

Simultaneous Parallel Reduction on SIMD Machines

Boleslaw K. Szymanski, William Maniatty
*Department of Computer Science, Rensselaer Polytechnic Institute
Troy, NY 12180-3590, USA*

and

Balaram Sinharoy
*IBM Large Scale Computing Division
Poughkeepsie, NY 12602, USA*

ABSTRACT

Proper distribution of operations among parallel processors in a large scientific computation executed on a distributed-memory machine can significantly reduce the total computation time. In this paper we consider an operation, called simultaneous parallel reduction, that is amenable to such optimization. Simultaneous reduction performs reduction operations in parallel, each operation reducing a one-dimensional consecutive section of a distributed array. Every element of the distributed array is used as an operand to many reductions executed concurrently over the overlapping array's sections. The simultaneous reduction is distinct from a more commonly considered parallel reduction which involves the parallel evaluation of a single reduction. Several algorithms achieving some of the lower bounds on simultaneous reduction complexity are presented under various assumptions about the properties of the binary operator of the reduction and of the communication cost of the target architectures.

Keywords: Parallel reduction, Simultaneous evaluation, Data parallelism, Algorithm complexity, Scalability

1. Introduction

Optimal placement of data and operations in large scientific computations executed on a distributed-memory machine can significantly reduce the total computation time. In this paper we consider the impact of operation placement on the efficiency of computations that arise in theoretical population biology [1]. The computation that motivated the study of simultaneous reduction algorithms models an isolated ecosystem by simulating the dynamics of multispecies interactions. The model represents the ecosystem as a cellular automata whose cells correspond to the habitat sites that are small enough to accommodate at most one organism. Each cell can be in one of q states in a state set $S = \{s_1, s_2, \dots, s_q\}$. The probability of a site's state transition from state s_i to state s_j , $P[s_i \rightarrow s_j]$ is a function, $f_{i,j}$, of the states of its neighboring sites. For simplicity, the relevant neighborhood is defined as a rectangle with the affected site located inside it (but not necessarily at the center). The size of this rectangle and the position of the affected site within the rectangle is defined by the ability of the species to propagate. Hence, these parameters depend on the physical characteristics of the species involved, the terrain of the habitat, and the existing air or water convection.

Let n_1, n_2 stand for the lengths of the sides of the rectangle and p_1, p_2 be the

offsets (measured from the lower left corner of the rectangle) of the desired position of the result. Let $s[i, j]$ be the current state at site (i, j) . If the habitat has $N_x \times N_y$ sites, then $s[1..N_x, 1..N_y]$ is a two-dimensional array. Let (m, q) be the coordinates of the site making a transition. The probability of transition from the current state to a new state $s' \in S$ is computed as:

$$Prob[s[m, q] \rightarrow s'] = \sum_{i=m-p_1}^{m+n_1-p_1} \sum_{j=q-p_2}^{q+n_2-p_2} f_{s[m, q], s'}(s[m, q], s[i, j])$$

The expression on the right is computed q times for each site in the habitat. Hence, its evaluation dominates the total execution time [2].

The above computation is an example of a reduction computed simultaneously over many overlapping continuous sections of an array. Other applications of such operations arise in computational biology (e.g., cluster recognition, fractal dimension computation [3]) and in physics (e.g., solvers for partial differential equations characterizing fluid flow).

Parallel Reduction is defined as the application of an associative binary operator across a list of elements [4,5]. Given an array A of size n and the binary operator \oplus , parallel reduction obtains the scalar result, r , as follows:

$$r = reduce(A, 1, n, \oplus) = \oplus_{i=1}^n A[i] = A[1] \oplus A[2] \oplus A[3] \oplus \dots \oplus A[n]$$

Simultaneous Parallel Reduction applies parallel reduction concurrently over many overlapping sections of an array. Given an array $A[1..N]$, a simultaneous parallel reduction for array sections of size n with an offset of the result r , $0 \leq r < n$, obtains an array $res[1..N]$ as follows:

$$\begin{aligned} res = SPR(A, n, N, r, \oplus) &= reduce_{i=1}^N(A, \max(1, i-r), \min(i+n-r-1, N), \oplus) \\ &= \prod_{i=1}^N \oplus_{j=\max(1, i-r)}^{\min(i+n-r-1, N)} A[j] \end{aligned} \quad (1)$$

where \prod denotes composition of the argument function. For fine-grain distributed-memory architecture, the array A will often be distributed over N processors (i.e., $A[i]$ will be stored on processor $p[i]$, for $1 \leq i \leq N$). In such a case, the result $res[i]$ is stored on processor $p[i]$. Equation (1) defines N parallel reductions that are evaluated simultaneously. In this paper, we assume that the array sections are of equal size, n , and that the results of reduction operations are placed at an arbitrary (but the same for all reductions) offset r within each section.

Reduction operations are frequently used in parallel algorithms [6] and binary operators in such reductions are required to be associative. Some examples are addition, multiplication, minimum, maximum, set union, and set intersection. Algorithms for many variations of parallel reduction have been studied extensively in the literature; however, the authors of this paper are not aware of any previous work on simultaneous parallel reduction that allows overlapping of the regions being reduced. A balanced binary tree implementation of parallel reduction on

mesh-connected architectures was presented in [7]. Another standard parallel reduction algorithm was introduced in [8] for tree topologies of arbitrary but bounded fan-in and for arbitrary tree depth. The sectioned prefix problem is a variant of parallel reduction that subdivides a single dimension of processors into non-overlapping contiguous regions of varying size. A multiple prefix algorithm that reduces non-contiguous, non-overlapping regions simultaneously was presented in [6].

In this paper, we consider efficient algorithms for simultaneous parallel reductions executed on an SIMD parallel computer. In the following, we assume that the binary operator used in reduction, \oplus , is associative. The presented algorithms were designed in such a way that the order of the binary operator is preserved, hence the operator may not be commutative. One algorithm presented in section 3.2 can be sped up if the operator \oplus has an inverse, denoted by \ominus . This condition is satisfied by such common operators as addition and multiplication (but not by minimum or maximum). We present here the simultaneous parallel reduction for the one-dimensional array sections (i.e., one-dimensional, continuous subarrays). The more general m -dimensional sections can be solved by applying the one-dimensional section algorithm along each dimension separately. The presented algorithms are parameterized by the size of the array sections being reduced, n , and the position (offset) of the final result, r , $0 \leq r < n$, measured from the leftmost element of the array section.

2. Complexity Metrics Used

SIMD parallel architectures vary widely in the methods and expenses involved in the inter-processor communication. Performance of two distinct algorithms with the same traditional complexity measures (which only tells about their asymptotic behavior) may perform quite differently on a particular architecture. To help determine the right algorithm for a given architecture, we provide the following complexity metrics, useful in expressing the traditional time-complexity for different architectures for the presented algorithms.

Operation count: c_o , the number of parallel arithmetic operations executed by a single processor.

Message count: c_{cm} , the number of parallel communication steps during computation. (During a parallel communication step, messages must be sent for the same distance and in the same direction relative to their sending processors.)

Hop count: $c_{ch}(i)$, the distance (i.e., the number of processors traversed on the route to the destination) that each message travels in the i -th parallel communication step. We will also use the total number of hops, $c_{ch} = \sum_{i=1}^{c_{cm}} c_{ch}(i)$.

Message size: c_{ms} , the number of words (i.e., single precision or integer numbers) sent in each message.

Depending on the considered architecture, the time complexity of an algorithm can be expressed as:

PRAM: as $T = c_o$.

SIMD with uniform communication latency: as $T = c_o + c_{cm} * latency$, where $latency$ represents the communication cost expressed in operation units. This formula is valid only if the message size c_{ms} is smaller than the packet size of the communication network: Otherwise, the number of packets sent has to be accounted for.

SIMD with hypercube interconnection: as $T = c_o + c_{cm} * latency + hop_cost * \sum_{i=1}^{c_{cm}} \log_2 c_{ch}(i)$, where hop_cost is the cost of traversing one link in the interconnection network [9]. Often hop_cost is a function of message size (e.g., MasPar xnet communication).

SIMD with mesh interconnection: as $T = c_o + c_{cm} * latency + c_{ch} * hop_cost$.

In a typical SIMD machine, like the CM-200 or the MasPar that was used in the implementation of the presented algorithms [10], the cost of a single xnet communication step^a, denoted c_c , can be expressed as the following function of the number of hops, c_{ch} , and the message size, c_{ms} :

$$c_c = latency + c_{ch} * single_hop_cost * c_{ms}$$

Hence, for such machines $hop_cost = single_hop_cost * c_{ms}$.

Since simultaneous reduction performs many parallel reductions over the array sections of size n , the lower bounds for the cost metrics are the same as for parallel reduction. For the fine-grain solution, i.e., when the number of processors is equal to N , the number of elements in array A , and each processor stores one element of A , these bounds are: $\lceil \log_2 n \rceil$ for the operation and message counts, $n - 1$ for the hop count, and 1 for the message size. In the following, we consider only algorithms for which each of the counts is no more than double of the corresponding lower bound.

3. Algorithms for Simultaneous Reduction

3.1. Recursive Combination Algorithm

In this section, we assume that the linear array of N processors initially stores a distributed array A which we want to overwrite with the resultant array res . Each processor $p[i]$ reduces n values of A held by the processors $p[\max(i - r, 1)], \dots, p[\min(i + n - r - 1, N)]$ and stores the result. We use the following notations in describing the algorithms.

^aThese machines also have so-called router communication for which the hop count is irrelevant because communication delay is independent of the distance of communicating processes. The router communication is slower than xnet communication and it is sensitive to the communication traffic, so we have not used router in the fine-grain implementation of the simultaneous parallel reduction. However, the MasPar and CM-200 computers with router communication exemplify SIMD architecture with uniform communication latency.

```

integer res := A[proc_id], j, dist := 1, lim; /* proc_id denotes the processor number */
if proc_id +  $2^{h(n)}$  ≤ n then lim := n -  $2^{h(n)}$  else lim := n endif;
for j = 0 to  $h(n)-1$  do
    if proc_id + dist ≤ lim then res := res ⊕ shift(res, -dist) endif;
    dist := dist + dist;
endfor;
if proc_id = 1 then res := res ⊕ shift(res, dist - n) endif;

```

Figure 1: Standard Parallel Reduction Algorithm with Leftmost Result Placement for $n > 1$.

$b(q, i)$ denotes the i -th bit (counting from the least to the most significant bits) of the integer q .

$h(q), l(q)$ denotes the bit position of the most (the least, respectively) significant non-zero bit of the argument q .

$\bar{z}(q)$ denotes the number of nonzero bits in q .

$shift(val, dist)$ shifts the argument val across $dist$ processors to the left (when $dist < 0$) or to the right (when $dist > 0$) from its original position (so val from processor $p[i]$ moves to processor $p[i + dist]$). Shift is done for all processors at the same time. For simplicity, we assume that the processor array is wrapped, i.e., processor $p[N]$ is followed by processor $p[1]$ ^b.

The basic parallel reduction algorithm is shown in Figure 1. It is assumed that the same program is executed on each processor and that array res is distributed over the processors in such a way that each processor stores a single element. An inspection of this algorithm reveals the following:

1. Simultaneous execution of this algorithm for overlapping sections would create subcomputations shared by many sections. Each subcomputation would reduce subsections with lengths equal to some power of two.
2. Each step in the individual parallel reduction combines the results of two subsections. Inside the for-loop the size of the left subsection is always a power of two. The right subsection can be of any size between one and the size of the left subsection. The size of the right subsection is defined by the relative position of the processor in the reduced section. The opposite is true for the last operation performed only by the first processor. However, in simultaneous parallel reduction, each processor represents the left subsection for some reductions and the right subsections for the others, which makes the algorithm in Figure 1 unusable (unless n is a power of 2, in which case the length of the left and right subsections are always equal).

^bIn the MPL language of the MasPar computer, $shift$ can be readily implemented using *xnetW* or *xnetE* statements.

```

integer  $r2n := A[proc\_id]$ ,  $right2n, res := 0_{\oplus}$ ,  $rightres, i, dist := 1$ ;
for  $i = 0$  to  $h(n-1) - 1$  do
  if  $b(r, i)$  then  $\{rightres, right2n\} := \{res, r2n\}$ ;
     $\{res, r2n\} := shift(\{res, r2n\}, dist)$ ;
  else  $\{rightres, right2n\} := shift(\{res, r2n\}, -dist)$ ;
  endif;
  if  $b(n, i)$  then  $res := r2n \oplus rightres$  endif;
   $r2n := r2n \oplus right2n$ ;
   $dist := dist + dist$ ;
endfor;
if  $h(n) = l(n)$  then  $res := r2n$  endif;
 $res := res \oplus shift(r2n, dist - n)$ ;

```

Figure 2: Recursive Combination Algorithm for Simultaneous Reduction.

3. If communication is always done in the left direction, the result is placed at the offset 0. Making an additional shift of variable res to the right at each step corresponding to the on-bit of the offset will place the result in the proper position. However, in such a case, the right argument of \oplus operator is shifted to the left and then (with the result of operation) to the right for the same distance. This could be simply avoided by changing res assignments to $shift(res, + \dots) \oplus res$ and therefore eliminating the left shift.

These observations led to the design of the algorithm presented in Figure 2. The presented algorithm assumes that $h(n)$ bit of binary representation of r is zero. If this is not the case, the correct algorithm can be easily obtained from the presented one by inverting directions (and arguments) of all shifts and using $n-r$ in place of r . Let ‘&’ denote the bitwise ‘and’ operator. Let n_i denotes $(i+1)$ rightmost bits of the binary representation of n , i.e., $n_i = n \& (2^{i+1} - 1)$. Likewise, let $r_i = r \& (2^{i+1} - 1)$. At the end of for-loop, the variable $r2n$ stores the results of reduction of subsections of the size $2^{i+1} = dist$. In other words, when $n_i > 0$, local variable res on processor i contains the reduction of the array section from $A[i - r_i]$ to $A[i - r_i + n_i - 1]$. Variable res represents the result of reduction that creates a subsection of size n_i , i.e., local variable $r2n$ on processor i contains the reduction of the array section from $A[i - r_i]$ to $A[i - r_i + 2^{i+1} - 1]$. Depending on the direction of the $shift$ operation, the right arguments of \oplus operation are either copied from the previous local values or obtained by $shift$. The left arguments are obtained in a complementary way to the right arguments.

The algorithm, written for clarity, uses zero of operator \oplus (which may not exist) and makes unnecessary operations and communication steps that can be easily eliminated by proper loop unrolling. The optimized program (not included here due to space limitation) skips all operations on $right2n$ for $0 \leq i \leq l(n)$. The complexity metrics of this algorithm are: $c_o = \log_2 n + \bar{z}(n) \leq 2 \log_2 n$, $c_{cm} = \lceil \log_2 n \rceil$, $c_{ch} = n-1$ and $1 \leq c_{ms} \leq 2$. When the hops are expensive, the algorithm minimizes the communication; however, it is not scalable and generates heavy communication

traffic.

3.2. Suffix-Prefix Algorithm

For this algorithm, the entire range N of array A is divided into N/n disjoint sections, each of size n . The algorithm consists of two stages:

1. executing the parallel prefix algorithm on disjoint sections of the input array A to obtain the final result in one location of each section and partial results in the others, and
2. applying operator \oplus to partial results obtained in the previous step.

The algorithm uses two auxiliary arrays, a prefix array, $P[1..N]$, and a suffix array, $S[1..N]$, defined as follows: for k -th disjoint section of array A containing all locations, i , such that $kn - n < i \leq kn$, the arrays P, S are:

$$P[i] = \bigoplus_{j=kn-n+1}^i A[j] \quad \text{for } kn - n < i < kn, \quad \text{and} \quad S[i] = \bigoplus_{j=i}^{kn} A[j] \quad (2)$$

Once the arrays P and S are known, the simultaneous reduction can be evaluated at the left end^c of the section as follows:

$$res[i] = \begin{cases} S[i] & \text{if } i = kn - n + 1, \\ S[i] \oplus P[i + n - 1] & \text{otherwise.} \end{cases}$$

This computation is inexpensive and requires a single communication *shift* for the distance $n - 1$ and a single application of the operator \oplus . The cost of evaluating arrays P and S dominates the complexity of this algorithm.

Fine-grain parallelism case. In this case, there are N processors, each storing

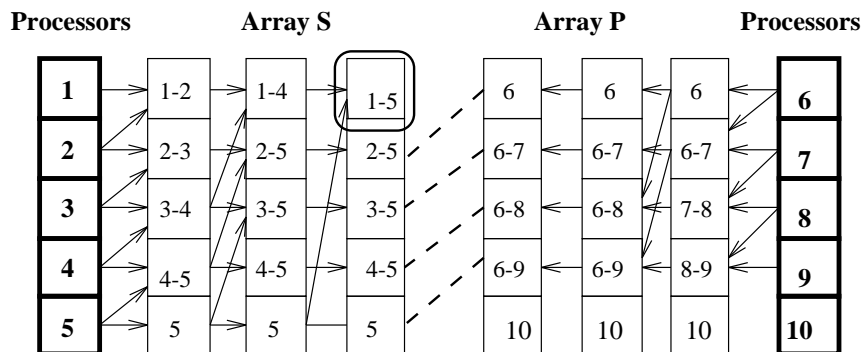


Figure 3: Prefix-Suffix Algorithm for $n = 5$.

a single element of the input array A . By using the *parallel prefix* algorithm shown in Figure 1 for each interval in parallel, the array S (or P) can be obtained at a cost of $c_o = c_{cm} = \lceil \log_2 n \rceil$; $c_{ch} = n - 1$; $c_{ms} = 1$.

^cRight-end placement of the result can be achieved by switching P and S in their roles, i.e., by using the expression $P[i] \oplus S[i - n + 1]$. Hence, the placement of the result at the given offset can always be achieved by a single shift for the distance of at most $n/2$.

If the inverse operator \ominus exists, then array P can be obtained in one step:

$$P[i] = S[kn - n + 1] \ominus S[i + 1] \quad \text{for } kn - n < i < kn$$

The value of $S[kn - n + 1]$ can be broadcast to section k in $\lceil \log_2 n \rceil$ communication shifts; hence, the cost metrics of this version of the algorithm are: $c_o = \lceil \log_2 n \rceil + 2$ and $c_{cm} = 2\lceil \log_2 n \rceil + 1$. If hops are costly, the value of $S[kn - n + 1]$ can be broadcast in $n - 1$ sequential shifts for a unit distance. The result can be sent to proper position in one shift for at most $n/2$ distance, so $c_{ch} \leq n - 1 + n - 1 + n - 1 + n/2 < 7n/2$.

When the inverse operator does not exist (e.g., the operator \oplus is maximum or minimum), array P can be obtained by another application of the parallel prefix algorithm over $n - 1$ elements and in the opposite direction than the one producing

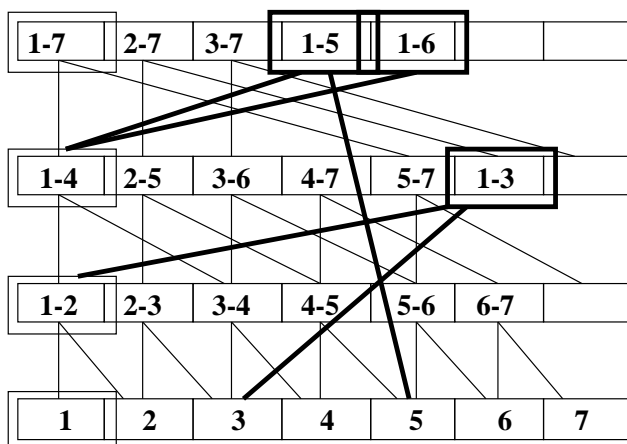


Figure 4: Evaluation of Arrays P and S on PRAM.

S , as shown in Figure 3. Therefore the complexity of the algorithm in this case is $c_o = c_{cm} < 2\lceil \log_2 n \rceil + 1$, and $c_{ch} < 7n/2$. However, during the execution of the parallel prefix algorithm, some processors are idle and some partial results are already prefixes for some elements. Hence, the PRAM implementation can take advantage of this observation to produce both P and S in just $\lceil \log_2 n \rceil$ steps. Shown in Figure 4 is the parallel prefix algorithm for the creation of array S that also generates all the prefixes needed for array P (prefixes needed for P are enclosed in the bold boxes). It can be shown that all prefixes of the form $(1..2^i)$ are byproducts of the suffix generation on processor 1. In step $i > 1$, there are 2^{i-1} processors idle, numbered from $n - 2^{i-1} + 1$ to n . They can be used to evaluate all prefixes from $(1..2^{i-1} + 1)$ to $(1..2^i - 1)$ as shown in Figure 4. The same processing can be used in each disjoint section.

Scalable case. In this case, we assume that the number of processors P is not greater than N/n ; hence, each disjoint section has at most one processor allocated to itself. For simplicity, we consider the case when $k = N/(nP)$ is an integer. If a section is assigned to a single processor, then arrays P and S can be computed in $2n - 2$ operations without involving any communication. Each processor just needs

to communicate the single boundary section, so the complexity of the algorithm is: $c_o = 3k(n-1) \approx 3N/P$; $c_{cm} = 1$; $c_{ch} = 1$; $c_{ms} = n$. The work (without communication) done by all processors is nearly constant and is about $3N$. Only when the inverse operation exists, can a single processor execute a smaller number of steps than the work of parallel processors^d. When the communication cost is included, the work becomes dependent on the number of processors and is about $3N + P * latency$.

3.3. In Pursuit of the Asymptotic Minimum

An interesting open problem is to find the asymptotically minimum operation and message counts for an arbitrary interval size. In this section, we briefly discuss two algorithms, *decomposition* and *factorization*, that approach this minimum. A more detailed description of these algorithms is given in [11].

The decomposition algorithm is based on factoring out bit patterns in the binary representation of the interval length n . Let $d_o(n)$ be the operation count (i.e., the number of binary reduction operation steps) for the decomposition algorithm operating on the interval of size n . Let $q \leq \log_2 n$ be an integer selected by the algorithm designer. The algorithm factors out bit patterns of increasing length: $2^1, 2^2, \dots, 2^q$ first from the binary representation of $n = n_1^{(0)}$ and next from the created quotients $n_j^{(i)}$. Patterns of length 2^i are denoted $p_1^{(i)}, p_2^{(i)}$, etc. In the i -th step, all quotients $n_1^{(i-1)}, n_2^{(i-1)}$, etc., are decomposed by factoring all patterns of length 2^i out and creating in the process quotients $n_1^{(i)}, n_2^{(i)}, \dots$.

In [11] we established by induction the following two properties of such decomposition, valid at each level of decomposition $i \geq 1$: (i) there are exactly three non-zero patterns to be factored out from each quotient; namely: $p_1^{(i)} = 1$, $p_2^{(i)} = 2^{2^i-1}$ and $p_3^{(i)} = 2^{2^i-1} + 1$, (ii) for each quotient $n_j^{(i)}$, if $b(n_j^{(i)}, k) = 1$ then $k = m2^i + 1$ for some integer m .

These two properties imply that the number n will be decomposed into the following sum of products of factors and quotients:

$$n = \sum_{1 \leq j_1, j_2, \dots, j_q \leq 3} n_t^{(q)} \prod_{k=1}^q p_{j_k}^{(k)} \text{ where } t = \sum_{k=0}^q 3^{q-1} j_q \quad (3)$$

The decomposition algorithm combines in parallel all quotients and then all patterns using the recursive combination algorithm from Section 3.1. The final step is to combine obtained products. Putting together all the needed operations yields:

$$d_o(n) < \log_2 n + \frac{\log_2 n}{2^q} + 2^q + q(1 + \log_2 3) \quad (4)$$

Let's consider, for example, the interval size $n = 15$, or, in binary notation, $n = (1111)_2$, so $h(n) = 3$, $\bar{z}(n) = 4$. The recursive combination algorithm presented in

^dWith the inverse operation, a single processor can evaluate all prefixes $P[1..N]$ over the entire range of data N and then find the result for $i > n$ as $res[i] = P[i] \ominus P[i-n]$, hence executing only $2N - n$ operations. However, when the reductions perform arithmetic operations on floating-point numbers, such an algorithm will introduce large rounding errors for larger values of N which makes it impractical.

Section 3.1 will require $h(n) + \bar{z}(n) - 1 = 6$ parallel operation steps, three steps to build recursively 2^3 subsections and another three to combine partial results into the final result. However, there is one pattern of length two (repeated two times) in the binary representation of n , so $n = (1)_2(11)_2(0101)_2$. Two steps are needed to create the first pattern and another three for the second pattern (to combine recursively subtrees of size three into the result). Altogether, just five steps are required by the decomposition algorithm. The different evaluation subtrees of the two algorithms discussed above are shown in Figure 5.

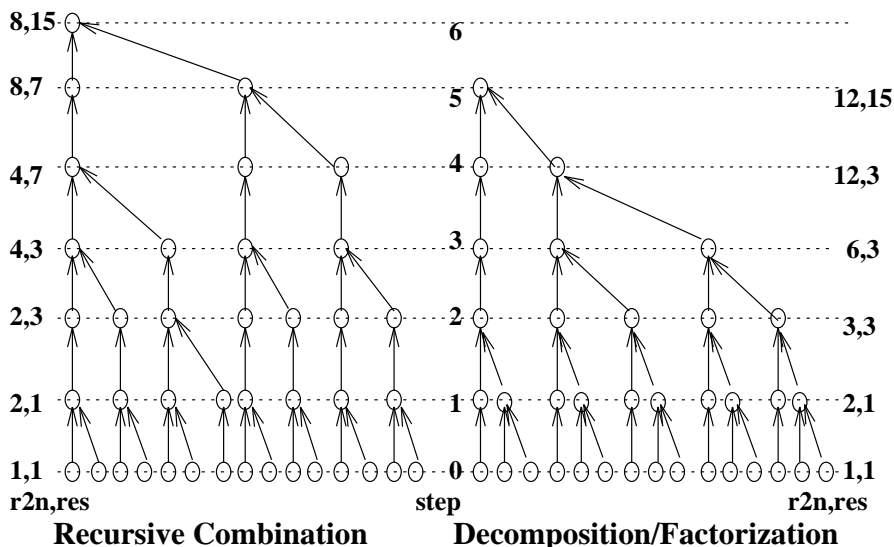


Figure 5: Subtrees for Recursive Combination vs. Decomposition/Factorization for $n = 15$ (sizes of subtrees in $r2n, res$ are shown for each \oplus operation step).

We have shown in [11] that to minimize $d_o(n)$ for the given n it is necessary to select $q \approx \frac{\log_2 \log_2 n}{2}$, and the corresponding number of operations is about $\log_2 n + 2\sqrt{\log_2 n}$. The same approach applied to the cost of execution on uniform latency machines yields: $q \approx \log_{12} \log_2 n - \log_{12}(2/3)$ and the corresponding cost is

$$d(n) \approx 2(\log_2 n + \frac{2}{3}(\frac{3}{2} \log_2 n / 2)^c) \text{ where } c = \log_{12} 6 \approx 0.721057 \dots$$

Similar analysis of hop count (see [11] for details) provides the results shown in Table 2.

A more efficient decomposition scheme is used in the following *factorization* algorithm. Let $c(n)$ be the message and operation count of this algorithm and $s(n)$ the largest subtree of the size of the power of two constructed during evaluation. For each interval size n , this algorithm selects the most efficient method of obtaining the result from the following three:

1. recursive combination for n ; in this case $s(n) = h(n)$,

n	$\lceil \log_2 n \rceil$	$\max_{k \leq n} c(k)$	k
8	3	4	7
64	6	8	43
512	9	13	479
4096	12	16	2399
32768	15	20	21407

Table 1: Operation and Message Counts for the Algorithm Based on Factoring.

2. composition of subtrees n_1, n_2 , where $n = n_1 n_2$; i.e., building first subtrees of size n_1 and then combining n_2 results; here $s(n) = s(n_1)$,
3. additive combination of subtrees n_1, n_2 , where $n = n_1 + n_2$; i.e., building a subtree n_1 and n_2 separately, except for sharing of $\min(s(n_1), s(n_2))$ steps; in this case $s(n) = \max(s(n_1), s(n_2))$.

It is interesting to notice that the recursive combination is a special case of the additive combination in which terms n_1, n_2 are restricted to powers of two. For the previously considered example of $n = 15$, the factorization algorithm will find out that $n = 3 * 5$ and requires only five steps (the evaluation subtrees are the same as for the decomposition algorithm, as shown in Figure 5). The advantage of the factorization algorithm over the decomposition algorithm can be seen for many n 's. Consider, for example, $n = 47$. The factorization algorithm will find out that $n = 2 + 5 * 9$, which yields an eight-step algorithm. The decomposition algorithm will use the following patterns $n = (1)_2(11)_2(101)_2 + (100000)_2(01)_2(0001)_2$; hence, it requires five steps to obtain one non-trivial quotient and another five to build non-trivial patterns; altogether ten steps. The factorization algorithm performs better because the decomposition algorithm is restricted to patterns (factors) that have length equal to some power of two. For $n = 47$, even the recursive combination will do better because it will use $(100000)_2 + (1000)_2 + (100)_2 + (10)_2 + 1$ as terms in the additive combination, yielding nine steps. The ability of a factorization algorithm to use $45 = 5 * 9$ as a subterm yields a more efficient solution.

For a given n , we have designed an algorithm that determines the value of $c(n)$ and the corresponding method of evaluation of simultaneous reduction. For each $k = 1, 2, \dots, n$, the algorithm checks k compositions and k additive combinations for the best method of evaluating the simultaneous reduction for an interval of size k and thereby establishes $c(k)$. The complexity of this algorithm is $O(n^2)$. The results obtained from the algorithm for $n \leq 2^{15}$ are shown in Table 1. The authors do not know any tight upper bound for the value of $c(n)$.

4. Conclusion

Table 2 summarizes the total complexity of the presented algorithms for various architectures. All logarithms are of base 2 and UCL stands for Uniform Communication Latency. For simplicity, whenever relevant, communication latency is assumed to be 1 and the hop cost is equal to the number of data sent in the message. Each

algorithm is optimal for the different architecture. The best entry in each category is printed in bold in Table 2.

Architecture	Algorithm			
	Recursive Combination	Prefix-Suffix with \ominus	Prefix-Suffix without \ominus	Decomposition
PRAM SIMD:	$2 \log n$	$\log n$	$\log n$	$\log n + 2\sqrt{\log n}$
UCL	$3 \log n$	$3 \log n$	$4 \log n$	$2 \log n + 1.78(\log n)^{0.72}$
Hypercube	$\log^2 n$	$\log^2 n / 2$	$\log^2 n$	$3 \log^2 n$
Mesh	n	$7n/2$	$7n/2$	$5n/3$

Table 2: Complexity of the Algorithms on Different Architectures.

The factorization algorithm is efficient only when the parameters are known at compile time, because determining factorization is computationally expensive. With this exception, the presented algorithms could be efficiently implemented as library routines. Some applications use simultaneous reduction with parameters known at compile time. In such cases, evaluation of some conditional statements at run-time can be avoided. This was the case for the ecological simulation discussed in Section 1. To avoid any conditional operations based on the interval size and the offset during run-time, we have designed a simple code generator, which inputs the given interval size and the offset, and produces object code that implements the proper algorithm for these parameters.

Acknowledgments

Thanks are due to the anonymous reviewers for their helpful comments on the earlier version of this paper. This work was partially supported by the ONR grant N00014-93-1-0076 and by a grant from IBM Corporation. The contents of this entry does not necessarily reflect the position or policy of the U.S. Government—no official endorsement should be inferred or implied.

References

1. B. Szymanski and T. Caraco, Spatial analysis of vector-borne disease: a four species model, *Evolutionary Ecology*, **8** (1994) 299–314.
2. B. Maniatty, B. Szymanski and T. Caraco, TEMPEST: A Fast Spatially Explicit Model of Epidemics on Parallel Machines, *Proc. High Performance Computing Symposium* (A. Tentner (ed.), SCS Publishers, San Diego, 1994) 114-119.
3. B.T Milne, Aggregation and neutral models in fractal landscapes, *Trends in Ecology and Evolution*, **139** (1992) 32–57.
4. G. Andrews, *Concurrent Programming: Principles and Practice* (Benjamin & Cummings, Redwood City, 1991).
5. K. Hwang, *Advanced Computer Architectures*, (McGraw-Hill, New York, 1993), 466.
6. J.L. Sanz and R. Cypher, Data reduction and fast routing: A strategy for efficient algorithms for message-passing parallel computers, *Algorithmica*, **7** (1992) 77–89.

7. A. Gibbons and R. Ziani, The balanced binary tree technique on mesh-connected computers, *Information Processing Letters*, **37** (1991) 101–109.
8. S. Miguet and Y. Robert, Reduction operators on a distributed memory machine with a reconfigurable interconnection, *IEEE Trans. Parallel and Distributed Systems*, **3** (1992) 501–512.
9. D. Nassimi, Parallel algorithms for the classes of $\pm 2^b$ DESCEND and ASCEND computations on SIMD hypercube, *IEEE Trans. Parallel and Distributed Systems*, **4** (1993) 1372–1381.
10. B. Maniatty, B. Szymanski and B. Sinharoy, Efficiency of data alignment on Mas-Par, *SIGPLAN Notices*, **28** (1993) 48–51.
11. B. Szymanski, B. Maniatty, and B. Sinharoy, Simultaneous parallel reduction, *Technical Report CS 92-31*, (Department of Computer Science, RPI, Troy, NY, 1992).