プログラミング－言語・基礎・実践－　5－17
(1991. 11. 22)

ネットワークに埋め込み可能な並列プレフィックスアルゴリズム群

武末 勝
NTT ソフトウェア研究所

あらまし　本論文は、$(v_0, v_1, \ldots, v_{n-1})$ から全てのプレフィックス $x_i = v_0 \theta v_1 \theta \cdots \theta v_i$ $(i = 0, 1, \ldots, n-1)$ を並列にネットワーク内で計算する一群のアルゴリズムを提案する。ただし、θ はアソシアティブな二項演算子である。各アルゴリズムは、オメガネットワーク、ハイパキューブなどのネットワークのスイッチとスイッチ間接続（インターコネクション）内に埋め込まれ、$s \times s$ スイッチ を用いた場合、$\mathbf{O}(\log_s n)$ 時間で実行される。各種のネットワークは相互に等価、あるいは埋め込み可能なので、基本アルゴリズムから上記の一群のアルゴリズムが導かれる。基本アルゴリズムでは、各プロセッサ p_i は 値 v_i をプロセッサ p_k $(k = i+1, i+2, \ldots, n-1)$ にマルチキャストする。p_i への道筋のネットワークスイッチ内で v_j $(j = 0, 1, \ldots, i-1)$ が相互に結合され、$(i-1)$ 番目のプレフィックス x_{i-1} が計算されて p_i に受信される。たがって、p_i は i 番目のプレフィックス $x_i = x_{i-1} \theta v_i$ を計算できる。

A Family of Parallel Prefix Algorithms Embedded in Networks

Masaru TAKESUE
NTT Software Laboratories
3-9-11 Midori-Cho, Musashino-Shi, Tokyo 180 Japan
(phone: 0422 (59) 3896, e-mail: takesue@lucifer.ntt.jp)

Abstract This paper presents a family of algorithms for producing, from $(v_0, v_1, \ldots, v_{n-1})$, all initial prefixes $x_i = v_0 \theta v_1 \theta \cdots \theta v_i$ $(i = 0, 1, \ldots, n-1)$ in parallel in interconnection networks such as the omega network and hypercube, where θ is an associative binary operator. Each algorithm can be embedded in the switches and interconnections of the network, and can be executed in $\mathbf{O}(\log_s n)$ time steps provided that the network connecting n processors is constructed by using an $s \times s$ switch. The objective of these algorithms is thus not necessarily to improve the time and space required to execute them, but to attain a communication pattern that fits the topology of the network. Because one type of network can be made equivalent to, or can be embedded in, another type of network, a family of algorithms can be derived from one basic algorithm. In the basic algorithm, in principle, every processor p_i multicasts v_i to the processors p_k $(k = i+1, i+2, \ldots, n-1)$. En route to p_i, v_j $(j = 0, 1, \ldots, i-1)$ are combined with each other in the switches to calculate the $(i-1)$-th initial prefix x_{i-1} that is received by p_i, which thus computes the i-th initial prefix $x_i = x_{i-1} \theta v_i$.

Key words Parallel prefix operation, algorithm embedding, embedding in networks, combining network, multistage networks, hypercube.

1 Introduction

The network is the key component of a multiprocessing system, and should be used efficiently as computing power for parallel computation. However, the primitives for efficient parallel computation are not yet clear, and should be determined and supported at the architecture level, especially in the network. One such primitive is the parallel prefix operation on $(v_0, v_1, \ldots, v_{n-1})$, which computes the i-th initial prefix $\theta_{k=0}^{i} v_k = v_0 \theta v_1 \theta \cdots \theta v_i$ for all i $(0 \leq i \leq n-1)$, where θ is an associative binary operator [3]. The parallel prefix operation is effective in many algorithms such as the evaluation of polynomials [1], the solution to recurrence equations [2], boolean circuits [3], sorting and merging, and graph [8] and list manipulations [9]. A parallel prefix algorithm, which is here referred to as the mask shuffling algorithm, was first suggested by Stone [1] for polynomial evaluation. Since then, many of parallel prefix algorithms have been proposed, and these can be classified, in principle, as recursive doubling [2, 4, 6, 7], and as tree sweeping [3, 5, 8].

The mask shuffling and recusive doubling algorithms cannot easily be embedded in multistage (or indirect connection) networks such as the omega network [10], because it is in general difficult to provide necessary data to every network switch, where the main calculation of these algorithms is embedded. On the other hand, the tree sweeping algorithm needs two sweeps of the tree; in the first sweep, a partial result is left in each node, and, in the second sweep, the partial result is recombined with another partial result in the node. Hence, the second partial results must return via the same route in the reverse direction in the network. This algorithm can thus be embedded in a multistage network with recombining capability.

This paper presents a family of new parallel prefix algorithms that can be easily embedded in indirect (multistage) networks, without the recombining capability, such as the omega network, as well as in direct networks such as the hypercube. Hence, the purpose of these algorithms is not necessarily to reduce the time and space required to execute them, but to attain a communication pattern in the algorithms that fits in with the topology of the network.

The rest of this paper is organized as follows. Section 2 outlines the properties of representative networks, presents the principle behind the family of algorithms, and proposes two basic parallel prefix algorithms that are equivalent to each other, but cannot be embedded in the networks. Section 3 presents, as extensions of the basic algorithms, a family of parallel prefix algorithms that can be embedded in the indirect and direct networks. Section 4 concludes this paper.

2 Foundations

2.1 Outline of Networks

Representative networks, i.e., the omega network, delta network, indirect binary n-cube, and hypercube, and the relationships among them are outlined here. A common structure of multistage networks is shown in Fig. 1.

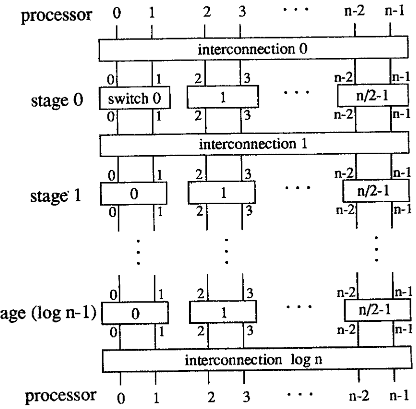

Fig. 1. A common structure of multistage networks.

In the following, it is assumed that a 2×2 switch is used in multistage networks. A multistage network is then composed of $m = \log_2 n$ stages, each stage comprising $n/2$ switches, where n is the number of processors in the system. It is also assumed that the input and output of the network are the same n processors.

1) Indirect (or Multistage) Networks: In the omega network, every interconnection (see Fig. 1), except the last one, implements a *shuffle* function; an address $A = a_{m-1}a_{m-2}\cdots a_0$ in binary expression is mapped as $\mathit{shuffle}(a_{m-1}a_{m-2}\cdots a_0) = a_{m-2}a_{m-3}\cdots a_0a_{m-1}$. The last interconnection equals an identity function; $\mathit{id}(A) = A$. The network switch implements a $\mathit{replace}_0$ function that replaces the least significant bit (LSB) of an address with a given bit b; $\mathit{replace}_0(a_{m-1}a_{m-2}\cdots a_0, b) = a_{m-1}a_{m-2}\cdots a_1b$. A packet is sent from a source processor with address $S = s_{m-1}s_{m-2}\cdots s_0$ to a destination processor with address $D = d_{m-1}d_{m-2}\cdots d_0$ by recursively applying $S_{k+1} = \mathit{replace}_0(\mathit{shuffle}(S_k), d_{m-1-k})$ with k from 0 to $m-1$, where $S_0 = S$ and $S_m = D$.

As described above, the omega network replaces s_{m-1-k} with d_{m-1-k} in stage k. Accordingly, viewing the omega network from another angle, the destination address D can be attained if a cube_{m-1-k} function is recursively applied ($0 \leq k \leq m-1$) to the source address S, where $\mathit{cube}_k(a_{m-1}a_{m-2}\cdots a_k\cdots a_0) = a_{m-1}a_{m-2}\cdots\overline{a_k}\cdots a_0$ [14]. That is, assuming that $\mathit{cube1}_k(A, t)$ is a function which executes $\mathit{cube}_k(A)$ only when $t = 1$, and that $S \oplus D = T$ ($= t_{m-1}t_{m-2}\cdots t_0$) ($\oplus$: exclusive OR), then $D_{k+1} = \mathit{cube1}_{m-1-k}(D_k, t_{m-1-k})$, where $D_0 = S$ and $D_m = D$. The omega network can therefore be embedded in the hypercube as described in *2)*.

The delta network [12] and the indirect binary n-cube (ICube) are equivalent to the omega network with renumbering [11]. That is, with shuffled S, the delta network is equivalent to the omega network; Delta($\mathit{shuffle}(S), D$) $\equiv$ Omega(S, D). Similarly, with reversed S and D, the ICube is equivalent to the omega network; ICube($\mathit{reverse}(S), \mathit{reverse}(D)$) $\equiv$ Omega(S, D).

2) Direct Network (Hypercube): The hypercube [13] is a direct connection network where 2^m ($= n$) switches are interconnected by direct links to organize an m-dimensional hypercube; the switch with address S is interconnected with m switches with addresses $\mathit{cube}_i(S)$ ($i = 0, 1, \ldots, m-1$). Only one processor is attached to a switch. A packet is routed from a processor S to a processor D in r steps, where r is the Hamming distance between S and D. That is, assuming that r bits $t_{one(r-1)}, t_{one(r-2)}, \cdots, t_{one(0)}$ of the routing tag T ($= S \oplus D$) are equal to 1, the packet is routed from the switch with address D_k to switch $D_{k+1} = \mathit{cube}_{one(r-1-k)}(D_k)$, where $D_0 = S$ and $D_r = D$. This routing method is in principle the same as the method with $\mathit{cube1}$ in the omega network, $D_{k+1} = \mathit{cube1}_{m-1-k}(D_k, t_{m-1-k})$. The omega network can thus be embedded in the hypercube.

2.2 Principle for Embedding

This section focuses on embedding in the omega network, and hence on the *shuffle* function (topology), since representative networks can be made equivalent to, or can be embedded in the omega network as described in the previous section. Assume that the omega network can upwardly multicast v_i in p_i to p_k ($k = i, i+1, \ldots, n-1$). When each processor p_i upwardly multicasts v_i ($i = 0, 1, \ldots, n-1$), p_i receives $v_0, v_1, \cdots, v_i$, and hence can compute the i-th initial prefix $x_i = \theta_{k=0}^{i} v_k$. This trivial algorithm requires $\mathbf{O}(n)$ computation steps, however suggests the possibility of a nontrivial one; if v's can be combined in the network to produce partial initial prefixes, the computation steps will be reduced to $\mathbf{O}(\log n)$.

To upwardly multicast and combine v's in the network, let's introduce two functions, Θ and σ. Assume that V, $V^{(k)}$ ($1 \leq k \leq m$), W, $W^{(k)}$, and X are n-dimensional vectors of which elements respectively equal v_i, $v_i^{(k)}$, w_i, $w_i^{(k)}$, and x_i ($0 \leq i \leq n-1$). Then, $\Theta(V, W)$ is a function that returns $(V, W^{(1)})$, where $w_{2k}^{(1)} = w_{2k}\theta w_{2k+1}$ and $w_{2k+1}^{(1)} = w_{2k}\theta w_{2k+1}\theta v_{2k}$ ($0 \leq k \leq n/2 - 1$). Note that $n = 2^m$. Another function $\sigma(V, W) = (\sigma(V), \sigma(W))$ returns $(V^{(1)}, W^{(1)})$, where $v_k^{(1)} = v_{shuffle^{-1}(k)}$ and $w_k^{(1)} = w_{shuffle^{-1}(k)}$ ($0 \leq k \leq n-1$). That is, $\sigma(V, W)$ shuffles the indexes of V and W. With these two functions, n initial prefixes X are produced in parallel from V as follows, where the composition $f(g(z))$ of two function, f and g, is represented as $f \circ g$.

[*Theorem 1*] Let $(V^{(j)}, W^{(j)})$ be the result of j application of $\Theta \circ \sigma$ to (V, W), $(\Theta \circ \sigma)^j(V, W)$, with initial $V = (v_0, v_1, \cdots, v_{n-1})$ and initial W of $\mathbf{0}$ ($w_k = 0$ for all k). Then, the i-th initial prefix $x_i = \theta_{k=0}^{i} v_k$ is equal to $w_i^{(m)}\theta v_i^{(m)}$; that is,

$X = W^{(m)}\theta V^{(m)}$.

(*Proof*) Because the index (or address) i of v_i is composed of m bits, and because $\mathit{shuffle}^m(i) = i$, $V^{(m)} = V$. It is thus sufficient to prove that $w_i^{(m)} = x_{i-1}$, the $(i-1)$-th initial prefix. At the s-th application of Θ (after the s-th application of σ), the arguments $v_{2k}^{(s)}$ and $v_{2k+1}^{(s)}$ equal respectively such v_p and v_q as $q = \mathit{cube}_{m-s}(p)$. This is because $\mathit{shuffle}^{-s}(2k) = \mathit{cube}_{m-s}(\mathit{shuffle}^{-s}(2k+1))$. Hence, v_p affects $w_{2k+1}^{(s+1)}$ $(= w_{2k}^{(s)}\theta w_{2k+1}^{(s)}\theta v_p)$. To probe the index j of v_j that affects $w_i^{(m)}$, let's start the probing from the last application of Θ. Assuming that $i = a_{m-1}a_{m-2}\cdots a_0$, in the m-th application, the index j of the v_j that may affect the $w_i^{(m)}$ equals $j = a_{m-1}a_{m-2}\cdots\overline{a_0}$. The $w_i^{(m)}$ and $w_j^{(m)}$ (i or $j = 2k$ and j or $i = 2k+1$) may be affected in the $(m-1)$-th application by the v's whose indexes equal $a_{m-1}a_{m-2}\cdots\overline{a_1}x$ (x means don't care). Similarly, the v's with indexes $a_{m-1}a_{m-2}\cdots\overline{a_k}xx\cdots x$ in the $(m-k)$-th application, and with indexes $\overline{a_{m-1}}xx\cdots x$ in the 1st application, may affect the value $w_i^{(m)}$. Of these $a_{m-1}a_{m-2}\cdots\overline{a_k}xx\cdots x$ $(k = m-1, m-2, \ldots, 0)$, v's with indexes satisfying $\overline{a_k} = 0$ really affect the $w_i^{(m)}$. Consequently, assuming that bit $a_{one(k)}$ $(r-1 \geq k \geq 0)$ of i equals one, the v's with indexes $a_{m-1}a_{m-2}\cdots a_{one(k)-1}\overline{a_{one(k)}}xx\cdots x$ $(k = r-1, r-2, \ldots, 0)$, which are equal to $i-1, i-2, \ldots, 0$, are combined to produce $w_i^{(m)}$. That is, $w_i^{(m)} = \theta_{k=0}^{i-1} v_k = x_{i-1}$, and therefore $x_i = w_i^{(m)}\theta v_i^{(m)}$. (Q.E.D.)

With this theorem, the time steps required to perform the parallel prefix operation is $\mathbf{O}(2\log_2 n)$, because 2 steps are required in the Θ. In general, considering a chunk of s data in place of the two-data chunk, w_{2k} and w_{2k+1} (and v_{2k} and v_{2k+1}), $\Theta(V, W)$ is extended so as to return $(V, W^{(1)})$ where $w_{sk+j}^{(1)} = (\theta_{i=0}^{s-1} w_{sk+i})\theta(\theta_{i=0}^{j-1} v_{sk+i})$ $(0 \leq k \leq n/s - 1,\ 0 \leq j \leq s-1)$. Since the first and second terms are computed independently by using, for example, the recursive doubling algorithm in $\log_2 s$ steps, $w_{sk+j}^{(1)}$ is calculated in $\log_2 s + 1$ steps. Hence, with theorem 1, the parallel prefix operation is carried out in $\mathbf{O}((\log_2 s + 1)\log_s n)$ time steps in the general case.

As described in the proof of theorem 1, $\mathit{shuffle}^j(p) = 2k$ and $\mathit{shuffle}^j(q) = 2k+1$, provided that $q = \mathit{cube}_{m-j}(p)$ $(1 \leq j \leq m)$. Accordingly, by using the *cube* function, the parallel prefix operation can be carried out in the following way. A functions $\Theta_j(V, W)$ that is used in place of Θ and σ is defined to produce $(V, W^{(1)})$ such that $w_i^{(1)} = w_i\theta w_c$ if bit j (note that the LSB is bit 0) of i equals 0, otherwise, $w_i^{(1)} = w_i\theta w_c\theta v_c$, and $c = \mathit{cube}_j(i)$ $(m-1 \leq j \leq 0)$. That is, Θ_j applies Θ to w_i and w_c (and v_c). Notice that Θ is equal to Θ_0.

[*Theorem 2*] Assume that $(V^{(j)}, W^{(j)})$ is the result of j applications of Θ_k to (V, W) with k from $m-1$ to $m-j$, $(\Theta_{m-j} \circ \cdots \circ \Theta_{m-2} \circ \Theta_{m-1})(V, W)$, with initial $V = (v_0, v_1, \ldots, v_{n-1})$ and initial W of $\mathbf{0}$. Then, all initial prefixes X are given by $X = W^{(m)}\theta V^{(m)}$.

(*Proof*) The proof is omitted, because this theorem is equivalent to theorem 1, and the proofs are also equivalent.

2.3 Basic Algorithms

Two basic parallel prefix algorithms, a recursive shuffling and recursive cubing algorithms, are presented here. These algorithms can be executed in parallel by n processors, but they cannot be embedded in the network.

```
procedure recursive_shuffling();
  {for j:=0 to log n-1 do
    {forall i:= 0 to n-1 do in parallel
      {new_V[shuffle(i)]:= V[i];
       new_W[shuffle(i)]:= W[i];}
     forall i:= 0 to n-1 do in parallel
      {if bit_0(i) = 0 then
         W[i]:= new_W[i] θ new_W[i+1];
       else
         W[i]:= new_W[i] θ new_W[i-1]
                θ new_V[i-1];
       V[i]:= new_V[i];}}
  forall i:= 0 to n-1 do in parallel
    X[i]:= V[i] θ W[i];}
```

Fig. 2. The recursive shuffling algorithm.

The recursive shuffling algorithm (Fig. 2) is a straightforward implementation based on theorem 1. This algorithm uses five arrays of n elements, V and new_V for v's, W and new_W for w's, and X for x's. Initially, elements of V[i] is set equal to v_i, and those of other arrays are set to zero. The first forall statement corresponds to the function σ, and the second forall carries out the same function as Θ. The last forall executes $x_{i-1}\theta v_i$ to leave x_i in X[i]. The recursive cubing algorithm shown in Fig. 3 is an implementation of theorem 2. The first forall statement corresponds to Θ_j.

An example of the recursive shuffling algorithm when $n = 8$ is shown in Fig. 4. V and W are initial vectors, and the two vectors above the double lines show the result of $\sigma(V^{(k)}, W^{(k)})$ $(0 \le k \le 2)$. Notice that the output $V^{(k+1)}$ of $\Theta(\sigma(V^{(k)}), \sigma(W^{(k)}))$ equals the input $\sigma(V^{(k)})$, and hence the first vector denoted $V^{(k+1)}$ above the double lines means both the first input and the first output of Θ, of which output $W^{(k+1)}$ is shown under the double lines.

```
procedure recursive_cubing();
  {for j:= log n-1 down to 0 do
    {forall i:= 0 to n-1 do in parallel
      {if bit$_j$(i) = 0 then
        new_W[i]:= W[i] $\theta$ W[cube$_j$(i)];
      else
        new_W[i]:= W[i] $\theta$ W[cube$_j$(i)]
                   $\theta$ V[cube$_j$(i)];}
    forall i:= 0 to n-1 do in parallel
      W[i]:= new_W[i];}
  forall i:= 0 to n-1 do in parallel
    X[i]:= V[i] $\theta$ W[i];}
```

Fig. 3. The recursive cubing algorithm.

The 5-th initial prefix x_5, for instance, is computed as follows. In the first application of Θ, the partial inputs $(v_{2k}, v_{2k+1}, w_{2k}$, and $w_{2k+1})$ when $k = 1$ are $v_2^{(1)} = v^{(0)}_{shuffle^{-1}(2)} = v_1^{(0)}$ $(= v_1)$, $v_3^{(1)} = v_5$, $w^{(0)}_{shuffle^{-1}(2)} = w_1$, and $w^{(0)}_{shuffle^{-1}(3)} = w_5$. Hence, $w_2^{(1)} = w_1\theta w_5 = 0$, and $w_3^{(1)} = w_1\theta w_5\theta v_1 = v_1$. Similarly, in the second application of Θ and when $k = 3$, the partial inputs are v_5 $(= v_3^{(1)})$, v_7 $(= v_7^{(1)})$, v_1 $(= w_3^{(1)})$, and v_3 $(= w_7^{(1)})$, and $w_6^{(2)}$ and $w_7^{(2)}$ are thus equal to $v_1\theta v_3$ $(= \theta_0^1 v_{2i+1})$ and $v_1\theta v_3\theta v_5$ $(= \theta_0^2 v_{2i+1})$, respectively. Then, in the third application and when $k = 2$, v_4, v_5, $\theta_0^1 v_{2i}$, and $\theta_0^1 v_{2i+1}$ are the partial inputs, and the result $w_4^{(3)}$ and $w_5^{(3)}$ thus equals $\theta_0^3 v_i$ and $\theta_0^4 v_i$. Finally, the fifth initial prefix x_5 is achieved by $w_5^{(3)}\theta v_5^{(3)} = (\theta_0^4 v_i)\theta v_5$.

		k=0		1		2		3	
	V	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
	W	0	0	0	0	0	0	0	0
σ									
	$V^{(1)}$	v_0	v_4	v_1	v_5	v_2	v_6	v_3	v_7
		0	0	0	0	0	0	0	0
Θ									
	$W^{(1)}$	0	v_0	0	v_1	0	v_2	0	v_3
σ									
	$V^{(2)}$	v_0	v_2	v_4	v_6	v_1	v_3	v_5	v_7
		0	0	v_0	v_2	0	0	v_1	v_3
Θ									
	$W^{(2)}$	0	v_0	$\theta_0^1 v_{2i}$	$\theta_0^2 v_{2i}$	0	v_1	$\theta_0^1 v_{2i+1}$	$\theta_0^2 v_{2i+1}$
σ									
	$V^{(3)}$	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
		0	0	v_0	v_1	$\theta_0^1 v_{2i}$	$\theta_0^1 v_{2i+1}$	$\theta_0^2 v_{2i}$	$\theta_0^2 v_{2i+1}$
Θ									
	$W^{(3)}$	0	v_0	$\theta_0^1 v_i$	$\theta_0^2 v_i$	$\theta_0^3 v_i$	$\theta_0^4 v_i$	$\theta_0^5 v_i$	$\theta_0^6 v_i$
	X	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Fig. 4. Parallel prefix operation with functions σ and Θ; $(\Theta\circ\text{shuffle})^3(V, W)$ produces $x_i = \theta_{k=0}^{i} v_k$ $(0 \leqq i \leqq 7)$.

3 Embedded Algorithms

3.1 The Algorithms for the Omega Network

A parallel prefix algorithm that can be embedded in the omega network, the omega traverse algorithm, is shown in Fig. 5. This algorithm uses two-dimensional arrays for V and W. The arrays A[j,2*i] and A[j,2*i+1] (A = V or W; $0 \le j \le \log n - 1$; $0 \le i \le n/2 - 1$) are implemented in switch i of stage j. A[-1,i] and A[log n,i] are data in processor p_i, and V[-1,i] and W[-1,i] are initialized to v_i and 0, respectively. The italic statements show the operations performed via the interconnections (Fig. 1), and the rest except for the last statement are executed in the switches; the last statement X[ ]:= V[ ] θ W[ ] is executed in the processors. The first forall statement in the for-loop corresponds to the function σ, which is thus implemented in the interconnection of the original omega network. The second forall in the for-loop implements the function Θ. Hence, this algorithm is equivalent to the recursive shuffling algorithm, and can compute all initial prefixes in parallel in $2\log_2 n + 1$ time steps; +1 is for computing X[ ].

```
procedure omega_traverse();
  {for j:=0 to log n-1 do
    {forall i:= 0 to n-1 do in parallel
      { V[j,shuffle(i)]:= V[j-1,i];
        W[j,shuffle(i)]:= W[j-1,i];}
     forall i:= 0 to n/2-1 do in parallel
      {W[j,2*i]:= W[j,2*i] θ W[j,2*i+1];
       W[j,2*i+1]:= W[j,2*i] θ V[j,2*i];}}
   forall i:= 0 to n-1 do in parallel
    { V[log n,i]:= V[log n-1,i];
      W[log n,i]:= W[log n-1,i];
      X[i]:= V[log n,i] θ W[log n,i];}}
```

Fig. 5. The omega traverse algorithm.

An example of this algorithm is the same as the one shown in Fig. 4, provided that V, W, and X are in the processors, that $v_{2k}^{(j)}$, $w_{2k+1}^{(j)}$ $(0 \leq k \leq 3,\ 1 \leq j \leq 3)$ and Θ are implemented in switch k of stage $j-1$, and that σ is implemented by the interconnection (function *shuffle*).

Since Delta(*shuffle*$(S), D) \equiv$ Omega(S, D) as discussed in section 2, a delta traverse algorithm (omitted for the space) is the same as the omega traverse except when $j = 0$, where *shuffle*(S) is performed by n processors by using normal packets. The number of time steps required for the parallel prefix operation in the delta traverse is $2\log_2 n + 2$.

3.2 The Algorithm for the Indirect Binary n-Cube

ICube(*reverse*(S),*reverse*(D)) $\equiv$ Omega(S, D), so the ICube traverse algorithm shown in Fig. 6, where use of arrays V, W, and X is the same as in Fig. 5, reverses the indexes of v_i in the first and last statements. These reverses are performed by the processors. The second forall statement performs the function *id* of interconnection 0 of the ICube, and the first and second foralls in the for-loop correspond to the Θ in every switch and the *shuffle*$^{-1}$ interconnection, respectively.

```
procedure indirect_binary_n_cube_traverse();
  {forall i:= 0 to n-1 do in parallel
    V[-1,reverse(i)]:= V[-1,i];
   forall i:= 0 to n-1 do in parallel
    { V[0,i]:= V[-1,i];
      W[0,i]:= W[-1,i];}
   for j:=0 to log n-1 do
    {forall i:= 0 to n/2-1 do in parallel
      {W[j,2*i]:= W[j,2*i] θ W[j,2*i+1];
       W[j,2*i+1]:= W[j,2*i] θ V[j,2*i];}
     forall i:= 0 to n-1 do in parallel
      { V[j+1,shuffle^-1(i)]:= V[j,i];
        W[j+1,shuffle^-1(i)]:= W[j,i];}}
   forall i:= 0 to n-1 do in parallel
    X[reverse(i)]:= V[log n,i] θ W[log n,i];}
```

Fig. 6. The indirect binary n-cube traverse algorithm.

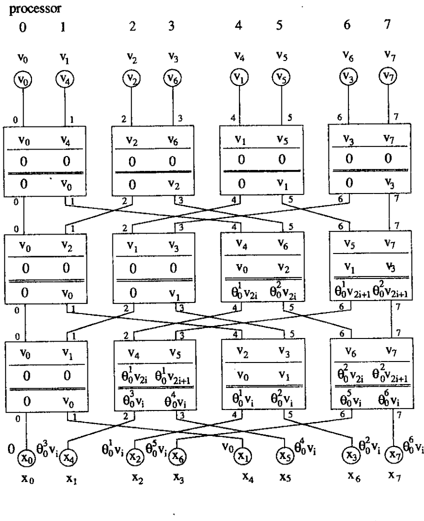
processor
0
1
2
3
4
5
6
7
stage
0
1
2
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7

Fig. 7. Parallel prefix operation with the indirect binary n-cube traverse algorithm.

Hence, without the reverse of the indexes, this algorithm can be embedded in the ICube. The number of time steps required for executing the parallel prefix operation is $2\log_2 n + 2$.

An example of this algorithm when $n = 8$ is shown in Fig. 7, where the notation is almost the same as in Fig. 4. The interconnection functions except for the first are *shuffle*$^{-1}$. Let's trace to get the fourth initial prefix x_4. Processor 4 sends v_4 to processor $reverse(4) = 1$, that sends v_4 to switch 0 in stage 0. In the switch, W[0,1] (corresponding to $w_1^{(1)}$ of theorem 1) becomes equal to v_0, and is sent with v_4 to input 4 (= *shuffle*$^{-1}$(1), where 1 means output 1 of stage 0) of switch 2 in stage 1, where W[1,4] ($w_4^{(2)}$) is set equal to $v_0 \theta v_2$ (= $\theta_0^1 v_{2i}$). Similarly, this value is sent with v_4 to input 2 of switch 1 in stage 2 to produce W[2,2] ($w_2^{(3)}$) of $\theta_0^3 v_i$, which is sent with v_4 to processor 1. Processor 1 combines the two received values to produce $(\theta_0^3 v_i)\theta v_4 = \theta_0^4 v_i = x_4$, and sends it to processor $reverse(1) = 4$. Consequently, processor 4 receives the 4-th initial prefix x_4.

3.3 The Algorithm for the Hypercube

An algorithm which can be embedded in the hypercube is shown in Fig. 8. This hypercube traverse algorithm is in principle the same as the recursive cubing shown in Fig. 3. Initially, v_i and 0 are held in V[-1,i] and W[log n,i] of processor p_i, respectively. The first forall statement initializes the switches; p_i sends v_i and w of 0 to switch i belonging to p_i. The for-loop is equivalent to the for loop in Fig. 3. However, to decrease the amount of communication between switches i and $cube_j$(i), switch i sends W[j,i] θ V[0,i] or W[j,i] to W[j-1,$cube_j$(i)] in switch $cube_j$(i), depending on bit j of address i (bit$_j$(i)). Hence, in the second forall of the for-loop, if bit$_j$(i) equals 1, W[j-1,i] becomes equal to W[j,i]θW[j,$cube_j$(i)]θV[0,$cube_j$(i)], or W[j,i]θW[j,$cube_j$(i)] otherwise. The for-loop thus performs the function Θ_j. In the last forall statement, the i-th initial prefix value x_i is sent to X[i] in p_i. This algorithm performs the parallel prefix operation in $2\log_2 n + 2$ time steps.

An example of the hypercube traverse algorithm when $n = 8$ (3-dimensional cube) is shown in Fig. 9, where (a), (b), and (c) show the operations by $\Theta_j(V^{(2-j)}, W^{(2-j)})$ for j of 2, 1, and 0, respectively. A line with an arrow along the edge means that a value is transferred from the starting vertex to the ending vertex. The edge with only one transfer in figures (a) to (c) means that 0 is sent in the opposite direction to that shown by the line with the arrow. The value in parentheses near vertex i means w_i (W[i] shown in Fig. 3) after the application of Θ_j.

```
procedure hypercube_traverse();
  {forall i:= 0 to n-1 do in parallel
     {V[0,i]:= V[-1,i];
      W[log n-1,i]:= W[log n,i];}
   for j:= log n-1 down to 0 do
     {forall i:= 0 to n-1 do in parallel
        if bit_j(i) = 0 then
           W[j-1,cube_j(i)]:= W[j,i]
                          θ V[0,i];
        else
           W[j-1,cube_j(i)]:= W[j,i];
      forall i:= 0 to n-1 do in parallel
         W[j-1,i]:= W[j,i] θ W[j-1,i];}
   forall i:= 0 to n-1 do in parallel
     X[i]:= V[0,i] θ W[-1,i];}
```

Fig. 8. The hypercube traverse algorithm.

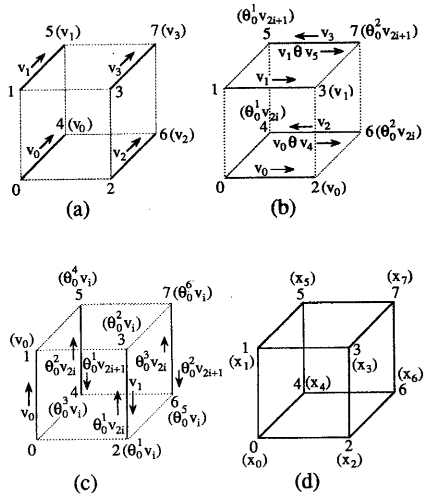

Fig. 9. Parallel prefix operation with the hypercube traverse algorithm. The computing proceeds from (a) to (d).

For instance, the operations to attain x_4 are as follows. In (a), vertex 4 receives v_0 from vertex 0, and produces $w_4 = v_0$. Similarly, relating vertices 5 and 6 set w_5 and w_6 equal to v_1 and v_2. In figure (b), vertex 4 sends $w_4 \theta v_4 = v_0 \theta v_4$ to vertex 6, receives w_6 $(= v_2)$ from it, and produces $w_4 \theta w_6 = v_0 \theta v_2$ $(= \theta_0^1 v_{2i})$ as the new w_4. Similarly, the new w_5 equals the $\theta_0^1 v_{2i+1}$ that is received by vertex 4 in figure (c). Accordingly, result w_4 becomes equal to $\theta_0^3 v_i$. Finally, in figure (d), x_4 is attained by combining w_4 and v_4; $w_4 \theta v_4 = \theta_0^4 v_i$.

4 Conclusions

A family of parallel prefix algorithms, that can be embedded in the switches of the cube-type networks, i.e., omega network, delta network, indirect binary n-cube, and hypercube, have been presented. These algorithms are based on two basic algorithms, the recursive shuffling and recursive cubing algorithms. These two algorithms are equivalent. The number of time steps required to execute the parallel prefix operation in the network is $\mathbf{O}((\log_2 s + 1) log_s n)$, where n is the number of processors in the system, and the network is composed by using an $s \times s$ switch.

The recursive shuffling algorithm alternately executes two functions, σ (or *shuffle*) and Θ, and hence is straightforwardly embedded in the omega network by implementing the latter function in each switch; the former is the interconnection function of the original omega network. The recursive cubing algorithm can be constructed by using the function *cube* in place of *shuffle*. The delta network and indirect binary n-cube are equivalent to the omega network with renumbering, and hence the algorithm for the omega network can be easily modified so as to be embedded in these networks. The algorithm for the hypercube is achieved based on the basis of recursive cubing.

Acknowledgments

The author would like to acknowledge the stimulating discussions with the members of the Software Laboratories.

References

[1] H. S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Comput., Vol. C-20, No. 2, pp. 153-161, Feb. 1971.

[2] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," IEEE Trans. on Comput., Vol. C-22, No. 8, pp. 786-793, Aug. 1973.

[3] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," Jour. ACM, Vol. 27, No. 4, pp. 831-838, Oct. 1980.

[4] C. P. Kruskal, L. Rudolph, and M. Snir, "The Power of Parallel Prefix," International Conference on Parallel Processing (ICPP), pp. 180-185, Aug. 1985.

[5] C. P. Kruskal, L. Rudolph, and M. Snir, "Efficient Synchronization on Multiprocessors with Shared Memory," Annual ACM Symposium on Principle of Distributed Computing, pp. 218-228, Aug. 1986.

[6] B. D. Lubachevsky and A. G. Breenberg, "Simple, Efficient Asynchronous Parallel Prefix Algorithms," ICPP, pp. 66-69, Aug. 1987.

[7] R. Cole and U. Vishkin, "Faster Optimal Parallel Prefix Sums and List Ranking," Information and Control, Vol. 81, pp. 334-352, 1989.

[8] G. E. Blelloch, "Scans as Primitive Parallel Operations," IEEE Trans. on Comput., Vol. 38, No. 11, pp. 1526-1538, Nov. 1989.

[9] M. Takesue, "D-Lisp: A Parallel Processing Lisp based on Distributed List," Proc. 1989 Joint Symposium on Parallel Processing (JSPP'89) sponsored by IPSJ and IEICE of Japan, pp. 259-266, Feb. 1989.

[10] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," IEEE Trans. on Comput., Vol. C-24, No. 12, pp. 1145-1155, Dec. 1975.

[11] M. C. Pease, "The Indirect Binary n-Cube Microprocessor Array," IEEE Trans. on Comput., Vol. C-26, No. 5, pp. 458-473, May 1977.

[12] J. H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," IEEE Trans. on Comput., Vol. C-30, No. 10, pp. 771-780, Oct. 1981.

[13] K. Padmanabhan, "Cube Structures for Multiprocessors," Commu. ACM, Vol. 33, No. 1, pp. 43-52, Jan. 1990.

[14] H. J. Siegel, "Interconnection Networks for Large-Scale Parallel Processing," Reading, Lexington Books, 1985.