# Diff: A Powerful Parallel Skeleton

Seiichi Adachi, Hideya Iwasaki, Zhenjiang Hu Department of Information Engineering, the University of Tokyo 7–3–1 Hongo, Bunkyo-ku, Tokyo 113–8656 Japan

Abstract Skeleton parallel programming encourages programmers to build a parallel program from ready-made components for which efficient implementations are known to exist, making both the parallel program development and the parallelization process easier. However, programmers often suffer from the difficulty to choose a proper combination of parallel primitives so as to construct efficient parallel programs. To overcome this difficulty, we propose a new powerful parallel skeleton diff derived from the diffusion theorem, showing how it can be used to naturally code efficient solutions to problems, and how it can be efficiently implemented in parallel using MPI (Message Passing Interface).

*Keywords:* Parallel Skeleton, Bird–Meertens Formalism, Program Transformation, MPI.

## 1 Introduction

Parallel programming has proved to be a difficult task, 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 [4, 5, 8]. Examples of skeletons are *forall* in High Performance Fortran [6], the apply-tocall and scan in NESL [3], and a fixed set of higher order functions such as map, reduce and scan in BMF [9]. In this skeleton approach, programmers are encouraged to build a parallel program from ready-made components whose efficient implementations are known to exist, not being concerned with the lower level details of the implementation.

However, developing efficient parallel pro-

grams still remains as a big challenge. Programmers often find it hard to choose proper parallel primitives and to integrate them well in order to develop efficient parallel programs, especially when the given problems are a bit complicated.

As an example, consider the problem of checking whether tags are well matched or not in a document written in XML (eXtensible Markup Language). This problem is of practical interest, but design of an efficient  $O(\log N)$  parallel program using parallel skeletons is not easy, where N denotes the number of separated words in the document.

In this paper, we shall propose a new powerful parallel skeleton, called diff, which attains the following new features.

- First, it is *general* enough to cover all existing parallel skeletons as in the BMF parallel model [9]. In other words, the existing parallel skeletons can be considered as special cases of the new skeleton.
- Second, it is more *natural* to describe algorithms with more complicated dependency than the existing skeletons like scans [3].
- Third, it has a nice cost model, and can be *efficiently* implemented in parallel.

The organization of this paper is as follows. We shall review the existing parallel skeletons and the diffusion theorem in Section 2. After defining our parallel skeleton by abstracting parallel programs derivable by the diffusion theorem in Section 3, we show how to implement it in parallel using MPI (Message Passing Interface) in Section 4. An experimental result is presented in Section 5. We conclude in Section 6.

#### 2 The Diffusion Theorem

In this section, we briefly review the notational conventions and some basic concepts in Bird-Meertens Formalism [1, 8] (BMF for short), some related results, particularly the diffusion theorem on which our new skeleton is constructed.

*Function application* is denoted by a space and the argument which may be written without brackets. Thus f a means f (a). Functions are curried, and application associates to the left. Thus f a b means (f a) b. Function application binds stronger than any other operator, so  $f \ a \oplus b$  means  $(f \ a) \oplus b$ , and not  $f(a \oplus b)$ . Function composition is denoted by a centralized circle  $\circ$ . By definition, we have  $(f \circ g) a = f (g a)$ . Infix binary operators will often be denoted by  $\oplus$ ,  $\otimes$ ,  $\odot$  and can be *sectioned*; an infix binary operator like  $\oplus$  can be turned into unary or binary functions 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.

The most important skeletons in BMF are map, reduce and scan. The map is the operator which applies a function to every element in a list. Informally, we have

map 
$$f [x_1, ..., x_n] = [f x_1, ..., f x_n].$$

The reduce is the operator which collapses a list into a single value by repeated applications of some associative binary operator. Informally, for an associative binary operator  $\odot$ , we have

reduce (
$$\odot$$
) [ $x_1, x_2, \dots, x_n$ ]  
=  $x_1 \odot x_2 \odot \dots \odot x_n$ .

The scan is the operator that accumulates all intermediate results for computation of reduce.

Informally, for an associative binary operator  $\odot$  with its unit  $\iota_{\odot}$ , we have

scan (
$$\odot$$
)  $[x_1, x_2, \dots, x_n]$   
=  $[\iota_{\odot}, x_1, x_1 \odot x_2, \dots, x_1 \odot x_2 \odot \cdots \odot x_n]$ .

Diffusion is a transformation turning a recursive definition into a composition of our higher order functions, namely map, reduce and scan.

**Theorem (Diffusion** [7]) Given a function hdefined in the following recursive form:

$$h [] c = g_1 c$$
  

$$h (x : xs) c = k (x, c) \oplus h xs (c \otimes g_2 x).$$

If  $\oplus$  and  $\otimes$  are associative and have units, then h can be diffused into the following form.

$$\begin{array}{l} h \ xs \ c = \mathsf{reduce} \ (\oplus) \ (\mathsf{map} \ k \ as) \oplus g_1 \ b \\ \mathsf{where} \\ bs \ + \ [b] = \mathsf{map} \ (c \otimes) \ (\mathsf{scan} \ (\otimes) \ (\mathsf{map} \ g_2 \ xs)) \\ as = \ zip \ xs \ bs \end{array}$$

Note that the list concatenation operator ++ is used as a *pattern* in right hand side of the above equation. Our proposed diffusion transformation has the following two features. First, the diffusion transformation can be applied to a wide class of recursive functions of interest. Second, the resultant parallel program is effi*cient*, in the sense that if the original program uses O(N) sequential time, then the derived parallel one takes at most  $O(\log N)$  time.

To see how the diffusion theorem works in practice, consider a simple problem of eliminating smaller elements. An element is said to be *smaller* if it is less than some element before itself in the list. For example, for the list [1, 4, 2, 3, 5, 7], 2 and 3 are smaller elements, and thus the resultant list is [1, 4, 5, 7]. This problem can be solved directly; scan the list from left to right and eliminate every element which is less than the maximum of the scanned elements. That is,

$$se[]c = []$$
$$se(x : xs)c$$

= if x < c then se xs c else [x] + se xs x.

The second equation is not in the form where the theorem can be applied. A simple transformation of merging two recursive calls into a single one soon gives the following equation.

se (x : xs) c= (if x < c then [] else [x]) ++ se xs (if x < c then c else x)

Now matching the recursive definition of se with that in the diffusion theorem yields:

```
\begin{array}{l} se \ xs \ c = \mathsf{reduce} \ (\oplus) \ (\mathsf{map} \ k \ as) \oplus g_1 \ b \\ \mathsf{where} \\ bs \ + \ [b] = \mathsf{map} \ (c \otimes) \ (\mathsf{scan} \ (\otimes) \ (\mathsf{map} \ g_2 \ xs)) \\ as = zip \ xs \ bs \\ p \oplus q = p \ + q \\ c \otimes a = \mathsf{if} \ a < c \ \mathsf{then} \ c \ \mathsf{else} \ a \\ k \ (x, c) = \mathsf{if} \ x < c \ \mathsf{then} \ [] \ \mathsf{else} \ [x] \\ g_1 \ c = [] \\ g_2 \ x = x. \end{array}
```

Consequently, we have come to an efficient parallel algorithm for this problem.

# 3 The Diff Skeleton

In this section, we shall give the definition of our new skeleton diff, named after its underlying theorem, and demonstrate how it can be used to describe algorithms in a natural way.

#### Definition (Diff Parallel Skeleton)

diff  $(\oplus)$   $(\otimes) k g_1 g_2 xs c$ = reduce  $(\oplus)$  (map k as)  $\oplus g_1 b$ where  $bs ++ [b] = map (c \otimes) (scan (\otimes) (map g_2 xs))$ as = zip xs bs

where  $\oplus$  and  $\otimes$  are associative operations with units.  $\hfill\blacksquare$ 

Diff is a higher-order function which describes a general pattern of efficient parallel programs. Its definition comes directly from the diffusion theorem, abstracting important operators and functions out of the body of the new definition of h.

Returning to the example of se in Section 2, we can easily code it in terms of diff as follows.

se 
$$xs c = diff (\oplus) (\otimes) k g_1 g_2 xs c$$
  
where  
 $p \oplus q = p + + q$   
 $c \otimes a = if a < c$  then  $c$  else  $a$   
 $k (x, c) = if x < c$  then  $[]$  else  $[x]$   
 $g_1 c = []$ 

 $g_2 x = x$ 

Though the definition of diff might seem to be a bit complicated, it is quite adequate for a general skeleton in parallel programming for the following reasons.

- Since diff is directly derived from the consequence of the diffusion theorem, many natural recursive functions whose definitions have the form of h in the theorem can be expressed in terms of diff.
- In order to write a parallel program using diff, programmers only need to find suitable actual parameters given to diff. In many cases, it is easy to find k and  $g_1$ , because they are functions applied to each element of an input list. Therefore, programmers can focus on finding suitable associative operators  $\oplus$  and  $\otimes$  together with  $g_2$ .
- Although the definition of diff looks rather complicated, programmers need not know in detail how primitive skeletons such as map, reduce, and scan are combined together. In other words, diff provides an abstraction of a *good* combination of primitive skeletons, solving the problem of the skeleton approach, as pointed out in the introduction.
- Diff has an efficient implementation in parallel environment. In the following sections, we present an implementation using the MPI library and some experimental results.

For a more complicated and practical example, recall the tag matching problem in the introduction. It would be difficult to choose adequate BMF skeletons to develop its efficient parallel program. But with the diffusion theorem, we can derive systematically a parallel algorithm using diff in almost the same way as described in [7].

In fact, we can start with a naive sequential program to solve this problem using a stack. When an open tag is encountered, it is pushed onto the stack. In the case of a close tag, first it is compared with the top of the stack and if it corresponds, then the corresponding open tag is popped off from the top of the stack. This straightforward program tm can be expressed recursively with an accumulation parameter in the following way.

$$\begin{split} tm \ [] \ cs &= isEmpty \ cs \\ tm \ (x : xs) \ cs \\ &= \text{if} \ isOpen \ x \ \text{then} \ tm \ xs \ (push \ x \ cs) \\ &\text{else} \ \text{if} \ isClose \ x \ \text{then} \ notEmpty \ cs \ \land \\ & match \ x \ (top \ cs) \ \land \ tm \ xs \ (pop \ xs) \\ &\text{else} \ tm \ xs \ cs \end{split}$$

Here, for simplicity, XML document file is supposed to be pre-processed and *tm* receives as input a list of separated tags and words. After some steps for finding both suitable associative operators and an adequate representation of a stack [7], we are able to apply the diffusion theorem to get the following efficient parallel program in terms of diff. Since space is limited, we show only the final result.

$$\begin{array}{l} tm \, xs \, cs = {\rm diff} \, (\wedge) \, (\otimes) \, k \ isEmpty \, g_2 \, xs \, cs \\ \text{where} \\ k \, (x, \, cs) \\ = {\rm if} \ isOpen \, x \ {\rm then} \ True \\ {\rm else} \ if \ isOpen \, x \ {\rm then} \ True \\ g_2 \, x = {\rm if} \ isOpen \, x \ {\rm then} \ ([x], 1, 0) \\ {\rm else} \ f \ isOpen \, x \ {\rm then} \ ([x], 1, 0) \\ {\rm else} \ if \ isOpen \, x \ {\rm then} \ ([x], 0, 1) \\ {\rm else} \ ([], 0, 0) \\ (s_1, n_1, m_1) \otimes (s_2, n_2, m_2) \\ = {\rm if} \ n_1 \leq m_2 \ {\rm then} \ (s_2, n_2, m_1 + m_2 - n_1) \\ {\rm else} \\ (s_2 \ + \ drop \, m_2 \, s_1, \ n_1 + n_2 - m_2, \ m_1) \end{array}$$

Function  $drop \ n \ xs$  is to drop the first n elements from list xs. In the above definition, a stack is represented as a triple: 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 accumulation parameter cs of tm is the empty stack ([], 0, 0).

# 4 An Implementation of Diff

We implemented diff using MPI, a standard parallel library widely used for parallel pro-

gramming from massively parallel computers to PC clusters.

## 4.1 The Algorithm

In order to give an efficient implementation of diff, it is necessary to *fuse* (or *merge*) as many functional compositions as possible without sacrificing inherent parallelism, by exploiting some techniques for fusion transformation [2]. By this transformation, we can eliminate intermediate data structures passed through composition to gain efficiency. In the definition of diff, the following parts are taken into consideration.

- By using scanl which can have an initial value (seed) of scanning: scanl (⊙) e [x<sub>1</sub>, x<sub>2</sub>,..., x<sub>n</sub>] = [e, e ⊙ x<sub>1</sub>, e ⊙ x<sub>1</sub> ⊙ x<sub>2</sub>, ..., e ⊙ x<sub>1</sub> ⊙ x<sub>2</sub> ⊙ ··· ⊙ x<sub>n</sub>], we can fuse map (c ⊗) (scan (⊗) ys) into scanl (⊗) c ys.
- By using map2, a natural extension of map, which traverses two lists simultaneously:

 $\begin{array}{l} \max 2 \ f \ [x_1, x_2, \dots, x_n] \ [y_1, y_2, \dots, y_n] \\ = \ [f \ (x_1, y_1), \ f \ (x_2, y_2), \dots, \ f \ (x_n, y_n)], \\ \text{we can avoid } zip \text{ operation.} \end{array}$ 

• By using reducer, a kind of reduction which can also have an initial value (seed) of its operation:

reducer  $\odot e[x_1, x_2, \ldots, x_n]$ 

 $= x_1 \odot x_2 \odot \cdots \odot x_n \odot e,$ 

we are able to fuse reduce  $(\oplus)$   $ys \oplus e$  into reducer  $(\oplus) e ys$ .

Therefore the definition of diff after these fusion transformations becomes:

Our implementation is based on the above definition. Let N be the number of elements of the entire input list, and P be the number of processors. We assume that these elements are divided into P sublists, and each sublist is distributed beforehand to the corresponding processor. So each processor has a list of

length N/P. Also we assume that each processor is assigned an integer between 0 to P-1 called processor identifier (PID). According to the distribution of the input list, we can think a tree structure of all processors; each internal node of this tree is assigned to the same processor as that of its left child (Figure 1(a)).

Implementations of map, map2, reducer are straightforward. For map and map2, each processor simply traverses its list (represented as an array) sequentially, hence they need O(N/P) time if computing each application of the mapped function is in constant time. For  $\operatorname{reducer}(\odot)e$ , each processor first reduces its local list and gets its local value, then according to the tree structure, upwardly sweeps the values reducing by  $\odot$ , by inter-processor communication. After the upward sweep, processor 0 has the resultant value v of the sweep, and calculating  $v \odot e$  leads to the final value of reducer. The cost of local reduction is O(P/N) and that of the upward sweep is  $O(\log P)$ , so the total cost of reducer ( $\odot$ ) e is  $O(N/P + \log P)$  time, provided that  $\odot$  is a constant time operation.

To implement scanl  $(\odot) e$ , we adopted an algorithm based on Blelloch's [3], but with the following extensions. First, our algorithm allows the scan to accept an initial value e other than just the unit of  $\odot$ . This enhances the descriptive power of scanl without extra overhead. Second, taking data dependencies within diff into account, we carefully store in local memory of processors some values which will be consumed later by map2 and reducer. This is quite helpful to reduce total cost of interprocessor communication.

The detailed step of our algorithm to calculate scanl  $(\odot) e$  is as follows. To illustrate the idea concretely, we show the process of computing scanl (+) 50  $[1, \ldots, 8]$  in Figure 1.

- **Step 1** First, each processor scans the distributed list locally to form a scanned list. This phase can be executed totally in parallel and needs O(N/P) time.
- Step 2 Similar to the process of reducer, upward sweep of the final values from the scanned lists using  $\odot$  is executed. This



Figure 1: Implementation of scanl.

phase needs  $O(\log P)$  time. After this phase, processor 0 has the value of the scan of the input list (Figure 1(b)).

- Step 3 Starting from the root of the tree structure, downward sweep to the leaves is executed. Initially, the initial value eis put on the root (Figure 1(c)). At each step, every node applies  $\odot$  to its own and its left child's values and then passes the application result to its right child. The node also passes its own value to the left (Figure 1(d)~(e)).
- Step 4 Finally, each processor maps the resultant value obtained in the previous phase to the locally scanned list using the operator  $\odot$  (Figure 1(f)). Obviously this phase needs O(N/P) time.

Total cost of scanl  $(\odot) e$  is also  $O(N/P + \log P)$  time, provided that  $\odot$  can be carried out in constant time.

It is worth noting that the values of 53, 60, 71 in Figure 1(f) reside repeatedly in adjacent processors: 53 in PID  $0 \sim 1$ , 60 in PID  $1 \sim 2$ and 71 in PID  $2 \sim 3$ . In addition, processor 0 can have the final value (86) of the entire scanl operation by adding 50 (the initial value) to 36 (the value obtained after Step 2 of the algorithm). These kinds of memoization avoid extra inter-processor communication in diff, leading to an efficient implementation.

To sum up, our implementation of diff uses  $O(N/P + \log P)$  parallel time, but the constant factor is rather small.

#### 4.2 The Library in C++

We have implemented diff and other simpler skeletons in C++ along with MPI. With this implementation, we provide these skeletons as a library which programmers can use easily. To make the skeletons practical, they must be flexible (or *polymorphic*) enough to accept various types of user-defined functions.

For example, consider map which has the type of  $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ , where a and b are parameterized type variables. Type variables will be instantiated to some concrete types such as lnt and Char depending on problems. Rather than defining many library functions for map, such as a function for a = lnt and b = lnt and another function for a = lnt and b = Char, we provide a polymorphic function which can generate various instances according to the given types.

Another problem is that even though we can fix the types of a and b to some concrete types, say a and b are both lnt, the function used in map may have the type of int(\*)(int) or void(\*)(int\*,int\*) in C++. Library function for map must accept these variety of functions.

Since this kind of programming in C is not easy, we have decided to use the template and overloading mechanisms in C++. Template enables us to parameterize types in function definition, while overloading enables typedirected selection of adequate C++ function. By using these mechanisms, we are able to make the functions compatible to any combinations of data types. For example, the C++ function **cmap** which implements **map** skeleton is defined as follows.

In this definition, lists are supposed to be

represented as arrays.

We have implemented diff in the same way. One definition of the C++ function cdiff which implements the skeleton diff is as follows.

Note that there are other definitions of cdiff for overloading. In this way, the programmer only needs to define suitable actual parameters — five functions, pointers to processing data and data types.

In the current version of our library, the programmer needs to give correct data types of MPI::Datatype, which are used in communication. In the above definition, db and dc correspond to classes B and C respectively. Although they might be derivable from other parameters, in this version, it is the programmer's responsibility to give the corresponding data types.

## 5 An Experiment

To take a look at the effectiveness of the diff skeleton, we have conducted a preliminary experiment on the tag matching problem, whose parallel program in terms of diff is introduced in Section 3.

Figure 2 shows the results of the program executed on our PC cluster consisting of 3 multiprocessor (total of 9 processors) PC's connected by a 100Base-TX fast Ethernet. The MPI implementation used is MPICH<sup>1</sup> version 1.1.

The cost of parallelized tm is  $O(N/P + \log P)$ time. Due to the restrictions of our computer environment and to the fact that we are using a PC cluster, the communication time becomes

<sup>&</sup>lt;sup>1</sup>http://www-unix.mcs.anl.gov/mpi/mpich/



Figure 2: Experimental Result of tm.

a substantial factor.

We have executed *tm* with data size (the length of input list) of 50,000 whose levels of nested tags are within 10. It can be seen from the results in Figure 2 that the speed up is not linear. This is due to that fact we are using a PC cluster where the communication cost cannot be ignored. However, the execution time for 9 processes is approximately 14%, which shows that we have achieved a fairly good result. We believe that a result close to linear speed up could be achieved if it is executed on a shared memory parallel machine. This clearly shows the effectiveness of the diffusion skeleton in creating an efficient parallel program.

### 6 Conclusion

In this paper, we have proposed a new skeleton diff for parallel programming. Thanks to the underlying diffusion theorem, recursive functions with some specific form are easily turned into the form using diff skeleton. Since diff can be efficiently implemented, these recursive functions can enjoy good performance on parallel environments.

Although our discussion in this paper is limited to the list data type, our diffusion theorem can be extended to other recursive data types such as trees [7]. According to this extension, we can think of new parallel skeletons depending on the data type of interest. Our future work is to implement these skeletons in an efficient way to make them practically useful in parallel programming.

#### References

- Bird, R. An introduction to the theory of lists. In M. Broy, editor, *Logic of Pro*gramming and Calculi of Discrete Design, pages 5-42. Springer-Verlag, 1987.
- [2] Bird, R. Introduction to Functional Programming using Haskell. Prentice Hall, 1998.
- [3] Blelloch, G.E. Scans as primitive operations. *IEEE Trans. on Computers*, 38(11):1526-1538, 1989.
- [4] Cole, M. Algorithmic skeletons : a structured approach to the management of parallel computation. Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.
- [5] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N., Wu, Q. and While, R.L. Parallel programming using skeleton functions. In *Parallel Architectures & Languages Europe*. Springer-Verlag, 1993.
- [6] High performance Fortran language specification. In High Performance Fortran Forum, 1993.
- [7] Hu, Z., Takeichi, M. and Iwasaki, H. Diffusion: Calculating efficient parallel programs. Proc. 1999 ACM SIG-PLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pages 85–94, San Antonio, Texas, 1999. BRICS Notes Series NS-99-1.
- [8] Skillicorn, D.B. Foundations of Parallel Programming. Cambridge University Press, 1994.
- [9] Skillicorn, D.B. The Bird-Meertens Formalism as a Parallel Model. In NATO ARW "Software for Parallel Computation", 1992.