and apply these implementations to Neville's algorithm. Furthermore, a 2-level systolic array for computing generalized divided differences—which are the coefficients of the Hermite interpolating polynomial—and a systolic scheme for the multivariate case are presented in [5]. They also show that these observations easily apply to iterated interpolation (i.e., the interpolating polynomials can be evaluated implicitly by only reprogramming the processors).

In this paper, we:

- define, in Section 2, the polynomial interpolation problem and give the Aitken and the Neville recursions;
- summarize, in Sections 3–6, the results in [15] and [5];
- present, in Section 7, a novel scheme, based on the mathematics of the problem, for decomposing a large interpolation problem so that it can be computed on smaller systolic arrays.

2. POLYNOMIAL INTERPOLATION

The polynomial interpolation problem is defined as follows:

Input: An ordered collection of $n+1$ natural numbers $\mathbf{m}=(m_0, m_1, \ldots, m_n)$, and the values and the derivatives of a function $f(x)$ evaluated at the $n+1$ node points x_i from field F,

$$x_0: f(x_0), f^{(1)}(x_0), f^{(2)}(x_0), \ldots, f^{(m_0-1)}(x_0)$$

$$x_1: f(x_1), f^{(1)}(x_1), f^{(2)}(x_1), \ldots, f^{(m_1-1)}(x_1)$$

$$\vdots$$

$$x_n: f(x_n), f^{(1)}(x_n), f^{(2)}(x_n), \ldots, f^{(m_n-1)}(x_n).$$

PROBLEM *Find a polynomial $p_N(x) \in F[x]$ of degree N such that*

$$p_N^{(k)}(x_i) = f^{(k)}(x_i), \quad for \quad 0 \leqq i \leqq n, \quad 0 \leqq k \leqq m_i - 1,$$

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

The interpolating polynomial of degree N exists and is unique, provided that $x_i \neq x_j$ for $i \neq j$ [2]. 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 $0 \leqq i \leqq n$, 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 $0 \leqq i \leqq n$, but which usually is in the same interval as the x_i's. This polynomial can be used to approximate the derivatives of the function $f(x)$ as well.

The interpolating polynomial can be given as

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

In the literature, this polynomial is called the *Hermite Interpolating* polynomial. The simplest instance of the Hermite interpolating polynomial is when $\mathbf{m}=(1,1,\ldots,1)$, which corresponds to the Newton interpolating polynomial. The coefficients of (1) are referred to as the *generalized divided differences* [11, 13, 25]; those for the Newton interpolating polynomial are simply called the *divided differences.* The coefficients of the Newton polynomial interpolating the function $f(x)$ at the node points x_i for $0 \leqq i \leqq n$ can be computed using the well-known algorithms of Neville and Aitken [11, 13]. 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 and Neville recursions also can be used to compute the generalized divided differences. If we define [2]

$$
f_{i^k} \equiv \underbrace{f_{iii\ldots i}}_{k \text{ times}} \equiv \frac{1}{(k-1)!} f^{(k-1)}(x_i),
\tag{2}
$$

then the Aitken algorithm for generalized divided differences uses the recursion:

$$
f_{0^{a_0}\ldots i^{a_i}, j^{a_j}} = \frac{f_{0^{a_0}\ldots i^{a_i}, j^{a_j-1}} - f_{0^{a_0}\ldots i^{a_i-1}, j^{a_j}}}{x_i - x_j}
\tag{3}
$$

for $0 \leqq i \leqq n-1$ and $i+1 \leqq j \leqq n$; the corresponding Neville algorithm uses the recursion:

$$
f_{i^{a_i}\ldots j^{a_j}} = \frac{f_{i^{a_i}\ldots j^{a_j-1}} - f_{i^{a_i-1}\ldots j^{a_j}}}{x_i - x_j}
\tag{4}
$$

for $0 \leqq i \leqq n-1$ and $i+1 \leqq j \leqq n$, where $a_i \leqq m_i$ for $0 \leqq i \leqq n$. These recursion formulae specialize to the Newton case when $\mathbf{m}=(1,1,\ldots,1)$.

3. AITKEN PROCESS DEPENDENCE GRAPH

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 look at the algorithms that compute the entries. This allows us to form the *process dependence graph* of each algorithm to compute the divided differences.

Let matrices $A_{ij} \in F^{m_i \times m_j}$ contain the generalized divided differences, where $A_{ij}(p, q)$ denotes an entry for $1 \leqq p \leqq m_i$ and $1 \leqq q \leqq m_j$, where

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

for $0 \leqq i \leqq n-1$, $i < j \leqq n$, and $1 \leqq p \leqq m_i$, $1 \leqq q \leqq m_j$.

We illustrate for $n=4$ *and* $\mathbf{m} = (1, 1, 1, 1, 1)$. In this case, we have

$$A_{ij}(1, 1) \stackrel{\text{def}}{=} A_{ij} = f_{012 \dots ij}.$$

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 recursion (3), filling the Aitken table as follows:

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

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

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

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

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, (i.e., 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 $\mathbf{m} = (1, 1, 1, 1, 1)$ in Figure 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 to dependences between the variables used in the operations. The node at point (i, j) computes A_{ij} by performing the operation

$$A_{ij} = \frac{A_{i-1,i} - A_{i-1,j}}{x_i - x_j}. \tag{5}$$

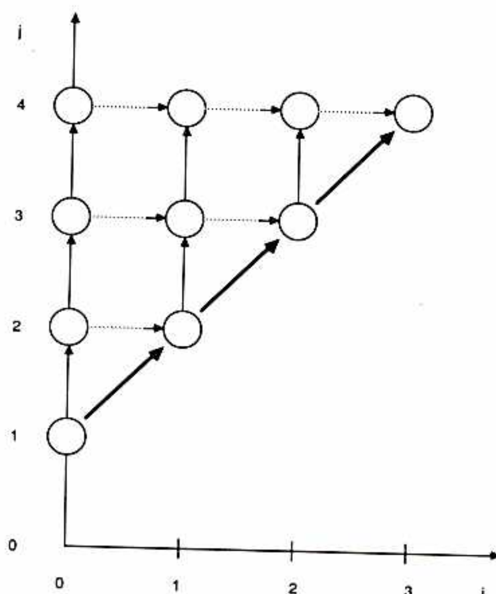

Figure 1 Process dependence graph of the Aitken algorithm, G_A, for $n=4$.

The necessary operands are present for the operations performed in the nodes, as a comparison of Figure 1 with the Aitken table reveals.

These observations can be used to compute the generalized divided differences, for which $m_i > 1$ and A_{ij} are matrices of dimension $m_i \times m_j$. If the nodes represent the operations to compute the entries of the matrix A_{ij}, then the process dependence graph for the generalized divided difference tables is the same as the Newton case, illustrated in Figure 1. The node at point (i, j) now represents a rectangular $m_i \times m_j$ mesh of smaller nodes, where the entries of the matrix A_{ij} are computed [5]. If $m_i = 1$ for $0 \leqq i \leqq n$, 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. We thus can view Figure 1 as the process dependence graph of the general case, where the nodes represent rectangular meshes of simpler nodes (operations). This hierarchical view simplifies the treatment of each case, and allows us to embed the process dependence graph in spacetime, producing 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 [17, 18, 6, 16, 8, 20, 21]).

4. THE McKEOWN ARRAY

In this section, we present McKeown's array [15]. We embed the process dependence graph for Aitken's algorithm in spacetime: the abscissa is interpreted as time (t); the ordinate as space (s). The linear embedding, E_1, is as follows:

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

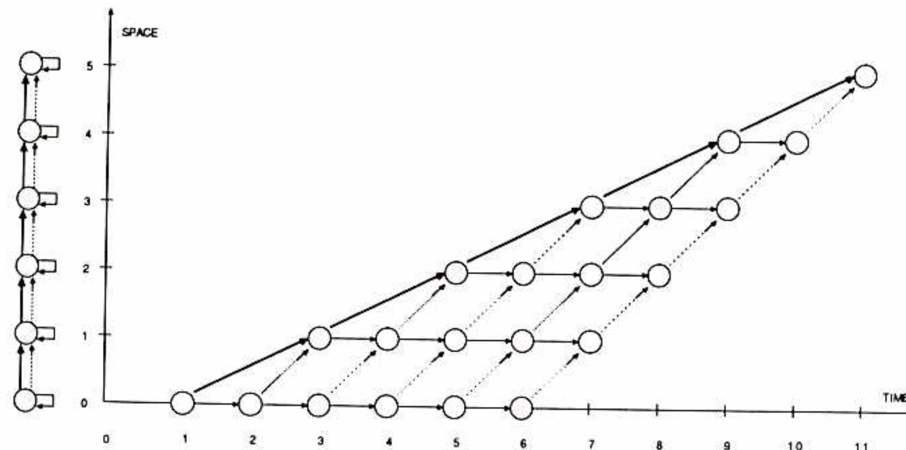

Figure 2 A spacetime representation of McKeown's array for $n=6$. Its spatial projection is a linear array of n processors.

The result, depicted in Figure 2 for $n=6$, is McKeown's array. Data flowing south$\rightarrow$north in Figure 1, flows in the direction of time (perpendicular to space) in the McKeown design: It is in the processors' memory. Data flowing west$\rightarrow$east in Figure 1, flows up through the McKeown array. Data flowing south$\rightarrow$east in Figure 1, also flows up through the McKeown array, but at half the speed of the data flowing west$\rightarrow$east.

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

5. SPACETIME-OPTIMAL ARRAYS

A graph's embedding is referred to as *spacetime-optimal* when it is space-minimal among those embeddings that are time-optimal. We now present a spacetime-optimal bilateral array. Data flowing south$\rightarrow$north in the McKeown array (Figure 1), moves up the bilateral array; data flowing west$\rightarrow$east in Figure 1, moves down the bilateral array. The spacetime embedding, E_2, is presented in two steps.

Step 1 We initially embed the process dependence graph as follows:

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

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

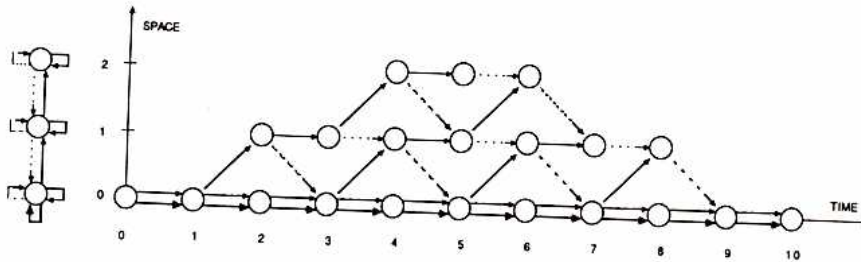

Figure 3 Spacetime-optimal embedding E_2 of G_A for $n=6$. Its spatial projection is a linear array of $\lceil n/2 \rceil$ processors.

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

$$T_2 = [C] \quad \text{where} \quad C = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix}$$

and

$$y = [C]x \equiv [Cx] \quad \text{where} \quad \left\lfloor \begin{bmatrix} i \\ j \end{bmatrix} \right\rfloor \equiv \begin{bmatrix} \lfloor i \rfloor \\ \lfloor j \rfloor \end{bmatrix}.$$

Processor efficiency (i.e., the percentage of time that a processor is used) is doubled asymptotically by this nonlinear transformation. Figure 3 illustrates the result.

This design has two phases of data movement which alternate with each step. Regardless of the phase, data flowing south$\rightarrow$east in Figure 1, flows in the direction of time: It is remembered.

In phase A, data flowing south$\rightarrow$north in Figure 1, flows up through the array; data flowing west$\rightarrow$east in Figure 1, flows in the direction of time.

In phase B, data flowing south$\rightarrow$north in Figure 1, flows in the direction of time; data flowing west$\rightarrow$east in Figure 1, flows up through the array.

Of those spacetime embeddings that are time-optimal, this embedding also is space-minimal.

THEOREM 1 *Embedding E_2 of process dependence graph G_A is spacetime-optimal* [5].

We briefly mention two other spacetime-optimal embeddings of the process dependence graph of Figure 1. The first may be thought of as a modification of the McKeown array. To obtain it, the process dependence graph is embedded as follows:

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

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

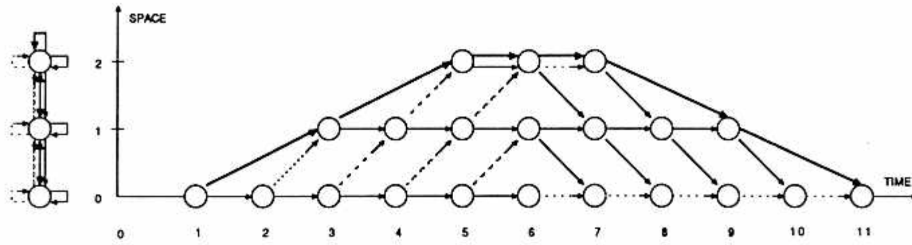

Figure 4 Spacetime-optimal embedding E_3 of G_A for $n=6$. Its spatial projection is a linear of $\lceil n/2 \rceil$ processors. In the figure $n=6$.

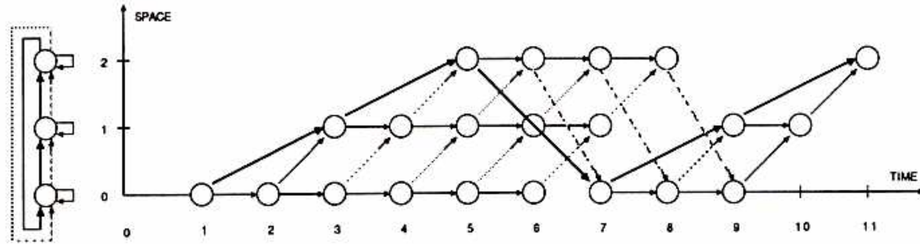

Figure 5 Spacetime-optimal embedding E_4 of G_A for $n=6$. Its spatial projection is a ring of $\lceil n/2 \rceil$ processors.

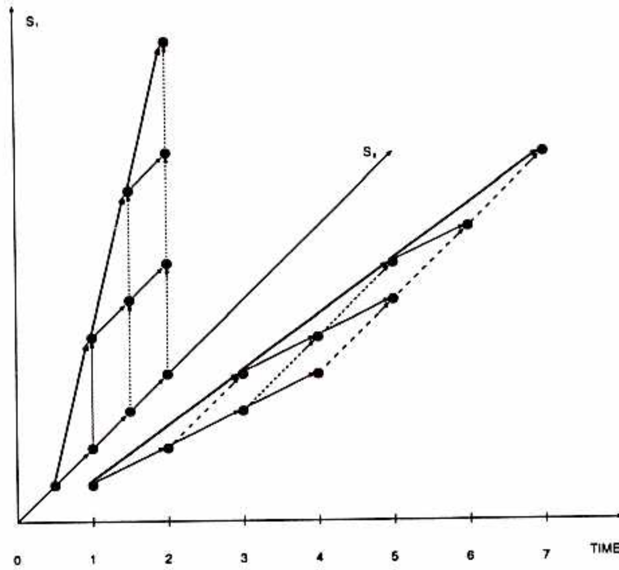

Figure 6 An embedding of G_A whose spatial projection is a triangular array of processors.

This embedding, E_3, is illustrated, for $n=6$, in Figure 4.

The second spacetime-optimal embedding is another variation of McKeown's array. We connect the endpoints of the linear array, making a *ring* of processors. More formally, we nonlinearly embed the process dependence graph as follows:

$$t := i + j;$$

$$s := i \bmod \left\lfloor \frac{n}{2} \right\rfloor.$$

This embedding, E_4, is illustrated, for $n=6$, in Figure 5. It 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 the above two embeddings use $2n-1$ steps and $\lceil n/2 \rceil$ processors, they too are spacetime-optimal.

6. A TWO-DIMENSIONAL ARRAY

We now present a *2-d* array for computing the process dependence graph of Figure 1. This is done by embedding the process dependence graph into a *3-d* space. One way to do this is with a linear embedding, E_5:

$$\begin{bmatrix} t \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}.$$

Figure 6 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 has indices whose sum is k executes its process at step k. Execution completes after $2n-1$ 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 step. Figure 7 is intended to illustrate the pipeline quality of this array; it shows two process dependence graphs embedded in spacetime such that execution of the 2nd starts one step after the 1st: Executing k such process dependence graphs uses $2n+k-2$ steps.

7. LARGE INTERPOLATION ON SMALLER ARRAYS

One of the well-known constraints in the VLSI implementation of algorithms is what we call the *extent problem*. Processor arrays are of fixed extent, and we must use them effectively. A parallel exists in strongly typed languages like Pascal,

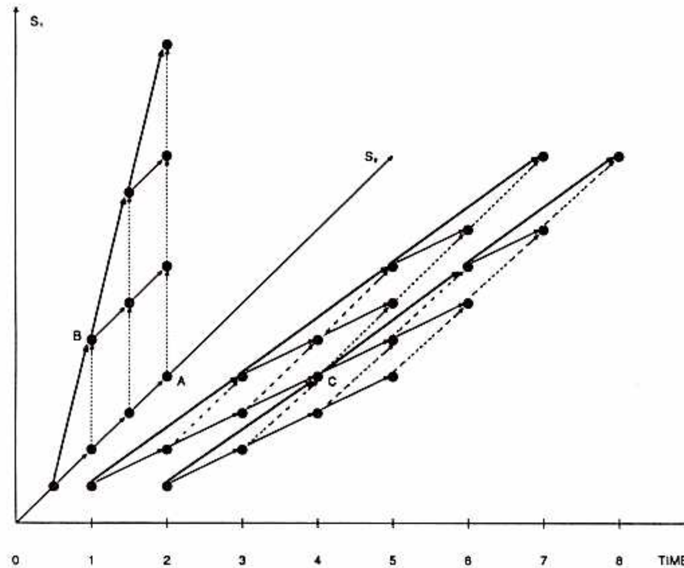

Figure 7 An embedding of two copies of G_A. There are two distinct processes that appear as though they are embedded in the same point (C) of spacetime. They, in fact, are embedded in distinct spatial coordinates. The process from the first copy of G_A executes on processor A while the process from the second copy of G_A executes on processor B.

where procedures and functions using arrays cannot handle arrays of different extent, even if the bounds are not exceeded. The software version of the problem is solved by making suitable improvements to the language-compiler capabilities. None of these methods can handle the hardware version of the problem, which requires a direct modification of the algorithm. The size of the problem rarely is fixed beforehand. There are three cases depending on whether the size of the problem is smaller than, equal to, or larger than the array:

Equal to In this case there should be no difficulty in using the processor array effectively.

Smaller than In this case, much depends on the structure of the algorithm. In most cases this situation is handled easily, by disabling some units (assuming this is possible). Hence, a less efficient use of the array results. There are situations, however, in which this is not possible. For example, when a processor array recirculates data, or uses wrap-around connections (e.g., Cannon's array for matrix product [4, 7]). Again, the situation is handled by suitably increasing the size of the problem with data elements that 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.

Larger than This is the most interesting situation. Heller's corollary [10] to Murphy's law states:

- No matter what special-purpose device is available, there is a problem too large for it.

- The problem will manifest itself only after the device is acquired and can no longer be modified.
- The problem cannot be ignored.

We thus are advised to consider this case in some detail. The difficulty of this case is directly proportional to the algorithm's degree of decomposability. At best, the algorithm can be decomposed into blocks whose size equals the extent of the processor array. In this case, partial results can be composed using a processor array of the same size (perhaps the same array). Work on general methods for this problem has been reported by Moldovan and Fortes [19]. Problem-specific methods also have been given considerable attention (see, e.g. Schreiber and Kuekes [23] and Schreiber [22]).

We now formulate Newton and Hermite polynomial interpolation as a system of linear equations of the following form:

$$\mathbf{N}\mathbf{c}=\mathbf{f} \tag{6}$$

where $\mathbf{N}\in F^{(n+1)\times(n+1)}$ and $\mathbf{c},\mathbf{f}\in F^{n+1}$. Using the standard representation of a polynomial, the matrix $\mathbf{N}$ is the transpose of a *Vandermonde* matrix. An interpolating polynomial in the standard representation thus can be found by solving a Vandermonde system of linear equations [3, 24]. An important property of this formulation is that it permits us to decompose a polynomial interpolation problem into smaller polynomial interpolations, and small matrix-vector products.

Traditionally, the Newton and Hermite interpolation problems are solved by using the Aitken and Neville recursions rather than by being formulating as linear algebra problems; the recursions are easy to program, and the memory requirements are modest: To compute the coefficients of an n-degree Newton interpolating polynomial, we use only $O(n^2)$ arithmetic operations, and $O(n)$ memory. We now show that when a Newton polynomial interpolation of size $n+1$ is cast as a linear algebra problem of the form (6), it can be decomposed into $q+1$ polynomial interpolations and matrix-vector products, each of size $p+1$, where $n+1=(p+1)(q+1)$.

This formulation and decomposition generalize to Hermite interpolation. We illustrate the decomposition for Newton interpolation. Here, $\mathbf{c}$ and $\mathbf{f}$ are vectors containing the divided differences, and the function values, respectively. That is $\mathbf{f}=[f_0,f_1,\ldots,f_n]^T$ and $\mathbf{c}=[c_0,c_1,\ldots,c_n]^T$ where $c_i=f_{012\ldots i}$ for $0\leqq i\leqq n$. The matrix $\mathbf{N}$ is lower-triangular, containing the prefix products of x_i-x_j, for $0\leqq i\neq j\leqq n$. We refer to $\mathbf{N}$ as the *Newton-Vandermonde* matrix. We illustrate the above as follows: If we write the interpolating polynomial in Newton form

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

then its coefficients can be obtained by solving

$$p_n(x_i)=f_i,\quad 0\leqq i\leqq n. \tag{7}$$

That is, we solve the following set of linear equations:

$$\begin{aligned}
&c_0 &&= f_0\\
&c_0 + c_1(x_1 - x_0) &&= f_1\\
&c_0 + c_1(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1) &&= f_2\\
& &&\vdots\\
&c_0 + c_1(x_n - x_0) + \quad \cdots \quad + c_n(x_n - x_0)\ldots(x_n - x_{n-1}) &&= f_n.
\end{aligned}$$

For notational simplicity let

$$\pi_{ij}^{k} = \prod_{r=j}^{k} (x_i - x_r) \tag{8}$$

for $1 \leqq i \leqq n$ and $0 \leqq j \leqq k < i$. The above linear system of equations, then, in matrix notation is

$$\begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ \pi_{10}^{0} & 1 & 0 & \ldots & 0 \\ \pi_{20}^{0} & \pi_{20}^{1} & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \pi_{n0}^{0} & \pi_{n0}^{1} & \pi_{n0}^{2} & \ldots & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \tag{9}$$

(or more compactly, $\mathbf{Nc} = \mathbf{f}$).

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

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

where

$$\mathbf{N}_{ij} = \begin{bmatrix} \pi_{i(p+1),0}^{j(p+1)} & \pi_{i(p+1),0}^{j(p+1)+1} & \ldots & \pi_{i(p+1),0}^{j(p+1)+p} \\ \pi_{i(p+1)+1,0}^{j(p+1)} & \pi_{i(p+1)+1,0}^{j(p+1)+1} & \ldots & \pi_{i(p+1)+1,0}^{j(p+1)+p} \\ \vdots & \vdots & \vdots & \vdots \\ \pi_{i(p+1)+p,0}^{j(p+1)} & \pi_{i(p+1)+p,0}^{j(p+1)+1} & \ldots & \pi_{i(p+1)+p,0}^{j(p+1)+p} \end{bmatrix} \tag{11}$$

and

$$\mathbf{c}_i=\begin{bmatrix} c_{i(p+1)} \\ c_{i(p+1)+1} \\ \vdots \\ c_{i(p+1)+p} \end{bmatrix}, \quad \mathbf{f}_i=\begin{bmatrix} f_{i(p+1)} \\ f_{i(p+1)+1} \\ \vdots \\ f_{i(p+1)+p} \end{bmatrix}. \tag{12}$$

Matrices $\mathbf{N}_{ii} \in F^{(p+1)\times(p+1)}$ are lower-triangular with 1's on the diagonal; the zero matrices are of the same dimension as the $\mathbf{N}_{ij}$ matrices. System (10) can be solved via forward substitution.

```
    Procedure Partition_and_Solve
    BEGIN
1:  c_0 = N_00^-1 f_0
    FOR i=1 TO q DO
        BEGIN
2:      e_i = f_i
        FOR j=0 TO i-1 DO
3:          e_i = e_i - N_ij c_j
4:      c_i = N_ii^-1 e_i
        END FOR
    END PROCEDURE.
```

This algorithm is *not* required to solve an arbitrary linear system. A linear system involving $\mathbf{N}_{ii}$ is equivalent to Newton interpolation for some pairs of points. For example, consider the solution to the system $\mathbf{N}_{00}\mathbf{c}_0=\mathbf{e}_0$. The solution of the system of equations

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

is equivalent to performing Newton interpolation with the pairs of the points (x_j, e_j), for $0 \leqq j \leqq p$, i.e.,

$$\mathbf{N}_{00}^{-1}\mathbf{e}_0=\mathbf{c}_0=\textbf{Newton_Interpolation}\ (x_j, e_j; 0 \leqq j \leqq p).$$

Now, we consider solving $\mathbf{N}_{ii}\mathbf{c}_i=\mathbf{e}_i$ for $1 \leqq i \leqq q$. Since

$$\mathbf{N}_{ii}=\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \pi_{i(p+1)+1,0}^{i(p+1)} & 1 & 0 & \dots & 0 \\ \pi_{i(p+1)+2,0}^{i(p+1)} & \pi_{i(p+1)+2,0}^{i(p+1)+1} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \pi_{i(p+1)+p,0}^{i(p+1)} & \pi_{i(p+1)+p,0}^{i(p+1)+1} & \pi_{i(p+1)+p,0}^{i(p+1)+2} & \dots & 1 \end{bmatrix} \tag{13}$$

an element of $\mathbf{N}_{ii}$ (below the diagonal) can be written as

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

for $0 \leqq s < r \leqq p$. Using definition (8), we have that

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

Thus, $\mathbf{N}_{ii}$ can be written as the product of a diagonal matrix and a triangular matrix:

$$\mathbf{N}_{ii} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ \pi_{i(p+1)+1,0}^{i(p+1)-1} \times \pi_{i(p+1)+1,i(p+1)}^{i(p+1)} & 1 & \cdots & 0 \\ \pi_{i(p+1)+2,0}^{i(p+1)-1} \times \pi_{i(p+1)+2,i(p+1)}^{i(p+1)} & \pi_{i(p+1)+2,0}^{i(p+1)-1} \times \pi_{i(p+1)+2,i(p+1)}^{i(p+1)+1} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ \pi_{i(p+1)+p,0}^{i(p+1)-1} \times \pi_{i(p+1)+p,i(p+1)}^{i(p+1)} & \pi_{i(p+1)+p,0}^{i(p+1)-1} \times \pi_{i(p+1)+p,i(p+1)}^{i(p+1)+1} & \cdots & 1 \end{bmatrix} = \mathbf{D}_i \mathbf{M}_{ii}$$

where

$$\mathbf{D}_i = diagonal\,[1, \pi_{i(p+1)+1,0}^{i(p+1)-1}, \pi_{i(p+1)+2,0}^{i(p+1)-1}, \ldots, \pi_{i(p+1)+p,0}^{i(p+1)-1}]$$

and

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

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

$$\mathbf{c}_i = \mathbf{N}_{ii}^{-1}\mathbf{e}_i = \mathbf{M}_{ii}^{-1}\mathbf{D}_{ii}^{-1}\mathbf{e}_i = \textbf{Newton_Interpolation}\,(x_j, d_j;\, i(p+1) \leqq j \leqq i(p+1)+p).$$

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

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

for $0 \leqq r, s \leqq p$. Thus, an element of the first column of N_{ij} is written as

$$\mathbf{N}_{ij}(r,0) = \pi^{j(p+1)}_{i(p+1)+r,0} = \prod_{k=0}^{j(p+1)} (x_{i(p+1)+r} - x_k)$$

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

$$= \mathbf{N}_{i,j-1}(r,p) \times (x_{i(p+1)+r} - x_{j(p+1)}). \tag{14}$$

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

$$\mathbf{N}_{ij}(r,s) = \prod_{k=0}^{j(p+1)+s} (x_{i(p+1)+r} - x_k)$$

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

$$= \mathbf{N}_{ij}(r,s-1) \times (x_{i(p+1)+r} - x_{j(p+1)+s}). \tag{15}$$

Using (14) and (15) we can obtain the process dependence graph for the computation of the entries of $\mathbf{N}_{ij}$. The resulting graph is a mesh of size $(p+1) \times (p+1)$, as depicted in Figure 8.

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

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

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

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

The 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 8. We depict the process dependence graph for computing

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

in Figure 9.

Thus, steps 1, 3, and 4 of **Procedure Partition_and_Solve** become

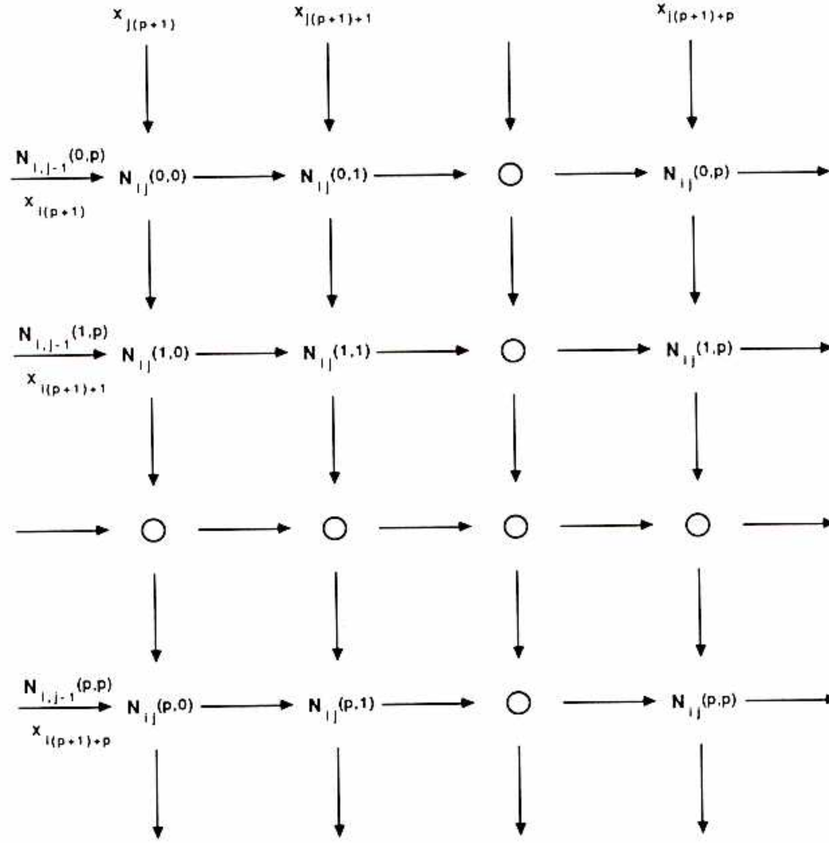

Figure 8 Process dependence graph for computation of $\mathbf{N}_{ij}(r,s)$ for $0 \leqq r, s \leqq p$.

1: $\mathbf{c}_0 =$ **Newton_Interpolation** $(x_j, f_j; 0 \leqq j \leqq p)$

3.1: **construct** $\mathbf{N}_{ij}(r,s)$ using $\mathbf{N}_{i,j-1}(r,p)$, $x_{j(p+1)+r}$, and $x_{j(p+1)+s}$ for $0 \leqq r, s \leqq p$

3.2: $\mathbf{e}_i = \mathbf{e}_i - \mathbf{N}_{ij}\mathbf{c}_j$

4.1: **construct** $\mathbf{D}_i(r,r)$ using x_k for $1 \leqq r \leqq p$ and $0 \leqq k \leqq i(p+1)+p$

4.2: $\mathbf{d}_i(0) = \mathbf{e}_i(0)$ and $\mathbf{d}_i(r) = \mathbf{e}_i(r)/\mathbf{D}_i(r,r)$ for $1 \leqq r \leqq p$

4.3: $\mathbf{c}_i =$ **Newton_Interpolation** $(x_j, d_j; i(p+1) \leqq j \leqq i(p+1)+p)$.

We explain this algorithm with an example. Let $n=5$, and choose $p=2$ and $q=1$. The Newton-Vandermonde matrix becomes

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ x_{10} & 1 & 0 & 0 & 0 & 0 \\ x_{20} & x_{20}x_{21} & 1 & 0 & 0 & 0 \\ x_{30} & x_{30}x_{31} & x_{30}x_{31}x_{32} & 1 & 0 & 0 \\ x_{40} & x_{40}x_{41} & x_{40}x_{41}x_{42} & x_{40}x_{41}x_{42}x_{43} & 1 & 0 \\ x_{50} & x_{50}x_{51} & x_{50}x_{51}x_{52} & x_{50}x_{51}x_{52}x_{53} & x_{50}x_{51}x_{52}x_{53}x_{54} & 1 \end{bmatrix}$$

where $x_{ij} = x_i - x_j$.

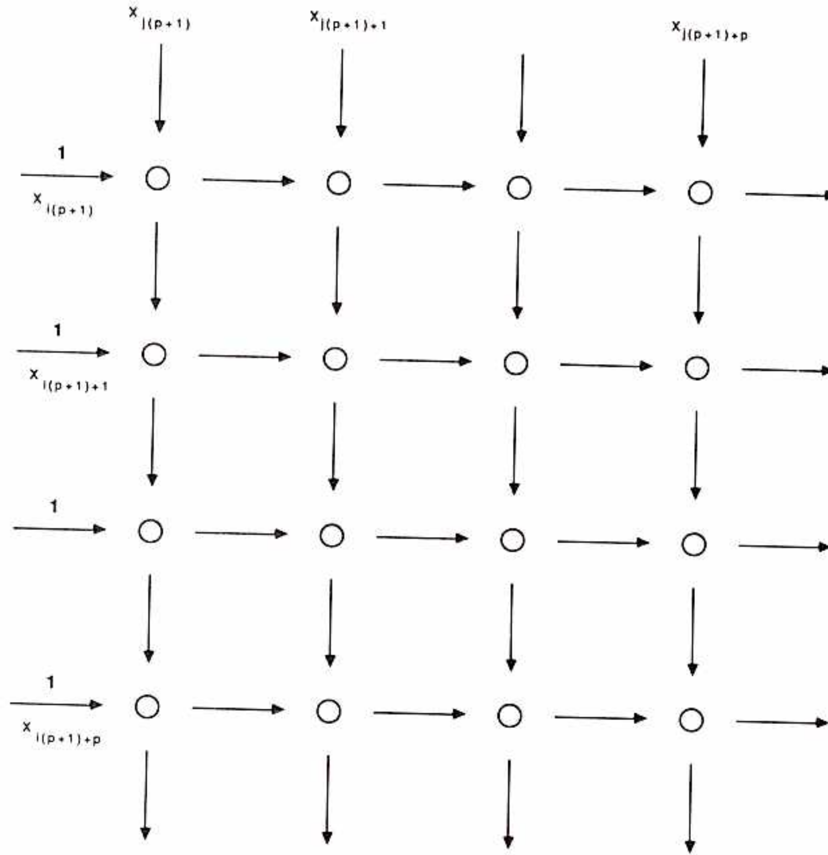

Figure 9 Process dependence graph for computation of $\prod_{k=j(p+1)}^{(j+1)(p+1)-1}(x_{i(p+1)+r}-x_k)$ for $1 \leqq r \leqq p$.

Step 1 The partitioning of this matrix yields the equation $\mathbf{N}_{00}\mathbf{c}_0=\mathbf{f}_0$, i.e.,

$$\begin{bmatrix} 1 & 0 & 0 \\ x_{10} & 1 & 0 \\ x_{20} & x_{20}x_{21} & 1 \end{bmatrix}\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}=\begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix}.$$

It follows from the above discussion that solution of the above system is found as

$$\mathbf{c}_0=[c_0, c_1, c_2]^T=\textbf{Newton_Interpolation}(x_0, x_1, x_2; f_0, f_1, f_2).$$

Step 3.1 We have $\mathbf{N}_{10}\mathbf{c}_0+\mathbf{N}_{11}\mathbf{c}_1=\mathbf{f}_1$. Thus, we need to construct matrix $\mathbf{N}_{10}$ using the data dependence graph in Figure 8, as explained earlier.

Step 3.2 We then compute $\mathbf{f}_1-\mathbf{N}_{10}\mathbf{c}_0=\mathbf{N}_{11}\mathbf{c}_1(=\mathbf{e}_1)$.

Step 4.1 According to the decomposition $\mathbf{N}_{11}=\mathbf{D}_1\mathbf{M}_{11}$, i.e.,

$$\begin{bmatrix} 1 & 0 & 0 \\ x_{40}x_{41}x_{42}x_{43} & 1 & 0 \\ x_{50}x_{51}x_{52}x_{53} & x_{50}x_{51}x_{52}x_{53}x_{54} & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & x_{40}x_{41}x_{42} & 0 \\ 0 & 0 & x_{50}x_{51}x_{52} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ x_{43} & 1 & 0 \\ x_{53} & x_{53}x_{54} & 0 \end{bmatrix},$$

we compute the elements of $\mathbf{D}_1$.

Step 4.2 Then $\mathbf{d}_1 = \mathbf{D}_1^{-1}\mathbf{e}_1$, i.e.,

$$[d_3, d_4, d_5]^T = \left[\frac{e_3}{\mathbf{D}_1(0,0)}, \frac{e_4}{\mathbf{D}_1(1,1)}, \frac{e_5}{\mathbf{D}_1(2,2)}\right]^T = \left[e_3, \frac{e_4}{x_{40}x_{41}x_{42}}, \frac{e_5}{x_{50}x_{51}x_{52}}\right]^T.$$

Step 4.3 Finally, we solve the system $\mathbf{N}_{11}\mathbf{c}_1 = \mathbf{d}_1$, i.e.,

$$\begin{bmatrix} 1 & 0 & 0 \\ x_{43} & 1 & 0 \\ x_{53} & x_{53}x_{54} & 1 \end{bmatrix} \begin{bmatrix} c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} d_3 \\ d_4 \\ d_5 \end{bmatrix}$$

to find $\mathbf{c}_1$, using Newton interpolation:

$$\mathbf{c}_1 = [c_3, c_4, c_5]^T = \textbf{Newton_Interpolation}(x_3, x_4, x_5; d_3, d_4, d_5).$$

These operations are summarized in Figure 10. Hence, the partitioning of the Newton-Vandermonde matrix leads to three types of computations:

Steps 1 and 4.3 Polynomial interpolation of size $p+1$.

Step 3.1 Construction of matrices $\mathbf{N}_{ij}$, which has a $(p+1)\times(p+1)$ mesh as its process dependence graph.

Step 3.2 Matrix-vector product operation $\mathbf{N}_{ij}\mathbf{c}_j$ which also has a $(p+1)\times(p+1)$ mesh as its process dependence graph (see, e.g., [14]).

Step 4.1 Construction of diagonal matrices $\mathbf{D}_i$, which has a $p\times p$ mesh as its process dependence graph.

This decompositioning, thus, yields process dependence graphs that can be embedded in spacetime, producing linear arrays of size $p+1$.

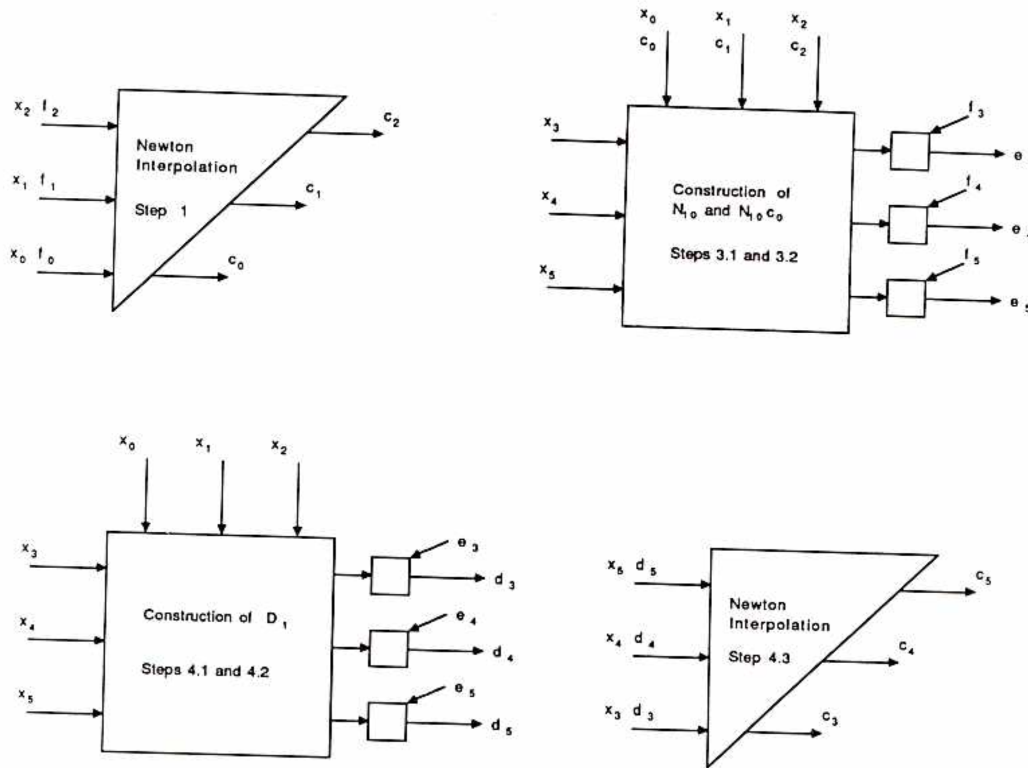

Figure 10 Large interpolation on smaller arrays.

8. DISCUSSION AND CONCLUSION

In this paper, we:

1) Briefly presented systolic designs for performing Aitken's method of iterated interpolation. All of them are time-optimal; several are spacetime-optimal.

Each of the various 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* [12], (2) the Warp [1], and (3) the Matrix-1 [9].

2) Cast the polynomial interpolation as a system of linear equations. Its corresponding matrix, the Newton-Vandermonde matrix, is then block-structured in order to partition the solution of the resultant system. Computing the coefficients of the system of n linear equations, and solving it, can be done conveniently on either smaller linear arrays (i.e., approximately $n^{1/2}$ processors), or smaller 2-d arrays (i.e., approximately $n^{1/2} \times n^{1/2}$ processors).

*Transputer is a trademark of INMOS, Ltd.

Our method of decomposing a large interpolation problem via the Newton-Vandermonde matrix generalizes to the Hermite case. In this case, each coefficient of the Newton-Vandermonde matrix becomes a submatrix of coefficients, creating a block structure within the already block-structured Newton-Vandermonde matrix. A presentation of details is tedious but straightforward, hence omitted.

References

[1] A. M. Annaratone, E. Arnould, T. Gross, H.-T. Kung, M. Lam, O. Menzilcioglu and J. Webb, The WARP computer: Architecture, implementation, and performance, *IEEE Trans. on Computers* **C-36,** 12 (1987), 1523–1538.

[2] I. S. Berezin and N. P. Zhidkov, *Computing Methods,* Vol. 1, Addison-Wesley, 1965.

[3] A. Bjorck and V. Pereyra, Solution of Vandermonde systems of equation, *Mathematics of Computation* **24,** 112 (1970), 893–903.

[4] L. E. Cannon, *A Cellular Computer to Implement the Kalman Filtering Algorithm,* Ph.D. Dissertation, Montana State University, 1969.

[5] P. R. Cappello, E. Gallopoulos and Ç. K. Koç, Systolic computation of interpolating polynomials, *Computing* **45,** 2 (1990), 95–117.

[6] P. R. Cappello and K. Steiglitz, Unifying VLSI array designs with linear transformations of space-time, in *Advances in Computer Research* (Preparata, F. P., ed.), JAI Press, 1984, Vol. 2, 23–65.

[7] E. Dekel, D. Nassimi and S. Sahni, Parallel matrix and graph algorithms, *SIAM Journal on Computing* **10,** 4 (1981), 657–675.

[8] J. A. B. Fortes and D. I. Moldovan, Parallelism detection and algorithm transformation techniques useful for VLSI architecture design, *J. Parallel Distrib. Comput.* **2** (1985), 277–301.

[9] D. E. Foulser and R. Schreiber, The Saxby matrix-1: A general-purpose systolic computer, *IEEE Computer* **20,** 7 (1987), 35–43.

[10] D. Heller, Partitioning big matrices for small systolic arrays, in *VLSI and Modern Signal Processing* (Kung, S. Y., Whitehouse, H. J. and Kailath, T., eds.), Prentice-Hall, 1985, 185–199.

[11] F. B. Hildebrand, *Introduction to Numerical Analysis,* McGraw-Hill, 1956.

[12] IMS T800 transputer, Rpt. 72 TRN 117 01, INMOS Ltd., Almondsbury, Bristol, UK, November 1986.

[13] F. Krogh, Efficient algorithms for polynomial interpolation and divided differences, *Mathematics of Computation* **24,** 109 (1970), 185–190.

[14] H. T. Kung and C. E. Leiserson, Algorithms for VLSI processor Arrays, in *Introduction to VLSI Systems* (Mead, C. and Conway, L., eds), Addison-Wesley, 1980, 271–292.

[15] G. P. McKeown, Iterated interpolation using a systolic array, *ACM Transactions on Mathematical Software* **12,** 2 (1986), 162–170.

[16] W. L. Miranker and A. Winkler, Spacetime representations of computational structures, *Computing* **32** (1984), 93–114.

[17] D. I. Moldovan, On the analysis and synthesis of VLSI algorithms, *IEEE Transactions on Computers* **C-31** (1982), 1121–1126.

[18] D. I. Moldovan, On the design of algorithms for VLSI systolic arrays, *Proc. IEEE* **71,** 1 (1983), 113–120.

[19] D. I. Moldovan and J. A. B. Fortes, Partitioning and mapping algorithms into fixed size systolic arrays, *IEEE Transactions on Computers* **C-35,** 1 (1986), 1–12.

[20] P. Quinton, Automatic synthesis of systolic arrays from uniform recurrent equations, *Proc. 11th Ann. Symp. on Computer Architecture* (1984), 208–214.

[21] S. K. Rao, *Regular Iterative Algorithms and Their Implementation on Processor Arrays,* Ph.D. Dissertation, Stanford University, October, 1985.

[22] R. Schreiber, Solving eigenvalue and singular value problems on an undersized systolic array, *SIAM J. on Scientific and Statistical Computing* **7,** 2 (1986), 441–451.

[23] R. Schreiber and P. K. Kuekes, Systolic linear algebra machines in digital signal processing, in *VLSI and Modern Signal Processing* (Kung, S.-Y., Whitehouse, H. J. and Kailath, T., eds.), Prentice-Hall, Englewood Cliffs, NJ, 1985.

[24] W. P. Tang and G. H. Golub, The block decomposition of a Vandermonde matrix and its applications, *BIT* **21,** 4 (1981), 505–517.

[25] N. K. Tsao and R. Prior, On multipoint numerical interpolation, *ACM Transactions on Mathematical Software* **4,** 1 (1978), 51–56.