Efficient Parallel Skeletons for Nested Data Structures

Tomonari Takahashi[†], Hideya Iwasaki[‡], Zhenjiang Hu[†]

[†] 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 *Skeleton programming enables programmers to build parallel programs easier by providing efficient ready-made skeletons. In addition to primitive skeletons, diffusion skeleton has been proposed for rather complicated problems, abstracting a good combination of primitive skeletons. However, since they equally treat each element in target data, the existing skeletons are not always efficient for nested data structure where sizes of inner elements may be remarkably different. To resolve this problem, this paper proposes a new parallel skeleton, segmented diffusion, specially for nested data structure. The proposed skeleton is a nested version of the diffusion skeleton and similarly provides a proper combination of primitive nested skeletons. This paper also describes implementation of the skeleton and some experimental results, showing that the proposed skeleton is efficient and can be applied to a wider class of problems.*

Keywords: Skeleton Programming, Nested Data Structure, Segmented Skeleton, Bird–Meertens Formalism

1 Introduction

Parallel skeletons are a set of functions which make both programming and parallelization easier. By using parallel skeletons with efficient implementations, programmers can write efficient parallel programs without the deep knowledge of parallel algorithms and hardware. Programming using parallel skeletons is called *skeleton parallel programming* [5, 6, 9], or simply *skeleton programming*. Examples of such parallel skeletons are higher order functions of map, reduce and scan in BMF [8].

Unfortunately, parallel skeletons are often too primitive to describe programs solving a bit complicated problems. In order to make programs efficient, programmers are required to choose appropriate primitive skeletons and combine them in a suitable way. It is not an easy task, since programming is apt to become a process with much trial and error.

To overcome this problem, we proposed a parallel skeleton, namely *diffusion skeleton* diff [1]. This skeleton is derived from the *Diffusion Theorem* [7] and is defined in terms of primitive skeletons map, reduce and scan. It abstracts a 'good' combination of parallel primitives, and thanks to the underlying theorem, recursive functions defined naturally in some specific form over recursive data structure can be, under some conditions, turned into the form using the diff skeleton.

These skeletons including the diff are certainly efficient enough for *flat* data structure, e.g. a list of integers, by distributing almost the same number of elements to each processor. However, for the case of *nested* data structure, e.g. a list of lists of integers, simply distributing the same number of inner lists to each processor cannot lead to a good result. The reason is that when the target data is 'irregular' (the

sizes of inner lists are remarkably different), the amount of data assigned to each processor is not balanced. This load unbalance cancels the effect of parallelization, and as a result, the total computation time depends on that of the processor burdened with the maximum load.

To remedy this situation, skeletons specially for nested data structure[4], i.e. *segmented* skeletons, have been proposed. Segmented skeletons deal with nested data structure as flat one with some additional information about the original (nested) data (see Section 3). This flat representation enables us to distribute almost the same amount of elements of basic type to each processor equally, and to make use of efficient algorithms for flat data structure. An example of such skeleton is the *segmented scan* [3].

For segmented skeletons, however, the same problem as the case of skeletons for flat data structure remains: *how to* combine segmented skeletons to have a good effect of parallelization?

In this paper, we shall propose a new parallel segmented skeleton called *segmented diffusion* (s-diff), which abstracts a good combination of (primitive) segmented skeletons. The skeleton applies the diff skeleton to every element (element of depth 1) within a given nested data. We also give an implementation of s-diff, and describe some experimental results which show the effect of this new skeleton.

2 Diffusion Skeleton on Flat Data Structure

In this section, we give some notational conventions, and basic concepts in Bird-Meertens Formalism (BMF) [2, 9], which is the parallel computation model of this research. We also review the Diffusion Theorem [7] and the definition of the diffusion skeleton [1].

2.1 Notations

Function application is denoted by a space and the argument which may be written without brackets. 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 $(f \circ g)\, a = f\ (g\, a)$. Note that function composition is an associative operator. Infix binary operators will often be denoted by $\oplus$, $\otimes$, $\odot$, etc. 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, and denoted by sequentially listing each element separating by comma within square bracket, e.g. $[1, 2, 3]$. A list is either *empty*, or a *cons* of a value and a list. We write $[\,]$ for the empty list and $x : xs$ for cons whose first element is x and the rest is the list xs. We also write $xs +\!\!+ ys$ for the concatenation of two lists xs and ys. For example, $[1] +\!\!+ [2, 3]$ denotes a list with three elements $[1, 2, 3]$.

Given a binary operator $\oplus$ and two lists, we define the *element-wise* binary operator $\widetilde{\oplus}$. It applies $\oplus$ to corresponding elements of the two lists. Informally speaking,

$$[x_1, x_2, \ldots, x_n]\ \widetilde{\oplus}\ [y_1, y_2, \ldots, y_n] = [x_1 \oplus y_1,\ x_2 \oplus y_2,\ \ldots,\ x_n \oplus y_n].$$

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

The most important (primitive) skeletons in BMF are map, reduce and scan. Their informal definitions are:

$$\mathsf{map}\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1,\ f\ x_2,\ \ldots,\ f\ x_n]$$
$$\mathsf{reduce}\ (\oplus)\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$
$$\mathsf{scan}\ (\oplus)\ [x_1, x_2, \ldots, x_n] = [\iota_\oplus,\ x_1,\ x_1 \oplus x_2,\ \ldots,\ x_1 \oplus x_2 \oplus \cdots \oplus x_n].$$

where $\oplus$ is an associative binary operator whose unit is $\iota_\oplus$.

The length of the result of scan is $N + 1$ when its input list is of length N, but sometimes it is convenient to have the result of the same length. Scan′ and prescan [4] skeletons can be used for this purpose. Their informal definitions are:

$\mathsf{scan}'\ (\oplus)\ [x_1, x_2, \ldots, x_n]$
$= [x_1,\ x_1 \oplus x_2,\ \ldots,\ x_1 \oplus x_2 \oplus \cdots \oplus x_n]$
$\mathsf{prescan}\ (\oplus)\ [x_1, x_2, \ldots, x_{n-1}, x_n]$
$= [\iota_\oplus,\ x_1,\ x_1 \oplus x_2,\ \ldots,\ x_1 \oplus x_2 \oplus \cdots \oplus x_{n-1}]$.

It may be curious that x_n is not used at all in the result of prescan. However, from the implementation point of view, $x_1 \oplus x_2 \oplus \cdots \oplus x_n$ can be easily calculated as a side effect of computation of prescan.

For these skeletons, we have efficient parallel implementations. Let the size of input list be N, and the number of processors be P. Then map needs $O(N/P)$ time, reduce, scan, scan$'$ and prescan need $O(N/P + \log P)$ time [3, 1], assuming that calculation of f and $\oplus$ needs constant time.

2.2 The Diffusion Skeleton

Diffusion Theorem [7] describes a transformation rule from a recursive definition into a composition of 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.

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

This transformation can be applied to a wide class of recursive functions which naturally induct over a given list. Based upon this theorem, the diff parallel skeleton [1] is defined.

Definition (Diffusion Skeleton)

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

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

Clearly, the function h in the Diffusion Theorem can be represented in terms of the diff skeleton as:

$h\ xs\ c = \mathsf{diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ xs\ c$.

Diff is a higher-order function which describes a general pattern of efficient parallel programs. It gives a good combination of primitive skeletons map, reduce and scan, hiding the details how they are combined. Also, as described in [1], programs in terms of diff can be executed efficiently in parallel environment, in the sense that if the original program uses $O(N)$ sequential time, then the parallel one using diff takes at most $O(\log N)$ time.

3 Primitive Segmented Skeletons

In this section, we introduce a set of *primitive* segmented skeletons [4] for nested data structure. In the rest of this paper, we restrict our target nested data structure to a list of flat lists, that is, a list whose depth is two such as $[[1, 2], [3, 4, 5]]$. We call them *nested lists*.

To make segmented skeletons efficient, we represent a nested list using a flat list of pairs. Each element in this flat list is a pair of *flag*, a boolean value, and an element of inner list of the original nested list. If the element is the first of an inner list, *flag* is T, otherwise *flag* is F. We use the term *specification* as the nested list of depth two itself, and its *implementation* as the list of pairs described here.

For example, if the specification is

$[[x_1, x_2, x_3], [x_4, x_5], [x_6], [x_7, x_8]]$,

then its implementation is the following list of length 8.

$[(\mathsf{T}, x_1), (\mathsf{F}, x_2), (\mathsf{F}, x_3), (\mathsf{T}, x_4),$
$(\mathsf{F}, x_5), (\mathsf{T}, x_6), (\mathsf{T}, x_7), (\mathsf{F}, x_8)]$.

Using this implementation with a flat list, each processor can be assigned almost the same number of data elements, and therefore, reasonable load balancing can be achieved. For the above example, if there are four processors, they are assigned to $[(\mathsf{T}, x_1), (\mathsf{F}, x_2)]$, $[(\mathsf{F}, x_3), (\mathsf{T}, x_4)]$, $[(\mathsf{F}, x_5),$

$(\mathsf{T}, x_6)]$, and $[(\mathsf{T}, x_7), (\mathsf{F}, x_8)]$, respectively. Note that elements of the same inner list in the specification may be divided and assigned to more than one processors.

We have parallel skeletons for nested lists which correspond to map, reduce, scan family for flat lists. To each inner element within a given nested list, they apply corresponding skeletons for flat lists. As described above, we have two kinds of representations for nested list: specification and implementation. We put 'S-' to the names of segmented skeletons for specification, and 's-' for their implementations. For example, S-map is the segmented map for specification, and s-map is that for implementation which are used in the real programming

We show definitions of segmented map, scan, prescan and reduce below.

Segmented map

Segmented map applies map to each list within a given nested list.

- Specification:

  S-map $f\ [\,] = [\,]$
  S-map $f\ (xs : xss) =$ map $f\ xs$: S-map $f\ xss$.

- Implementation:

  s-map $f\ [\,] = [\,]$
  s-map $f\ ((b, x) : bxs) = (b,\ f\,x)$: s-map $f\ bxs$.

Segmented scan and prescan

Segmented scan and prescan apply scan′ and prescan respectively to each list within a given nested list.

- Specification:

  S-scan $(\oplus)\ [\,] = [\,]$
  S-scan $(\oplus)\ (xs : xss)$
  $=$ scan′ $(\oplus)\ xs$: S-scan $(\oplus)\ xss$
  S-prescan $(\oplus)\ [\,] = [\,]$
  S-prescan $(\oplus)\ (xs : xss)$
  $=$ prescan $(\oplus)\ xs$: S-prescan $(\oplus)\ xss$.

- Implementation:

  s-scan $(\oplus)\ bxs =$ scan′ $(\otimes)\ bxs\ \widetilde{\ominus}\ bxs$
  where $(b_1, x_1) \otimes (\mathsf{T}, x_2) = (\mathsf{T}, x_2)$
  $(b_1, x_1) \otimes (\mathsf{F}, x_2) = (b_1,\ x_1 \oplus x_2)$
  $(b_1, x_1) \ominus (b_2, x_2) = (b_1, x_2)$

  s-prescan $(\oplus)\ bxs = rshift\ \iota_\oplus$ (s-scan $(\oplus)\ bxs$).

The s-scan skeleton makes use of scan′ with the associative operator $\otimes$. This operator scans and combines the values in a given list, paying attention to flag values. If the T flag, the sign of the beginning of a new inner list, is encountered, it resets the scanning value. However, since scan′ with $\otimes$ collapses flag values, $\widetilde{\ominus}$ is necessary to restore correct flags.

In the implementation of s-prescan, *rshift* is used to shift elements of each inner list within the original nested list to the right direction, dropping the rightmost element and supplying $\iota_\oplus$ into the leftmost position. For example,

$$rshift\ \iota_\oplus\ [(\mathsf{T}, x_1), (\mathsf{F}, x_2), (\mathsf{T}, x_3), (\mathsf{F}, x_4)]$$
$$= [(\mathsf{T}, \iota_\oplus), (\mathsf{F}, x_1), (\mathsf{T}, \iota_\oplus), (\mathsf{F}, x_3)].$$

At the first sight, this operation seems to be complex, but under the condition that *rshift* is applied to the result of s-scan skeleton, it is simple and inexpensive since adjacent processors can share the 'border' value of s-scan.

Segmented reduce

Segmented reduce applies reduce to each list within a given nested list. Note that the resulting list must be flat one.

- Specification:

  S-reduce $(\oplus)\ [\,] = [\,]$
  S-reduce $(\oplus)\ (xs : xss)$
  $=$ reduce $(\oplus)\ xs$: S-reduce $(\oplus)\ xss$.

- Implementation:

  s-reduce $(\oplus)\ bxs = ext$ (s-scan $(\oplus)\ bxs$)
  where
  $ext\ [\,] = [\,]$
  $ext\ [(b, x)] = x$
  $ext\ ((b, x_1) : (\mathsf{T}, x_2) : bxs)$
  $= x_1 : ext\ ((\mathsf{T}, x_2) : bxs)$
  $ext\ ((b, x_1) : (\mathsf{F}, x_2) : bxs)$
  $= ext\ ((\mathsf{F}, x_2) : bxs)$.

Because the result of reduce is equal to the rightmost value of scan, function *ext* gathers only the rightmost values from flat representation of nested list. To do this, *ext* pays attention to the flag, and picks up values just before T flag together with the last value.

4 Segmented Diffusion Skeleton

As mentioned in Introduction, diff is certainly efficient for flat data structure, however, direct application of diff to irregular nested data structure cannot show their full power because of unbalance of the sizes of inner lists. In this section, we shall propose a new parallel skeleton, *segmented diffusion*, which can be applied to a wide class of problems, integrating primitive segmented skeletons.

As an example, assume that we are given a document composed of many paragraphs of various lengths. It contains many open and close tags like XML document, and each paragraph must be consistent, i.e. open and close tags must perfectly match within each paragraph. And assume that the document is represented as a list of lists xss, where each inner list correspond to each paragraph. If we have a function *tagmatch* which judges whether a given list (paragraph) has consistent open and close tags, then the problem is specified as map *tagmatch xss*. Since it is known that *tagmatch* can be expressed in terms of diff skeleton [7, 1], this problem can be seen as to map the diffusion skeleton to the entire document (nested list). The segmented diffusion skeleton is necessary to write a program for this kind of problem.

Segmented diffusion skeleton applies diff to every element of the nested list.

- Specification:

$$\begin{aligned}&\mathsf{S\text{-}diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ [\,]\ c = [\,]\\&\mathsf{S\text{-}diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ (xs : xss)\ c\\&= \mathsf{diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ xs\ c\\&\qquad\qquad : \mathsf{S\text{-}diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ xss\ c.\end{aligned}$$

Implementation of this segmented diffusion skeleton can be expressed as a combination of primitive segmented skeletons. We show the result first, and then give the underlying theorem of the implementation.

- Implementation:

$$\begin{aligned}&\mathsf{s\text{-}diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ bxs\ c\\&= \mathsf{s\text{-}reduce}\ (\oplus)\ (\mathsf{s\text{-}map}\ k\ bzs)\ \ \widetilde{\oplus}\ \ (\mathsf{map}\ g_1\ bs)\\&\quad \text{where}\\&\quad bys = \mathsf{s\text{-}map}\ (c\otimes)\\&\qquad\qquad (\mathsf{s\text{-}prescan}\ (\otimes)\ (\mathsf{s\text{-}map}\ g_2\ bxs))\\&\quad bs\ =\ \mathsf{map}\ (c\otimes)\\&\qquad\qquad (\mathsf{s\text{-}reduce}\ (\otimes)\ (\mathsf{s\text{-}map}\ g_2\ bxs))\\&\quad bzs\ =\ bxs\ \widetilde{\ominus}\ \ bys\\&\quad (b_1, x) \ominus (b_2, y) = (b_1, (x, y)).\end{aligned}$$

The s-diff can be efficient on parallel environment, since all the skeletons used in s-diff can be implemented efficiently, and they are well combined for implementation of the segmented diffusion skeleton.

This implementation is based on our follwing theorem.

Theorem (Segmented Diffusion)

$$\begin{aligned}&\mathsf{S\text{-}diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ xss\ c\\&= \mathsf{S\text{-}reduce}\ (\oplus)\ (\mathsf{S\text{-}map}\ k\ ass)\ \ \widetilde{\oplus}\ \ (\mathsf{map}\ g_1\ bs)\\&\quad \text{where}\\&\quad bss = \mathsf{S\text{-}map}\ (c\otimes)\\&\qquad\qquad (\mathsf{S\text{-}prescan}\ (\otimes)\ (\mathsf{S\text{-}map}\ g_2\ xss))\\&\quad bs\ =\ \mathsf{map}\ (c\otimes)\\&\qquad\qquad (\mathsf{S\text{-}reduce}\ (\otimes)\ (\mathsf{S\text{-}map}\ g_2\ xss))\\&\quad ass\ =\ \mathit{S\text{-}zip}\ \ xss\ bss\\&\quad \mathit{S\text{-}zip}\ [\,]\ [\,] = [\,]\\&\quad \mathit{S\text{-}zip}\ (ps : pss)\ (qs : qss)\\&\qquad = \mathit{zip}\ ps\ qs : \mathit{S\text{-}zip}\ pss\ qss.\end{aligned}$$

Proof: In this proof, for simplicity, we omit operators and functions ($\oplus$, $\otimes$, k, g_1 and g_2) and an accumulating parameter c in diff and S-diff. The proof proceeds in an inductive way.

- Base case ($xss = [\,]$).

  It is easy to show that S-diff $[\,] = [\,]$. Since the space is limited, we omit this caluculation process.

- Inductive case ($xss = xs' : xss'$).

  We show that

  $$\mathsf{S\text{-}diff}\ (xs' : xss') = \mathsf{diff}\ xs' : \mathsf{S\text{-}diff}\ xss'.$$

$$\begin{aligned}&\mathsf{S\text{-}diff}\ (xs' : xss')\\&= \{\text{new definition}\}\\&\quad \mathsf{S\text{-}reduce}\ (\oplus)\ (\mathsf{S\text{-}map}\ k\ ass)\ \ \widetilde{\oplus}\ \ (\mathsf{map}\ g_1\ bs)\\&\quad \text{where}\\&\quad bss = \mathsf{s\text{-}map}\ (c\otimes)\\&\qquad\qquad (\mathsf{S\text{-}prescan}\ (\otimes)\ (\mathsf{S\text{-}map}\ g_2\ (xs' : xss')))\\&\quad bs = \mathsf{map}\ (c\otimes)\\&\qquad\qquad (\mathsf{S\text{-}reduce}\ (\otimes)\ (\mathsf{S\text{-}map}\ g_2\ (xs' : xss')))\\&\quad ass = \mathit{S\text{-}zip}\ (xs' : xss')\ bss.\end{aligned}$$

Here, we proceed calculation of bss and bs separately.

$$
\begin{aligned}
bss &= \text{S-map}\ (c\otimes) \\
&\qquad (\text{S-prescan}\ (\otimes)\ (\text{S-map}\ g_2\ (xs' : xss'))) \\
&= \{\text{S-map, S-reduce, map}\} \\
&\quad \text{map}\ (c\otimes)\ (\text{prescan}\ (\otimes)\ (\text{map}\ g_2\ xs')) \\
&\quad : \text{s-map}\,(c\otimes)\,(\text{S-prescan}\,(\otimes)(\text{S-map}\,g_2\,xss')) \\
&= bs' : bss' \\
&\quad \text{where} \\
&\qquad bs' = \text{map}\,(c\otimes)\,(\text{prescan}\,(\otimes)\,(\text{map}\,g_2\,xs')) \\
&\qquad bss' = \text{s-map}\ (c\otimes) \\
&\qquad\qquad (\text{S-prescan}\,(\otimes)(\text{S-map}\,g_2\,xss')).
\end{aligned}
$$

$$
\begin{aligned}
bs &= \text{map}\ (c\otimes) \\
&\qquad (\text{S-reduce}\ (\otimes)\ (\text{S-map}\ g_2\ (xs' : xss'))) \\
&= \{\text{map, S-reduce, S-map}\} \\
&\quad c \otimes (\text{reduce}\ (\otimes)\ (\text{map}\ g_2\ xs')) \\
&\qquad : \text{map}\,(c\otimes)\,(\text{S-reduce}\,(\otimes)\,(\text{S-map}\,g_2\,xss')) \\
&= b'' : bs'' \\
&\quad \text{where} \\
&\qquad b'' = c \otimes (\text{reduce}\ (\otimes)\ (\text{map}\ g_2\ xs')) \\
&\qquad bs'' = \text{map}\ (c\otimes) \\
&\qquad\qquad (\text{S-reduce}\ (\otimes)\ (\text{S-map}\ g_2\ xss')).
\end{aligned}
$$

Next, we calculate ass.

$$
\begin{aligned}
ass &= \textit{S-zip}\ (xs' : xss')\ bss \\
&= \textit{S-zip}\ (xs' : xss')\ (bs' : bss') \\
&= \textit{zip}\ xs'\ bs' : \textit{S-zip}\ xss'\ bss' \\
&= as' : ass' \\
&\quad \text{where} \quad as' = \textit{zip}\ xs'\ bs' \\
&\qquad\qquad\ \ ass' = \textit{S-zip}\ xss'\ bss'.
\end{aligned}
$$

Then we return to the first equation.

$$
\begin{aligned}
&\text{S-reduce}\ (\oplus)\ (\text{S-map}\ k\ ass)\ \widetilde{\oplus}\ (\text{map}\ g_1\ bs) \\
&= \text{S-reduce}\ (\oplus)\ (\text{S-map}\ k\ (as' : ass') \\
&\qquad\qquad \widetilde{\oplus}\ (\text{map}\ g_1\ (b' : bs'')) \\
&= \{\text{map, S-map, S-reduce}\} \\
&\quad \text{reduce}\ (\oplus)\ (\text{map}\ k\ as') \oplus g_1\ b'' \\
&\qquad : \text{S-reduce}\,(\oplus)\,(\text{S-map}\,k\,ass')\ \widetilde{\oplus}\ \text{map}\,g_1\,bs''.
\end{aligned}
$$

Putting the above results together, we have:

$$
\begin{aligned}
&\text{S-diff}\ (\oplus)(\otimes)\ k\ g_1\ g_2\ (xs' : xss')\ c \\
&= \text{reduce}\ (\oplus)\ (\text{map}\ k\ as') \oplus g_1\ b'' \\
&\qquad : \text{S-reduce}\,(\oplus)\,(\text{S-map}\,k\,ass')\ \widetilde{\oplus}\ \text{map}\,g_1\,bs'' \\
&\quad \text{where} \\
&\qquad bs' = \text{map}\ (c\otimes) \\
&\qquad\qquad (\text{prescan}\ (\otimes)\ (\text{map}\ g_2\ xs')) \\
&\qquad bss' = \text{s-map}\ (c\otimes) \\
&\qquad\qquad (\text{S-prescan}\ (\otimes)\ (\text{S-map}\ g_2\ xss')) \\
&\qquad b'' = c \otimes (\text{reduce}\ (\otimes)\ (\text{map}\ g_2\ xs')) \\
&\qquad bs'' = \text{map}\ (c\otimes) \\
&\qquad\qquad (\text{S-reduce}\ (\otimes)\ (\text{S-map}\ g_2\ xss')) \\
&\qquad as' = \textit{zip}\ xs'\ bs' \\
&\qquad ass' = \textit{S-zip}\ xss'\ bss' \\
&= \{\text{inductive hypothesis and definition of diff}\} \\
&\quad \text{diff}\ xs'\ : \text{S-diff}\ xss'.
\end{aligned}
$$

Then S-diff skeleton has been proved to be equivalent to the specification. ∎

With the suitable conversion of data format, we can define s-diff skeleton described before.

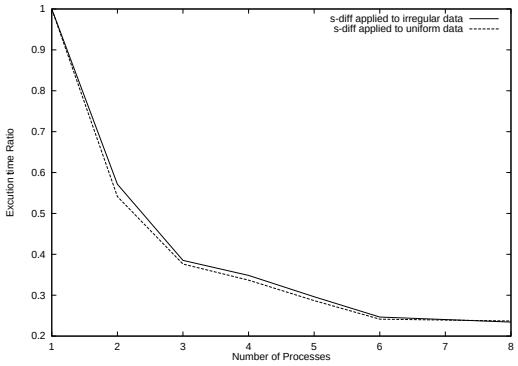

Figure 1: s-diff to uniform and irregular data

5 An experiment

To take a look at the effectiveness of the segmented diffusion, we made some preliminary experiments on the paragraph-wise tag matching problem in Section 4. We implemented the primitive (segmented) skeletons given in Section 3 and segmented diffusion s-diff by C language and MPI library. Our machine environment consists of two multiprocessor PCs: one has 4 processors (PentiumIII Xeon 550MHz) and the other has also 4 processors (PentiumIII Xeon 450MHz).

Test data of the entire document has 100 paragraphs with 5,000,000 characters. Figure 1 shows the results. The solid line represents the case where the sizes of paragraphs are quite unbalanced, and the broken line represents the case where the sizes of paragraphs are almost uniform. We can see that two lines are very

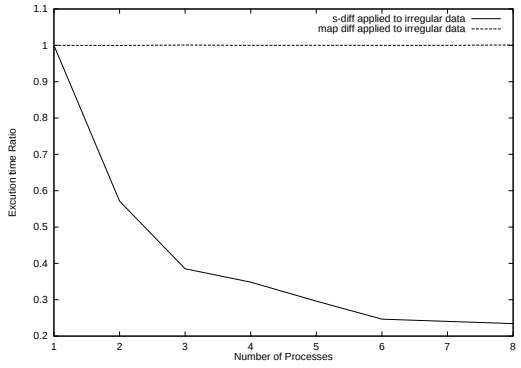

Figure 2: s-diff and map diff to irregular data

similar, showing that the proposed s-diff is efficient enough even though the input (nested) data is irregular.

To compare s-diff skeleton with a combination of map and diff skeleton for flat data structure, we have conducted another experiment. The input data is the same as that used in the experiment of the solid line in Figure 1, except that the original representation is used in this experiment. Same number of inner lists in the nested list are distributed to each processor without taking their sizes into account, and they are supplied to diff skeleton. The result is shown by a broken line in Figure 2. The total computation time does not decrease even though total number of processors increases. This fact suggests that the computation time depends on the paragraph of the largest size. The solid line is the same as that of Figure 1, but we plot it in Figure 2 for comparison.

These results clearly show the effectiveness of the proposed segmented diffusion skeleton.

6 Conclusion

In this paper, we have proposed a new segmented skeleton s-diff for nested data structure. Combined with other parallel segmented primitives, segmented diffusion can be applied to a wide class of problems on nested data structure. Since the proposed skeleton can be efficiently implemented, even though a given nested data is irregular, we can enjoy good performance on parallel environment. Our future work is to extend the s-diff skeleton to other recursive data types such as trees [7] to make the skeleton more useful in parallel programming.

References

[1] S. Adachi, H. Iwasaki, and Z. Hu. Diff: A powerful parallel skeleton. In *PDPTA 2000*, pages 525–527 (Vol.4), 2000. CSREA Press.

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

[4] G.E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[5] M. Cole. *Algorithmic skeletons : a structured approach to the management of parallel computation*. Research Monographs in Parallel and Distributed Computing, Pitman, 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 1993.

[7] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. *Proc. PEPM99*, pages 85–94, 1999. BRICS Notes Series NS-99-1.

[8] D.B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In *NATO ARW "Software for Parallel Computation"*, June 1992.

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