Parallel Processing Letters,
© World Scientific Publishing Company

AN ANALYTICAL METHOD FOR PARALLELIZATION OF RECURSIVE FUNCTIONS

JOONSEON AHN* and TAISOOK HAN
Division of Computer Science, Korea Advanced Institute of Science and Technology
Taejon, 305-701, Republic of Korea

Received (received date)
Revised (revised date)
Communicated by Christian Lengauer

ABSTRACT

Programming with parallel skeletons is an attractive framework because it encourages programmers to develop efficient and portable parallel programs. However, extracting parallelism from sequential specifications and constructing efficient parallel programs using the skeletons are still difficult tasks. In this paper, we propose an analytical approach to transforming recursive functions on general recursive data structures into compositions of parallel skeletons. Using static slicing, we have defined a classification of subexpressions based on their data-parallelism. Then, skeleton-based parallel programs are generated from the classification. To extend the scope of parallelization, we have adopted more general parallel skeletons which do not require the associativity of argument functions. In this way, our analytical method can parallelize recursive functions with complex data flows.

Keywords: data parallelism, parallelization, functional languages, parallel skeletons, data flow analysis, static slice

1. Introduction

Skeleton-based parallel programming has been attracting much attention as a framework for expressing data-parallelism in functional languages. In this framework, parallel programs are composed of high-level functions, called parallel skeletons, which express algorithmic forms common to a range of parallel computations on data collections. This method enables efficient and portable parallel programming by encouraging programmers to build parallel programs from ready-made skeletons for which efficient implementations are known to exist[1,2,3].

However, data-parallel programming is still difficult. In functional languages, data collections such as lists or trees are declared recursively and applications on such data structures are naturally specified using recursion. It is not easy to extract predefined forms of data-parallelism from such specifications and integrate them into efficient parallel programs. Therefore, there have been many attempts

*Correspondence Address: Division of Computer Science, Department of Electrical Engineering & Computer Science, KAIST, 373-1, Kusong-Dong, Yusong-Gu, Taejon, 305-701, Republic of Korea. Tel.82-42-869-5576. Email : jsahn@pllab.kaist.ac.kr,han@cs.kaist.ac.kr

to automatically generate skeleton-based parallel programs from more friendly sequential recursive programs.

Most of these attempts use the calculational approach to program transformation[4]. They are based on the list homomorphism lemma[5]. Because this lemma can only be applied to very restricted forms of functions, many researchers tried to extend the scope of the lemma to more general forms such as almost homomorphisms[6,7], functions on sequential lists[8,9,6,10], and functions on general recursive data structures[11].

However, the calculational methods have some restrictions. To apply the lemma, the parallelizing translator must find appropriate associative operators, which is a difficult and sometimes impossible task. Also, in these methods, input programs must be expressed in pre-defined forms so that transformation rules can be applied. However, it is not straightforward to turn recursive functions with complex data flows into such forms, even if the functions have useful data parallelism.

```
datatype tree = Leaf of int | Node of int * tree * tree

fun inorder (Leaf x, n) = ((Leaf n_1)_2,1_3)_4
|   inorder (Node(x,lt,rt), n) =
  let
    (t1_5,s1_6) = (inorder (lt_7,n_8)_9)_10;
    (t2_11,s2_12) = (inorder (rt_13,((1_14+n_15)_16+s1_17)_18)_19)_20
  in
    ((Node ((n_21+s1_22)_23,t1_24,t2_25)_26)_27, ((s1_28+s2_29)_30+1_31)_32)_33
  end_34
```

Fig. 1. `inorder` : an in-order numbering function[11].

Fig. 1 shows a function which numbers every node of a tree in in-order starting from a given number[11]. The subscripts are labels of subexpressions and pattern variables in the function bodies.[a] Because the size of the left subtree is required to number the right subtree in in-order, `inorder` returns not only the numbered tree but also the tree size to do the numbering in one traversal of the tree. Because the same function is applied to each node of a tree, there can be a useful data-parallelism in this function. However, because the results of recursions are assigned to tuples and the variables of the tuples are used separately, the flow of data is complex. Furthermore, there even exists a data dependency between recursive calls. Therefore, it is difficult to turn the second function body into an application of associative operators to the results of recursions and local computations. Hence, the calculational methods cannot directly parallelize this function[11].

In this paper, we propose an analytical method for transforming call-by-value recursive functions on general recursive data structures into compositions of parallel skeletons. Our research has the following novel aspects:

- We have used more general parallel skeletons which do not require the associativity of argument functions. Our parallel skeletons can guarantee an $O(h)$

[a]This identification is used in Section 3.

parallel execution time with $(n/log\ n)$ processors.[b] This is unsatisfactory compared with the $O(log\ n)$ execution time of parallel skeletons using associative operators. However, we can transform more general recursive functions into compositions of parallel skeletons with this freedom.

- We define a classification of subexpressions in recursive functions based on their parallelism, which can be analyzed using a static slicing analysis. Then, subexpressions of each class can be evaluated using an appropriate parallel skeleton. In this way, we can parallelize recursive functions with complex data flows. This analytical method has become possible because we adopted the general parallel skeletons.

Because of dependencies among recursions, it is not straightforward to transform recursive functions on general recursive data into compositions of parallel skeletons. We could overcome such a difficulty successfully using an analytical method and the more general parallel skeletons.

Our research is presented as follows. In Section 2, we present the parallel skeletons for our object programs. In Section 3, we define the form of recursive functions to be parallelized, then present a classification and an analysis of data-parallelism of recursive functions. In Section 4, the generation of object parallel programs is explained using the `inorder` example. Finally, we conclude in Section 5.

2. Polytypic Parallel Skeletons

Our object parallel programs express parallelism using polytypic parallel skeletons which are generically defined for general recursive data types[12,11,3].

In functional languages, recursive data types are declared as follows.

$$\begin{array}{lcl} \texttt{datatype}\ rtype & = & C_1\ \texttt{of}\ t_1 * rtype^1 * \ldots * rtype^{r_1} \\ & & \ldots \\ & | & C_m\ \texttt{of}\ t_m * rtype^1 * \ldots * rtype^{r_m} \end{array}$$

We have assumed that a data constructor C_i has one non-recursive argument of t_i type and r_i recursive arguments($r_i \geq 0$). Given a value $C_i(v, x_1, \ldots, x_{r_i})$ of *rtype*, we have named v the non-recursive field and x_j the recursive field.

For *rtype*, our parallel skeletons are defined as follows, where $\bar{f} = (f_1, \ldots, f_m)$.

- $map\ \bar{f}\ C_i(v, x_1, \ldots, x_{r_i}) = C_i((f_i\ v),\ map\ \bar{f}\ x_1, \ldots,\ map\ \bar{f}\ x_{r_i})$
- $reduce\ \bar{f}\ C_i(v, x_1, \ldots, x_{r_i}) = f_i(v,\ reduce\ \bar{f}\ x_1, \ldots,\ reduce\ \bar{f}\ x_{r_i})$
- $scan_{up}\ \bar{f}\ C_i(v, x_1, \ldots, x_{r_i}) = C_i(f_i(v, root\ x'_1, \ldots, root\ x'_{r_i}), x'_1, \ldots, x'_{r_i})$
  where $scan_{up}\ \bar{f}\ x_j = x'_j\ (1 \leq j \leq r_i)$ and $root\ C_i(v, x_1, \ldots, x_{r_i}) = v$
- $scan_{dn}\ \bar{f}\ C_i(v, x_1, \ldots, x_{r_i})\ ap\ = C_i(v', scan_{dn}\ \bar{f}\ x_1\ ap_1, \ldots, scan_{dn}\ \bar{f}\ x_{r_i}\ ap_{r_i})$
  where $f_i\ (a, ap) = (v', ap_1, \ldots, ap_{r_i})$.
- $zip\ C_i(v, x_1, \ldots, x_{r_i})\ C_i(w, y_1, \ldots, y_{r_i}) = C_i((v, w), zip\ x_1\ y_1, \ldots, zip\ x_{r_i}\ y_{r_i})$

[b]When we state time complexities in this paper, h means the height of tree-like input data structures and n means the number of elements.

The implementation of *reduce*, $scan_{up}$, and $scan_{dn}$ has been studied by many researchers. We will only summarize the well-known results[13,14,3]. Naive implementations require the parallel execution time of $O(max(T(f_i))_{i=1}^{m} \times h)$ with $(n/log\ n)$ processors where $T(f_i)$ denotes the computing time for f_i. If the tree contraction algorithm can be applied, they can be computed in $O(log\ n \times max(T(f_i))_{i=1}^{m})$ time using $(n/log\ n)$ processors.

3. Parallelism of Recursive Functions

3.1. Input Programs

Recursive functions on recursive data structures are not always suitable for parallelization. Therefore, we have defined the scope of input functions.

Definition 1 Recursive Functions for Parallelization
Recursive functions to be parallelized are those which can be turned into the following form

$$\begin{array}{llcl} \texttt{fun} & f\ (\kappa_1(pat_1, x_{11}, \ldots, x_{1l_1}),\ ap_1) & = & E_1[\langle f\ (x_{1j}, e_{1j})\rangle_{j=1}^{l_1}] \\ \mid & f\ (\kappa_2(pat_2, x_{21}, \ldots, x_{2l_2}),\ ap_2) & = & E_2[\langle f\ (x_{2j}, e_{2j})\rangle_{j=1}^{l_2}] \\ & & & \ldots \\ \mid & f\ (\kappa_n(pat_n, x_{n1}, \ldots, x_{nl_n}),\ ap_n) & = & E_n[\langle f\ (x_{nj}, e_{nj})\rangle_{j=1}^{l_n}] \end{array}$$

where

- *$E_i[\ ]$'s are expression contexts, and E_i's and e_{ij}'s contain neither f nor x_{ik}.*
- *$\langle f\ (x_{ij}, e_{ij})\rangle_{j=1}^{l_i}$ are l_i holes being filled with recursive calls $f\ (x_{ij}, e_{ij})$, where l_i is the number of recursive fields for the data constructor κ_i and e_{ij}'s are actual accumulation parameters for recursive calls.*
- *pat_i's are patterns matched to non-recursive fields of recursive parameters, and x_{ij}'s and ap_i's are variable patterns.* □

The above definition does not require any actual transformation but only restricts the scope of input functions. Given any recursive function, we can easily check the above condition by scanning its formal parameters, recursive calls, and usages of x_{ij}'s.

Recursive functions of Definition 1 have a tuple value as their actual parameter, where the first element is of a recursive type. We have named the first element the recursive parameter and the second element the accumulation parameter. Functions of Definition 1 apply the same function to all the recursive parts of a recursive parameter propagating values via accumulation parameters. Thus, we can extract data-parallelism from these functions.

3.2. Parallelism of Recursive Functions

For functions of Definition 1, the simultaneous function application to all the recursive parts is restrained by the data flow among recursions. This data flow arises from the accumulation parameters and the results of recursions.

The value of the accumulation parameter is initially available only for the function application on the root node. Accumulation parameters for other applications must be evaluated along recursions. If recursive calls are in a function body, computations which use their results must be deferred until the return of the recursions.

When there are multiple recursions in a function body, some results of a recursion may be used to evaluate accumulation parameters for other recursions. In this case, the sibling recursions cannot be evaluated in parallel. However, if the recursion results which are used to evaluate the accumulation parameters can be evaluated without its own accumulation parameter, these results can be evaluated in parallel before the evaluation of accumulation parameters.

In the `inorder` function, `s1`, `t1`, `s2`, and `t2` are the results of recursions. Because `s1` is used to evaluate the accumulation parameter for the second recursion, there exist dependencies between the two recursions. However, because the accumulation parameters `n` and `(1+n)+s1` are not used to evaluate `s1` and `s2` respectively, the values of `s1` and `s2` can be evaluated in parallel without accumulation parameters. Therefore, `inorder` can be executed in parallel as follows. First, the sizes of subtrees, `s1` and `s2`, can be evaluated in bottom-up order by an upward accumulation. Then, the accumulation parameters for all recursions can be evaluated in top-down order by a downward accumulation. Using these results, the in-order number for each tree node can be simultaneously evaluated by a map operation. Finally, the overall result is generated in bottom-up order by a reduce operation.

Considering the above dependencies, we can classify subexpressions in function bodies based on their conditions of data-parallel evaluation. To define the classification, we must formalize the usage relation among subexpressions and results of recursions. We have used static slicing to define this relation[15]. A static slice is a set of parts of a program that may affect values computed at a designated program point for some possible input values. For two subexpressions with labels l_1 and l_2 in a program f, $l_2 \in SS_f(l_1)$ means that some value evaluated from the expression l_2 can be used to evaluate the expression l_1. In this case, we will say l_1 uses l_2.

$$
\begin{aligned}
&SS_f(1) = SS_f(8) = SS_f(15) = SS_f(16) = SS_f(18) = SS_f(21) = \\
&\qquad \{3,6,8,12,14,15,16,17,18,28,29,30,31,32\} \\
&SS_f(2) = \{1,3,6,8,12,14,15,16,17,18,28,29,30,31,32\} \\
&SS_f(3) = SS_f(7) = SS_f(13) = SS_f(14) = SS_f(31) = \{\ \} \\
&SS_f(4) = \{1,2,3,6,8,12,14,15,16,17,18,28,29,30,31,32\} \\
&SS_f(5) = SS_f(11) = SS_f(24) = SS_f(25) = SS_f(26) = SS_f(27) = SS_f(33) = \\
&\qquad \{1,2,3,5,6,8,11,12,14,15,16,17,18,21,22,23,24,25,26,27,28,29,30,31,32\} \\
&SS_f(6) = SS_f(12) = SS_f(17) = SS_f(22) = SS_f(28) = SS_f(29) = SS_f(30) = SS_f(32) = \\
&\qquad \{3,6,12,28,29,30,31,32\} \\
&SS_f(9) = \{3,6,7,8,12,14,15,16,17,18,28,29,30,31,32\} \\
&SS_f(10) = SS_f(20) = \\
&\qquad \{1,2,3,4,5,6,8,11,12,14,15,16,17,18,21,22,23,24,25,26,27,28,29,30,31,32,33,34\} \\
&SS_f(19) = \{3,6,8,12,13,14,15,16,17,18,28,29,30,31,32\} \\
&SS_f(23) = \{3,6,8,12,14,15,16,17,18,21,22,28,29,30,31,32\} \\
&SS_f(34) = \{1,2,3,5,6,8,11,12,14,15,16,17,18,21,22,23,24,25,26,27,28,29,30,31,32,33\}
\end{aligned}
$$

Fig. 2. The static slice of `inorder`.

Fig. 2 shows the static slice of `inorder`. The numbers are subscripts of Fig. 1. In the result, $SS_f(5) \ni 27$ because $SS_f(34) \ni 27$, $SS_f(10) \ni 34$, and the left elements of tuple values returned from $(\texttt{inorder}\ (\texttt{lt}_7,\texttt{n}_8)_9)_{10}$ are assigned to $\texttt{t1}_5$. However, $SS_f(5) \not\ni 32$ because the right elements of values from $(\texttt{inorder}\ (\texttt{lt}_7,\texttt{n}_8)_9)_{10}$ are assigned not to $\texttt{t1}_5$ but to $\texttt{s1}_6$. A static slice can be defined based on operational semantics, and we can analyze all the static slices of a function using abstract interpretation in $O(N^3)$ time where N is the number of subexpressions[16].

Now we can formally define the classification of subexpressions. We have given a color for each class. We assume that recursive calls are always used in the form of $p = f(x, e)$ in `let` bindings without loss of generality.

Definition 2 Formal Definition of Colors

Given a function definition $\texttt{fun}\ f\ pat_1 = e_1 \mid \ldots \mid f\ pat_n = e_n$ *and the static slicing function* SS_f*, let*

$$
\begin{aligned}
&L: \textit{ the set of labels in } e_1, \ldots, e_n \\
&ApLabels = \{l \mid p = f(x,e) \textit{ is a recursive call in } e_1, \ldots, e_n, \quad l \textit{ is the label of } e\} \\
&ApUse = \{l \in L \mid SS_f(l) \cap ApLabels \neq \phi\} \\
&ApUse' = L - ApUse \\
&RecResults = \{l \mid p = f(x,e) \textit{ is a recursive call in } e_1, \ldots, e_n, \\
&\qquad\qquad l \textit{ is a label of a variable pattern in } p\} \\
&RedResults = \{l \mid l \in RecResults, \quad SS_f(l) \cap ApLabels = \phi\} \\
&BlackResults = RecResults - RedResults.
\end{aligned}
$$

Then, the set of labels of each color is defined as follows.

$$
\begin{aligned}
&White = ApUse' \cap \{l \mid SS_f(l) \cap RecResults = \phi\} \\
&Red = (ApUse' - White) \cap \{l \mid \exists l' \in RedResults \textit{ s.t. } l \in SS_f(l')\} \\
&Blue = (ApUse' - White) \cap \{l \mid \quad \nexists l' \in RedResults \textit{ s.t. } l \in SS_f(l')\} \\
&Yellow = ApUse \cap \\
&\qquad \{l \mid SS_f(l) \cap BlackResults = \phi \textit{ and } \exists l' \in ApLabels \textit{ s.t. } l \in SS_f(l')\} \\
&Green = ApUse \cap \\
&\qquad \{l \mid SS_f(l) \cap BlackResults = \phi \textit{ and } \nexists l' \in ApLabels \textit{ s.t. } l \in SS_f(l')\} \\
&Black = ApUse \cap \\
&\qquad \{l \mid SS_f(l) \cap BlackResults \neq \phi \textit{ and } \nexists l' \in ApLabels \textit{ s.t. } l \in SS_f(l')\} \qquad \square
\end{aligned}
$$

$SS_f(l) \cap ApLabels \neq \phi$ means that expression l uses some value of the accumulation parameters, and *ApUse* is the set of such expressions. *RecResults* is the set of variable patterns to which the results of recursive calls are assigned. *RecResults* is divided into *RedResults* and *BlackResults*: the recursion results that do not use accumulation parameters are assigned to variables in *RedResults* and the other results are assigned to variables in *BlackResults*. We call elements of *RedResults* and *BlackResults* red-results and black-results, respectively.

The set of subexpressions of each color can be defined using the above sets. *White* expressions do not use any accumulation parameters or results of recursions, so they can be simultaneously evaluated on all nodes by *map*. *Red* expressions evaluate red-results using red-results, so they are evaluated propagating red-results upwards by $scan_{up}$. After *White* and *Red* expressions have been evaluated, *Blue* expressions can be evaluated by *map*.

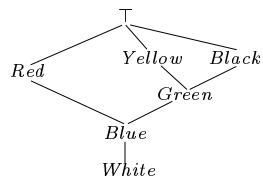

Fig. 3. The lattice for the coloring analysis.

Yellow expressions compute accumulation parameters. Because an accumulation parameter can be used to evaluate other accumulation parameters, they are computed by $scan_{dn}$. After accumulation parameters are evaluated for all recursions, *Green* expressions can be evaluated by *map*. *Black* expressions compute the final result using results from subtrees. They are evaluated by *reduce*.

There can be subexpressions which do not belong to any color. This occurs when a black-result of a recursion is used to evaluate an accumulation parameter of another recursive call. In this case, the input program cannot be transformed into a composition of our parallel skeletons because the accumulation parameters of the sibling recursions cannot be evaluated in parallel. For example, in Fig. 1, if we replace the subexpression $((\mathtt{s1}_{28}+\mathtt{s2}_{29})_{30}+1_{31})_{32}$ with $((\mathtt{s1}_{28}+\mathtt{s2}_{29})_{30}+\mathtt{n}_{31})_{32}$, $\mathtt{s1}_6$ becomes a black-result because the accumulation parameter $\mathtt{n}_8$ is used to evaluate its value, where $\mathtt{s1}_6$ is used to evaluate the second accumulation parameter $((1_{14}+\mathtt{n}_{15})_{16}+\mathtt{s1}_{17})_{18}$. Then, we cannot parallelize this function because the two accumulation parameters cannot be evaluated in parallel.

3.3. Analysis of Parallelism

Using a sound approximation of a static slice, we can analyze the colors of subexpressions from their definition. Fig. 3 is the lattice for the analysis. Because we use an approximation of a static slice which may be incomplete, some expressions of a color can be analyzed to be of an upper color. In these cases, though the efficiency may be reduced, the object program can complete all the computations.

Sometimes, some expressions can be analyzed to belong to ⊤. This occurs when the input program is not parallelizable or the approximation of its static slice is too coarse. In both cases, the parallelization fails.

Fig. 4 is the result of the coloring analysis of `inorder`. `t1` and `t2` are black-

```
fun inorder (Leaf x, n) = ((Leaf n:G):G,1:W):G
|   inorder (Node(x,lt,rt), n) =
   let
      (t1:Bk,s1:R) = (inorder (lt:W,n:Y):G):Bk;
      (t2:Bk,s2:R) = (inorder (rt:W,((1:W+n:Y):Y+s1:B):Y):G):Bk
   in
      ((Node((n:G+s1:B):G,t1:Bk,t2:Bk):Bk):Bk,((s1:R+s2:R):R+1:W):R):Bk
end:Bk
```

Fig. 4. Colors of subexpressions in `inorder`.

results because accumulation parameters are used to compute the first results, and `s1` and `s2` are red-results. `(1+n)+s1` uses an accumulation parameter and evaluates the accumulation parameter for the second recursive call, so it belongs to *Yellow*. `(s1+s2)+1` uses red-results and evaluates red-results, so it belongs to *Red*. Although `n+s1` uses an accumulation parameter and a red-result, it belongs to *Green* because it is not used to evaluate any red-results or accumulation parameters.

4. Generating Data-Parallel Programs

4.1. The Form of Object Parallel Programs

Object parallel programs have the following form. Expressions of each color are used to generate argument functions for the appropriate parallel skeleton.

$$
\begin{array}{l}
\texttt{fun}\ f\ (x, ap)\ = \\
\quad \texttt{let} \\
\qquad W_res\ =\ zip\ x\ (map\ (f_1^{\mathcal{W}}, \ldots, f_m^{\mathcal{W}})\ x); \\
\qquad R_res\ =\ zip\ W_res\ (scan_{up}\ (f_1^{\mathcal{R}}, \ldots, f_m^{\mathcal{R}})\ W_res); \\
\qquad B_res\ =\ zip\ R_res\ (map\ (f_1^{\mathcal{B}}, \ldots, f_m^{\mathcal{B}})\ R_res); \\
\qquad Y_res\ =\ zip\ B_res\ (scan_{dn}\ (f_1^{\mathcal{Y}}, \ldots, f_m^{\mathcal{Y}})\ B_res\ ap); \\
\qquad G_res\ =\ zip\ Y_res\ (map\ (f_1^{\mathcal{G}}, \ldots, f_m^{\mathcal{G}})\ Y_res) \\
\quad \texttt{in} \\
\qquad reduce\ (f_1^{\mathcal{BK}}, \ldots, f_m^{\mathcal{BK}})\ G_res \\
\quad \texttt{end}
\end{array}
$$

Parallel skeletons in the `let` bindings compute values of *White*, *Red*, *Blue*, *Yellow*, and *Green* expressions. Each result is combined with the previous results using *zip*. Each parallel operation can be omitted unless computations of the corresponding color are necessary. The overall result is obtained by evaluating *Black* expressions.

4.2. The Procedure for Parallel Program Generation

Given a recursive function whose subexpressions' colors are analyzed, a parallel program is generated in the following steps. We will explain the procedure using our `inorder` program example.

4.2.1. Preprocessing for Code Generation

We replace each recursive call with a tuple of new variables. The new variables are also added to the formal parameters of argument functions for $scan_{up}$ or *reduce* skeletons, and the results of recursive calls are propagated via these formal parameters. This is done as follows.

(i) We get color tuples from the patterns to which the results of recursive calls are assigned. For `inorder`, we get (*Black*, *Red*) and (*Black*, *Red*) from the patterns `(t1:Bk,s1:R)` and `(t2:Bk,s2:R)` respectively.

(ii) The most detailed color tuple, which we name the result-color, is generated from the previous color tuples by selecting more detailed parts. For example, we can get $((Red, Black), (Red, Black))$ from $(Black, (Red, Black))$ and $((Red,\ Black)\ , Black)$. For `inorder`, we get the result-color $(Black, Red)$ from $(Black, Red)$ and $(Black, Red)$.

(iii) Accumulation parameters of recursive calls are extracted so that they can be used for code generation after recursive calls are replaced with variable tuples.

```
(t1,s1)=inorder(lt,n);          Ap1=n;(t1,s1)=inorder(lt,Ap1);
(t2,s2)=inorder(rt,1+n+s1)  ⇒   Ap2=1+n+s1;(t2,s2)=inorder(rt,Ap2)
```

(iv) Recursive calls are replaced with variable tuples, which are of the same structure as the result-color. For `inorder`, we replace two recursive calls with `(Res1:Bk,Res2:R)` and `(Res3:Bk,Res4:R)`, which are generated from the result-color $(Black, Red)$.

(v) Assignments to constructor or tuple patterns are divided if possible.

Fig. 5 shows the result from the preprocessing. Recursive calls are replaced with variable tuples and the resulting assignments of `(Res1:Bk,Res2:R)` and `(Res3:Bk, Res4:R)` to the tuple patterns `(t1:Bk,s1:R)` and `(t2:Bk,s2:R)` are divided.

```
fun inorder (Leaf x, n) = ((Leaf n:G):G,1:W):G
|   inorder (Node(x,lt,rt), n) =
   let
      Ap1:Y = n:Y; t1:Bk = Res1:Bk; s1:R = Res2:R;
      Ap2:Y = ((1:W+n:Y):Y+s1:B):Y; t2:Bk = Res3:Bk; s2:R = Res4:R
   in
      ((Node((n:G+s1:B):G,t1:Bk,t2:Bk):Bk):Bk,((s1:R+s2:R):R+1:W):R):Bk
end:Bk
```

Fig. 5. The `inorder` program after preprocessing.

4.2.2. Extracting Subexpressions of Each Color

We extract computations of each color from the preprocessed function bodies in the order of *White*, *Red*, *Blue*, *Yellow*, and *Green*. Atomic expressions such as variables or constants are not extracted because they need no computation.

Subexpressions are extracted in the form of assignments to variables. If a subexpression of the color for the extraction is the whole right-hand side of an assignment, the assignment is extracted. Otherwise, we replace the subexpression with a new variable, and an assignment of the subexpression to the new variable is returned.

In the first function body of `inorder`, there are only *Green* computations. The result of the extraction with *Green* is ⟨`temp1=(Leaf n,1)`⟩, and the function body becomes `temp1`. In the second function body, there are *Red*, *Yellow Green*, and *Black* computations. The results of the *Red*, *Yellow*, and *Green* extractions are ⟨`s1=Res2; s2=Res4; temp2=(s1+s2)+1`⟩, ⟨`Ap1=n; Ap2=(1+n)+s1`⟩, and ⟨`temp3=(n+s1)`⟩, respectively. And after all these extractions, the second function body becomes ⟨`let t1=Res1;t2=Res3 in (Node(temp3,t1,t2),temp2) end`⟩, which has only *Black* computations.

4.2.3. Building Argument Functions for Parallel Skeletons

We can generate a parallel program using the results from the previous steps. For `inorder`, we have obtained the result-color (*Black*, *Red*), replaced the recursive calls in the function bodies with the variable tuples `(Res1,Res2)` and `(Res3,Res4)`, and extracted subexpressions of *Red*, *Yellow*, and *Green* colors.

Fig. 6 shows the parallel in-order numbering function automatically generated by our prototype parallelizing translator, which has been implemented using SML/NJ. Because there are no *White* and *Blue* computations, the first and the second *map* operations are omitted in the function.

In the formal parameters of the argument functions, `x` is matched to non-recursive fields of tree nodes and other parts are matched to the results from the previous parallel operations and recursive calls, or accumulation parameters. Function bodies have the form of `let` expressions, whose assignments are the results of subexpression extractions and whose result parts are tuples of assigned variables.

In `inorder`, its red-result is the size of each tree. `inorder_R1` and `inorder_R2` are generated from *Red* expressions and return a tuple of the tree size and intermediate results. For `Leaf` nodes, there are no *Red* computations and we can get the red-result `1` by matching the result-color (*Black*, *Red*) with the function body `((Leaf n:G):G,1:W):G`. Therefore, `inorder_R1` returns a tuple of the red-result `1` and a dummy value `()`. `()` expresses that there are no intermediate results. In `inorder_R2`, the formal parameters `(Res2,_)` and `(Res4,_)` are matched to results from child nodes, where `Res2` and `Res4` are matched to red-results and `_`'s are matched to intermediate results. For the result part, the red-result `temp2` is obtained by matching (*Black*, *Red*) with the function body remaining after the *Red* extraction and `(s1,s2,temp2)` is the tuple of the intermediate results.

Yellow expressions are transformed into argument functions for $scan_{dn}$. In these functions, the accumulation parameter `n` is added to the formal parameters, and accumulation parameters for recursions on child nodes are added to the result parts. Because `Leaf` nodes have no child, `inorder_Y1` returns only the current accumulation parameter. In the result part of `inorder_Y2`, `(n,Ap1,Ap2)` is the tuple of an accumulation parameter and intermediate results from the *Yellow* expressions, and `Ap1` and `Ap2` are accumulation parameters for recursive calls.

`inorder_G1` and `inorder_G2` are generated from the results of the *Green* extractions. In their formal parameters, `(_,_)` and `(_,(s1,s2,temp2))` are matched to the results from the $scan_{up}$ operation, and `n` and `(n,Ap1,Ap2)` are matched to the results from the $scan_{dn}$ operation. Using these previous results, the applications of `inorder_G1` and `inorder_G2` can be simultaneously computed on all nodes.

`inorder_BK1` and `inorder_BK2` compute the final results for `Leaf` and `Node` nodes, respectively. Their formal parameters are matched to the intermediate results from the previous computations and the final results from the recursions. Their function bodies are the remaining expressions after the *Green* extractions.

Fig. 7 shows an execution of `par_inorder`. We can further optimize `par_inorder` so that it does not store useless intermediate results such as `s2`, `Ap1` and `Ap2`.

```
fun par_inorder (tree,ap) =
   let
      R_res = zip tree (scan_up (inorder_R1,inorder_R2) tree);
      Y_res = zip R_res (scan_dn (inorder_Y1,inorder_Y2) R_res ap);
      G_res = zip Y_res (map (inorder_G1,inorder_G2) Y_res)
   in
       reduce (inorder_BK1,inorder_BK2) G_res
   end
fun inorder_R1 x = (1,())
fun inorder_R2 (x,(Res2,_),(Res4,_)) =
   let s1 = Res2; s2 = Res4; temp2 = (s1+s2)+1 in (temp2,(s1,s2,temp2)) end
fun inorder_Y1 ((x,(_,_)), n) = n
fun inorder_Y2 ((x,(_,(s1,s2,temp2))), n) =
   let Ap1 = n; Ap2 = 1+n+s1 in ((n,Ap1,Ap2),Ap1,Ap2) end
fun inorder_G1 ((x,(_,_)),n) = let temp1 = (Leaf n,1) in temp1 end
fun inorder_G2 ((x,(_,(s1,s2,temp2))),(n,Ap1,Ap2)) =
   let temp3 = (n+s1) in temp3 end
fun inorder_BK1 (((x,(_,_)),n),temp1) = temp1
fun inorder_BK2 ((((x,(_,(s1,s2,temp2))),(n,Ap1,Ap2)),temp3),
                 (Res1,Res2),(Res3,Res4)) =
   let t1 = Res1; t2 = Res3 in (Node(temp3,t1,t2),temp2) end
```

Fig. 6. An object parallel program from `inorder`.

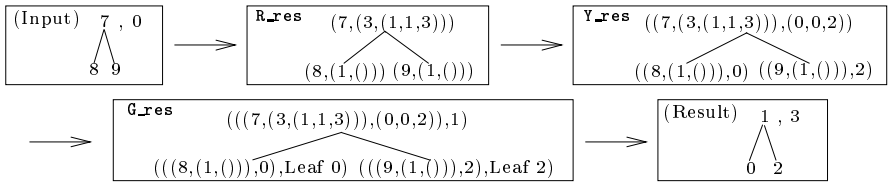

Fig. 7. A parallel computation of `par_inorder` on `(Node(7,Leaf 8,Leaf 9), 0)`.

The time complexities of object parallel programs depend on the implementation of the parallel skeletons. Assuming a naive implementation, `par_inorder` can be computed in $O(h)$ parallel time using $n/(log\ n)$ processors, which is not optimal. However, because the argument functions of `par_inorder` are composed of associative operators, it can easily be transformed into a composition of the parallel skeletons which use associative argument functions[11]. In this way, we can get an optimal in-order numbering program having $O(log\ n)$ parallel time complexity.

5. Conclusion

The parallel implementation of recursive functions can be divided into two tasks, transforming sequential functions into compositions of parallel skeletons and implementing the parallel skeletons. Concentrating on the former, we have proposed a method for transforming recursive functions on general recursive data structures into skeleton-based parallel programs. By adopting an analytical approach and general parallel skeletons which do not require the associativity of argument functions, we can parallelize recursive functions with complex data flows.

After our transformation, it becomes easier to find associative argument func-

tions which guarantees more efficient implementation of the parallel skeletons. Also, our resulting programs can fully utilize efficient implementations of parallel skeletons such as tree contraction methods[13,14,3] or efficient linear reductions with non-associative operators[17].

Acknowledgements

Kwangkeun Yi and the anonymous referees made important suggestions. This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center(AITrc).

References

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

[2] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, Q. Wu and R.L. While, Parallel programming using skeleton functions, in *PARLE'93, Parallel Architectures and Languages Europe*, Berlin, DE, June 1993, 146–160.

[3] D.B. Skillicorn, Parallel implementation of tree skeletons, *Journal of Parallel and Distributed Computing* **39**:2 (1996) 115–125.

[4] Akihiko Takano, Zhenjiang Hu and Masato Takechi, Program transformation in calculation form, *ACM Computing Surveys* **30**:3es (1998) Article 7.

[5] Richard Bird, An introduction to the theory of lists, in *Logic of Programming and Calculi of Discrete Design*, eds. M. Broy (Springer-Verlag, 1987) 5–42.

[6] Sergei Gorlatch, Extracting and implementing list homomorphisms in parallel program development, *Science of Computer Programming* **33**:1 (1999) 1–27.

[7] Zhenjiang Hu, Hideya Iwasaki and Masato Takechi, Formal derivation of efficient parallel programs by construction of list homomorphisms, *ACM Transactions on Programming Languages and Systems* **19**:3 (1997) 444-461.

[8] A. Geser and S. Gorlatch, Parallelizing functional programs by generalization, *Journal of Functional Programming* **9**:6 (1999) 649–673.

[9] Jeremy Gibbons, The third homomorphism theorem, *Journal of Functional Programming* **6**:4 (1996) 657–665.

[10] Zhenjiang Hu, Masato Takechi and Wei-Ngan Chin, Parallelization in calculation forms, in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Sandiego, CA, USA, January 1998, 316–328.

[11] Zhenjiang Hu, Masato Takechi and Hideya Iwasaki, Diffusion: Calculating efficient parallel programs, in *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Texas, USA, January 1999, 85–94.

[12] Jeremy Gibbons, Polytypic downwards accumulations, in *Mathematics of Program Construction '98*, Marstrand, Sweden, June 1998.

[13] K. Abrahamson, N. Dadoun, D.G.Kirkpatrick and T.Przytycka, A simple parallel tree contraction algorithm, *Journal of Algorithms* **10** (1989) 287–302.

[14] Jeremy Gibbons, Efficient parallel algorithms for tree accumulations, *Science of Computer Programming* **23**:1 (1994) 1–18.

[15] Frank Tip, A survey of program slicing techniques, Technical Report CS-R9438 (1994), Centrum voor Wiskunde en Informatica.

[16] Joonseon Ahn and Taisook Han, Static slicing of a first-order functional language based on operational semantics, Technical Report CS/TR-99-144 (1999), KAIST.

[17] Christoph Wedler and Christian Lengauer, On linear list recursion in parallel, *Acta Informatica* **35**:10 (1998) 875–909.