AutoPar: Automatic Parallelization of Functional Programs

Michael Dever & G. W. Hamilton
{mdever, hamilton}@computing.dcu.ie

Dublin City University

Abstract. In this paper we present a novel, fully automatic transformation technique which parallelizes functional programs defined using any data-type. In order to parallelize these programs our technique first derives conversion functions which allow the data used by a given program to be well-partitioned. Following this the given program is redefined to make use of this well-partitioned data. Finally, the resulting program is explicitly parallelized. In addition to the automatic parallelization technique, we also present the results of applying the technique to a sample program.

1 Introduction

As the pervasiveness of parallel architectures in computing increases, so does the need for efficiently implemented parallel software. However, the development of parallel software is inherently more difficult than that of sequential software as developers are typically comfortable developing sequentially and can have problems thinking in a parallel setting [29]. Yet, as the limitations of single-core processor speeds are reached, the developer has no choice but to reach for parallel implementations to obtain the required performance increases.

Functional languages are well suited to parallelization due to their lack of side-effects, a result of which is that their functions are stateless. Therefore, one process executing a pure function on a set of data can have no impact on another process executing a function on another set of data as long as there are no data dependencies between them. This gives functional programs a semantically transparent *implicit task parallelism.*

Due to the nature of functional languages, many functions make use of intermediate data-structures to generate results. The use of intermediate data-structures can often result in inefficiencies, both in terms of execution time and memory performance [38]. When evaluated in a parallel environment, the use of intermediate data-structures can result in unnecessary communication between parallel processes. Elimination of these intermediate data-structures is the motivation for many functional language program transformation techniques, such as that of *distillation* [14, 16] which is capable of obtaining a super-linear increase in efficiency.

The automatic parallelization technique presented in this paper makes use of our previously published automatic partitioning technique [9], which facilitates the partitioning of data of any type into a corresponding *join*-list. This is used to generate functions which allow data of any type to be converted into a well-partitioned *join*-list and also allow the data in a well-partitioned *join*-list to be converted back to its original form.

Following the definition of these functions, we distill the given program which results in an equivalent program defined in terms of a well-partitioned *join*-list, in which the use of intermediate data-structures has been eliminated. Upon distilling a program defined on a well-partitioned *join*-list, we then extract the parallelizable expressions in the program. Finally, we apply another transformation to the distilled program which parallelizes expressions operating on well-partitioned *join*-lists.

The remainder of this paper is structured as follows: Section 2 describes the language used throughout this paper. Section 3 describes how we implement explicit parallelization using Glasgow parallel Haskell. Section 4 presents an overview of the distillation program transformation technique and describes how we use distillation to convert programs into equivalent programs defined on well-partitioned data. Section 5 describes our automatic parallelization technique. Section 6 presents the application of the automatic parallelization technique to a sample program. Section 7 presents a selection of related work and compares our techniques with these works and Section 8 presents our conclusions.

2 Language

The simple higher-order language to be used throughout this paper is shown in Fig. 1. Within this language, a data-type T can be defined with the constructors $c_1, \ldots, c_m$ each of which may include other types as parameters. Polymorphism is supported in this language via the use of type variables, α. Constructors are of a fixed arity, and within $c\ e_1 \ldots e_k$, k must be equal to constructor c's arity. We use $(e :: t)$ to denote an expression e of type t. Case expressions must only have non-nested patterns. Techniques exist to transform nested patterns into equivalent non-nested versions [1,37].

Within this language, we use **let** $x_1 = e_1\ \ldots\ x_n = e_n$ **in** e_o to represent a series of nested **let** statements as shown below:

$$\begin{array}{ll}\textbf{let}\ x_1 = e_1\ \ldots\ x_n = e_n\ \textbf{in}\ e_o \equiv & \textbf{let}\ x_1 = e_1 \\ & \quad \vdots \\ & \quad \textbf{let}\ x_n = e_n \\ & \quad \textbf{in}\ \ e_0\end{array}$$

The type definitions for *cons*-lists and *join*-lists are as shown in Fig. 2. We use the usual Haskell notation when dealing with *cons*-lists: [] represents an empty *cons*-list, (Nil), $[x]$ represents a *cons*-list containing one element, $(Cons\ x\ Nil)$, and $(x : xs)$ represents the *cons*-list containing the *head* x and the *tail* xs, $(Cons\ x\ xs)$.

$t ::=$	α	Type Variable
$\mid$	$T\ t_1 \ldots t_g$	Type Application
$\mathbf{data}\ T\ \alpha_1 \ldots \alpha_g ::=$	$c_1\ t_{1_1}\ \ldots\ t_{1_{n_1}}$	Data-Type
$\vdots$		
$\mid$	$c_m\ t_{m_1}\ \ldots\ t_{m_{n_m}}$	
$e ::=$	x	Variable
$\mid$	$c\ e_1 \ldots e_k$	Constructor
$\mid$	f	Function
$\mid$	$\lambda x.e$	Lambda Abstraction
$\mid$	$e_0\ e_1$	Application
$\mid$	$\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k$	Case Expression
$\mid$	$\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0$	Let Expression
$\mid$	$e_0\ \mathbf{where}\ f_1 = e_1\ \ldots\ f_n = e_n$	Where Expression
$p ::=$	$c\ x_1\ \ldots\ x_k$	Pattern

Fig. 1: Language Definition

$$
\begin{aligned}
&\textit{data List a} &&::= \textit{Nil}\\
& &&\mid \textit{Cons a (List a)}\\[1em]
&\textit{data JList a} &&::= \textit{Singleton a}\\
& &&\mid \textit{Join (JList a) (JList a)}
\end{aligned}
$$

Fig. 2: *cons*-List and *join*-List Type Definitions

3 Glasgow Parallel Haskell

In order to parallelize the various programs described in this paper we make use of Glasgow parallel Haskell (GpH) [36] which is an extension to Haskell. GpH supports parallelism by using strategies for controlling the parallelism involved. Parallelism is introduced via *sparking* (applying the **par** strategy) and evaluation order is determined by applying the **pseq** strategy. As an example, the expression x *'par'* y **may** spark the evaluation of x in parallel with that of y, and is semantically equivalent to y. As a result of this, when using x *'par'* y, the developer indicates that they believe evaluating x in parallel may be useful, but leave it up to the runtime to determine whether or not the evaluation of x is run in parallel with that of y [25]. *pseq* is used to control evaluation order as x *'pseq'* y will strictly evaluate x before y. Usually, this is used because y cannot be evaluated until x has been.

As an example, the expression x *'par'* (y *'pseq'* $x + y$) sparks the evaluation of x in parallel with the strict evaluation of y. After y has been evaluated, $x + y$ is then evaluated. If the parallel evaluation of x has not been completed at this point, then it will be evaluated sequentially as part of $x + y$. As a result of this x *'par'* (y *'pseq'* $x+y$) is semantically equivalent to $x+y$, but we may see some performance gain from sparking the evaluation of x in parallel. Below is a simple example of the use of GpH, which calculates fibonacci numbers in parallel:

```
fib = λx.case x of
          0 → 1
          1 → 1
          n → let x  = fib (n − 1)
              in let y  = fib (n − 2)
                 in x ‘par′ (y ‘pseq′ x + y)
```

Given a number x, the function fib sparks the evaluation of fib $(n - 1)$ to *weak-head normal form* in parallel with the full evaluation of fib $(n - 2)$. When fib $(n - 2)$ has been fully evaluated, it is then added to the result of the evaluation of fib $(n-1)$. fib $(n-1)$ can be fully evaluated in parallel by having the *rdeepseq* strategy applied to it.

We have selected Glasgow parallel Haskell for our implementation language, due to its conceptual simplicity, its semantic transparency and its separation of algorithm and strategy. Another reason for the selection of Glasgow parallel Haskell is its management of threads: it handles the creation/deletion of threads, and determines whether or not a thread should be sparked depending on the number of threads currently executing.

4 Transforming Data to Well-Partitioned *Join*-Lists

There are many existing automated parallelization techniques [2–7, 10, 19–22, 26, 30, 31, 35], which, while powerful, require that their input programs are defined using a *cons*-list, for which there is a straightforward conversion to a well-partitioned *join*-list. This is an unreasonable burden to place upon a developer as it may not be intuitive or practical to define their program in terms of a *cons*-list.

To solve this problem we have previously defined a novel transformation technique that allows for the automatic partitioning of an instance of any data-type [9]. A high-level overview of the automatic partitioning technique is presented in Figure 3. We combine this technique with distillation in order to automatically convert a given program into one defined on well-partitioned data. We do not give a full description of the automatic partitioning technique here, it is sufficient to know that it consists of the following four steps:

1. Given a program defined on an instantiated data-type, τ, we use the definition of τ to define a corresponding data-type, τ', instances of which will contain the non-inductive components from data of type τ.
2. Derive a *partitioning function*, $partition_\tau$, which will allow data of type τ to be converted into a well-partitioned *join*-list containing data of type τ'.

3. Derive a *rebuilding function*, $rebuild_\tau$, which will convert a *join*-list containing data of type τ' into data of type τ.
4. Distill a program equivalent to the given program which is defined on a well-partitioned *join*-list.

Using these four steps, we can automatically convert a given program into an equivalent program defined on well-partitioned data. Section 4.1 presents an overview of the distillation program transformation technique. Section 4.2 describes how we combine distillation and the automatic partitioning technique in order to convert a program into an equivalent one defined on well-partitioned data.

$$\begin{tikzcd}
\boxed{x :: \tau} \arrow[rr, bend left, "partition_\tau"] & & \boxed{x' :: JList\ \tau'} \arrow[ll, bend left, "rebuild_\tau"]
\end{tikzcd}$$

Fig. 3: Data Partitioning Functions

4.1 Distillation

Distillation [14, 16–18] is a powerful fold/unfold based program transformation technique which eliminates intermediate data-structures from higher-order functional programs. It is capable of obtaining a super-linear increase in efficiency and is significantly more powerful than the *positive-supercompilation* program transformation technique [15, 32, 33] which is only capable of obtaining a linear increase in efficiency [34].

Distillation essentially performs normal-order reduction according to the reduction rules defined in [14]. *Folding* is performed on encountering a renaming of a previously encountered expression, and *generalization* is performed to ensure the termination of the transformation process. The expressions compared prior to folding or generalization within the distillation transformation are the results of symbolic evaluation of the expressions, whereas in positive-supercompilation, the syntax of the expressions are compared. Generalization is performed upon encountering an expression which contains an embedding of a previously encountered expression. This is performed according to the *homeomorphic embedding relation*, which is used to detect divergence within term rewriting systems [8].

We do not give a full description of the distillation algorithm here; details can be found in [14]. The distillation algorithm is not required to understand the remainder of this paper, it is sufficient to know that distillation can be used to eliminate the use of intermediate data structures in expressions.

4.2 Distilling Programs on Well-Partitioned Data

Given a sequential program, f, defined using an instantiated data-type, τ, we first define a conversion function which will convert data of type τ into a well partitioned *join*-list, $partition_\tau$, and one which will convert a *join*-list into data of type τ, $rebuild_\tau$, as depicted in Figure 3.

By applying distillation, denoted $\mathcal{D}$, to the composition of f and $rebuild_\tau$, $\mathcal{D}[\![f \circ rebuild_\tau]\!]$, we can automatically generate a function, f_{wp}, which is equivalent to f but is defined on a well-partitioned *join*-list containing data of type τ'. A high level overview of this process is presented in Figure 4.

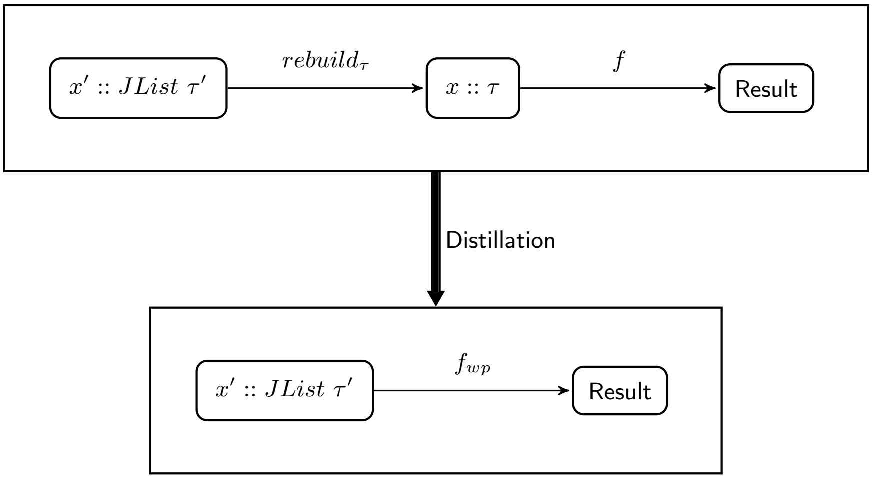

Fig. 4: Distillation of Programs on Well-Partitioned Data

Obviously, f_{wp} is defined on a well-partitioned *join*-list, whereas f is defined on data of type τ. We can generate the correct input for f_{wp} by applying $partition_\tau$ to the input of f and using the result of this as the input to f_{wp}.

5 Automatically Parallelizing Functional Programs

Given a program, once we have distilled an equivalent program defined on a well-partitioned *join*-list, this can be transformed into an equivalent explicitly parallelized program defined on a well-partitioned *join*-list. Our *novel* parallelization technique consists of two steps:

1. Ensure that expressions which are parallelizable are independent.

2. Explicitly parallelize the resulting parallelizable expressions.

Using these two steps, we can automatically convert a given program into an equivalent explicitly parallelized program that operates on well-partitioned data. Section 5.1 describes the process by which we ensure that expressions which are parallelizable are independent and Section 5.2 describes the process which explictly parallelizes a program defined on well-partitioned data.

5.1 Extracting Independent Parallelizable Expressions

Prior to explicitly parallelizing the output of distillation we must first make sure that any expressions that will be parallelized are independent of each other. Given an expression, e, we define its parallelizable expressions as the set of unique maximal sub-expressions of e which operate on a single *join*-list. That is, each parallelizable expression of e must be the largest sub-expression of e which operates on a *join*-list and is not a variable. Once we have identified all the parallelizable expressions within e, these are extracted into a **let** statement which is equivalent to e.

As an example, consider the function f, as defined below:

$$\begin{aligned} f = \lambda x.\ \lambda n.&\mathbf{case}\ x\ \mathbf{of} \\ &\quad Join\ l\ r \rightarrow f\ r\ (f\ l\ n) \end{aligned}$$

Within f, both the *join*-lists l and r are evaluated, therefore we identify the parallelizable expressions in which they are evaluated, $f\ l\ n$ and $f\ r$ respectively. These are then extracted into a **let** statement equivalent to $f\ r\ (f\ l\ n)$ as follows:

$$\begin{aligned} f = \lambda x.\ \lambda n.&\mathbf{case}\ x\ \mathbf{of} \\ &\quad Join\ l\ r \rightarrow \mathbf{let}\ x_1 = f\ l\ n \\ &\qquad\qquad\qquad\quad\ x_2 = f\ r \\ &\qquad\qquad\qquad\ \mathbf{in}\ \ x_2\ x_1 \end{aligned}$$

As the output of distillation contains no **let** statements, this process allows us to explicitly parallelize expressions that operate on well-partitioned *join*-lists, as any extracted expression within a **let** statement must be a parallelizable expression.

5.2 Explicit Parallelization of Functional Programs

Given a distilled program which operates on a well-partitioned *join*-list, f_{wp}, which has had its parallelizable expressions made independent, the final step of the automatic parallelization technique is to apply another transformation, $\mathcal{T}_p$, to f_{wp} in order to explicitly parallelize expressions which operate on well-partitioned *join*-lists. The result of this transformation is a function, f_{par}, which is equivalent to f but is defined on a well-partitioned *join*-list and has been explicitly parallelized. A high-level overview of this process is shown in Figure 5. The transformation rules for $\mathcal{T}_p$ are defined as shown in Fig. 6.

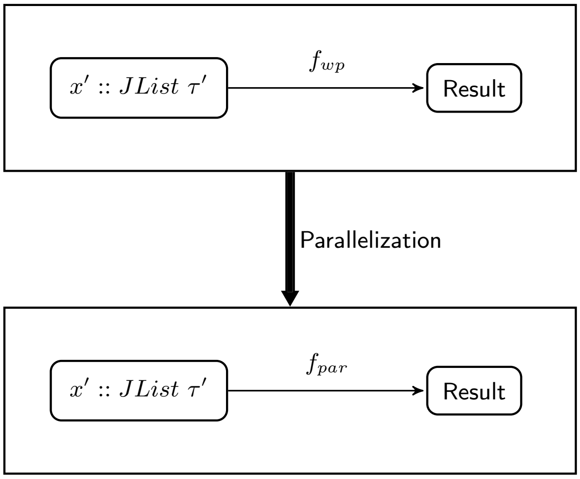

Fig. 5: Parallelization of Program defined on Well-Partitioned Data

The majority of the transformation rules should be self explanatory: variables, abstraction variables, constructor names are left unmodified. The bodies of constructors, applications, abstractions and function definitions are transformed according to the parallelization rules, as are the selectors and branch expressions of **case** statements.

The most interesting of the presented transformation rules is the one that deals with **let** statements. As each extracted expression within a **let** statement is a parallelizable expression its evaluation should be sparked in parallel. However, as these may contain further parallelizable expressions, we first apply the parallelization algorithm to the extracted expressions, as well as to the body of the **let** statement. Finally, we spark the evaluation of all but one of the parallelizable expressions in parallel with the full evaluation of the last parallelizable expression, x_n. This means that the evaluation of the parallelized version of e_0 will not begin until x_n has been fully evaluated, at which point, hopefully, the remaining parallelized expressions will have been evaluated in parallel.

As an example, consider again the definition of f shown previously, which has had its parallelized expressions extracted. Application of $\mathcal{T}_p$ to f results in the definition of f_{par} shown below:

$$\begin{array}{l}
f_{par} = \lambda x.\ \lambda n.\mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad Join\ l\ r \rightarrow \mathbf{let}\ x_1 = f_{par}\ l\ n \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad x_2 = f_{par}\ r \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \ x_1\ 'par'\ x_2\ 'pseq'\ x_2\ x_1
\end{array}$$

$$
\begin{array}{ll}
\mathcal{T}_p[\![x]\!] & = x \\
\mathcal{T}_p[\![c\ e_1\ \ldots\ e_n]\!] & = c\ \mathcal{T}_p[\![e_1]\!]\ \ldots\ \mathcal{T}_p[\![e_n]\!] \\
\mathcal{T}_p[\![f]\!] & = f \\
\mathcal{T}_p[\![\lambda x.e]\!] & = \lambda x.\mathcal{T}_p[\![e]\!] \\
\mathcal{T}_p[\![e_0\ e_1]\!] & = \mathcal{T}_p[\![e_0]\!]\ \mathcal{T}_p[\![e_1]\!] \\
\mathcal{T}_p[\![\mathbf{case}\ x\ \mathbf{of}\ p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k]\!] & = \mathbf{case}\ x\ \mathbf{of}\ p_1 \rightarrow \mathcal{T}_p[\![e_1]\!] \mid \ldots \mid p_k \rightarrow \mathcal{T}_p[\![e_k]\!] \\
\mathcal{T}_p[\![e_0\ \mathbf{where}\ f_1 = e_1\ \ldots\ f_n = e_n]\!] & = \mathcal{T}_p[\![e_0]\!]\ \mathbf{where}\ f_1 = \mathcal{T}_p[\![e_1]\!]\ \ldots f_n = \mathcal{T}_p[\![e_n]\!] \\
\mathcal{T}_p[\![\mathbf{let}\ x_1 = e_1\ \ldots\ x_n = e_n \mathbf{in}\ e_0]\!] &
\end{array}
$$

$$
= \begin{cases}
\mathbf{let}\ x_1 = \mathcal{T}_p[\![e_1]\!] \\
\qquad \vdots \\
\quad\ \ x_n = \mathcal{T}_p[\![e_n]\!] \\
\mathbf{in}\ \ x_1\ \textit{'par'}\ \ldots\ x_{n-1}\ \textit{'par'}\ x_n\ \textit{'pseq'}\ \mathcal{T}_p[\![e_0]\!]
\end{cases}
$$

Fig. 6: Transformation Rules for Parallelization

6 Automatic Parallelization of a Sample Program

This section presents an example of the application of the automatic parallelization technique to the program *sumList*, which calculates the sum of a *cons*-list of numbers, as shown below:

$$
\begin{array}{l}
sumList = \lambda xs.\mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad\qquad\quad Nil \qquad\quad \rightarrow 0 \\
\qquad\qquad\qquad\quad Cons\ x\ xs \rightarrow x + sumList\ xs
\end{array}
$$

The first step in applying the automatic parallelization technique to *sumList* is to derive the functions for converting data of type *List Int* to and from a well-partitioned *join*-list containing data of type *List′* using our automatic partitioning technique, the results of which are shown below:

$$
\begin{array}{lll}
data\ List' & ::= & Nil' \\
 & \mid & Cons'\ Int \\
 & & \\
partition_{(List\ Int)} & = & partition \circ flatten_{(List\ Int)} \\
 & & \\
flatten_{(List\ Int)} & = & \lambda xs.\mathbf{case}\ xs\ \mathbf{of} \\
 & & \quad Nil \qquad\quad \rightarrow [Nil'] \\
 & & \quad Cons\ x_1\ x_2 \rightarrow [Cons'\ x_1] +\!\!+ flatten_{(List\ Int)}\ x_2 \\
 & & \\
rebuild_{(List\ Int)} & = & fst \circ unflatten_{(List\ Int)} \circ rebuild \\
 & & \\
unflatten_{(List\ Int)} & = & \lambda xs.\mathbf{case}\ xs\ \mathbf{of} \\
 & & \quad (x : xs) \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
 & & \qquad\qquad Nil' \qquad \rightarrow (Nil,\ xs) \\
 & & \qquad\qquad Cons'\ x_1 \rightarrow \mathbf{case}\ unflatten_{(List\ Int)}\ xs\ \mathbf{of} \\
 & & \qquad\qquad\qquad\qquad (x_2,\ xs_2) \rightarrow (Cons\ x_1\ x_2, xs_2)
\end{array}
$$

After generating these conversion functions, we compose *sumList* with $rebuild_{(List\ Int)}$ and distill this composition, $\mathcal{D}[\![sumList \circ rebuild_{(List\ Int)}]\!]$, to generate an efficient sequential program equivalent to *sumList* defined on a partitioned *join*-list. The resulting function, $sumList_{wp}$ is shown below:

$$
\begin{array}{l}
sumList_{wp} = \lambda x.\mathbf{case}\ x\ \mathbf{of} \\
\qquad \begin{array}{lll}
Singleton\ x & \rightarrow & \mathbf{case}\ x\ \mathbf{of} \\
 & & \quad Nil' \rightarrow 0 \\
Join\ l\ r & \rightarrow & sumList'_{wp}\ l\ (sumList_{wp}\ r)
\end{array}
\end{array}
$$

$$
\begin{array}{l}
sumList'_{wp} = \lambda x\ n.\mathbf{case}\ x\ \mathbf{of} \\
\qquad \begin{array}{lll}
Singleton\ x & \rightarrow & \mathbf{case}\ x\ \mathbf{of} \\
 & & \quad Nil' \quad\ \ \rightarrow n \\
 & & \quad Cons'\ x \rightarrow x + n \\
Join\ l\ r & \rightarrow & sumList'_{wp}\ l\ (sumList'_{wp}\ r\ n)
\end{array}
\end{array}
$$

After defining $sumList_{wp}$, we extract its parallelizable expressions, resulting in the definition of $sumList_{wp}$, shown below:

$$
\begin{array}{l}
sumList_{wp} = \lambda x.\mathbf{case}\ x\ \mathbf{of} \\
\qquad \begin{array}{lll}
Singleton\ x & \rightarrow & \mathbf{case}\ x\ \mathbf{of} \\
 & & \quad Nil' \rightarrow 0 \\
Join\ l\ r & \rightarrow & \mathbf{let}\ l' = sumList'_{wp}\ l \\
 & & \quad\ \ r' = sumList_{wp}\ r \\
 & & \mathbf{in}\ \ l'\ r'
\end{array}
\end{array}
$$

$$
\begin{array}{l}
sumList'_{wp} = \lambda x\ n.\mathbf{case}\ x\ \mathbf{of} \\
\qquad \begin{array}{lll}
Singleton\ x & \rightarrow & \mathbf{case}\ x\ \mathbf{of} \\
 & & \quad Nil' \quad\ \ \rightarrow n \\
 & & \quad Cons'\ x \rightarrow x + n \\
Join\ l\ r & \rightarrow & \mathbf{let}\ l' = sumList'_{wp}\ l \\
 & & \quad\ \ r' = sumList'_{wp}\ r\ n \\
 & & \mathbf{in}\ \ l'\ r'
\end{array}
\end{array}
$$

Finally, after defining $sumList_{wp}$, and extracting its parallelizable expressions, we apply $\mathcal{T}_p$ to $sumList_{wp}$ in order to explicitly parallelize its operations on well-partitioned *join*-lists. The resulting definition of $sumList_{par}$ is shown below:

```
$sumList_{par} = \lambda x.$case $x$ of
                    $Singleton\ x \rightarrow$ case $x$ of
                                      $Nil' \rightarrow 0$
                    $Join\ l\ r \quad\ \rightarrow$ let $l' = sumList'_{par}\ l$
                                          $r' = sumList_{par}\ r$
                                      in $l'\ 'par'\ r'\ 'pseq'\ l'\ r'$

$sumList'_{par} = \lambda x\ n.$case $x$ of
                    $Singleton\ x \rightarrow$ case $x$ of
                                      $Nil' \quad\ \rightarrow n$
                                      $Cons'\ x \rightarrow x + n$
                    $Join\ l\ r \quad\ \rightarrow$ let $l' = sumList'_{par}\ l$
                                          $r' = sumList'_{par}\ r\ n$
                                      in $l'\ 'par'\ r'\ 'pseq'\ l'\ r'$
```

By making distillation aware of the definition of the + operator, it can derive the necessary associativity that allows for each child of a *Join* to be evaluated in parallel. It is worth noting that in the case of the above program, when evaluating the left child of a *Join* we create a partial application which can be evaluated in parallel with the evaluation of the right child. This partial application is equivalent to $(\lambda r.l + r)$, where r is the result of the evaluation of the right operand.

As both children are roughly equal in size, each parallel process created will have a roughly equal amount of work to do. In contrast, with respect to the original *sumList* defined on *cons*-lists, if the processing of both x and *sumList xs* are performed in parallel, one process will have one element of the list to evaluate, while the other will have the remainder of the list to evaluate, which is undesirable.

7 Related Work

There are many existing works that aim to automate the parallelization process, however many of these works simply assume or require that they are provided with data for which there is a straight-forward conversion to a well-partitioned form. For example, list-homomorphisms [2–4, 10, 26] and their derivative works [5, 12, 13, 19–22, 30, 31] require that they are supplied with data in the form of a *cons*-list for which there is a simple conversion to a well-partitioned *join*-list. These techniques also require the specification of associative/distributive operators to be used as part of the parallelization process, which places more work in the hands of the developer.

Chin et al.'s [6, 7, 35] work on parallelization via context-preservation also makes use of *join*-lists as part of its parallelization process. Given a program, this technique derives two programs in *pre-parallel* form, which are then generalized. The resulting generalized function may contain undefined functions which can be defined using an inductive derivation. While such an approach is indeed powerful and does allow such complex functions as those with accumulating parameters, nested recursion and conditional statements, it also has its drawbacks. One such drawback is that it requires that associativity and distributivity be specified for primitive functions by the developer. The technique is therefore *semi-automatic* and a fully automatic technique would be more desirable. While [35] presents an informal overview of the technique, a more concrete version was specified by Chin. et. al. in [6, 7]. However, these more formal versions are still only semi-automatic and are defined for a first-order language and require that the associativity/distributivity of operators be specified. This technique is also only applicable to *list-paramorphisms* [27] and while this encompasses a large number of function definitions, it is unrealistic to expect developers to define functions in this form. Further restrictions also exist in the transformation technique as the *context-preservation property* must hold in order to ensure the function can be parallelized.

While these techniques are certainly powerful, requiring that programs are only developed in terms of *cons*-lists is an unrealistic burden to place upon the developer, as is requiring that the associativity/distributivity of operators be specified. An important limitation to these techniques is that they are only applicable to lists, excluding the large class of programs that are defined on trees. One approach to parallelizing trees is that of Morihata et. al.'s [28] redefinition of the third homomorphism theorem [11] which is generalized to apply to trees. This approach makes use of zippers [23] to model the path from the root of a tree to an arbitrary leaf. While this approach presents an interesting approach to partitioning the data contained within a binary tree, the partitioning technique is quite complicated and relies upon zippers. It also presents no concrete methodology for generating zippers from binary-trees and assumes that the developer has provided such a function. It also requires that the user specify two functions in upward and downward form [28] which is quite restrictive so it is not realistic to expect a developer to define their programs in such a manner.

8 Conclusion

In conclusion, this paper has presented a novel, fully automatic parallelization technique for functional programs defined on any data-type. By defining a technique by which a developer can automatically parallelize programs, the difficulties associated with the parallelization process can be removed from the developer, who can continue developing software within the 'comfortable' sequential side of development and have equivalent parallel software derived automatically as needed.

Where existing automated parallelization techniques are restrictive with respect to the form of their input programs and the types they are defined on, the presented parallelization technique holds no such restrictions due to its use of our automatic data partitioning technique which converts data of any type into a well-partitioned form. To the best of the authors knowledge this is the first automated parallelization technique that is applicable to programs defined on any data-type.

One potential problem with our automatic parallelization technique is that it may get down to such a fine level of granularity of divide-and-conquer parallelism, that it is merely sparking a large number of trivial processes in parallel. This is obviously undesirable due to being wasteful of resources and potentially having a negative impact on efficiency due to the overheads associated with sparking parallel processes.

A solution to this problem is to control the level of granularity via the use of thresholding to govern the sparking of new parallel processes. Such an approach will also give the developer a measure of control over the parallelism obtained in the output program. Thresholding can be used to prevent the parallelization of *join*-lists whose size falls below a certain point. Research is ongoing to determine an optimal thresholding strategy for our automatically parallelized programs.

While our research is focused on divide-and-conquer task parallelization, it is also worth noting the potential for the system to be used as part of a data-parallel approach. If $partition_{(T\ T_1 \ldots T_g)}$ is redefined to generate a flattened list representation of the input data-structure, this could then be partitioned into chunks and distributed across a data-parallel architecture, such as a GPU. This would require a redefinition of $\mathcal{T}_p$ in order to support such an approach, which would implement the data-parallelism in the resulting program ensuring that the same function is applied to each chunk of data in parallel. Research is currently underway to extend the presented work to support automatic partitioning to enable GPU parallelization [24].

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 10/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre.

References

1. L. Augustsson. Compiling Pattern Matching. *Functional Programming Languages and Computer Architecture*, 1985.
2. R. Backhouse. An Exploration of the Bird-Meertens Formalism. Technical report, In STOP Summer School on Constructive Algorithmics, Abeland, 1989.
3. R. Bird. Constructive Functional Programming. *STOP Summer School on Constructive Algorithmics*, 1989.
4. R. S. Bird. An Introduction to the Theory of Lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42, New York, NY, USA, 1987. Springer-Verlag New York, Inc.
5. G. E. Blelloch. Scans as Primitive Operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.
6. W.-N. Chin, S.-C. Khoo, Z. Hu, and M. Takeichi. Deriving Parallel Codes via Invariants. In J. Palsberg, editor, *Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 75–94. Springer Berlin Heidelberg, 2000.
7. W. N. Chin, A. Takano, Z. Hu, W. ngan Chin, A. Takano, and Z. Hu. Parallelization via Context Preservation. In *In IEEE Intl Conference on Computer Languages*, pages 153–162. IEEE CS Press, 1998.

8. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.
9. M. Dever and G. W. Hamilton. Automatically Partitioning Data to Facilitate the Parallelization of Functional Programs. *Proceedings of the Eight International Andrei Ershov Memorial Conference*, July 2014.
10. M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72. University of Utrecht, September 1992.
11. J. Gibbons. The Third Homomorphism Theorem. *Journal of Functional Programming*, 6(4):657–665, 1996. Earlier version appeared in C. B. Jay, editor, *Computing: The Australian Theory Seminar*, Sydney, December 1994, p. 62–69.
12. S. Gorlatch. Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In *In Annual European Conference on Parallel Processing, LNCS 1124*, pages 401–408. Springer-Verlag, 1996.
13. S. Gorlatch. Systematic Extraction and Implementation of Divide-and-Conquer Parallelism. In *Programming languages: Implementation, Logics and Programs, Lecture Notes in Computer Science 1140*, pages 274–288. Springer-Verlag, 1996.
14. G. Hamilton and N. Jones. Distillation and Labelled Transition Systems. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation*, pages 15–24, January 2012.
15. G. Hamilton and N. Jones. Proving the Correctness of Unfold/Fold Program Transformations using Bisimulation. *Lecture Notes in Computer Science*, 7162:153–169, 2012.
16. G. W. Hamilton. Distillation: Extracting the Essence of Programs. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation*, 2007.
17. G. W. Hamilton. Extracting the Essence of Distillation. *Proceedings of the Seventh International Andrei Ershov Memorial Conference: Perspectives of System Informatics*, 2009.
18. G. W. Hamilton and G. Mendel-Gleason. A Graph-Based Definition of Distillation. *Proceedings of the Second International Workshop on Metacomputation in Russia*, 2010.
19. Z. Hu, H. Iwasaki, and M. Takechi. Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms. *ACM Trans. Program. Lang. Syst.*, 19(3):444–461, May 1997.
20. Z. Hu, M. Takeichi, and W.-N. Chin. Parallelization in Calculational Forms. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 316–328, New York, NY, USA, 1998. ACM.
21. Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating Efficient Parallel Programs. In *In 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–94, 1999.
22. Z. Hu, T. Yokoyama, and M. Takeichi. Program Optimizations and Transformations in Calculation Form. In *Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 144–168, Berlin, Heidelberg, 2006. Springer-Verlag.
23. G. Huet. The Zipper. *J. Funct. Program.*, 7(5):549–554, Sept. 1997.
24. V. Kannan and G. W. Hamilton. Distillation to Extract Data Parallel Computations. *Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation*, July 2014.

25. H.-W. Loidl, P. W. Trinder, K. Hammond, A. Al Zain, and C. A. Baker-Finch. Semi-Explicit Parallel Programming in a Purely Functional Style: GpH. In M. Alexander and B. Gardner, editors, *Process Algebra for Parallel and Distributed Processing: Algebraic Languages in Specification-Based Software Development*, pages 47–76. Chapman and Hall, Dec. 2008.
26. G. Malcolm. Homomorphisms and Promotability. In *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*, pages 335–347, London, UK, 1989. Springer-Verlag.
27. L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
28. A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. The Third Homomorphism Theorem on Trees: Downward & Upward lead to Divide-and-Conquer. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 177–185, New York, NY, USA, 2009. ACM.
29. D. Skillicorn. *Foundations of Parallel Programming.* Cambridge International Series on Parallel Computation. Cambridge University Press, 2005.
30. D. B. Skillicorn. Architecture-Independent Parallel Computation. *Computer*, 23:38–50, December 1990.
31. D. B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In *Software for Parallel Computation, volume 106 of NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.
32. M. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. *International Logic Programming Symposium*, pages 465–479, 1995.
33. M. Sørensen, R. Glück, and N. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 1(1), January 1993.
34. M. H. Sørensen. Turchin's Supercompiler Revisited - An Operational Theory of Positive Information Propagation, 1996.
35. Y. M. Teo, W.-N. Chin, and S. H. Tan. Deriving Efficient Parallel Programs for Complex Recurrences. In *Proceedings of the second international symposium on Parallel symbolic computation*, PASCO '97, pages 101–110, New York, NY, USA, 1997. ACM.
36. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.
37. P. Wadler. Efficient Compilation of Pattern Matching. In S. P. Jones, editor, *The Implementation of Functional Programming Languages.*, pages 78–103. Prentice-Hall, 1987.
38. P. Wadler. Deforestation: Transforming Programs to Eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.