PARALLEL PREFIX OPERATION ON THE OPTICAL COMMUNICATION PARALLEL MODEL AND APPLICATIONS

Luis A. Trejo

Depto. de Ciencias Computacionales.
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
e-mail: ltrejo@sunserv.cem.itesm.mx

Abstract: The parallel prefix operation has been considered as a basic building block for parallel algorithms. It is in fact, one of the parallel operation primitives in Blelloch's Scan Model of computation and in its more general form, the multiprefix operation, it is one of the main parallel primitives on which Ranade's Fluent abstract machine relies. We believe that in a similar way, the OCP (Optical Communication Parallel) model can be provided with some basic parallel primitives, namely a prefix (and multiprefix) operation and an operation to realize arbitrary $h$-relations. We show in this paper that such primitives can be efficiently implemented on the OCP model. We illustrate the use of the primitives by giving an $O((\log R/\log m)((n/p) + m \log p))$ radix sort algorithm.

Key words: scan operation, multiprefix, optical communication, radix sort, parallel primitive, computation model.

1 Introduction

The parallel prefix computation, also known as all partial sums or scan operation, appears as an elementary operation during the solution of more general problems, as in the solution of tridiagonal system of equations and $m$th-order linear recurrence systems; when implementing the multiprefix operation or a radix sort, etc. Formally, the parallel prefix computation can be defined as to be given inputs $x_0, x_1, \ldots, x_{n-1}$ compute outputs $x_0 \oplus x_1 \oplus \cdots \oplus x_i$ for $0 \leq i \leq n - 1$ where $\oplus$ is an associative operator.

The parallel prefix operation has been considered as a basic building block for parallel algorithms. It is in fact, one of the parallel operation primitives in the Scan model of computation [4]. Currently, the scan primitives are part of the parallel instruction set (PARIS) of the CM-2. In the same paper, Blelloch suggested that some scan operations be included as unit time primitives in the P-RAM models since they could be efficiently implemented to perform as fast as memory references to a shared memory. Therefore, in the Scan model (P-RAM model + scan primitives), the asymptotic performance of many P-RAM algorithms can be improved by an $O(\log n)$ factor; moreover, the description of such algorithms becomes simpler.

By way of contrast, Ranade et al. in [17] consider the scan operation as a special case of a more general one: the multiprefix operation. The multiprefix operation is one of the instruction primitives on which Ranade’s Fluent abstract machine relies. The Fluent machine consists of $14 \times 2^{13}$ nodes organized as a 13-dimensional butterfly connected to a shared memory which is organized as follows: $M$ shared variables are randomly distributed among the $p$ processors, so each processor is assigned $M/p$ shared variables. Hence, the heart of the Fluent machine relies on a very simple router, which effectively eliminates the possibility of congestion. The multiprefix operation in the Fluent machine terminates in $O(\log p)$ steps with overwhelming probability. Conversely to the multiprefix operation on the Fluent machine, we will see later in section 5 how our proposition to implement
the multiprefix operation relies on the use of prefix operations. In this paper we show how the parallel prefix and the multiprefix operations, can be efficiently implemented on the OCP (Optical Communication Parallel) model. We believe that the OCP model can be provided with such basic parallel primitives including an operation to realize arbitrary $h$-relations.

This paper is organized as follows: in the following section, a brief survey of previous work on the parallel prefix problem and its applications is given. In section 3 the model of computation used through this paper is described. Next, in section 4, the implementation of the parallel prefix operation on such a model is presented. Finally in section 5, applications using the prefix operation as a building primitive are given. They include a radix sort for both cases, $n = p$ and $n > p$. The implementation of the multiprefix operation is also considered. Some work carried out on $h$-relations is discussed ultimately.

2 Previous Work

Research has been done on circuit implementations to solve the parallel prefix problem. Parallel prefix circuits first appeared in the construction of circuits to add two binary numbers in 1963 [15]. Later in 1980, Ladner and Fischer [12] proposed a recursive construction for parallel prefix circuits. They noticed that it was straightforward to construct a circuit of minimum size, $n - 1$, but with a similar depth. On the other hand, the straight construction of a circuit with the minimum possible depth, $\lceil \log n \rceil$, would yield to a circuit of size $\Omega(n \log n)$. Their results described a family of restricted parallel prefix circuits $P_k(n)$, with $n$ inputs, depth at most $k + \lceil \log n \rceil$, and a size bounded by $4n$. Fich [8], in 1983 improves the work of Ladner and Fischer by giving new upper and lower bounds for the number of gates (size) of parallel prefix circuits with minimum depth, i.e. depth $\lceil \log n \rceil$, when the number of inputs is a power of two. Later in 1986, Snir [20] considered the general problem of tradeoffs between depth (delay) and size (number of nodes) for parallel prefix circuits. He derived a lower bound by showing that the depth $t$ and the size $s$ of parallel prefix circuits are related by the inequality $t + s \geq 2n - 2$. Hence a parallel prefix is said to be $(t, s)$-optimal if $t + s = 2n - 2$. He gave an algorithm to design $(t, s)$-optimal parallel prefix circuits for $t$ over the range $2 \log n - 2 \leq t \leq n - 1$. Lakshmivarahan et al. [13] improved Snir work by giving an algorithm for the design of $(t, s)$-optimal circuits with $t$ in the range $\log n \leq t \leq 2 \log n - 3$.

A parallel implementation of the prefix computation on the perfect shuffle network was proposed by Stone [21] in 1971 for polynomial evaluation, which required $O(\log n)$ time using $O(n)$ processors. He improved the algorithm by requiring the same complexity in time but using $O(n/\log n)$ processors and showed its application to solving tridiagonal systems of equations [22]. In 1982 Greenberg et al. [10] considered the problem of computing the terms of an $m$th-order linear recurrence system, reducing the problem to a parallel prefix computation. Blelloch [4] considers the parallel prefix computation (which is called scan operation on the Scan model) as a basic building block for parallel algorithms and suggested that some scan operations be included on the PRAM model as primitives. Schwartz [19] was the first to suggest a segmented version of the scan operation, also known as the multiprefix operation. In fact, the scan operation is a special case of the most general multiprefix operation, which is one of the instruction primitives of the Fluent abstract machine [17].

3 The Computation Model

The model of computation used in this paper is the Optical Communication Parallel (OCP) computer [9]. A $p$-node OCP consists of $p$ processors and $p$ memory modules; a memory module can be thought as being the processor's local memory. The OCP is characterized by the following properties:
Processors communicate with each other by exchanging messages,

A processor sends a message to a chosen processor by directing to it a light beam,

If two or more processors intend to send a message to the same processor, no transmission is successful, since light beams cannot interact, and retransmission must take place,

A successful transmission is acknowledged by sending back a confirmation message,

A processor knows in constant time if the transmission of its message has been successful.

For a survey of optical computing see [3, 6, 7, 18, 26] and references therein. One of the major drawbacks in current optical technology is the (very long) time needed to redirect the light beam, making actual optical methods less competitive than systems based on electronic devices.

The OCP model can be considered as being equivalent to an EREW PRAM, with the former being more restrictive than the latter. In both models, simultaneous read/write access by two or more processors is not allowed. The difference lies on the object being accessed: in a EREW PRAM, access to a memory address in the shared memory is exclusive, while in the OCP model, the object to be accessed is represented by a memory module composed of a given number of memory addresses: simultaneous access by two or more processors to the same memory module is not allowed.

4 The Parallel Prefix Operation

The algorithm presented below has been derived from the cascade sum method to solve a particular case of a general first-order recurrence (see recurrences in [11]). The algorithm computes the partial sums \( S_i = x_0 \oplus x_1 \oplus \ldots \oplus x_i \) for \( 0 \leq i \leq n-1 \) assuming one input per processor. Initially, \( x_i \) is loaded on processor \( p_i \). Each processor \( p_i \) will hold a local variable \( S_i \) which will be the data to be transferred in every step of the algorithm. In the code given below, it is readily verified that in any step no more than one processor attempts to communicate to the same processor, making the algorithm well suited to work on the OCP model. The algorithm terminates after \( \log n \) parallel steps. Fig. 1 pictures the execution steps of the parallel prefix computation for \( n = 8 \).

Algorithm ParallelPrefix

\[
\text{for } i = 0 \text{ to } n - 1 \text{ do in parallel } \\
\quad S_i = x_i \\
\text{endfor} \\
\text{for } j = 0 \text{ to } \log n - 1 \text{ do } \\
\quad \text{for } i = 0 \text{ to } n - 1 - 2^j \text{ do in parallel } \\
\quad \quad \text{Send } (p_{i+2^j}, S_i) \\
\quad \text{endfor} \\
\text{endfor}
\]

The function \( \text{Send } (p_i, \text{val}) \) sends to processor \( p_i \) the value \( \text{val} \). This communication function implies a synchronous external communication between processor \( p_i \) and the processor issuing the call. In any step, let say that the value just received by \( p_i \) is stored in the local variable \( \text{Sum} \), then processor \( p_i \) performs the operation \( S_i = \text{Sum} \oplus S_i \).
Figure 1: Parallel prefix computation based on the cyclic-reduction method.

5 Some Applications

5.1 The Multiprefix Operation

The multiprefix operation can be thought as a set of prefix operations executed simultaneously. Each prefix operation is performed among processors referring to a same key value. Let $G_{key} = \{p_1, p_2, \ldots, p_k\}$ be the set of processors referring to the same key with $p_1 < p_2 < \ldots < p_k$. Henceforth, $G_{key}$ will be called the group key. Let say that processor $p_i \in G_{key}$ executes a multiprefix operation by issuing the call $MP(key, x_i)$, then at the end of the operation, $p_i$ will hold the value $x_1 \oplus x_2 \oplus \ldots \oplus x_i$ for $1 \leq i \leq k$. For example, if at time $T$ the following multiprefix calls are performed:

\[p_0 : MP(3, 1) \quad p_1 : MP(2, 40) \quad p_2 : MP(1, 15) \quad p_3 : MP(3, 5)\]

\[p_4 : MP(3, 10) \quad p_5 : MP(1, 50) \quad p_6 : MP(1, 20) \quad p_7 : MP(2, 12)\]

then at time $T + 1$, the value in each processor will be:

\[p_0 : 1 \quad p_1 : 40 \quad p_2 : 15 \quad p_3 : 6\]

\[p_4 : 16 \quad p_5 : 65 \quad p_6 : 85 \quad p_7 : 52\]

In our proposition, the multiprefix operation is performed in two stages. The first one is devoted to the composition of groups; forming the group $G_k = \{p_1, p_2, \ldots, p_q\}$ consists in connecting (logically) $p_i$ with $p_{i+1}$ for $1 \leq i < q$. During the second stage, the multiprefix operation itself is performed: a prefix operation is simultaneously executed within each $G_k$. We assume that we have $p$ processors and $n$ keys, a key per processor. We also assume that the value of each key is in the range $0, \ldots, R-1$. Using a similar technique as in a radix sort (see section 5.2), the binary representation of each key is decomposed into a sequence of $\log R / \log m$ blocks of length $\log m$ each. The algorithm presented below has a time complexity of
Algorithm MultiPrefix

First stage: forming groups

1. Repeat the subrouting Group (subkey) below \( \log R / \log m \) times. In each step, a new key block is passed as argument. The subkey values are in the range \( 0, \ldots, m - 1 \).

Subroutine Group (subkey)

(a) Each processor creates \( m \) buckets.
(b) The processor number (pid) is inserted in the bucket indexed by subkey. The value NULL is inserted in the other \( m - 1 \) buckets. This takes \( O(m) \) time.

\[
\text{bucket [subkey]} = \text{pid}; \\
\text{target} = \text{NULL};
\]

At the end of the subroutine, the target variable in processor \( p_i \) will hold the processor number, if any, to which \( p_i \) will connect during the next step.
(c) for \( i = 0 \) to \( m - 1 \) do

Perform a parallel prefix operation using the value bucket[\( i \)] as operand. \( pn \) is the value just received. The \( \oplus \)-operator of the prefix operation is defined by the following C code:

\[
\text{if } (pn) \{
\text{if } (\text{bucket } [i])
\text{if } (!\text{target } \&\& \text{subkey } == i)
\text{target} = pn;
\text{else} \text{bucket } [i] = pn;
\}
\]

This step takes \( O(m \log p) \) time.

Second stage: performing the multiprefix operation

1. Perform simultaneously a prefix operation within each \( G_k \).

Fig. 2 shows an example of the multiprefix operation for \( p = 8 \) and \( n = 8 \). The keys are in the range \( 0, \ldots, 15 \). Let \( m \) be equal to 4. The figure illustrates only the first stage of the main algorithm. During the first step, \( m \) prefix operations are performed over the \( p \) processors. In the following steps, the prefix operations are performed independently within each \( G_k \) formed in the previous step. For instance, the second step in the figure is performed simultaneously within \( G_0 \) and \( G_2 \), where \( G_0 = \{1, 2, 4, 5, 6\} \) and \( G_2 = \{0, 3, 7\} \). A global parallel prefix can be obtained from a multiprefix operation in which all the keys are equal.
5.2 Radix Sort \( (n = p) \)

In this section we show how a radix sort can be easily implemented using the prefix and multiprefix operations presented so far. We assume that \( n \) is the number of items to be sorted using \( p \) processors; there is one item per processor \((n = p)\). A radix sort considers that the value of the items are in the range \( 0, \ldots, R - 1 \). The binary representation of each item value is decomposed into a sequence of \( \log R / \log m \) blocks of length \( \log m \) each, where \( m \) is the size of the radix. The algorithm we present below has a time complexity of

\[
O\left(\frac{\log R}{\log m} \log p \right),
\]

as for the multiprefix operation. At present, the following definition will be helpful.

**Definition.** If \( G_k = \{p_1, p_2, \ldots, p_l\} \) where \( p_1 < p_2 < p_l \), let \( H(G_k) \), the head of group \( k \), be the smallest processor number inside \( G_k \), i.e. \( H(G_k) = p_1 \). Likewise, let \( Q(G_k) \), the queue of group \( k \), be the greatest processor number inside \( G_k \), i.e. \( Q(G_k) = p_l \).

Our proposition consists of two stages. The first one is devoted to compose the \( G_k \)'s (using the same technique as in the multiprefix operation). The second stage will relate the \( G_k \)'s in a predetermined

---

**Figure 2:** The first stage of the multiprefix operation: forming groups.
ordered; that is, the $G_k$'s will be sorted in the following manner: $H(G_k)$ will be connected to $Q(G_{k'})$ if $k'$ is the smallest key value greater than $k$. The second stage is performed in two passes. The first one performs $m$ prefix operations in one direction (say from left to right). A second pass, performing $m$ prefix operations in the opposite direction (say right to left), is needed since some Head-Queue connections during the first pass are either not possible or wrong. The algorithm $GroupSort$ (subkey) below is repeated $\log R/\log m$ times. In each step, a new item block is passed as the function argument. No data is permuted at any moment so the resulting connection pattern gives the desired ordered sequence. Anyway, data permutation (i.e. 1-relation) during intermediate steps can be done in time $O(1)$, realizing the adequate modifications to the algorithm.

**Algorithm $GroupSort$ (subkey)**

**First stage: forming groups**

1. Initialize the following variables:
   
   $Head = FALSE$;  
   $next\_group = 0$;

2. Form the $G_k$'s using the first stage of the multiprefix operation $Group$ (subkey);

3. Mark the head elements ($H(G_k)$)  
   
   $if (!target)$  
   $Head = TRUE$;  

**Second stage: sorting groups**

Prefix operations during the first (second) pass is performed from left to right (right to left) over $p$ processors. The value in $bucket[i]$ is used as operand of the prefix operation and $pn$ is the received value.

**First pass**

1. Reinitialize buckets (see multiprefix first stage).  
2. for $i = 0$ to $m - 1$ do  
   
   (a) Perform a prefix operation using the following $\oplus$-operator:  
   
   if ($pn \&\& !bucket[i]$) $bucket[i] = pn$;  

   (b) Perform Head-Queue connections  
   
   $if (Head \&\& i > subkey \&\& !target) \{$  
   
   $target = pn;$  
   
   $next\_group = i;$  

   $\}$

**Second pass**

1. Reinitialize buckets (see multiprefix first stage).  
2. for $i = 0$ to $m - 1$ do  
   
   (a) Perform a prefix operation using the following $\oplus$-operator:  
   
   $if (pn) bucket[i] = pn;$  

1159
(b) Complete (or correct) Head-Queue connections

\[
\text{if } (!\text{target} \mid i \leq \text{next
dgroup}) \\
\text{target} = pn; \\
\text{Head} = 0;
\]

Figs. 3 and 4 show an example of the radix sort algorithm for \( p = 8 \) and \( n = 8 \). Clearly, the first stage of the algorithm and the first pass in the second stage can be realized by performing only a group of \( m \) prefix operations (instead of two), modifying slightly the algorithm. Therefore, the number of parallel steps required by the radix sort for \( n = p \) is \((\log R/\log m)(2m \log p)\).

<table>
<thead>
<tr>
<th>Initial Data Loading:</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

First Step

First stage: forming groups

Initial connection pattern and subkey values

\[
\begin{align*}
10 & \rightarrow 00 \\
0 & \rightarrow 1 \\
0 & \rightarrow 2 \\
10 & \rightarrow 3 \\
0 & \rightarrow 4 \\
0 & \rightarrow 5 \\
0 & \rightarrow 6 \\
10 & \rightarrow 7
\end{align*}
\]

After first stage

Second stage: sorting groups

First and second passes

Figure 3: Radix sort for \( p = 8, n = 8 \). First step.

5.3 Radix Sort \((n > p)\)

In this section we present an implementation of the radix sort for the case \( n > p \) having \( n/p \) items per processor. The same problem conditions given in section 5.2 are used. Here again we aim to illustrate the use of the prefix operation presented previously. The function \textit{BucketSort (subkey)} given below is repeated \( \log R/\log m \) times; in each step, a new key block is passed as the function argument. Before giving the algorithm, we show how the global rank of an item is computed.

Computing the rank of an item

Each processor starts by creating \( m \) buckets, one for each of the subkey (the item in the current step) values in the range \( 0, \ldots, m - 1 \). Next, the \( n/p \) local items are placed in the appropriate bucket. Now, let \( L_i^j \) denote the list of items in bucket \( j \) at processor \( i \), then the sorted sequence is given by:

1160
Second Step

First stage: forming groups
Initial connection pattern and subkey values

\[
\begin{array}{cccccccc}
10 & 10 & 10 & 01 & 10 & 10 & 01 & 10 \\
0 & 3 & 7 & 1 & 2 & 4 & 5 & 6
\end{array}
\]

After first stage

\[
\begin{array}{cccccccc}
10 & 10 & 10 & 01 & 10 & 10 & 01 & 10 \\
0 & 3 & 7 & 1 & 2 & 4 & 5 & 6
\end{array}
\]

Second stage: sorting groups
First and second passes

\[
\begin{array}{cccccccc}
10 & 10 & 10 & 01 & 10 & 10 & 01 & 10 \\
0 & 3 & 7 & 1 & 2 & 4 & 5 & 6
\end{array}
\]

Final connection pattern

\[
\begin{array}{cccccccc}
10 & 10 & 10 & 8 & 8 & 8 & 4 & 4 \\
0 & 3 & 7 & 2 & 4 & 6 & 1 & 5
\end{array}
\]

Figure 4: Radix sort for \( p = 8, n = 8 \). Second step.

\[
L_0^0, L_1^0, \ldots, L_{p-1}^0, L_0^1, \ldots, L_{p-1}^1, L_0^{m-1}, \ldots, L_{p-1}^{m-1}.
\]

Let \(|L_j^i|\) be the length (number of items) of \(L_j^i\). By means of a prefix computation, processor \(i\) can compute the following partial sum:

\[
S_j^i = \sum_{q=0}^{i} |L_q^j|,
\]

for \(j = 0, \ldots, m - 1\). It can readily be verified that the global rank \(r\) of the \(k\)th item in bucket \(j\) at processor \(i\) is given by the following equation:

\[
r = S_{p-1}^0 + S_{p-1}^1 + \ldots + S_{p-1}^{j-1} + S_{i-1}^j + k.
\]

Finally, an item with rank \(r\) should be placed in processor \(\lfloor \frac{(r-1)p}{n} \rfloor\).

Algorithm BucketSort (subkey)

1. Each processor creates \(m\) buckets.
2. Place local items in the appropriate bucket. This takes \(O(n/p)\) time.
3. for \(j = 0\) to \(m - 1\) do
Compute $S_i^j$ by means of a prefix computation.

The value $S_{p-1}^j$ computed by processor $p - 1$ is broadcast to all other processors. A broadcast operation is performed using a prefix computation with the $\oplus$-operator modified accordingly. This step requires $m$ prefix and $m$ broadcast operations for a time complexity of $O(m \log p)$.

4. Compute the global rank and the final destination for every item. This takes $O(n/p)$.

5. Send items to the appropriate processor. This can be achieved in $O(n/p)$ steps by performing a conflict-free permutation (1-relation) in every step. See below.

6. Sort locally the new items at each processor. This can be done in $O(n/p)$ using a serial version of BucketSort, since the rank of an item in processor $i$ is in the range $[i \times \frac{n}{p}, i \times \frac{n}{p} + \frac{n}{p} - 1].$

The total time complexity of the BucketSort algorithm is

$$O\left(\frac{n}{p} + m \log p\right).$$

Our radix sort algorithm to sort $n$ items in the range $0, \ldots, R - 1$ performs $\log R / \log m$ iterations of the BucketSort algorithm, for a final time complexity of

$$O \left( \frac{\log R}{\log m} \left( \frac{n}{p} + m \log p \right) \right).$$

$h$-Relations

The radix sort algorithm presented above requires realizing arbitrary $h$-relations as an intermediate step. In an $h$-relation each processor is the source as well as the destination of at most $h$ messages. In the case of the radix sort, each processor holds $n/p$ items. Each of them is granted a destination processor number in the range $0, \ldots, p - 1$. Manzini in [15], gives an implementation of the radix sort on the hypercube. He proposed a MultiRoute algorithm to realize arbitrary $n/p$-relations in time $O((n/p) \log p + p \log^2 p)$. His algorithm works in two phases. The first one uses the MultiBalance algorithm given in [17] to load balance each group $g$ over the $p$ processors. In this case, the value of a group $g$ is in the range $0, \ldots, p - 1$. An item belongs to group $g$ if its destination processor number is equal to $g$. The second phase simply routes the balanced items using a predetermined policy: processor $p_i$ starts routing items from group $i$.

Geréb-Graus and Tsantilas [9, 10] proposed a simple and practical randomized algorithm on a $p$-processor OCP which performs in $\Theta(h + \log p \log \log p)$ steps. Anderson and Miller [2] and Valiant [25] gave an $\Theta(h + \log p)$ algorithm but expensive in terms of constant factors as well as in complexity from a practical point of view.

Recently, a randomized solution to the same problem has been proposed by Adamo in [1] in the context of Clos networks. His algorithm does not perform multiple load balancing, and it can be completed in $O(n/p)$ parallel communication steps with overwhelming probability (see his paper for a detailed description of the algorithm).

Another alternative to the solution of the problem, which performs in $O(n/p)$ parallel time on a $p$-processor OCP and for an specified range of $n$, consists in decomposing the $h$-relation into $h$ constant time permutations. The algorithm is given in [24] and has two main stages: the first one
is devoted to schedule the data movements so as to avoid communication conflicts between any two processors. The second stage performs $h$ conflict-free permutations. It is assumed that $h$ is at least equal to $p$, the number of processors (i.e. $n/p \geq p$).

Acknowledgment

The author is very grateful to all members of the D.I.C., C.C. for their precious help and support.

6 Conclusions

We have presented an algorithm to perform the parallel prefix operation, which is well suited to work on the OCP model of computation. We mentioned that the prefix operation is one of the parallel primitives of the Scan Model and in its more general form, the multiprefix operation, it is one of the main parallel primitives on which Ranade's Fluent abstract machine relies. We believe that in a similar way, the OCP model can be provided with some basic parallel primitives, namely a prefix (and multiprefix) operation and an operation to realize arbitrary $h$-relations. We showed in this paper that these operations can be efficiently implemented on the OCP model. We illustrated the use of the primitives by giving an $O((\log R/\log m)((n/p) + m \log p))$ radix sort algorithm.

References


