Deriving Efficient Parallel Programs for Complex Recurrences

W.N. CHIN*
Hitachi Advanced Research Laboratory &
National University of Singapore

S.H. TAN and Y.M. TEO
National University of Singapore

Abstract

We propose a method to synthesize parallel divide-and-conquer programs from non-trivial sequential recurrences. Traditionally, such derivation methods are based on schematic rules which attempt to match each given sequential program to a prescribed set of program schemes that have parallel counterparts. Instead of relying on specialized program schemes, we propose a new approach to parallelization based on techniques built using elementary transformation rules.

Our approach requires an induction to recover parallelism from sequential programs. To achieve this, we apply a second-order generalisation step to selected instances of sequential equations, before an *inductive derivation* procedure. The new approach is systematic enough to be semi-automated, and shall be shown to be widely applicable using a range of examples.

1 Introduction

Most programs are more easily written via sequential specifications. Functional programs are no exception. As an example, the ubiquitous list data structure used in functional programming is naturally defined as a sequential data structure. With it, many user-defined functions are directly expressed in their sequential form. A simple example is given below, where ':' is the infix version of *'Cons'* for the *List* data type.

```
data List a = Nil  | a : (List a) ;
sum(Nil,c)         = 0;
sum(x:xs,c)        = c+(x+sum(xs,c));
```

The two pattern-matching equations of *sum* are for a base case and a recursive case. In the latter case, the result is computed sequentially via repeated invocations of the recursive call. To make this example more interesting, we have added an extra (constant) parameter to the summation function in order to generate $sum([a_1,..,a_n],c) = \sum_{i=1}^{n} a_i + (c \times n)$.

*Part of this work was done while the first author was visiting Institute of Information Science, Academia Sinica, Taiwan.

To obtain a parallel version of *sum*, we would have to express it using an *Append*-list (rather than *Cons*-list) data structure, as shown below:

```
sum(Nil,c)         = 0;
sum([x],c)         = (c+x);
sum(xr++xs,c)      = sum(xr,c) + sum(xs,c);
```

Note that ++ is normally for list-concatenation. However, when used as a LHS pattern of an equation, we shall assume that it is executed backwards to split an input list into two sub-lists. For balanced parallelism, we may require that the splitted sub-lists be of about equal sizes, say $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, where n is the size of the list.

One noticeable problem remains: *lists cannot be split efficiently.* However, with the parallel equation, it is quite easy to apply a data type transformation to change the *sum* function to use the *array* type (or *binary tree* type) with constant-time split operation. In fact, Rao and Walinsky [RW93] even treated the above form of pattern-matching as syntactic sugar for the *array*-type, where *xr++xs* would denote the splitting of an *n*-item array into two sub-arrays, *xr* and *xs*, of sizes 2^m and $n - 2^m$ respectively where $2^m < n \leq 2^{m+1}$. Parallel data types, such as *arrays* or *binary tree*, can be introduced to completely replace the sequential *List*-type if efficient construction (or retrieval) of the input data type via the *Cons* operation is not needed. Otherwise, we can allow the parallel data type to co-exist with the *List*-data type, in order to support our parallel algorithms.

Another point to note is that we are primarily concerned with obtaining abstract divide-and-conquer algorithms from sequential specifications in this paper. We do *not* deal with the issue of mapping these programs to particular parallel architectures, nor provide suitable cost models to justify the synthesized programs. A number of other works, such as [AS96, PP93], have dealt with the more intricate implementation issue using divide-and-conquer programs as starting points. These works are therefore complimentary to our proposal. We intend to explore some of these techniques for a more complete treatment of this work, in the near future.

Our present contribution is an enhanced method capable of parallelizing complex sequential recurrences, expressed via recursive functions. It is based on equational rules of Burstall and Darlington [BD77], augmented with a *second-order generalisation* step and an *inductive derivation* procedure. Given a sequential function (with suitable properties), the proposed method can systematically derive the function's parallel equations. The purpose of this paper is to describe the parallelization method and to show that it is applicable across a reasonably wide range of programs. In particular, we can handle sequential recursive functions which are often regarded as difficult to parallelize, including those with accumulative parameters, nested recurrences, conditional constructs, and non-linear recurrences with multiple recursive calls. Where applicable, new auxiliary function definitions are automatically synthesized by our method.

Section 2 presents a simple language, and some classification schemes for functions and parameters. In Section 3, we introduce our parallelization method and highlight the key techniques of (i) getting a desired pre-parallel form, (ii) applying second-order generalisation, and (iii) using inductive derivation for unknown functions. Sections 4 outlines the scope of the proposed method. Section 5 to 8 highlight the method through parallelizing programs with *accumulative parameters*, *nested recurrences*, *non-linear recurrences*, and *conditional recurrences*, respectively. Lastly, some related works are discussed in Section 9, followed by a conclusion in Section 10.

2 Language and Terminologies

We consider a strict first-order functional language:

Defn 1: *A Simple Language*
Components of our simple language include:

$$\begin{array}{lll} P & ::= [M_i]_{i=0}^{n} & \text{(Program)} \\ M & ::= [F_i]_{i=0}^{n} & \text{(Mutual Rec. Set)} \\ F & ::= \{f(p_{i1},\ldots,p_{in}) = t_i\}_{i=0}^{m} & \text{(Set of Equations)} \\ t & ::= v \mid C(t_1,\ldots,t_n) \mid f(t_1,\ldots,t_n) & \\ & \quad\quad \mid \mathit{let}\ \{p_i = t_i\}_{i=0}^{n}\ \mathit{in}\ t & \\ & \quad\quad \mid \mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3 & \text{(Expression)} \\ p & ::= v \mid C(p_1,\ldots,p_n) & \text{(Pattern)} \end{array}$$

□

Expressions of this language include variables (v), data constructors (C), functions (f), *if* and *let* constructs. Each function f is defined using a set of pattern-matching equations. A number of such functions are collected into each mutual-recursive set (denoted by M), which in turn is part of the main program (denoted by P).

We also introduce a special context notation with multiple holes of the form $\widehat{e}\langle t_1,\ldots,t_n\rangle$ where $\widehat{e}\langle\rangle$ is the expression context, and $[t_i]_{i\in 1..n}$ are the sub-terms abstracted from the m-holes, $[\langle\rangle_i]_{i\in 1..n}$. Specifically, $\widehat{e}\langle t_i\rangle_{i\in 1..n} \equiv (\widehat{e}\langle\rangle)\,[t_i/\langle\rangle_i]_{i\in 1..n}$ which stands for the direct substitution of sub-terms, $t_1,\ldots,t_n$, into their respective holes, $\langle\rangle_1,\ldots,\langle\rangle_n$, for context $\widehat{e}\langle\rangle$. There is a subtle difference between *context holes* and *variables*. When substituting *variables*, we must perform variable renaming in order to avoid *name clashes*; but this consideration is omitted for *holes*.

Sequential functions are often expressed as either *linear* or *non-linear* recurrences, and may be either *self*, *mutual* or *auxiliary* recursive. If they contain conditional constructs, prior to their recursive calls, we also refer to them as *conditional* recurrences. These classifications of functions are defined next.

Defn 2: *Self, Auxiliary & Mutual Recurrence*
A function, *f*, is said to be a *self-recurrence* if its *recursive set* of functions, from its call graph, is simply {*f*} itself. It is said to be *mutual-recurrence* if its *recursive set* of functions contain other functions too. Lastly, it is said to be an *auxiliary recurrence* if we can include auxiliary functions, as part of its recursive set. □

For example, consider some sequential functions in Figure 1, and its corresponding call graph in Figure 2. Functions *label* and *comp* are self-recurrences, since their mutual-recursive sets consist of just the functions themselves. Function *depart* can also be considered as a self-recurrence, since *arrive* is *not* a mutual recursive call. However, *arrive* shares a common recursion argument with *depart*. As a result, we can nominate this function as an *auxiliary recursive* call of *depart*. Hence, *depart* can be classified as an auxiliary recurrence instead.

Also, function *mul_add* has a conditional construct prior to its recursive calls. We refer to this category of functions as conditional recurrences. Formally:

Defn 3: *Conditional Recurrence*
An equation is said to be *conditional recursive* if there exists one or more conditional construct(s) *prior* to its *recursive call(s)*. In other words, recursive call(s) exist as sub-term(s) of the outer conditional construct. A function definition is said to be a *conditional recurrence* if at least one of its equation is *conditional recursive*. □

Recurrences are also frequently classified based on the number of recursive calls in the RHS of their function definition, as follows.

Defn 4: *Linear & Non-Linear Recurrence*
An equation is said to be *linear recursive* if it has exactly one *recursive* call in its RHS. It is said to be *non-linear recursive* if it contains more than one *recursive call* in its RHS.
A function definition is said to be a *linear recurrence* if it has only one *linear recursive* equation, while the other equations are *non-recursive*. It is said to be a *non-linear recurrence*, otherwise. □

For example, functions *label*, *comp* and *arrive* are *linear recurrences*, while *depart* and *mul_add* are *non-linear* since their equations contain multiple recursive calls. Linear recurrences are typically easier to parallelize, but our proposed synthesis method adapts well to non-linear recurrences too.

The term *recurrence* normally refers to recursive equation for numerically indexable objects, such as arrays. For example, if we capture the outputs and inputs of the simple simulation program (consisting of *arrive* and *depart* in Figure 1) using some arrays, we could rewrite the two recursive functions in the conventional recurrence form, as follows:

```
arrive[0]    = 0;
arrive[n+1]  = a[n+1] + arrive[n];
depart[0]    = 0;
depart[n+1] = s[n+1] + max(arrive[n+1], depart[n]);
```

where *arrive* and *depart* denote two output arrays for computing arrival and departure times of events, while *a* and *s* denote two input arrays for inter-arrival gaps and service times. Since recursive functions can always be used to represent such recurrences, we shall use the term *recurrences* and *recursive functions* interchangeably in this paper.

```
label(Nil,no)      = Nil;
label(x:xs,no)     = (x,no):label(xs,no+1);
comp(Nil)          = 0;
comp((x,y):xs)     = x+(y*comp(xs));
arrive(Nil)        = 0;
arrive((s,a):xs)   = a + arrive(xs);
depart(Nil)        = 0;
depart((s,a):xs)   = s + max(arrive((s,a):xs), depart(xs));
mul_add(Nil)       = 0 ;
mul_add(x:xs)      = if x<10 then x*mul_add(xs)  else x+mul_add(xs) ;
```

Figure 1: Sequential Functions which could be Parallelized

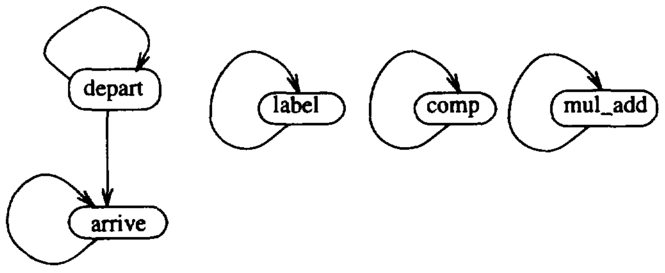

Figure 2: A Call Graph

Another important consideration for recursive functions are the parameters. In particular, we classify the parameters into two major groups - *recursion* and *non-recursion* parameters.

Defn 5: *Recursion and Non-Recursion Arguments*
Consider a *self-recursive* equation:

$$f(p_1,\ldots,p_n,v_{n+1},\ldots,v_m) \\ = \widehat{e}\langle f(vr_1,\ldots,vr_n,tr_{n+1},\ldots,tr_m)\rangle_{r\in M} \;;$$

with *recursion parameters*, $\{p_i\}_{i=1}^n$, and *non-recursion parameters*, $\{v_j\}_{j=n+1}^m$, that are disjoint (i.e. $\forall r \in M.(\forall i \in 1..n.\ vr_i \not\sqsubseteq \{tr_{n+1},\ldots,tr_m\}))$.
We introduce the following parameter classifications:

1. Recursion arguments from $\bigcup_{r\in M}\{vr_i\}_{i=1}^n$ must satisfy: $\forall r \in M.\forall i \in 1..n.\ vr_i \sqsubset p_i$ where $\sqsubset$ denotes the proper sub-term relationship.

2. Non-recursion arguments $\{tr_j\}_{j=n+1}^m$ from every recursive call, are either:

    - *constant*, whereby $tr_j \equiv v_j$; or
    - *accumulative*, whereby $v_j \sqsubset tr_j \wedge NumOccurs(v_j,\{tr_i\}_{i=n+1}^m) = 1$; or
    - *roving*, whereby $IsVar(tr_j) \wedge tr_j \sqsubseteq \{p_1,\ldots,p_n,v_{n+1},\ldots,v_{j-1},v_{j+1},\ldots,v_m\} \wedge tr_j \not\sqsubseteq \bigcup_{r\in M}\{vr_i\}_{i=1}^n$.

□

Note that *NumOccurs* returns the number of variable occurrences in a sub-term, while *IsVar* tests if a given input is a variable.

For example, consider:

$$f(a:(b:cs),n+1,v_1,v_2,v_3,v_4) \\ = \widehat{e}\langle f(cs,n,v_1,v_2+1,v_4,v_3)\rangle$$

The first two parameters are *recursion* parameters, the next two parameters have *constant* and *accumulative* properties, while the last two swapping parameters have the *roving* property.

These types of non-recursion parameters are fairly common. We identify them separately, in order to allow them to be handled appropriately by our parallelization method. For example, *accumulative* parameters are usually handled more carefully than *constant* parameters, as we shall see later.

3 The Method

Our parallelization method is based on elementary transformation rules, which are in turn organised into four larger transformation steps, called *stages*, with specific objectives, as shown below.

Procedure 1: *Four Stages of Parallelization*

Stage 1: Determine a desired *pre-parallel form* for the initial recursive equation.

Stage 2: Obtain a second recursive equation with the same pre-parallel form.

Stage 3: Use *second-order generalisation* to obtain a *template equation* from the two pre-parallel equations. This template may have one or more unknown functions.

Stage 4: *Derive* the unknown functions.

The above parallelization method works essentially via generalisation from examples, as outlined in Figure 3 with *sum* as an example. The first two stages attempt to obtain two sequential equations with a form, called pre-parallel form. Though our equations are still sequential, we refer to

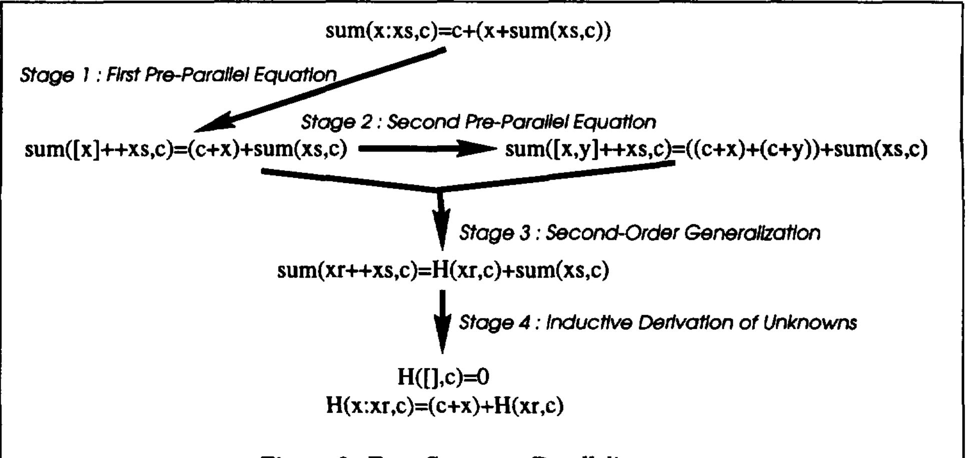
sum(x:xs,c)=c+(x+sum(xs,c))
Stage 1 : First Pre-Parallel Equation
Stage 2 : Second Pre-Parallel Equation
sum([x]++xs,c)=(c+x)+sum(xs,c)
sum([x,y]++xs,c)=((c+x)+(c+y))+sum(xs,c)
Stage 3 : Second-Order Generalization
sum(xr++xs,c)=H(xr,c)+sum(xs,c)
Stage 4 : Inductive Derivation of Unknowns
H([],c)=0
H(x:xr,c)=(c+x)+H(xr,c)

Figure 3: Four Stages to Parallelize *sum*

them as having pre-parallel form since they have structures which are close to the desired parallel equation. A couple of heuristics (conditions) are employed, as described later in Section 3.1. The next stage is effectively an induction step which attempts to recover the more general parallel equation from the two sequential (but pre-parallel) equations. This stage uses second-order generalisation to obtain an equation template with one or more unknown functions. The unknown functions are then synthesized through an inductive derivation procedure in Stage 4. For our earlier example, the unknown function H was found to be equivalent to *sum*, yielding the expected parallel equation:

$sum(xr{+}{+}xs,c) \quad = sum(xr,c) + sum(xs,c);$

Occasionally, the unknown functions may be new auxiliary functions or generalized versions of the initial functions. Under those circumstances, we have to re-apply the parallelization method in order to obtain further parallel equations. We elaborate on the main techniques of our parallelization method next.

3.1 Desired Pre-Parallel Form

The first two stages attempt to obtain two sequential equations with similar *pre-parallel* form. Two simple heuristics are used to achieve this.

Defn 6: *Heuristic Conditions for Desired Pre-Parallel Form*

(H1) All function calls (e.g. *sum*, *++*, *+*) in the LHS and RHS, leading to the recursion variables, e.g. *xs* of *sum*, should be either ***associative*** or ***distributive***.

(H2) The recursion (and accumulative) variables, e.g. *xs* of *sum*, are significant. Its depth[a] from the root of the LHS and RHS should be as ***shallow*** as possible.

□

[a]Depth is defined to be the distance from the root of an expression tree. For example, the depths of variable occurrences c,x,xs,c in $(c+(x+sum(xs,c)))$ are 1,2,3,3 respectively.

Our method targets divide-&-conquer equation with *++* as the split operator (for *List*-type). As this split operator is associative, the first condition is needed to help obtain a matching pre-parallel form. Also, the second condition helps improve the chance of successful parallelization by reducing the number of function operators leading to the recursion variables. As these are required to be associative (or distributive), the fewer the better. This second condition also minimises the number of unknown functions which might arise.

These conditions are ***heuristic*** in nature, since they do not guarantee the presence of similar pre-parallel equations (for a later generalisation stage), nor necessarily give only a single acceptable outcome.

Guided by heuristic conditions (H1) and (H2), we can transform the equation of *sum*, as follows:

$$\begin{aligned} LHS &= sum(x{:}xs,c) \\ &\quad ;\ \text{(H1) replace } cons \text{ by associative } {+}{+} \\ &= sum([x]{+}{+}xs,c) \\ RHS &= c+(x+sum(xs,c)) \\ &\quad ;\ \text{(H2) reduce depth of } xs \text{ from 3 to 2} \\ &= (c+x)+sum(xs,c) \end{aligned}$$

Hence, a suitable pre-parallel equation is:

$$sum(\underline{[x]}{+}{+}xs,\underline{c}) = \underline{(c+x)} + sum(xs,\underline{c})$$

Similarly, a second pre-parallel equation can be obtained by first unfolding the recursive call and then guided by the two heuristics, to obtain:

$$sum(\underline{([x]{+}{+}[y])}{+}{+}xs),\underline{c}) = \underline{((c+x)+(c+y))}+sum(xs,\underline{c});$$

The above two equations are identical, except for subterms underlined. The underlined sub-expressions are known as ***abstractable subterms***, while the common skeletal structure is known as a ***pre-parallel context***.

Defn 7: *Pre-Parallel Context & Abstractable Subterms*

Each *maximal* sub-term, not including any recursion (or accumulative) variables, shall be known as a ***abstractable sub-term***. A sub-term is said to be *maximal* with respect to a given property, if it is not contained inside another term with the same property.

Given an expression (or equation) e, we can decompose it via $\widehat{e^P}\langle h_i\rangle_{i\in 1..n}$ where $\widehat{e^P}\langle\rangle$ is a ***pre-parallel context***, and $[h_i]_{i\in 1..n}$ are the ***abstractable subterms***.

□

For example, the first equation of *sum* is decomposed into a pre-parallel context and four abstractable subterms, via:

$$sum(\langle\rangle_1 ++ xs, \langle\rangle_2) = \langle\rangle_3 + sum(xs, \langle\rangle_4)\langle[x], c, (c+x), c\rangle$$

Note the abstracted sub-terms, $[x], c, (c+x), c$, can be substituted back into context holes, $\langle\rangle_1, \langle\rangle_2, \langle\rangle_3, \langle\rangle_4$, in order to obtain our pre-parallel equation. For convenience, we will also refer to the *abstractable sub-terms* (and their corresponding holes) as just *pre-parallel holes*, in order to distinguish them from the *pre-parallel context* which they are being separated from.

3.2 Second-Order Generalisation

Once we have two sequential equations with a common pre-parallel context, we can invoke a second-order generalisation rule to obtain a *parallel template equation*. This technique is similar to *generalisation from examples* mechanism commonly advocated for machine learning [DM86], which has been found to be useful also for theorem-proving [Hag95]. In our case, the sequential equations are used as examples in order to obtain more general (parallel) counterparts. A matching process applies appropriate generalisations to those holes that *mismatch*. The generalisation used is second-order, since functional unknowns may be introduced into the template.

Consider the two pre-parallel equations of *sum*. Initially, the LHS is matched. A pair of mismatched expressions is detected at *'[x]'* for the first equation and *'[x]++[y]'* for the second equation. This mismatch can be resolved by replacing the two sub-terms by a new variable, *xr*, generalising the LHS to *sum(xr++xs,c)*. Such a generalisation is said to be first-order because only *object variables* (e.g. *xr*) are introduced. The new recursion parameter, *xr++xs*, now contains two variables, *xr* and *xs*, called *leading* and *trailing* recursion variables, respectively. Leading variables are those introduced by first-order generalisation, while trailing variables are inherited from the original equations.

We now process the RHS, focusing again on the holes of the pre-parallel equations. If the two sub-terms are identical to their original LHS terms, we replace them by their corresponding LHS variable. This occurs at the constant variable *c* of the recursive call, *sum(xs,c)*. However, if the sub-terms mismatch, we use the following second-order generalisation rule.

Defn 8: *Second-Order Generalisation Rule*
If two terms, t_1 and t_2, at a pre-parallel hole in the RHS *mismatch*; we replace it by $H(\vec{v})$ where H is a new function-type variable, and $\vec{v}$ consists of:

- *All* leading recursion variables;
- *All* roving variables;
- *Selected* trailing recursion variables, if they are present in t_1 or t_2;
- *Selected* constant/accumulative variables, if they are present in t_1 or t_2.

□

We include *all* leading recursion and roving parameters, as they may indirectly contribute to the pre-parallel holes, even when their variables are not present in the mismatched sub-terms.

A mismatch occurs for the expressions: *'c+x'* (of the first equation) and *'(c+x)+(c+y)'* (of the second equation). This mismatch is resolved by supplying a generalised expression *H(xr,c)* where *H* is a function-type unknown and *xr* is the leading recursion variable, while *c* is a constant variable present. The trailing recursion variable, *xs*, is not selected because it is neither present in *'c+x'* nor *'(c+x)+(c+y)'*. The final parallel template equation, with unknown *H*, is thus:

$$sum(xr{+}{+}xs,c) = H(xr,c) + sum(xs,c)$$

3.3 Inductive Derivation Procedure

Next, a derivation procedure is given to obtain inductive definitions for the unknown functions. Apart from obtaining such definitions, the corresponding derivation also serves as a correctness (induction) proof for the parallel equation. The inductive derivation procedure consists of the following steps:

Procedure 2: *Inductive Derivation Procedure for Unknown Functions*

Step 1 *Instantiate* the leading recursion variable(s) to the base and recursive cases.

Step 2 *Simplify* the LHS.

Step 3 Apply an *induction* step (for the recursive case).

Step 4 Transform the LHS so that its pre-parallel form (or context) is *similar* to the RHS.

Step 5 *Unify* both LHS and RHS.

Of the five steps, Step 4 appears most intricate. However, it is guided by the need for LHS and RHS to be unifiable. For a concrete example, consider the parallel template equation of *sum* that was obtained in the previous stage. The unknown function is *H*. Step 1 instantiates *H*'s recursion argument, *xr*, to its two cases: *Nil* and *(x:xr)*. These two instantiations are then followed by simplification (Step 2), induction (Step 3), and unification-enabling steps (Step 4 and 5), as shown in Figure 4. Note how the laws $(b \Rightarrow 0+b)$ and $(a+(b+c) \Rightarrow (a+b)+c)$ are used in Step 4, in order to allow the unification of LHS and RHS to succeed via a common pre-parallel context, later in Step 5.

From this derivation, we obtain the following definition for unknown function *H*.

```
H(Nil,c)      = 0;
H(x:xr,c)     = (c+x) + H(xr,c);
```

We check the definition of each newly derived unknown to see if it is equivalent to some previously known function definition. If this is so, we replace each unknown function by a call to the already known function definition. Otherwise, our program will still not be truly parallel, and we would have to apply the parallelization method again to the newly derived function definition.

For example, the definition of *H* is found to be syntactically identical to an earlier definition of *sum*. We can therefore replace *H* with *sum*, and thus obtain the following parallel equation.

$$sum(xr{+}{+}xs,c) = sum(xr,c) + sum(xs,c)$$

There is also a need to obtain a base case equation for the singleton input. This is omitted here as it can be obtained easily via partial evaluation (and simplification) techniques [BEJ88].

```
Step 1 Instantiate xr=Nil:
sum(Nil++xs,c)          = H(Nil,c) + sum(xs,c)
LHS                     = sum(Nil++xs,c)               ; Step 2 Unfold ++
                        = sum(xs,c)                    ; Step 4 Law of +
                        = 0 + sum(xs,c)
RHS                     = H(Nil,c)+ sum(xs,c)
Step 5 Unify both LHS and RHS, yielding:
H(Nil,c)                = 0

Step 1 Instantiate xr=(x:xr):
sum((x:xr)++xs,c)       = H(x:xr,c)+ sum(xs,c)
LHS                     = sum((x:xr)++xs,c)            ; Step 2 Unfold ++
                        = sum(x:(xr++xs),c)            ; Step 2 Unfold sum
                        = (c+x)+sum(xr++xs,c)          ; Step 3 Apply induction
                        = (c+x)+(H(xr,c)+sum(xs,c))    ; Step 4 Assoc. of +
                        = ((c+x)+H(xr,c))+sum(xs,c)
RHS                     = H(x:xr,c)+ sum(xs,c)
Step 5 Unify both LHS and RHS, yielding:
 H(x:xr,c)              = (c+x)+H(xr,c)
```

Figure 4: Derivation of Unknown Functions for Parallel Template

4 Scope of Method

Our proposed parallelization method is for deriving a certain class of divide-and-conquer algorithms with simple splitting operations. It relies on special program properties (such as associativity) to help manipulate the recursive equations to a common pre-parallel form. These laws must be provided for primitive operators. As for user-defined functions, it is often possible to synthesize distributive laws (e.g. over ++) using our method. Both types of laws should be accumulated in a library for future use. The need for such laws and their suitable application is the main reason why we have currently classified our method as being *semi-automatic*. Future improvement to our method would be assessed by how such laws may be systematically generated, and appropriately utilised.

The class of target programs include both *List homomorphism* [Bir87], as well as *near-homomorphism* [Col95]. While near-homomorphism normally requires additional effort for parallelization, *List* homomorphism is a special class of functions that directly has the following divide-and-conquer form:

$$\begin{aligned} F1([]) &= \mathcal{U} \\ F1([x]) &= \mathcal{F}(x) \\ F1(xr{+}{+}xs) &= \mathcal{G}(F1(xr),F1(xs)) \end{aligned}$$

where $\mathcal{G}$ is associative, with $\mathcal{U}$ as its identity. Bird's Homomorphism Theorem showed that such a function is also equivalent to a simple composition of two higher-order functions, as shown below.

$$F1(xs)=reduce(\mathcal{G},\mathcal{U},map(\mathcal{F},xs))$$

With the aid of such schematic equivalence, programmers are expected to construct their programs using higher-order functions (like *map, reduce*) in order to facilitate parallelization. However, the homomorphism sub-class is somewhat limiting since many programs lie outside it. Our new method, being based on elementary transformation rules, does not compel programmers to use a restricted set of higher-order functions. In addition, a single parallelization method (with selective enhancements) is applicable to a reasonably wide range functions, beyond homomorphism, including programs with the following characteristics:

(F2) Without Nil case equation.
(F3) Without an identity in the Nil case.
(F4) Accumulative Parameters.
(F5) Roving Parameters.
(F6) Multiple Recursion Parameters.
(F7) Nested Recursion Parameters.
(F8) Primitive Recurrences.
(F9) Tail Recurrences.
(F10) Nested Recurrences.
(F11) Auxiliary and Mutual Recurrences.
(F12) Conditional and Tupled Recurrences.

In this paper, only programs from more complex recurrences, such as *F4* (accumulative parameters), *F10* (nested recurrence), *F11* (auxiliary non-linear recurrence), and *F12* (conditional recurrence) are highlighted. An expanded version of this paper will describe the other sub-classes of parallelizable functions too.

5 Accumulative Parameters

Consider recursive functions with accumulative parameters. An example is the following function to enumerate the elements of a list.

```
label :: (List a, Int) → List (a, Int);
label(Nil,no)         = Nil;
label(x:xs,no)        = (x,no):label(xs,no+1);
```

Unlike the homomorphism program scheme (*F1*), this function has an extra accumulative parameter. Stages 1 and 2 can obtain the following two similar pre-parallel equations.

$$\begin{aligned} label([x]{+}{+}xs,no) &= \underline{[(x,no)]{+}{+}label(xs,no{+}\underline{1})}; \\ label(\underline{([x]{+}{+}[y])}{+}{+}xs,no) &= \underline{(\underline{[(x,no)]}{+}{+}[(x,no{+}1)])}{+}{+} \\ &\qquad label(xs,no{+}\underline{(1{+}1)}); \end{aligned}$$

Notice that the outer operators leading to variables *xs* and *no* are either associative (i.e. ++, +) or are potentially distributive (i.e. *label*). After second-order generalisation by Stage 3, we obtain:

$$label(mr{+}{+}xs,no) \;= H(mr,no){+}{+}label(xs,no{+}G(mr));$$

On further derivation by Stage 4, we confirm that *H≡label*, while *G* is a new auxiliary function:

```
G(Nil)            = 0;
G(x:xs)           = 1+G(xs);
```

The function *G* can be similarly parallelized by our inductive method to obtain:

```
G(xr++xs)         = G(xr)+G(xs);
```

The two parallel equations are thus:

```
label(xr++xs,no)  = label(xr,no)++label(xs,no+G(xr));
G(xr++xs)         = G(xr)+G(xs);
```

Though the above program exhibits good parallelism, it is currently not efficient. This is because function *label* has two recursive calls, *label(xr,no)* and *G(xr)*, which traverse the same sublist *xr* twice. Such calls, with a shared recursion argument, cause multiple traversals and/or redundant calls. A classic approach for optimizing such programs is to use the tupling method of [Chi93]. For the *label* example, the tupling method can *automatically* introduce a new tuple function:

```
tup(zs,no)        = (label(zs,no),G(zs));
```

After transformation, the final *efficient* parallel definition for *label* is:

```
label(xs,no)      = let (u,_)=tup(xs,no) in u ;
tup(Nil,no)       = (Nil,0);
tup([x],no)       = ([(x,no)],1);
tup(xr++xs,no)    = let {(a,b) = tup(xr,no) ;
                          z     = no+b ;
                         (u,v) =tup(xs,z) }
                    in (a++u,b+v) ;
```

The parallel characteristics of the above *tup* function may not be apparent. In particular, the *z* parameter of the second recursive call of *tup(xs,z)* actually depends on an output from the first recursive call *tup(xr,no)*. Nevertheless, function *tup* has a similar structure as the highly versatile *scan* function, popularised by Blelloch [Ble89]. Like *scan*, it can be implemented efficiently in a multi-processor system which supports bi-directional tree-like communications - using parallel computation time proportional to $O(\log n)$ where n is the length of the list. Two phases are employed for its parallel computation. An upsweep phase in the computation can be used to compute the second values of the tuple (i.e. *G(zs)*), before a downsweep phase is used to compute the first values of the tuple (i.e. *label(zs,no)*).

6 Nested Recurrences

Consider the following linear but nested recurrence :

```
comp(Nil)         = 0;
comp((x,y):xs)    = x+(y*comp(xs));
```

We refer to this as a *nested recurrence* as its recursive call is nested (more deeply) at depth 2 with + and * as its outer auxiliary operators. A pre-parallel equation obtainable in Stage 1 is:

```
comp([(x,y)]++xs)  = x+(y*comp(xs));
```

Using the associative properties of + and *, and the distributive law of * over +, a second pre-parallel equation can be obtained in Stage 2, *guided by heuristics H1* and *H2*, as follows.

```
comp([(x,y)]++xs))
      = x+(y*comp(xs))                ; unfold  comp
comp([(x,y)]++([(a,b)]++xs))
      = x+(y*(a+(b*comp(xs))))        ; assoc. law of ++
comp(([(x,y)]++[(a,b)])++xs)
      = x+(y*(a+(b*comp(xs))))        ; distr. law of * over +
      = x+(y*a+y*(b*comp(xs)))        ; assoc. law of +
      = (x+y*a)+(y*(b*comp(xs)))      ; assoc. law of *
      = (x+y*a)+((y*b)*comp(xs))
```

The two equations obtained have the same pre-parallel context, namely: $comp(\langle\rangle_1 ++xs)=\langle\rangle_2+(\langle\rangle_3 *comp(xs))$. Stage 3 (second-order generalisation) can now obtain a template equation with unknowns *H* and *G*:

```
comp(ms++xs) = H(ms) + G(ms)*comp(xs)
```

An inductive derivation by Stage 4 can synthesize definitions for the two unknown functions. The definition for *H* is syntactically identical to *comp*, but *G* has a new auxiliary definition:

```
G(Nil)            = 1;
G((m,n):ms)       = n*G(ms);
```

As *G* is a *List*-homomorphism, it is easily parallelized by our method. Hence, the two parallel equations are:

```
comp(ms++xs) = comp(ms) + G(ms)*comp(xs);
G(ms++xs))   = G(ms)*G(xs);
```

Though the above program exhibits good parallelism, it is not efficient due to the presence of redundant *G* calls. As before, we can rectify this situation by applying the tupling method [Chi93]. Specifically, this method will introduce a new tuple function:

```
comptup(xs) = (comp(xs),G(xs));
```

before it is transformed to:

```
comptup(Nil)      = (0,1);
comptup([(m,n)])  = (m,n);
comptup(ms++xs) = let { (a,b)=comptup(ms) ;
                        (c,d)=comptup(xs) }
                  in (a+c*b,b*d);
```

The final tupled function *comptup* is now a *List*-homomorphism, even though the initial sequential version of *comp* isn't. Cole refers to functions, like *comp*, as near-homomorphism [Col95], and suggested to search manually for *more general* tupled functions with the requisite property. Our inductive parallelization method can synthesize the needed auxiliary functions automatically. In conjunction with the tupling method, it can systematically yield efficient and parallel programs as the desired target.

7 Non-Linear Recurrences

We now look at how our inductive method can directly handle non-linear recurrences. We use the example of a simple simulation program with a single queue/server. Assume the event list is represented by a list of pairs of positive numbers (with suitable random distribution):

$$[(s_n,a_n),(s_{n-1},a_{n-1}),\ldots,(s_1,a_1)]$$

where $a_1,..,a_n$ are the inter-arrival time gaps between the n events, and $s_1,..,s_n$ are the corresponding service times. Note that the events are ordered right-to-left, with the first event represented by the rightmost element of the list. With this representation, we can define functions to compute the final arrival and departure times for a list of events, as follows:

```
arrive(Nil)        = 0;
arrive((s,a):xs)   = a + arrive(xs);
depart(Nil)        = 0;
depart((s,a):xs)   = s + max(arrive((s,a):xs), depart(xs));
```

The above specification is easy enough to write (and read) but it is presently sequentially-oriented. The parallelization of function *arrive* is trivial since it is a *List* homomorphism. We obtain:

```
arrive(xr++xs) = arrive(xr) + arrive(xs);
```

The parallelization of *depart* is more tricky and not immediately obvious. In particular, note that *depart* is actually a non-linear recurrence with two recursive calls, *arrive((s,a):xs)* and *depart(xs)*, with *arrive* as an auxiliary function of *depart*. Our method attempts to obtain a pre-parallel form which keeps the common recursion variable *xs*, of the two recursive calls, at the shallowest depth. Guided by the earlier two heuristics, we obtain our first pre-parallel equation:

$$depart(\underline{[(s,a)]}{+}{+}xs) = max(\underline{(s+a)}+arrive(xs),\ \underline{s}+depart(xs))$$

with the holes of the pre-parallel form shown underlined. In Stage 2, we obtain a second recursive equation with a similar pre-parallel form, namely:

$$depart\ (\underline{[(s,a),(s2,a2)]}{+}{+}xs) = max(\underline{max(s+a+a2,s+s2+a2)}+arrive(xs),\underline{(s+s2)}+depart(xs))$$

These two pre-parallel equations are used in Stage 3 to obtain a parallel template equation, shown below, with two unknown functions, *G* and *H*.

```
depart(xr++xs) = max(G(xr)+arrive(xs),H(xr)+depart(xs))
```

During the inductive derivation in Stage 4, we confirm that *G≡depart*, while *H* is a new function with the following definition.

```
H(Nil)        = 0;
H((s,a):xs)   = s + H(xs);
```

Like *arrive*, *H* also belongs to the homomorphism class. Thus, we now have three parallel equations:

```
H(xr++xs)      = H(xr)+H(xs)
arrive(xr++xs) = arrive(xr) + arrive(xs)
depart(xr++xs) = max(depart(xr)+arrive(xs)
                     ,H(xr)+depart(xs))
```

It is really *not* obvious from the sequential version of *depart*, that such a parallel equation follows. A fairly intricate technique appears to be used in this particular divide-and-conquer algorithm. Specifically, *H(xr)+depart(xs)* and *depart(xr)+arrive(xs)* denotes two possible scenarios which might occur for events in *xr*, namely (i) server is continuously busy, or (ii) server has at least one free gap. In the latter case, the finishing time of *depart(xr++xs)* does not depend on *depart(xs)* at all. This somewhat deep insight has been mechanically synthesized!

The *depart* equation is currently inefficient as there are multiple recursive calls (in the RHS) which operates on the same data structures, e.g. *(H(xr), depart(xr))* and *(arrive(xs), depart(xs))*. This again calls for the tupling method which can automatically introduce the following tuple function definition:

```
tup(xs)          = (arrive(xs),H(xs),depart(xs))
```

After transformation, we obtain the following efficient and parallel tupled program:

```
tup(Nil)         = (0,0,0)
tup([(s,a)])     = (a,s,s+a)
tup(xr++xs)      = let { (b,c,d) = tup(xr);
                         (u,v,w) = tup(xs) }
                   in (b+u, c+v, max(d+u,c+w))
```

Though our example was taken from [GLM90], our synthesis was never guided by their final parallel program which was casted as a 2x2 matrix multiplication solution (with + and *max* as operators). A nice consequence is that our parallel program (without being constrained by the matrix notation) uses a smaller 1x3 tuple that requires fewer operations.

8 Conditional Recurrences

Recurrences with outer conditional constructs are also parallelizable by our method, with suitable extensions as outlined in [CDG96]. One difficult scenario occurs when these recurrences contain more than one branch with recursive calls. A somewhat tricky example is shown below with two recursive branches.

```
mul_add(Nil)     = 0 ;
mul_add(x:xs)    = if x<10 then x*mul_add(xs)
                     else x+mul_add(xs) ;
```

To handle such recurrence with multiple recursive branches, we propose a simple extension (for Stage 1) to combine the multiple branches into a single recursive branch, namely:

Procedure 3: *Combining Recursive Branches*

Step 1 Find a common pre-parallel form for the different recursive branches.

Step 2 Combine and unify the multiple recursive branches.

Step 1 is to find a common pre-parallel form for all the recursive branches. This is achieved by obtaining a desired pre-parallel form for each of the branches, before attempting to unify them via suitable transformations. In the case of *mul_add*, a common pre-parallel form can be obtained if the first branch is converted to $(\underline{0}+\underline{x}*mul\_add(xs))$, while the second branch is converted to $(\underline{x}+\underline{1}*mul\_add(xs))$. This step can be achieved via a search-based (exploratory) transformation using ***means-end analysis*** with the objective of unifying two expressions.

With a common pre-parallel context, Step 2 would now unify all recursive branches into a single branch, by pushing the outer conditional into each hole of the common pre-parallel context. With $\widehat{e^P}\langle\rangle$ as a common pre-parallel context for multiple branches, the following rule performs the desired transformation, where *cond* denotes a guarded multi-branch conditional.

$$cond\ \{b_i \rightarrow \widehat{e^P}\langle t_{ij}\rangle_{j\in M}\}_{i\in N} \equiv \widehat{e^P}\langle cond\{b_i \rightarrow t_{ij}\}_{i\in N}\rangle_{j\in M}$$

When applied to *mul_add*, we have:

```
mul_add(x:xs) = if x<10 then 0+x*mul_add(xs)
                 else x+1*mul_add(xs);
              = (if x<10 then 0 else x)+
                  (if x<10 then x else 1)*mul_add(xs);
```

With a single recursive branch, it is straightforward to apply our inductive parallelization method to obtain:

```
mul_add([x])      = if x<10 then 0 else x ;
mul_add(xr++xs) = mul_add(xr)+G(xr)*mul_add(xs) ;
G([x])            = if x<10 then x else 1 ;
G(xr++xs)         = G(xr)*G(xs) ;
```

As before, the redundant calls of *G* can also be eliminated by the tupling method of [Chi93], with an alternative formulation of tupling in calculational style given by [HITT97].

9 Related Work

As described earlier, a popular approach for synthesizing parallel functional programs is to use the Bird-Meertens formalism [Ski90]. The emphasis there is to construct programs using a small set of common higher-order functions (such as *map*, *reduce*) from which it is often possible to directly derive the divide-and-conquer homomorphism *without* using induction. However, the homomorphism sub-class (i.e. *F1*) is rather limiting since many programs lie outside it. As illustrated in this paper, a more fundamental approach is possible, based on the elementary rules with induction capability provided by second-order generalisation and an inductive derivation procedure. There is absolutely no need to restrict user programs to a closed set of higher-order functions. Our method is capable of generating suitable parallel equations directly. Where necessary, we use other techniques, such as tupling, to obtain more efficient parallel programs.

In traditional imperative languages (e.g. Fortran), there are also many on-going efforts at developing sophisticated techniques for parallelizing iterative loops. Lately, we became aware of a more systematic method for parallelizing complex scans and reductions [FG94, GF95]. This method is based on a parallel reduction of *function composition* where function-type values are propagated - relying on such composition being inherently associative. However, the complexity of the functions propagated could get progressively worse unless they match a certain *template* form. The steps needed for finding such template form is similar to our method's Stage 1 and 2 for finding a common pre-parallel form. Our technique was discovered independently with the initial methodology presented in [Chi90]. While practical, Fischer and Ghuloum's method is somewhat less general, as it is based around iterative loops rather than the more general recursive functions, and it does not give the actual parallel equations (which are also useful as laws for transformation methods such as fusion [Chi92]). Also, without the heuristics for obtaining desired pre-parallel form, their parallelization may result in extra (unnecessary) auxiliary values to compute.

10 Conclusion

The list of program sub-classes illustrated in this paper is fairly extensive but not exhaustive. In particular, we have not considered possible combinations of the different sub-classes. As the parallelization method is based on elementary rules, it can handle hybrid combinations easily.

Though general, our method has limitations. It is, so far, for only recursive functions whose outer auxiliary functions (in the RHS) possess appropriate semantic properties (e.g. associative or distributive laws) in order to allow similar pre-parallel equations to be obtained. Occasionally, some auxiliary operators might not have the required properties, while generalised versions of them do. In such a situation, our method fails. For example, we are unable to synthesize a parallel merge-sort program if the *insert* function of type *(Int,[Int])* $\rightarrow$ *[Int]* is used, instead of the more general *merge* function of type *([Int],[Int])* $\rightarrow$ *[Int]*. The problem is that *insert* is non-associative which prevented us from obtaining two equations with the same pre-parallel form. It might be fruitful to investigate new techniques to synthesize, where possible, generalised functions with the associative property. Also, the synthesized divide-and-conquer structure is presently restricted to a simple divide operation (e.g. *++*). Programs, like *quicksort*, which decompose their input lists more cleverly, are not handled by our method. A promising extension of our work was recently reported by Voicu [Voi96]. By manually fixing a known divide operator and directly providing a target parallel form (with an unknown conquer operator), Voicu showed how bitonic sort and quicksort can be synthesized from the selection sort algorithm. Further foray in this direction, with reduced manual intervention, may yield another promising outcome for parallel algorithm synthesis.

As a task, parallel programming is probably an order of magnitude more difficult than sequential programming. Apart from the need to design algorithms with adequate parallelism, programmers often have to grapple with a host of other issues, such as different target architectures, dynamic versus static data/process allocations, possible non-determinism, effective grain sizes, communication overheads, and complications of parallel debugging. Many parallel hardware platforms (including distributed systems) are becoming more affordable lately. However, software still remains a major obstacle to more wide-spread parallel computing. In the functional programming arena, work in the area of parallelizing compilers [FCO90], algorithmic skeletons for better mapping to architecture[Col88, MH88], and program visualization [HLP95], are contributing towards the goals of more declarative parallel programming. The work reported here is an attempt to contribute in the same direction. Apart from efforts to improve the proposed parallelization methodology, we are currently working on an implementation which will allow the level of automation for the methodology to be systematically determined.

Acknowledgement: This work is supported by EEC KIT143 ConFuPro (Concurrent Functional Programming : Theory and Practice) grant, and NUS research grant RP920614. The authors would like to thank Richard Bird, Tyng-Ruey Chuang, John Darlington, John Hughes, Luke Ong and Rinus Plasmeijer for facilitating short & fruitful visits to their institutions.

References

[AS96] K. Achatz and W. Schulte. Massive parallelization of divide-and-conquer algorithms over powerlists. *Science of Computer Programming*, 26:59–78, 1996.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, January 1977.

[BEJ88] D. Bjørner, A.P. Ershov, and N.D. Jones. *Workshop on Partial Evaluation and Mixed Computations*. Gl Avarnes, Denmark, North-Holland, 1988.

[Bir87] 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.

[Ble89] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[CDG96] W.N. Chin, J Darlington, and Y. Guo. Parallelizing conditional recurrences. In *2nd EuroPar Conference*, Lyon, France, (LNCS 1123) Berlin Heidelberg New York: Springer, August 1996.

[Chi90] Wei-Ngan Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, March 1990.

[Chi92] Wei-Ngan Chin. Safe fusion of functional expressions. In *7th ACM LISP and Functional Programming Conference*, pages 11–20, San Francisco, California, June 1992. ACM Press.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *3rd ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, ACM Press, June 1993. ACM Press.

[Col88] Murray I. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. PhD thesis, University of Edinburgh, 1988.

[Col95] Murray I. Cole. Parallel programming with list homomorphism. *Parallel Processing Letters*, 5(2):191–203, 1995.

[DM86] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1:145–176, 1986.

[FCO90] J.T. Feo, D.C. Cann, and R.R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–366, 1990.

[FG94] 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.

[GF95] 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.

[GLM90] A.G. Greenberg, B.D. Lubachevsky, and I. Mitrani. Unboundedly parallel simulation via recurrence relations. In *ACM SIGMETRICS*, pages 1–12, September 1990.

[Hag95] Masami Hagiya. A typed λ-calculus for proving-by-example and bottom-up generalization procedure. *Theoretical Computer Science*, 137:3–23, 1995.

[HITT97] Z.J. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple traversals. In *2nd ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, Netherlands, June 1997. ACM Press (to appear).

[HLP95] K Hammond, H.W. Loidl, and A. Partridge. Visualising granularity in parallel programs: A graphical winnowing system for Haskell. In *High Performance Functional Programming Conference*, USA, April 1995.

[MH88] Z.G Mou and P. Hudak. An algebraic model for divide and conquer and its parallelism. *The Journal of Supercomputing*, 2, 1988.

[PP93] J.F. Prins and D.W. Palmer. Transforming high-level data parallel programs into vector operations. In *4th Principles and Practice of Parallel Programming*, pages 119–128, San Diego, California (ACM Press), May 1993.

[RW93] 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.

[Ski90] D. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.

[Voi96] Razvan Voicu. Synthesizing parallel divide-and-conquer algorithms using the list interleave operator. In *2nd ASIAN Computer Science Conference*, Singapore, (LNCS 1179, pg 359–360) Berlin Heidelberg New York: Springer, December 1996.