Parallelization via Context Preservation

Wei-Ngan CHIN[*]
National University of Singapore

Akihiko TAKANO
Hitachi Advanced Research Laboratory

Zhenjiang HU
University of Tokyo

Abstract

Abstract program schemes, such as scan or homomorphism, can capture a wide range of data parallel programs. While versatile, these schemes are of limited practical use on their own. A key problem is that the more natural sequential specifications may not have associative combine operators required by these schemes. As a result, they often fail to be immediately identified. To resolve this problem, we propose a method to systematically derive parallel programs from sequential definitions. This method is special in that it can automatically invent auxiliary functions needed by associative combine operators. Apart from a formalisation, we also provide new theorems, based on the notion of context preservation, to guarantee parallelization for a precise class of sequential programs.

1 Introduction

It is well-recognised that a key problem of parallel computing remains the development of efficient and correct parallel software. This task is further complicated by the variety of parallel architectures available, from vector processors to loosely-coupled distributed systems. The difficulty of parallel programming has led to calls by a number of researchers for a more declarative approach towards parallelism [25, 12]. Two main ideas here are an architecture-independent functional language to capture the essence of computation, and a small set of generic parallel primitives (or schemes). Two basic parallel primitives on lists are:

$$
\begin{array}{ll}
map_f(Nil) & = Nil \\
map_f(x{:}xs) & = f(x){:}map_f(xs) \\
reduce_\oplus([x]) & = x \\
reduce_\oplus(x{:}xs) & = x \oplus reduce_\oplus(xs)
\end{array}
$$

Such higher-order primitives can capture a wide range of sequential functions over lists. Note that *Nil* and *:* are data constructors for list. Also, we shall use $[x_1, \ldots, x_n]$ to represent a list of n-elements, while $+\!\!+$ is the list-concatenation operator but denotes list-splitting when used as a pattern in the LHS. Though given sequential definitions, these two functions also have corresponding parallel implementation. The function *map* can apply a function f to every element of a sequence in parallel, as follows:

$$map_f([x_1, x_2, \ldots, x_n]) = [f(x_1), f(x_2), \ldots, f(x_n)]$$

With an associative $\oplus$ operation, the function *reduce* can perform a tree-like parallel reduction via:

$$reduce_\oplus(xr +\!\!+ xs) \quad = reduce_\oplus(xr) \oplus reduce_\oplus(xs)$$

These two functions are thus very useful building blocks for parallel programming. In fact, a simple composition of the two operators, via $hom_{(f,\oplus)}(xs) = reduce_\oplus(map_f(xs))$, can capture a wide-range of data parallel programs. Such a composition, called *append-list homomorphism* (herewith referred as *homomorphism*) [2], can be expressed as:

$$
\begin{array}{ll}
hom_{(f,\oplus)}([x]) & = f(x) \\
hom_{(f,\oplus)}(xr +\!\!+ xs) & = hom_{(f,\oplus)}(xr) \oplus hom_{(f,\oplus)}(xs)
\end{array}
$$

In order for $hom_{(f,\oplus)}$ to be well-defined, its combine operator $\oplus$ must be associative on its expected domain (i.e. range of $hom_{(f,\oplus)}$ itself) in order to match the associativity of $+\!\!+$. That homomorphisms are extremely useful can be confirmed by expressing the widely popular parallel prefix operator, called *scan*, in terms of a homomorphism (where $last(u)$ returns the final element of list u), as follows:

$$
\begin{array}{l}
scan_\otimes(xs) = hom_{(id,k)}(xs) \text{ where} \\
\qquad \{\ id(x) = x\ ;\ k(u,v) = u +\!\!+ map_{(last(u)\ \otimes)}(v)\ \}
\end{array}
$$

Blelloch and co-workers [3, 4] have shown that *scan* can be used to build effective parallel programs for a wide range of problems, including important examples from the domains of numerical analysis, sorting, lexical analysis, graph and tree algorithms, and also in computational geometry.

While these parallel operators may be extremely versatile, they are relatively difficult to use. The main difficulty stems from the fact that the combine operator (i.e. $\oplus$) for tree-like reduction must be associative (on its domain). It is much easier for programmers to initially specify their programs in a sequential setting. This approach can facilitate the development of robust software since debugging is also simpler under this setting. Unfortunately, sequential programs are often quite different from their parallel counterparts since the latter must satisfy an associative requirement. Specifically, they often fail to match up with the sequential form of the generic primitives, such as $scan_\otimes$ or $homo_{(f,\oplus)}$. Consider the polynomial function:

$$poly([c_0, \ldots, c_n], x) = c_0 + c_1 * x + c_2 * x^2 + \ldots + c_n * x^n$$

which can be naturally expressed sequentially as:

$$
\begin{array}{ll}
poly([c], x) & = c \\
poly(c{:}cs, x) & = comb1(c, poly(cs, x)) \\
& \quad \text{where } comb1(c, r) = c + (r * x)
\end{array}
$$

[*] This work was done while the first author was a Visiting Research Scientist at Hitachi Advanced Research Laboratory.

To appear in the IEEE International Conference on Computer Languages, 14-16 May 1998, Chicago, USA, IEEE Press.

Sadly, this definition does not match with *reduce* since its combine operator, *comb1*, is not associative. Nevertheless, with the help of an auxiliary function, *prod*, a parallel program for *poly* can be written as:

$$\begin{array}{ll} poly([c],x) & = c \\ poly(cr{+\!\!+}cs,x) & = poly(cr,x)+(poly(cs,x)*prod(cr,x)) \\ prod([c],x) & = x \\ prod(cr{+\!\!+}cs,x) & = prod(cs,x)*prod(cr,x) \end{array}$$

From here, an automated tupling method[7, 18] would introduce $ptup(cs,x)=(poly(cs,x),prod(cs,x))$ before transforming to the following homomorphism.

$$\begin{array}{ll} poly(cs,x) & = let\ (u,\_)=ptup(cs,x)\ in\ u \\ ptup([c],x) & = (c,x) \\ ptup(cr{+\!\!+}cs,x) & = comb2(ptup(cr,x),ptup(cs,x))\ where \\ & \quad comb2((p_1,u_1),(p_2,u_2)) = (p_1 + p_2 * u_1, u_2 * u_1) \end{array}$$

Such mismatch between sequential and parallel programs are already recognized by a number of people, including Cole [11] and Gorlatch [16]. The key to obtaining a better match is the introduction of suitable auxiliary functions (such as *prod*) that would allow associative combine operators (such as *comb2*) to be expressed. Past proposals require these auxiliary functions to be given. Cole required auxiliary functions to be provided as part of tupled homomorphism, while Gorlatch asked for an extra rightward sequential function, which often has the needed auxiliary functions. We propose an innovative method to generalize sequential programs into parallel ones, and invent the necessary auxiliary functions. Our contributions are:

- We provide a systematic approach to derive a class of parallel programs directly from sequential codes. Our method is formally defined, and uses a novel inductive derivation procedure to synthesize auxiliary functions needed in parallel algorithms. (Sec. 3 and 4)
- We give two new theorems, called context preservation theorems, that highlight sufficient conditions which guarantee efficient parallelism for a wide class of sequential programs. (Sec. 5)
- We show that our method can directly derive a wide range of parallel programs, including those beyond homomorphism. (Sec. 6 and 7)

The next section gives an informal review.

2 Generalizing from Sequential Code

Most optimisation techniques, such as partial evaluation[20] and deforestation[27], are based on the notion of program specialisation, whose goal is to specialise programs to their specific contexts of use. However, parallelization is an entirely different beast as the goal is to obtain the more *general* parallel equations from their sequential counterparts. Due to this difference, we have developed a new method for parallelization based on generalizing from sequential examples. An initial (informal) proposal can be found in [6]. We shall outline its main steps using an accumulative version of *scan*.

$$\begin{array}{ll} ascan_{\otimes}([x],w) & = [w \otimes x] \\ ascan_{\otimes}(x{:}xs,w) & = [w \otimes x]{+\!\!+}ascan_{\otimes}(xs,w \otimes x) \end{array}$$

To parallelize, we perform the following four steps.

Step 1 : Find Two Equations with Common Context

We first obtain two sequential equations of $ascan_{\otimes}$ with a common context, from the original equation.

$$\begin{array}{ll} ascan_{\otimes}(\underline{[x]}{+\!\!+}xs,w) & = \underline{[w \otimes x]}{+\!\!+}ascan_{\otimes}(x,w \otimes \underline{x}) \\ ascan_{\otimes}(\underline{[x,y]}{+\!\!+}xs,w) & = \underline{[w \otimes x, w \otimes x \otimes y]} \\ & \quad {+\!\!+}ascan_{\otimes}(xs,w \otimes \underline{(x \otimes y)}) \end{array}$$

The two equations have a common context, namely:

$$ascan_{\otimes}(\_{+\!\!+}xs,w) = \_{+\!\!+}(ascan_{\otimes}(xs,w{\otimes}\_)$$

with their respective *recursive call* and *accumulative parameter* at the same positions in the two equations.

Step 2 : Second-Order Generalization

A second-order generalization process is now made to *generalize* the two sequential equations, into a single parallel equation. In the process, one or more unknown functions may be introduced for the *mismatched* sub-terms (shown underlined), as follows:

$$ascan_{\otimes}(\underline{xr}{+\!\!+}xs,w) \;=\; \underline{uH(xr,w)}{+\!\!+}ascan_{\otimes}(xs,w \otimes \underline{uK(xr)})$$

First-order variable (i.e. xr) is introduced to replace a mismatched subterm in the LHS, while second-order variables (i.e. unknown functions uH and uK), applied to suitable arguments, are introduced to replace the mismatched subterms in the RHS.

Step 3 : Inductive Derivation for Unknown Functions

Apply an inductive derivation procedure to synthesize definitions for the unknown functions. By applying appropriate instantiation to the recursion argument, we could simplify and normalise the LHS till it reaches a form which could be *unified* with the RHS. This can yield definitions for the unknown functions. In the case of $ascan_{\otimes}$, we obtain:

From Base Case $xr=[x]$, inductive derivation yields:

$$\begin{array}{ll} uH([x],w) & = [w \otimes x] \\ uK([x]) & = x \end{array}$$

From Step Case $xr=xa{+\!\!+}xb$, inductive derivation yields:

$$\begin{array}{ll} uH(xa{+\!\!+}xb,w) & = uH(xa,w){+\!\!+}uH(xb,w \otimes uK(xa)) \\ uK(xa{+\!\!+}xb) & = uK(xa) \otimes uK(xb) \end{array}$$

The newly derived functions of uH and uK are checked to see if they are syntactically equivalent to known functions being parallelized. A simple check confirms that $uH{\equiv}ascan_{\otimes}$ where $\equiv$ denotes syntactic equivalence modulo alpha renaming. We substitute $[uH{\mapsto}ascan_{\otimes}]$ to remove unnecessary functions.

Step 4 : Tupling for Efficient Parallelism

Our parallel equations may contain redundant calls. This inefficiency can be removed by applying the automated tupling method of [7]. In the case of $ascan_{\otimes}$, redundant uK calls exist. Tupling can initially introduce:

$$ascantup_{\otimes}(xs,w) \quad = (ascan_{\otimes}(xs,w),uK(xs))$$

This can then be transformed to the following efficient and parallel code.

$$\begin{array}{ll} ascan_{\otimes}(xs,w) & = let\ (u,\_)=ascantup_{\otimes}(xs,w)\ in\ u \\ ascantup_{\otimes}([x],w) & = ([w \otimes x],x) \\ ascantup_{\otimes}(xa{+\!\!+}xb,w) & = let\ \{(a,u)=ascantup_{\otimes}(xa,w); \\ & \qquad\quad (b,v)=ascantup_{\otimes}(xb,w \otimes u)\} \\ & \quad in\ (a{+\!\!+}b,u \otimes v) \end{array}$$

Our method is primarily for obtaining abstract data parallel algorithms from sequential specifications. We do *not* deal with the issue of mapping these programs to particular parallel architectures, nor provide suitable cost models to justify the derived programs. A number of other works, such as [1, 23], have dealt with the more intricate implementation issue using divide-and-conquer programs as starting points. These works are complimentary to our proposal.

Nevertheless, a serious shortcoming of earlier work in [6, 10] is that the methodology was largely informal. To rectify this shortcoming, this paper formalises the key steps of our method and states precisely the class of sequential programs which could be handled by out method.

3 Acceptable Sequential Programs

We are mainly interested in a class of divide-and-conquer algorithms with simple divide operator (i.e. $+\!\!+$). For simplicity, we use a strict first-order functional language:

Defn 1: *A First-Order Language*

$$\begin{array}{lll} F & ::= & \{f(p_{i,1},\ldots,p_{i,n_i}) = t_i\}_{i=1}^{m} \\ t & ::= & v \mid c(t_1,\ldots,t_n) \mid f(t_1,\ldots,t_n) \\ & & \mid let\ (v_1,\ldots,v_n) = t_1\ in\ t_2 \\ & & \mid if\ t_1\ then\ t_2\ else\ t_3 \\ p & ::= & v \mid c(p_1,\ldots,p_n) \end{array}$$

This language contains the usual constructor sub-terms, function calls, *let* and *if* constructs. Also, each function f is defined by a set of pattern-matching equations.

We propose to group functions into mutual-recursive sets, and classify their calls, as follows:

Defn 2: *Mutual Recursive Functions*
Consider a program P with a set of functions $\{f_i\}_{i\in M}$. We define a function dependency relation $f_i \stackrel{P}{\to} f_j$ to hold if f_j is a callee of f_i, or there is a transitive relationship $\exists\, k \in M. f_i \stackrel{P}{\to} f_k \wedge f_k \stackrel{P}{\to} f_j$.
A set of functions G is said to be *mutual-recursive* if it forms a strongly connected component in the transitive relation $\stackrel{P}{\to}$, as follows:

$$\forall f_i, f_j \in G.\, f_i \not\equiv f_j \textbf{ implies } f_i \stackrel{P}{\to} f_j \ \wedge\ f_j \stackrel{P}{\to} f_i$$

Defn 3: *Types of Function Calls*
Consider a function f_i. Each call $f_j(\ldots)$ in the RHS of the definition of f_i can be classified as:

- a *self-recursive* call of f_i if $f_j \equiv f_i$
- a *mutual-recursive* call of f_i if $(f_j \stackrel{P}{\to} f_i) \wedge (f_j \not\equiv f_i)$
- an *auxiliary* call of f_i if $\neg(f_j \stackrel{P}{\to} f_i)$.

3.1 Linear Self-Recursive Functions

We shall focus on sequential functions that are inductively defined over linear data types with an associative decomposition operator. For practical reasons, we restrict the linear data type to *List* (and *Integer* as a special case) with $+\!\!+$ as its decomposition operator. Also, we shall only perform parallelization for a special sub-class of sequential functions, called *linear self-recursive* functions.

Defn 4: *Linear Self-Recursive Functions*
A function f is said to be *self-recursive* if it contains only self-recursive calls, but not mutual recursive calls. In addition, it is said to be *linear self-recursive* (LSR) if its definition contains exactly one self-recursive call.

A list-based version of LSR-function will have exactly one recursive equation of the form:
$f(p_1,\ldots,p_m,v_1,\ldots,v_n) = E_r\langle f(xs_1,\ldots,xs_m,t_1,\ldots,t_n)\rangle$

where $\forall i \in 1..m.\ xs_i \sqsubset p_i$ and $Vars([p_1,\ldots,p_m])$ are allowed to occur freely in E_r and $\{t_i\}_{i\in 1..n}$. (Note $\sqsubset$ is a proper subterm predicate, while $Vars(e)$ returns the set of free variables in e). Also, there exists exactly one self-recursive f call. The context $E_r\langle\,\rangle$ and $\{t_i\}_{i\in 1..n}$ do not contain other self or mutual-recursive calls. If we restrict to a single pattern parameter (i.e. $m = 1$) and simple pattern (i.e. $p_1 = x_1 : xs_1$), our list-based LSR-function reduces to a *list paramorphism* [21]. In the rest of this paper, we shall focus mainly on *list paramorphisms.* Nevertheless, our results carry over to functions beyond list paramorphism (and homomorphism), as outlined in Sec. 7.

Ex. 1: *Sample Recursive Functions*

```
arrive(Nil)        = 0
arrive((s,a):xs)   = a + arrive(xs)
depart(Nil)        = 0
depart((s,a):xs)   = s+max(a+arrive(xs),depart(xs))
max([x])           = x
max(x:xs)          = if x>max(xs) then x else max(xs)
filter(Nil,c)      = Nil
filter(x:xs,c)     = if x<c then filter(xs,c) else x:filter(xs,c)
fib(0)             = 0
fib(Succ(n))       = fib'(n)
fib'(0)            = 1
fib'(Succ(n))      = fib'(n)+fib(n)
second(Nil,w)      = Nil
second(x:xs,w)     = second'(xs,w+x)
second'(Nil,w)     = Nil
second'(x:xs,w)    = (x+w):second(xs,w)
```

All functions above, except *fib*, *fib'*, *second* and *second'*, are self-recursive. Of these, only *arrive* and *depart* (to compute the arrival/departure times of a single server/queue simulation) are linear self-recursive, though *depart* also has an auxiliary recursive call. Functions *max* and *filter* are not LSR since they contain multiple self-recursive calls. Functions that are not LSR will be ignored by our method. To minimise loss of generality, we propose three pre-processing techniques to convert sequential programs to linear self-recursive form, where possible. They are (i) *tupling* [7], (ii) *unfolding of unnecessary mutual recursion*, and (iii) conditional normalisation [8].

These techniques are helpful in the following ways. Tupling can eliminate the redundant calls of *max*[1], *fib* and *fib'* by using *let* abstraction to capture the duplicated calls. The unnecessary intermediate mutual recursive calls of *second* and *second'* can be eliminated by unfolding, while conditional normalisation can be applied to *filter* to combine recursive calls in different conditional branches together. As a result of these techniques, we can obtain the following converted LSR-functions.

Ex. 2: *Converted Linear Self-Recursive Functions*

```
max(x:xs)            = let z=max(xs) in if x>z then x else z
filter(x:xs,c)       = (if x<c then Nil else [x])++filter(xs,c)
fib(n)               = let (u,_)=fibtup(n) in u
fib'(n)              = let (_,v)=fibtup(n) in v
fibtup(0)            = (0,1)
fibtup(Succ(n))      = let (u,v)=fibtup(n) in (v,u+v)
second([x],w)        = Nil
second(x:(y:xs),w)   = (y+(w+x)):second(xs,w+x)
second'([x],w)       = [x+w]
second'(x:(y:xs),w)  = (x+w):second'(xs,w+y)
```

3.2 Types of Parameters

There are two types of parameters. The main inductive parameters are called the *recursion* parameters, while all other parameters are referred to as the *auxiliary* parameters.

[1] In the case of *max*, we use a special tuple with only one call.

Defn 5: *Recursion and Auxiliary Parameters*
Consider the recursive equation of an LSR-function:

$$f(p_1,\ldots,p_m,v_1,\ldots,v_n) = E_r\langle f(xs_1,\ldots,xs_m,t_1,\ldots,t_n)\rangle \quad st\ \forall i\in 1..m.\ xs_i \sqsubset p_i$$

The pattern parameters, $p_1,\ldots,p_m$, are referred to as the *recursion* parameters. All the other parameters, $v_1,\ldots,v_n$, are called the *auxiliary* parameters.

The auxiliary parameters are divided into three distinct groups. Each parameter v_j is classified as:

1. v_j is *non-recursive* if
   $\forall i\in 1..m.\ \forall v\in Vars(t_j)\ \cap\ \{v_i\}_{i=1}^n.\ \neg(v \xrightarrow{f} v_j)$
2. v_j is *accumulative* if $\forall i\in 1..m.\ v_j\in Vars(t_j)$
   $\wedge\ (\forall v\in (Vars(t_j)\ \cap\ \{v_i\}_{i=1}^n - \{v_j\}).\ \neg(v\xrightarrow{f} v_j))$
3. v_j is *mutual-dependent* if
   $v_j\xrightarrow{f} v_j\ \wedge\ (\exists v\in (Vars(t_j)\ \cap\ \{v_i\}_{i=1}^n) - \{v_j\}).\ (v\xrightarrow{f} v_j))$

Note $v_i\xrightarrow{f} v_j$ is a transitive dependency relation which holds if the parameter v_i depends on parameter v_j in the self-recursive caller-callee transition of f. Each *non-recursive* parameter does not have circular dependency with anyone including itself. Each *accumulative* parameter has a circular dependency with itself, but not with others. However, each *mutual parameter* has cyclic dependency with other parameters.

To simplify matter, we shall convert all auxiliary parameters to become accumulative. A set of mutual dependent parameters can be converted to a single accumulative parameter by tupling the parameters. Also, each non-recursive parameter v_j can be made into an accumulative one through the rewriting $t_j \Rightarrow\ k(t_j,v_j)$ where $k(x,y)=x$.

Ex. 3: Consider:

$$\begin{aligned}&foo(a:v_0,v_1,v_2,v_3,v_4,v_5)\\ &\qquad = E\langle foo(v_0,v_1,v_2+1,v_2+v_5+v_3,a,v_5*v_3)\rangle\end{aligned}$$

The parameter $(a:v_0)$ is the recursion parameter, parameters v_1 and v_2 are accumulative, v_4 is non-recursive, while v_3 and v_5 are mutual dependent. By tupling v_3 and v_5, and using a k function for v_4, foo can be transformed to use only accumulative parameters, as follows:

$$\begin{aligned}&foo(v_0,v_1,v_2,v_3,v_4,v_5) = foo'(v_0,v_1,v_2,(v_3,v_5),v_4)\\ &foo'(a:v_0,v_1,v_2,(v_3,v_5),v_4)\\ &\quad = E\langle foo'(v_0,v_1,v_2+1,h(v_3,v_5),k(a,v_4))\rangle \text{ where}\\ &\qquad \{h(w_1,w_2)=(v_2+w_2+w_1,w_2*w_1);\ k(x,y)=x\ \}\end{aligned}$$

4 Context Replication & Preservation

We introduce a context notation to explicitly extract sub-terms from a given expression.

Defn 6: *Context Extraction*
Given an expression E and sub-terms $\langle e_1,\ldots,e_n\rangle$, we shall express its extraction by:

$$E\Rightarrow_A E'\langle e_1,\ldots,e_n\rangle$$

or more precisely $E \overset{\langle e_1,\ldots,e_n\rangle}{\Rightarrow_A} E'\langle e_1,\ldots,e_n\rangle$
The context E' itself will be represented by:

$$E'\equiv \hat\lambda\langle \__1,\ldots,\__n\rangle.[e_i\mapsto \__i]_{i=1}^n\ E$$

where $\langle \__1,\ldots,\__n\rangle$ is a set of new hole variables, and $([e_i\mapsto \__i]_{i=1}^n\ E)$ is a substitution notation to denote the replacement of each occurrence of e_i in E by $\__i$.

In abbreviation, we denote $\langle e_1,\ldots,e_n\rangle$ by either $\langle e_i\rangle_{i\in 1..n}$ or $\langle e_i\rangle_{i=1}^n$. Also, a sequence of terms $e_1,\ldots,e_n$ may sometimes be abbreviated as $\lfloor e\rfloor$ or $\lfloor e_i\rfloor_{i=1}^n$. The context abstraction, $(\hat\lambda\langle\__i\rangle_{i=1}^n.E_c)$, and its application, $(\hat\lambda\langle\__i\rangle_{i=1}^n.E_c)\langle e_i\rangle_{i=1}^n$, are similar to lambda abstraction and application. The key difference is that context notation[2] is formulated at the meta-level with expressions as data values. Likewise, $\langle\ldots\rangle$ can be viewed as a meta-level equivalent of the tuple construct, $(\ldots)$, at the object level.

We introduce a special sub-class of contexts, called *skeletal contexts.*

Defn 7: *Skeletal Context*
A context, E, is said to be a *skeletal context* if every sub-term in E contains at least one hole.

Given a context, E_j, we can make it into a skeletal context by extracting all maximal sub-terms that do not contain holes. This process shall be denoted by $E_j\Rightarrow_A^S (\hat\lambda\langle\__i\rangle_{i\in N}.[t_i\mapsto\__i]_{i\in N}\,E_j)\langle t_i\rangle_{i\in N}$ where $\langle t_i\rangle_{i\in N}$ are all maximal sub-terms without holes from E_j.

Ex. 4: Consider $(\hat\lambda\langle\__1\rangle.(u+v)*\__1)$. This context is not skeletal yet, but can be converted by:

$$(\hat\lambda\langle\__1\rangle.(u+v)*\__1)\Rightarrow_A^S(\hat\lambda\langle\__2\rangle.\ \hat\lambda\langle\__1\rangle.\__2*\__1)\langle u+v\rangle$$

We also introduce the notion of context composition and their transformation below.

Defn 8: *Context Composition*
Consider an expression which is formed by a nested context application:

$$E_1\langle E_2\langle r\rangle\rangle$$

To provide a more concise view, we allow contexts to be composed, as follows:

$$E_1\langle E_2\langle r\rangle\rangle = (E_1\ \circ\ E_2)\langle r\rangle$$

Defn 9: *Context Transformation*
Context may be transformed (or simplified) by either applying laws or unfolding. For example, the context below is simplified using the identity laws, $0+a\Rightarrow a$ and $a*1\Rightarrow a$.

$$(x+(0+(y+(\__i*1))))\Rightarrow_T(x+(y+\__i))$$

For simplicity, we shall also abbreviate a two-step process $(E_1\Rightarrow_T E_2\Rightarrow_A^S E_3\langle t_i\rangle_{i\in N})$ by $(E_1\Rightarrow_T^S E_3\langle t_i\rangle_{i\in N})$.

As outlined in Sec. 2, we must find a common context in the presence of recursive unfolding to parallelize successfully. To formalise this idea, we introduce the notions of *context replication* and *preservation*.

Defn 10: *Recurring Context*
Consider the recursive equation of a LSR-function below.

$$f(x:xs,\lfloor v_i\rfloor_{i=1}^n)\quad=\quad Er\langle f(xs,\lfloor Dr_i\langle v_i\rangle\rfloor_{i=1}^n)\rangle \qquad (1)$$

We refer to Er (or its skeletal version) as the *recurring context* (or simply *R-context*) of the recursive call, and each Dr_i (or its skeletal version) as the *R-context* of the accumulative parameter, v_i.

To specially mark the *hole* of each R-context, E, we shall write it as $E=(\hat\lambda\langle\underline{\bullet}\rangle.\ldots)$. The hole $\underline{\bullet}$ is referred to as the *recurring hole* (or simply *R-hole*) of its context.

[2]This context notation may also be expressed using a restricted version of two-level lambda calculus.

Ex. 5: Consider the definition of $ascan_{\otimes}$. The skeletal R-context for its recursive call is $(\hat{\lambda}\langle\underline{\bullet}\rangle.\_\_{1} +\!\!+ \underline{\bullet})$, while that for its accumulative parameter is $(\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet} \oplus \_\_{1})$.

Defn 11: *Context Replication*
Consider the equation of (1). When its recursive call is unfolded, each of the R-contexts, will be *replicated*, as shown below.

$$\begin{aligned} & f(x:(y:xs), \lfloor v_i \rfloor_{i=1}^{n}) \\ & \quad = \sigma_1 Er\langle \sigma_2 Er\langle f(xs, \lfloor \sigma_2 Dr_i\langle \sigma_1 Dr_i\langle v_i\rangle\rangle \rfloor_{i=1}^{n})\rangle\rangle \\ & \quad = (\sigma_1 Er \circ \sigma_2 Er)\ \langle f(xs, \lfloor(\sigma_2 Dr_i \circ \sigma_1 Dr_i)\langle v_i\rangle \rfloor_{i=1}^{n})\rangle \\ & \text{where } \sigma_1 = [xs \mapsto y{:}xs];\ \sigma_2 = [x \mapsto y, \lfloor v_i \mapsto (\sigma_1 Dr_i)\langle v_i\rangle \rfloor_{i=1}^{n}] \end{aligned}$$

The R-context of the recursive call has been replicated by $(\sigma_1 Er \circ \sigma_2 Er)$, while the R-contexts of each accumulative parameter, v_i, has been replicated by $(\sigma_2 Dr_i \circ \sigma_1 Dr_i)$.

Obs. 1: *Substitution Distributes over Skeletal Context*
The skeletal structure of a context is not changed by substitution of its object variables.

$$\sigma Es(\langle t_i\rangle_{i\in N}) = Es\langle \sigma t_i\rangle_{i\in N}$$

In order to determine common contexts for the recursive call and its accumulative parameters, we expect their skeletal contexts to be preserved modulo replication. This requirement can be stated as:

Defn 12: *Context Preservation Modulo Replication*
A R-context E is said to be *preserved modulo replication* if the following holds:

$$\begin{gathered} \exists Es.\,(E \Rightarrow_{T}^{S} Es\langle t_i\rangle_{i\in N}) \wedge \\ (Es\langle \alpha_i\rangle_{i\in N} \circ Es\langle \beta_i\rangle_{i\in N} \Rightarrow_{T} Es\langle \Omega_i\rangle_{i\in N}) \end{gathered}$$

where α_i and β_i are variables, and Ω_i denote subterms not containing (including via local variables) the R-hole, $\underline{\bullet}$. The skeletal context Es is said to be the *common context* despite replication/unfolding.

Ex. 6: Consider $(\hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 + (\underline{\bullet} * \alpha_2))$. By using the associative and distributive properties of $+$ and $*$, this context can be *preserved modulo replication* since:

$$\begin{aligned} & (\hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 + (\underline{\bullet}*\alpha_2)) \circ (\hat{\lambda}\langle\underline{\bullet}\rangle.\beta_1 + (\underline{\bullet}*\beta_2)) \\ & \quad \Rightarrow_{T} \{\text{ defn of } \circ \} \\ & \qquad (\hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 + ((\beta_1 + (\underline{\bullet}*\beta_2))*\alpha_2)) \\ & \quad \Rightarrow_{T} \{\text{ distributivity of } * \text{ over } + \} \\ & \qquad (\hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 + (\beta_1*\alpha_2 + (\underline{\bullet}*\beta_2)*\alpha_2)) \\ & \quad \Rightarrow_{T} \{\text{ associativity of } * \text{ and } + \} \\ & \qquad (\hat{\lambda}\langle\underline{\bullet}\rangle.(\alpha_1 + \beta_1*\alpha_2) + \underline{\bullet}*(\beta_2*\alpha_2)) \\ & \quad \Rightarrow_{A}^{S} \{\text{ skeletal abstraction } \} \\ & \qquad (\hat{\lambda}\langle \_\_{1}, \_\_{2}\rangle.\hat{\lambda}\langle\underline{\bullet}\rangle.(\_\_{1}) + \underline{\bullet}*(\_\_{2}))\langle \alpha_1 + \beta_1*\alpha_2,\ \beta_2*\alpha_2\rangle \end{aligned}$$

5 Two Theorems for Parallelization

We propose two theorems which state sufficient conditions for the existence of a common context and show how they lead to successful parallelization. The first theorem is a special case of the second, covering a sub-class of LSR-functions known as *list catamorphism* [22], as opposed to the more general *list paramorphism*. By relying on the stated conditions for context preservations, we can show that parallel equations can be derived and later transformed to efficient parallel code by tupling, in the presence of complex accumulative parameters.

The second theorem allows auxiliary recursive calls to be present. By relying on the same context preservation requirement as Theorem 1, we can again yield suitable parallel equations. However, to guarantee efficient parallel code, we must impose extra conditions to allow the auxiliary recursive calls to be shared as parameters, and later tupled with the other functions. These two theorems provide constructive proofs for parallelization, since their inductive derivations (for synthesizing unknown functions) can be viewed as inductive proofs which follow from the stated conditions.

Theorem 1: *Context Preservation (Catamorphism)*
Consider a linear self-recursive function where $xs \notin Vars([Er, \lfloor Dr_j \rfloor_{j=1}^{n}])$.

$$\begin{aligned} f(Nil, \lfloor v_j \rfloor_{j=1}^{n}) &= E_z \\ f(x:xs, \lfloor v_j \rfloor_{j=1}^{n}) &= Er\langle f(xs, \lfloor Dr_j\langle v_j\rangle \rfloor_{j=1}^{n})\rangle \end{aligned}$$

This function can be successfully parallelized if the following context preservations hold for the recursive call, and each of its accumulative parameters.

- Preserving the R-Context of Recursive Call
$$\begin{gathered} \exists Es.\,(Er \Rightarrow_{T}^{S} Es\langle s_j\rangle_{j\in M}) \wedge \\ (Es\langle \alpha_j\rangle_{j\in M} \circ Es\langle \beta_j\rangle_{j\in M} \Rightarrow_{T}^{S} Es\langle \Omega_j\rangle_{j\in M}) \end{gathered}$$
- Preserving the R-Context of Accumulative Argument
$$\begin{gathered} \forall j \in 1..n.\, \exists Ds_j.\,(Dr_j \Rightarrow_{T}^{S} Ds_j\langle s_{j,i}\rangle_{i\in N_j}\langle v_j\rangle) \wedge \\ (Ds_j\langle \alpha_{j,i}\rangle_{i\in N_j} \circ Ds_j\langle \beta_{j,i}\rangle_{i\in N_j} \Rightarrow_{T}^{S} Ds_j\langle \Omega_{j,i}\rangle_{i\in N_j}) \end{gathered}$$

Proof : This is a special case of Theorem 2. □

Apart from a formalisation, Theorem 1 improves on our earlier result [10] in providing the ability to parallelize functions with arbitrary accumulative parameters, as long as each of them could be context preserved separately. To illustrate, consider the following example with two accumulative parameters:

$$\begin{aligned} & f(Nil, w1, w2) = Nil \\ & f(x{:}xs, w1, w2) = [x*w1*w2] +\!\!+ f(xs, w1+x, w2*w1) \end{aligned}$$

All R-contexts, $(\hat{\lambda}\langle\underline{\bullet}\rangle.\alpha_1 +\!\!+ \underline{\bullet})$, $(\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet}+\alpha_1)$ and $(\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet}*\alpha_1)$ can be preserved modulo replication. As a result, we could perform generalization and inductive derivation to obtain:

$$\begin{aligned} f(xr +\!\!+ xs, w1, w2) &= f(xr, w1, w2) +\!\!+ \\ & \qquad f(xs, w1 + uG_f(xr), w2 * uH_f(xr, w1)) \\ uG_f(xr +\!\!+ xs) &= uG_f(xr) + uG_f(xs) \\ uH_f(xr +\!\!+ xs, w1) &= uH_f(xr, w1) * uH_f(xs, w1 + uG_f(xr)) \end{aligned}$$

Despite the presence of interdependent accumulative parameters in f and uH_f, our theorem guarantees that tupling will succeed to yield efficient parallel programs. In particular, it is possible to obtain efficient code by tupling the following function. Note that $f(xs,w1,w2)$ and $uH_f(xs,w1)$ shares both xs and $w1$ as common arguments, and should have their shared parameters *synchronized* for successful tupling.

$$tupf(xs, w1, w2) = (f(xs, w1, w2), uG_f(xs), uH_f(xs, w1))$$

Our theorem shows that accumulative parameters, after parallelization, always synchronize with the recursion parameter. A detailed discussion on parameter synchronization for tupling is beyond the scope of this paper. The interested reader may refer to [9].

Theorem 2: *Context Preservation (Paramorphism)*
Consider an LSR-function where context preservation hold for Es and $\{Ds_j\}_{i=1}^{n}$:

$$\begin{array}{lll} f(Nil, \lfloor v_j \rfloor_{j=1}^{n}) & = & E_z \\ f(x:xs, \lfloor v_j \rfloor_{j=1}^{n}) & = & Es\langle s_j \rangle_{j \in M} \langle f(xs, \lfloor Ds_j \langle s_{j,i} \rangle_{i \in N_j} \langle v_j \rangle \rfloor_{j=1}^{n}) \rangle \end{array}$$

Parallel equations for this function can be derived as guaranteed by Theorem 1. Though this function can be parallelized, the presence of variable xs in $\{s_i\}_{i \in M} \cup \bigcup_{j \in 1..n} \{s_{j,i}\}_{i \in N_j}$ may lead to redundant auxiliary calls. To obtain efficient parallel program, we further require that each occurrence of xs be captured by an auxiliary recursive call which can later be abstracted as parameters of newly synthesized functions. This can be expressed by rewriting f to:

$$\begin{array}{l} f(x:xs, \lfloor v_j \rfloor_{j=1}^{n}) = [u_k \mapsto g_k(xs, \lfloor t_k \rfloor)]_{k \in P} \\ \qquad Es\langle \sigma_0\, s_i \rangle_{i \in M}\ \langle f(xs, \lfloor Ds_j \langle \sigma_0\, s_{j,i} \rangle_{i \in N_j} \langle v_j \rangle \rfloor_{j=1}^{n}) \rangle \\ \qquad\qquad \text{where } \sigma_0 = [g_k(xs, \lfloor t_k \rfloor) \mapsto u_k]_{k \in P} \end{array}$$

such that $xs \notin \{\sigma_0\, s_i\}_{i \in M} \cup \bigcup_{j \in 1..n} \{\sigma_0\, s_{j,i}\}_{i \in N_j}$. Note that $\{g_k\}_{k \in P}$ is a set of occurrences of auxiliary calls in the RHS of f, while we assume $\{g_k\}_{k \in G}$ to be a closed set of auxiliary function calls to be handled during parallization, with $P \subseteq G$. To ensure that $\{g_k\}_{k \in G}$ are efficiently shared, we further impose two conditions:

- Each call from $\{g_k\}_{k \in G}$ has been parallelized as:

  $$\begin{array}{lll} g_k(Nil, \lfloor w_k \rfloor) & = & Ez_k \\ g_k(xb +\!\!+ xs, \lfloor w_k \rfloor) & = & Eg_k\langle \lfloor g_l(xs, \lfloor w_k \rfloor) \rfloor_{l \in Q_k}, \\ & & \lfloor g_l(xb, \lfloor t_l \rfloor) \rfloor_{l \in Q'_k} \rangle \end{array}$$

  where $xs, xb \notin \lfloor t_l \rfloor$ and $Q_k, Q'_k \subseteq G$.

- It returns a non-recursive data type as result. Auxiliary calls which return large (recursive-type) data structures incur heavy communication overheads, when parallized. An example is the identity function $id(xb +\!\!+ xs) = id(xb) +\!\!+ id(xs)$, which (easily) leads to parallel but inefficient programs.

The proof of this theorem is given in the Appendix. To illustrate the theorem, consider the parallelization of *depart* given earlier. The R-context of this recursive call (after normalisation) is $(\hat{\lambda}\langle \underline{\bullet} \rangle.max(\alpha_1, \alpha_2 + \underline{\bullet}))$. This can be context preserved as follows:

$$\begin{array}{l} (\hat{\lambda}\langle \underline{\bullet} \rangle.max(\alpha_1,\alpha_2+\underline{\bullet})) \circ (\hat{\lambda}\langle \underline{\bullet} \rangle.max(\beta_1,\beta_2+\underline{\bullet})) \\ \quad \Rightarrow_T \{ \text{ definition of } \circ \} \\ \qquad (\hat{\lambda}\langle \underline{\bullet} \rangle.max(\alpha_1,\alpha_2+max(\beta_1,\beta_2+\underline{\bullet}))) \\ \quad \Rightarrow_T \{ \text{ distributivity of + over } max \} \\ \qquad (\hat{\lambda}\langle \underline{\bullet} \rangle.max(\alpha_1,max(\alpha_2+\beta_1,\alpha_2+(\beta_2+\underline{\bullet})))) \\ \quad \Rightarrow_T \{ \text{ associativity of + and } max \} \\ \qquad (\hat{\lambda}\langle \underline{\bullet} \rangle.max(max(\alpha_1,\alpha_2+\beta_1),(\alpha_2+\beta_2)+\underline{\bullet})) \\ \quad \Rightarrow_A^S \{ \text{ skeletal abstraction } \} \\ \qquad (\hat{\lambda}\langle \_{1}, \_{2} \rangle.(\hat{\lambda}\langle \underline{\bullet} \rangle.max(\_{1},\_{2}+\underline{\bullet})))\langle max(\alpha_1, \alpha_2+\beta_1) \\ \qquad\qquad , \alpha_2+\beta_2 \rangle \end{array}$$

We are thus able to apply generalization and inductive derivation. However, to share the auxiliary recursive call, we should abstract this call as a parameter of an unknown function, as shown in the following generalization:

$$\begin{array}{lll} depart(xr +\!\!+ xs) & = & max(uH_{dep}(xr, arrive(xs)), \\ & & \quad uG_{dep}(xr) + depart(xs)) \end{array}$$

Since the auxiliary function can be parallelized as:

$$arrive(xr +\!\!+ xs) = arrive(xr) + arrive(xs)$$

we are able to obtain the following equations during inductive derivation for *depart*.

$$\begin{array}{lll} uH_{dep}([(s,a)],w) & = & (s+a+w) \\ uH_{dep}(xr +\!\!+ xs, w) & = & max(uH_{dep}(xr, arrive(xs)+w), \\ & & \quad uG_{dep}(xr) + uH_{dep}(xs, w)) \\ uG_{dep}([(s,a)]) & = & s \\ uG_{dep}(xr +\!\!+ xs) & = & uG_{dep}(xr) + uG_{dep}(xs) \end{array}$$

Tupling can be applied to the four functions *depart*, *arrive*, uH_{dep} and uG_{dep} to obtain efficient parallel code. Also, the parallel equations of *depart*, uH_{dep} and uG_{dep} satisfy the auxiliary requirement of Theorem 2, and may themselves be used as auxiliary calls for other functions.

The key idea of sharing auxiliary recursive calls via parameter abstraction (during inductive derivation) is an innovation first introduced in [19]. We refine this technique by addressing the type of accumulative parameters allowable for such auxiliary functions.

6 Sample Preservable R-Contexts

We have presented two theorems which highlight sufficient conditions for deriving efficient parallel programs. This result is quite general as it covers major program schemes known to have data parallel implementation, such as scan and homomorphism. What is even more interesting is that the context preservation requirement can be weaker than the associativity requirement that is needed by homomorphism. Another useful point is that the R-contexts of recursive call and accumulative parameters can be separately checked for context preservation.

To illustrate the scope of our method, we shall examine four basic R-contexts, together with their sample program schemes listed in Figure 1.

A R-context is said to be *simple* if it has only one outer call for its R-hole. For example, the two R-contexts of $ascan_{\otimes}$ are simple ones. A R-context is said to be *nested* if its R-hole is surrounded by two or more outer calls, e.g. *poly* and *depart*. If the R-hole lies inside a conditional construct (e.g. *max*), then its R-context is said to be *conditional*. Lastly, a R-context is said to be *tupled* if it uses let-abstraction to prevent the duplicate occurrences of R-hole through local variables (which can occur multiple times). These local variables are called *proxies* for the R-hole. An example of this R-context is in *fibtup*.

These sample R-contexts are not exhaustive. In particular, we have not considered composite R-contexts with multiple features, such as those that are both conditional and tupled. In order to preserve R-contexts, we use a procedure which attempts to simplify and normalise the replicated contexts via the following heuristic.

Defn 13: *Heuristic for Context Normalisation*
To normalise and simplify R-contexts, we shall apply laws which reduce the *depths* and *occurrences* of the R-holes, wherever possible.

The operational details of this heuristic is beyond the scope of the present paper. Also, for simplicity, we shall assume that basic laws needed to normalise R-contexts will be provided by users. Sufficient conditions for preserving the four sample R-contexts are described next.

Types of R-Contexts	Example Program Schemes
Simple $\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet}\ \tilde{\otimes}\ \alpha_1$	$g_{sim}([x]) \quad = c(x)$ $g_{sim}(x{:}xs) \quad = g_{sim}(xs)\ \tilde{\otimes}\ a(x)$
Nested $\hat{\lambda}\langle\underline{\bullet}\rangle.(\underline{\bullet}\ \tilde{\otimes}\ \alpha_1)\ \tilde{\oplus}\ \alpha_2$	$g_{nest}([x]) \quad = c(x)$ $g_{nest}(x{:}xs) \quad =(g_{nest}(xs)\ \tilde{\otimes}\ a(x))\ \tilde{\oplus}\ b(x)$
Conditional $\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \alpha_1\ then\ \alpha_2\ else\ \underline{\bullet}\ \tilde{\otimes}\ \alpha_3$	$g_{if}([x]) \quad = b(x)$ $g_{if}(x{:}xs) \quad = if\ p(x)\ then\ c(x)\ else\ g_{if}(xs)\ \tilde{\otimes}\ a(x)$
Tupled $\hat{\lambda}\langle\underline{\bullet}\rangle.let\ (u,v)=\underline{\bullet}\ in$ $((u\ \tilde{\otimes}\ \alpha_1) \oplus (v\ \tilde{\otimes}\ \alpha_2) \oplus \alpha_3$ $,(u\ \tilde{\otimes}\ \alpha_4) \oplus (v\ \tilde{\otimes}\ \alpha_5) \oplus \alpha_6)$	$g_{tup}([x]) \quad = (c_1(x),\ c_2(x))$ $g_{tup}(x{:}xs) \quad =let\ (u,v)=g_{tup}(xs)\ in$ $((u\ \tilde{\otimes}\ a_{1,1}(x)) \oplus (v\ \tilde{\otimes}\ a_{1,2}(x)) \oplus b_1(x)$ $,(u\ \tilde{\otimes}\ a_{2,1}(x)) \oplus (v\ \tilde{\otimes}\ a_{2,2}(x)) \oplus b_2(x))$

Figure 1: Some Preservable R-Contexts

6.1 Simple R-Context

Consider $(\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet}\ \tilde{\otimes}\ \alpha_1)$. A law which is sufficient to preserve this R-context is:

$$(u\ \tilde{\otimes}\ v)\ \tilde{\otimes}\ w = u\ \tilde{\otimes}\ (v \otimes w) \tag{2}$$

We refer to this law as *weak associativity*. Both associative operations (e.g. $(u+v)+w = u+(v+w)$) and semi-associative operations (e.g. $(u-v)-w = u-(v+w)$) are special cases of this law. Context preservation can be shown, as follows.

$$\begin{array}{l}
(\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet}\ \tilde{\otimes}\ \alpha_1) \circ (\hat{\lambda}\langle\underline{\bullet}\rangle.\underline{\bullet}\ \tilde{\otimes}\ \beta_1) \\
\quad \Rightarrow_T \{\ \text{definition of } \circ\ \} \\
\quad\quad \hat{\lambda}\langle\underline{\bullet}\rangle.(\underline{\bullet}\ \tilde{\otimes}\ \beta_1)\ \tilde{\otimes}\ \alpha_1 \\
\quad \Rightarrow_T \{\ \text{weak-associative law of } \tilde{\otimes}\ \} \\
\quad\quad \hat{\lambda}\langle\underline{\bullet}\rangle.(\underline{\bullet}\ \tilde{\otimes}\ (\beta_1 \otimes \alpha_1) \\
\quad \Rightarrow_A^S \{\ \text{skeletal abstraction}\ \} \\
\quad\quad (\hat{\lambda}\langle _{}_1\rangle.\hat{\lambda}\langle\underline{\bullet}\rangle.(\underline{\bullet}\ \tilde{\otimes}\ _{}_1)\langle(\beta_1 \otimes \alpha_1)\rangle
\end{array}$$

The parallel equations obtained from generalization and inductive derivation are:

$$\begin{array}{rcl}
g_{sim}([x]) &=& c(x) \\
g_{sim}(xr{+\!\!+}xs) &=& g_{sim}(xs)\ \tilde{\otimes}\ uH_{sim}(xr) \\
uH_{sim}([x]) &=& a(x) \\
uH_{sim}(xr{+\!\!+}xs) &=& uH_{sim}(xs) \otimes uH_{sim}(xr)
\end{array}$$

If $\tilde{\otimes}$ is fully associative and $c(x){\equiv}a(x)$, then $uH_{sim}{\equiv}g_{sim}$. Otherwise, tupling will be used to combine the two calls. Note that law (2) is more general than semi-associativity, since $\otimes$ need not be associative. Under this scenario, the parallel version of the auxiliary uH_{sim} may return different results (exhibiting non-determinism) depending on how the input list is split. Nevertheless, as shown in our theorem, the main parallel function g_{sim} remains well-defined and correct with respect to the sequential definition. Allowing non-deterministic auxiliary computation appears to be an unexpected benefit. Finding useful examples of such parallel programs needs more investigation.

6.2 Nested R-Context

Consider a nested R-context $(\hat{\lambda}\langle\underline{\bullet}\rangle.(\underline{\bullet}\ \tilde{\otimes}\ \alpha_1)\ \tilde{\oplus}\ \alpha_2)$. A sufficient condition to preserve this context is to require that both $\tilde{\otimes}$ and $\tilde{\oplus}$ have weak-associativity, and also the following distributive property holds.

$$(u\ \tilde{\oplus}\ v)\ \tilde{\otimes}\ w = (u\ \tilde{\otimes}\ w)\ \tilde{\oplus}\ (v\ \tilde{\otimes}\ w) \tag{3}$$

Context preservation can be shown for its replicated R-contexts, as follows.

$$\begin{array}{l}
(\hat{\lambda}\langle\underline{\bullet}\rangle.(\underline{\bullet}\ \tilde{\otimes}\ \alpha_1)\ \tilde{\oplus}\ \alpha_2) \circ (\hat{\lambda}\langle\underline{\bullet}\rangle.(\underline{\bullet}\ \tilde{\otimes}\ \beta_1))\ \tilde{\oplus}\ \beta_2) \\
\quad \Rightarrow_T \{\ \text{definition of } \circ\ \} \\
\quad\quad (\hat{\lambda}\langle\underline{\bullet}\rangle.(((\underline{\bullet}\ \tilde{\otimes}\ \beta_1)\ \tilde{\oplus}\ \beta_2)\ \tilde{\otimes}\ \alpha_1)\ \tilde{\oplus}\ \alpha_2) \\
\quad \Rightarrow_T \{\ \text{distributive law } (u\ \tilde{\oplus}\ v)\ \tilde{\otimes}\ w = (u\ \tilde{\otimes}\ w)\ \tilde{\oplus}\ (v\ \tilde{\otimes}\ w)\ \} \\
\quad\quad (\hat{\lambda}\langle\underline{\bullet}\rangle.(((\underline{\bullet}\ \tilde{\otimes}\ \beta_1)\ \tilde{\otimes}\ \alpha_1)\ \tilde{\oplus}\ (\beta_2\ \tilde{\otimes}\ \alpha_1))\ \tilde{\oplus}\ \alpha_2) \\
\quad \Rightarrow_T \{\ \text{weak-associative law of } \tilde{\otimes}\ \} \\
\quad\quad (\hat{\lambda}\langle\underline{\bullet}\rangle.((\underline{\bullet}\ \tilde{\otimes}\ (\beta_1 \otimes \alpha_1))\ \tilde{\oplus}\ (\beta_2\ \tilde{\otimes}\ \alpha_1))\ \tilde{\oplus}\ \alpha_2) \\
\quad \Rightarrow_T \{\ \text{weak-associative law of } \tilde{\oplus}\ \} \\
\quad\quad (\hat{\lambda}\langle\underline{\bullet}\rangle.(\underline{\bullet}\ \tilde{\otimes}\ (\beta_1 \otimes \alpha_1))\ \tilde{\oplus}\ ((\beta_2\ \tilde{\otimes}\ \alpha_1) \oplus \alpha_2)) \\
\quad \Rightarrow_A^S \{\ \text{skeletal abstraction}\ \} \\
\quad\quad (\hat{\lambda}\langle _{}_1, _{}_2\rangle.\hat{\lambda}\langle\underline{\bullet}\rangle.(\underline{\bullet}\ \tilde{\otimes}\ _{}_1)\ \tilde{\oplus}\ _{}_2)\langle(\beta_1 \otimes \alpha_1), (\beta_2 \tilde{\otimes} \alpha_1) \oplus \alpha_2\rangle
\end{array}$$

The corresponding parallel equations (from generalization and inductive derivation) are:

$$\begin{array}{rcl}
g_{nest}([x]) &=& c(x) \\
g_{nest}(xr{+\!\!+}xs) &=& (g_{nest}(xs)\ \tilde{\otimes}\ uH_{nest}(xr))\ \tilde{\oplus}\ uK_{nest}(xr) \\
uH_{nest}([x]) &=& a(x) \\
uH_{nest}(xv{+\!\!+}xw) &=& uH_{nest}(xw) \otimes uH_{nest}(xv) \\
uK_{nest}([x]) &=& b(x) \\
uK_{nest}(xv{+\!\!+}xw) &=& (uK_{nest}(xw)\ \tilde{\otimes}\ uH_{nest}(xv)) \oplus uK_{nest}(xv)
\end{array}$$

If $\tilde{\oplus}$ is fully associative and $c(x){\equiv}b(x)$, we can confirm that $g_{nest}{\equiv}uK_{nest}(xs)$. Tupling can later be applied to the remaining calls to eliminate redundant computation.

6.3 Conditional R-Context

There exist rules for normalising conditional R-contexts [8]. Using them on our sample conditional R-context, we can show that context preservation holds, as follows:

$$\begin{array}{l}
(\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \alpha_1\ then\ \alpha_2\ else\ \underline{\bullet}\ \tilde{\otimes}\ \alpha_3) \circ \\
\quad\quad\quad (\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \beta_1\ then\ \beta_2\ else\ \underline{\bullet}\ \tilde{\otimes}\ \beta_3) \\
\quad \Rightarrow_T \{\ \text{definition of } \circ\ \} \\
\quad\quad (\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \alpha_1\ then\ \alpha_2\ else\ (if\ \beta_1\ then\ \beta_2\ else\ \underline{\bullet}\ \tilde{\otimes}\ \beta_3)\ \tilde{\otimes}\ \alpha_3) \\
\quad \Rightarrow_T \{\ \text{float inner } if \text{ outwards}\ \} \\
\quad\quad (\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \alpha_1\ then\ \alpha_2\ else\ (if\ \beta_1\ then\ \beta_2\ \tilde{\otimes}\ \alpha_3 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad else\ (\underline{\bullet}\ \tilde{\otimes}\ \beta_3)\ \tilde{\otimes}\ \alpha_3)) \\
\quad \Rightarrow_T \{\ \text{flatten } if\ \} \\
\quad\quad (\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \alpha_1 \vee \beta_1\ then\ (if\ \alpha_1\ then\ \alpha_2\ else\ \beta_2\ \tilde{\otimes}\ \alpha_3) \\
\quad\quad\quad\quad else\ (\underline{\bullet}\ \tilde{\otimes}\ \beta_3)\ \tilde{\otimes}\ \alpha_3)) \\
\quad \Rightarrow_T \{\ \text{weak-associative law of } \tilde{\otimes}\ \} \\
\quad\quad (\hat{\lambda}\langle\underline{\bullet}\rangle.if\ \alpha_1 \vee \beta_1\ then\ (if\ \alpha_1\ then\ \alpha_2\ else\ \beta_2\ \tilde{\otimes}\ \alpha_3) \\
\quad\quad\quad\quad else\ (\underline{\bullet}\ \tilde{\otimes}\ (\beta_3 \otimes \alpha_3)))
\end{array}$$

$\Rightarrow^S_A$ { skeletal abstraction }

$$(\hat\lambda\langle\_{1},\_{2},\_{3}\rangle.\hat\lambda\langle\underline{\bullet}\rangle.if\ \_{1}\ then\ \_{2}\ else\ (\underline{\bullet}\ \tilde\otimes\ \_{3}))$$
$$\langle\alpha_1\vee\beta_1, if\ \alpha_1\ then\ \alpha_2\ else\ \beta_2\ \tilde\otimes\ \alpha_3, \beta_3\otimes\alpha_3\rangle$$

Due to the need to preserve an outer call, we require a weak-associative law for $\tilde\otimes$. Second-order generalization and inductive derivation will then yield:

$$\begin{array}{rcl} g_{if}([x]) & = & b(x) \\ g_{if}(xr{+\!\!+}xs) & = & if\ uG_{if}(xr)\ then\ uH_{if}(xr) \\ & & else\ g_{if}(xs)\ \tilde\otimes\ uJ_{if}(xr) \\ uG_{if}([x]) & = & p(x) \\ uG_{if}(xr{+\!\!+}xs) & = & uG_{if}(xs) \vee uG_{if}(xr) \\ uJ_{if}([x]) & = & a(x) \\ uJ_{if}(xr{+\!\!+}xs) & = & uJ_{if}(xs) \otimes uJ_{if}(xr) \\ uH_{if}([x]) & = & c(x) \\ uH_{if}(xr{+\!\!+}xs) & = & if\ uG_{if}(xr)\ then\ uH_{if}(xr) \\ & & else\ uH_{if}(xs)\ \tilde\otimes\ uJ_{if}(xv) \end{array}$$

If $b(x){\equiv}c(x)$, we can confirm that $uH_{if}{\equiv}g_{if}$. Also, tupling can be applied to the remaining functions to obtain efficient parallel code.

6.4 Tupled R-Context

Tupled R-context can also be context preserved. However, due to the duplication of its proxy variables, we must minimise their occurrences for context preservation to hold. Consider:

$$\hat\lambda\langle\underline{\bullet}\rangle.let\ (u,v){=}\underline{\bullet}\ in$$
$$((u\ \tilde\otimes\ \alpha_1) \oplus (v\ \tilde\otimes\ \alpha_2) \oplus \alpha_3, (u\ \tilde\otimes\ \alpha_4) \oplus (v\ \tilde\otimes\ \alpha_5) \oplus \alpha_6)$$

For context preservation, we require weak-associativity for $\tilde\otimes$, and also associativity and commutativity of $\oplus$ for re-arranging the sub-terms. In addition, we need the following law to help reduce duplicated occurrences of the proxy variables (matching w).

$$(w\ \tilde\otimes\ a) \oplus (w\ \tilde\otimes\ b) = w\ \tilde\otimes\ (a \oplus b) \qquad (4)$$

This law is actually a converse of (3). Nevertheless, no looping occurs since we rely on the heuristic in Defn 13 to apply the laws selectively. Context preservation of this tupled R-context is left as an exercise.

7 Beyond Homomorphisms

One advantage of relying on a set of small but expressive transformation rules for our method is that it can be made widely applicable. This section summarises how efficient parallel programs, outside of homomorphism, could be derived too. We have already seen (in Sec 2 & 5) how functions with accumulative parameters could be parallelized, as long as context preservation also holds for their R-contexts. Similarly, if our sequential programs use auxiliary functions, we can easily adopt them as parameters of parent functions during parallelization. As stated in Sec. 5, as long as the auxiliary function themselves could be parallelized, their parent functions can also be efficiently parallelized if context preservation holds for their R-contexts. This again yields efficient parallel programs outside of homomorphism.

Thus far, we have only considered list paramorphic programs without nested, nor multiple recursion parameters. As mentioned earlier, LSR-functions with nested and/or multiple recursion parameters could be parallelized. The context preservation requirement is unchanged. However, we must place some constraints on how the input lists are being be split. Consider:

$$\begin{array}{ll} second(Nil,w) & = Nil \\ second([x],w) & = Nil \\ second(x{:}y{:}xs,w) & = [(y+(w+x))]{+\!\!+}second(xs,w+x) \\ vprod(Nil,Nil) & = Nil \\ vprod(x{:}xs,y{:}ys) & = [x{*}y]{+\!\!+}vprod(xs,ys) \end{array}$$

Two R-contexts $(\hat\lambda\langle\underline{\bullet}\rangle.\alpha_1{+\!\!+}\underline{\bullet})$ and $(\hat\lambda\langle\underline{\bullet}\rangle.\alpha_1+\underline{\bullet})$ of *second*, and the sole R-context of *vprod* each satisfy the context preservation property. Thus, parallelization should be possible. The only difficulty is that the recursion parameter of *second* is nested two constructors deep, while *vprod* uses two recursion parameters. As a result, the recursive call of *second* must unroll down its input list in *multiple of two* elements, while the *vprod* call must unroll its two input lists in *tandem*. To capture this requirement, we modify the generalization procedure for the LHS pattern as follows.

Defn 14: *Constrained List Decomposition*
Given a LHS pattern $[v_1,\ldots,v_c]{+\!\!+}xs$ where c is a constant, we shall generalize it as a constrained decomposition $xr^{|cn|}{+\!\!+}xs$ where the suffix $|cn|$ on xr denotes the constraints: (i) $length(xr){=}c{\times}n$, and (ii) n is an integer variable representing the number of recursive unrolling.

Applying this generalization, we obtain:

$$\begin{array}{ll} second(xr^{|2n|}{+\!\!+}xs,w) & = uH_{sec}(xr){+\!\!+} \\ & \quad second(xs,w+uG_{sec}(xr)) \\ vprod(xr^{|n|}{+\!\!+}xs,yr^{|n|}{+\!\!+}ys) & = uH_{vprod}(xr,yr){+\!\!+}vprod(xs,ys) \end{array}$$

Inductive derivation could then yield:

$$\begin{array}{ll} uH_{sec}([x,y],w) & = [y+(w+x)] \\ uH_{sec}(xr^{|2n|}{+\!\!+}xs,w) & = uH_{sec}(xr,w){+\!\!+} \\ & \quad uH_{sec}(xs,w+uG_{sec}(xr) \\ uG_{sec}([x,y]) & = x \\ uG_{sec}(xr^{|2n|}{+\!\!+}xs) & = uG_{sec}(xr){+\!\!+}uG_{sec}(xs) \\ uH_{vprod}([x],[y]) & = [x{*}y] \\ uH_{vprod}(xr^{|n|}{+\!\!+}xs,yr^{|n|}{+\!\!+}ys) & = uH_{vprod}(xr,yr){+\!\!+} \\ & \quad uH_{vprod}(xs,ys) \end{array}$$

There is a requirement for length of xr in both *second* and uG_{sec} to be a multiple of two, while the length of xr and yr in *vprod* and uH_{vprod} must be both equal to n. By induction, the length of xs in uG_{sec} will also be a multiple of two, while the length of xs and ys in uH_{vprod} would be identical. (This constrained splitting can be easily implemented after our input lists have been converted to the more efficient array counterparts. The data-type conversion itself shall not be described here.) Thus, sequential programs outside list-paramorphism can be parallelized to yield non-homomorphic programs.

8 Related Work

Generic program schemes have been advocated for use in structured parallel programming, both for imperative programs expressed as linear recurrences through a classic result of [26] and for functional programs via append-list homomorphism [25]. While extremely versatile, most sequential specifications fail to match up directly with these schemes.

An approach to this problem is to constructively transform user programs to match these schemes. However, direct calculations of homomorphic functions often require deep intuition and the support of ad-hoc lemmas [24, 17], in order to massage the sequential definitions into having the needed associative combine operator. An alternative approach is to have programmers supply both a leftward (cons-based) and a rightward (snoc-based) versions of the sequential program. This method was systematically formulated by [15] based on an earlier idea from Bird & Meertens, and later given a constructive procedure by Gorlatch [16]. As shown here, it is possible to parallelize systematically using only a leftward sequential program.

There is also tremendous interests in the parallelization of loops for imperative languages (e.g. Fortran). A work which is similar to our method was independently conceived by Fischer & Ghoulum [13, 14]. By modelling loops via functions, they noted that these function-type values could be reduced (in parallel) via associative function composition. However, the propagated function-type values can only be efficiently propagated (by parallel reduction) if they have a template form closed under composition. This requirement is similar to the need to find a common context under recursive call unfolding [6]. Compared to our new work, their framework is somewhat informal and considerably less general. In particular, loops only correspond to tail-recursive functions with accumulative parameters. By using a single modeling function for all the accumulators, they achieve less precise outcomes.

Instead of deriving parallel programs from first principle (e.g. via context preservation modulo replication), another approach for improving parallelization is to develop more specialised program schemes that could provide a better match up with common sequential programs. A step in this direction has recently been formalised in [19]. By relying on a novel synthesis lemma (which combines the requirements of both generalization and inductive derivation), a good set of commonly used schemes can be recognized by a parallelization algorithm in calculating parallel programs. The main advantage of this approach is a cheap and practical parallelization algorithm, at some expense to generality.

9 Conclusion

We have formally presented a method for parallelizing a class of sequential programs. The method relies on the successful preservation (modulo replication/unfolding) of the R-contexts of recursive call and each accumulative argument, before generalization and inductive derivation could yield corresponding parallel equations. A unique characteristic of our method is that it can automatically invent auxiliary functions needed for parallelization. Sufficient conditions for deriving efficient parallel equations (with the help of tupling) are described in two key theorems.

The notion of context preservation is central to our parallelization method. Useful laws needed to ensure this can either be given by users or be directly synthesized. A suitable procedure for applying these laws is based on the need to preserve the desired R-context. Apart from the heuristic of minimising both the depths and occurrences of R-holes, we can also turn to a technique, called *rippling* [5], which has been very successful in automated theorem-proving for guiding the step case of inductive proofs.

Our method is unique in its ability to synthesize parallel programs both within and beyond homomorphism. In addition, it may also be possible for us to recover from failures when a particular R-context could not be preserved. In particular, the resulting context may suggest either a *new* or *generalized* R-context to be attempted. We leave this topic for future investigation. This much enhanced potential for parallelization is made possible by our adoption of small but expressive transformation rules, together with appropriate theorems and strategies for guiding their applications.

Acknowledgements: Discussions with Arne Glenstrup, Masami Hagiya, and Masato Takeichi have helped shaped several ideas in this paper. Thanks also to ICCL referees for helpful comments.

References

[1] K. Achatz and W. Schulte. Massive parallelization of divide-and-conquer algorithms over powerlists. *Science of Computer Programming*, 26:59–78, 1996.

[2] Richard S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design (Springer Verlag, ed M Broy)*, pages 3–42, 1987.

[3] G. Blelloch, G. L. Miller, and D. Talmor. Developing a practical projection-based parallel Delaunay algorithm. In *Proceedings ACM Symposium on Computational Geometry*, May 1996.

[4] G.E. Blelloch, S Chatterjee, J.C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *4th Principles and Practice of Parallel Programming*, pages 102–111, San Diego, California (ACM Press), May 1993.

[5] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[6] Wei-Ngan Chin. Synthesizing parallel lemma. In *Proc of a JSPS Seminar on Parallel Programming Systems, World Scientific Publishing*, pages 201–217, Tokyo, Japan, May 1992.

[7] Wei-Ngan Chin. Towards an automated tupling strategy. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, June 1993. ACM Press.

[8] W.N. Chin, J Darlington, and Y. Guo. Parallelizing conditional recurrences. In *2nd Annual EuroPar Conference*, Lyon, France, (LNCS 1123) Berlin Heidelberg New York: Springer, August 1996.

[9] W.N. Chin, S.C. Khoo, and P. Thiemann. Synchronisation analyses for multiple recursion parameters. In *Intl Dagstuhl Seminar on Partial Evaluation (LNCS 1110)*, pages 33–53, Germany, February 1996.

[10] W.N. Chin, S.H. Tan, and Y.M. Teo. Deriving efficient parallel programs for complex recurrences. In *2nd Intl Conference on Parallel Symbolic Computation*, pages 101–110, Maui, Hawaii, July 1997. ACM Press.

[11] Murray I. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–203, 1995.

[12] J. Darlington, Y. Guo, HW. To, and J. Yang. Parallel skeletons for structured composition. In *ACM PPoPP*, pages 19–28, Santa Barbara, California, ACM Press, 1995.

[13] A.L. Fischer and A.M. Ghuloum. Parallelizing complex scans and reductions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 135–136, Orlando, Florida, ACM Press, 1994.

[14] A.M. Ghuloum and A.L. Fischer. Flattening and parallelizing irregular applications, recurrent loop nests. In *3rd ACM Principles and Practice of Parallel Programming*, pages 58–67, Santa Barbara, California, ACM Press, 1995.

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

[16] Sergei Gorlatch. Extracting and implementing list homomorphisms in parallel program development. *To appear in Science of Computer Programming*, 1998.

[17] Z.N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.

[18] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple traversals. In *2nd ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, Netherlands, June 1997. ACM Press.

[19] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th Annual ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, January 1998. ACM Press.

[20] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[21] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[22] E. Meijer, MM. Fokkinga, and RA Paterson. Functional programming with bananas, lenses, envelopes and barded wires. In *5th ACM Functional programming Languages and Computer Architecture Conference*, Cambridge, Massachusetts (LNCS, vol 523, pp. 124–144), August 1991.

[23] Pushpa Rao and Clifford Walinsky. An equational language for data parallelism. In *4th Principles and Practice of Parallel Programming*, pages 112–118, San Diego, California (ACM Press), May 1993.

[24] Paul Roe. *Parallel Programming using Functional Languages (Report CSC 91/R3)*. PhD thesis, University of Glasgow, 1991.

[25] D. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.

[26] Harold S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1(4):287–307, 1975.

[27] Phil Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, Nancy, France, (LNCS, vol 300, pp. 344–358), March 1988.

A Proof of Theorem 2

Assume the second-order generalization below with the auxiliary recursive calls extracted out as arguments of the unknown functions.

$$\begin{aligned}&f(xr+\!\!+xs, \lfloor v_i \rfloor_{i=1}^{n}) = [u_k \mapsto g_k(xs, \lfloor t'_k \rfloor)]_{k\in P}\, Es\langle uH_i(xr, \lfloor w_i \rfloor)\rangle_{i\in M} \\ &\quad \langle f(xs, \lfloor Ds_j\langle uG_{j,i}(xr, \lfloor w_{j,i}\rfloor)\rangle_{i\in N_j}\langle v_j\rangle\rfloor_{j=1}^{n})\rangle\end{aligned}$$

Note that $\{\lfloor w_i \rfloor\}_{i\in M}$ draws from $\{v_i\}_{i=1}^{n} q \cup \{u_k\}_{k\in P}$, while $\{\{\lfloor w_{j,i}\rfloor\}_{i\in N_j}\}_{j=1}^{n}$ draws from $\{v_i\}_{i=1}^{n} - \{v_j\} \cup \{u_k\}_{k\in P}$. Inductive derivation of base and recursive cases follows:

$$\begin{aligned}&f([x]+\!\!+xs, \lfloor v_j \rfloor_{j=1}^{n}) = [u_k \mapsto g_k(xs, \lfloor t'_k \rfloor)]_{k\in P}\, Es\langle uH_i([x], \lfloor w_i \rfloor)\rangle_{i\in M} \\ &\quad \langle f(xs, \lfloor Ds_j\langle uG_{j,i}([x], \lfloor w_{j,i}\rfloor)\rangle_{i\in N_j}\langle v_j\rangle\rfloor_{j=1}^{n})\rangle \\ &LHS = \{\text{ unfold } f(x:xs,\ldots)\} \\ &\qquad [u_k \mapsto g_k(xs, \lfloor t'_k \rfloor)]_{k\in P} Es\langle \sigma_0 s_i\rangle_{i\in M} \\ &\qquad\quad \langle f(xs, \lfloor Ds_j\langle \sigma_0 s_{j,i}\rangle_{i\in N_j}\langle v_j\rangle\rfloor_{j=1}^{n})\rangle \\ &f((xa+\!\!+xb)+\!\!+xs), \lfloor v_j\rfloor_{j=1}^{n}) \\ &\quad = [u_k \mapsto g_k(xs, \lfloor t'_k \rfloor)]_{k\in P} Es\langle uH_i(xa+\!\!+xb, \lfloor w_i\rfloor)\rangle_{i\in M} \\ &\qquad\quad \langle f(xs, \lfloor Ds_j\langle uG_{j,i}(xa+\!\!+xb, \lfloor w_{j,i}\rfloor)\rangle_{i\in N_j}\langle v_j\rangle\rfloor_{j=1}^{n})\rangle \\ &LHS = \{\text{ associativity of } +\!\!+\} \\ &\qquad f(xa+\!\!+(xb+\!\!+xs), \lfloor v_j\rfloor_{j=1}^{n}) \\ &\quad = \{\text{ unfold } f(xa+\!\!+(xb+\!\!+xs),\ldots)\ \} \\ &\qquad [u'_k \mapsto g_k(xb+\!\!+xs, \lfloor t'_k \rfloor)]_{k\in P} Es\langle uH_i(xa, \sigma_1\lfloor w_i\rfloor)\rangle_{i\in M} \\ &\qquad\qquad \langle f(xb+\!\!+xs, \lfloor Ds_j\langle uG_{j,i}(xa, \sigma_1\lfloor w_{j,i}\rfloor)\rangle_{i\in N_j}\langle v_j\rangle\rfloor_{j=1}^{n})\rangle \\ &\quad = \{\text{ unfold } f(xb+\!\!+xs,\ldots)\ \} \\ &\qquad [u'_k \mapsto g_k(xb+\!\!+xs, \lfloor t'_k \rfloor)]_{k\in P} Es\langle uH_i(xa, \sigma_1\lfloor w_i\rfloor)\rangle_{i\in M} \\ &\qquad\quad \langle [u_k \mapsto g_k(xs, \lfloor t'_k \rfloor)]_{k\in P} Es\langle uH_i(xb, \lfloor \sigma_2\, w_i\rfloor)\rangle_{i\in M} \\ &\qquad\qquad \langle f(xs, \lfloor Ds_j\langle uG_{j,i}(xb, \lfloor \sigma_2 w_{j,i}\rfloor)\rangle_{i\in N_j} \\ &\qquad\qquad \langle Ds_j\langle uG_{j,i}(xa, \sigma_1\lfloor w_{j,i}\rfloor)\rangle_{i\in N_j}\langle v_j\rangle\rangle\ \rfloor_{j=1}^{n})\ \rangle\rangle \\ &\quad = \{\text{ context preservations for } Es \text{ and } \{Ds_j\}_{j=1}^{n}\ \} \\ &\qquad [u'_k \mapsto g_k(xb+\!\!+xs, \lfloor t'_k \rfloor)]_{k\in P} \circ [u_k \mapsto g_k(xs, \lfloor t'_k \rfloor)]_{k\in P} \\ &\qquad\qquad Es\langle \theta_0\, \Omega_i\rangle_{i\in M}\ \langle f(xs, \lfloor Ds_j\langle \theta_j\, \Omega_{j,i}\rangle_{i\in N_j}\langle v_j\rangle\rfloor_{j=1}^{n})\rangle \\ &\quad = \{\text{ unfold } g_k(xb+\!\!+xs,\ldots) \text{ calls }\} \\ &\qquad [u'_k \mapsto Eg_k\langle \lfloor g_l(xs, \lfloor t'_l\rfloor)\rfloor_{l\in Q_k}, \lfloor g_l(xb, \phi_l\lfloor t_l\rfloor)\rfloor_{l\in Q'_k}\rangle]_{k\in P} \circ \\ &\qquad\quad [u_k \mapsto g_k(xs, \lfloor t'_k \rfloor)]_{k\in P} \\ &\qquad\qquad Es\langle \theta_0\, \Omega_i\rangle_{i\in M}\ \langle f(xs, \lfloor Ds_j\langle \theta_j\, \Omega_{j,i}\rangle_{i\in N_j}\langle v_j\rangle\rfloor_{j=1}^{n})\rangle \\ &\quad = \{\text{ abstract } g_k(xs,\ldots) \text{ calls }\} \\ &\qquad [u_k \mapsto g_k(xs, \lfloor t'_k \rfloor)]_{k\in P} Es\langle \sigma_3 \circ \theta_0\, \Omega_i\rangle_{i\in M} \\ &\qquad\quad \langle f(xs, \lfloor Ds_j\langle \sigma_3 \circ \theta_j\, \Omega_{j,i}\rangle_{i\in N_j}\langle v_j\rangle\rfloor_{j=1}^{n})\rangle\end{aligned}$$

The substitutions used are:

$$\begin{aligned}\sigma_1 &= [u_k \mapsto u'_k]_{k\in P} \\ \sigma_2 &= [v_j \mapsto Ds_j\langle uG_{j,i}(xa, \sigma_1\lfloor w_{j,i}\rfloor)\rangle_{i\in N_j}\langle v_j\rangle]_{j=1}^{n} \\ \sigma_3 &= [u'_k \mapsto Eg_k\langle \lfloor u_l\rfloor_{l\in Q_k}, \lfloor g_l(xb, \phi_l\lfloor t'_l\rfloor)\rfloor_{l\in Q'_k}\rangle]_{k\in P} \\ \phi_l &= [\lfloor w_l \mapsto t'_l\rfloor]\ \forall l \in P \\ \theta_0 &= [\alpha_i \mapsto uH_i(xa, \sigma_1\lfloor w_i\rfloor)]_{i\in M} \circ [\alpha'_i \mapsto uH_i(xb, \sigma_2\lfloor w_i\rfloor)]_{i\in M} \\ \theta_j &= [\alpha_{j,i} \mapsto uG_{j,i}(xb, \sigma_2\lfloor w_{j,i}\rfloor)]_{i\in N_j} \circ \\ &\qquad [\alpha'_{j,i} \mapsto uG_{j,i}(xa, \sigma_1\lfloor w_{j,i}\rfloor)]_{i\in N_j}\ \forall j \in 1..n\end{aligned}$$

Unifying the outcomes from LHS and RHS yield the following parallel equations:

$$\begin{aligned}uH_i(xa+\!\!+xb, \lfloor w_i\rfloor) &= \sigma_3 \circ \theta_0\Omega_i\ \forall i \in M \\ uG_{j,i}(xa+\!\!+xb, \lfloor w_{j,i}\rfloor) &= \sigma_3 \circ \theta_j\Omega_{j,i}\ \forall i \in N_j\ \forall j \in 1..n\end{aligned}$$

Occurrences of $\{uH_i(xa, \sigma_1\lfloor w_i\rfloor)\}_{i\in M}$ & $\{uH_i(xb, \sigma_2\lfloor w_i\rfloor)\}_{i\in M}$ may occur in $\{\sigma_3 \circ \theta_0\, \Omega_i\}_{i\in M}$, while the function calls from $\{uG_{j,i}(xa, \sigma_1\lfloor w_{j,i}\rfloor)\}_{i\in N_j}$ and $\{uG_{j,i}(xb, \sigma_2\lfloor w_{j,i}\rfloor)\}_{i\in N_j}$ may occur in $\{\sigma_3 \circ \theta_j\, \Omega_{j,i}\}_{i\in N_j}$. From these calls, it would appear that the new definitions may have accumulative parameters of the form, $\{\sigma_2\lfloor w_i\rfloor\}_{i\in M}$ and $\bigcup_{j\in 1..n}\{\sigma_2\lfloor w_{j,i}\rfloor\}_{i\in N_j}$, for calls with the xb recursion argument. However, these accumulators are drawn from $\{Ds_j\langle uG_{j,i}(xa, \sigma_1\lfloor w_{j,i}\rfloor)\rangle_{i\in N_j}\langle v_j\rangle\}_{j\in 1..n}$ in σ_2. Due to this restriction, each accumulative parameter of the synthesized functions always synchronize with the recursion parameter. Hence, they could be successfully tupled [9]. Similarly, the auxiliary recursive functions $\{g_k\}_{k\in P}$ and their calls have the same synchronizable characteristics. Hence, tupling will succeed in its elimination of the redundant calls of $\{g_k\}_{k\in P}, \{uH_i\}_{i\in M}$ and $\bigcup_{j\in 1..n}\{uG_{j,i}\}_{i\in N_j}$ to yield a efficient parallel program.