

Computer Science 6365

April 22, 2008

## Lecture #27: Linear Recurrences

Professor: S. Lennart Johnsson

TA: Wei Ding

# 1 Recurrences

A linear, first order recurrence is a problem of the form

$$x(j) = a(j)x(j-1) + y(j), \quad x(1) = y(1) \quad (x(0) = 0), \quad j = \{1, 2, \dots, P\}.$$

Linear recurrences of this form arises in, for instance, the solution of bidiagonal systems of equations, as shown below

$$\begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -a(2) & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -a(3) & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & -a(4) & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -a(5) & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & -a(6) & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -a(7) & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -a(8) & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ x_P \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ y_P \end{bmatrix}$$

Such a bidiagonal system of equations corresponds to the forward solution of a system  $Ax = y$  after LU-factorization of  $A$ , where  $A$  is a tridiagonal system of equations and no pivoting is used. The solution of the upper bidiagonal system of equations,  $Ux = y$  corresponds to a backward running recurrence. In matrix form we have

$$\begin{bmatrix} u(1,1) & u(1,2) & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & u(2,2) & u(2,3) & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & u(3,3) & u(3,4) & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & u(4,4) & u(4,5) & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & u(5,5) & u(5,6) & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & u(6,6) & u(6,7) & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & u(7,7) & u(7,8) \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & u(8,8) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ x_8 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ y_8 \end{bmatrix}$$

For  $U$  in the factorization  $A = LU$  of a tridiagonal matrix, both diagonals of  $U$  assume arbitrary values. In recurrence form, the upper bidiagonal system of equations can be written as

$$u(j,j)x(j) + u(j,j+1)x(j+1) = y(j), \text{ where } x(P) = y(P)/u(P,P), \quad j = \{P-1, P_2, \dots, 1\}.$$

or

$$x(j) = -\frac{u(j, j+1)}{u(j, j)}x(j+1) + \frac{y(j)}{u(j, j)}, \text{ where } x(P) = y(P)/u(P, P), \quad j = \{P-1, P-2, \dots, 1\}.$$

Let  $\frac{u(j, j+1)}{u(j, j)} = -b(j)$  and  $\frac{y(j)}{u(j, j)} = y'(j)$ . Then,

$$x(j) = b(j)x(j+1) + y'(j), \text{ where } x(P) = y'(P), \quad j = \{P-1, P-2, \dots, 1\}.$$

and we have derived the backward running linear recurrence. The scaling of the matrix and the right hand side corresponds to the multiplication of the system of equations  $Ux = y$  by a diagonal matrix  $D$ , where  $d_j = 1/u(j, j)$ .

Linear recurrences also appears in the computation of eigenvalues of symmetric tridiagonal matrices.

In practice, it is often the case that multiple recurrences must be computed for a collection of right hand sides in the matrix formulation. For instance, in the solution of tridiagonal systems of equations, the right hand side may correspond to a particular loading condition. But, it is often of interest to consider many such conditions. Thus, several recurrence relationships are solved with the same set of coefficients. In matrix notation, the solution of multiple recurrences with the same coefficients can be written

$$BX = Y,$$

where  $B$  is a bidiagonal matrix of size  $P \times P$ .  $X$  and  $Y$  are matrices of shape  $P \times R$ , where  $R$  is the number of recurrences, or the number of right hand sides.

Even more interesting from an applications point of view is the solution of multiple recurrences with different recurrence coefficients. Such a need occurs in the solution of partial differential equations in several cases. For instance, we have already seen that the approximation of a second derivative with a second order centered difference approximation yields a tridiagonal matrix. Indeed, a regular discretization of Cartesian product domains in two or more dimensions and second order accurate centered differences of second order partial derivatives result in separate tridiagonal matrices for each grid line along which derivatives are formed. The solution of such systems of equations by, for instance, the *Alternating Direction Method* [18] or the *Fourier Analysis Cyclic Reduction method* (FACR) [2, 7, 19, 20] results in the need to solve multiple tridiagonal systems of equations with, in general, different coefficients for each of these systems. The coefficients represents local properties of the medium, which may vary with the location.

For multidimensional problems, it may be required to solve recurrence relationships along the different axes of the arrays. This is indeed the case for the *Alternating Direction Methods* for the solution of tridiagonal systems of equations. In distributed memory architectures, as well as in architectures with a banked memory system, the data reference pattern along multiple axis raises some interesting problems with respect to data distribution [13, 12, 11].

Kogge and Stone [15] devised an efficient algorithm for the solution of general recurrence relations. The algorithm is known as recursive doubling. In fact, most of their derivation is based on binary trees, while the actual algorithm they present has a different data reference pattern. We refer to the binary tree algorithm as *odd–even cyclic reduction* and to the other as *parallel cyclic reduction* [8]. We will also discuss a technique for the solution of multiple recurrences, *balanced cyclic reduction* [10]. When there are more variables (equations) than there are processing nodes, then we can emulate a large machine on the smaller machine at hand, or devise algorithms that have an explicit sequential component in addition to the parallel component. We discuss so-called *substructuring* as a method to combine sequential and parallel processing. Thus, we will discuss

- odd–even cyclic reduction
- parallel cyclic reduction
- balanced cyclic reduction
- substructuring

We have already seen a special case of a linear recurrence, namely the evaluation of all ordered partial sums of the leading  $j$  elements of a one–dimensional array, or

$$x(j) = \sum_{i=1}^j y(i), \quad j = \{1, 2, \dots, P\}.$$

If the array  $y$  has the values [1, 2, 3, 4, 5, 6], then the resulting array  $x$  has the values [1, 3, 6, 10, 15, 21].

This special linear recurrence for which  $a(j) \equiv 1$  is a *prefix* or *scan* computation, where the prefix operator in this case is  $+$ . With the prefix being multiplication the recurrence becomes

$$x(j) = \prod_{i=1}^j y(i), \quad j = \{1, 2, \dots, P\}.$$

Other prefixes are, for instance, max and min. We have already presented one way of implementing prefix operations in parallel, *parallel prefix*, on complete binary trees. That implementation in the terminology above is an odd–even cyclic reduction algorithm. The fact that all coefficients  $a(j)$  are equal to 1 considerably simplifies the computations. Thus, we will present the algorithms in the context of parallel prefix computations before discussing algorithms for general linear recurrences.

Higher order recurrences occurs often in practice. For instance, a filter with  $m$  weights convolves  $m$  elements, and form an  $m$ th order recurrence.

$$x(j) = a(j, 1)x(j - 1) + a(j, 2)x(j - 2) + \dots + a(j, j - m)x(j - m) + y(j)$$

Similarly, LU-factorization of a banded system of equations results in factors  $L$  and  $U$ , each of which defines a higher order recurrence relationship. Thus, for  $b_l$  lower diagonals in  $L$ , the systems of equations  $Lz = y$  defines a  $b_l$ th order forward linear recurrence. Similarly, with  $b_u$  upper diagonals in  $U$ , the system  $Ux = z$  defines an  $b_u$ th order backward linear recurrence.

Higher order linear recurrences can be converted into first order linear *matrix* recurrences, as opposed to *scalar* recurrences that we have discussed so far.

## 2 First order linear recurrences

### 2.1 Parallel Prefix computations

We will first consider the case where there are as many processing nodes as there are variables. Then, we will consider the solution of recurrence relations with multiple variables per node, followed by the solution of multiple recurrences.

There are two commonly used parallel techniques to solve linear recurrences. We refer to one as *odd-even cyclic reduction* and to the other as *parallel cyclic reduction*.

#### 2.1.1 Odd-Even Cyclic Reduction

The odd-even cyclic reduction algorithm uses a divide-and-conquer approach to compute the results. The algorithm consists of a *reduction phase* and a *backsubstitution phase*. The basic idea in the reduction phase is to reduce the number of equations by a factor of two for each step of the algorithm. This reduction in odd-even cyclic reduction is accomplished by recursively eliminating odd, or even, numbered equations and variables. The data interactions of the algorithm are illustrated in Figure 1.

#### Algorithm description

We illustrate the use of the odd-even cyclic reduction algorithm for parallel prefix evaluation for the  $+$ -prefix.

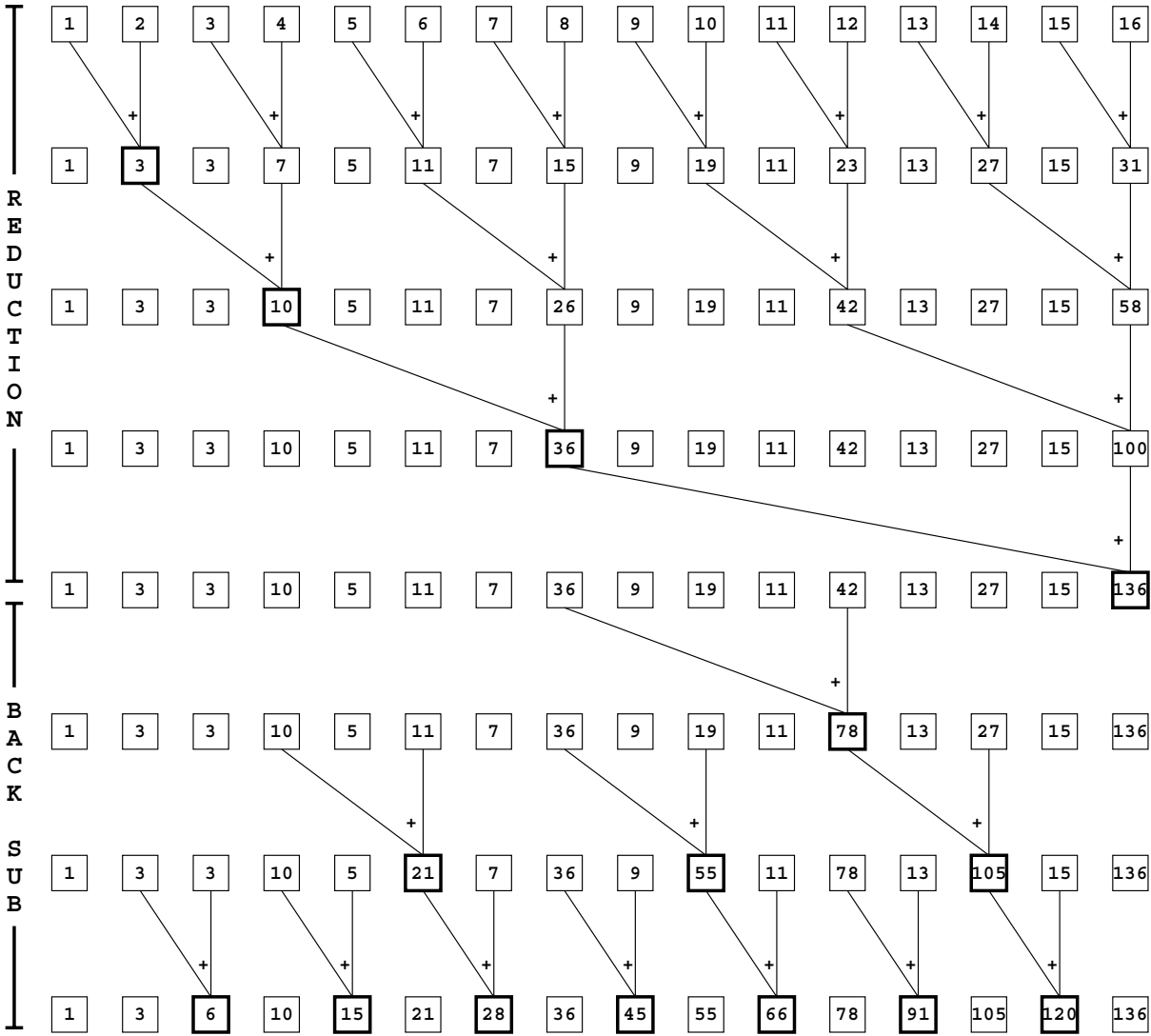


Figure 1: Odd–even cyclic reduction applied to a linear recurrence.

$$\begin{aligned}
 x(0) &= 0 \\
 x(j) &= x(j-1) + y(j) & j = \{1, 2, 3, \dots, P\} \\
 x(j) &= x(j-2) + \underbrace{y(j) + y(j-1)}_{y^{(1)}(j)} & j = \{2, 4, 6, \dots, P\} \\
 x(j) &= x(j-4) + \underbrace{y^{(1)}(j) + y^{(1)}(j-2)}_{y^{(2)}(j)} & j = \{4, 8, 12, \dots, P\} \\
 &\vdots \\
 &\vdots \\
 &\vdots \\
 x(P) &= x(0) + \underbrace{y^{(p-1)}(P) + y^{(p-1)}(P - P/2)}_{y^{(p)}(P)}
 \end{aligned}$$

For simplicity, let  $P = 2^p$ . Computing the result in-place, the reduction computations are defined by

```

for  $k = 1$  to  $p$  do
  for  $j = 2^k$  step  $2^k$  to  $2^p$  do
     $y(j) \leftarrow y(j) + y(j - 2^{k-1})$ 
  endfor
endfor

```

The above set of equations evaluates  $x(2^i) = y^{(i)}(2^i)$  for  $i = \{1, 2, 3, \dots, p\}$  in  $p$  steps. The remaining  $P - p - 1$  values are computed through backsubstitution. Ignoring the superscripts on  $y$ , the backsubstitution process is

$$\begin{aligned}
 x(3 \cdot 2^{p-2}) &= y(2 \cdot 2^{p-2}) + y(3 \cdot 2^{p-2}) \\
 x(j \cdot 2^{p-3}) &= y((j-1) \cdot 2^{p-3}) + y(j \cdot 2^{p-3}) & j = \{3, 5, 7\} \\
 x(j \cdot 2^{p-4}) &= y((j-1) \cdot 2^{p-4}) + y(j \cdot 2^{p-4}) & j = \{3, 5, 7, \dots, 15\} \\
 &\vdots \\
 &\vdots \\
 &\vdots \\
 x(j) &= y(j-1) + y(j) & j = \{3, 5, 7, \dots, P-1\}
 \end{aligned}$$

or formally for the in-place algorithm

```

for  $k = p - 2$  to 0 do
  for  $j = 3$  step 2 to  $2^{p-k} - 1$  do
     $y(j \cdot 2^k) \leftarrow y(j \cdot 2^k) + y((j - 1)2^k)$ 
  endfor
endfor
    
```

The indices used in the backsubstitution are just the “odd” elements missing in the reduction phase for each  $k$ . For example, in the reduction phase the indices for  $k = 2$  are  $\{4,8,12,16,\dots\}$ , while in the backsubstitution phase they are  $\{6,10,14,18,\dots\}$ .

An example for  $P = 8$  is shown below. The entries in the columns represent current values of the components of  $y$ . Values in boldface represent values computed in the step represented by the row. Thus,  $y(2)$ ,  $y(4)$ ,  $y(6)$ , and  $y(8)$  are updated in the first reduction step,  $y(4)$  and  $y(8)$  in the second step, etc.

$y(1)$	$y(2)$	$y(3)$	$y(4)$	$y(5)$	$y(6)$	$y(7)$	$y(8)$	
1	2	3	4	5	6	7	8	Initial
1	<b>3</b>	3	<b>7</b>	5	<b>11</b>	7	<b>15</b>	Reduction Phase, $k=1, j=2,4,6,8$
1	3	3	<b>10</b>	5	11	7	<b>26</b>	Reduction Phase, $k=2, j=4,8$
1	3	3	10	5	11	7	<b>36</b>	Reduction Phase, $k=3, j=8$
1	3	3	10	5	<b>21</b>	7	36	Backsubstitution Phase, $k=1, j=6$
1	3	<b>6</b>	10	<b>15</b>	21	<b>28</b>	36	Backsubstitution Phase, $k=0, j=3,5,7$

### Matrix representation of odd–even cyclic reduction

Let the original system of equations be  $Bx = y$ , where  $B$  is a lower bidiagonal matrix with all entries on the main diagonal being 1, and all entries on the first subdiagonal being -1. The first reduction step for a +–prefix can be represented by the matrix  $R_1$ .

$$R_1 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \end{bmatrix}$$

operating on the matrix equation from the left, i.e.,

$$\begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ x_P \end{bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ y_P \end{bmatrix}$$

The matrix  $B$  is modified in the first reduction step to  $B_1 = R_1 B$ , where

$$B_1 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 0 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 0 & 1 \end{bmatrix}.$$

The matrix  $R_2$  for the second reduction step is defined by

$$R_2 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 \end{bmatrix}$$

and  $B_2 = R_2 B_1$ , where

$$B_2 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 0 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot \\ \cdot & \cdot & \cdot & -1 & \cdot & 0 & 0 & 1 \end{bmatrix}.$$

For  $N = 8$  the last reduction matrix is  $R_3$ , which is easily seen to be

$$R_3 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 \end{bmatrix}.$$

and  $B_3 = R_3 B_2$ , or

$$B_3 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 0 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot \\ \cdot & \cdot & \cdot & 0 & \cdot & 0 & 0 & 1 \end{bmatrix}.$$

From this equation, it is clear that  $x_P = x_8$  can be determined from the last equation. It is also readily seen that  $x_2$  is available from  $B_1x = R_1y$ , and  $x_4$  from  $B_2x = R_2R_1y$ .

$$R_2R_1 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & 1 & 1 \end{bmatrix}.$$

$$R_3R_2R_1 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

The backsubstitution matrices  $S_2$  and  $S_1$  in our example are

$$S_2 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

and

$$S_1 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}.$$

It follows that

$$S_2B_3 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 0 & 0 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot \\ \cdot & \cdot & \cdot & 0 & \cdot & 0 & 0 & 1 \end{bmatrix}$$

and

$$S_2R_3R_2R_1 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

and that  $x_6$  can now be derived from  $S_2B_3x = S_2R_3R_2R_1y$

Furthermore,

$$S_1S_2B_3 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 0 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 0 & 0 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 0 & 1 & \cdot \\ \cdot & \cdot & \cdot & 0 & \cdot & 0 & 0 & 1 \end{bmatrix}$$

and

$$S_1S_2R_3R_2R_1 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & 1 & 1 & \cdot & \cdot \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdot \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

and  $x_3$ ,  $x_5$  and  $x_7$  can now be derived from  $S_1S_2B_3x = x = S_1S_2R_3R - 2R_1y$ .

Clearly,  $B^{-1} = S_1S_2R_3R_2R_1$ , since  $S_1S_2B_3 = S_1S_2R_3R_2R_1B = I$ .

### Arithmetic complexity

The number of steps in the reduction phase is  $p$ , and the number of steps in the backsubstitution phase, is  $p - 1$ . The total number of steps is  $2p - 1$ . The number of arithmetic operations in step  $k$ ,  $k = \{1, 2, 3, \dots, p\}$  of the reduction phase is  $2^{p-k}$ . In the backsubstitution phase the number of arithmetic operations in step  $k$ ,  $k = \{p - 2, p - 3, \dots, 0\}$  is  $2^{p-k-1} - 1$ . Thus, the total number of arithmetic operations is  $P - 1$  for the reduction phase, and  $P - \log_2 P - 1$  for the backsubstitution phase. The grand total for odd–even cyclic reduction is  $2P - \log_2 P - 2$ . The obvious sequential algorithm requires  $P - 1$  additions.

Parallel algorithms that require more operations than the corresponding sequential algorithm are known as *inconsistent* algorithms. Many parallel algorithms are inconsistent with their sequential counterpart. Odd–even cyclic reduction for parallel prefix computations is inconsistent by about a factor of two. The fact that a parallel algorithm is inconsistent with respect to a sequential algorithm is often most significant when the resources are limited and resource contention arises, as for instance when a large parallel machine is emulated on a smaller parallel machine.

### Communication complexity

The total number of elements communicated between nodes in step  $k$  of the reduction phase is  $2^{p-k}$ , and in step  $k$  of the backsubstitution phase the number of communicated elements is  $2^{p-k-1} - 1$ . Thus, the total number of elements being moved is  $2P - \log_2 P - 2$ . The total *bandwidth*, i.e., the total number of communication links traversed for the required data motion depends upon the topology of the network and the mapping of the index space to the nodes of the network.

**Remark:** The above parallel algorithm is also suitable for vector computers. In the first reduction step, the vector length is  $P/2$ ; in the second it is  $P/4$ , etc. The vector length is reduced by a factor of two for each step. In the backsubstitution, the vector length in step  $k$  is  $2^{p-1-k} - 1$ . The total number of vector operations is equal to the number of steps, or  $2 \log_2 P - 1$ .

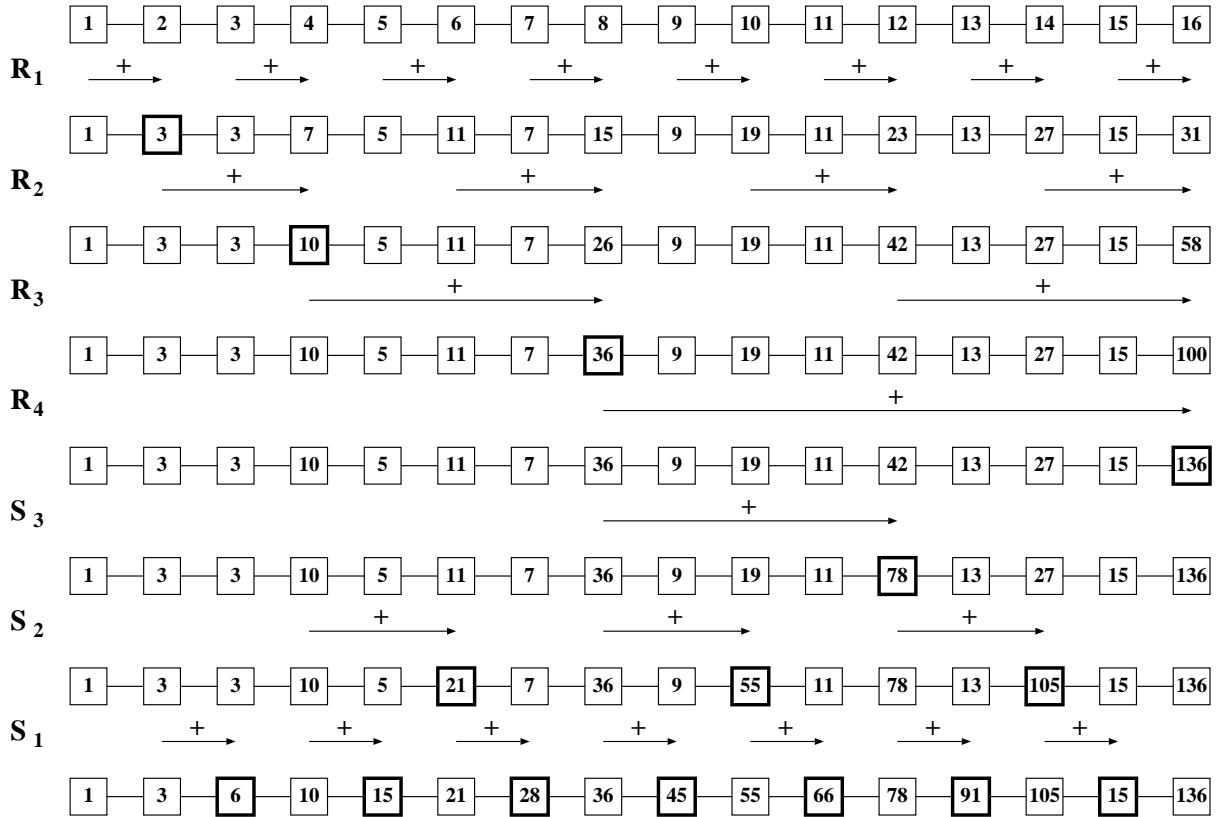


Figure 2: Solving a first order linear recurrence on a linear array through odd–even cyclic reduction.

### 2.1.2 Odd–even cyclic reduction on a linear array

In a *linear array* with one data element per node and elements mapped to nodes based on their index, each element must traverse a distance of  $2^{k-1}$  in step  $k$  of the reduction phase, as shown in Figure 2. For the backsubstitution phase, each element must traverse a distance of  $2^k$  in step  $k$ .

Thus, on a linear array the bandwidth requirement is:

$$\text{Reduction phase: } \sum_{k=1}^{\log_2 P} 2^{p-k} 2^{k-1} = \frac{P}{2} \log_2 P.$$

$$\text{Backsubstitution: } \sum_{k=\log_2 P-2}^0 2^{p-k-1} 2^k = \frac{P}{2} (\log_2 P - 1).$$

$$\text{Total bandwidth required: } P \log_2 P - \frac{P}{2}.$$

Note that even though the communication paths may be as long as  $P/2$ , there is no contention for communication channels. In a packet switched communication system, the communication time is proportional to the distance data must move in the absence of contention. Thus, in a packet switched system, the communication time for the reduction phase is proportional to  $P - 1$ , while for the backsubstitution phase it is proportional to  $P/2 - 1$ .

Thus, in a packet switched communication system for which the communication time dominates

over the arithmetic time, the speedup is  $O(1)$  and the efficiency is  $O(\frac{1}{P})$ .

In a communication system based on circuit switching or wormhole routing, the communication time in the absence of contention is of the form  $c_p d + c_b M$ , where  $c_p d$  is the time to establish the path of length  $d$ , and  $c_b M$  is the time to transfer a message of size  $M$ . If  $c_p d \ll c_b M$ , then the total time is of order  $O(p)$ , and the speedup is  $O(\frac{P}{\log_2 P})$  and the efficiency is  $O(\frac{1}{\log_2 P})$ . However, for large linear arrays and only one element per node, it is unlikely that  $c_p d \ll c_b M$ , and a linear array with circuit switching or wormhole routing behaves asymptotically as if packet switching is used. The large diameter of the linear array limits the performance, the speedup, and the efficiency.

### 2.1.3 Odd–even cyclic reduction on a complete binary tree

A binary tree implementation of parallel prefix computations by odd–even cyclic reduction is shown in Figure 3. It is clear that each step requires unit distance communication. The bandwidth requirement is  $2(2P - 1)$ , which is a factor of  $\frac{1}{4} \log_2 P$  lower than for a linear array implementation.  $2 \log_2 P - 1$  arithmetic steps are required, just as on the linear array.

A binary tree algorithm is as follows:

```

if leaf node then
    send value to parent
    upon receipt of data from parent
        add it to the local value
if internal node then
    add the value received from the left child
        to the value received from the right child
    send the result to parent
    if the node is not on the path from the
        leftmost leaf to the root then
        upon receipt of a value from the parent node
            add it to the value received from the left child
            and forward the result to the right child
        forward the value received from the parent to the left child
    else forward the value received from the left child to the right child
if root node
    forward the value received from the left child to the right child
  
```

Note that the above algorithm requires fewer arithmetic operations than the algorithm depicted in Figure 3.

The arithmetic complexity is the same as for a linear array. The communication complexity is proportional to  $2 \log_2 P$ , whether the communication system is packet switched or circuit switched. Note however, that even though this bound is lower than for the linear array for packet switched communication, the difference may not be as large as indicated by the bounds.

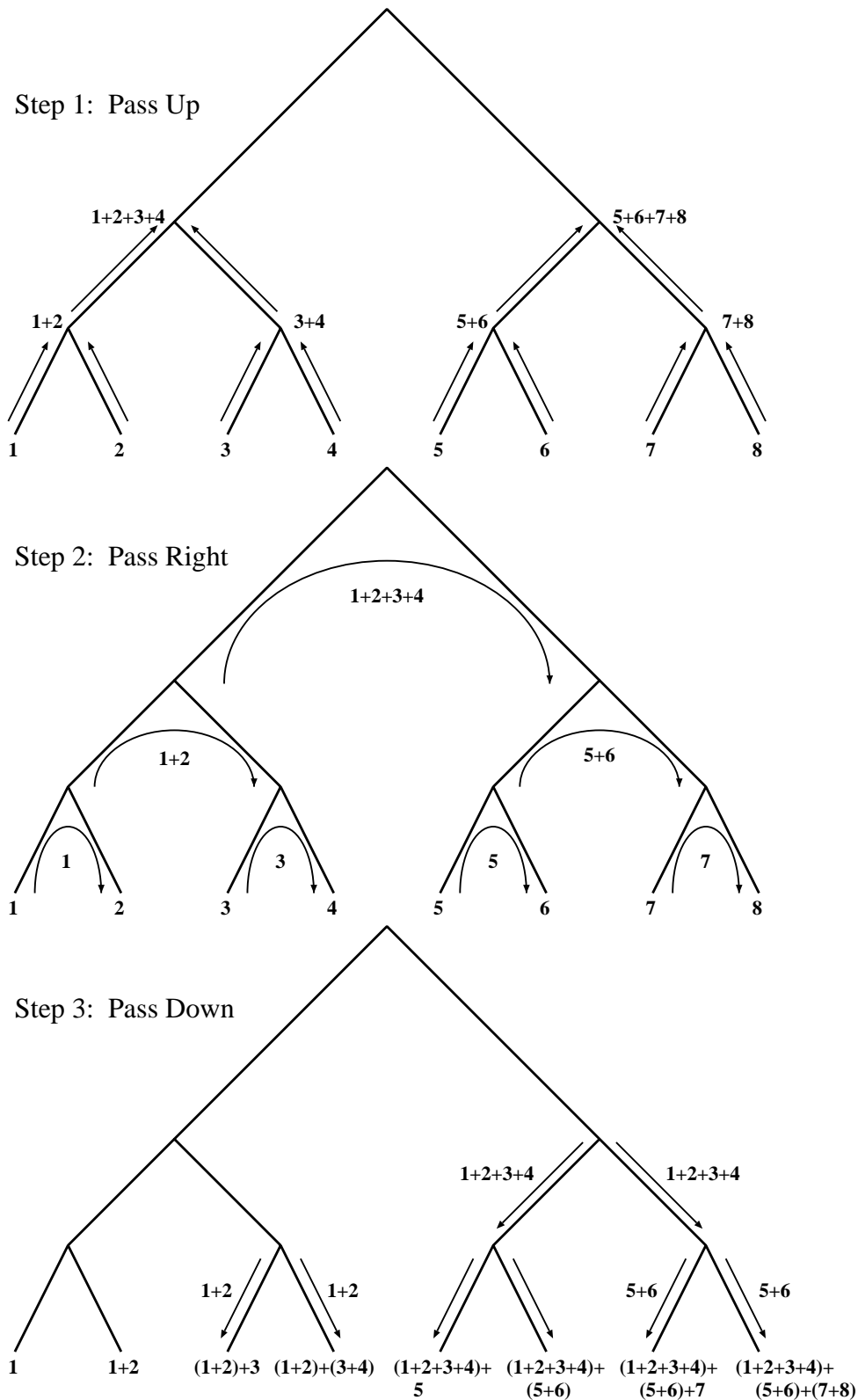


Figure 3: Application of odd–even cyclic reduction to the solution of a linear recurrence on a complete binary tree.

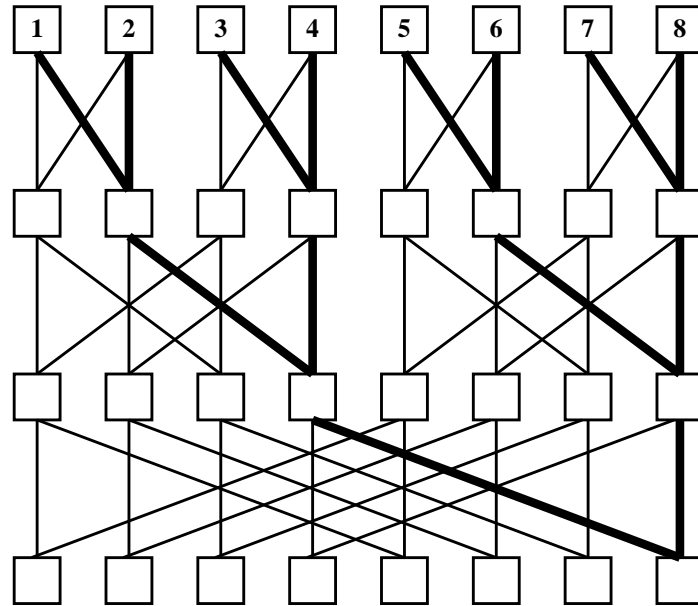


Figure 4: Application of odd–even cyclic reduction to the solution of a linear recurrence on a butterfly network.

The linear array has only short channels, while the complete binary tree has channels that at best are of a length up to  $O(\sqrt{P}/\log_2 P)$ , or even length  $P/4$  with all leaf nodes on the boundary.

What is the speedup for odd–even cyclic reduction for parallel prefix computation on the complete binary tree? What is the efficiency?

In our presentation of odd–even cyclic reduction for parallel prefix computation, all variables were mapped to leaves of the tree. Can the efficiency be improved by mapping variables to all nodes of the tree?

### 2.1.4 Odd–even cyclic reduction on a butterfly network

Complete binary trees are subgraphs of the butterfly network. Thus, our binary tree algorithm can also be used with no slowdown on a butterfly network, as illustrated in Figure 4.

Though the complete binary tree emulation shown in Figure 4 is perfectly valid, parallel prefix computations can be computed in one pass through the butterfly network, while the tree emulation requires two passes, one from top to bottom, the other from bottom to top for backsubstitution.

Devise a one pass butterfly network algorithm for parallel prefix computations!

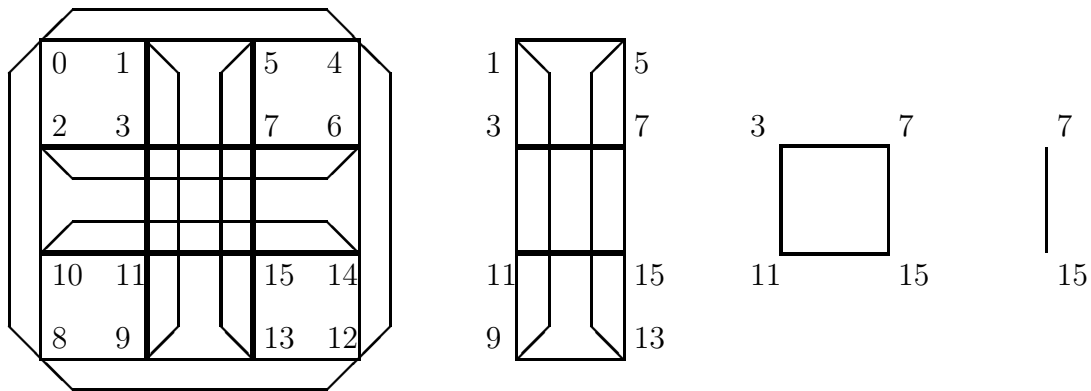


Figure 5: Application of odd–even cyclic reduction to the solution of linear recurrences on a binary cube through folding.

### 2.1.5 Odd–even cyclic reduction on a binary cube

Complete binary trees are *not* subgraphs of binary cubes. However, using a cube of a size equal to the number of equations, and carefully mapping internal tree nodes to a subset of the cube nodes used for the leaf nodes of the tree, allows each step to proceed with just nearest neighbor communications.

The reduction phase for a binary mapping of the data to the cube nodes is illustrated in Figure 5. The algorithm can intuitively be described as a *folding* algorithm [10]. The cube is folded one dimension at a time into a zero dimensional cube for the final reduction stage. The folding takes place in successively higher dimensions. The parent node of a pair of tree nodes is mapped to one of the nodes in the pair. Thus, in this mapping, the cube node to which the root is mapped has  $\log_2 P$  nodes mapped to it. For instance, in Figure 5, cube node 15 has four nodes mapped to it; one leaf node, and three internal nodes of the tree.

With the data mapped to the binary cube in binary–reflected Gray code order, the folding algorithm can still be applied, but the equation being updated is either the local variable, or the received variable. The following simple rule can be derived from the binary–reflected Gray code:

```

nodes with even binary addresses send their variables to the next odd node
if the local index is odd and the binary node address is odd then
    update the local variable
if the local index is even and the binary node address is odd then
    update the received variable and make it the local variable.
    
```

The rule is applied recursively. The algorithm in fact converts the binary–reflected Gray code embedding of active variables to a binary code embedding of those variables.

The binary cube algorithms are load imbalanced in that some nodes participate in several

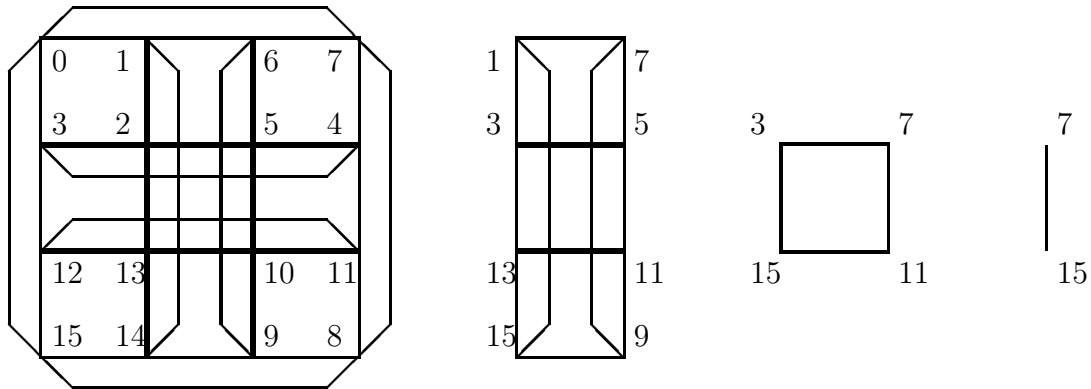


Figure 6: Application of odd–even cyclic reduction to the solution of linear recurrences on a binary cube with indices mapped by Gray code.

reduction steps. Temporary storage is required for as many variables as the number of reduction steps in which a node participates.

The arithmetic and communication complexity is the same as for the binary tree algorithm.

### 2.1.6 Discussion of odd–even cyclic reduction for parallel prefix computations

The characteristics of our odd–even cyclic reduction algorithms for parallel prefix computations on a few networks are summarized in Table 1. Our implementations on  $P$  node linear arrays and binary cube network require that some nodes participate in several steps of each phase of the algorithm, and that several variables be stored in nodes participating in more than one step. The uneven storage need can be avoided at the expense of additional communication.

The linear array is clearly not attractive if communication distance is important, which is the case for packet switching communication systems. For pipelined communication systems, and so called wormhole routing, the distance is of very small significance. Contention is important in both communication models. But, for the solution of linear recurrences as described above, contention does not arise. However, in the case that multiple linear recurrences shall be solved, then contention may become an issue, and so will computational load imbalance. We will consider these issues further below.

Finally we remark that of the networks we investigated, only the linear array have unit length channels. If the wire length determines the time with which the communication system operates, then the time bound may be multiplied by a factor proportional to  $\sqrt{P}/\log_2 P$  to  $\sqrt{P}$  for the complete binary tree,  $\sqrt{P}$  for the binary cube, and  $P/2$  for the butterfly network.

What is the speedup and efficiencies if the clock period is proportional to the wire length and the channel width inversely proportional to the number of channels per node?

Total number of arithmetic operations:	$2P - \log_2 P - 2$			
Arithmetic operations in sequence:	$2 \log_2 P - 1$			
Total number of data elements moved:	$2P - \log_2 P - 2$			
Number of communication actions:	$2 \log_2 P$			
	linear array	compl. binary tree	butterfly network	binary cube
Total comm. bandw. req	$P \log_2 P - \frac{P}{2}$	$4P - \log_2 P - 2$	$4P - \log_2 P - 2$	$2(P - 1)$
Max comm. dist.	$\frac{P}{2}$	1	1	1
Max comp. load/ node	$\log_2 P$	1	1	$\log_2 P$
Max storage/ node	$\log_2 P$	1	1	$\log_2 P$
Number of nodes	$P$	$2P - 1$	$P \log_2 P$	$P$

Table 1: Summary of computational characteristics of odd–even cyclic reduction for linear recurrences on a few networks.

### 2.1.7 Parallel Cyclic Reduction

The parallel cyclic reduction algorithm performs a reduction operation on all equations for which a solution has not yet been determined. The data interaction is illustrated in Figure 7. Parallel cyclic reduction requires no backsubstitution. Note, however, that the odd–even cyclic reduction algorithm requires less work in each step. In each step, odd–even cyclic reduction only performs a reduction on half of the equations in the preceding step.

#### Algorithm Description

The parallel cyclic reduction process is shown below for  $P = 2^p$ :

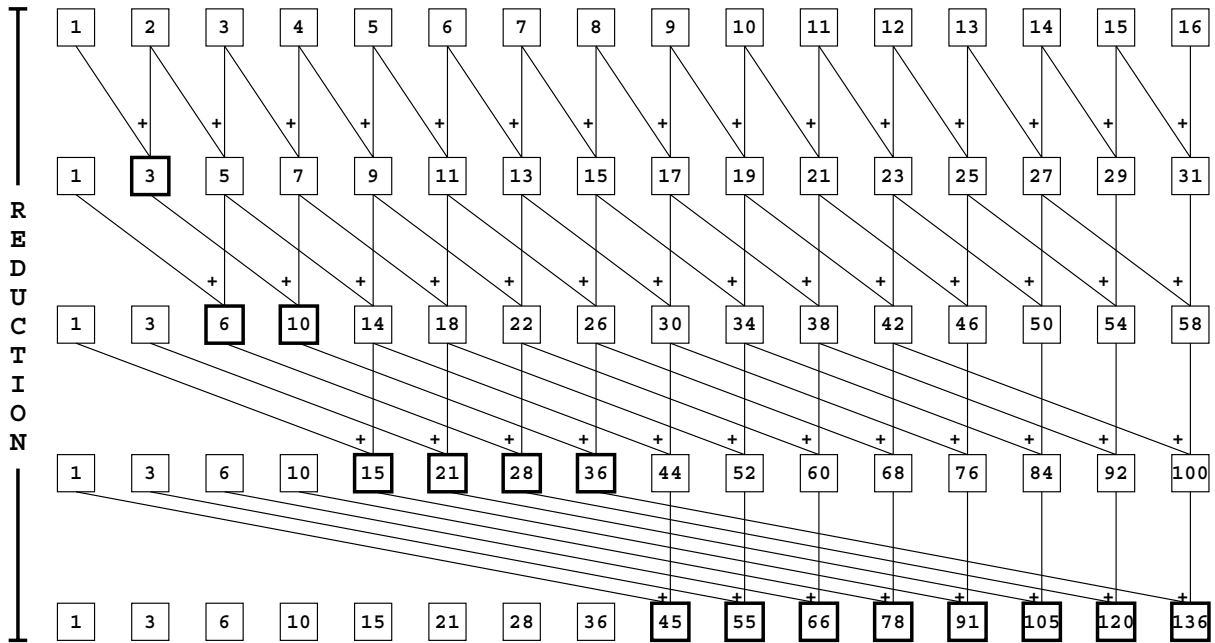


Figure 7: Parallel cyclic reduction for linear recurrences.

$$x(0) = 0$$

$$x(j) = x(j - 1) + y(j) \quad j = \{1, 2, 3, \dots, P\}$$

$$x(j) = x(j - 2) + \underbrace{y(j) + y(j - 1)}_{y^{(1)}(j)} \quad j = \{2, 3, 4, \dots, P\}$$

$$x(j) = x(j - 4) + \underbrace{y^{(1)}(j) + y^{(1)}(j - 2)}_{y^{(2)}(j)} \quad j = \{3, 4, 5, \dots, P\}$$

$$x(j) = x(j - 8) + \underbrace{y^{(1)}(j) + y^{(1)}(j - 4)}_{y^{(2)}(j)} \quad j = \{5, 6, 7, \dots, P\}$$

⋮  
⋮  
⋮

$$x(j) = x(j - P) + \underbrace{y^{(p-1)}(j) + y^{(p-1)}(j - P/2)}_{y^{(p)}(j)} \quad j = \{P/2 + 1, P/2 + 2, \dots, P\}$$

Performing the computations in-place for all  $p$  steps allow the computations to be defined by:

**for**  $k = 1$  **to**  $p$  **do**

```

for  $j = 2^{k-1} + 1$  to  $2^p$  do
     $y(j) \leftarrow y(j) + y(j - 2^{k-1})$ 
endfor
endfor
    
```

For  $P = 8$  the computations proceed as follows:

y(1)	y(2)	y(3)	y(4)	y(5)	y(6)	y(7)	y(8)	
1	2	3	4	5	6	7	8	Initial
1	<b>3</b>	<b>5</b>	<b>7</b>	<b>9</b>	<b>11</b>	<b>13</b>	<b>15</b>	k=1, j=2,3,4,5,6,7,8
1	3	<b>6</b>	<b>10</b>	<b>14</b>	<b>18</b>	<b>22</b>	<b>26</b>	k=2, j=3,4,5,6,7,8
1	3	6	10	<b>15</b>	<b>21</b>	<b>28</b>	<b>36</b>	k=3, j=5,6,7,8

In each step, the current result is added to itself shifted by an appropriate distance.

**Matrix representation of parallel cyclic reduction**

In matrix representation, the first step in the doubling process is represented by the matrix  $R_1^p$ , where

$$R_1^p = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \end{bmatrix}.$$

Applying  $R_1^p$  from the left results in

$$\begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ x_P \end{bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ y_P \end{bmatrix}.$$

Then, the matrix  $B$  after the first reduction step is modified to  $B_1^p = R_1^p B$ , where

$$B_1^p = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -1 & 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -1 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & -1 & 0 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -1 & 0 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & -1 & 0 & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -1 & 0 & 1 \end{bmatrix}$$

The matrix  $R_2^p$  for the second reduction step is defined by

$$R_2^p = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 \end{bmatrix}$$

and  $B_2^p = R_2^p B_1^p$ , where

$$B_2^p = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ -1 & 0 & 0 & 0 & 1 & \cdot & \cdot & \cdot \\ \cdot & -1 & 0 & 0 & 0 & 1 & \cdot & \cdot \\ \cdot & \cdot & -1 & 0 & 0 & 0 & 1 & \cdot \\ \cdot & \cdot & \cdot & -1 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

For  $N = 8$  the last reduction matrix is  $R_3^p$ , which is easily seen to be

$$R_3^p = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

$B_3^p = R_3^p B_2^p$ , or

$$B_3^p = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 & 1 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 & 0 & 1 & \cdot & \cdot \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \cdot \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Thus, it is clear that  $R_3^p R_2^p R_1^p = B^{-1}$ , and no backsubstitution is required. The product  $R_2^p R_1^p$  is easily seen to be

$$R_2^p R_1^p = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & 1 & 1 & \cdot \end{bmatrix}$$

and  $R_3^p R_2^p R_1^p$  is

$$R_3^p R_2^p R_1^p = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdot & \cdot \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdot \end{bmatrix}.$$

The number of steps is  $\log_2 P$ . The number of operations in step  $k$ ,  $k = \{1, 2, \dots, \log_2 P\}$  is  $P - 2^{k-1}$ . The total number of operations is  $P \log_2 P - \sum_{j=0}^{\log_2 P - 1} 2^j = P(\log_2 P - 1) + 1$ .

The obvious sequential algorithm, which proceeds from the first to the last equation, requires  $P - 1$  arithmetic operations. Thus, parallel cyclic reduction requires approximately a factor of  $\log_2 P$  more operations than the sequential algorithm.

Parallel cyclic reduction is also suitable for vector computers. A total of  $\log_2 P$  vector operations are required. The vector length in step  $k$  is  $P - 2^{k-1}$ . Compared to odd–even cyclic reduction parallel cyclic reduction requires almost exactly half as many vector operations, but a factor of  $\frac{1}{2} \log_2 P$  more arithmetic operations. If the time for a vector operation is dominated by the startup time for the vector operation, then it may be beneficial to use parallel cyclic reduction, even though more work is performed. And compared to the sequential algorithm that does not vectorize, the increase in computational efficiency must be at least a factor of  $O(\log_2 P)$  in order to justify the use of parallel cyclic reduction.

In parallel architectures with unbounded parallelism and communication bandwidth, parallel cyclic reduction may yield a higher performance. However, if the arithmetic time is dominating, as is the case for sufficiently large  $P$  relative to the number of nodes, or if the communications bandwidth is a limiting factor, then odd–even cyclic reduction is likely to yield better performance. We will now consider implementations of parallel cyclic reduction on a few network topologies.

### 2.1.8 Parallel cyclic reduction on a linear array

We first consider a linear array of  $P$  nodes for  $P$  variables. We note that each element in step  $k$  moves a distance of  $2^{k-1}$ . The total bandwidth required is:

$$\sum_{k=1}^{\log_2 P} (P - 2^{k-1})2^{k-1} = P(P - 1) - \frac{P^2 - 1}{3} = \frac{2P^2 - 3P + 1}{3}$$

The communications bandwidth required by parallel cyclic reduction is a factor of about  $2P/(3 \log_2 P)$  higher than that of odd–even cyclic reduction. Moreover, from Figure 7 it is clear that severe contention occurs on a linear array. Indeed, the number of edges of the graph representing parallel cyclic reduction mapped to an edge in the linear array for the different stages doubles for each stage, starting at 1 and ending with  $\frac{P}{2}$ .

Half of the nodes perform  $\log_2 P$  operations,  $\frac{P}{4}$  nodes perform  $\log_2 P - 1$  operations, etc.

Thus, we conclude that whereas the arithmetic time for parallel cyclic reduction is  $O(\log_2 P)$ , the communication time due to contention is  $O(P)$ . If the communication time dominates, then the speedup is  $O(1)$  regardless of whether a packet switched or circuit switched communication system is used.

### 2.1.9 Parallel cyclic reduction on a complete binary tree

Mapping the variables to the leaf nodes of a complete binary tree, then applying parallel cyclic reduction leads to long communication paths in each step of the algorithm, and heavy congestion at the root. Indeed, the congestion is the same as for the linear array, but the communication distance is no longer nearest neighbor. Thus, with this mapping, parallel cyclic reduction is not suitable for complete binary trees.

Is there another mapping and associated implementation of parallel cyclic reduction that yields substantially better performance?

### 2.1.10 Parallel cyclic reduction on a butterfly network

The connectivity of the parallel cyclic reduction graph is similar to that of the butterfly network, but not identical. The parallel cyclic reduction graph is a proper subgraph of the so called *PM2I* or *data manipulator* network. Figure 8 shows a data manipulator network. This network has  $p + 1$  stages, each containing  $2^p$  nodes, just as the butterfly network. However, with nodes within a stage  $j$  labeled  $(i, j)$ , where  $0 \leq i < 2^p$  and  $0 \leq j \leq p$  the connectivity is;

$$(i, j) \leftrightarrow (i, j + 1), \quad 0 \leq i < 2^p, \quad 0 \leq j < p$$

$$(i, j) \leftrightarrow (i + 2^j, j + 1), \quad 0 \leq i < 2^p - 2^j, \quad 0 \leq j < p$$

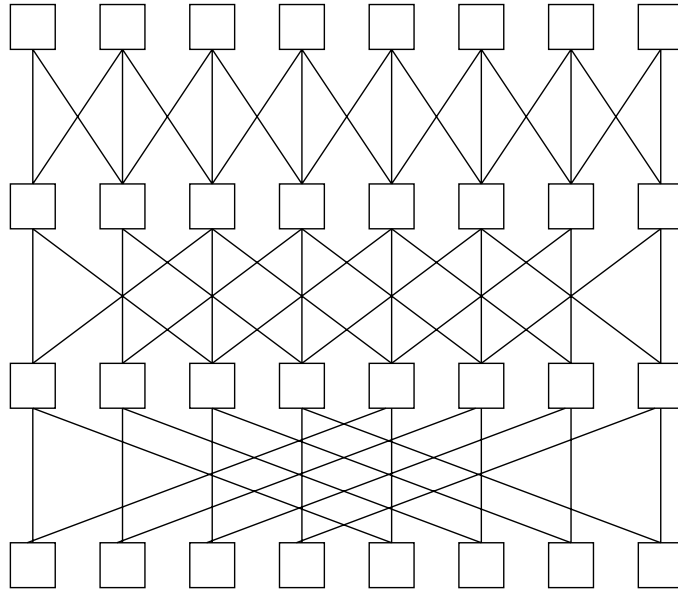


Figure 8: A data manipulator network for  $p = 3$ .

$$(i, j) \leftrightarrow (i - 2^j, j + 1), \quad 2^j \leq i < 2^p, \quad 0 \leq j < p$$

It is immediately clear that parallel cyclic reduction for  $P$  variables can be implemented on a PM2I network with  $P(\log_1 P + 1)$  nodes with nearest neighbor communication and full parallelism.

A butterfly network does not have the ideal connectivity, but has a connectivity that is appropriate for an emulation of the parallel cyclic reduction graph with no slow down, i.e., the graph can be emulated with full concurrency and nearest neighbor communication. The emulation requires the use of one temporary variable in each node. The temporary variable keeps a running sum. The butterfly network emulation is illustrated in Figure 9.

Both the arithmetic time and the communication time is of order  $O(\log_2 P)$ . the total number of arithmetic operations is  $O(P \log_2 P)$  and the total communication bandwidth used is also  $O(P \log_2 P)$ .

The speedup is  $O(\frac{P}{\log_2 P})$  and the efficiency is  $O(\frac{1}{\log_2^2 P})$ . If wire length is limiting the communication rate, then the speedup and efficiency may decrease by a factor of  $O(\sqrt{P})$ .

The butterfly network, as well as the PM2I network, has an inherent inefficiency by a factor of  $\log_2 P$  for parallel cyclic reduction, since only one stage at a time is used. The efficiency can be improved for the computation of multiple recurrences through pipelining.

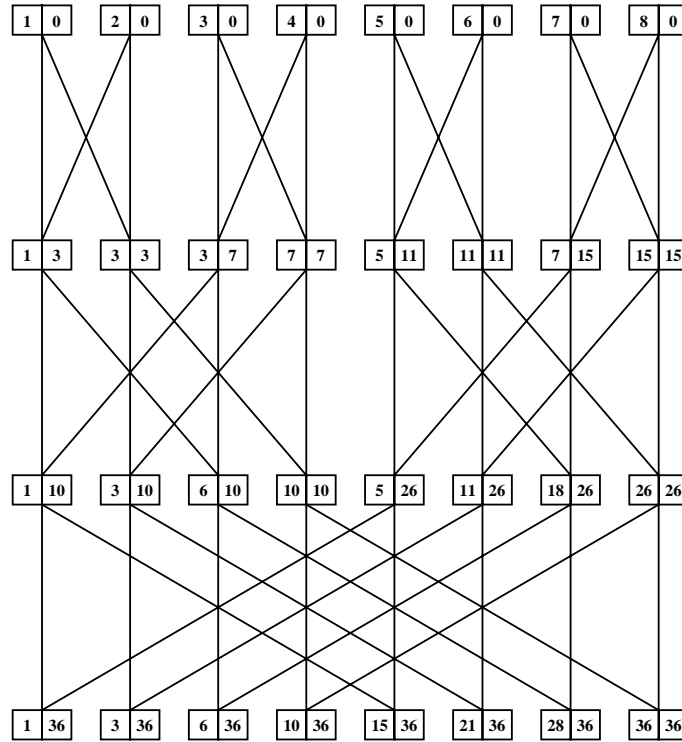


Figure 9: Implementing parallel cyclic reduction on a butterfly network.

### 2.1.11 Parallel cyclic reduction on a binary cube

The binary cube can be viewed as a butterfly network collapsed by identifying all nodes in a column in Figure 9. Hence, we expect that the binary cube may improve upon the efficiency of the butterfly network by a factor of  $\log_2 P$  for the computation of a single recurrence by parallel cyclic reduction.

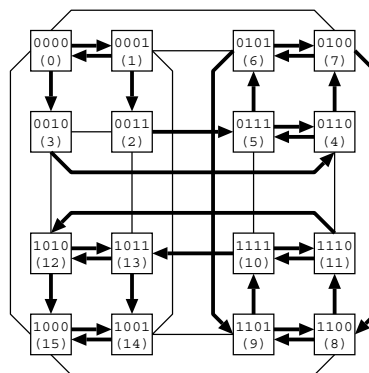
With the data mapped to the cube nodes by a binary–reflected Gray code nodes  $i \pm 1$  are adjacent for any  $i$ . Nodes  $i \pm 2^k$  are at distance two. Thus, the dilation is two for edges of the form  $(i, j) \rightarrow (i + 2^j, j + 1)$  for  $j > 0$ , with a mapping of parallel cyclic reduction nodes  $(i, j)$  to cube node  $i$  for all  $j$ . The mapping of the edges of the parallel cyclic reduction graph to the edges of the cube can be made such that the edge load is one for each emulation stage, i.e., there is no contention for communication channels [10]. This mapping is shown in Figure 10. The routing is

- For the communication between nodes holding index  $i$  and index  $i + 2^k$ ,  $k > 0$  send data first in the lowest dimension that differs between  $i$  and  $i + 2^k$ , then in the other dimension.

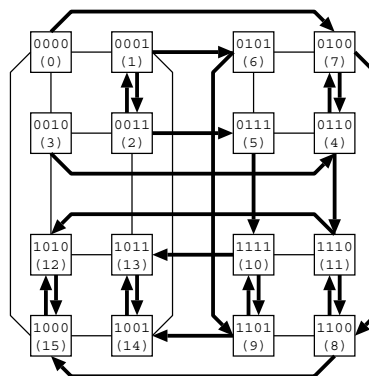
Note that for a given  $k$  the lowest order that differ between  $i$  and  $i + 2^k$  is the same for all  $i$ . Only the higher dimension that differs between  $i$  and  $i + 2^k$  depend upon  $i$ .

Since there is no contention with the suggested routing, each communication is effectively a unit time operation in a circuit switched or wormhole routing system. In a packet switched

Step 2



Step 3



Step 4

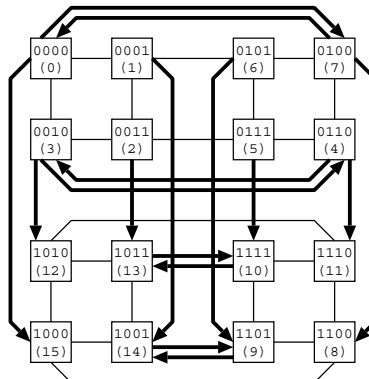


Figure 10: Communication for steps two, three and four for parallel cyclic reduction on a 16 node binary cube. Bidirectional links.

Total number of arithmetic operations:	$P(\log_2 P - 1) + 1$			
Arithmetic operations in sequence:	$\log_2 P$			
Total number of data elements moved:	$P(\log_2 P - 1) + 1$			
Number of communication actions:	$\log_2 P$			
	linear array	compl. bin. tree	butterfly network	binary cube
Total comm. bandw. req	$\frac{2(P^2-1)}{3}$		$P \log_2 P - \frac{P}{2}$	$2P \log_2 P - 3(P - 1)$
Max comm. dist.	$\frac{P}{2}$	$\log_2 P$	1	2
Max edge load	$\frac{P}{2}$	$\frac{P}{2}$	1	1
Max comp. load/ node	$\log_2 P$		1	$\log_2 P$
Max storage/ node	1		2	1
Number of nodes	$P$	$2P - 1$	$P \log_2 P$	$P$

Table 2: Summary of computational characteristics of parallel cyclic reduction for linear recurrences on a few networks.

routing system, each communication requires two units of time, except the first communication, due to the distance of two between  $i$  and  $i + 2^k$ ,  $k > 0$ .

If each connection between a pair of nodes only is used in one direction, then the edge-load is two. But, since the two data items communicated across an edge is send during different communication cycles, there is still no contention in a packet switched system. Figure 11 shows emulation steps two through four of parallel cyclic reduction on a binary cube for  $P = 16$ , assuming unidirectional edges.

Can the parallel cyclic reduction graph be emulated using only nearest neighbor communication?

For the binary-reflected Gray code embedding of the variables to nodes, the arithmetic time is of order  $O(\log_2 P)$  and so is the communication time. The speedup is of order  $O(\frac{P}{\log_2 P})$  and the efficiency is of order  $O(\frac{1}{\log_2 P})$ , an improvement over the butterfly network by a factor of  $\log_2 P$ , as expected.

For a fixed area, which of the two networks, butterfly and binary cube, is the fastest for parallel cyclic reduction evaluation of parallel prefix? Which is the fastest if the channel width is determining the running time and the width per node is fixed? What if the width per partition of  $M$  nodes is fixed?

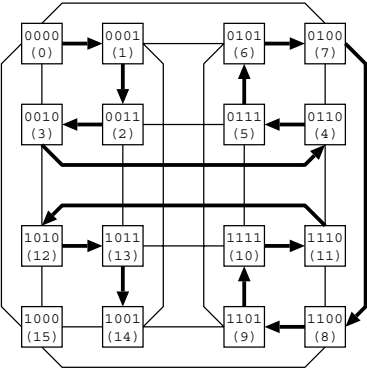
What is the communication time for parallel cyclic reduction on a binary cube with data mapped in binary code instead of binary-reflected Gray code?

### 2.1.12 Discussion

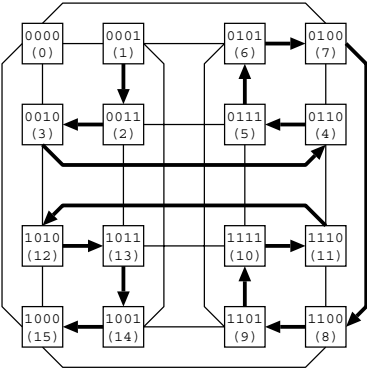
Table 2 summarizes the computational characteristics of parallel cyclic reduction for parallel prefix computations.

For parallel cyclic reduction the butterfly network and the binary cube are clearly preferable with respect to communication. The binary cube achieves the same communication complexity

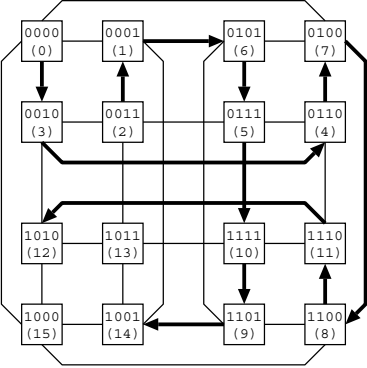
Step 2, First Communication



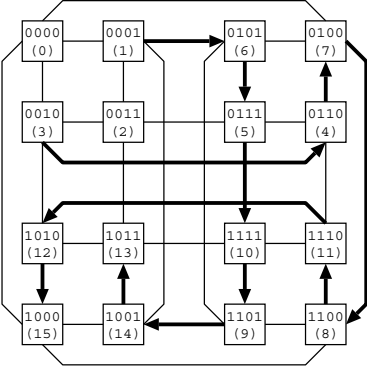
Step 2, Second Communication



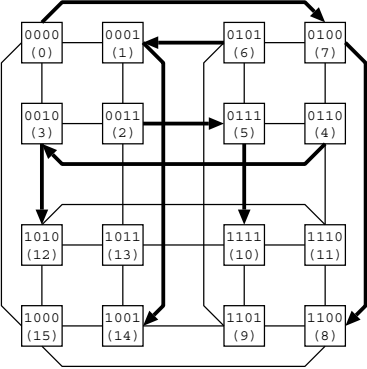
Step 3, First Communication



Step 3, Second Communication



Step 4, First Communication



Step 4, Second Communication

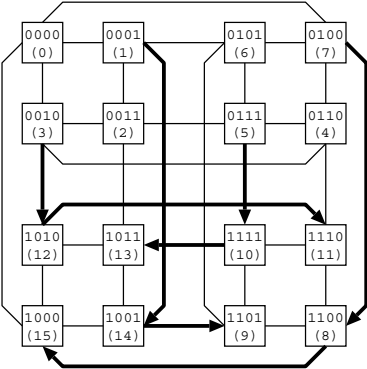


Figure 11: Communication for steps two, three and four for parallel cyclic reduction on a 16 node binary cube. Unidirectional links.

Measurement	OECR	PCR
Parallel Steps	$2 \log N - 1$	$\log N$
Arithmetic Operations	$2N - \log N - 2$	$N(\log N - 1) + 1$
Communication Operations	$2N - \log N - 2$	$N(\log N - 1) + 1$
Bandwidth Required on Linear Array	$N \log N - \frac{N}{2}$	$\frac{2N^2 - 3N + 1}{3}$

Table 3: Comparison of odd–even and parallel cyclic reduction on linear arrays

Algorithm	Vector Operations	Vector Lengths
OECR (Reduction)	$\log N$	$\frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \dots, 1$
OECR (Backsubstitution)	$\log N - 1$	$1, 3, 7, \dots, \frac{N}{2} - 1$
PCR	$\log N$	$N - 1, N - 2, N - 4, \dots, N - \frac{N}{2}$

Table 4: Comparison of odd–even and parallel cyclic reduction on vector architectures.

as the butterfly network with a factor of  $\log_2 P$  less nodes.

### 2.1.13 Comparing odd–even and parallel cyclic reduction

Both algorithms are inconsistent. A sequential algorithm requires  $P - 1$  operations, odd–even cyclic reduction requires  $2P - \log_2 P - 1$  operations, and parallel cyclic reduction requires  $P(\log_2 P - 1) + 1$  operations. The characteristics are summarized in Table 3.

Comparing a linear array implementation of parallel cyclic reduction and odd–even cyclic reduction with one data element per node, then if the arithmetic time dominates, parallel cyclic reduction yields better performance, while odd–even cyclic reduction yields better performance if the communication time dominates. Parallel cyclic reduction is more communication intensive, and performs relatively better on networks with a large bisection width.

Table 4 summarizes the computational characteristics with respect to vector architectures. When vector startup time dominates, parallel cyclic reduction may yield better performance than odd–even cyclic reduction. However, for sufficiently large  $P$ , arithmetic time will dominate, making odd–even cyclic reduction likely to perform better.

Derive implementations of odd–even cyclic reduction and parallel cyclic reduction on meshes in two and three dimensions!

### 2.1.14 Oversized linear recurrences

It is rarely the case that there is a sufficient number of nodes to allow for one variable per node. With a consecutive (block) allocation scheme  $\frac{P}{N}$  successive elements are allocated to each of  $N$  nodes. Emulating an odd–even or parallel cyclic reduction algorithm for  $P$  fold parallelism on an  $N < P$  node array is inefficient both due to the fact that both algorithms are inconsistent, and that unnecessary communication may be required.

Substructuring, domain decomposition, or condensation are terms used for techniques for reduc-

$$\begin{array}{|cccc|}
 \hline
 1 & & & \\
 -1 & 1 & & \\
 & -1 & 1 & \\
 & & -1 & 1 \\
 \hline
 & & -1 & 1 \\
 & & & -1 & 1 \\
 & & & & -1 & 1 \\
 & & & & & -1 & 1 \\
 \hline
 & & & & -1 & 1 \\
 & & & & & -1 & 1 \\
 & & & & & & -1 & 1 \\
 & & & & & & & -1 & 1 \\
 \hline
 & & & & & & -1 & 1 \\
 & & & & & & & -1 & 1 \\
 & & & & & & & & -1 & 1 \\
 & & & & & & & & & -1 & 1 \\
 \hline
 \end{array}
 \begin{array}{|c|}
 \hline
 x_0 \\
 x_1 \\
 x_2 \\
 x_3 \\
 \hline
 x_4 \\
 x_5 \\
 x_6 \\
 x_7 \\
 \hline
 x_8 \\
 x_9 \\
 x_{10} \\
 x_{11} \\
 \hline
 x_{12} \\
 x_{13} \\
 x_{14} \\
 x_{15} \\
 \hline
 \end{array}
 =
 \begin{array}{|c|}
 \hline
 y_0 \\
 y_1 \\
 y_2 \\
 y_3 \\
 \hline
 y_4 \\
 y_5 \\
 y_6 \\
 y_7 \\
 \hline
 y_8 \\
 y_9 \\
 y_{10} \\
 y_{11} \\
 \hline
 y_{12} \\
 y_{13} \\
 y_{14} \\
 y_{15} \\
 \hline
 \end{array}$$

Figure 12: Consecutive data allocation for a substructured linear recurrence solver.

ing the communication requirements for the solution of systems of equations on multiprocessors. The techniques may also be advantageous on single processors. Substructuring and domain decomposition may have a profound impact on the numerical properties of the algorithm.

A substructured linear recurrence solver is illustrated in Figures 12 – 14. The idea is that the substitution process is carried out locally as far as possible, then a distributed system with one variable per processing node is solved, followed by local backsubstitution. In matrix form, the local substitution generates *fill-in* in all except the first partition. The result of the local substitution on the system of equations is illustrated in Figure 13. The actions on the set of variables is shown in Figure 14. By considering a system of equations consisting of the last system in each partition, a reduced system containing  $N$  equations and  $N$  unknowns must be solved. This reduced set forms a linear first order recurrence, and odd–even cyclic reduction or parallel cyclic can be applied to this system, as described previously.

The local reduction and backsubstitution requires  $2(\frac{P}{N} - 1)$  arithmetic operations. The distributed linear recurrence solution requires  $2 \log_2 N - 1$  operations in sequence, as before. Thus, for  $P \gg N$ , the speedup with respect to arithmetic is  $\sim \frac{P-1}{2(\frac{P}{N}-1)+2 \log_2 N-1} \approx \frac{N}{2}$ . The efficiency approaches  $\frac{1}{2}$  for  $P \gg N$ . The source of the inefficiency with respect to arithmetic is that approximately twice as many operations are performed as in the sequential algorithm.



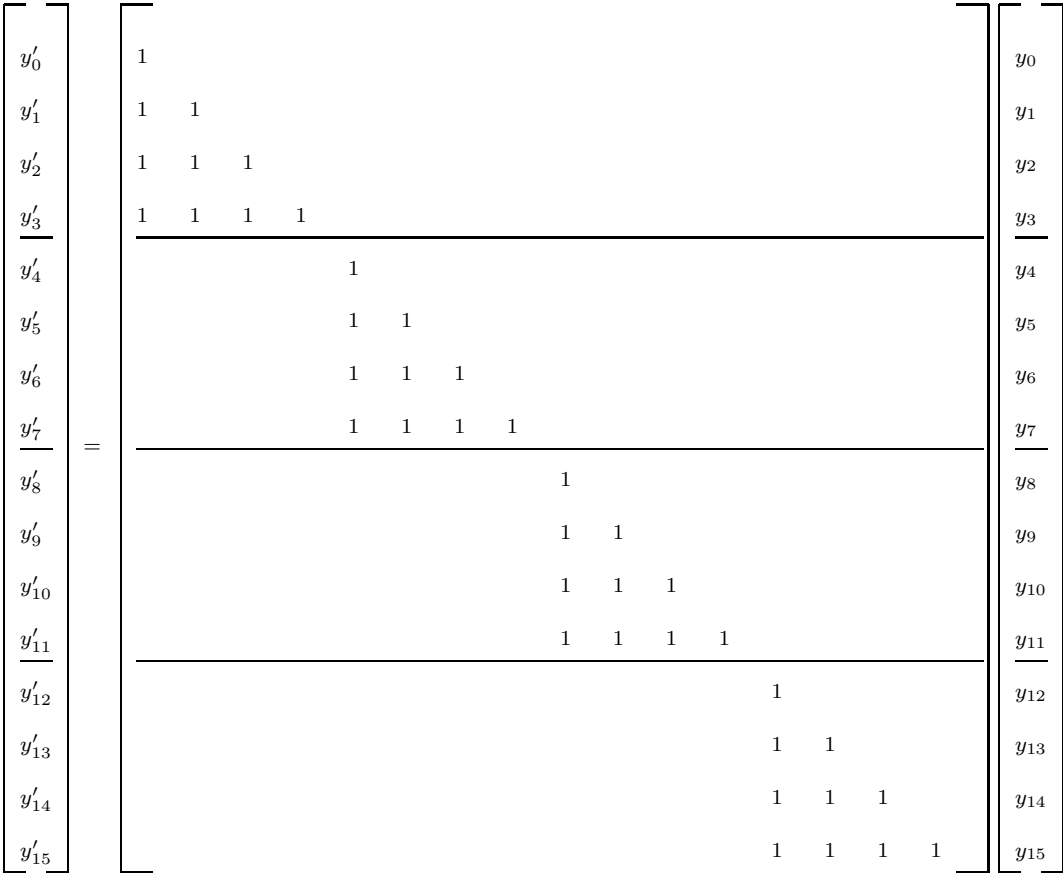


Figure 14: A substructured linear recurrence solver.

For  $P \gg N$ , substructuring results in a communication time that always is of a lower order than the arithmetic time, and the speedup and efficiencies approaches that determined by the arithmetic time alone.

**2.1.15 Multiple linear recurrences**

Separate prefix operations may be required on each of the rows of a two-dimensional array, as for instance in using the Alternating Direction Method in two dimensions. Let,  $y$  be an array of two or more dimensions. The result  $x$  is an array of the same shape. In two dimensions the computations are

$$x(i, j) = x(i, j - 1) + y(i, j), \quad 1 \leq j \leq Q, \quad 1 \leq i \leq P$$

for arrays of shape  $P \times Q$  and a linear recurrence along rows (the  $Q$ -axis).

If the  $Q$ -axis is local to a node, then the computations are *embarrassingly parallel*. No communication is required, and the sequential algorithm shall be used in each node. As long as  $P > N$ , load-balance can be achieved.

0	1	2	3	4	5	6	7
$y(0, 0)$	$y(0, 1)$	$y(0, 2)$	$y(0, 3)$	$y(0, 4)$	$y(0, 5)$	$y(0, 6)$	$y(0, 7)$
$y(1, 0)$	$y(1, 1)$	$y(1, 2)$	$y(1, 3)$	$y(1, 4)$	$y(1, 5)$	$y(1, 6)$	$y(1, 7)$
$y(2, 0)$	$y(2, 1)$	$y(2, 2)$	$y(2, 3)$	$y(2, 4)$	$y(2, 5)$	$y(2, 6)$	$y(2, 7)$
$y(3, 0)$	$y(3, 1)$	$y(3, 2)$	$y(3, 3)$	$y(3, 4)$	$y(3, 5)$	$y(3, 6)$	$y(3, 7)$
$y(4, 0)$	$y(4, 1)$	$y(4, 2)$	$y(4, 3)$	$y(4, 4)$	$y(4, 5)$	$y(4, 6)$	$y(4, 7)$
$y(5, 0)$	$y(5, 1)$	$y(5, 2)$	$y(5, 3)$	$y(5, 4)$	$y(5, 5)$	$y(5, 6)$	$y(5, 7)$
$y(6, 0)$	$y(6, 1)$	$y(6, 2)$	$y(6, 3)$	$y(6, 4)$	$y(6, 5)$	$y(6, 6)$	$y(6, 7)$
$y(7, 0)$	$y(7, 1)$	$y(7, 2)$	$y(7, 3)$	$y(7, 4)$	$y(7, 5)$	$y(7, 6)$	$y(7, 7)$

Figure 15: A data allocation for multiple instance linear recurrence computations.

If the  $Q$ -axis is distributed across  $N$  nodes, then one technique is to transpose the array such that the  $Q$ -axis becomes entirely local, use the sequential algorithm in each node, followed by a second transpose to restore the initial data allocation. As long as  $P > N$  load-balance is achieved. The arithmetic efficiency is the best possible, and the communication expense is two transpositions. Figure 15 shows the data layout for an  $8 \times 8$  data array assigned to 8 nodes with the  $P$ -axis being local to a node.

Another technique to perform the  $P$  parallel prefix operations would be to use the algorithms discussed previously. For the butterfly network computations using the parallel cyclic reduction algorithm can be pipelined. For the other networks we considered, namely the linear array, the binary tree and the binary cube, the parallel cyclic reduction algorithm is not a good choice, since at least half of the nodes are used in each step of the algorithm. With parallel cyclic reduction a new computation can be initiated only once for every  $\log_2 N$  steps. Though in our implementation of odd-even cyclic reduction one node was used in each of the  $\log_2 N$  steps for the distributed part in the solution of an oversized linear recurrence, we will now show how *balanced cyclic reduction* for multiple recurrences can be used to yield good load-balance.

The reduction steps of balanced cyclic reduction are shown in Figure 16, where the reduction to odd or even equations is made for the first and second half of the number of instances, respectively.

From Figure 16 it is clear that the communication and computational requirements are perfectly balanced whenever  $Q \bmod P = 0$ . The number of arithmetic operations performed in sequence for the reduction phase is  $\frac{Q}{P}(\frac{P}{2} + \frac{P}{4} + \dots + 1) = \frac{Q}{P}(P - 1) \approx Q$ . For the backsubstitution phase  $\frac{Q}{P}(P - 2)$  arithmetic operations are required. Thus, with respect to arithmetic operations the speedup is  $\sim \frac{Q(P-1)}{2Q} \approx \frac{P}{2}$ . The efficiency is  $\sim \frac{1}{2}$ .

Balanced reduction (and backsubstitution) reduces the communication requirements in an analogous way. The communication distance remains the same for each communication step, but compared to the unbalanced algorithm resulting from a direct application of the algorithm for a single system, the number of variables that must be communicated is reduced by a factor of two in each communication step. In all, the balanced algorithm yields a  $\log_2 N$  factor reduction in communication time compared to the unbalanced algorithm.

Red. step	0	1	2	3	4	5	6	7
1	—	$y^1(0, 1)$	—	$y^1(0, 3)$	—	$y^1(0, 5)$	—	$y^1(0, 7)$
	—	$y^1(1, 1)$	—	$y^1(1, 3)$	—	$y^1(1, 5)$	—	$y^1(1, 7)$
	—	$y^1(2, 1)$	—	$y^1(2, 3)$	—	$y^1(2, 5)$	—	$y^1(2, 7)$
	—	$y^1(3, 1)$	—	$y^1(3, 3)$	—	$y^1(3, 5)$	—	$y^1(3, 7)$
	$y^1(4, 0)$	—	$y^1(4, 2)$	—	$y^1(4, 4)$	—	$y^1(4, 6)$	—
	$y^1(5, 0)$	—	$y^1(5, 2)$	—	$y^1(5, 4)$	—	$y^1(5, 6)$	—
	$y^1(6, 0)$	—	$y^1(6, 2)$	—	$y^1(6, 4)$	—	$y^1(6, 6)$	—
	$y^1(7, 0)$	—	$y^1(7, 2)$	—	$y^1(7, 4)$	—	$y^1(7, 6)$	—
2	—	—	—	$y^2(0, 3)$	—	—	—	$y^2(0, 7)$
	—	—	—	$y^2(1, 3)$	—	—	—	$y^2(1, 7)$
	—	$y^2(2, 1)$	—	—	—	$y^2(2, 5)$	—	—
	—	$y^2(3, 1)$	—	—	—	$y^2(3, 5)$	—	—
	—	—	$y^2(4, 2)$	—	—	—	$y^2(4, 6)$	—
	—	—	$y^2(5, 2)$	—	—	—	$y^2(5, 6)$	—
	$y^2(6, 0)$	—	—	—	$y^2(6, 4)$	—	—	—
	$y^2(7, 0)$	—	—	—	$y^2(7, 4)$	—	—	—
3	—	—	—	—	—	—	—	$y^3(0, 7)$
	—	—	—	$y^3(1, 3)$	—	—	—	—
	—	—	—	—	—	$y^3(2, 5)$	—	—
	—	$y^3(3, 1)$	—	—	—	—	—	—
	—	—	$y^3(5, 2)$	—	—	—	$y^3(4, 6)$	—
	—	—	—	—	—	—	—	—
	—	—	—	—	$y^3(6, 4)$	—	—	—
	$y^3(7, 0)$	—	—	—	—	—	—	—

Figure 16: Balanced odd–even reduction for multiple instance linear recurrence computations.

What is the speedup and efficiency of balanced cyclic reduction when both arithmetic and communication time is accounted for on a linear array? one a binary cube?

How does the balanced reduction algorithm perform on a complete binary tree? A butterfly network?

## 2.2 General first order recurrence

We now return to the general first order recurrence

$$x(j) = a(j)x(j - 1) + y(j), \quad y(0) = 0, \quad j = \{1, 2, \dots, P\}$$

The general linear recurrence can be represented in matrix form as

$$\begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ -a(2) & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & -a(3) & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & -a(4) & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & -a(5) & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & -a(6) & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -a(7) & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -a(8) & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ x_P \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ y_P \end{bmatrix}$$

carrying out the substitution in the recurrence a few times we get

$$\begin{aligned} x(1) &= y(1) \\ x(2) &= a(2)x(1) + y(2) = \\ x(3) &= a(3)x(2) + y(3) = a(3)a(2)y(1) + a(3)y(2) + y(3) \\ x(4) &= a(4)x(3) + y(4) = a(4)a(3)a(2)y(1) + a(4)a(3)y(2) + a(4)y(3) + y(4) \\ &\vdots \qquad \qquad \qquad \vdots \end{aligned}$$

In general, we have

$$x(i) = \sum_{j=1}^i \left( \prod_{k=j+1}^i a(k) \right) y(j)$$

Note that if  $a(i) = z$  for all  $i$ , then  $x(i) = \sum_{j=0}^{i-1} y(i - j)z^j$ , i.e.,  $x(i)$  is an  $i$ th order polynomial. Indeed, the substitution process in unexpanded form yields

$$\begin{aligned} x(1) &= y(1) \\ x(2) &= a(2)x(1) + y(2) = \\ x(3) &= a(3)x(2) + y(3) = a(3)(a(2)y(1) + y(2)) + y(3) \\ x(4) &= a(4)x(3) + y(4) = a(4)(a(3)(a(2)y(1) + y(2)) + y(3)) + y(4) \\ x(5) &= a(5)x(4) + y(5) = a(5)(a(4)(a(3)(a(2)y(1) + y(2)) + y(3)) + y(4)) + y(5) \\ &\vdots \qquad \qquad \qquad \vdots \end{aligned}$$

which we recognize as Horner's rule. The sequential evaluation of  $x(i)$  by Horner's rule requires  $2(i - 1)$  arithmetic operations, which is the best possible.

### 2.2.1 Evaluating general linear recurrences through parallel prefix computations

Returning now to the expression  $x(i) = \sum_{j=1}^i \left( \prod_{k=j+1}^i a(k) \right) y(j)$ , we recognize that if  $a(k) \equiv 1$  for all  $k$ , then we have a  $+$  parallel prefix. Similarly, if  $y(1) = 1$  and  $y(j) = 0$  for all  $j$ , then we have a  $\times$  parallel prefix.

Let  $T(i, j) = \prod_{k=j}^i a(k)$ . Then,  $x(i) = \sum_{j=1}^i T(i, j+1)y(j)$ . But,  $T(i, j+1) = \frac{T(i,1)}{T(j,1)}$ . Thus,

$$x(i) = \sum_{j=1}^i T(i, 1) \frac{y(j)}{T(j, 1)} = T(i, 1) \sum_{j=1}^i g(j),$$

where  $g(j) = \frac{y(j)}{T(j,1)}$ . This expression suggests the following evaluation procedure for  $x(i)$  for all  $i$ .

- Evaluate  $T(i, 1)$  for all  $i$  by a  $\times$  parallel prefix operation.
- Evaluate  $g(j)$  for all  $j$  by an elementwise divide.
- Evaluate  $h(i) = \sum_{j=1}^i g(j)$  for all  $i$  through a  $+$  parallel prefix.
- Evaluate  $x(i)$  by a pointwise multiplication  $T(i, 1)h(i)$ .

This procedure requires a total of  $2(P - 1) + 2P$  arithmetic operations for the evaluation of  $P - 1$  terms ( $2, 3, \dots, P$ ) of the recurrence on a sequential computer. Using odd-even cyclic reduction for a parallel evaluation of the prefix operations the total number of arithmetic operations become  $2(2(P - 1) - \log_2 P) + 2P = 6P - \log_2 P - 4$ .

Unfortunately, the  $\times$  parallel prefix operation may result in overflow or underflow. Numerically the scheme is not robust [3]. A remedy could be to take the logarithm of the products, and use a  $+$  parallel prefix on the logarithms and then exponentiate the result. However, this procedure is both time consuming and not very accurate.

An alternate formulation with the goal of using parallel prefix evaluation for the recurrence is the following

$$\begin{bmatrix} x(i) \\ 1 \end{bmatrix} = \begin{bmatrix} a(i) & y(i) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x(i-1) \\ 1 \end{bmatrix}$$

Let  $M(i) = \begin{bmatrix} a(i) & y(i) \\ 0 & 1 \end{bmatrix}$ . Then,

$$\begin{bmatrix} x(i) \\ 1 \end{bmatrix} = M(i)M(i-1)\cdots M(2) \begin{bmatrix} y(1) \\ 1 \end{bmatrix},$$

and all  $x(i)$  can be evaluated using a  $\times$  parallel prefix computation on  $2 \times 2$  matrices. Note that due to the particular form of the matrices, each product only requires three arithmetic operations and returns a matrix with the same structure, i.e,

$$M(i)M(i-1) = \begin{bmatrix} a(i) & y(i) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a(i-1) & y(i-1) \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} a & \\ 0 & 1 \end{bmatrix}.$$

Furthermore, the multiplication of one of these matrices with a vector requires two arithmetic operations. Thus, in the matrix formulation we have not introduced any new operations. Sequential evaluation can still be performed in  $2(P-1)$  arithmetic operations. Evaluating the  $\times$  parallel prefix operation with odd–even cyclic reduction results in  $3(2(P-1) - \log_2 P)$  arithmetic operations, followed by  $2(P-1)$  arithmetic operations for the evaluation of all  $x(i)$ ,  $i = 2, 3, \dots, P$ . Thus, with this parallel evaluation scheme the total number of arithmetic operations is  $8(P-1) - 3\log_2 P$ , which exceeds the arithmetic count of the previous algorithm for the evaluation by approximately  $2P$  operations.

### 2.2.2 Odd–even cyclic reduction for general linear recurrences

We will now present an odd–even cyclic reduction version of Kogge and Stone’s [15] algorithm for general linear recurrences. The algorithm requires a total of  $5P - \log_2 P - 6$  arithmetic operations. In the description of the algorithm we use the definition

$$Q(i, j) = \sum_{k=j}^i T(i, k+1)y(k)$$

From this definition it follows that

$$\begin{aligned} Q(i, j) &= \sum_{k=j}^m T(i, k+1)y(k) + \sum_{k=m+1}^i T(i, k+1)y(k) \\ &= T(i, m+1) \sum_{k=j}^m T(m, k+1)y(k) + \sum_{k=m+1}^i T(i, k+1)y(k) \\ &= T(i, m+1)Q(m, j) + Q(i, m+1) \end{aligned}$$

Note further that  $Q(i, i) = y(i)$ .

The reduction phase of the odd–even cyclic reduction algorithm is as follows:

$$\begin{aligned}
 x(0) &= 0 \\
 &\vdots \\
 &\vdots \\
 x(j) &= a(j)x(j-1) + y(j) \\
 &= T(j, j)x(j-1) + y(j) && j = \{1, 2, 3, \dots, P\} \\
 \\
 x(j) &= T(j, j)T(j-1, j-1)x(j-2) + T(j, j)y(j-1) + y(j) \\
 &= T(j, j-1)x(j-2) + Q(j, j-1) && j = \{2, 4, 6, \dots, P\} \\
 \\
 x(j) &= T(j, j-1)T(j-2, j-3)x(j-4) + T(j, j-1)Q(j-2, j-3) + Q(j, j-1) \\
 &= T(j, j-3)x(j-4) + Q(j, j-3) && j = \{4, 8, 12, \dots, P\} \\
 \\
 &\vdots \\
 &\vdots \\
 &\vdots
 \end{aligned}$$

In particular, for  $j = 2^p$  we have

$$x(2^p) = T(2^p, 1)x(0) + Q(2^p, 1) = Q(2^p, 1)$$

Thus, for the reduction phase of odd–even cyclic reduction for a general linear recurrence we need to perform the computations

```

for  $k = 1$  to  $p$  do
  for  $j = 2^k$  step  $2^k$  to  $2^p$  do
     $T(j, j - 2^k + 1) \leftarrow T(j, j - 2^{k-1} + 1)T(j - 2^{k-1}, j - 2^k)$ 
     $Q(j, j - 2^k + 1) \leftarrow T(j, j - 2^{k-1} + 1) \times Q(j - 2^{k-1}, j - 2^k) + Q(j, j - 2^{k-1} + 1)$ 
  endfor
endfor

```

For the backsubstitution phase we perform the computations

```

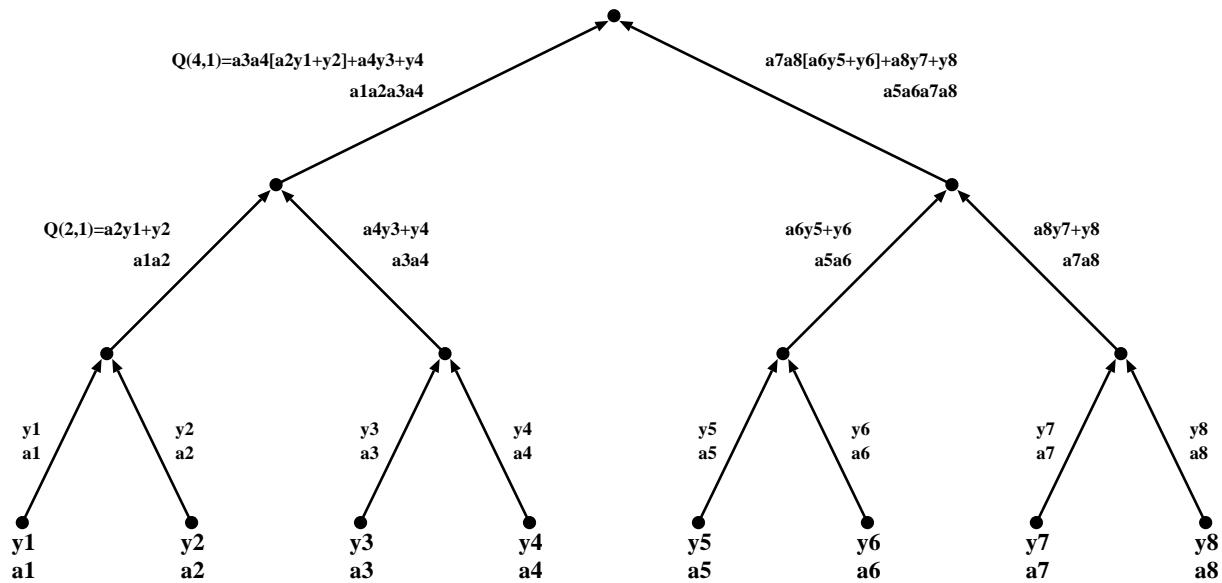
for  $k = p - 1$  step  $-1$  to  $1$  do
  for  $j = 2 \cdot 2^k$  step  $2^k$  to  $2^p$  do
     $Q(j - 2^{k-1}, 1) \leftarrow T(j - 2^{k-1}, j - 2^k + 1) \times Q(j - 2^k, 1) + Q(j - 2^{k-1}, j - 2^k + 1)$ 
  endfor
endfor

```

Each reduction step requires three arithmetic operations in a node and each backsubstitution step requires two arithmetic operations in a node. Figure 17 shows the reduction and backsubstitution phases of odd–even cyclic reduction for the solution of a general linear recurrence.

Write a binary tree algorithm for odd–even cyclic reduction!

Reduction Phase



Back Substitution Phase

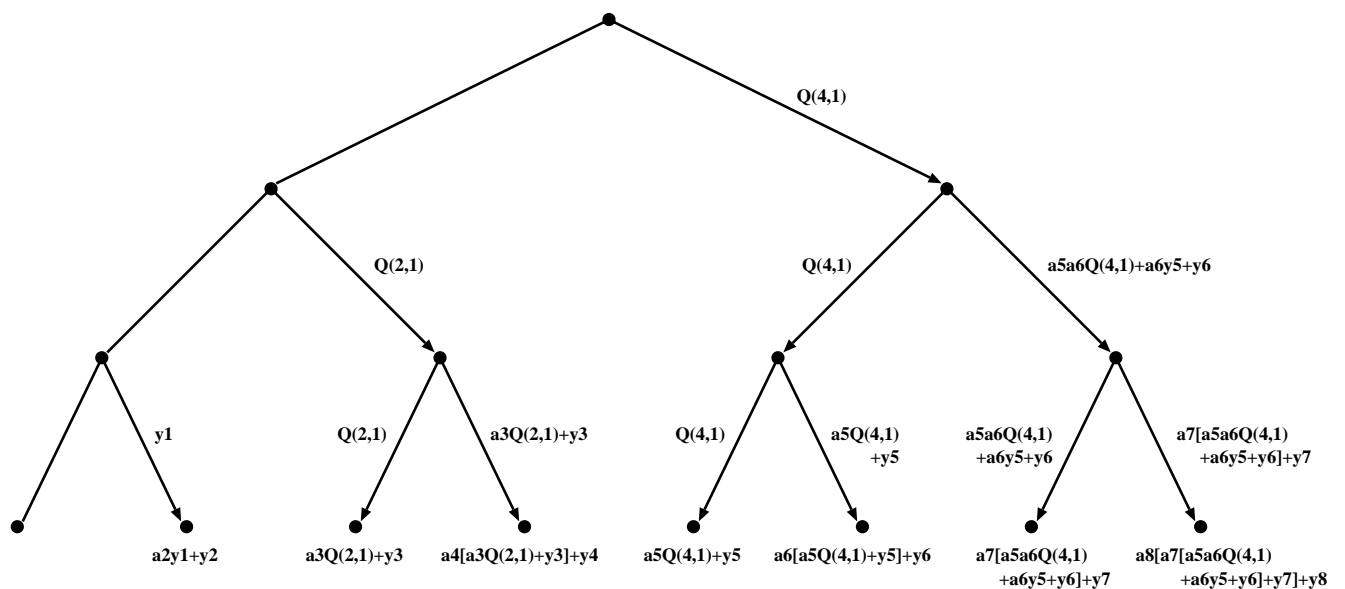


Figure 17: The reduction and back substitution phases of odd–even cyclic reduction for a general linear recurrence

### 2.2.3 Parallel cyclic reduction for general linear recurrences

The solution of a general linear recurrence can also be expressed using parallel cyclic reduction. Using the arrays  $a$  and  $y$  for all  $\log_2 P$  steps, the computations are defined by [15]

```

for  $k = 1$  to  $p$  do
  for  $j = 2^k$  to  $2^p$  do
     $y(j) \leftarrow y(j) + a(j)y(j - 2^{k-1})$ 
     $a(j) \leftarrow a(j) \times a(j - 2^{k-1})$ 
  endfor
endfor

```

Compared to the parallel prefix computation, the communication is doubled, since both  $T$  and  $Q$  must be communicated.

### 2.2.4 Algorithms with few multiplications

In the conventional binary number system multiplication either requires more time or more hardware resources than addition and subtraction. Division is even more resource demanding than multiplication.

Kung [16] has devised a collection of algorithms for parallel architectures that use addition on the longest path and at most a few steps of embarrassingly parallel multiplication and/or division. Thus, for instance, instead of a  $\times$  parallel prefix requiring a time of  $\log_2 P$  multiplications for the computation of  $x^2, x^3, \dots, x^P$ , Kung's algorithm computes this prefix operation in  $\log_2 P$  additions plus a constant number of parallel addition/multiplication or division.

We illustrate the idea in computing  $x^P$ . First, Borodin and Munro [1] has shown that if division is not used,  $\log_2 P$  multiplications in sequence is indeed the minimum number of operations required for the evaluation, regardless of the number of processors used. Hence, it is necessary to find an algorithm using division to evaluate  $x^P$ . Kung's algorithm is based on the following observation

$$C = \sum_{i=1}^P \frac{s_i}{x - r_i} = \frac{G(x)}{\prod_{i=1}^P (x - r_i)}$$

To make effective use of this relationship in computing  $x^P$ ,  $r_i$  for  $i = 1, 2, \dots, P$  are chosen as the roots of the equation  $x^P - r = 0$ , and  $s_i$  are chosen such that  $G(x) \equiv 1$ . With this choice,

$$C = \frac{1}{x^P - r}$$

and

$$x^P = \frac{1}{C} + r$$

Assuming that  $s_i$  and  $r_i$  are already known, the computations consists in

1. Compute  $x - r_i$  for  $i = \{1, 2, \dots, P\}$ .
2. Compute  $\frac{s_i}{x - r_i}$  for  $i = \{1, 2, \dots, P\}$ .
3. Compute  $C$ .
4. Compute  $x^P = \frac{1}{C} + r$ .

The total time is  $t_A \cdot \log_P + 2(t_A + t_D)$  on  $P$  processors, where  $t_A$  is the time for addition/subtraction and  $t_D$  is the time for division.

For the algorithm to work we need to show that  $s_i, i = \{1, 2, \dots, P\}$ , can be determined such that  $G(x) \equiv 1$ , and choose  $r$ . We have

$$G(x) = \sum_{i=1}^P s_i \prod_{j \neq i} (x - r_j)$$

Thus,  $G(r_k) = s_k \prod_{j \neq k} (r_k - r_j)$ . We can find a simpler form for the product by considering the derivative of  $x^P - r = \prod_{j=1}^P (x - r_j)$  at  $r_k$ .

$$Px^{P-1} = \sum_{i=1}^P \prod_{j \neq i} (x - r_j)$$

and for  $x = r_k$

$$Pr_k^{P-1} = \prod_{j \neq k} (r_k - r_j)$$

Thus,  $G(r_k) = Ps_k r_k^{P-1}$ . Choosing  $s_k = \frac{r_k}{Pr_k}$  yields  $G(r_k) = \frac{r_k^P}{r} = 1$ , since  $r_k$  is a root of  $x^P - r = 0$ . Hence, with  $s_i = \frac{r_i}{Pr}$  for  $i = \{1, 2, \dots, P\}$ ,  $G(r_i) = 1$  for all  $P$  values of  $i$ . But, since  $G(x)$  is a polynomial of degree  $P - 1$  it must be true that  $G(x) \equiv 1$ .

It remains to choose  $r$ . It should be selected such that good numerical properties are achieved. In particular,  $r \neq x$ .

The algorithm can be extended to the computation of  $A^P$ , where  $A$  is a matrix. The operation  $1/(A - r_i I) = (A - r_i I)^{-1}$  represents matrix inversion. Thus, for all  $i, r_i \neq \lambda_k$ , where  $\lambda_k, k = \{1, 2, \dots, P$  are the eigenvalues of  $A$ .

To perform the  $\times$  parallel prefix computation we use the above algorithm to compute  $z_2 = x^2, z_3 = x^3, \dots, z_{\sqrt{P}} = x^{\sqrt{P}}$ , which can be accomplished using  $2 + 3 + \dots + \sqrt{P} = \frac{\sqrt{P}(\sqrt{P}+1)}{2} < P$  processors. Then, we compute  $y_i = z_{\sqrt{P}}^i$  for  $i = \{2, 3, \dots, \sqrt{P}\}$ , which again can be accomplished with the algorithm above using less than  $P$  processors. Finally, we compute  $v_{i,j} = y_i z_j$  for  $i, j = \{1, 2, \dots, \sqrt{P} - 1\}$ , which can be made in one step with  $P$  processors. Note that  $v_{i,j} = x^{i\sqrt{P}+j}$ . Hence, the parallel prefix algorithm can be summarized as

1. Compute  $z_2 = x^2, z_3 = x^3, \dots, z_{\sqrt{P}} = x^{\sqrt{P}}$ .
2. Compute  $y_2 = z_{\sqrt{P}}^2, y_3 = z_{\sqrt{P}}^3, \dots, y_{\sqrt{P}} = z_{\sqrt{P}}^{\sqrt{P}}$ .
3. Compute  $v_{i,j} = y_i z_j$  for  $i, j = \{1, 2, \dots, \sqrt{P} - 1\}$ .

The time required is  $2(t_A \log_2 \sqrt{P} + 2(t_A + t_D)) + t_M = t_A \log_2 P + 4(t_A + t_D) + t_M$ , where  $t_M$  is the time for a multiplication.

For the polynomial evaluation  $\sum_{i=1}^P a(i)x^i$ , we can use the  $\times$  parallel prefix algorithm above, followed by one step of parallel multiplication, and a parallel summation. The time is proportional to  $2t_A \log_2 P + \text{const}$ . The factor of two can be removed at the expense of additional processors [16].

Finally we note that a product of the form  $\prod_{i=1}^P (x+a_i)$  can be evaluated using techniques almost identical to those used in computing  $x^P$ . The only difference is that we need not introduce the variable  $r$ , and that  $s_i = (\prod_{j \neq i} (a_j - a_i))^{-1}$ .

### 2.2.5 Additional references

Hardware implementations of linear recurrences have been studied by Gajski [5, 6]. Additional implementation results are also provided by Kogge in [14].

## 3 Higher order recurrences

An  $m$ th order linear recurrence has the form

$$x(j) = a(j, 1)x(j - 1) + a(j, 2)x(j - 2) + \dots + a(j, j - m)x(j - m) + y(j)$$

$m$  initial values are required to start the recurrence. It can be converted into a first order matrix recurrence by defining

$$Z(j) = \begin{bmatrix} x(j) \\ x(j - 1) \\ x(j - 2) \\ \vdots \\ x(j - m + 1) \end{bmatrix}$$

$$A(j) = \begin{bmatrix} a(j, 1) & a(j, 2) & \cdots & a(j, m-1) & a(j, m) \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

and

$$Y(j) = \begin{bmatrix} y(j) \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Then,

$$Z(j) = A(j)Z(j-1) + Y(j)$$

where  $A$  is an  $m \times m$  matrix. However, unlike our  $2 \times 2$  matrices for the first order linear recurrence, the structure of a product of two matrices is no longer the same as that of the matrices forming the product. The matrices fill in with nonzero values. Moreover, each  $Z(j)$  only contains one new component of  $Z$ . Hence, computing the sequence  $Z(2)$   $Z(3)$ ,  $Z(4)$ , etc is very wasteful. The efficiency can be improved by only computing every  $m$ th value.

$$Z((k+1)m) = \left( \prod_{j=km+1}^{(k+1)m} A(j) \right) Z(km) + \sum_{r=km+1}^{(k+1)m} \left( \prod_{j=r+1}^{(k+1)m} A(j) \right) Y(r)$$

Define  $X(k+1) = Z((k+1)m)$ ,  $U(k+1) = \left( \prod_{j=km+1}^{(k+1)m} A(j) \right)$   
and  $V(k+1) = \sum_{r=km+1}^{(k+1)m} \left( \prod_{j=r+1}^{(k+1)m} A(j) \right) Y(r)$ .

Then, the  $m$ th order recurrence can be expressed as

$$X(k+1) = U(k+1)X(k) + V(k+1), \quad k = \{1, 2, \dots, P/m\}$$

$U$  is a dense  $m \times m$  matrix.

## 4 Lower bounds

Odd–even cyclic reduction is inconsistent with sequential parallel prefix computations by about a factor of two. For general linear recurrences odd–even cyclic reduction requires approximately

$5P$  operations versus  $2P$  operations for the sequential algorithm for the evaluation of  $P$  steps of the recurrence.

Parallel cyclic reduction requires about  $P \log_2 P$  arithmetic operations, and is inconsistent by a factor of about  $\log_2 P$  for parallel prefix computations. For general linear recurrences parallel cyclic reduction requires about  $3P \log_2 P$  operations.

This raises the issue of whether a more efficient parallel algorithm than odd–even cyclic reduction exists. Hyafil and Kung [9] have shown the following result.

**Theorem 1** *Any algorithm for evaluation of a general linear recurrence of  $P$  terms, must require at least  $3P - t/2$  operations, where  $t$  is the running time of the algorithm.*

The sequential algorithm requires time  $t = 2P$ , and the number of arithmetic operations must be at least  $3P - (2P)/2 = 2P$ , which indeed is the number of operations for the obvious sequential algorithm. The faster the algorithm runs, the more operations are required, as a minimum. For an algorithm that runs in time  $5 \log_2 P$ , such as odd–even cyclic reduction, the number of operations must be at least about  $3P$ . Hence, odd–even cyclic reduction is suboptimal by at most a factor of  $\frac{5}{3}$  with respect to the total amount of work. Parallel cyclic reduction that runs in time  $3 \log_2 P$  is suboptimal by a factor of about  $\log_2 P$ .

Hyafil and Kung also showed that for a relatively small number of processing nodes,  $N \ll P$ , the best possible speedup for any parallel algorithm is  $\sim \frac{2}{3}N$ .

Kung [16] have also showed that  $\log_2 P$  operations in sequence is a lower bound for the evaluation of *any rational function* of degree  $P$ . The proof is based on induction in the growth of the degree of the polynomials that can be generated with computational nodes with at most two inputs, i.e., a node can add/subtract, multiply or divide two elements in a time of  $t_A$ ,  $t_M$ , or  $t_D$ . Hence, the  $P$ th iteration of a linear recurrence cannot be evaluated in time less than  $\log_2 P$ .

Additional complexity results on parallel prefix computations are contained in [17].

## 5 Non–Linear recurrences

Non–linear recurrences cannot be sped up by more than a constant factor, as proved by Kung [16]. The proof is based on the growth of the degree of a rational function. Let  $x_i = f(x_{i-1})$ , then if the degree of  $f$ , defined as the maximum degree of the nominator or denominator is  $d$ , then the recurrence results in  $x_P$  having degree  $d^P$ . Thus, the time to evaluate a recurrence of degree  $d > 1$  is at least  $P \log_2 d$ , i.e., the time must be proportional to  $P$  as for any sequential algorithm. This result implies that recurrences of the form

$$x_i = \frac{1}{2} \left( x_{i-1} + \frac{1}{x_{i-1}} \right)$$

can be sped up through parallelism by at most a constant factor. (This Newton iteration scheme is the common form for computing square roots.)

## 6 Stability

For a discussion of numerical accuracy in parallel prefix and recurrence computations see for instance [3] and [4].

## References

- [1] A. Borodin and I Munro. Notes on efficient and optimal algorithms. Technical report, University of Waterloo, 1972.
- [2] Billy L. Buzbee, Gene H. Golub, and C W. Nielson. On direct methods for solving Poisson's equations. *SIAM J. Numer. Anal.*, 7(4):627–656, December 1970.
- [3] James Demmel. A specification for floating-point parallel prefix, 1992. Private Communication.
- [4] P. Dubois and Gary Rodrigue. *An Analysis of the Recursive Doubling Algorithm*, pages 299–305. Academic Press, 1977.
- [5] Daniel D. Gajski. Recurrence semigroups and their relation to data storage in fast recurrence solvers on parallel machines. Technical Report UIUCDCS-R-80-1037, Department of Computer Science, Univ of Illinois, 1980.
- [6] Daniel D. Gajski. An algorithm for solving linear recurrence systems on parallel and pipelined machines. *IEEE Trans. Computers*, C-30(3):190–206, March 1981.
- [7] Roger W. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. *J. ACM*, 12:95–113, 1965.
- [8] Roger W. Hockney and C.R. Jesshope. *Parallel Computers*. Adam Hilger, 1981.
- [9] L. Hyafil and H.T. Kung. The complexity of parallel evaluation of linear recurrences. *The Journal of the ACM*, 24(3):513–521, July 1977.
- [10] S. Lennart Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. Statist. Comput.*, 8(3):354–392, May 1987.
- [11] S. Lennart Johnsson and Ching-Tien Ho. Optimizing tridiagonal solvers for alternating direction methods on Boolean cube multiprocessors. extended abstract. In *The Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 96–98, 1990.
- [12] S. Lennart Johnsson, Yousef Saad, and Martin H. Schultz. Alternating direction methods on multiprocessors. *SIAM J. Sci. Statist. Comput.*, 8(5):686–700, 1987.
- [13] Thomas L. Jordan. *A Guide to Parallel Computation and some Cray-1 Experiences*, pages 1–50. Academic Press, 1982.
- [14] P.M. Kogge. Parallel solution of recurrence problems. *IBM J. Res. Dev.*, 18(3):138–148, March 1974.
- [15] P.M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers*, 22(8):786–792, 1973.
- [16] H.T. Kung. New algorithms and lower bounds for the parallel evaluation of certain rational expressions. *Journal of the ACM*, 23(2):252–261, April 1976.

- [17] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *The Journal of the ACM*, 27(4):831–838, October 1980.
- [18] R. Richtmyer and K.W. Morton. *Difference Methods for Initial-Value Problems*. Wiley-Interscience, 1967.
- [19] Paul N. Swarztrauber. The methods of cyclic reduction, Fourier analysis, and the FACR algorithm for the discrete solution of Poisson’s equation on a rectangle. *SIAM Review*, 19:490–501, 1977.
- [20] Clive Temperton. On the FACR(1) algorithm for the discrete Poisson equation. *J. of Computational Physics*, 34:314–329, 1980.