Parallel Prefix Computation with Few Processors [*]

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

Çetin Kaya Koç
Department of Electrical Engineering
University of Houston
Houston, TX 77204

Abstract

We present a parallel prefix algorithm which uses $\frac{2(p+1)}{p(p+1)+2}n-1$ arithmetic and $\frac{p(p-1)}{p(p+1)+2}n+\frac{1}{2}p(p-1)$ routing steps to compute the prefixes of n elements on a distributed-memory multiprocessor with $p < n$ nodes. The algorithm is compared with the distributed-memory implementation of the parallel prefix algorithm proposed by Kruskal, Rudolph, and Snir. We show that there is a trade-off between the two algorithms in terms of the number of processors, and the parameter $\tau = \tau_R/\tau_A$, which is the ratio of the time required to transfer an operand to the time required to perform the operation of the prefix problem. The new algorithm is shown to be more efficient when n is large and $p^2(p-1) \leq \frac{4}{\tau}$.

CR Categories: F.1.2, F.2, G.4.

1980 Mathematics Subject Classification (1985 Revision): 68Q10, 68Q25.

Key Words: Parallel algorithm, parallel prefix, distributed-memory multiprocessor, algorithm efficiency.

1 Introduction

Given an ordered n-tuple $(x_1, x_2, \ldots, x_n)$ of elements of a set closed under an associative binary operation $*$, the *prefix problem* is the computation of the partial products $y_i = x_1 * x_2 * \cdots * x_i$ for $1 \leq i \leq n$. The prefix problem arises in various settings including circuit design problems where $*$ is a simple boolean operation, and numerical problems where $*$ can be as complicated as floating-point matrix multiplication. For example, parallel algorithms for computing the Newton and Hermite interpolating polynomials make use of parallel prefix algorithms where the x_i's are floating-point numbers and $*$ is a floating-point addition or multiplication [5]. Solution of kth order linear recurrences can be obtained by a parallel prefix algorithm where $*$ is $k \times k$ matrix multiplication [9, 8, 7]. Tridiagonal systems can be solved with Stone's recursive doubling algorithm by computing the prefixes of 2×2 matrices whose entries are floating-point numbers [16]. More generally, the recursive doubling algorithm can be used

[*]This work is supported in part by the US Army Research Office Grant No. DAAL03–91–G–0106.

to solve a banded linear system with bandwidth $W = 2k + 1$ by computing the prefix product of $k \times k$ matrices.

Parallel prefix circuits have applications in the design of optimal-area adders [2] and the simulation of sequential circuits by combinational circuits [11]. The reader may refer to Fich's paper [6] for a review of the literature on parallel prefix circuits and further applications. In a parallel prefix circuit, the concern is to reduce the depth (D) and the size (S) of the circuit. Upper and lower bounds on the size for a restricted family of circuits with minimum or near minimum depth appears in [6]. Snir proved the interesting lower bound that $S + D \geq 2n - 2$ [15].

Prefixes of n elements can be computed trivially in $n - 1$ steps sequentially where at each step a single $*$ operation is performed. There are several parallel prefix algorithms [11, 2, 6, 10, 15, 12], given either in the arithmetic circuit or PRAM model of parallel computation. Asynchronous algorithms [13] and implementation on various ensemble architectures [14, 3, 4, 1] have also been considered.

In this paper, we focus on the performance of parallel prefix algorithms on distributed-memory multiprocessors. We assume that we are given $p < n$ processors with a routing mechanism to send an operand from one processor to any other processor. An *arithmetic step* (τ_A) is defined as the time required to perform a $*$ operation by a single processor, and a *routing step* (τ_R) as the time required to transfer an operand from one processor to another. It is also assumed that the processors are identical and the architecture is completely connected, i.e., τ_A and τ_R are constants.

First in §2 we describe the distributed-memory multiprocessor implementation of a parallel prefix algorithm given in [10]. We then present a new suboptimal parallel prefix algorithm which achieves higher efficiency for small values of p, and when $\tau_R < \tau_A$. The efficiency of these two algorithms as a function of the number of processors and the parameter $\tau = \tau_R/\tau_A$ is analyzed in §4 together with a comparison of their arithmetic complexities to the lower bound obtained by Snir in [15].

2 The KRS Parallel Prefix Algorithm

First we consider the distributed-memory implementation of the parallel prefix algorithm (henceforth named the KRS algorithm) given by Kruskal, Rudolph, and Snir. This algorithm is designed using the EREW PRAM computation model in [10]. When $p = n$, the KRS algorithm reduces to Stone's recursive doubling algorithm. It follows that, on a distributed memory multiprocessor with n processors, prefixes of n elements can be computed in $\log n$ arithmetic and $\log n$ routing steps.

When $p < n$ processors $P_1, P_2, \ldots, P_p$ are available, we first partition the list $(x_1, x_2, \ldots, x_n)$ into p sublists, each containing $m = n/p$ contiguous elements. The sequential prefix algorithm is then applied within each sublist. Thus, processor P_j computes the prefixes of $(x_{(j-1)m+1}, x_{(j-1)m+2}, \ldots, x_{(j-1)m+m})$ for $j = 1, 2, \ldots, p$ sequentially using the locally available data. This step takes $m - 1$ arithmetic steps. Let

$$y_{jm} = x_{(j-1)m+1} * x_{(j-1)m+2} * \cdots * x_{(j-1)m+m}$$

for $j = 1, 2, \ldots, p$. We apply the recursive doubling algorithm to compute the prefixes of the list $(y_m, y_{2m}, \ldots, y_{pm})$ using all p processors. This step of the KRS algorithm takes $\log p$ arithmetic and $\log p$ routing steps.

Now we have the term

$$y_m * y_{2m} * \cdots * y_{jm}$$

at processor P_j for $j = 1, 2, \ldots, p$. This quantity, replacing y_{jm} at P_j, is sent from processor P_j to processor P_{j+1} for $j = 1, 2, \ldots, p - 1$ in a single parallel routing operation. The received item is then multiplied with every prefix term in processor P_{j+1} except the last one. This step also requires $m-1$ parallel arithmetic steps. Summing the contribution of arithmetic and routing steps, we have

Theorem 1 *The KRS algorithm computes the prefixes of n elements on a distributed-memory multiprocessor with $p \leq n$ nodes using $A_p(n) = 2\frac{n}{p} + \log p - 2$ arithmetic and $R_p(n) = \log p + 1$ routing steps.*

The details for the distributed-memory implementation of the KRS algorithm and its implementation on the hypercube multiprocessor can be found in [4]. The KRS algorithm achieves linear speedup for $p < n$. Furthermore, the number of routing operations required is very small; $R_p(n) = \log p + 1$, which is not a function of the input size. However, for small values of p the KRS algorithm is not efficient in terms of its arithmetic complexity. For example, when $p = 2$ we have

$$A_2(n) = 2\frac{n}{2} + \log 2 - 2 = n - 1$$

which is the number of operations required to perform this computation sequentially. Thus in this case having 2 processors instead of 1 provides no reduction in the execution time.

The optimal value of $A_p(n)$ for 2 processors is $A_2^{opt}(n) = \frac{2n-2}{3}$ as given by Snir [15]. Furthermore, Snir has provided parallel prefix circuits with depth $D = \frac{2n}{w+1}$ where w is the *width* of the circuit, i.e., the number of processors required to execute the algorithm in D parallel arithmetic steps [15]. Here the PRAM computation model is used where interprocessor communication is not an issue. Thus Snir's parallel prefix algorithm is optimal (up to an additive constant) if one considers only the number of arithmetic steps.

In the next section we present a new parallel prefix algorithm which is suboptimal in terms of the number of arithmetic steps, but more efficient than the distributed-memory implementation of the KRS algorithm for small values of p and τ.

3 A New Parallel Prefix Algorithm

We propose the following two-phase algorithm for computing the prefixes of $(x_1, x_2, \ldots, x_n)$ on a distributed-memory multiprocessor with $p < n$ processors. Let α_p be a rational number with $0 < \alpha_p < 1$, to be determined later. As a function of α_p, the first phase of the algorithm is to partition $(x_1, x_2, \ldots, x_n)$ into two sublists $L_1 = (x_1, x_2, \ldots, x_{\alpha_p n})$ and $L_2 = (x_{\alpha_p n+1}, x_{\alpha_p n+2}, \ldots, x_n)$ of lengths $\alpha_p n$ and $(1-\alpha_p)n$, respectively. Then we assign $p-1$ processors $P_1, P_2, \ldots, P_{p-1}$ for the computation of the prefixes of L_1, and a single processor P_p to the computation of the prefixes of L_2. In the second phase, all of the prefixes of the given list are computed by combining the partial products available. The further partitioning of the data in L_1 is done recursively by assigning the first $\alpha_{p-1}\alpha_p n$ elements of L_1 to the first $p-2$ processors $P_1, P_2, \ldots, P_{p-2}$ and the remaining $(1-\alpha_{p-1})\alpha_p n$ elements to processor P_{p-1}, and so on. Our rule in picking the numbers α_p in this partitioning scheme is as follows:

> *Choose α_p in such a way that the number of parallel arithmetic steps performed by $P_1, \ldots, P_{p-1}$ to compute the prefixes of the list L_1 is the same as the number of arithmetic steps performed by processor P_p to sequentially compute the prefixes of L_2.*

Therefore in the first phase of the algorithm, the prefixes of the $\alpha_p n$ terms in L_1 are computed by $p-1$ processors while the last processor computes the prefixes of the $(1-\alpha_p)n$ elements in L_2. In the second phase of the algorithm, we essentially perform a scatter operation to equally distribute all the computed terms among p processors to finish the remaining work.

Note that during the computation of the prefixes of L_1 there is some time spent for routing operations among $P_1, P_2, \ldots, P_{p-1}$. By our choice of α_p, the idle time experienced by processor P_p is exactly equal to the time spent by $P_1, P_2, \ldots, P_{p-1}$ for these routing operations.

In the following analysis we will ignore the time spent for the initial loading of the data and the final unloading of the prefix terms computed. The prefix terms may have been scattered among the processors, i.e., since we do not require the prefixes of the terms in list L_j to be computed by processor P_j, these quantities may not be found in processor L_j after the execution of the algorithm by all processors. However, it turns out that the longest prefix term of list L_j will always be computed by and thus found in processor P_j.

In order to determine the fractions α_p explicitly for $p = 2, 3, \ldots$, we will first take a closer look at the boundary cases $p = 2$ and $p = 3$.

Case $p = 2$ **:** Here we assign the first $\alpha_2 n$ elements (L_1) to processor P_1 and the remaining $(1 - \alpha_2)n$ elements (L_2) to processor P_2. The processors independently perform sequential prefix computation with their local data. According to the stated rule, we determine α_2 so that P_1 and P_2 perform an equal number of arithmetic operations. Since $r - 1$ operations are required to compute the prefixes of r elements sequentially, this trivially implies that

$$\alpha_2 n - 1 = (1 - \alpha_2)n - 1.$$

Thus we pick $\alpha_2 = \frac{1}{2}$. After the sequential prefix is performed, we have the prefixes of the elements of L_1 in processor P_1 and the prefixes of the elements of L_2 in processor P_2. We then transfer the term $x_1 * x_2 * \cdots * x_{\alpha_2 n}$ from processor P_1 to processor P_2. After this step, the first half of the prefix terms computed in P_2, i.e.,

$$y_{\alpha_2 n+1}, y_{\alpha_2 n+2}, \ldots, y_{(\alpha_2 + \frac{1-\alpha_2}{2})n} \ ,$$

are forwarded to processor P_1. Now each processor works on its own data and the data just received to compute the remaining prefixes by combining cross products. Notice that at the end of the execution the longest prefix product terms of lists L_1 and L_2 will be in processors P_1 and P_2, respectively. The total number of parallel arithmetic steps required for the algorithm is found to be

$$A_2(n) = \alpha_2 n - 1 + \frac{(1 - \alpha_2)n}{2} = \frac{3}{4}n - 1 \ .$$

The number of routing steps required is

$$R_2(n) = \frac{(1 - \alpha_2)n}{2} + 1 = \frac{1}{4}n + 1 \ .$$

Case $p = 3$ **:** Here we assign the initial $\alpha_3 n$ elements of the input list to the first two processors P_1 and P_2, and the remaining $(1 - \alpha_3)n$ to P_3. P_1 and P_2 execute the parallel prefix algorithm with $\alpha_3 n$ elements using the algorithm for $p = 2$ above, while processor P_3 performs a sequential prefix algorithm on $(1 - \alpha_3)n$ elements. Thus, by our selection of α_3, we must have

$$A_2(\alpha_3 n) = (1 - \alpha_3)n - 1 \ .$$

Thus

$$\frac{3}{4}(\alpha_3 n) - 1 = (1 - \alpha_3)n - 1 \ ,$$

which implies that we should pick $\alpha_3 = \frac{4}{7}$. In the second phase, as before, all three processors equally share the work to compute the remaining prefixes. The total number of parallel arithmetic steps required for the algorithm is easily computed to be

$$A_3(n) = \frac{3}{4}\alpha_3 n - 1 + \frac{(1-\alpha_3)n}{3} = \frac{4}{7}n - 1 \ .$$

To determine the number of routing steps, we note that in addition to the number of routing steps performed by the first two processors internally, we need to equally distribute $(1-\alpha_3)n$ elements among three processors, and also to send the last term (the longest prefix product of list L_2) computed by processor P_2 to processors P_1 and P_3. The first task is achieved by sending $\frac{(1-\alpha_3)n}{3}$ terms from processor P_3 to processor P_1, and an equal number of terms from processor P_3 to processor P_2. This requires $\frac{2(1-\alpha_3)n}{3}$ communication steps. Thus, the the total number of routing steps is found to be

$$R_3(n) = R_2(\alpha_3 n) + \frac{2(1-\alpha_3)n}{3} + 2 = \frac{1}{4}\frac{4}{7}n + 1 + \frac{2}{7}n + 2 = \frac{3}{7}n + 3 \ .$$

Note that $A_2(n) + R_2(n) = n$ and $A_3(n) + R_3(n) = n + 2$. In general, one can show that $A_p(n) + R_p(n) = n + \frac{1}{2}p(p-1) - 1$. More precisely, we have

Theorem 2 *The above algorithm computes the prefixes of n elements on a distributed-memory multiprocessor with $p < n$ nodes using $A_p(n) = \frac{2(p+1)}{p(p+1)+2}n - 1$ parallel arithmetic and $R_p(n) = \frac{p(p-1)}{p(p+1)+2}n + \frac{1}{2}p(p-1)$ routing steps with $\alpha_p = \frac{p(p-1)+2}{p(p+1)+2}$.*

Proof The partitioning for the algorithm is depicted in Figure 1. According to our rule the number of parallel arithmetic steps performed by processors $P_1, P_2, \ldots, P_{p-1}$ must be equal to the number of arithmetic steps performed by the last processor P_p. Thus, to find the total number of arithmetic steps required, we add the number of arithmetic operations performed by processor P_p (phase one) to the number of arithmetic steps required by all p processors to compute the prefixes of the elements indexed from $\alpha_p n + 1$ to n (phase two). This gives

$$A_p(n) = (1-\alpha_p)n - 1 + \frac{(1-\alpha_p)n}{p} = \frac{p+1}{p}(1-\alpha_p)n - 1 \ .$$

Let $A_p(n) = V_p n - 1$, i.e.,

$$V_p = \frac{p+1}{p}(1-\alpha_p) \ , \qquad (1)$$

then we have $V_{p-1}\alpha_p n - 1 = (1-\alpha_p)n - 1$, as can be seen from Figure 1. Thus

$$\alpha_p = \frac{1}{1+V_{p-1}} \ . \qquad (2)$$

It also follows from equation (1) that

$$V_p = \frac{p+1}{p}\frac{V_{p-1}}{1+V_{p-1}} \ . \qquad (3)$$

A recursion for $R_p(n)$ can be given as

$$R_p(n) = R_{p-1}(\alpha_p n) + \frac{p-1}{p}(1-\alpha_p)n + p - 1 \ , \qquad (4)$$

where the first term comes from the routing operations performed by $p-1$ processors and the second term is the number of routing operations required to send $(1-\alpha_p)n$ terms from the last processor to all the others. Finally $p-1$ routing operations are required to send the last prefix value from processor P_{p-1} to all the other processors. These operations are illustrated in Figure 2.

Since $V_2 = \frac{3}{4}$, $\alpha_2 = \frac{1}{2}$, and $R_2(n) = \frac{1}{4}n + 1$, using these initial values and induction on p in (3), (2), and (4), we obtain

$$V_p = \frac{2(p+1)}{p(p+1)+2} \ , \quad \alpha_p = \frac{p(p-1)+2}{p(p+1)+2} \ , \text{ and } \quad R_p(n) = \frac{p(p-1)}{p(p+1)+2}n + \frac{1}{2}p(p-1) \ ,$$

as claimed. Since $A_p(n) = V_p n - 1$, we also have

$$A_p = \frac{2(p+1)}{p(p+1)+2}n - 1 \ .$$

□

4 Efficiency Analysis

In Figure 3, $A_p(n)$ for the KRS algorithm and the new algorithm is shown for $n = 1024$ and $2 \le p \le 10$, together with the optimal number of arithmetic operations $A_2^{opt}(n) = \frac{2n-2}{p+1}$. We see that when p is small, the new algorithm is quite efficient in terms of arithmetic complexity but inefficient as far as the total number of routing operations is concerned. However there is a trade-off between the new algorithm and the KRS algorithm as a function of τ. As we mentioned in the introduction, the operation $*$ can be as simple as a boolean function, or as complex as multiplication of two $k \times k$ matrices with floating-point entries. The total execution time can be expressed as a function of the time required to perform a $*$ operation (τ_A) and the time required to perform a routing operation (τ_R). For the KRS algorithm, we obtain

$$T_{KRS} = \left[2\frac{n}{p} + \log p - 2\right]\tau_A + [\log p + 1]\,\tau_R \ . \tag{5}$$

For the new algorithm the total execution time is given as

$$T_{new} = \left[\frac{2(p+1)}{p(p+1)+2}n - 1\right]\tau_A + \left[\frac{p(p-1)}{p(p+1)+2}n + \frac{1}{2}p(p-1)\right]\tau_R \ . \tag{6}$$

The efficiency of these parallel algorithms with respect to the optimal sequential algorithm is computed as

$$E = \frac{(n-1)\tau_A}{p\left[A_p(n)\tau_A + R_p(n)\tau_R\right]} = \frac{n-1}{p\left[A_p(n) + R_p(n)\tau\right]}$$

which is a function of the ratio $\tau = \tau_R/\tau_A$. Figure 4 illustrates the efficiency of these two algorithms as τ ranges from 0 to 2 for $p = 2$ and $n = 1024$. Also in Figure 5, the efficiency is shown as a function of p for $\tau = 0.01$ and $n = 1024$. As it can be seen from Figure 4, for $p = 2$ the new algorithm is more efficient if $\tau < 1$, otherwise the KRS algorithm is preferred. Similarly, we observe from Figure 5 that if $\tau = 0.01$ then for $p > 8$ we have $E_{KRS} > E_{new}$ and for $p < 8$ we have $E_{KRS} < E_{new}$.

¿From (5) and (6), we derive that for n large,

$$\lim_{n\to\infty} E_{KRS} = \frac{1}{2} \ ,$$

which is independent of τ, and

$$\lim_{n\to\infty} E_{new} = \frac{p^2+p+2}{\tau p^3 + (2-\tau)p^2 + 2p} \ .$$

Thus, $E_{new} \geq E_{KRS}$ whenever

$$\frac{p^2+p+2}{\tau p^3 + (2-\tau)p^2 + 2p} \geq \frac{1}{2} \ .$$

Thus, the new algorithm is more efficient than the KRS algorithm for

$$p^2(p-1) \leq \frac{4}{\tau} \ .$$

Finally we note that most distributed-memory parallel computers available on the market are capable of overlapping communication with computation. Thus, a more careful analysis of the algorithm can also be made by overlapping some of the communication with computation performed by processors. Such analysis implies that the ratio τ_R/τ_A is effectively smaller than for the nonoverlapping case, which in turn means the new parallel prefix algorithm will obtain higher speedup.

References

[1] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[2] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31(3):260–264, March 1982.

[3] D. A. Carlson. Modified mesh-connected parallel computers. *IEEE Transactions on Computers*, 37(10):1315–1321, October 1988.

[4] Ö. Eğecioğlu, Ç. K. Koç, and A. J. Laub. A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors. *Journal of Computational and Applied Mathematics*, 27(1+2):95–108, 1989.

[5] Ö. Eğecioğlu, E. Gallopoulos, and Ç. K. Koç. Parallel Hermite interpolation: An algebraic approach. *Computing*, 42(4):291–307, 1989.

[6] F. E. Fich. New bounds for parallel prefix circuits. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 100–109, 1983.

[7] A. G. Greenberg, R. E. Ladner, M. Paterson, and Z. Galil. Efficient parallel algorithms for linear recurrence computation. *Information Processing Letters*, 15(1):31–35, 1982.

[8] L. Hyafil and H. T. Kung. The complexity of parallel evaluation of linear recurrences. *Journal of the ACM*, 24(3):513–521, July 1977.

[9] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, 22(8):786–792, August 1973.

[10] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, 34(10):965–968, October 1985.

[11] R. Ladner and M. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[12] S. Lakshmivarahan, C. Yang, and S. K. Dhall. On a new class of optimal parallel prefix circuits with $(\text{SIZE} + \text{DEPTH}) = 2n - 2$ and $\lceil \log n \rceil \leq \text{DEPTH} \leq (2\lceil \log n \rceil - 3)$. In *Proceedings of the International Conference on Parallel Processing*, pages 58–65, August 17–21 1987.

[13] B. D. Lubachevsky and A. G. Greenberg. Simple, efficient asynchronous parallel prefix algorithms. In *Proceedings of the International Conference on Parallel Processing*, pages 66–69, August 17–21 1987.

[14] H. Meijer and S. G. Akl. Optimal computation of prefix sums on a binary tree of processors. *International Journal of Parallel Programming*, 16(2):127–136, 1987.

[15] M. Snir. Depth-size trade-offs for parallel prefix computation. *Journal of Algorithms*, 7(2):185–201, 1986.

[16] H. S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20(1):27–38, January 1973.