Diffusion after Fusion

– Deriving Efficient Parallel Algorithms –

Raku Shirasawa†, Zhenjiang Hu†, Hideya Iwasaki‡
† Department of Mathematical Engineering
The University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo 113–8656 Japan

‡ Department of Computer Science
The University of Electro-Communications
1–5–1 Chofugaoka, Chofu-shi, Tokyo 182–8585 Japan

Abstract *Parallel skeletons are ready-made components whose efficient implementations are known to exist. Using them, programmers can develop parallel programs without concerning about lower level of implementation. However, programmers may often suffer from the difficulty to choose a proper combination of parallel skeletons so as to construct efficient parallel programs. In this paper, we propose a new method, namely diffusion after fusion, which can not only guide a programmer to develop efficient parallel programs in a systematic way, but also be suitable for optimization of skeleton parallel programs. In this method, first we remove unnecessary intermediate data structure by fusion without being conscious of parallelism. Then we parallelize the fused program by diffusion. Using the Line-of-sight problem, this paper shows that the proposed method is quite effective.*

Keywords: Parallel Skeleton, Bird–Meertens Formalism, Fusion, Diffusion, Program Transformation.

1 Introduction

Parallel programming is notoriously difficult. Factors contributing to this difficulty include the complexity of concurrency, the effect of resource allocation on performance and the portability of the programs. To overcome this difficulty, skeleton programming [5, 6] has been proposed, where programmers are able to develop parallel programs using skeletons, higher order parallel forms, without being conscious of the low level architectures.

However, developing *efficient* parallel programs still remains as a hard task. There are two main obstacles. First, it is not obvious how to choose proper parallel skeletons and integrate them well in order to express given problems in terms of skeletons. This task often needs much experience, so the advantages of the skeleton programming may be wiped out. Second, it is often the case that programs in skeletons are easy to understand but may include inefficiency, so the use of skeletons in parallel programming does not always result in efficient programs. To resolve this problem, there have been many researches. In particular, Gorlatch [7] showed how to derive more efficient parallel programs from inefficient ones by ad hoc algebraic transformation over skeletons; an initial program is transformed to a more efficient parallel program, while each step has to preserve parallelism.

In this paper, we propose a new parallel programming method, namely, *diffusion after fusion*, enabling programmers to derive efficient parallel programs from simple but inefficient ones in a more systematic way. In this method, first we remove unnecessary intermediate data structures in a given program by fusion *without*

being conscious of parallelism. Then, we *diffuse* the obtained program into that in terms of parallel skeletons. Our main contribution can be summarized as follows.

- We give a more *systematic* method for both designing and optimizing parallel programs. Unlike the existing ad hoc methods like [7], our approach resolves the usual inconsistent problem between optimization by program transformation and preservation of parallelism, due to the separation of optimizing process and parallelizing process.
- Our method is *general* and can be applied to a wide class of problems. The first reason is that we use the function *foldr* as the basis of our transformation. It is one of the most general recursive forms for description of computation over a sequence of values. The second reason is that we use two general and powerful transformation rules, Fusion Theorem [2] and Diffusion Theorem [9], for optimization and parallelization.
- Derived programs are practically *efficient* concerning with implementation. We have implemented our derived algorithms using diff [1], a powerful parallel skeleton, and our experimental results show the promising of our method.

2 Preliminaries

In this section, we briefly review the notational conventions and some basic concepts in Bird Meertens Formalisms (BMF for short) [2, 8, 10], which is the parallel computation model of this research, and point out some related results which will be used in the rest of this paper.

2.1 Functions and Lists

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 $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. *Function composition* is denoted by a centralized circle $\circ$. By definition, we have $(f \circ g)\,a = f\,(g\,a)$. Function composition is an associative operator. Infix binary operators will often be denoted by $\oplus, \otimes$ and can be *sectioned*; an infix binary operator like $\oplus$ can be turned into unary on binary functions by

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

Besides, the following functions will be used later, and they are informally defined by:

$$\begin{aligned}
&\mathit{max}\;[x_1, x_2, \ldots, x_n] = x_1 \uparrow x_2 \uparrow \ldots \uparrow x_n\\
&\mathit{foldr}\;(\oplus)\;e\;[x_1, x_2, \ldots, x_n]\\
&\qquad = x_1 \oplus (x_2 \oplus (\ldots \oplus (x_n \oplus e)))\\
&\mathit{zip}\;[x_1, \ldots, x_n]\;[y_1, \ldots, y_n]\\
&\qquad = [(x_1, y_1), \ldots, (x_n, y_n)]\\
&\mathit{zipwith}\;f\;[x_1, \ldots, x_n]\;[y_1, \ldots, y_n]\\
&\qquad = [f\;x_1\;y_1, \ldots, f\;x_n\;y_n],
\end{aligned}$$

where operator $(\uparrow)$ returns the bigger of two operands.

Lists are finite sequences of values of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a, and $x +\!\!+ y$ for the concatenation of two lists x and y. 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 $[x] +\!\!+ xs$.

For other notations we follow those of the functional language Haskell.

2.2 Parallel Skeletons in BMF

BMF [2] is a nice architecture-independent parallel computation model [11], consisting of a small fixed set of specific higher order functions which can be regarded as parallel primitives suitable for parallel implementation. Three important higher order functions, i.e. parallel skeletons, are *map*, *reduce* and *scan*.

The *map* skeleton applies a function to every element in a list. Informally, we have

$$map\ k\ [x_1, x_2, \ldots, x_n] = [k\, x_1, k\, x_2, \ldots, k\, x_n].$$

The *reduce* skeleton collapses a list into a single value by repeated application of some associative binary operator. Informally, for an associative binary operator $\oplus$, we have

$$reduce\ (\oplus)\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n.$$

The *scan* skeleton accumulates all intermediate results for computation of reduce. Informally, for an associative binary operator $\oplus$ with its unit $\iota_\oplus$, we have

$$\begin{aligned} &scan\ (\oplus)\ [x_1, x_2, \ldots, x_n] \\ &\quad = [\iota_\oplus, x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n]. \end{aligned}$$

3 The Line-of-sight Problem: A Running Example

To explain our idea, we use the line-of-sight problem as our running example. The problem can be described as follows. Given a terrain map in the form of a grid of altitudes and an observation point, find which points are visible along a ray originating at the observation point. For instance, the input is a sequence (list) of points something like

$$[(1,1), (5,2), (2,3), (7,4), (19,5), (2,6)].$$

Each element in the sequence is a pair (a, d), where a stands for the altitude of the point and d for the distance from the observation point. Clearly, a point P of (a, d) is visible if its angle, $\arctan(a/d)$, is maximum among those of points before P. For the above example, we would like to have the result of

$$[True, True, False, False, True, False].$$

This problem is of much interest, and has been studied in [3] where a clever and efficient parallel program using the *scan* skeleton is given. However, it remains unclear how to obtain such a clever program, and in particular, how to obtain such kind of efficient parallel programs for other similar but different problems. In this paper, we would like to demonstrate how to systematically derive such kind of efficient parallel programs.

Using the BMF skeletons, one can write down the following initial and naive solution.

$$\begin{aligned} &los\ xs = isvisible\ ms\ as \\ &\quad \text{where } as = map\ angle\ xs \\ &\qquad\ \ ms = map\ max\ (inits\ as) \end{aligned}$$

where each point is checked whether its angle is greater than or equal to the corresponding maximum angle from the start point. The definitions of functions used in *los* are:

$$\begin{aligned} &isvisible\ ms\ as &&= zipwith\ (=)\ ms\ as \\ &angle\ (a, d) &&= \arctan\ (a/d) \\ &inits\ [\,] &&= [\,] \\ &inits\ (x : xs) &&= [x] : map\ (x :)\ (inits\ xs). \end{aligned}$$

This initial solution is clear and parallelism has been well specified by the skeletons *map* and *reduce*. However, according to the cost model in [3], it is rather inefficient with $O(N^3)$ work, $O(\log N)$ depth, and much data communication due to *inits*, where N denotes the number of points. By our approach, we can systematically derive an efficient one with just the same depth but only $O(N)$ work and very little data communication.

4 Fusion and Diffusion

Before proposing our approach, we explain two concepts, fusion and diffusion, which play important roles in our new method.

4.1 Fusion

Fusion [2] is a standard optimizing process whereby compositions of small functions are fused into a single one and unnecessary intermediate data structures are removed.

A recursive function f which consumes a list can be naturally defined with e and $\oplus$ as follows.

$$\begin{aligned} f\ [\,] &= e \\ f\ (x : xs) &= x \oplus f\ xs \end{aligned}$$

Such function f is expressed by $foldr\ (\oplus)\ e$.

It is well known that *foldr* is suitable for program transformation in that there is a general rule called *Fusion Theorem* [2].

Theorem (Fusion)

$$h\ e = e',\ h\ (x \oplus r) = x \otimes (h\ r)$$
$$\Longrightarrow h \circ foldr\ (\oplus)\ e = foldr\ (\otimes)\ e' \quad \blacksquare$$

As an example of the application of this theorem, let us consider the following function:

$$F\ xs = map\ max\ (inits\ (map\ g\ xs)).$$

Noticing that function $(map\ g)$ can be expressed as

$$map\ g = foldr\ (\oplus)\ [\,]$$
$$\text{where } x \oplus r = g\ x : r,$$

according to the theorem, we can get e' and $\otimes$ based on the following relation.

$$e' = map\ max\ (inits\ [\,])$$
$$x \otimes map\ max\ (inits\ r)$$
$$= map\ max\ (inits\ (x \oplus r)) \qquad (*)$$

It is obvious that $e' = [\,]$. Next,

$$\begin{array}{ll} & map\ max\ (inits\ (x \oplus r)) \\ = & \quad \{\text{def. of } \oplus\} \\ & map\ max\ (inits\ (g\ x : r)) \\ = & \quad \{\text{def. of } inits\} \\ & map\ max\ ([g\ x] : (map\ (g\ x :)\ (inits\ r))) \\ = & \quad \{\text{def. of } map, \text{ } map \text{ promotion}\} \\ & max\ [g\ x] : map\ (max \circ (g\ x :))\ (inits\ r) \\ = & \quad \{\text{def. of } max, \text{ } map \text{ promotion}\} \\ & g\ x : map\ (g\ x \uparrow)\ (map\ max\ (inits\ r)). \end{array}$$

In this calculation process, we applied a rule called *map promotion*. This rule is described as follows.

$$map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$$

Matching the last expression with LHS of $(*)$ soon gives:

$$a \otimes b = g\ a : map\ (g\ a \uparrow)\ (map\ max\ (inits\ b)).$$

As a result, we have got the fused version of F in terms of *foldr*.

$$F = foldr\ (\otimes)\ [\,]$$

4.2 Diffusion

Diffusion [9] is a transformation turning a recursive definition into a composition of parallel skeletons, namely *map*, *reduce* and *scan*.

Theorem (Diffusion) Given is a function h defined 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.

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

This 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 gives a good combination of parallel skeletons, and is *efficient*, in the sense that if the original program uses $O(N)$ sequential time, then the derived parallel one takes at most $O(\log N)$ parallel time. We give an example of an application of the theorem to program transformation in Section 5.

Based on the diffusion theorem, a parallel skeleton called diff [1] is defined.

Definition (diff)

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

5 Diffusion after Fusion

This section first describes the strategy of our approach, called *diffusion after fusion*, for obtaining an efficient parallel program in terms of parallel skeletons. Then we apply it to our running example, the Line-of-sight problem.

5.1 Derivation Strategy

Given a problem, *diffusion after fusion* can be summarized as follows.

Step 1: Specifying Initial Solution using Skeletons

We describe the problem straightforwardly in terms of the parallel skeletons of *map*, *reduce* and *scan*. In this step, rather than concerning with efficiency, we should concentrate on expressing concisely our solution, and specifying explicitly parallelism using skeletons.

Step 2: Eliminating Unnecessary Computation by Fusion

We eliminate unnecessary intermediate data passed between functions by applying the fusion transformation. This is an important optimization process for eliminating memory spaces as well as for reducing data communication. Notice that in this step, we do not care about parallelism. This is in sharp contrast to existing approaches like [7], where all transformation is required to keep parallelism.

Step 3: Parallelizing by Diffusion

We parallelize the program obtained in Step 2 by applying the Diffusion Theorem, resulting in an efficient parallel program using parallel skeletons. The difficulty lies in the derivation of associative binary operators as required in the theorem. Fortunately, with the context preservation approach [4], we can systematically derive such associative operators.

Step 4: Coding with diff Library

We code the parallel program obtained in Step 3 using some existing programming language depending on the environment. In our experiment discussed in Section 6, we used our MPI skeleton library for C++ [1]. This coding process is actually a straightforward translation.

5.2 Example

We shall demonstrate our approach by the running example Line-of-sight problem (*los*) in Section 3.

Step 1:

Initial solution has already been given in Section 3.

$$
\begin{array}{l}
los\ xs = isvisible\ ms\ as \\
\text{where} \\
\quad as = map\ angle\ xs \\
\quad ms = map\ max\ (inits\ as) \\
\quad isvisible\ ms\ as = zipwith\ (=)\ ms\ as \\
\quad angle\ (a,d) = \arctan\ (a/d) \\
\quad inits\ [\,] = [\,] \\
\quad inits\ (x:xs) = [x] : map\ (x:)\ (inits\ xs)
\end{array}
$$

Step 2:

We will try to eliminate unnecessary intermediate data structures in *ms* and *isvisible*. For *ms*, i.e., we want to fuse the definition of the following F_{ms} into a single *foldr*.

$$F_{ms}\ xs = map\ max\ (inits\ (map\ angle\ xs))$$

This fusion process has already been given in Section 4.1 with the following result.

$$
\begin{array}{l}
F_{ms} = foldr\ (\otimes)\ [\,] \\
\quad x \otimes r = angle\ x : map\ (angle\ x \uparrow)\ r
\end{array}
$$

Here, the obtained F_{ms} can be improved further, because *map* is called in each recursive step. In this case, we rewrite the function to another one with extra parameter, removing potential inefficiency.

$$F'_{ms}\ xs\ c = map\ (c \uparrow)\ (F'_{ms}\ xs)$$

By the generalization method [9], we can have

$$
\begin{array}{l}
F_{ms}\ xs = F'_{ms}\ xs\ 0 \\
F'_{ms}\ [\,]\ c = [\,] \\
F'_{ms}\ (x:xs)\ c \\
\qquad = (c \uparrow angle\ x) : F'_{ms}\ xs\ (c \uparrow angle\ x).
\end{array}
$$

Next, we fuse $los\ xs = isvisible\ ms\ as$. Because the function F'_{ms} has an extra argument, we change the definition of *los* accordingly to

$$
\begin{array}{l}
los\ xs = los'\ xs\ 0 \\
los'\ xs\ c \\
\quad = zipwith\ (=)\ (F'_{ms}\ xs\ c)\ (map\ angle\ xs).
\end{array}
$$

Now we can apply the fusion transformation similarly. Since the space is limited, we show only the following result which does not generate intermediate data structure.

$$
\begin{aligned}
&los\ xs = los'\ xs\ 0\\
&los'\ [\,]\ c = [\,]\\
&los'\ (x : xs)\ c\\
&\quad = (c \leq angle\ x) : los'\ xs\ (c \uparrow angle\ x).
\end{aligned}
$$

Step 3:

After obtaining an efficient *linear* algorithm for the problem, we proceed to parallelize los' using the Diffusion Theorem. To apply the theorem, we have to make sure that two operators ($:$ and $\uparrow$) in the above definition are, as required by the theorem, associative. The operator $\uparrow$ is obviously associative. For $:$, we can represent it using the associative operator $+\!\!+$. So the function los' becomes:

$$
\begin{aligned}
&los\ xs = los'\ xs\ 0\\
&los'\ [\,]\ c = [\,]\\
&los'\ (x : xs)\ c\\
&\quad = [c \leq angle\ x] +\!\!+ los'\ xs\ (c \uparrow angle\ x)
\end{aligned}
$$

Applying the diffusion theorem gives the following program using the parallel skeleton diff.

$$
\begin{aligned}
&los'\ xs\ c = \mathsf{diff}\ (+\!\!+)\ (\uparrow)\ k\ g_1\ g_2\ xs\ c\\
&\mathsf{where}\\
&\quad k\ (x, c) = [c \leq angle\ x]\\
&\quad g_1\ c = [\,]\\
&\quad g_2\ x = angle\ x.
\end{aligned}
$$

Step 4:

Now we code the obtained program using C++ and MPI Library. Thanks to the library functions [1] which implements the diff skeleton, we need only to code the arguments to diff: $g_1, g_2, k, \oplus$ and $\otimes$.

```
typedef pair {
    int altitude, distance;
};
int* oplus(int*,int*) {...}
int otimes(int,int) {...}
int k(pair x,int c) {...}
int g1(int c) {...}
int g2(pair x) {...}
```

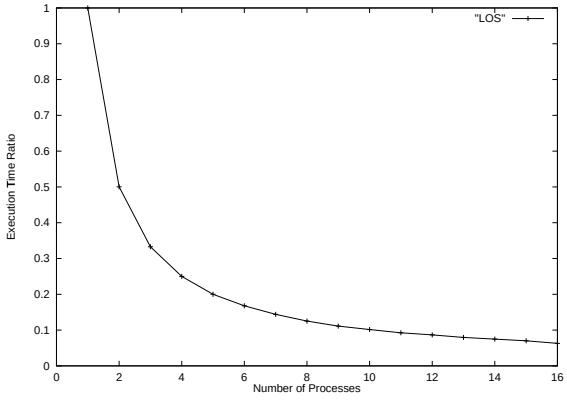

Figure 1: Experimental Result of *los*

```
int* los(pair *xs) {
    return los'(xs,0);
}
int* los'(pair *xs,int c) {
    return diff(oplus,otimes,k,
                g1,g2,xs,c);
}
```

6 An Experiment

To take a look at the effectiveness of a program derived by our method, we have conducted preliminary experiments on the line-of-sight problem given in Section 5.

Figure 1 shows the result of the program executed on SR2201, which is a HITACHI's massively parallel supercomputer (300MFLOPS/PE, 256MB/PE of memory size, 300MB/s of communication rate), using 1 to 16 processors. We have executed *los* with data size (the length of the input list) of 100,000. The linear speed-up shows the effectiveness of our method.

7 Another Example

Another example is the maximum segment sum (*mss*) problem. This problem has been studied by many researchers [10, 8, 7], and our result gives another efficient parallel program.

Starting with the following obviously correct solution to the problem:

$$
\begin{aligned}
&mss = max \circ map\ sum \circ segs \\
&segs\ [\,] = [\,] \\
&segs\ (x : xs) = inits\ (x : xs) +\!\!+\ segs\ xs,
\end{aligned}
$$

we can systematically derive the follow efficient parallel program.

$$
\begin{aligned}
&mss\ xs = y \text{ where } (x, y, z, w) = F''_{mss}\ xs \\
&F''_{mss}\ xs = reduce\ (\oplus)\ (map\ k\ xs) \\
&\text{where} \\
&\quad (x_1, x_2, x_3, x_4) \oplus (y_1, y_2, y_3, y_4) \\
&\qquad = ((y_1 + x_3) \uparrow x_1, (y_1 + x_4) \uparrow y_2 \uparrow x_2, \\
&\qquad\qquad y_3 + x_3, (x_4 + y_3) \uparrow y_4) \\
&\quad k\ x = (x, 0, x, 0)
\end{aligned}
$$

8 Conclusion

In this paper, we have proposed a new method for parallel programming. Thanks to the underlying fusion theorem, diffusion theorem and diff skeleton, many skeleton parallel programs can be transformed into more efficient ones.

Although this paper is about skeleton programs over list data structures, the fusion theorem and the diffusion theorem can be extended to other recursive data types such as trees [2, 9]. We are investigating how to deal with skeleton programs on trees efficiently.

References

[1] S. Adachi, H. Iwasaki, and Z. Hu. Diff: A powerful parallel skeleton. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Application*, pages 525–527 (Vol.4), Las Vegas, 2000. CSREA Press.

[2] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[3] Guy E. Blelloch. Scans as primitive operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[4] W.N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. *IEEE Computer Society International Conference on Computer Languages ICCL'98*, May 1998.

[5] M. Cole. *Algorithmic skeletons : a structured approach to the management of parallel computation.* Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[6] 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. In *Parallel Architectures & Languages Europe.* Springer-Verlag, June 93.

[7] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Annual European Conference on Parallel Processing, LNCS 1124*, pages 401–408, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

[8] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM TOPLAS*, 19(3):444–461, 1997.

[9] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–94, San Antonio, Texas, January 1999. BRICS Notes Series NS-99-1.

[10] D.B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

[11] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.