

# A New Parallel Skeleton for General Accumulative Computations

Hideya Iwasaki<sup>1</sup> and Zhenjiang Hu<sup>2</sup>

<sup>1</sup> Department of Computer Science, The University of Electro-Communications, 1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585 Japan. Email: iwasaki@cs.uec.ac.jp

<sup>2</sup> Graduate School of Information Science and Technology, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656 Japan. Email: hu@mist.i.u-tokyo.ac.jp

---

Skeletal parallel programming enables programmers to build a parallel program from ready-made components (parallel primitives) for which efficient implementations are known to exist, making both the parallel program development and the parallelization process easier. Constructing efficient parallel programs is often difficult, however, due to difficulties in selecting a proper combination of parallel primitives and in implementing this combination without having unnecessary creations and exchanges of data among parallel primitives and processors. To overcome these difficulties, we propose a powerful and general parallel skeleton, *accumulate*, which can be used to naturally code efficient solutions to problems as well as be efficiently implemented in parallel using MPI (Message Passing Interface).

---

**KEY WORDS:** Skeletal parallel programming; Bird–Meertens formalism; data parallel skeleton; program transformation; MPI.

## 1. INTRODUCTION

Parallel programming has proved to be difficult, requiring expert knowledge of both parallel algorithms and hardware architectures to achieve good results. The use of parallel skeletons can help to structure this process and make both programming and parallelization easier.<sup>(1–3)</sup> Programming using parallel skeletons is called *skeletal parallel programming*, or simply *skeletal programming*. Its approach is to abstract generic and recurring patterns within parallel programs as *skeletons* and then provide the skeletons as a library of “building blocks” whose parallel implementations are transparent to the programmer.

Skeletal parallel programming was first proposed by Cole,<sup>(1)</sup> and much research<sup>(4–14)</sup> has been devoted to it. Skeletons can be classified into two groups: *task parallel* skeletons and *data parallel* skeletons. Task parallel skeletons capture the parallelism through the execution of several different tasks. For example, `Pipeline1for1` in `eSkel`<sup>(13)</sup> is used for pipelining computation, and `farm` in `P3L`<sup>(5,6)</sup> is a skeleton for farming computation that uses several workers. Data parallel skeletons capture the simultaneous computations on the data partitioned among processors. Examples of this kind of skeleton are `forall` in High Performance Fortran,<sup>(15)</sup> `apply-to-call` in NESL,<sup>(7,8)</sup> and a fixed set of higher order functions such as `map`, `reduce`, `scan`, and `zip` in the Bird–Meertens Formalism<sup>(16,17)</sup> (BMF for short).

Skeletal programming is not restricted to a specific application area;<sup>(14)</sup> it provides general patterns of parallel programming to help programmers write higher-level and structured parallel programs. Skeletal programming is attractive for the three reasons. First, it simplifies parallel programming because programmers are able to build parallel programs from ready-made components (skeletons) with known efficient implementations, without being concerned with the lower level details of their implementations. Second, programs using skeletons are portable, because skeletons are abstracted toolkits whose specifications are independent of parallel environments. Third, communication problems like deadlocks and starvation are avoided, and a cost model enables us to predict the performance of a program.

Despite these advantages of skeletal programming, developing *efficient* programs, especially ones based on the use of data parallel skeletons, still remains a big challenge for two reasons. First, it is hard to choose appropriate parallel skeletons and to integrate them well so as to develop efficient parallel programs, especially when the given problem is complicated. This is due to the gap between the simplicity of parallel skeletons and the complexity of the algorithms for the problem. Second, although a single skeleton can be efficiently implemented, a combination of skeletons cannot be easily executed efficiently. This is because skeletal parallel programs are apt to introduce many intermediate data structures for communication among skeletons. To achieve good results, we have to eliminate unnecessary creations and traversals of such data structures and unnecessary inter-process communications (exchanges of data).

In this paper, we propose a powerful and general parallel skeleton called `accumulate` and describe its efficient implementation in C++ with MPI (Message Passing Interface)<sup>(18)</sup> as a solution to the above problems. Unlike the approaches that apply such optimizations as loop restructuring

to the target program, our approach provides a general recursive computation with accumulation as a library function (skeleton) with an optimized implementation. We are based on the data parallel programming model of BMF, which provides us with a concise way to describe and manipulate parallel programs. The main advantages of `accumulate` can be summarized as follows.

- The `accumulate` skeleton abstracts a typical pattern of *recursive functions* using an accumulative parameter. It provides a more natural way of describing algorithms with complicated dependencies than existing skeletons, like `scan`.<sup>(7)</sup> In fact, since `accumulate` is derived from the *diffusion theorem*,<sup>(19)</sup> most useful recursive algorithms can be captured using this skeleton.
- The `accumulate` skeleton has an architecture-independent and efficient parallel implementation. It effectively eliminates both multiple traversals of the same data structure and unnecessary creation of intermediate data structures with the help of the *fusion transformation*.<sup>(20–22)</sup> In addition, it completely eliminates unnecessary data exchanges between processors.
- The `accumulate` skeleton is efficiently implemented in C++ using MPI. MPI was selected as the base of our parallel implementation because it offers standardized parallel constructs and makes the implementation of `accumulate` more portable and practical. The `accumulate` skeleton is a polymorphic function that can accept various data types without any special syntax. This is in sharp contrast to `Skil`,<sup>(9, 10)</sup> in which enhanced syntax for describing polymorphism and functional features is introduced into a C-based language.

The organization of this paper is as follows. We review existing parallel skeletons and the diffusion theorem in Section 2. In Section 3, we define our parallel skeleton, `accumulate`, by abstracting parallel programs derivable from the diffusion theorem. In Section 4, we describe an algorithm for implementing the proposed skeleton in parallel and describe the library we developed in C++ using MPI. We present some experimental results in Section 5 and conclude with a brief summary in Section 6.

When we discuss implementations and time complexities of skeletons, we assume that  $N$  represents the number of elements in the input list and  $P$  represents the number of processors. Each processor is assigned an integer, called the “processor identifier (PID)”, between 0 and  $P - 1$ . We also assume that elements of the input list are divided into  $P$  sublists, each of which has been already distributed to the corresponding processor. Thus, each processor has a list of length  $N/P$ .

## 2. PRELIMINARIES

In this section, we briefly review our notational conventions and some basic concepts of BMF.<sup>(16,17)</sup> We also review the diffusion theorem, from which accumulate is derived.

### 2.1. Notations

We use the functional notation to describe the definitions of skeletons and programs because of its conciseness and clarity. Those who are familiar with the functional language Haskell<sup>(20)</sup> should have no problem understanding our notation.

*Function application* is denoted by a space and the argument is written without brackets. Thus,  $f a$  means  $f(a)$ . Functions are curried, and applications associate to the left. Thus,  $f a b$  means  $(f a) b$ . A function application binds stronger than any other operator, so  $f a \oplus b$  means  $(f a) \oplus b$ , but not  $f(a \oplus b)$ . Infix binary operators will often be denoted by  $\oplus$ ,  $\otimes$ , etc. and can be *sectioned*; an infix binary operator like  $\oplus$  is turned into a unary or binary function by

$$a \oplus b = (a \oplus) b = (\oplus b) a = (\oplus) a b.$$

Lists are finite sequences of values of the same type. A list is either empty, a singleton, or the concatenation of two lists. We write  $[]$  for the empty list,  $[a]$  for the singleton list with element  $a$ , and  $xs ++ ys$  for the concatenation of lists  $xs$  and  $ys$ . Concatenation is associative, and  $[]$  is its unit. For example, the term  $[1] ++ [2] ++ [3]$  denotes a list with three elements, often abbreviated to  $[1, 2, 3]$ . We also write  $x : xs$  for  $[x] ++ xs$ .

### 2.2. Primitive Skeletons in BMF

The most important skeletons in BMF are `map`, `reduce`, `scan`, and `zip`.

The `map` skeleton is the operator that applies a function to every element in a list. Informally, we have

$$\text{map } f [x_1, x_2, \dots, x_N] = [f x_1, f x_2, \dots, f x_N].$$

The cost of this skeleton is  $O(N/P)$ , provided that  $f$  is a constant-time function.

The `reduce` skeleton is the operator that collapses a list into a single value by repeated application of an associative binary operator. Informally, for associative binary operator  $\odot$  with unit  $t_\odot$ , we have

$$\begin{aligned} \text{reduce } (\odot) [] &= t_\odot \\ \text{reduce } (\odot) [x_1, x_2, \dots, x_N] &= x_1 \odot x_2 \odot \dots \odot x_N. \end{aligned}$$

If  $\odot$  is a constant time operator, the cost of `reduce` is  $O(N/P + \log P)$  based on the divide-and-conquer style calculation.

The `scanskeleton` is the operator that accumulates all intermediate results of the `reduce` computation. Informally, for associative binary operator  $\odot$  with unit  $t_\odot$ , we have

$$\text{scan}(\odot) [x_1, x_2, \dots, x_N] = [t_\odot, x_1, x_1 \odot x_2, \dots, x_1 \odot x_2 \odot \dots \odot x_N].$$

The `scanskeleton` has a time complexity of  $O(N/P + \log P)$  using the Blelloch's algorithm,<sup>(7)</sup> provided that  $\odot$  is a constant time operator.

Finally, the `zip` skeleton is the operator that merges two lists into a single list pairing the corresponding elements:

$$\text{zip} [x_1, x_2, \dots, x_N] [y_1, y_2, \dots, y_N] = [(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)].$$

Obviously, the cost of `zip` is  $O(N/P)$ .

### 2.3. The Diffusion Theorem

The recursive function in which we are interested is defined on lists in the following form.

$$\begin{aligned} h [] c &= g c \\ h (x : xs) c &= p(x, c) \oplus h xs (c \otimes q x). \end{aligned}$$

The second argument of  $h$  is an accumulative one used to pass information to the next recursive call of  $h$ . While this definition appears complicated, it can be easily understood as follows.

- If the input list is empty, the result is computed by applying some function  $g$  to accumulative parameter  $c$ .
- If the input list is not empty and its head (`car`) and tail (`cdr`) parts are  $x$  and  $xs$  respectively, then the result is generated by combining the following two values using some binary operator  $\oplus$ :
  - the result of applying  $p$  to  $x$  (head value) and  $c$  (the accumulative parameter), and
  - the recursive call of  $h$  to  $xs$  (the rest part of the input list), with  $c \otimes q x$  as its accumulative parameter.

Since function  $h$  of the above form represents the most natural recursive definition on lists with a single accumulative parameter, it is general enough to capture many algorithms.<sup>(23)</sup> As a matter of fact, when the accumulative parameter is unnecessary,  $h$  is the so-called *catamorphism*<sup>(24, 22)</sup>

or *foldr*, one of the standard functions provided in most functional language systems. Many useful functions can be expressed in the form of a catamorphism.<sup>(21)</sup>

As an example, consider the elimination of all smaller elements from a list. An element is said to be *smaller* if it is less than a previous element in the list. For example, for the list  $[1, 4, 2, 3, 5, 7]$ , 2 and 3 are smaller elements, and thus the result is  $[1, 4, 5, 7]$ . It is easy to solve this problem by scanning the list from left to right and eliminating every element that is less than the maximum of the already scanned elements. This maximum value can be held by an accumulative parameter whose initial value is  $-\infty$ , i.e.,  $\text{smaller } [1, 4, 2, 3, 5, 7] (-\infty) = [1, 4, 5, 7]$ . Thus, the function *smaller* is defined by the following recursive form with an accumulative parameter.

$$\begin{aligned} \text{smaller } [] c &= [] \\ \text{smaller } (x : xs) c &= \mathbf{if } x < c \mathbf{ then } \text{smaller } xs c \mathbf{ else } [x] \\ &\quad ++ \text{smaller } xs x. \end{aligned}$$

The second definition clause has two occurrences of a recursive call to *smaller*. A simple transformation of merging these two occurrences into a single one immediately gives the following definition:

$$\begin{aligned} \text{smaller } [] c &= [] \\ \text{smaller } (x : xs) c &= (\mathbf{if } x < c \mathbf{ then } [] \mathbf{ else } [x]) \\ &\quad ++ \text{smaller } xs (\mathbf{if } x < c \mathbf{ then } c \mathbf{ else } x). \end{aligned}$$

Thus, *smaller* can be described in the recursive form in which we are interested.

While it looks easy to write the solutions of problems in the recursive form of *h*, it is not obvious how to re-code them in terms of skeletons. The diffusion theorem<sup>(19)</sup> is a transformation rule for turning the above form of the recursive definition, which has an accumulative parameter, into a composition of BMF skeletons.

**Theorem (Diffusion).** Given is a function *h* defined in the following recursive form:

$$\begin{aligned} h [] c &= g c \\ h (x : xs) c &= p (x, c) \oplus h xs (c \otimes q x), \end{aligned}$$

if  $\oplus$  and  $\otimes$  are associative and have units, then *h* can be transformed into the following compositional form:

$$\begin{aligned} h xs c &= \text{reduce } (\oplus) (\text{map } p as) \oplus g b \\ &\quad \mathbf{where } bs ++ [b] = \text{map } (c \otimes) (\text{scan } (\otimes) (\text{map } q xs)) \end{aligned}$$

$$as = \text{zip } xs \text{ } bs.$$

Note that the list concatenation operator “+” is used as a *pattern* on the left side of the equation in the above **where** clause.

*Proof.* We can prove this theorem by induction on the first parameter of  $h$ , as shown by the following calculation.

*Base case.*  $xs = []$

$$\begin{aligned}
& h [] c \\
&= \text{reduce } (\oplus) (\text{map } p \text{ } as) \oplus g b \\
&\quad \text{where } bs ++ [b] = \text{map } (c \otimes) (\text{scan } (\otimes) (\text{map } q [])) \\
&\quad\quad as = \text{zip } [] \text{ } bs \\
&= \{ \text{Definition of map} \} \\
&\quad \text{reduce } (\oplus) (\text{map } p \text{ } as) \oplus g b \\
&\quad \text{where } bs ++ [b] = \text{map } (c \otimes) (\text{scan } (\otimes) []) \\
&\quad\quad as = \text{zip } [] \text{ } bs \\
&= \{ \text{Definition of scan} \} \\
&\quad \text{reduce } (\oplus) (\text{map } p \text{ } as) \oplus g b \\
&\quad \text{where } bs ++ [b] = \text{map } (c \otimes) [t_{\otimes}] \\
&\quad\quad as = \text{zip } [] \text{ } bs \\
&= \{ \text{Definition of map; } c \otimes t_{\otimes} = c \} \\
&\quad \text{reduce } (\oplus) (\text{map } p \text{ } as) \oplus g b \\
&\quad \text{where } bs ++ [b] = [c] \\
&\quad\quad as = \text{zip } [] \text{ } bs \\
&= \{ \text{Pattern matching: } bs = [], b = c; \text{ Definition of zip} \} \\
&\quad \text{reduce } (\oplus) (\text{map } p [] ) \oplus g c \\
&= \{ \text{Definition of map} \} \\
&\quad \text{reduce } (\oplus) [] \oplus g c \\
&= \{ \text{Definition of reduce} \} \\
&\quad t_{\oplus} \oplus g c \\
&= g c
\end{aligned}$$

*Inductive case.*  $xs = x : xs'$

$$\begin{aligned}
& h (x : xs') c \\
&= \text{reduce } (\oplus) (\text{map } p \text{ } as) \oplus g b \\
&\quad \text{where } bs ++ [b] = \text{map } (c \otimes) (\text{scan } (\otimes) (\text{map } q (x : xs')))
\end{aligned}$$

$$\begin{aligned}
& as = \text{zip } (x : xs') \ bs \\
= & \{ \text{Definition of map} \} \\
& \text{reduce } (\oplus) \ (\text{map } p \ as) \oplus \ g \ b \\
& \text{where } bs \ ++ \ [b] = \text{map } (c \otimes) \ (\text{scan } (\otimes) \ (q \ x : \text{map } q \ xs')) \\
& \quad as = \text{zip } (x : xs') \ bs \\
= & \{ \text{Definition of scan;} \} \\
& \{ \text{scan } (\otimes) \ (y : ys) = \iota_{\otimes} : \text{map } (y \otimes) \ (\text{scan } (\otimes) \ ys) \} \\
& \text{reduce } (\oplus) \ (\text{map } p \ as) \oplus \ g \ b \\
& \text{where } bs \ ++ \ [b] \\
& \quad = \text{map } (c \otimes) \ (\iota_{\otimes} : \text{map } (q \ x \otimes) \ (\text{scan } (\otimes) \ (\text{map } q \ xs'))) \\
& \quad as = \text{zip } (x : xs') \ bs \\
= & \{ \text{Definition of map;} \ c \otimes \iota_{\otimes} = c \} \\
& \text{reduce } (\oplus) \ (\text{map } p \ as) \oplus \ g \ b \\
& \text{where } bs \ ++ \ [b] \\
& \quad = c : \text{map } (c \otimes) \ (\text{map } (q \ x \otimes) \ (\text{scan } (\otimes) \ (\text{map } q \ xs'))) \\
& \quad as = \text{zip } (x : xs') \ bs \\
= & \{ \text{map } f \ (\text{map } g) = \text{map } (f \circ g); \text{Associativity of } \otimes \} \\
& \text{reduce } (\oplus) \ (\text{map } p \ as) \oplus \ g \ b \\
& \text{where } bs \ ++ \ [b] = c : \text{map } ((c \otimes q \ x) \otimes) \ (\text{scan } (\otimes) \ (\text{map } q \ xs')) \\
& \quad as = \text{zip } (x : xs') \ bs \\
= & \{ \text{Pattern matching: } bs = c : bs' \} \\
& \text{reduce } (\oplus) \ (\text{map } p \ as) \oplus \ g \ b \\
& \text{where } bs' \ ++ \ [b] = \text{map } ((c \otimes q \ x) \otimes) \ (\text{scan } (\otimes) \ (\text{map } q \ xs')) \\
& \quad as = \text{zip } (x : xs') \ (c : bs') \\
= & \{ \text{Definition of zip;} \ as = a : as' \} \\
& \text{reduce } (\oplus) \ (\text{map } p \ (a : as')) \oplus \ g \ b \\
& \text{where } bs' \ ++ \ [b] = \text{map } ((c \otimes q \ x) \otimes) \ (\text{scan } (\otimes) \ (\text{map } q \ xs')) \\
& \quad a : as' = (x, c) : \text{zip } xs' \ bs' \\
= & \{ \text{Definition of map;} \ a = (x, c) \} \\
& \text{reduce } (\oplus) \ (p \ (x, c) : \text{map } p \ as') \oplus \ g \ b \\
& \text{where } bs' \ ++ \ [b] = \text{map } ((c \otimes q \ x) \otimes) \ (\text{scan } (\otimes) \ (\text{map } q \ xs')) \\
& \quad as' = \text{zip } xs' \ bs' \\
= & \{ \text{Definition of reduce} \} \\
& p \ (x, c) \oplus (\text{reduce } (\oplus) \ (\text{map } p \ as') \oplus \ g \ b) \\
& \text{where } bs' \ ++ \ [b] = \text{map } ((c \otimes q \ x) \otimes) \ (\text{scan } (\otimes) \ (\text{map } q \ xs')) \\
& \quad as' = \text{zip } xs' \ bs' \\
= & \{ \text{Induction hypothesis} \}
\end{aligned}$$

$$p(x, c) \oplus h \, xs' (c \otimes q x) \quad \blacksquare$$

It is worth noting that the transformed program in the BMF skeletons is efficient, in the sense that if the original (recursive) program uses  $O(N)$  sequential time, then the diffused one takes  $O(N/P + \log P)$  parallel time. Section 4 describes a detailed algorithm for computing the transformed parallel program in  $O(N/P + \log P)$  time.

To see how the diffusion theorem works, recall the example of eliminating smaller elements in a list, in which *smaller* has the following definition:

$$\begin{aligned} \text{smaller } [] c &= [] \\ \text{smaller } (x : xs) c &= (\text{if } x < c \text{ then } [] \text{ else } [x]) \\ &\quad ++ \text{smaller } xs (\text{if } x < c \text{ then } c \text{ else } x). \end{aligned}$$

Matching the above recursive definition with that in the diffusion theorem yields

$$\begin{aligned} \text{smaller } xs c &= \text{reduce } (++) (\text{map } p \, as) ++ g \, b \\ \text{where } bs ++ [b] &= \text{map } (c \otimes) (\text{scan } (\otimes) (\text{map } q \, xs)) \\ as &= \text{zip } xs \, bs \\ c \otimes x &= \text{if } x < c \text{ then } c \text{ else } x \\ p(x, c) &= \text{if } x < c \text{ then } [] \text{ else } [x] \\ g \, c &= [] \\ q \, x &= x. \end{aligned}$$

Consequently, we have obtained an  $O(N/P + \log P)$  parallel algorithm.

### 3. THE accumulate PARALLEL SKELETON

As described above, the diffusion theorem is applicable to a wide class of recursive functions on lists, abstracting a good combination of primitive parallel skeletons *map*, *reduce*, *scan*, and *zip* in BMF. Our new parallel skeleton, *accumulate*, is defined based on the theorem.

**Definition.** Let  $g$ ,  $p$ , and  $q$  be functions, and let  $\oplus$  and  $\otimes$  be associative operations with units. The *accumulate* skeleton is defined by

$$\begin{aligned} \text{accumulate } [] c &= g \, c \\ \text{accumulate } (x : xs) c &= p(x, c) \oplus \text{accumulate } xs (c \otimes q x). \end{aligned}$$

Since *accumulate* is uniquely determined by  $g$ ,  $p$ ,  $q$ ,  $\oplus$ , and  $\otimes$ , we can parameterize them and use the special notation  $[[g, (p, \oplus), (q, \otimes)]]$  for *accumulate*.

The **accumulate** skeleton was previously called **diff**,<sup>(25)</sup> but it has been renamed so as to reflect its characteristic feature of describing data dependencies.

It is a direct consequence of the diffusion theorem that **accumulate** can be rewritten as

$$\begin{aligned} \llbracket g, (p, \oplus), (q, \otimes) \rrbracket xs \ c = & \text{reduce } (\oplus) (\text{map } p \ as) \oplus g \ b \\ \text{where } bs \ ++ \ [b] = & \text{map } (c \ \otimes) (\text{scan } (\otimes) (\text{map } q \ xs)) \\ as = & \text{zip } xs \ bs. \end{aligned}$$

The **accumulate** skeleton is a data parallel skeleton that acts on partitioned data and abstracts good combinations of primitive skeletons such as **map**, **reduce**, **scan**, and **zip**. To use **accumulate**, programmers need not know in detail how these primitive skeletons are combined. Thus, **accumulate** is quite adequate for a general skeleton in parallel programming that solves the problems pointed out in the Introduction.

Returning to the example of *smaller* in the previous section, we are able to code it directly in terms of **accumulate** as follows:

$$\begin{aligned} \text{smaller } xs \ c = & \llbracket g, (p, +), (q, \otimes) \rrbracket xs \ c \\ \text{where } c \ \otimes \ a = & \text{if } a < c \ \text{then } c \ \text{else } a \\ p \ (x, c) = & \text{if } x < c \ \text{then } [] \ \text{else } [x] \\ g \ c = & [] \\ q \ x = & x. \end{aligned}$$

As seen here, to write a parallel program in terms of **accumulate**, programmers need only find suitable parameters given in  $\llbracket g, (p, \oplus), (q, \otimes) \rrbracket$  notation. In many cases, it is easy to find  $g$  and  $p$  because they are functions applied to each element of an input list in the recursive definition of interest. The only possible difficulty is to find suitable *associative* operators  $\oplus$  and  $\otimes$  together with  $q$ . The *context preservation* transformation<sup>(26)</sup> may provide a systematic way to deal with this difficulty but a detailed discussion on this point is beyond the scope of this paper.

To see how powerful and practical the new skeleton is for describing parallel algorithms, consider another problem of checking whether tags are well matched or not in a document written in XML (eXtensible Markup Language). Although this problem is of practical interest, designing an efficient  $O(\log N)$  program, where  $N$  denotes the number of separated words in the document, using parallel skeletons is not easy.

We can start with a naive sequential program to solve this problem using a stack. When an open tag is encountered, it is placed on the stack. For a close tag, first it is compared with the open tag at the stack top, and if they correspond, the open tag is removed from the stack. This straightfor-

ward algorithm is recursively defined with an accumulative parameter that represents the stack in the following way:

$$\begin{aligned}
 \text{tagmatch } [] \text{ } cs &= \text{isEmpty } cs \\
 \text{tagmatch } (x : xs) \text{ } cs &= \text{if } \text{isOpen } x \text{ then } \text{tagmatch } xs \text{ (push } x \text{ } cs) \\
 &\quad \text{else if } \text{isClose } x \text{ then } \text{notEmpty } cs \wedge \text{match } x \text{ (top } cs) \\
 &\quad \quad \quad \wedge \text{tagmatch } xs \text{ (pop } xs) \\
 &\quad \text{else } \text{tagmatch } xs \text{ } cs.
 \end{aligned}$$

Here, for simplicity, an XML document file is represented as a list of separated tags and words. Since the detailed process for finding both suitable associative operators and an adequate representation of a stack can be found in elsewhere,<sup>(19)</sup> we omit its description here. Applying the diffusion theorem gives the following efficient parallel program of *tagmatch* in terms of *accumulate*:

$$\begin{aligned}
 \text{tagmatch } xs \text{ } cs &= \llbracket \text{isEmpty}, (p, \wedge), (q, \otimes) \rrbracket xs \text{ } cs \\
 \text{where} \\
 p(x, cs) &= \text{if } \text{isOpen } x \text{ then } \text{True} \\
 &\quad \text{else if } \text{isClose } x \text{ then } \text{notEmpty } cs \wedge \text{match } x \text{ (top } cs) \\
 &\quad \text{else } \text{True} \\
 qx &= \text{if } \text{isOpen } x \text{ then } ([x], 1, 0) \\
 &\quad \text{else if } \text{isClose } x \text{ then } ([], 0, 1) \\
 &\quad \text{else } ([], 0, 0) \\
 (s_1, n_1, m_1) \otimes (s_2, n_2, m_2) &= \text{if } n_1 \leq m_2 \text{ then } (s_2, n_2, m_1 + m_2 - n_1) \\
 &\quad \text{else } (s_2 ++ \text{drop } m_2 \text{ } s_1, n_1 + n_2 - m_2, m_1).
 \end{aligned}$$

Function *drop*  $m_2$   $s_1$  drops the first  $m_2$  elements from list  $s_1$ . In the above definition, a stack is represented as a tuple: its first element is a list of unclosed tags, the second is the length of the first element, and the third is the number of pop occurrences. The initial value of accumulative parameter  $cs$  of *tagmatch* is the empty stack  $([], 0, 0)$ .

#### 4. IMPLEMENTATION

Although not all combinations of primitive skeletons can guarantee the existence of efficient parallel implementation, the *accumulate* skeleton, a specific combination of primitive skeletons, can be efficiently implemented. We implemented *accumulate* using MPI, a standard parallel library widely used for parallel programming from massively parallel computers to PC clusters. We describe our architecture-independent algorithm

for the implementation of `accumulate` before giving our C++ library using MPI.

#### 4.1. The Algorithm

To implement `accumulate` efficiently we *fuse* (or merge) as many functional compositions as possible without sacrificing inherent parallelism by using optimization techniques of fusion transformation.<sup>(20, 22)</sup> Fusion transformation enables us to eliminate unnecessary intermediate data structures and thus avoid unnecessary traversals of data.

We start our description of `accumulate` in terms of the BMF primitive skeletons in Section 3:

$$\begin{aligned} \llbracket g, (p, \oplus), (q, \otimes) \rrbracket xs\ c = & \text{reduce } (\oplus) (\text{map } p\ as) \oplus g\ b \\ \text{where } bs\ ++\ [b] = & \text{map } (c\ \otimes) (\text{scan } (\otimes) (\text{map } q\ xs)) \\ as = & \text{zip } xs\ bs. \end{aligned}$$

A naive implementation of `accumulate` is to use the existing skeletons based on the above equation, but it is too inefficient because it generates many unnecessary intermediate data structures. For example, a list whose length is equal to that of  $xs$  is produced by `map`  $q\ xs$ , but immediately consumed by `scan`  $(\otimes)$  in the **where** clause. This list is an intermediate and transitory object that is not a part of the final answer. Similarly,  $as$  is transitory and consumed by `map`  $p$ . The fusion transformation helps to eliminate these intermediate data structures and yields the following form of `accumulate`:

$$\begin{aligned} \llbracket g, (p, \oplus), (q, \otimes) \rrbracket xs\ c = & \text{reducel2 } (\ominus) \iota_{\oplus} xs\ bs\ \oplus g\ b \\ \text{where } bs\ ++\ [b] = & \text{scanl } (\odot) c\ xs \\ u \ominus (v, w) = & u \oplus p\ (v, w) \\ s \odot t = & s \otimes q\ t, \end{aligned}$$

in which the following functions are introduced.

- The function `reducel2` collapses two lists simultaneously from left to right into a single value using an initial value  $e$  and some binary operator  $\odot$ .

$$\begin{aligned} \text{reducel2 } (\odot) e\ [x_1, x_2, \dots, x_N]\ [y_1, y_2, \dots, y_N] \\ = (\dots((e \odot (x_1, y_1)) \odot (x_2, y_2)) \odot \dots) \odot (x_N, y_N) \end{aligned}$$

Using this function, we can avoid using `zip` and fuse `reduce`  $(\oplus)$  `(map`  $p\ as)$  in the naive implementation of `accumulate` into `reducel2`  $(\ominus) \iota_{\oplus} xs\ bs$ , where  $\ominus$  is defined as  $u \ominus (v, w) = u \oplus p\ (v, w)$ .

- The function `scanl` scans a list from left to right with initial value  $e$  and some binary operator  $\odot$ .

$$\begin{aligned} \text{scanl } (\odot) e [x_1, x_2, \dots, x_N] \\ = [e, e \odot x_1, (e \odot x_1) \odot x_2, \dots, (\dots((e \odot x_1) \odot x_2) \odot \dots) \odot x_N] \end{aligned}$$

Using this function, we can fuse `map` ( $c \otimes$ ) (`scan` ( $\otimes$ ) (`map`  $q$   $xs$ )) in the naive implementation into `scanl` ( $\otimes$ )  $c$   $xs$ , where  $\otimes$  is defined as  $u \otimes v = u \otimes q v$ .

The binary operators given as the first argument to `reducel2` and `scanl` are not necessarily associative. Therefore, these functions have inherent sequentiality that makes them investigate input lists from left to right. Fortunately,  $\ominus$  and  $\otimes$ , which are arguments of `reducel2` and `scanl` respectively in `accumulate` after the fusion transformation, are defined in terms of associative operators  $\oplus$  and  $\otimes$ . This associativity enables `reducel2` and `scanl` to be calculated in parallel without sacrificing the parallelism in `accumulate`. The key idea is that computation on each processor is performed using a non-associative operator ( $\ominus$  or  $\otimes$ ), while results from all processors are combined using an associative operator ( $\oplus$  or  $\otimes$ ) based on the binary tree structure of processors.

Our parallel implementation is based on the rewritten definition of `accumulate` in terms of `reducel2` and `scanl`.

The efficient parallel implementation of `reducel2` ( $\ominus$ )  $xs$   $bs$  is based on the property

$$\begin{aligned} \text{reducel2 } (\ominus) \iota_{\oplus} (xs_0 ++ xs_1 ++ \dots ++ xs_{P-1}) \\ (bs_0 ++ bs_1 ++ \dots ++ bs_{P-1}) \\ = \text{reducel2 } (\ominus) \iota_{\oplus} xs_0 bs_0 \oplus \text{reducel2 } (\ominus) \iota_{\oplus} xs_1 bs_1 \\ \oplus \dots \oplus \text{reducel2 } (\ominus) \iota_{\oplus} xs_{P-1} bs_{P-1}, \end{aligned}$$

where  $xs_i$  and  $bs_i$  ( $1 \leq i < P$ ) are not empty and have the same length. The implementation is summarized as follows.

**Step 1** Each processor first reduces its local list,  $xs_i$ , from left to right by using  $\ominus$  and gets its local value.

**Step 2** Based on the binary tree structure of the processors given by the distribution of the input list, an upward sweep of the values of local reductions is performed by inter-processor communication using associative operator  $\oplus$ .

After the upward sweep, processor 0 has the resultant value of the sweep, and this is the final answer of the computation of `reducel2`. The

cost of local reduction is  $O(N/P)$ , and that of the upward sweep is  $O(\log P)$ . The total cost is thus  $O(N/P + \log P)$  time, provided that both  $p$  and  $\oplus$  (and consequently  $\ominus$ ) are constant time operations.

Our parallel implementation of `scanl` is more complicated than that of `reduce12` because it needs both upward and downward sweeps. More specifically to implement `scanl` ( $\otimes$ )  $c$   $x$ s, we used an extended version of Blelloch's algorithm.<sup>(7)</sup> First, similar to the case of `reduce12`, non-associative operator  $\otimes$  is used in the local scan on each processor, while associative operator  $\otimes$  is used in a combination of the half-finished results in the upward and downward sweeps. Second, our algorithm allows `scanl` to accept an initial value  $c$  other than only the unit of binary operator  $\otimes$ . This enhances the descriptive power of `scanl` without adding overhead.

The detailed step in our algorithm for calculating `scanl` ( $\otimes$ )  $c$   $x$ s is as follows. Figure 1 shows a specific example of calculating `scanl` ( $\otimes$ ) 100 [1, ..., 8] using four processors, where  $\otimes$  is defined as  $s \otimes t = s + 2 \times t$  ( $q$   $x$  and  $\otimes$  are defined as  $2 \times x$  and  $+$ , respectively). Based on the distribution of the input list, all processors form a tree structure, and each internal node of this tree is assigned to the same processor as that of its left child (Fig. 1(a)).

**Step 1** Each processor locally applies `scanl` ( $\otimes$ )  $t_{\otimes}$ , i.e., it scans the distributed list from left to right using  $\otimes$  to form a scanned list, without preserving the first value ( $t_{\otimes}$ ) of the list (Fig. 1(b)). This phase is executable totally in parallel in  $O(N/P)$  time.

**Step 2** Similar to the process of `reduce12`, upward sweep of the final values of the scanned lists is performed using the associative operator  $\otimes$ . This step needs  $O(\log P)$  time. After this step, processor 0 has the value of the reduction (72 in this example) of the entire input list (Fig. 1(c)).

**Step 3** From the root of the tree structure, downward sweep by  $\otimes$  to the leaves is performed. Initially, the seed value  $c$  (100 in this example) is put on the root (Fig. 1(d)). At each sweep step, every node applies  $\otimes$  to its own value and its left child's value and then passes the result to its right child (Figs. 1(e)–(f)).

**Step 4** Finally, each processor maps the resultant value obtained in the previous step to the locally scanned list using the operator  $\otimes$  (Fig. 1(g)). Obviously, this step needs  $O(N/P)$  time.

The total cost of `scanl` ( $\otimes$ )  $c$   $x$ s is also  $O(N/P + \log P)$  time, provided that  $q$  and  $\otimes$  (and thus  $\otimes$ ) can be carried out in constant time.

To sum up, our implementation of `accumulate` uses  $O(N/P + \log P)$  parallel time.

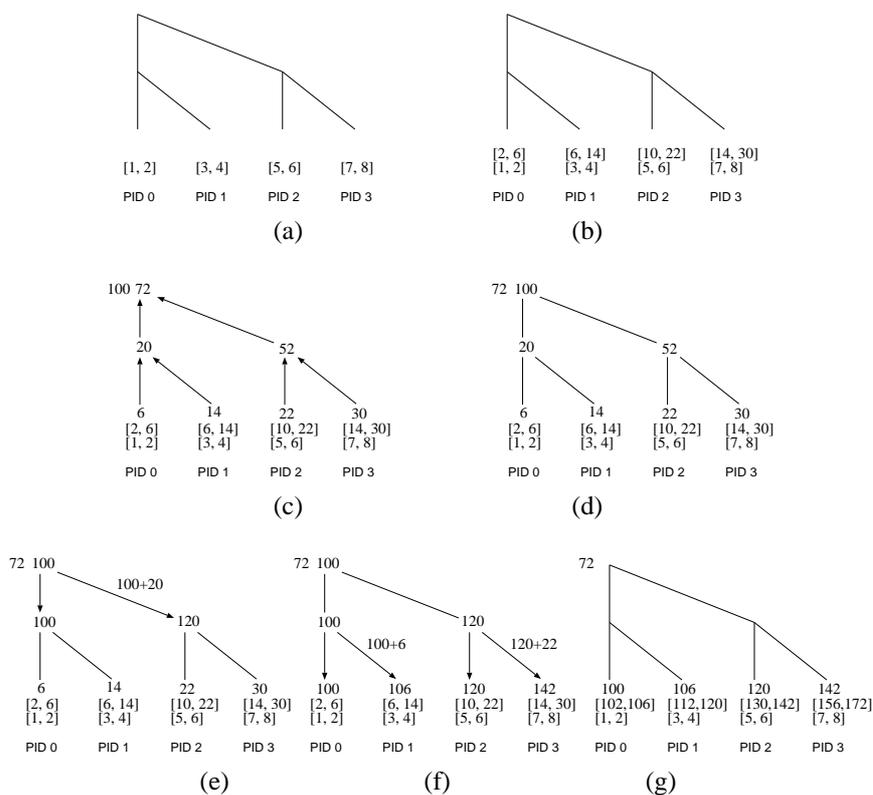


Fig. 1. Implementation of scanl.

There are two points to be noted for this efficient implementation of accumulate.

- Since the result of scanl (i.e.,  $bs$ ) is immediately used in reduce12, Step 4 in scanl and Step 1 in reduce12 can be fully overlapped. This avoids duplicate traversals of  $bs$ .
- As a side effect of the scanl execution process, processor 0 is able to obtain the last value of the resultant list of scanl simply by applying  $\otimes$  to the initial value (seed) of the scan and the result of the upward sweep. For example, as illustrated in Fig. 1(f), processor 0 can easily get the value 172 (the last value of the resultant list that resides on PID 3) by computing  $100 + 72$ , where 100 is the seed of scanl and

72 is the result of the upward sweep. This avoids extra inter-processor communication and leads to an efficient implementation.

## 4.2. The Library in C++

So far the efficient implementation of the new skeleton `accumulate` has been algorithmically addressed in the “functional” style. From the practical point of view, it is important to give a “procedural” implementation of the skeleton. We have thus implemented `accumulate` and other simpler skeletons as a library in C++ using MPI. We selected MPI because it is a standardized and portable library used on a wide variety of parallel computers. In fact, it is MPI that enables our implementation of `accumulate` to be easily ported only with re-compilation onto an IEEE 1394-based PC cluster and an Ethernet-connected PC cluster.

We aim to provide flexible skeletons that are capable of accepting various types of user-defined functions. For example, consider `map`; it has the type  $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ , i.e., the first argument is a function from type  $a$  to type  $b$ , the second argument is a list whose elements are of type  $a$ , and the result is a list whose elements are of type  $b$ , where  $a$  and  $b$  are *type variables*. Type variables are instantiated to concrete types such as `Int` or `Char` depending on the problem. Rather than preparing many instances of library functions for `map`, e.g., a `map` function for  $a = \text{Int}$  and  $b = \text{Int}$  and another `map` function for  $a = \text{Int}$  and  $b = \text{Char}$ , we provide a *polymorphic* function that generates various instances of `map` based on the given types.

Even though we fix  $a$  and  $b$ , say  $a = \text{Int}$  and  $b = \text{Char}$ , the C++ function used in `map` may have the type `char*(int)`, `char*(int*)`, `void*(int, char*)`, or `void*(int*, char*)`. The library function for `map` must accept these various types.

Since this kind of programming in C is not easy, we decided to use the template and overloading mechanisms in C++. Template enables parameterization of types in function definitions, while overloading enables type-directed selection of suitable C++ functions. Using these mechanisms, we can make the functions compatible with any combination of data types. The C++ function `map`,<sup>3</sup> which implements the `map` skeleton is defined as shown in Fig. 2, for example. In this definition of `map`, input and output lists are supposed to be represented by arrays, and the size (length) of the lists is passed to `map` via its argument.

We implemented the `accumulate` skeleton in the same way. Fragment of the C++ code of the `accumulate` function that implements  $[[g, (p, \oplus), (q, \otimes)]]$  is given in Fig. 3. The arguments `g`, `p`, `oplus`, `q`, and `otimes` correspond

<sup>3</sup> We use the typewriter face to denote the C++ functions we implemented.

```

template <class A, class B>
void map(B (*f)(A), A *from, B *to, int size)
{
    for (int i = 0; i < size; i++) to[i] = f(from[i]);
}
template <class A, class B>
void map(B (*f)(A*), A *from, B *to, int size)
{
    for (int i = 0; i < size; i++) to[i] = f(&from[i]);
}
template <class A, class B>
void map(void (*f)(A,B*), A *from, B *to, int size)
{
    for (int i = 0; i < size; i++) f(from[i], &to[i]);
}
template <class A, class B>
void map(void (*f)(A*,B*), A *from, B *to, int size)
{
    for (int i = 0; i < size; i++) f(&from[i], &to[i]);
}

```

Fig. 2. Definition of map.

to  $g$ ,  $p$ ,  $\oplus$ ,  $q$ , and  $\otimes$ , respectively. In this implementation, input list  $xs$  is represented as array  $xs$  whose length is  $size$ . During the calculation of `accumulate`, working area is needed for list  $bs$ . To avoid dynamic memory allocation of working area, programmers have to explicitly give the pointer to this work area as the eighth argument,  $bs$ , with length  $size+1$ . The initial value of accumulative parameter  $c$  is passed to `accumulate` in  $bs[0]$ .

There are two points that should be noted.

- The `accumulate` function in our library does not distribute the processing data (given as sixth argument  $xs$ ) among processors; it assumes that each processor already has the assigned data in its local memory. This reduces duplicated inter-processor communication.
- In the current version of our library, programmers need to give correct data types of `MPI::Datatype` for use in interprocess communication;  $db$  and  $dc$  correspond to classes  $B$  and  $C$ , respectively. Although the types might be derivable from other parameters, the programmer must give the corresponding data types.

```

template <class A, class B, class C>
C accumulate(C (*g)(B), C (*p)(A,B), C (*oplus)(C,C),
             B (*q)(A), B (*otimes)(B,B),
             A *xs, int size, B *bs,
             MPI::Datatype db, MPI::Datatype dc)
{ .... }
template <class A, class B, class C>
C accumulate(C (*g)(B*), C (*p)(A,B*), C (*oplus)(C,C),
             void (*q)(A,B*), void (*otimes)(B*,B*,B*),
             A *xs, int size, B *bs,
             MPI::Datatype db, MPI::Datatype dc)
{ .... }

```

Fig. 3. Fragment of the definition of `accumulate`.

To use `accumulate`, programmers need only define suitable parameters — five functions, pointers to processing data and working area, and data types. A program for the tag matching problem is given in Fig. 4. It is a direct translation of *tagmatch* in terms of `accumulate` given in Section 3. The functions `isepmty`, `p`, `andand`, `q`, and `otimes` correspond to the functions and operators in the definition of *tagmatch*. In this example, the call to `accumulate` in the main function matches the second template function of `accumulate` in Fig. 3.

For simplicity, we assume that an XML document is represented as an array of `unsigned char`, each of which is a “code number” of a tag or a “word number” in the document. Before calling `accumulate`, the main function divides this array into equally-sized fragments and distributes them to the corresponding processors. The macro `isopen(c)` returns a boolean value (represented in `char`) indicating whether `c` is an open tag or not. Similarly, the macro `isclose(c)` returns whether `c` is a close tag or not. In addition, the macro `matchp(o, c)` determines whether an open tag `o` and a close tag `c` match or not. The type `Stack` is a structure that implements the tuple representation of the stack. For simplicity, we assume that the maximum depth of the nesting open tags is within 10.

## 5. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the `accumulate` skeleton, we tested it on two problems:

- the tag matching problem whose parallel program in terms of `accumulate` is shown in Fig. 4, and

```

typedef struct { char st[10], n, m; } Stack;

char isempty(Stack *s)
{ return (s->n == 0); }

char p(unsigned char c, Stack *s)
{
    if (isopen(c)) return 1;
    else if (isclose(c)) return ((s->n > 0) && matchp(s->st[0], c));
    else return 1;
}

char andand(char x, char y)
{ return x && y; }

void q(unsigned char c, Stack *s)
{
    if (isopen(c)) { s->n = 1; s->m = 0; s->st[0] = c; }
    else { s->n = 0; s->m = ((isclose(c)) ? 1 : 0); }
}

void otimes(Stack *s1, Stack *s2, Stack *r)
{
    int n1 = s1->n, m1 = s1->m, n2 = s2->n, m2 = s2->m;
    if (n2 > 0 && r != s2)
        for (int i = 0; i < n2; i++) r->st[i] = s2->st[i];
    if (n1 <= m2) {
        r->n = n2; r->m = m1 + m2 - n1;
    } else {
        for (int i = 0; i < n1 - m2; i++)
            r->st[n2 + i] = s1->st[m2 + i]; r->n = n1 + n2 - m2; r->m = m1;
    }
}

int main(int argc, char *argv[])
{
    unsigned char *a;
    Stack *s;
    int ndata, r;
    MPI::Datatype stacktype;

    /* allocates two arrays a and s, and initializes a and stacktype */
    /* distributes each fragment of a to the corresponding processor */
    ...
    s[0].n = s[0].m = 0; /* initial value of accumulative parameter */
    r = accumulate(isempty, p, andand, q, otimes, a, ndata, s,
                  stacktype, MPI::CHAR);
    ...
}

```

Fig. 4. Using accumulate for tag matching problem.

- the line-of-sight problem.<sup>(7)</sup>

In the line-of-sight problem, given are a terrain map in the form of a grid of altitudes and an observation point, and the objective is to find which points are visible along a ray originating at the observation point. The terrain map is given as a list of points, each element of which is a pair  $(d, a)$ , where  $a$  is the altitude of the point and  $d$  is its distance from the observation point. For instance, if the list is  $[(1, 1), (2, 5), (3, 2), (4, 7), (5, 19), (6, 2)]$ , the answer is  $[True, True, False, False, True, False]$ , where *True* indicates that the corresponding point is visible. We simplified the problem to be to find the number of visible points.

The function *lineofsight*<sup>(27)</sup> to solve this simplified line-of-sight problem can be described in terms of *accumulate*:

```

lineofsight xs c = [[g, (p, +), (q, ↑)]] xs c
  where g c = 0
        p (x, c) = if c ≤ angle x then 1 else 0
        q x = angle x,

```

where *angle*  $(d, a)$  returns  $a/d$  and  $x \uparrow y$  returns the bigger of  $x$  and  $y$ . The accumulative parameter of *lineofsight* holds the maximum value of *angle* for the points investigated. The C++ program we used was based on the above form of *lineofsight*.

The parallel environment was a PC cluster system called *FireCluster*<sup>(28)</sup> with eight processors connected by an IEEE 1394 serial bus. The IEEE 1394 is becoming a standard for connecting PCs because it provides high-speed (400 Mbps) and low-latency (7.5  $\mu$ s) communications. It is thus well suited for programs that need frequent communications using a limited amount of data, like *accumulate*. The PCs had an 800 MHz Intel Celeron processor and 256 MB SDR-SDRAM. The operating system was Linux kernel 2.4.7., and the MPI implementation was MPICH 1.2.2. Our implementation of *accumulate* is not specific for the IEEE 1394 architecture; it uses general-purpose MPI functions (e.g., *MPI\_Send*) provided by the FireCluster system. Since this PC cluster is a distributed memory system, the entire input has to be divided and distributed to each processor before parallel computation.

We executed *tagmatch* and *lineofsight* with a data size (input list length) of 500,000.

Figures 5 and 6 show the execution times of the two programs. The vertical axis represents the execution time ratio (speedup) in percent compared to the time for a single processor, and the horizontal axis is the number of processors. The solid line shows the speedup when the time needed

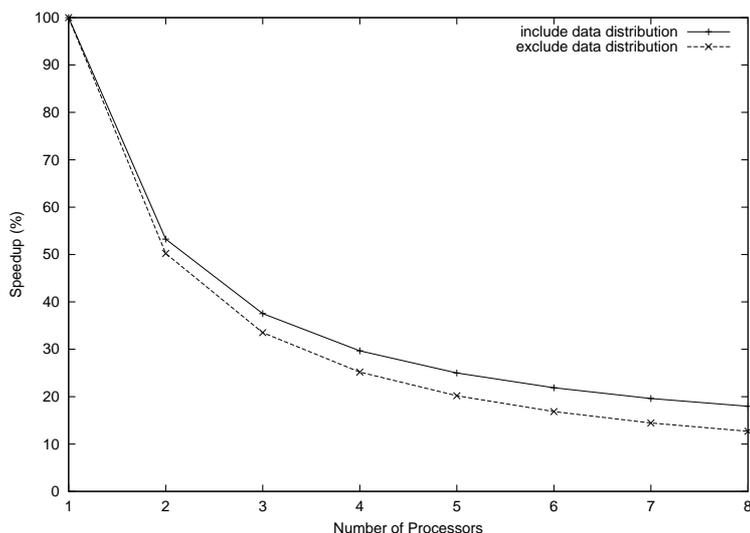


Fig. 5. Execution time of *tagmatch* on a PC cluster.

for distributing data (using inter-processor communications) to each processor is included, while the dotted line shows the speedup when the time for data distribution is excluded.

With both programs, there was linear speedup when the time for data distribution was excluded, as expected. With *tagmatch*, even when data distribution was included, there was almost linear speedup as the number of processors was increased. With *lineofsight*, the parallelization effect saturated. This saturation was due to the cost for data distribution of *lineofsight*, which is rather high because of the size of each element (a pair of two short integers) in its input list. This data distribution problem, however, does not detract from the effectiveness of `accumulate`:

- In practical applications, `accumulate` is likely to be used as a component of bigger programs. This means the input to `accumulate` will be the resulting list (array) of the previous computation, and its elements will have already been produced in the local memory of each processor. There will thus be no need to distribute data among processors before calling `accumulate`.
- If the input data to `accumulate` is provided by a filesystem like NFS, which is shared by all processors, each processor can read its assigned data into local memory by a fileaccess without data distribution.

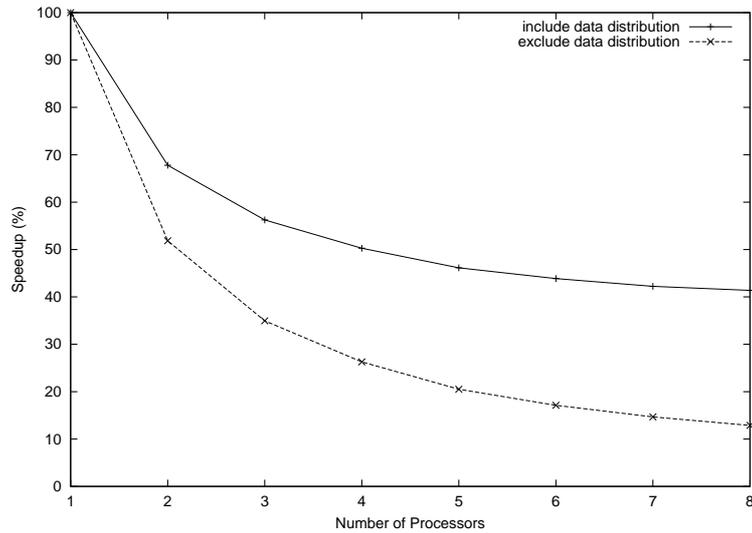


Fig. 6. Execution time of *lineofsight* on a PC cluster.

- While we used a distributed memory system in our experiments, which made data distribution unavoidable, we can expect a significant speedup from using `accumulate` in a shared memory environment, as shown by the results of the dotted line in Fig. 6,

This data distribution problem is true for all the skeletal programming systems proposed so far. In fact, many systems have ways to describe data distribution. For example, P3L<sup>(5,6)</sup> provides collective operations such as `scatter` for data parallel computations; Skil<sup>(9,10)</sup> and the skeleton library by Kuchen<sup>(11)</sup> have distributed data structures. While we directly use MPI primitives such as `MPI_Send` for data distribution, it would be better to provide abstracted libraries (or skeletons), which is left for future work.

To evaluate the efficiency of programs using `accumulate` compared to that of programs not using `accumulate`, we executed three programs for the same problem:

- the best sequential program as a baseline,
- the best parallel program without using skeletons, and
- a parallel program using `accumulate`.

We used the line-of-sight problem (with data size 500,000) because the algorithm in these three programs is the same, enabling us to identify the

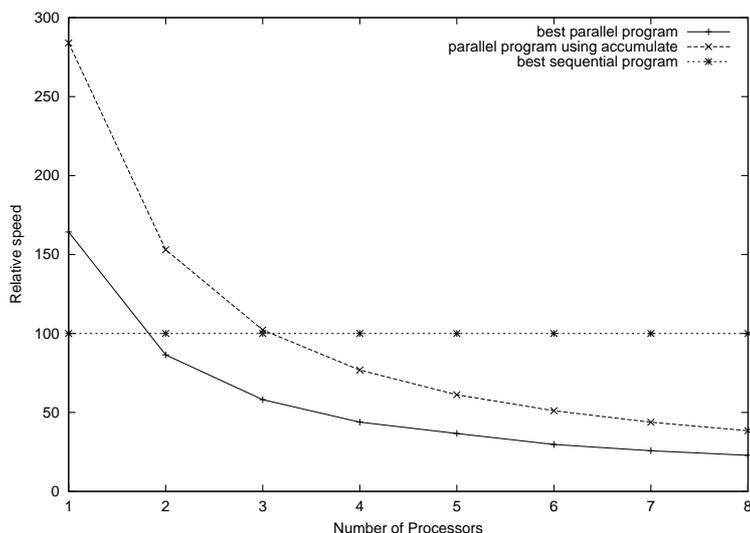


Fig. 7. Execution times of three programs for line-of-sight problem on a PC cluster.

differences based on the coding style of these programs. The best parallel program uses MPI primitives for inter-processor communication. The program using `accumulate` was compiled with inlining of the function calls of `g`, `p`, `oplus`, `q`, and `otimes` within `accumulate`.

The execution times (data distribution excluded) are shown in Fig. 7, with that of the best sequential program normalized to 100.

With more than three processors, the parallel program using `accumulate` was faster than the best sequential one. Compared to the best parallel program, the parallel program using `accumulate` was fast enough — the execution time was less than 1.8 times as long as that of the best parallel program. Using the `accumulate` function shortens the time for developing parallel programs, and the developed programs can be executed in a parallel environment in which speedup can be expected as the number of processors is increased.

The results of our experiments show the effectiveness of the `accumulate` skeleton in constructing efficient parallel programs.

## 6. CONCLUSION

We have described a new data parallel skeleton, `accumulate`, in parallel programming. It is applicable to a wide range of recursive functions. It

abstracts a good selection and combination of BMF skeletons, and programmers need not know the internal details of the skeleton.

We implemented `accumulate` using the MPI library. The implementation is based on the result of applying the fusion transformation to `acaccumulate`, which eliminates unnecessary intermediate data structures and exploits an extended version of Blelloch's algorithm. Since `accumulate` is efficiently implemented, recursive functions in the `accumulate` skeleton perform well in a parallel environment.

Although we limited our discussion to the list data type, the diffusion theorem can be extended to trees<sup>(19)</sup> and other general recursive data types. We can thus develop new parallel skeletons to match the data type of interest. We plan to implement such skeletons in an efficient way to make them more practical for parallel programming.

## ACKNOWLEDGMENTS

We are grateful to Yasuichi Nakayama and Kazuki Hyoudou of the University of Electro-Communications for providing the computing facilities and environment for the FireCluster.

## REFERENCES

1. M. Cole, *Algorithmic Skeletons: a Structured Approach to the Management of Parallel Computation*, Research Monographs in Parallel and Distributed Computing, Pitman (1989).
2. J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While, Parallel Programming using Skeleton Functions, *Proc. of the Conference on Parallel Architectures and Reduction Languages Europe (PARLE'93)*, Lecture Notes in Computer Science, Vol. 694, pp. 146–160, Springer-Verlag (1993).
3. D. B. Skillicorn, *Foundations of Parallel Programming*, Cambridge Series in Parallel Computation 6, Cambridge University Press (1994).
4. J. Darlington, Y. Guo, H. W. To, and J. Yang, Parallel Skeletons for Structured Composition, *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pp. 19–28 (1995).
5. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, P3L: A Structured High Level Programming Language and its Structured Support, *Concurrency: Practice and Experience*, 7(3):225–255 (1995).
6. M. Danelutto, F. Pasqualetti, and S. Pelagatti, Skeletons for Data Parallelism in P3L, *Proc. 3rd International Euro-Par Conference (Euro-Par'97)*, Lecture Notes in Computer Science, Vol. 1300, pp. 619–628, Springer-Verlag (1997).
7. G. E. Blelloch, Scans as Primitive Operations, *IEEE Trans. on Computers*, 38(11):1526–1538 (1989).
8. G. E. Blelloch, *NESL: a Nested Data-parallel Language*, Technical Report CMU-CS-95-170, Carnegie Mellon University (1995).
9. G. H. Botorog and H. Kuchen, Skil: An Imperative Language with Algorithmic Skeletons, *Proc. 5th International Symposium on High Performance Distributed Computing (HDPC'96)*, pp. 243–252 (1996).

10. G. H. Botorog and H. Kuchen, Efficient High-Level Parallel Programming, *Theoretical Computer Science*, **196**(1–2):71–107 (1998).
11. H. Kuchen, A Skeleton Library, *Proc. 8th International Euro-Par Conference (Euro-Par2002)*, *Lecture Notes in Computer Science*, Vol. 2400, pp. 620–629, Springer-Verlag (2002).
12. H. Kuchen and M. Cole, The Integration of Task and Data Parallel Skeletons, *Proc. 3rd International Workshop on Constructive Methods for Parallel Programming (CMPP2002)*, pp. 3–16 (2002).
13. eSkel Home Page, <http://homepages.inf.ed.ac.uk/mic/eSkel/>.
14. F. A. Rabhi and S. G. (Eds), *Patterns and Skeletons for Parallel and Distributed Computing*, Springer-Verlag (2002).
15. High Performance Fortran Forum, High Performance Fortran Language Specification (1993).
16. R. Bird, An Introduction to the Theory of Lists, *Proc. NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*, pp. 5–42 (1987).
17. D. B. Skillicorn, *The Bird-Meertens Formalism as a Parallel Model*, NATO ARW ‘Software for Parallel Computation’ (1992).
18. Message Passing Interface Forum, <http://www.mpi-forum.org/>.
19. Z. Hu, M. Takeichi, and H. Iwasaki, Diffusion: Calculating Efficient Parallel Programs, *Proc. 1999 ACM SIGPLAN International Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM’99)*, BRICS Notes Series NS-99-1, pp. 85–94 (1999).
20. R. Bird, *Introduction to Functional Programming using Haskell*, Prentice-Hall, New York (1998).
21. A. Gill, J. Launchbury, and S. P. Jones, A Short Cut to Deforestation, *Proc. 1993 Conference on Functional Programming Languages and Computer Architecture (FPCA’93)*, pp. 223–232, ACM Press (1993).
22. Z. Hu, H. Iwasaki, and M. Takeichi, Deriving Structural Hylomorphisms from Recursive Definitions, *Proc. 1996 International Conference on Functional Programming (ICFP’96)*, pp. 73–82, ACM Press (1996).
23. Z. Hu, H. Iwasaki, and M. Takeichi, An Accumulative Parallel Skeleton for All, *Proc. 2002 European Symposium on Programming, Lecture Notes in Computer Science*, Vol. 2305, pp. 83–97, Springer-Verlag (2002).
24. E. Meijer, M. Fokkinga, and R. Paterson, Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, *Proc. 1991 Conference on Functional Programming Languages and Computer Architecture (FPCA’91)*, *Lecture Notes in Computer Science*, Vol. 523, pp. 124–144, Springer-Verlag (1991).
25. S. Adachi, H. Iwasaki, and Z. Hu, Diff: A Powerful Parallel Skeleton, *Proc. 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’2000)*, pp. 2175–2181, CSREA Press (2000).
26. W.-N. Chin, A. Takano, and Z. Hu, Parallelization via Context Preservation, *Proc. 1998 IEEE Computer Society International Conference on Computer Languages (ICCL’98)*, pp. 153–162, IEEE Press (1998).
27. R. Shirasawa, Z. Hu, and H. Iwasaki, Diffusion after Fusion — Deriving Efficient Parallel Algorithms —, *Proc. 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’2001)*, pp. 735–741, CSREA Press (2001).
28. K. Hyoudou, R. Ozaki, and Y. Nakayama, A PC Cluster System Employing the IEEE 1394, *Proc. 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS’2002)*, pp. 489–494, ACTA Press (2002).