SIAM J. COMPUT.
Vol. 19, No. 1, pp. 133-142, February 1990

A FAST PARALLEL HORNER ALGORITHM*

MICHAEL L. DOWLING†

Abstract. The simple Horner algorithm solves the problem of evaluating a polynomial of degree d with n indeterminates; in this paper it is shown that its implementation on a parallel computer with $O(d)$ processors can achieve a complexity of $2\lceil \log_2 (d+1)\rceil \cdot (\lceil \log_2 n\rceil + 1)$. If, in addition, the evaluation of all partial derivatives is also sought, then the full Horner algorithm solves this problem on a parallel computer with $O(d^{n+1})$ processors, achieving a parallel complexity of $2\lceil \log_2 (d+1)\rceil \cdot (\lceil \log_2 (d+1)\rceil + \lceil \log_2 n\rceil + 1)$.

Key words. parallel algorithms, algebraic complexity, parallel polynomial evaluation

AMS(MOS) subject classifications. 68C25, 68C05, 68B10

1. Introduction. This article applies a technique for parallelising sequential programs to the problem of polynomial evaluation. Given a program implementation of the Horner algorithm, it is shown that sufficient information can be obtained from its semantics for the program to be reconstructed with a high degree of parallelisability. The reconstruction process has two phases, the first of which is to find optimal, parallel hyperplanes in the loops. The possible values that the index variables can take for a given loop constitute a subset A of integral n-space $\mathbf{N}^n$, where n is the number of nestings. An optimal hyperplane is an hyperplane in $\mathbf{N}^n$ containing no data dependencies, the details of which are given below, and that has a minimal number of translates that contain at least one element of A. Once such an hyperplane H has been found, the loop is reorganised so that each iteration of the outer most loop corresponds to iterating over all those indices in A belonging to a translate of H. The result of this phase is merely to reorder the sequence in which the iterations are performed, but so that all the inner loops can be executed simultaneously, for each possible value of the index variable in the outer loop. Since the same operations are being performed, the result of the first phase has no effect on the numerical stability of the algorithm. Moreover, the balance between the number of additions and multiplications enjoyed by the Horner algorithm is preserved, thereby making the transformed code well suited to execution on a vector computer with separate and independent functional units for addition and multiplication.

The second phase is to represent the values of the various iterations of the code, after the hyperplane transformation has been performed, as the solution of a lower triagonal, linear system of equations. One then applies a variant of the standard, numerical, cyclic reduction algorithm to solve this system in logarithmic time. Since the technique used in this paper operates primarily on program code, it has much wider applicability than merely to polynomial evaluation. With the Horner algorithm, however, the linear system of equations that one obtains is known explicitly, so that solutions can be computed very quickly.

Hitherto, the main problem with using vector and parallel computers was that the Horner algorithm is difficult to parallelise. As a result, much work has been devoted to finding completely new, parallel algorithms (cf. [2, p. 162]). Although such algorithms generally achieve logarithmic complexity, it is not usually possible to implement them

* Received by the editors February 16, 1988; accepted for publication (in revised form) April 11, 1989.

† Abteilung für Mathematische Optimierung, Institut für Angewandte Mathematik, Carolo-Wilhelmina Universität zu Braunschweig, Pockelsstrasse 14, D-3300 Braunschweig, Federal Republic of Germany (I1041301@DBSTU1.BITNET).

efficiently on extant computer hardware. In contrast, the first phase of the parallelisation process presented in this paper is very effective for vector computers, while further benefit can also be obtained by implementing the second phase for vector computers with several processors.

Since parallel algorithms are constructed from sequential ones, the objective of this paper is similar to that in [6], where it was shown that, if a sequential algorithm requires k operations, then there is a parallel version that requires $O(\log_2(d)\cdot\log_2(k))$ parallel steps. That result was improved by Valiant et al. in [14], where it was shown that the same complexity can be achieved with the use of $O((kd)^\alpha)$ processors for some constant α, as opposed to the $O(k^{\log_2 d})$ required by Hyafil. The most efficient lower bound known is $\max\{\log_2 d, \log_2 k\}$ (cf. [2]).

This paper is organised as follows. Section 2 introduces the method of hyperplane parallelisation for the univariate, full Horner algorithm. Here the degree of program loop nesting is only two, so that hyperplanes are merely straight lines, thereby making the basic technique readily comprehensible without introducing unnecessarily complicated terminology. It is this section that also introduces the notion of flow dependence developed by Banerjee in [1]. The idea of using hyperplanes to parallelise sequential code originated from [11], and was developed further in [3]. Having whole hyperplanes of iterations execute concurrently is essential to applying the cyclic reduction technique introduced in § 3, which begins with the univariate, simple Horner algorithm, where the application of cyclic reduction is particularly direct. This section ends by applying the same procedure to the full Horner algorithm after hyperplane parallelisation has already been applied. The remaining two sections apply these techniques to the bivariate case. Unfortunately, Horner evaluation for polynomials with n indeterminates requires n nested DO-loops; the idea of treating the bivariate case explicitly while only alluding to the general case is therefore designed to simplify otherwise excessively complicated notation. Proofs of the statements concerning arbitrary numbers of indeterminates can readily be supplied using induction.

2. The hyperplane parallelisation. Recall that if $p(x)=a_0+a_1x+\cdots+a_dx^d$ is a polynomial function of the single variable x over the real number field, then the simple Horner algorithm evaluates p at a point x according to the bracketing scheme:

$$p(x)=a_0+(a_1+\cdots+(a_{d-2}+(a_{d-1}+a_d\cdot x)\cdot x)\;\cdots)\cdot x.$$

This can be expressed as a recurrence formula:

$$\begin{aligned}&b_{d+1}=0 \quad \text{(initialisation)},\\ &b_j=a_j+b_{j+1}\cdot x, \qquad j=d,\cdots,0.\end{aligned}$$

The full Horner algorithm also evaluates all the derivatives $p^{(i)}(x)$ at the point x, for $i=0,\cdots,d$, and amounts to a d-fold repetition of the simple Horner algorithm. The resulting recurrence formulae are given below:

$$\left.\begin{aligned}&b_j^{(-1)}=a_j \quad \text{for } j=0,1,\cdots,d\\ &b_{d+1}^{(i)}=0 \quad \text{for } i=-1,0,\cdots,d\end{aligned}\right\}\quad \text{(initialisation)}$$

and

$$b_j^{(i)}=b_j^{(i-1)}+b_{j+1}^{(i)}\cdot x \quad \text{where } j=d,d-1,\cdots,i, \text{ and } i=0,1,\cdots,d.$$

Ultimately, $i|b_i^{(i)}=p^{(i)}(x)$, where $p^{(i)}(x)$ denotes the ith derivatives of the polynomial p, and $i=0,1,\cdots,d$; details can be found in [4]. Figure 1 depicts a possible FORTRAN implementation, where it is presumed that the array B has already been initialised.

```
      DO 1, I = 0, D
        DO 1, J = D-1, I, -1
1         B(I,J) = B(I-1,J)+X*B(I,J+1)
```

FIG. 1. *A naïve Horner code.*

Remark. Since the dependencies amongst the data determine the parallelisability of a code segment, it is necessary to include an explicit implementation here. If, for example, the array B(I,J) above were to be coded as B(J), the code would still be correct. However, the price for memory optimisation quite often is diminished performance as a parallel algorithm, as in the case in point (cf. [3]).The more redundancy there is in the representation of the data in a parallel computer, the more ways there are of addressing them without risk to the data's integrity.

The loop in Fig. 1 is clearly not amenable to parallel execution as each iteration requires the values of the previous iterations in order to proceed. It can, however, be restructured for concurrent execution in $2(d+1)$ parallel steps. To show this, the notion of data dependence is required.

DEFINITION. Two statements, s and t, are said to be *flow dependent* if s executes before t, and t uses a value computed during the execution of s.

In [1], Banerjee introduced three notions of data dependence, one of which is that of flow dependence. Since the other two do not occur in the above code, they shall not be discussed here. The key interest in data dependencies results from a theorem, due to Banerjee, where it is shown that if the statements of a block of code were to be permuted in a manner respecting the order of execution of dependent statements, then both the original and the permuted code will always produce the same output if given the same input. In particular, where there are no data dependencies present, the order of execution is immaterial, so that there is no obstacle to concurrent execution. These data dependencies have been exploited to good practical advantage (cf. Kuck et al. [10]), and also provide a means of treating parallelisation problems theoretically (cf. [3]).

If one considers the graph whose nodes correspond to the various iterates of a loop such as that shown in Fig. 1, and whose directed edges correspond to data dependencies, then the graph corresponding to the Horner code has nodes labelled by pairs (i, j), where $i = 0, 1, \cdots, d$ and $i \leqq j \leqq d$, and where there are flow dependence edges from nodes $(i-1, j)$ to (i, j), and also from $(i, j+1)$ to (i, j), whenever these ordered pairs correspond to nodes. The graph corresponding to a polynomial of degree five is illustrated in Fig. 2.

From this diagram, it is obvious why the implementation of the Horner code above does not admit concurrent execution; the nested loop executes down each column in turn, from left to right. As such, the loop iterates *precisely* along lines of data dependencies, so that sequential execution is obligatory. By executing along the diagonal lines, $i-j=\text{const.}$, the obstruction to parallel execution vanishes. In this simple, two-dimensional case, these lines constitute what more generally shall be called *optimal hyperlines.* Transforming the loop so that these diagonal lines are parallel to the j-axis, say via the unimodular transformation defined by setting $k=j-i$ and $l=j$, circumvents the problem. The resulting code is therefore parallelised. It can readily be seen that this transformation is indeed optimal in the sense of minimising the number of parallel steps. The resulting code is given in Fig. 3.

Note that now the inner DO-loop has been completely freed of data dependencies so that, for each value of the outer index K, the entire inner loop can be computed in

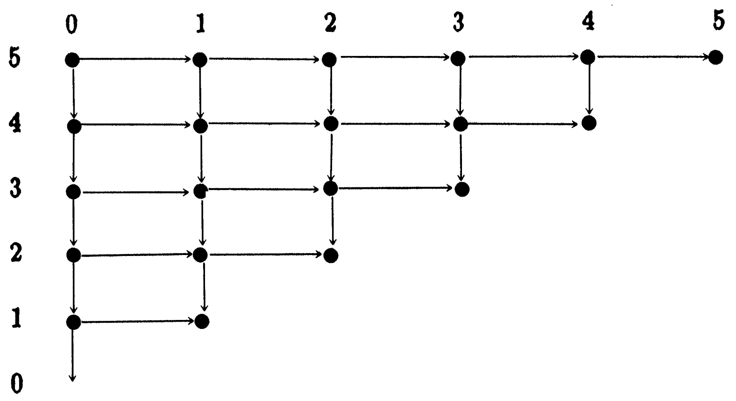

FIG. 2. *The dependence graph of the full Horner code.*

two parallel steps. The first step performs the multiplication while the second uses the result of the multiplication to compute the sum, and this for *all* possible values of L, for the current value of K. Execution is therefore completed in $2(d+1)$ parallel steps, whereas at most $d+1$ processing elements are required. This revised code is exactly equivalent to the original, so that it even yields the same, numerically insignificant digits as the naïve version. It shall soon be seen that this basic idea can be applied to multivariate polynomials to show that the full multivariate Horner algorithm also can be programmed with merely linear parallel complexity.

```
      DO 1, K = D, 0, -1
        DO 1, L = K, D-1
1         B(L-K,L)=B(L-K-1,L) + X*B(L-K, L+1)
```

FIG. 3. *A revised Horner code.*

3. A logarithmic reduction for univariate polynomials. The linear recurrence formula for the simple, univariate Horner algorithm amounts to solving the bidiagonal linear system of equations in Fig. 4.

The obvious solution is to observe that $b_d = a_d$, and $b_i = a_i + x \cdot b_{i+1}$, for each $i < d$. This is the method used in both the code segments above, but, as before, each iteration depends upon the value computed during the previous one, thereby precluding any parallel processing. The solution is to apply the cyclic reduction algorithm for solving tridiagonal systems in logarithmic time (cf. [8], [13] for details). For bidiagonal systems, cyclic reduction uses each odd numbered row to eliminate the off-diagonal entry in the even numbered row immediately below it. The result of this single parallel step is indicated in Fig. 5.

$$\begin{pmatrix} 1 & & & & & \\ -x & 1 & & & & \\ & -x & 1 & & & \\ & & -x & 1 & & \\ & & & \ddots & \ddots & \\ & & & & -x & 1 \end{pmatrix} \cdot \begin{pmatrix} b_d \\ b_{d-1} \\ b_{d-2} \\ \vdots \\ b_0 \end{pmatrix} = \begin{pmatrix} a_d \\ a_{d-1} \\ a_{d-2} \\ \vdots \\ a_0 \end{pmatrix}$$

FIG. 4. *The simple Horner linear system.*

$$\begin{pmatrix} 1 & & & & & & \\ 0 & 1 & & & & & \\ & -x & 1 & & & & \\ & -x^2 & 0 & 1 & & & \\ & & & -x & 1 & & \\ & & & -x^2 & 0 & 1 & \\ & & & \ddots & \ddots & \ddots & \ddots \end{pmatrix} \cdot \begin{pmatrix} b_d \\ b_{d-1} \\ b_{d-2} \\ b_{d-3} \\ b_{d-4} \\ b_{d-5} \\ \vdots \end{pmatrix} = \begin{pmatrix} a_d \\ a_{d-1}+xa_d \\ a_{d-2} \\ a_{d-3}+xa_{d-2} \\ a_{d-4} \\ a_{d-5}+xa_{d-4} \\ \vdots \end{pmatrix}$$

FIG. 5. *The first step of cyclic reduction.*

Note that now each odd numbered variable can be computed in a single, parallel step once the even numbered variables are known. These, on the other hand, are decoupled from the odd numbered variables, and satisfy a bidiagonal system of equations of half the original size. Repeating the process, one readily sees that $4\lceil \log_2 (d+1)\rceil$ parallel steps are required for the process to terminate, the factor of four corresponding to the simultaneous multiplications and subsequent subtractions, and to the fact that $\lceil \log_2 (d+1)\rceil$ iterations are required to reduce the system of linear equations to the trivial system containing only a single unknown. The same number of iterations are required for the subsequent substitutions, so that the total number of iterations is $2\lceil \log_2 (d+1)\rceil$, each requiring two parallel steps. Moreover, the steps requiring the most processors are the first and the last, both of which require $\lfloor (d+1)/2 \rfloor$.

Remark. The application of cyclic reduction above is essentially the binary splitting algorithm of Dorn (cf. [2, p. 132]).

One notes that, since powers of x accumulate in the off-diagonal entries as the computation progresses, the cyclic reduction version of the Horner algorithm is only stable for $|x| \leqq 1$. In contrast, the standard procedure will produce better results whenever the coefficients decrease sufficiently rapidly as the degree increases. On the other hand, where $|x| < 1$, it is not always necessary to continue the recursion until the bidiagonal system has been reduced to a scalar equation. Once x^n has been reduced to a value smaller than the rounding errors, the bidiagonal system may be regarded as being diagonal, and so soluble in a single, parallel step.

The revised code for the full Horner algorithm executes the whole lines, $j-i=k$, concurrently, while the input data required for each iteration are computed during the $(k-1)$st iteration. The result is essentially a bidiagonal system, but with vector rather than scalar unknowns, so that cyclic reduction can now be applied. More precisely, the revised code can be written in FORTRAN-8X style as follows:

```
      DO 1, K = D, 0, -1
    1   B(*-K,*) = B(*-(K+1),*) + X*B((*+1)-(K+1), (*+1)),
```

where `B(*-K,*)` corresponds to the $(d-k+1)$-dimensional vector `B(L-K,L)`, and where `L=K, ··· ,D`. (`B(-1,K)` has the value a_k.) Let $\mathbf{b}^{(k)}$ and $\mathbf{a}^{(k)}$ denote the $(d+2)$-dimensional vectors

$$\mathbf{b}_l^{(k)} = \begin{cases} \text{the value of B(L,K+L)} & \text{if } 0 \leqq \text{l} \leqq \text{d}-\text{k}, \\ 0 & \text{if } d-k+1 \leqq l \leqq d, \end{cases} \quad \text{and}$$

$$\mathbf{a}_l^{(k)} = \begin{cases} a_k & \text{if } l = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Here, k and l denote the values of K and L, respectively. The code segment above corresponds to the following vector recurrence formula

$$\mathbf{b}_l^{(k)} = \mathbf{b}_{l-1}^{(k+1)} + x\mathbf{b}_l^{(k+1)} + \mathbf{a}^{(k)}, \qquad k = d, d-1, \cdots, 0, \quad l = 1, \cdots, d-k,$$

which in turn can be written in matrix form as in Fig. 6 where I denotes the $(d+1)\times(d+1)$ identity matrix, and S is the shift operator, given by $S(\mathbf{v})_i=\mathbf{v}_{i+1}$ for $i<d+1$, and $S(\mathbf{v}_{d+2})=0$. Also, $\mathbf{a}^{(k)}$ is the vector whose sole, nonzero component is $\mathbf{a}_0^{(k)}=a_k$. In particular, $(i\,|\,\mathbf{b}_0)_i=p^{(i)}(x)$, the ith derivative of p.

$$\begin{pmatrix} I & & & & \\ -(S+xI) & I & & & \\ & -(S+xI) & I & & \\ & & \ddots & \ddots & \\ & & & -(S+xI) & I \end{pmatrix} \cdot \begin{pmatrix} \mathbf{b}^{(d)} \\ \mathbf{b}^{(d-1)} \\ \mathbf{b}^{(d-2)} \\ \vdots \\ \mathbf{b}^{(0)} \end{pmatrix} = \begin{pmatrix} \mathbf{a}^{(d)} \\ \mathbf{a}^{(d-1)} \\ \mathbf{a}^{(d-2)} \\ \vdots \\ \mathbf{a}^{(0)} \end{pmatrix}$$

FIG. 6. *A bidiagonal system for the full algorithm.*

Applying cyclic reduction blockwise, the number of blocks is halved during each reduction, so that the reduction phase terminates after $\lceil \log_2(d+1)\rceil$ iterations. During the kth reduction, the matrix $-(S+xI)^{2^k}$ accumulates in the subdiagonal blocks. This is the lower triagonal matrix whose rth lower subdiagonal contains the rth term in the expansion of $(x+1)^{2^k}$. All of these matrices can be computed in a single, parallel step from the currently computed entries. The number of nonzero entries in the right-hand side vector blocks doubles during each iteration so that during the kth iteration, $(S+xI)^{2^k}\mathbf{a}^{(2^{k-1})}+\mathbf{a}^{(2^k)}$ requires 2^k additions, and hence k parallel steps using cyclic doubling.

The second phase of the block, cyclic reduction algorithm entails the back substitution of $-(S+xI)^{-2^{k-1}}(\mathbf{a}^{(2^k)})$ in the 2^{k-1} blocks; a process that again involves 2^k additions, and hence k parallel steps. Note that the matrix entries of $-(S+xI)^{-2^k}$ are known, namely,

$$-(S+xI)^r_{ij}=\begin{cases}(-1)^{i-j+1}\dbinom{r+i-j-1}{i-j-1}x^{r+j-i} & \text{if } i\geqq j,\\ 0 & \text{otherwise.}\end{cases}$$

This follows easily from the well-known formula,

$$\binom{a}{b}=\binom{a-1}{b-1}+\binom{a-2}{b-2}+\cdots+\binom{b-1}{b-1}.$$

The resulting parallel complexity for the full, univariate Horner Algorithm is, therefore,

$$2(1+2+\cdots+\lceil\log_2(d+1)\rceil)=\lceil\log_2(d+1)\rceil(\lceil\log_2(d+1)\rceil+1)$$

where the factor of two results from considering both the phases required by cyclic reduction. Note that the maximal number of processors is $\lfloor (d+1)/2\rfloor$. During the first step, $\lfloor (d+1)/2\rfloor$ additions and multiplications are performed, one for every second block. Thereafter, the number of blocks is halved during each step, while the number of additions doubles, until $2^{\lceil (d+1)/2\rceil}$ additions are performed in a single block. (Identical blocks do not have to be computed more than once.)

4. The simple, multivariate, Horner algorithm. Evaluation of polynomials in several variables hinges on the fact that any element $f\in\mathbf{R}[x_1,\cdots,x_n]$ can be regarded as an element of $\mathbf{R}[x_1,\cdots,x_{n-1}][x_n]$, the ring of all polynomials in the single indeterminate, x_n, with coefficients in the polynomial ring $\mathbf{R}[x_1,\cdots,x_{n-1}]$. The preceding discussion of the univariate case therefore starts an inductive procedure for evaluating multivariate polynomials. Moreover, this applies equally to the simple algorithm as to the full Horner algorithm, which additionally computes all the partial derivatives.

The simple bivariate Horner algorithm for evaluating the polynomial

$$p(x, y) = \sum_{i+j=0}^{d} a_{ij} x^i y^j,$$

now corresponds to the linear recurrence formulae:

$$\left.\begin{aligned} &b_{ij} = 0 \quad \text{for } i+j = d+1, \\ &b_{ij} = a_{ij} + x \cdot b_{i+1j} \quad \text{for } i = d-j, d-j-1, \cdots, 1, \\ &b_{0j} = b_{0j} + x \cdot b_{1j} + y \cdot b_{0j+1}, \end{aligned}\right\} \quad j = d, d-1, \cdots, 0,$$

the computational part of which can be naïvely implemented as in Fig. 7.

```
   DO 1, J = D-1, 0, -1
     DO 2, I = D-J-1, -1
2      B(I,J) = A(I,J) + X*B(I+1,J)
1    B(0,J) = B(0,J) + Y*B(0,J+1)
```

FIG. 7. *The naïve, bivariate Horner code.*

The data dependency graph for this nested loop has vertices in bijective correspondence with the lower, triangular region $\{(i, j) | 0 \leqq i \leqq j \leqq d\}$, having flow dependencies from left to right between every pair of adjacent, horizontal vertices, and from top to bottom between every pair of adjacent, vertical vertices lying on the 0th vertical column. A corresponding, parallelising procedure similar to that of the full, univariate Horner can now be applied again. This time, one readily recognizes that the lines $i+j = \text{const.}$ are devoid of data dependencies, and provide optimal parallelisation in that no other choice of lines has fewer parallel translates with nonvoid intersections with the vertex set. A possible, parallelising transformation therefore corresponds to the unimodular change of variables $k = i+j$, and $l = j$. The transformed code now takes the form shown in Fig. 8.

```
   DO 1, K = D, 0, -1
     DO 2, L = 0, K
2      B(K-L,L) = A(K-L,L) + X*B(K-L+1,L)
1    B(0,K) = B(0,K) + Y*B(0,K+1)
```

FIG. 8. *The revised, bivariate Horner code.*

Note that each iteration of K requires three successive steps: the simultaneous multiplications, $x \cdot b_{k-l+1,l}$, for each value of l, concurrently with the multiplication of y with b_{0l+1}; the simultaneous additions of the products to the $a_{k-l,l}$; and finally, the addition on the last line. In general, where there are n indeterminates, all the multiplications can still be performed simultaneously, while the n additions can be performed in logarithmic time. The resultant complexity of the parallelised algorithm is therefore $(d+1)(\lceil \log_2 n \rceil + 1 \rceil)$, using $d+1$ processors.

The revised Horner code admits another interpretation as a linear recurrence formula as follows. Define $e^{(k)}$ and $a^{(k)}$ to be $(d+1)$-dimensional vectors by setting

$$e_l^{(k)} = \begin{cases} \text{values } \texttt{B(L,K-L)} & \text{if } l < k. \\ 0 & \text{otherwise,} \end{cases} \quad \text{and}$$

$$a_l^{(k)} = \begin{cases} a_{l,k-l} & \text{if } l < k, \\ 0 & \text{otherwise.} \end{cases}$$

Then, for $k=d, d-1, \cdots, 0$,

$$e_l^{(k)} = xe_{l+1}^{(k+1)} + a_l^{(k)} \quad \text{for } l=0,1,\cdots,k-1, \quad \text{and}$$
$$e_k^{(k)} = ye_{k+1}^{(k+1)} + xe_k^{(k+1)} + a_k^{(k)},$$

which, in turn, can be represented as a block, tridiagonal system of linear equations, with the $(d+1)\times(d+1)$ identity matrices I appearing along the main diagonal, the diagonal matrices $-xI$ along the first subdiagonal, and with $-yE_{dd}$ along the second subdiagonal. Here, E_{dd} denotes the matrix whose only nonzero element is a one in the bottom right-hand corner.

Applying cyclic reduction blockwise, it is not difficult to see that each reduction gives rise to a new system having half the number of blocks, but with $-x^kI$ and $-y^kE_{dd}$ accumulating along the first and second subdiagonals, respectively. The main difference between the bivariate and univariate cases is that, for the bivariate case, the components of the $e_k^{(k)}$ are coupled with *two* others, so that an extra addition is required for the back substitutions. More generally, evaluating a degree d polynomial in n indeterminates entails an n fold coupling, and hence an additional $\lceil \log_2 n \rceil$ parallel steps during the back substitution phase.

For polynomials with n indeterminates, the iterations corresponding to values of the index variables lying on the parallel hyperplanes $H_k = \{(i_1, \cdots, i_q) \mid \sum_{q=1}^n i_q = k\}$ depend only upon those of the previous hyperplanes H_{k+1}, so that cyclic reduction applies and reaches completion after $2\lceil \log_2 (d+1) \rceil$ iterations. Each iteration requires a single, parallel multiplication step, and n additions. The resulting complexity of the logarithmically reduced algorithm is therefore $2\lceil \log_2 (d+1)\rceil \cdot (\lceil \log_2 n \rceil + 1)$; at most $O(d^n)$ processors are required.

5. The full, multivariate, Horner algorithm. Computing all the derivatives of the bivariate polynomial p of the last section corresponds to implementing the following, linear recurrence formulae:

$$b_{jk}^{(-1)} = a_{jk} \quad \text{for all } 0 \leqq j \leqq k \leqq d,$$
$$b_{jk}^{(i)} = 0 \quad \text{for } 0 \leqq j \leqq k \leqq d+1,\ k=0,\cdots,d \text{ and for } i=-1,\ j+k=d+1,$$

and

$$\left.\begin{array}{ll} b_{jk}^{(i)} = b_{jk}^{(i-1)} + x\cdot b_{j+1k}^{(i)}, & 0 \leqq k \leqq d-i, \quad i<j \leqq d-k, \\ b_{ik}^{(i)} = b_{ik}^{(i-1)} + x\cdot b_{ii+1}^{(i)} + y\cdot b_{ik+1}^{(i)}, & 0 \leqq k \leqq d-i, \\ b_{jk}^{(i)} = b_{jk}^{(i-1)} + y\cdot b_{jk+1}^{(i)}, & 0 \leqq j \leqq i-1, \quad d-j \leqq k \leqq i-j \end{array}\right\} 0 \leqq i \leqq d.$$

It is not difficult to show that $i|j|b_{ij}^{(i+j)} = \partial^{i+j}p/\partial x^i \partial y^j$. The significant section of such recurrence formulae can be programmed as in Fig. 9.

```
      DO 1, I = 0, D
        DO 2, K = 0, D-I
          DO 2, J = D-K, I + 1, -1
2           B(I,J,K) = B(I-1,J,K) + X*B(I,J+1,K)
        DO 3, J = I-1, 0, -1
3         B(I,J,K) = B(I-1,J,K) + X*B(I,J+1,K) + Y*B(I,J,K+1)
        DO 1, J = 0, I-1
          DO 1, K = D-J, I-J, -1
1           B(I,J,K) = B(I-1,J,K) + Y*B(I,J,K+1)
```

FIG. 9. *The naïve, full, bivariate Horner code.*

The vertices of the corresponding data dependence graph correspond to the region in $\mathbf{N}^3$ bounded by the planes $i=0$, $j=0$, $k=0$, $j+k=d$, and $j+k-i=0$. There are flow dependencies between adjacent vertices from the back to the front, and from the top to the bottom, and also from the right to the left. The optimal, parallelising hyperplanes are the hyperplanes $j+k-i=r$, for some constant $0 \leqq r \leqq d$. In the general case, the corresponding hyperplanes are given by the equation $\sum_{q=1}^{n} i_q - i = \text{const}$. The hyperplane parallelisation procedure reduces the parallel complexity of the full, multivariate Horner algorithm to $d+1$ iterations, each of which requires $\lceil \log_2 n \rceil$ parallel additions (cf. Fig. 10 below).

```
      DO 1, R = D, 0, -1
        DO 1, S = R, D
          DO 1, T = 0, S
1           B(S-R,S-T,T) = B(S-R-1,S-T,T) + X*B(S-R,S-T+1,T) + Y*B(S-R,S-T,T,T+1)
```

FIG. 10. *The revised, full, bivariate Horner code.*

To apply cyclic reduction once again, it is first necessary to represent the values of B(R,S,T) appropriately as a vector, whereupon one observes that the resulting linear difference equation is a block banded system, with n subdiagonal blocks, each of which is a shift operator. Although the details are now unpleasant, it can now nevertheless be seen that $2\lceil \log_2(d+1)\rceil(\lceil \log_2(d+1)\rceil + \lceil \log_2 n \rceil + 1)$ parallel steps are required, while using $O(d^{n+1})$ processors.

6. Conclusion. The arguments presented here are evidence for the effectiveness of considering the data themselves as the measure of parallelisability of an algorithm, and the use of dependence graphs in algorithm analysis. The univariate and bivariate linearised algorithms were both implemented on the Cray-XMP in Berlin, where, as predicted, not only were the revised codes fully vectorised, but they also yielded the same results to the point of replicating the numerically insignificant digits of the naïve code.

The table below shows the timing results of the naïve and revised univariate Horner codes. The former predictably has a quadratic execution time, whereas the latter is almost linear. Any deviation from linearity is due to the fact that a vector computer is not a genuine, parallel computer, since it still executes its instructions strictly sequentially. Since the Cray compiler is not capable of vectorising more than just the innermost loop, it is not sensible to time algorithms whose codes have a higher nesting order, with two or more inner loops parallelised. For this reason, the full logarithmic complexity of the multivariate evaluation algorithm could not be tested. The times below are given in microseconds.

Degree:	16	32	48	64	80	96	112	128	144	160	178	192
Naïve:	67	218	454	776	1183	1676	2253	2916	3665	4500	5419	6432
Revised:	24	49	78	110	149	187	229	273	323	376	431	488

REFERENCES

[1] U. BANERJEE, *Speed-up of ordinary programs*, Ph.D. thesis, University of Illinois, 1979.
[2] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[3] M. L. DOWLING, *A mathematical theory for code parallelisation*, Ph.D. thesis, Carolo-Wilhelmina Universität zu Braunschweig, Braunschung, FRG, 1987.

[4] P. HENRICI, *Applied and Computational Complex Analysis, Vol.* 1, John Wiley, New York, 1974.

[5] W. G. HORNER, *Philosophical Transactions of the London Mathematical Society*, 109 (1819), pp. 308-335.

[6] L. HYAFIL, *On the parallel evaluation of multivariate polynomials*, SIAM J. Comput., 8 (1979), pp. 120-123.

[7] D. E. KNUTH, *The Art of Computer Programming, Vol.* 2: *Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.

[8] D. KERSHAW, *Solution of single, tridiagonal systems and vectorisation of the* ICCG-*Algorithm on the* Cray-1, in Parallel Computations, G. Rodrigue, ed., Academic Press, New York, 1982.

[9] L. KRONSJÖ, *Algorithms: Their Complexity and Efficiency*, 2nd ed., John Wiley, New York, 1987.

[10] D. J. KUCK, R. H. KUHN, B. LEASURE, AND M. WOLFE, *Advanced, retargetable vectoriser*, IEEE Tutorial for Super-Computers: Design and Applications, K. Hwang, ed., 1984, pp. 186-203.

[11] L. LAMPORT, *The parallel execution of* DO-*loops*, Comm. ACM, 17 (1974), pp. 83-93.

[12] W. RÖNSCH, *Stability aspects in using parallel algorithms*, Parallel Comput., 1 (1984), pp. 75-98.

[13] G. RODRIGUE, N. MADSEN, AND J. KARUSH, *Odd-even reduction for banded linear equations*, J. Assoc. Comput. Mach., 26 (1979), pp. 72-81.

[14] L. G. VALIANT, S. SKYUM, S. BERKOWITZ, AND C. RACKOFF, *Fast parallel computation of polynomials using few processors*, SIAM J. Comput., 12 (1983), pp. 641-644.