AutoPar: Automating the Parallelization of Functional Programs

Michael Dever

Bachelor of Science in Computer Applications

A Dissertation submitted in fulfilment of the
requirements for the award of
Doctor of Philosophy (Ph.D.)

to the

School of Computing,
Dublin City University

Supervisor:	Dr. Geoffrey Hamilton
Submitted:	September 2015

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: _____ (Candidate)

I.D. Number: _____

Date: _____

Contents

List of Figures

List of Tables

Abstract

Automating the Parallelisation of Functional Programs

Michael Dever

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 and is fraught with many pitfalls, such as race conditions and locking issues, amongst others. Developers are typically more comfortable developing sequentially, yet as the limitations of single-core processor speeds are reached, they have no choice but to reach for parallel implementations to obtain the required performance increases.

An obvious solution to the parallelisation problem is to allow developers to continue to develop sequentially and generate efficient parallel programs automatically from these sequential ones. There are many existing techniques which automate the parallelisation process, however these techniques place many constraints upon the programs they are applicable to.

This thesis defines a fully automatic parallelisation technique which places no restriction on its input programs and is applicable to programs defined using any data-type. The technique consists of two components: the first allows a given program to be redefined in terms of well-partitioned data. The second then explicitly parallelises the resulting program using Glasgow parallel Haskell.

The technique is applied to several Haskell programs, the results of which have then been benchmarked with respect to the performance of hand-parallelised versions of the original programs. The benchmarking process has recorded the execution time and parallel performance of each benchmark program. The evaluation of the benchmark results has allowed for the merit of the automated parallelisation technique to be shown.

Acknowledgements

I would like to express my gratitude to several people without whom I would never have completed this thesis.

Without question, my deepest gratitude and admiration goes to my supervisor, Dr. Geoff Hamilton, whose constant patience, support and encouragement made the research process much more rewarding.

My family also played an important part in getting me to this point and without them, I would not be who or where I am. I would especially like to acknowledge my grandparents, Paddy and Patricia, and my uncle Pat, as well as my sister, Aimée, whose support, encouragement and belief in me appears to be endless; I will be forever grateful.

My friends, of course, must be thanked too: Conor, Eileen, Eoghan, Gav, Gearalt (G9), Katie, Phil, Seanan and Sweeney, you have all contributed to who I am and I'm better for it.

Finally, I would also like to thank the DCU School of Computing and the LERO research centre for providing an enjoyable and rewarding environment to learn in. IBM and JBA must also be acknowledged for their support during an industrial placement during my research.

Chapter 1

Introduction

1.1 Background

As the pervasiveness of parallel architectures in modern computing increases and the limits of single core processor speeds are reached, it has become apparent that there is an obvious need for efficiently implemented parallel programs. Efficient parallel programs are required in order to create faster software, as it is not possible to simply rely upon increases in processor speed for the necessary gains anymore. However, there are many issues associated with the parallelisation process which a developer must face, such as considerations relating to the underlying parallel hardware, communication costs and other related issues, the selection of an appropriate programming language and parallelisation technique, and ensuring that the parallel processes created perform a significant amount of work. As developers are comfortable developing in a sequential environment where none of these issues present themselves, an obvious solution to the parallelisation problem is to allow developers to continue to develop sequentially and generate efficient parallel programs automatically from these sequential ones. However, these issues and many more need to be dealt with "under the hood" and as a result the development of such a system is an immensely complicated task, yet even with these numerous constraints and complications, it is a problem that must be addressed in order to improve program performance from this point out.

While there exist many differences between the dominant imperative and the lesser used, but still powerful functional programming paradigm, functional languages are well suited to parallel programming for several reasons.

One important benefit offered by that of functional languages is that of *referential transparency*. Expressions are by nature stateless: they are defined by their inputs and have no effect other than to calculate their results. Therefore, one process executing an expression on a set of data can have no impact on another process executing another expression on another set of data, provided there are no data dependencies. This lends functional programs an *implicit task parallelism*, in that expressions can be executed in parallel as long as there are no data dependencies between them.

Generally, software based approaches to parallelisation fall somewhere on a spectrum between *implicit* and *explicit* parallelism. At one end of the spectrum lies *implicit* parallelism, where the parallelism is hidden from the developer and they need not be concerned with the actual parallel implementation. As a result, they can simply concentrate on developing a solution to the given problem. Once a developer has implemented a solution the compiler or runtime can exploit any parallelism it identifies in the solution [20]. However, as the developer does not explicitly state how the program should be parallelised they have no control over the parallelism itself. As a result of this, if the compiler does not produce an optimal parallel program, it is difficult for the developer to tune its parallel performance without knowledge of the compiler's parallelisation process.

At the other end of the spectrum lies *explicit* parallelism which provides the developer with very fine-grained control of the parallelism involved in a program as the developer must explicitly state which instructions are to be evaluated in parallel. While there are certainly benefits associated with this approach, such as fine-grained control and the ability to debug programs that do not offer the expected parallel performance, it also presents certain difficulties. Where an implicit approach takes care of all of the issues associated with parallelisation, an explicitly parallel approach forces the developer to deal with all of these issues.

In between both ends of the spectrum lie other parallelisation techniques which offer varying degrees of both explicit and implicit parallelism. For example, some techniques [63, 80] provide straightforward methods of enabling parallelism via the use of annotations or directives within a program. These annotations can be used to mark (*explicitly*) expressions which should be evaluated in parallel, and it is then left to the run-time to manage their parallel evaluation (*implicitly*). The benefit of using such techniques is that they

allow the developer a measure of control over what is evaluated in parallel, but abstract away from the underlying parallel architecture and run-time, removing these issues and worries from the developer.

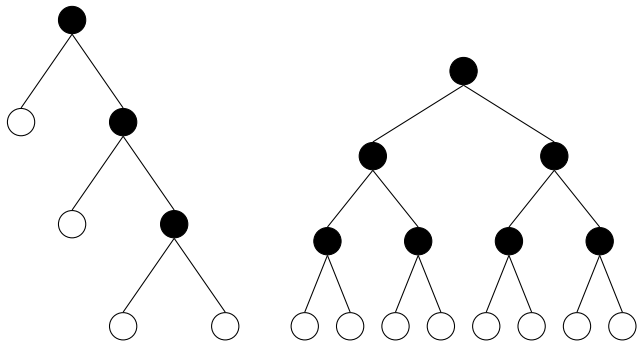
Along with deciding upon which approach to use, a developer must be aware of which parallel hardware model they will implement their program on. There are many parallel models such as *multi-core* (well suited to a task-parallel approach) and *GPU* (well suited to a data-parallel approach), amongst others.

Multi-core computers are machines which contain at least one processor consisting of many cores. These cores may not be identical and may use different memory models; for example, in some systems each core may have its own local memory and in others each core shares the same non-local memory. As multi-core computers are increasingly pervasive, both for personal and industrial use, it makes sense to focus the research presented in this thesis on such machines. As there is a pre-existing massive deployment of multi-core computers, from desktops and mobiles, to servers and purpose built high-performance machines, there is potential for the research presented in this thesis to have far reaching impact.

Further to just utilising one of these parallel models, many of these can be combined in order to extend the opportunities for parallelisation available to the developer. Simply deciding upon both a software and hardware model for a parallel program is not the end of the issues a developer may face. Central to the development of a parallel program is the data that the program evaluates. Whether developing using task or data parallelism, on a GPU or on a multicore processor, the decomposition or partitioning of the data that the program uses is central to developing an efficient parallel program.

As an example of the impact that the partitioning of data can have on parallel processing, consider a function operating on a binary-tree containing some data. If a developer parallelises this function in a divide-and-conquer fashion, but at each division, the tree is poorly partitioned as shown in Figure 1.1a, then the resulting parallel processes will be of unequal size, resulting in a very unbalanced distribution of work across parallel processes. However, if the tree is well-partitioned, as shown in Figure 1.1b, then the resulting processes will be of roughly equal size resulting in a more balanced distribution of work across parallel processes.

It is not always intuitive to define programs in terms of a data-type in



(a) Poorly Partitioned (b) Well-partitioned

Figure 1.1: Examples of Partitioned Data

which the data can be easily well-partitioned and ensure that the data it contains is well-partitioned. Even where this is possible, it can be difficult to implement a solution to a problem defined on such a data-type. For example, not all trees are binary-trees; *rose-trees* can have any number of children, and ensuring that these are well-partitioned is a more complicated process. Developing parallel programs to work with rose-trees is a lot more complicated than that of binary trees, and this complexity only increases when there are many nodes with differing numbers of children.

As can be seen from the above, the variety of concerns that a developer has with respect to parallel development is quite large, and only grows when combinations of these techniques, strategies, and models are used. In order to alleviate these concerns, the methods by which developers implement parallel programs need to change dramatically. As developers are more comfortable developing sequential programs [72], it makes sense to develop a solution to these problems that will allow them to continue to develop sequential programs, and automatically convert these programs into equivalent parallel programs that well-partition the data into a form suitable for parallelisation.

Such a solution would enable developers to be more productive, as they could then simply concentrate on solving problems sequentially. This would reduce the developer costs associated with parallel development as it would remove the time needed to implement parallel programs. It would also remove the need for the knowledge and specialist skill required to implement efficient parallel programs. Such a solution could also take into account the underlying architecture of the machine and use this information as part of

its parallelisation process to ensure full utilisation of the available hardware.

1.2 Research Objective & Requirements

The research presented in this thesis is concerned with employing program transformation techniques to *automatically* transform a sequential program into an equivalent parallel program that results in improved performance, specifically with respect to execution time across parallel processes. In order to complete this objective several core requirements have been identified that should be satisfied by the research:

1. The developer should only have to supply the sequential program to the parallelisation technique.
2. An equivalent parallel program should be derived from the sequential one.
3. The parallelisation technique should be *fully automatic*.
4. A developer should be allowed to develop their program in an intuitive manner.
5. A developer should not have to ensure that the data contained in their data-types is well-partitioned.

While there exist many works which automate the parallelisation process to some degree [70, 69, 7, 36, 39, 40, 41, 78, 13, 15], these techniques are often quite complicated and require additional information and/or that the input program is restricted in some way. Some such techniques simply assume that they are supplied with data that can be easily well-partitioned, but this may not always be the case and it is not realistic to force developers to ensure that their data fits into such a narrow form.

Such restrictions place an unnecessary burden upon developers in addition to the complexity of the parallelisation technique being used. Put simply, it should be possible to generate parallel programs from sequential programs without placing any burden on the developer beyond implementing the sequential program. If any extra information is required from the developer, it should be restricted only to information relating to the underlying parallel architecture e.g. the number of cores to be used or the number of parallel processes to be created.

1.3 Research Hypotheses & Questions

Upon completion of a broad review of related works and techniques, presented in Chapter 2, the following hypothesis was developed.

> **Research Hypothesis**
> *Program transformation techniques can be used to automatically partition data and automatically parallelise a given sequential program with performance comparable to a hand-parallelised version of the given program.*

In order to prove this hypothesis, a pathway from sequential programs defined on arbitrary data to equivalent parallel programs defined using well-partitioned data will have to be established. This will provide evidence of the benefits of program transformation with respect to the reduction of the overheads and intricacies of the parallelisation process currently placed upon developers.

In order to focus the research related to this hypothesis, several research questions have been identified:

R.Q. 1 *Given any data-type can a corresponding data-type be defined which will allow for efficient partitioning of the data?*

R.Q. 2 *Can program transformation be used to automatically redefine a program defined on any data-type into an equivalent program defined on well-partitioned data?*

R.Q. 3 *Given a program defined over any data, can program transformation be used to automatically parallelise that program using well-partitioned data?*

R.Q. 4 *Are the resulting automatically parallelised programs efficient with respect to the performance of a hand-parallelised version of the input program?*

While many existing works are based upon the idea that well-partitioned data tends to parallelise better than poorly-partitioned data [36, 39, 40, 41, 78, 13, 15], and some works define partitioning techniques for specific data-types [60, 53], no previous work, of which the author is aware, has defined a

general technique to convert any data into a well-partitioned representation automatically. The answers to R.Q. 1 and R.Q. 2 will determine whether or not it is possible to automatically partition the data contained in any data-type and whether or not a program defined on any data-type can be converted into one defined on well-partitioned data automatically.

In addition to this, while some previous works have developed semi-automatic parallelisation techniques based on known easily partitioned data-types [36, 39, 40, 41, 78, 13, 15], no previous work of which the author is aware has defined a means to automatically parallelise programs defined on any data-type. Thus, the answers to R.Q. 3 and R.Q. 4 will establish whether or not programs defined on any data-type can be automatically parallelised and determine the benefits of such a transformation technique.

1.4 Preliminaries

1.4.1 Language

The language which is used to present this research is a simple higher-order functional language. The language is named *hop* (for *h*igher-*o*rder *p*arallelisation).

data $T\ \alpha_1 \ldots \alpha_g ::= c_1\ t_{1_1}\ \ldots\ t_{1_{n_1}}$		Data-Type
	$\vdots$	
	$\mid\ c_m\ t_{m_1}\ \ldots\ t_{m_{n_m}}$	
$t ::= \alpha$		Type Variable
$\mid\ T\ t_1 \ldots t_g$		Type Application
$e ::= x$		Variable
$\mid\ c\ e_1 \ldots e_k$		Constructor Application
$\mid\ f$		Function
$\mid\ \lambda x.e$		Lambda Abstraction
$\mid\ e_0\ e_1$		Application
$\mid$ **case** e_0 **of** $p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k$		Case Expression
$\mid$ **let** $x = e$ **in** e'		Let Expression
$\mid\ e_0$ **where** $f_1 = e_1\ \ldots\ f_n = e_n$		Where Expression
$p ::= c\ x_1\ \ldots\ x_k$		Pattern

Figure 1.2: *hop* Language Definition

hop is defined as shown in 1.2, where 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 *hop* 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. Case expressions may only have non-nested patterns. Techniques exist to transform nested patterns into equivalent non-nested versions [2, 87].

hop also provides a multi-**let** statement, **let** $x_1 = e_1\ \ldots\ x_n = e_n$ **in** e_0, which is used as shorthand to represent a series of nested **let** statements as shown below:

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

The intended operational semantics of *hop* is normal order reduction. It is assumed that erroneous terms such as $(c\ e_1 \ldots e_k)\ e$ and **case** $(\lambda v.e)$ **of** $p_1 \rightarrow e_1 \mid \cdots \mid p_k \rightarrow e_k$ cannot occur. The variables in the patterns of **case** expressions, **let** statements and the arguments of λ-abstractions are *bound*; all other variables are *free*. $fv(e)$ and $bv(e)$ denote the free and bound variables respectively of an expression, e.

$$\begin{array}{lll} \mathit{data\ List\ a} & ::= & \mathit{Nil} \\ & \mid & \mathit{Cons\ a\ (List\ a)} \\ \\ \mathit{data\ Pair\ a\ b} & ::= & \mathit{Pair\ a\ b} \end{array}$$

Figure 1.3: *cons*-List and $Pair$ Type Definitions

Within *hop*, the type definitions for *cons*-lists and pairs are defined as shown in Fig. 1.3. The usual notations are used 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 *head* x and *tail* xs ($Cons\ x\ xs$). When dealing with pairs, $Pair\ x\ y$ may be denoted as a tuple (x, y). Expression substitution, denoted $e[e'/x]$, allows for simultaneously replacing all occurrences of the variable x with the expression e'.

$$
\begin{array}{rcl}
e & ::= & x \\
& | & c\ e_1\ \ldots\ e_n \\
& | & f \\
& | & e\ x \\
& | & \lambda x.e \\
& | & \textbf{case}\ x\ \textbf{of}\ p_1 \rightarrow e_1\ |\ldots|\ p_k \rightarrow e_k \\
& | & \textbf{let}\ x = e_0\ \textbf{in}\ e_1 \\
& | & f\ x_1\ \ldots\ x_n\ \textbf{where}\ f = \lambda x_1\ \ldots\ x_n.e
\end{array}
$$

Figure 1.4: *hoplet* Language

Additionally, *hop* contains some useful built-in functions: *split* which takes a *cons*-list and splits it in half returning a *Pair* containing the left and right halves of the split *cons*-list and $+\!\!+$ which concatenates two *cons*-lists. For example, *split* $[1, 2, 3]$ returns a *Pair* containing the two lists [1] and $[2, 3]$. Given two lists, [1] and $[2, 3]$, [1] $+\!\!+$ $[2, 3]$ results in the list $[1, 2, 3]$.

The implementation of the research presented in this thesis will be performed using Haskell. There are many benefits [81] to using such a Haskell for the purposes of this research and its implementation, described previously. One of the most important benefits of using Haskell is that it also provides many parallelisation techniques ranging from implicitly parallel to explicitly parallel, presented in Section 2.2, making it an ideal language to be used as part of an automatic parallelisation technique.

The program transformations defined in this thesis are presented using a **let**-normal form [67] of the *hop* language, called *hoplet*, which is shown in Figure 1.4. Within *hoplet* no variable, which has been introduced via a **let**-statement may appear as the selector of a **case** expression. In addition to this, the selectors of **case** expressions and application arguments may only be variables.

1.4.2 Evaluation Environment

One of the key components of the research presented in this thesis is the evaluation of the performance of parallel programs resulting from the automatic parallelisation technique it defines. In order to complete this, a series of benchmark programs, presented in Chapter 6 will be evaluated in a benchmark environment with respect to their execution times and parallel behaviours.

Each of the benchmark programs will be evaluated using a 2013 Mac Pro. The benchmark environment has $64GB$ of available DDR3 memory operating at $1866MHz$ and a 2 processor, 12-core Intel Xeon E5 Processor. Each of the twelve cores operates at $2.7GHz$ with a $30MB$ L3 cache. The benchmark environment runs OS X Yosemite 10.10 and each benchmark program will be compiled using GHC 7.8.3 using version 3.2.0.5 of the *parallel* package.

1.5 Proposed Solution

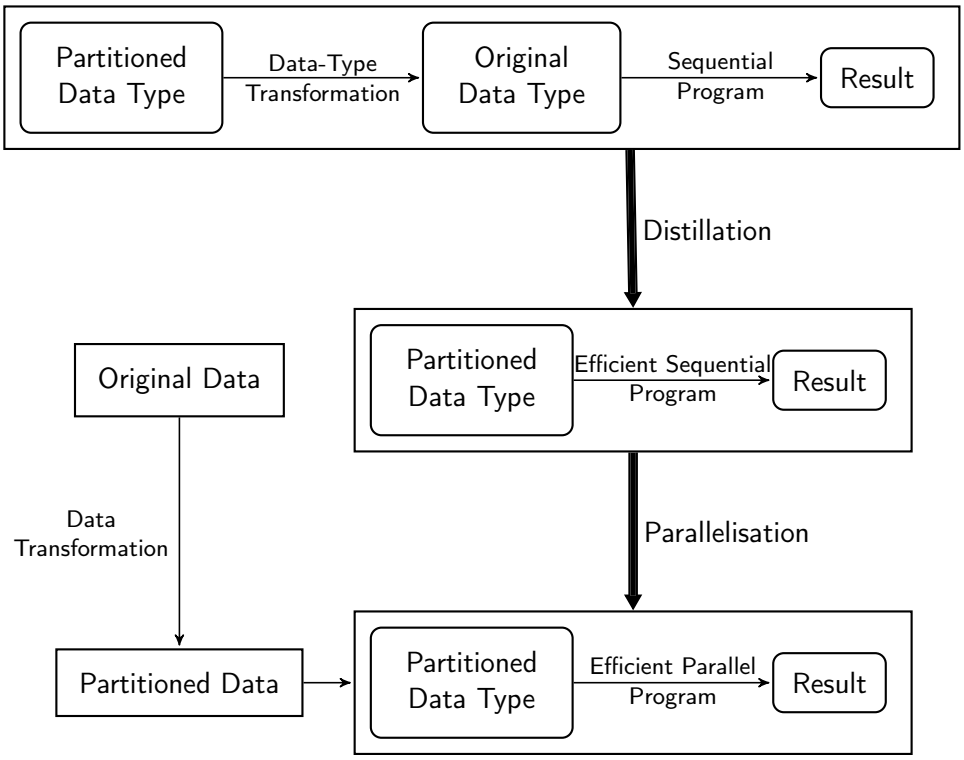


Figure 1.5: Proposed Solution

To achieve the research objectives described previously, the proposed solution is to design and evaluate an automatic parallelisation system. This system consists of two core components: a data-partitioning component which makes use of the *distillation* transformation system [30, 33], and a parallelising component. Using these components, a technique has been defined by which a developer can automatically convert a sequential program

into an equivalent parallel program defined using well-partitioned data. As a result, the difficulties associated with the parallelisation process will 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. A high-level diagram of the process is presented in Figure 1.5.

The data-partitioning component is used to derive an equivalent, well-partitioned representation of the data over which an input program is defined in order to ensure that an efficient parallel evaluation of the program can be derived. Distillation is used to compose this component with the original program in an efficient manner. Finally, a parallelisation transformation is applied to the distilled program in order to evaluate functions in parallel using an appropriate partitioning strategy.

1.6 Thesis Structure

This thesis presents a full working of the proposed solution, firstly presenting the necessary theory, followed by a full evaluation of the technique applied to several benchmark programs. The remainder of this thesis is structured as follows: Chapter 2 presents a review of related work: hand-parallelisation techniques, program transformation systems and existing work that automates the parallelisation process.

Chapter 3 presents a novel data partitioning technique which, given a program defined over any data-type, defines a function to convert an instance of that data-type into well-partitioned data. The data-type transformation system also defines a function to convert this well-partitioned data back into an instance of the original data-type. Using these conversion functions the partitioning technique makes use of distillation in order to redefine the given program in-to one defined on well-partitioned data.

Chapter 4 presents a novel, fully-automatic parallelisation transformation for functional programs which makes use of the data-type transformation presented in Chapter 3. After applying the automatic partitioning technique to a given program, the parallelisation transformation is applied in order to facilitate the parallel evaluation of functions using well-partitioned data.

Chapter 5 presents a further novel, fully automatic parallelisation transformation which makes use of a user supplied function in order to govern, or

threshold, the creation of many spurious parallel processes.

Chapter 6 presents detailed examples of the application of the automatic parallelisation technique to the programs used to evaluate the automatic parallelisation technique. Chapter 7 presents a thorough evaluation of the automatic parallelisation technique applied to these benchmark programs. Finally, Chapter 8 presents conclusions drawn from the material presented in this thesis and any potential improvements and suggestions for future work that have been identified.

Chapter 2

Related Work

2.1 Program Transformation

Program transformation is the process of taking an input program and manipulating it via various methodologies, discussed below, to a semantically equivalent program [64]. The goal of such a transformation is to enhance and improve the original program. Program transformation can be used to eliminate multiple accesses/traversals of data, to eliminate the use of intermediate data, convert programs from one language to another and to introduce particular behaviours to a program such as parallelisation, as well as for other purposes such as run-time or memory usage optimisation.

As a result of this survey of related work, two main approaches to the transformation of functional programs were identified. The first approach is fold/unfold based transformation systems, based on the works of Burstall and Darlington (1977) [10]. The second approach is calculational methods based transformation systems. The calculational methods based transformation systems described here are defined using the Bird-Meertens Formalisms (1987), which guarantee the correctness of the resulting program [5, 4, 6, 50, 19, 3, 69].

Each of these transformation approaches lead to the creation of several further powerful program transformations. For example, the fold/unfold technique of Burstall & Darlington resulted in the definitions of the *deforestion* transformation [88], defined in Section 2.1.1.1, which is capable of a linear increase in efficiency, the *positive-supercompilation* transformation [74], defined in Section 2.1.1.2, which is also capable of a linear increase in

efficiency and the *distillation* transformation [33], defined in Section 2.1.1.3, which is capable of a super-linear increase in efficiency. The Bird-Meertens Formalisms (BMF) led to the creation of techniques such as shortcut-fusion, defined in Section 2.1.2 which is the calculational equivalent of deforestation. The BMF also led to the creation of several powerful parallelisation transformations such as *diffusion*, defined in Section 2.3.2.5, and the *accumulate* skeleton, defined in Section 2.3.2.6, which are capable of parallelising complicated programs involving accumulating parameters, non-linear recursion and conditional statements.

2.1.1 Fold/Unfold Transformations

Burstall & Darlington [10] created one of the seminal works on program transformation in their *fold/unfold* based rules and strategies for program transformation. This approach is at the core of many transformation techniques, such as *deforestation*, *supercompilation* and *distillation*. The system is based upon a set of inference rules and laws for the application of these rules. At the cores of these inference rules lie the *folding* and *unfolding* transformations.

Given a function g, that has the body e, the unfolding rule allows a call to g to be replaced by a corresponding instance of e, with any arguments to g substituted into the correct positions. For example, if the function g is defined by $g = \lambda x\ y.x + y$, the unfolding rule allows the function call (g 1 2) can be replaced by the expression $(1+2)$. Conversely, the folding rule allows an instance of e to be replaced with an appropriate call to g. For example, the folding rule allows the expression $(1 + 2)$ to be replaced by the function call (g 1 2).

In addition to folding and unfolding, several other inference rules are introduced and are shown below:

Definition The *definition* rule allows for the introduction of a new function, for example:

$$\begin{array}{rl} a + b + c - d * n & \\ \Downarrow & \{\text{Define } f\} \\ f = \lambda a.\ \lambda b.\ \lambda c.\ \lambda d.\ \lambda n.\ a + b + c - d * n & \end{array}$$

Instantiation The *instantiation* rule allows for the introduction of instances of already known expressions, for example:

$$\begin{array}{lcll} f & = & \lambda x.g\ x\ n & \\ & & \Downarrow & \{\text{Instantiate } n = 1\} \\ f & = & \lambda x.g\ x\ 1 & \end{array}$$

Let Abstraction Given an expression e, with sub-expressions $e_1, \ldots, e_n$, the ***let**-abstraction* rule allows for $e_1, \ldots, e_n$ to be abstracted, for example:

$$\begin{array}{lcll} f & = & \lambda x.\lambda y.(x * x) - (y + y) & \\ & & \Downarrow & \{\text{Abstract } (x * x) \text{ and } (y + y)\} \\ f & = & \lambda x.\lambda y.\textbf{let } x' = x * x & \\ & & \quad\quad y' = y + y & \\ & & \textbf{in } x' - y' & \end{array}$$

Pettorossi & Proietti added to the work of Burstall & Darlington by presenting a system [64] containing much the same set of transformation rules, but with the addition of further transformation strategies: *composition, tupling* and *generalisation*. These strategies are shown below:

1. *Composition Strategy*
   When an expression $g\ (h\ x)$ occurs as a subexpression within an expression e:
   - Create a new function, using the definition rule, $f = \lambda x.g\ (h\ x)$
   - Find a recursive definition for f in which there is no occurrence of g or h.
   - Fold e, replacing all occurrences of $g\ (h\ x)$ with $f\ x$
     - The benefit of such a strategy is obvious when we consider that h may produce some intermediate data-structure consumed by g, and by removing the composition the production of this intermediate structure will be eliminated which will result in improved efficiency.
     - Deforestation [88, 22, 29] is a good example of the use of the composition strategy for transformation.

2. *Tupling Strategy*
   On encountering a function, f, of the form:

$$f = \lambda x_1 \ \ldots \ x_n . \ldots f_1(e_1) \ldots f_r(e_r) \ldots$$

   in which $e_1, ..., e_r$ all use the free variable x:

   - Create a new function, using the definition rule, shown below:

$$h = \lambda x \ y_1 \ \ldots \ y_m . (f_1(e_1), \ldots, \ f_r(e_r))$$

     in which $x, y_1, \ldots, y_m$ are free variables occurring in $e_1, \ldots, e_r$
   - Find a recursive definition for $h \ x \ y_1 \ \ldots \ y_m$ in which there are no occurrences of $f_1, ..., f_r$
   - Fold the body of f, along with **let** abstraction, to replace it with:

$$\begin{aligned} &\textbf{let } (u_1, \ldots, u_r) = h \ x \ y_1 \ \ldots \ y_m \\ &\textbf{in } f = \lambda x_1 \ \ldots \ x_n . \ldots u_1 \ \ldots \ u_r \ldots \end{aligned}$$

     - $u_1, \ldots \ u_r$ are freshly named variables.
     - It may also be worthwhile to replace all occurrences of $f_i(e_i)$ with the i-th projection of $h \ x \ y_1 \ \ldots \ y_m$, as this may avoid multiple access to the same data structure/function as the result of h will only be calculated once and can be reused repeatedly.
     - The tupling strategy is useful in cases where several functions share the result of some computation. In such cases, these functions can be tupled together, eliminating multiple accesses to an intermediate data structure [38].
     - It is worth noting that while the tupling strategy can be quite powerful if each of the components of the tuple it returns are evaluated, it is not without issue. If one or more of these remains unevaluated then this can cause space-leaks in the resulting program [75]. To solve this problem, there are many approaches which can be used to alleviate these space-leaks and improve the performance of tupled expressions [62, 14, 65].

3. *Generalisation Strategies*

- *Generalisation from expressions to variables*
  When a recursive function f occurs, such as $f = \lambda x_1 \ \ldots \ x_n.e$, in which e' is a sub-expression of e it is possible to:
  - Create a new generalised recursive function, using the definition rule, $g = \lambda x \ y_1 \ \ldots \ y_m.e[x/e']$, in which $x, y_1, ..., y_m$ are free variables.
  - Find a recursive definition for g.
  - Fold g, replacing calls to f with calls to g.
  - Generalising from expressions to variables can allow for the generation of tail-recursive functions where the original function was not tail-recursive.

  For example, consider the following non tail-recursive definition of the factorial function, $fact$, shown below:

$$\begin{aligned} fact = \lambda x.\ &\textbf{case}\ x == 0\ \textbf{of} \\ &True\ \ \rightarrow 1 \\ &False \rightarrow x * (fact\ (x-1)) \end{aligned}$$

  Generalisation from expressions to variables can be used to convert this definition into the following tail-recursive version:

$$fact = \lambda x.\ g\ 1\ x$$

$$\begin{aligned} g \quad = \lambda y.\ \lambda x.\ &\textbf{case}\ x == 0\ \textbf{of} \\ &True\ \ \rightarrow y \\ &False \rightarrow g\ (y * x)\ (x-1) \end{aligned}$$

- *Generalisation from functions to functions*
  On encountering a recursive definition, $f = \lambda x_1 \ \ldots \ x_n. \ldots e \ldots$ within a program P, this can be improved as follows:
  - Introduce a new function g, via the definition rule, with an arity of k, in which there are some expressions $e_1, ..., e_k$, with free variables $x_1, ..., x_n$, such that for all values of the variables, $\ldots (g\ e_1 ... e_k) \ldots \equiv \ldots e \ldots$ with respect to the program P.
  - Find a recursive definition for g
  - Replace all calls to f with calls to g

– Generalising from functions to functions may allow for an exponential reduction in execution time.

For example, consider the definition of the fibonacci function, fib, shown below:

$$
\begin{array}{l}
fib = \lambda x.\ \textbf{case}\ x == 0\ \textbf{of} \\
\qquad\qquad True\ \ \rightarrow 1 \\
\qquad\qquad False \rightarrow \textbf{case}\ x == 1\ \textbf{of} \\
\qquad\qquad\qquad\qquad True\ \ \rightarrow 1 \\
\qquad\qquad\qquad\qquad False \rightarrow (fib\ (x-2)) + (fib\ (x-1))
\end{array}
$$

By firstly generalising the constants 1 and 2, and then generalising from functions to functions, Pettorossi & Proietti are able to derive the following logarithmic definition of fib:

$$
\begin{array}{l}
fib = \lambda x.\ \textbf{case}\ x == 0\ \textbf{of} \\
\qquad True\ \ \rightarrow 1 \\
\qquad False \rightarrow \textbf{case}\ x == 1\ \textbf{of} \\
\qquad\qquad True\ \ \rightarrow 1 \\
\qquad\qquad False \rightarrow \textbf{case}\ p\ (x\ `div`\ 2)\ \textbf{of} \\
\qquad\qquad\qquad (a,b) \rightarrow \textbf{case}\ odd\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad True\ \ \rightarrow (a+b)^2 + 2*b*(a+b) \\
\qquad\qquad\qquad\qquad\qquad False \rightarrow (a+b)^2 + b^2
\end{array}
$$

$$
\begin{array}{l}
p\ \ = \lambda x.\ \textbf{case}\ x == 0\ \textbf{of} \\
\qquad\quad True\ \ \rightarrow (1,0) \\
\qquad\quad False \rightarrow \textbf{case}\ x == 1\ \textbf{of} \\
\qquad\qquad\qquad\qquad True\ \ \rightarrow (0,1) \\
\qquad\qquad\qquad\qquad False \rightarrow \textbf{case}\ p\ (x\ `div`\ 2)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad (a,b) \rightarrow \textbf{case}\ odd\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad True\ \ \rightarrow ((2*a*b+b^2),((a+b)^2+b^2)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad False \rightarrow ((a^2+b^2),(2*a*b+b^2))
\end{array}
$$

2.1.1.1 Deforestation

Following on from the above techniques Wadler [88] presented *deforestation*, an automatic transformation technique. Deforestation is an algorithm which eliminates intermediate trees and lists from functional programs by using the composition strategy, described above, and redefines functions in a new

form called *treeless form*, which is based upon previous work on *listlessness* [85, 86].

Definition 1 (Deforestation Theorem) Every expression defined using functions with treeless definitions can be effectively transformed to an expression with a treeless definition, without loss of efficiency.

According to the *deforestation theorem*, defined in Definition 1, any expression defined in terms of functions with treeless definitions can be efficiently transformed into one single treeless expression without any loss in efficiency. While the original definition of deforestation was applicable only to first-order languages, it has since been extended to support higher-order languages [51, 28, 29].

As an example, consider the function *sumSquares*, shown below:

$$
\begin{aligned}
upto &= \lambda m.\ \lambda n.\ \textbf{case}\ (m > n)\ \textbf{of} \\
&\qquad\qquad True\ \rightarrow [] \\
&\qquad\qquad False \rightarrow m : (upto\ (m+1)\ n)
\end{aligned}
$$

$$
sum = \lambda xs.\ sum'\ 0\ xs
$$

$$
\begin{aligned}
sum' &= \lambda a.\ \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad [] \rightarrow a \\
&\qquad\qquad (x : xs) \rightarrow sum'\ (a + x)\ xs
\end{aligned}
$$

$$
\begin{aligned}
squares &= \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad [] \rightarrow [] \\
&\qquad\qquad (x : xs) \rightarrow (square\ x) : (squares\ xs)
\end{aligned}
$$

$$
square = \lambda x.\ x * x
$$

$$
sumSquares = \lambda n.\ sum\ (squares\ (upto\ 1\ n))
$$

Application of deforestation to *sumSquares* results in the definition of $sumSquares'$ shown below, which has eliminated the use of all intermediate lists.

$$
sumSquares' = \lambda n.\ sumSquares''\ 0\ 1\ n
$$

$$
\begin{aligned}
sumSquares'' = \lambda x.\ \lambda y.\ \lambda n.\ &\textbf{case}\ (y > n)\ \textbf{of} \\
&True\ \rightarrow x \\
&False \rightarrow sumSquares''\ (x + square\ y)\ (y+1)\ n
\end{aligned}
$$

While this is obviously a powerful transformation system, it is restrictive in the sense that its input functions must have a treeless definition, which narrows the class of program it is capable of transforming. However, for higher-order deforestation this restriction may be removed by generalising expressions prior to transformation [28].

2.1.1.2 Supercompilation

Supercompilation, described by Turchin [82, 83], is another program transformation technique which eliminates the use of intermediate data within functional programs. Supercompilation was not very accessible in its original form, but was made more accessible via *positive-supercompilation* [74, 73, 34]. The core idea behind positive-supercompilation [74, 73] is that it uses a technique called *driving*, which in essence is a forced unfolding, to construct potentially infinite trees of states and transitions. These trees are then converted into a graph, by directing states to their predecessors, or by generalising new states and then driving again. As part of its process, positive-supercompilation performs *positive-information propagation*: passing known information about variables into the branches of case expressions.

Driving in positive-supercompilation can potentially be infinite and care must be taken to ensure that driving does indeed terminate. This can be guaranteed in part by performing folding; however this does not fully guarantee termination. Termination of positive-supercompilation is ensured by making use of a *whistle* to detect when there is a chance of non-termination. This whistle is blown when a *homeomorphic-embedding* of a previously encountered expression is encountered, and signals that generalisation should then be performed.

Definition 2 (Labelled transition systems) *A labelled transition system (LTS) is a 4-tuple* $l = (\mathcal{S}, s_0, Act, \rightarrow)$ *where:*

- $\mathcal{S}$ is a set of *states* of the LTS.
- $s_0 \in \mathcal{S}$ is the *start state*, denoted by $start(l)$.
- Act is a set of *actions* which can be one of the following:
  - x, a free variable;
  - c, a constructor in an application or case pattern;

- λ, a λ-abstraction;
- @, the function in an application;
- $\#i$, the i^{th} argument in an application or let;
- **case**, a case selector;
- **let**, a let body.

- $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ is a *transition relation*. We write $s \xrightarrow{\alpha} s'$ for a transition from state s to state s' via action α.

Hamilton et. al. [34] redefined positive-supercompilation using a *labelled transition system (LTS)* framework, where an LTS is defined as shown in Definition 2. With respect to labelled transition systems, **0** is used to denote an LTS with no transitions; the function $states(l)$ is used to denote the states of an LTS l; $s \rightarrow (\alpha_1, l_1), \ldots, (\alpha_n, l_n)$ is used to denote an LTS with root state s where $l_1 \ldots l_n$ are the LTSs obtained by following the transitions labelled $\alpha_1 \ldots \alpha_n$ respectively from s. The LTS used in Hamilton's definition of positive-supercompilation is used as the basis for discussion of both positive-supercompilation and the discussion of distillation [33], shown in Section 2.1.1.3.

Definition 3 (Embedding) A binary relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}'$ is an *embedding* of labelled transition system $l = (\mathcal{S}, s_0, Act, \delta)$ by $l' = (\mathcal{S}', s'_0, Act', \delta')$ if $(s_0, s'_0) \in \mathcal{R}$, and for every pair $(s_i, s'_i) \in \mathcal{R}$ one of the following holds:

1. $\forall s_j \in \mathcal{S}$ s.t. $(s_i \xrightarrow{\alpha} s_j) \in \delta.(\exists s'_j \in \mathcal{S}'$ s.t. $(s'_i \xrightarrow{\alpha} s'_j) \in \delta'.(s_j, s'_j) \in \mathcal{R})$
2. $\exists s'_j \in \mathcal{S}'$ s.t. $(s'_i \xrightarrow{\alpha} s'_j) \in \delta'.(s_i, s'_j) \in \mathcal{R}$

The first rule here is a *coupling* rule, while the second one is a *diving* rule. Two states are related by coupling if the same transitions are possible from each of them and the resulting states are also related by the embedding relation. Two states are related by diving if a transition can be followed in the embedding LTS and the resulting state is related to the embedded LTS state by the embedding relation. $l \lesssim l'$ is used to denote an LTS l that is coupled with the LTS l'.

Definition 4 (Generalisation of LTSs) The generalisation of LTS l with respect to LTS l' (denoted by $l \bigwedge l'$), where l' is embedded within l, is defined as shown in Fig. 2.1, where embedding is as defined in Definition 3.

Within these rules, if both LTSs have the same transitions at the top level, then these will be the transitions at the top level of the resulting generalised LTS, and the corresponding LTS components which are the targets of these transitions are further generalised. Unmatched LTS components are generalised by introducing a new generalisation variable x. The value of this variable is the unmatched LTS component, abstracted over the bound variables it contains to prevent these from being extracted outside their binders.

$$l \bigtriangledown l' = s \to (\mathbf{let}, l'')(\#1, l_1), \ldots, (\#n, l_n)$$
where $l \sqcap l' = (l'', \{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\})$ and s is a new state

$$l \sqcap l' = \begin{cases} (s \to (\alpha_1, l_1''), \ldots, (\alpha_n, l_n''), \bigcup_{i=1}^{n} \theta_i), & \text{if } l' \lesssim l \\ \text{where} & \\ l = s \to (\alpha_1, l_1), \ldots, (\alpha_n, l_n) & \\ l' = s' \to (\alpha_1, l_1'), \ldots, (\alpha_n, l_n') & \\ \forall i \in \{1 \ldots n\}. l_i \sqcap l_i' = (l_i'', \theta_i) & \\ (\mathcal{L}[\![x\ x_1 \ldots x_n,]\!]\ s\ \emptyset\ \emptyset, \{x \mapsto s_1 \to (\lambda, \ldots, s_n \to (\lambda, l) \ldots)\}), & \text{otherwise} \\ \text{where} & \\ x \text{ is fresh, } \{x_1 \ldots x_n\} = bv(l) \text{ and } s, s_1 \ldots s_n \text{ are new states} & \end{cases}$$

Figure 2.1: LTS Generalisation

Positive-supercompilation effectively performs a normal-order reduction on the LTS representation of the input program. The LTS representation of previously encountered recursive terms are 'memoized' by adding them to ρ. Recursive terms can be identified as those whose root node is the target of a renaming transition. If the LTS representation of the current recursive term is a renaming of a memoized one, then a transition is created back to the previous state, with the renaming represented by a **let**. If the LTS representation of the current recursive term is an embedding of a memoized one, then generalisation is performed, according to the generalisation rules defined in 2.1 and the components of the resulting generalised LTS are transformed separately. Generalisation ensures that a renaming of a previously encountered LTS representation of a term is eventually encountered, and that the transformation therefore terminates. If neither a renaming or embedding is encountered, then normal-order reduction is applied on the LTS representation of the current term.

If this normal-order reduction becomes 'stuck' as a result of encountering a variable in the redex position, then the context surrounding the redex is

further transformed. If the context surrounding a variable redex is a **case**, then information is propagated to each branch of the case to indicate that this variable has the value of the corresponding branch pattern.

As an example of positive-supercompilation, consider the naïve pattern matcher, *match*, shown below, where the (==) operator is defined as expected for the elements of a pattern:

$$
\begin{array}{l}
match = \lambda p.\lambda s.loop\ p\ s\ p\ s \\
\\
loop \quad = \lambda pp.\lambda ss.\lambda op.\lambda os.\ \textbf{case}\ pp\ \textbf{of} \\
\qquad\qquad\qquad\qquad [] \qquad\quad \rightarrow True \\
\qquad\qquad\qquad\qquad (p : pp) \rightarrow\ \textbf{case}\ ss\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad [] \qquad\quad \rightarrow False \\
\qquad\qquad\qquad\qquad\qquad\qquad (s : ss) \rightarrow\ \textbf{case}\ (p == s)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad True\ \rightarrow loop\ pp\ ss\ op\ os \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad False \rightarrow next\ op\ os \\
\\
next \quad = \lambda op.\lambda os.\ \textbf{case}\ os\ \textbf{of} \\
\qquad\qquad\qquad [] \qquad\quad \rightarrow False \\
\qquad\qquad\qquad (o : os) \rightarrow loop\ op\ os\ op\ os
\end{array}
$$

Application of positive-supercompilation to $(match\ AAB)$ results in the definition of $loop'$ shown below:

$$
\begin{array}{lll}
loop' & = \lambda s.\ \textbf{case}\ s\ \textbf{of} & \\
 & \quad [] & \rightarrow False \\
 & \quad (s : ss) & \rightarrow\ \textbf{case}\ (A == s)\ \textbf{of} \\
 & & \quad True\ \rightarrow loop''\ ss \\
 & & \quad False \rightarrow loop'\ ss
\end{array}
$$

$$
\begin{array}{lll}
loop'' & = \lambda s.\ \textbf{case}\ s\ \textbf{of} & \\
 & \quad [] & \rightarrow False \\
 & \quad (s : ss) & \rightarrow\ \textbf{case}\ (A == s)\ \textbf{of} \\
 & & \quad True\ \rightarrow loop'''\ ss \\
 & & \quad False \rightarrow\ \textbf{case}\ (A == s)\ \textbf{of} \\
 & & \qquad True\ \rightarrow loop''\ ss \\
 & & \qquad False \rightarrow loop'\ ss
\end{array}
$$

$$
\begin{array}{lll}
loop''' & = \lambda s.\ \textbf{case}\ s\ \textbf{of} & \\
 & \quad [] & \rightarrow False \\
 & \quad (s : ss) & \rightarrow\ \textbf{case}\ (B == s)\ \textbf{of} \\
 & & \quad True\ \rightarrow True \\
 & & \quad False \rightarrow\ \textbf{case}\ (A == s)\ \textbf{of} \\
 & & \qquad True\ \rightarrow loop'''\ ss \\
 & & \qquad False \rightarrow\ \textbf{case}\ (A == s)\ \textbf{of} \\
 & & \qquad\quad True\ \rightarrow loop''\ ss \\
 & & \qquad\quad False \rightarrow loop'\ ss
\end{array}
$$

Positive supercompilation of $(match\ AAB)$ produces a specialised pattern matcher similar to that of the Knuth-Morris-Pratt (KMP) [46] algorithm where deforestation does not. Positive supercompilation produces a version in which, if the matching test fails on the head of the input pattern, the head of the input is used to determine the next matching attempt. In the version produced by deforestation, when a match fails, the information relating to the head of the input pattern is lost, and the whole match must start again [74].

Positive supercompilation is more powerful than deforestation as it performs positive information propagation. Positive supercompilation, like deforestation, is only capable of obtaining a linear increase in the efficiency of transformed programs [76].

2.1.1.3 Distillation

While deforestation and positive-supercompilation are both powerful transformation techniques, there is another transformation, *distillation*, which is capable of obtaining a super-linear increase in efficiency [30, 31, 32, 33], and which will be used as part of the automatic parallelisation technique presented later in this thesis. Like positive-supercompilation [34], distillation

$$\mathcal{D}[\![l]\!]\ \kappa\ \rho = \begin{cases} \mathcal{D}_{\mathcal{F}1}[\![l]\!]\ \kappa\ \rho, & \text{if } \exists s \in \mathit{states}(l), \alpha.s \xrightarrow{\alpha} \mathit{start}(l) \\ \mathcal{D}_{\mathcal{R}}[\![l]\!]\ \kappa\ \rho, & \text{otherwise} \end{cases}$$

$$\mathcal{D}_{\mathcal{F}1}[\![l]\!]\ \kappa\ \rho = \begin{cases} s \to (\mathbf{let}, l'), (\#1, s_1 \to (x'_1, \mathbf{0})), \ldots, (\#n, s_n \to (x'_n, \mathbf{0})), & \text{if } \exists l' \in \rho, \sigma.(l'\sigma) \sim (\kappa \bullet l) \\ \text{where } l'\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\} \sim (\kappa \bullet l) \text{ and } s, s_1 \ldots s_n \text{ are new states} & \\ \mathcal{D}_{\mathcal{F}2}[\![(\mathcal{D}[\![l_0]\!]\ \langle\rangle\ \rho)\{1 \mapsto (\mathcal{D}[\![l_1]\!]\ \langle\rangle\ \rho), \ldots, n \mapsto (\mathcal{D}[\![l_n]\!]\ \langle\rangle\ \rho)\}]\!]\ \langle\rangle\ \rho, & \text{if } \exists l' \in \rho, \sigma.(l'\sigma) \lesssim (\kappa \bullet l) \\ \text{where } (\kappa \bullet l) \bigsqcup l' = s \to (\mathbf{let}, l_0), (\#1, l_1), \ldots, (\#n, l_n) & \\ \mathcal{D}_{\mathcal{R}}[\![l]\!]\ \kappa\ (\rho \cup \{\kappa \bullet l\}), & \text{otherwise} \end{cases}$$

$$\mathcal{D}_{\mathcal{F}2}[\![l]\!]\ \kappa\ \rho = \begin{cases} s \to (\mathbf{let}, l'), (\#1, s_1 \to (x'_1, \mathbf{0})), \ldots, (\#n, s_n \to (x'_n, \mathbf{0})), & \text{if } \exists l' \in \rho, \sigma.(l'\sigma) \sim (\kappa \bullet l) \\ \text{where } l'\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\} \sim (\kappa \bullet l) \text{ and } s, s_1 \ldots s_n \text{ are new states} & \\ s \to (\mathbf{let}, \mathcal{D}[\![l_0]\!]\ \langle\rangle\ \rho), (\#1, \mathcal{D}[\![l_1]\!]\ \langle\rangle\ \rho), \ldots, (\#n, \mathcal{D}[\![l_n]\!]\ \langle\rangle\ \rho), & \text{if } \exists l' \in \rho, \sigma.(l'\sigma) \lesssim (\kappa \bullet l) \\ \text{where } (\kappa \bullet l) \bigsqcup l' = s \to (\mathbf{let}, l_0), (\#1, l_1), \ldots, (\#n, l_n) & \\ \mathcal{D}_{\mathcal{R}}[\![l]\!]\ \kappa\ (\rho \cup \{\kappa \bullet l\}), & \text{otherwise} \end{cases}$$

$$\mathcal{D}_{\mathcal{R}}[\![s \to (x, \mathbf{0})]\!]\ \langle\rangle\ \rho = s \to (x, \mathbf{0})$$
$$\mathcal{D}_{\mathcal{R}}[\![s \to (x, \mathbf{0})]\!]\ \langle (s' \to (\mathbf{case}, \bullet), (c_1, l_1), \ldots, (c_n, l_n)) : \kappa \rangle\ \rho =$$
$$\quad s' \to (\mathbf{case}, s \to (x, \mathbf{0})), (c_1, \mathcal{D}[\![(\kappa \bullet l_1)\{x \mapsto l'_1\}]\!]\ \kappa\ \rho), \ldots, (c_n, \mathcal{D}[\![(\kappa \bullet l_n)\{x \mapsto l'_n\}]\!]\ \kappa\ \rho)$$
$$\quad \text{where } c_i \text{ is of arity } k,\ l'_i = \mathcal{L}[\![c_i\ 1 \ldots k]\!]\ s_i\ \emptyset\ \emptyset \text{ and } s_i \text{ is a new state}$$
$$\mathcal{D}_{\mathcal{R}}[\![s \to (x, \mathbf{0})]\!]\ \langle (s' \to (@, \bullet), (\#1, l)) : \kappa \rangle\ \rho = \mathcal{D}_{\mathcal{C}}[\![s \to (x, \mathbf{0})]\!]\ \langle (s' \to (@, \bullet), (\#1, l)) : \kappa \rangle\ \rho$$
$$\mathcal{D}_{\mathcal{R}}[\![s \to (c, \mathbf{0}), (\#1, l_1), \ldots, (\#n, l_n)]\!]\ \langle\rangle\ \rho = s \to (c, \mathbf{0}), (\#1, \mathcal{D}[\![l_1]\!]\ \langle\rangle\ \rho), \ldots, (\#n, \mathcal{D}[\![l_n]\!]\ \langle\rangle\ \rho)$$
$$\mathcal{D}_{\mathcal{R}}[\![s \to (c, \mathbf{0}), (\#1, l_1), \ldots, (\#n, l_n)]\!]\ \langle (s' \to (\mathbf{case}, \bullet), (c_1, l'_1), \ldots, (c_k, l'_k)) : \kappa \rangle\ \rho =$$
$$\quad \mathcal{D}[\![l'_i\{1 \mapsto l_1, \ldots, n \mapsto l_n\}]\!]\ \kappa\ \rho \text{ where } c = c_k$$
$$\mathcal{D}_{\mathcal{R}}[\![s \to (\lambda, l)]\!]\ \langle\rangle\ \rho = s \to (\lambda, \mathcal{D}[\![l]\!]\ \langle\rangle\ \rho)$$
$$\mathcal{D}_{\mathcal{R}}[\![s \to (\lambda, l)]\!]\ \langle (s' \to (@, \bullet), (\#1, l')) : \kappa \rangle\ \rho = \mathcal{D}[\![l\{1 \mapsto l'\}]\!]\ \kappa\ \rho$$
$$\mathcal{D}_{\mathcal{R}}[\![s \to (\mathbf{let}, l_0), (\#1, l_1), \ldots, (\#n, l_n)]\!]\ \kappa\ \rho = \mathcal{D}[\![l_0\{1 \mapsto l_1, \ldots, n \mapsto l_n\}]\!]\ \kappa\ \rho$$
$$\mathcal{D}_{\mathcal{R}}[\![s \to (@, l), (\#1, l')]\!]\ \kappa\ \rho = \mathcal{D}_{\mathcal{R}}[\![l]\!]\ \langle (s \to (@, \bullet), (\#1, l')) : \kappa \rangle\ \rho$$
$$\mathcal{D}_{\mathcal{R}}[\![s \to (\mathbf{case}, l_0), (c_1, l_1), \ldots, (c_n, l_n)]\!]\ \kappa\ \rho = \mathcal{D}_{\mathcal{R}}[\![l_0]\!]\ \langle (s' \to (\mathbf{case}, \bullet), (c_1, l_1), \ldots, (c_n, l_n)) : \kappa \rangle\ \rho$$

$$\mathcal{D}_{\mathcal{C}}[\![l]\!]\ \langle\rangle\ \rho = l$$
$$\mathcal{D}_{\mathcal{C}}[\![l]\!]\ \langle (s \to (@, \bullet), (\#1, l')) : \kappa \rangle\ \rho = \mathcal{D}_{\mathcal{C}}[\![s \to (@, l), (\#1, \mathcal{D}[\![l']\!]\ \langle\rangle\ \rho)]\!]\ \kappa\ \rho)$$
$$\mathcal{D}_{\mathcal{C}}[\![l]\!]\ \langle (s \to (\mathbf{case}, \bullet), (c_1, l_1), \ldots, (c_n, l_n)) : \kappa \rangle\ \rho =$$
$$\quad s \to (\mathbf{case}, l), (c_1, \mathcal{D}[\![l_1]\!]\ \kappa\ \rho), \ldots, (c_n, \mathcal{D}[\![l_n]\!]\ \kappa\ \rho)$$

Figure 2.2: Distillation Transformation Rules

performs driving to obtain a potentially infinite representation of an input program's behaviour, however, many of the sub-terms which are generalised in positive-supercompilation may actually be intermediate within the resulting LTS, but will not be further reduced by transformation. This is because generalisation takes place with respect to unevaluated terms which may contain intermediate terms.

In distillation, generalisation is performed in a similar manner to positive-supercompilation, but the resulting sub-terms are then transformed separately, before being re-combined and a second transformation pass applied. Generalisation in this second transformation pass takes place with respect to evaluated terms which will not contain intermediate terms, so over-generalisation will not occur.

Distillation also takes as its input the LTS representation of the original program and produces as its output a transformed LTS, from which a new (hopefully improved) program can be residualised (see Definition 5). The LTS resulting from the transformation of the LTS representation l of a program is given by $\mathcal{D}\ \langle\rangle\ \emptyset\ \emptyset$ where the rules $\mathcal{D}$ as shown in Fig. 2.2 are defined on a LTS and its surrounding context (denoted by κ) where the parameter ρ contains the LTS representations of memoized terms and θ contains terms previously extracted by generalisation.

Definition 5 (Extraction of Residual Program from LTS) A residual program can be constructed from a LTS l as $\mathcal{R}[\![l]\!]\ \emptyset$ using the rules $\mathcal{R}$ as shown in Fig. 2.3. If the current state of l is the target of a renaming transition, then it must correspond to a recursive function, so a new recursive function is defined along with a corresponding call. The parameter ρ contains the set of new function calls that have been created, and associates them with their corresponding state. On re-encountering one of these states, the corresponding function call is used.

Distillation also performs a normal-order reduction on the LTS representation of the input program. Previously encountered recursive terms are 'memoized' by adding them to ρ. The rules for unfolding/folding these recursive terms are given by $\mathcal{D}_{\mathcal{F}\infty}$. If the current term is a renaming of a memoized one, then folding is performed by creating a transition back to the previous state, with the renaming represented by a **let**. If the current term is an embedding of a memoized one, then generalisation is performed

$$\mathcal{R}[\![l]\!]\ \rho = \begin{cases} e, & \text{if } \exists (s,e) \in \rho.s = start(l) \\ f\ (x_1 \ldots x_n) & \\ \textbf{where} & \\ f = \lambda x_1 \ldots x_n.(\mathcal{R}'[\![l]\!]\ (\rho \cup \{(start(l), f\ x_1 \ldots x_n)\})), & \\ \quad \text{if } \exists s \in states(l), \alpha.s \xrightarrow{\alpha} start(l)\ (f \text{ is fresh}, \{x_1 \ldots x_n\} = fv(l)) & \\ \mathcal{R}'[\![l]\!]\ \rho, & \text{otherwise} \end{cases}$$

$$\begin{array}{lcl}
\mathcal{R}'[\![s \to (x, \mathbf{0})]\!]\ \rho & = & x \\
\mathcal{R}'[\![s \to (c, \mathbf{0}), (\#1, l_1), \ldots, (\#n, l_n)]\!]\ \rho & & \\
 & = & c\ (\mathcal{R}[\![l_1]\!]\ \rho) \ldots (\mathcal{R}[\![l_n]\!]\ \rho) \\
\mathcal{R}'[\![s \to (\lambda, l)]\!]\ \rho & = & \lambda x.\ (\mathcal{R}[\![l]\!]\ \rho) \\
 & & \text{where } x \text{ is fresh} \\
\mathcal{R}'[\![s \to (@, l_0), (\#1, l_1)]\!]\ \rho & = & (\mathcal{R}[\![l_0]\!]\ \rho)\ (\mathcal{R}[\![l_1]\!]\ \rho) \\
\mathcal{R}'[\![s \to (\mathbf{case}, l_0), (c_1, l_1), \ldots, (c_n, l_n)]\!]\ \rho & & \\
\quad = \mathbf{case}\ (\mathcal{R}[\![l_0]\!]\ \rho)\ \mathbf{of}\ p_1 \Rightarrow (\mathcal{R}[\![l_1]\!]\ \rho)\ |\cdots|\ p_n \Rightarrow (\mathcal{R}[\![l_n]\!]\ \rho) & & \\
\quad \text{where } c_i \text{ is of arity } k \text{ and } p_i = c_i\ x_1 \ldots x_k\ (x_1 \ldots x_k \text{ are fresh}) & & \\
\mathcal{R}'[\![s \to (\mathbf{let}, l_0), (\#1, l_1), \ldots, (\#n, l_n)]\!]\ \rho & & \\
\quad = \mathbf{let}\ x_1 = (\mathcal{R}[\![l_1]\!]\ \rho), \ldots, x_n = (\mathcal{R}[\![l_n]\!]\ \rho)\ \mathbf{in}\ (\mathcal{R}[\![l_0]\!]\ \rho) & & \\
\quad \text{where } x_1 \ldots x_n = bv(l_0) & & \\
\mathcal{R}'[\![s \to (\mathbf{let}, l), (\#1, s_1 \to x_1), \ldots, (\#n, s_n \to x_n)]\!]\ \rho & & \\
\quad = (\mathcal{R}[\![l]\!]\ \rho)\{1 \mapsto x_1, \ldots, n \mapsto x_n\} & &
\end{array}$$

Figure 2.3: Rules For Residualisation

and the separate components of the resulting LTS are then transformed separately, but then re-combined and a second unfold/fold pass $\mathcal{D}_{\mathcal{F}\in}$ is applied to the resulting term. In this second pass, if the current term is a renaming of a memoized one, then folding is also performed, but if the current term is an embedding of a memoized one, then generalisation is performed as in positive-supercompilation with the separate components of the resulting LTS being transformed separately. For both folding passes, if neither a renaming or embedding is encountered, the rules $\mathcal{D}_{\mathcal{R}}$ are applied to perform normal-order reduction on the LTS representation of the current term. The rules $\mathcal{D}_{\mathcal{C}}$ are applied when this normal-order reduction becomes 'stuck' as a result of encountering a variable in the redex position. In this case, the context surrounding the redex is further transformed. If the context surrounding a variable redex is a **case**, then information is propagated to each branch of the case to indicate that this variable has the value of the corresponding branch pattern.

As an example, consider the program shown below:

$$
\begin{array}{l}
app\ (arev\ xs\ ys)\ zs \\
\textbf{where} \\
app \quad = \quad \lambda xs.\ \lambda ys.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad [] \qquad\ \ \rightarrow ys \\
\qquad\qquad\qquad\qquad\quad (x : xs) \rightarrow (x : app\ xs\ ys) \\
\\
arev \quad = \quad \lambda xs.\ \lambda ys.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad [] \qquad\ \ \rightarrow ys \\
\qquad\qquad\qquad\qquad\quad (x : xs) \rightarrow arev\ xs\ (x : ys)
\end{array}
$$

Application of distillation to $app\ (arev\ xs\ ys)\ zs$ results in an equivalent program in which the use of intermediate data has been eliminated.

$$
\begin{array}{l}
f\ xs\ (g\ ys\ zs) \\
\textbf{where} \\
f \quad = \quad \lambda xs.\ \lambda v.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\quad [] \qquad\ \ \rightarrow v \\
\qquad\qquad\qquad\quad (x : xs) \rightarrow f\ xs\ (x : v) \\
\\
g \quad = \quad \lambda ys.\ \lambda zs.\ \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad\qquad\quad [] \qquad\ \ \rightarrow zs \\
\qquad\qquad\qquad\quad (y : ys) \rightarrow y : (g\ ys\ zs)
\end{array}
$$

Due to distillation, the intermediate list $arev\ xs\ ys$ used in the original function has been eliminated, however positive-supercompilation is not capable of such an optimisation. In positive-supercompilation, only the silent transitions within an LTS are removed; removing silent transitions can only produce a linear speedup in programs, as shown by Sørensen [76], as there will only be a constant number of silent transitions between each recursive call of a function. Distillation furthers this with its identification of extracted expressions resulting from generalisation, which, in combination with the removal of silent transitions, allows for its superlinear increases in efficiency [33].

Definition 6 (Distilled Form) The expressions resulting from distillation are in distilled form, $de^{\{\}}$, where within an expression of the form de^{ρ}, ρ denotes the set of all variables which have been introduced via **let** expressions, and cannot therefore appear as the selectors of **case** expressions. Distilled form, $de^{\{\}}$, is defined as shown in Figure 2.4.

Variables which have been introduced via the use of **let** expressions are added to the parameter ρ and cannot appear in the selectors of **case** expres-

$$
\begin{array}{lcll}
de^{\rho} & ::= & x & \\
 & | & c\ de_1^{\rho}\ \ldots\ de_n^{\rho} & \\
 & | & f & \\
 & | & de^{\rho}\ x & \\
 & | & \lambda x.de^{\rho} & \\
 & | & \textbf{case}\ x\ \textbf{of}\ p_1 \rightarrow de_1^{\rho}\ |\ldots|\ p_k \rightarrow de_k^{\rho} & \textbf{where}\ x \notin \rho \\
 & | & \textbf{let}\ x = de_0^{\rho}\ \textbf{in}\ de_1^{(\rho\ \cup\ \{x\})} & \\
 & | & f\ x_1\ \ldots\ x_n\ \textbf{where}\ f = \lambda x_1\ \ldots\ x_n.de^{\rho} &
\end{array}
$$

Figure 2.4: Distilled Form

sions. As a result of this, expressions in distilled form create no intermediate data structures.

Expressions resulting from distillation are in a specialised form known as *distilled form*, shown in Definition 6, which is identical to that of the **let**-normal form of the *hop* language, *hoplet*, as shown in Figure 1.4.

2.1.2 Calculational Methods Transformations

Alongside fold/unfold systems is another powerful program transformation technique: transformation via calculational methods. The calculation methods based transformations considered here are defined using the Bird-Meertens Formalisms (BMF) [5, 4, 50, 19, 3], a series of calculational laws which allow for concise descriptions of program transformations which preserve meaning and correctness [42].

At the core of many calculational program transformation techniques lies the idea of a *homomorphism*, a very general class of function that manipulates algebraic data [42]. *List-homomorphisms* are an important class of homomorphism that represent recursive functions defined over *cons*-lists and are formally defined as shown in Definition 7.

Definition 7 (List-Homomorphism) A function f is defined as a list-homomorphism if it matches the following definition, where $\oplus$ is an associative binary operator.

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

$(\!|e, \oplus|\!)_l$ denotes the unique function f [39, 42].

For example, the commonly used function *map*, which simply maps a function f across a list, can be represented by the list-homomorphism $(\!|[], \oplus|\!)_l$ **where** $x \oplus r = (f\ x) : r$. Many more complicated functions that operate over lists and are difficult to represent using a single list-homomorphism, can be represented using compositions of list-homomorphisms [42].

While list-homomorphisms are quite a general form, there exist a significant number of programs that cannot be described using list-homomorphisms and as a result there are many works based on extending the power of homomorphisms. One such extension is that of the *list-mutumorphism* which is an even more general form of function that encompasses all primitive recursive functions defined on lists, and are defined as shown in Definition 8.

Definition 8 (List-Mutumorphism) A function f_k is a list-mutumorphism with respect to the functions $f_1, \ldots, f_{k-1}, f_{k+1}, \ldots, f_n$ if each function, f_i where $i \in 1..n$ matches the following form, all e_i are given constants and all $\oplus_i$ are given binary functions:

$$\begin{aligned} f_i\ [] &= e_i \\ f_i\ (x : xs) &= x \oplus_i (f_1\ xs, \ldots, f_k\ xs, \ldots, f_n\ xs) \end{aligned}$$

$[\![(e_1, \ldots, e_k, \ldots, e_n), (\oplus_1, \ldots, \oplus_k, \ldots, \oplus_n)]\!]_l$ denotes the list-mutumorphism f_k [42].

List-mutumorphisms are nested loops which can be reduced to a near-homomorphism [17] which contains a single loop via the *flattening* transformation [38, 42]. Near-homomorphisms are functions that when composed with an auxiliary projection function, π, become homomorphic and are defined as shown in Definition 9. A projection function, π_i allows the i-th element of a tuple to be extracted, for example: $\pi_3\ (a, b, c) = c$.

Definition 9 (Near-Homomorphism) Given a function, g, a list-homomorphism, f, and a projection function, π, a near-homomorphism for g is defined as:

$$g \equiv \pi \circ f$$

The flattening transformation can be performed by combining a projection function, π_k, which returns the k-th element of a tuple, and a list-homomorphism according to the rule shown in Definition 10.

Definition 10 (Flattening Rule) A list-mutumorphism, f_k, may be flattened according to the following rule:

$$\begin{aligned}&[\![(e_1,\ldots,e_k,\ldots,e_n),(\oplus_1,\ldots,\oplus_k,\ldots,\oplus_n)]\!]_l = \pi_k \circ (\!|e,\oplus|\!)_l\\&\textbf{where } x \oplus y = (x \oplus_1 y,\ldots,x \oplus_k y,\ldots,x \oplus_n y)\\&\qquad e = (e_1,\ldots,e_k,\ldots,e_n)\end{aligned}$$

As an example of the application of the flattening transformation, consider the mutumorphic definition of the function *biggers* shown below, which calculates the set of 'bigger' numbers in a given list, where a bigger number is defined as one that is larger than the sum of all numbers following it in the given list.

$$biggers = [\![([],0),(\oplus_1,\oplus_2)]\!]_l$$

According to the flattening rule, this can be flattened into an equivalent single homomorphism, which only has a single loop, and is defined as follows:

$$\begin{aligned}&biggers = fst \circ (\!|([],0),\oplus|\!)_l\\&\textbf{where } \oplus = \lambda a.\ \lambda(r,s).\ \textbf{case } a \geq s \textbf{ of}\\&\qquad\qquad\qquad\qquad True\ \rightarrow a : r\\&\qquad\qquad\qquad\qquad False \rightarrow (r, a+s)\end{aligned}$$

Another technique that allows more programs to be expressed as list-homomorphisms is that of *tupling*, which, like the tupling strategy of Proietti & Pettorossi [64], is aimed at eliminating multiple accesses of the same data. As an example, consider two homomorphisms, $(\!|e_1,\oplus_1|\!)_l$ and $(\!|e_2,\oplus_2|\!)_l$, that are both applied to the list x. In this case, it is possible to tuple both $(\!|e_1,\oplus_1|\!)_l$ and $(\!|e_2,\oplus_2|\!)_l$ together into one homomorphism [42], according to the rule shown in Definition 11. Tupling these two homomorphisms together will eliminate the multiple accesses to x [38], however it is worth noting that like the tupling strategy of Proietti & Pettorossi, this approach can also be prone to space leaks [75].

Definition 11 (Tupling Rule) Where two list-homomorphisms access the same data tupling them together into one list-homomorphism, according to the following rule, can make them more efficient.

$$\begin{aligned}&((\!|e_1,\oplus_1|\!)_l\ x, (\!|e_2,\oplus_2|\!)_l\ x) = (\!|(e_1,e_2),\oplus|\!)_l\ x\\&\textbf{where } x \oplus (y_1,y_2) = (x \oplus_1 y_1, x \oplus_2 y_2)\end{aligned}$$

For example, consider the function average shown below:

$$average = \lambda x.sum\ x\ 'div'\ length\ x$$

This may be re-written using homomorphisms as the following:

$$average = \lambda x.(\!|0, +|\!)_l\ x)\ 'div'\ (\!|0, \lambda x.\lambda y.1 + y|\!)_l\ x)$$

Using the tupling rule to eliminate multiple accesses to x, this can then be redefined as:

$$\begin{aligned} average = \lambda x.\ &\textbf{case}\ tup\ x\ \textbf{of} \\ &\quad Pair\ s\ l \rightarrow s\ 'div'\ l \\ \textbf{where}\ tup = &(\!|(0,0), \lambda a.\ \lambda(s,l).\ (a+s, 1+l)|\!)_l \end{aligned}$$

Both of these techniques, *flattening* and *tupling* when combined with another technique called *shortcut-fusion* form the core of a technique known as *loop-fusion*. Shortcut-fusion is the calculational equivalent of the fold/unfold transformation deforestation [88, 22], and simply states that if a function can be defined in terms of the function *build*, shown in Definition 12, then it can easily be fused into a list-homomorphism from its right [42], as shown in Definition 13. However, this requires a 'warm-up' rule, shown in Definition 14, which allows *build* to be derived.

Definition 12 (Build)

$$build\ g = g([], (:))$$

Definition 13 (Shortcut-Fusion) If an expression can be defined in terms of *build*, then it can be fused into a homomorphism easily.

$$(\!|e, \oplus|\!)_l \circ\ build\ g = g(e, \oplus)$$

Definition 14 (Warm Up)

$$(\!|e, \oplus|\!)_l = build\ (\lambda(d, \otimes).(\!|d, \otimes|\!)_l \circ (\!|e, \oplus|\!)_l)$$

As an example of the application of shortcut-fusion, consider the function $sumBig$ shown below. This uses the version of $biggers$ resulting from the flattening transformation presented earlier to calculate the sum of all bigger numbers in a given list.

$$sumBig = (\!|0, +|\!)_l \circ biggers$$

Applying warm-up to the definition of $biggers$ results in the following definition, defined in terms of $build$:

$$\begin{aligned} biggers &= build(\lambda(d, \otimes).\ fst \circ (\!|(d, 0), \oplus'|\!)_l) \\ \textbf{where } \oplus' &= \lambda a.\ \lambda(r, s).\ \textbf{case } a \geq s \textbf{ of} \\ &\qquad\qquad True\ \ \rightarrow a \otimes r \\ &\qquad\qquad False \rightarrow (r, a + s) \end{aligned}$$

Application of the shortcut-fusion rule to the resulting definition of $sumBig$ results in the following homomorphic definition of $sumBig$:

$$\begin{aligned} sumBig &= fst \circ (\!|(0, 0), \otimes|\!)_l \\ \textbf{where } \otimes &= \lambda a.\ \lambda(r, s).\ \textbf{case } a \geq s \textbf{ of} \\ &\qquad\qquad True\ \ \rightarrow a + r \\ &\qquad\qquad False \rightarrow (r, a + s) \end{aligned}$$

Using the above calculational rules loop fusion may be performed by performing the five following steps:

1. Represent as many recursive list functions as possible using list-mutumorphisms.

2. Apply flattening to remove nested loops, converting the list-mutumorphisms to list-homomorphisms.

3. Use promotion rules [42] and shortcut-fusion to remove dependent non-nested loops.

4. Use tupling to gather homomorphisms, eliminating multiple accesses to the same data.

5. Embed resulting homomorphisms in the output program.

While these calculational transformation techniques are powerful, and may help developers to derive more efficient programs, they are not without their pitfalls. Tupling and loop-fusion as a result (as it depends on tupling) are susceptible to space leaks if any tupled values are not used, and shortcut-fusion, like deforestation is only capable of obtaining a linear increase in efficiency. While list-homomorphisms, near-homomorphisms and list-mutumorphisms are quite general forms for defining functions, they are still restrictive and it is unrealistic to expect developers to have to define their programs in these forms [77]. Programs defined outside of these restrictive forms would need to be transformed form to use these techniques.

2.2 Parallelisation of Functional Programs

2.2.1 Glasgow Parallel Haskell

One approach to the explicit parallelisation of functional programs is that of Glasgow parallel Haskell (GpH) [80] 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 enforced by applying the **pseq** strategy. As an example, the expression $x \text{ `}par\text{`} y$ **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 $x \text{ `}par\text{`} y$, 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 [48]. *pseq* is used to control evaluation order as $x \text{ }'pseq'\text{ } y$ will force the evaluation of x before y. Usually, this is used because y cannot or should not be evaluated until x has been. The *rdeepseq* strategy can be used to fully evaluate an expression where the expression $(rdeepseq\ x) \text{ `}par\text{`} (rdeepseq\ y)$ will spark the full evaluation of x in parallel with the full evaluation of y.

As an example, the expression $x \text{ `}par\text{`} (y \text{ }'pseq'\ x{+}y)$ sparks the evaluation of x in parallel with the 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 $x \text{ `}par\text{`} (y \text{ }'pseq'\ x{+}y)$ 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:

$$
\begin{aligned}
fib = \lambda x.\ &\textbf{case}\ x\ \textbf{of} \\
&0 \rightarrow 1 \\
&1 \rightarrow 1 \\
&n \rightarrow\ \textbf{let}\ x = \mathit{fib}\ (n-1) \\
&\qquad\ \textbf{in}\ \ \textbf{let}\ y = \mathit{fib}\ (n-2) \\
&\qquad\qquad\ \textbf{in}\ x\ `\mathit{par}`\ (y\ `\mathit{pseq}`\ x + y)
\end{aligned}
$$

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 evaluated, it is then added to the result of the evaluation of $fib\ (n-1)$. $fib\ (n-1)$ can be fully evaluated to *normal form* in parallel by having the *rdeepseq* strategy applied to it, however this is the same as WHNF for integers.

2.2.1.1 Paraforming

Paraforming [9] is a parallelisation technique that uses a refactoring tool to assist the developer in writing efficient parallel programs using GpH. Paraforming allows for the introduction of data and task parallelism, with two main refactorings: *Introduce Data Parallelism* and *Introduce Task Parallelism*, with other refactorings to increase the granularity of parallelism in each.

Data Parallelism is enabled via the *Introduce Data Parallelism* refactoring which simply applies the *parList* strategy to a list expression. The granularity of parallelism within this can be controlled via the *Introduce Clustering* refactoring, which takes a *parList* and a chunk size and applies a *parListChunk* strategy instead. Using the *Introduce Data Parallelism* refactoring requires that the list be fully evaluated initially.

The granularity of parallelism in both of these refactorings can be controlled via further refactorings: *Introduce Thresholding* simply allows the developer to disable the parallelism once the granularity of parallelism falls below a certain point. The *Modify Evaluation Degree* refactoring can be used to control the degree to which the parallelised expressions are evaluated through the use of the *rdeepseq* strategy, removing the need for expressions to be evaluated sequentially on demand.

As an example, if we consider a function, $sumMap$, that maps a function across a list, and sums the result of this:

$$sumMap = \lambda f\ xs.sum\ (map\ f\ xs)$$

By clustering this calculation, using the $parListChunk$ strategy, we can parallelise this function, where c represents the chunk size:

$$smData = \lambda f\ xs\ c.sum\ (map\ f\ xs\ `using`\ parListChunk\ c\ rdeepseq)$$

Task Parallelism is enabled via the *Introduce Task Parallelism* refactoring, which sparks the parallel evaluation of a given computation within a **let**. There are two such refactorings: the first simply takes a computation, x, as an argument and locates the definition of x. Once this has been located a **let** statement is introduced which sparks the parallel evaluation of x. Following this, any references to x are replaced by the result of the parallel evaluation of x. The second refactoring takes a computation, x, and a set of existing parallel sparks, s. Again the definition of x is located, but its evaluation is sparked in parallel with that of the existing sparks in s and all further references to x are replaced with the result of its parallel evaluation.

To add task parallelism to the definition of $sumMap$, we can convert it to a *divide-and-conquer* version:

$$\begin{aligned}
sumMap = \lambda f\ xs.\ &\mathbf{case}\ xs\ \mathbf{of}\\
&[]\ \ \rightarrow 0\\
&[x] \rightarrow f\ x\\
&xs\ \rightarrow\ \mathbf{let}\ (l,r) = splitAt\ (length\ xs\ `div`\ 2)\ xs\\
&\qquad\ \mathbf{in}\ \ \mathbf{let}\ s1 = sumMap\ f\ l\\
&\qquad\qquad\ \mathbf{in}\ \ \mathbf{let}\ s2 = sumMap\ f\ r\\
&\qquad\qquad\qquad\ \mathbf{in}\ s1 + s2
\end{aligned}$$

The *Introduce Task Parallelism* refactoring is then used to introduce parallelism for the variable $s1$. As a result the evaluation of $s1$ is therefore sparked in parallel, following which the same refactoring is also used to introduce parallelism for the variable $s2$, whose evaluation is sparked in parallel with that of $s1$.

```
smTask = λf xs. case xs of
                  []   → 0
                  [x] → f x
                  xs  → let (l, r)= splitAt (length xs `div` 2) xs
                        in let s1 = sumMap f l
                           in let s2 = sumMap f r
                              in s1 `par` (s2 `pseq` s1 + s2)
```

2.2.2 Data Parallel Haskell

Data Parallel Haskell (DPH) [45] provides nested data-parallelism, inspired by the NESL language [8]. Conventional parallel languages require input arrays to be flat, or one-dimensional, but NESL and DPH allow, for example, arrays of arrays to be parallelised and for each of these sub-arrays to be processed in parallel [12]. Though the program being parallelised is defined in terms of these *nested* arrays, these are later flattened to ensure that the parallel program itself uses flat arrays, and *fusion* [22] is also performed to remove the intermediate data that can be problematic when using nested data-parallelism.

Support for nested arrays is added to DPH via a new array type, denoted [: t :]. As an example, the *sumList* program can be defined using Data Parallel Haskell, in a divide-and-conquer fashion as follows:

```
sumList xs
where
sumList = λxs. case (length xs) of
                 0 → 0
                 1 → xs !! 0
                 n → let lth = (lengthP xs) `div` 2
                     in let (l, r) = splitP xs lth
                        in let sl = [: sumList x | x ← [: l, r :] :]
                           in (sl !! 0) + (sl !! 1)
```

Given a list, xs, the function *sumList* returns the sum of the numbers contained in that list. If xs is empty the result is 0 and if xs has only one element then the result is that element, which can be extracted from the list using the operator !!, which given a list ys and an index i returns the element contained in ys at index i. If xs has more than one element, *sumList* splits the list in half recursively calculating the sum of each half of the list, the results of which are stored in a nested array. Once the sums of each half

of the list has been calculated, they can be extracted from the nested array and added together.

This can be efficiently and automatically parallelised. One benefit of using DPH is that, although the parallelism is explicit, the developer need only be concerned with representing the solution using a divide-and-conquer approach. DPH is certainly a powerful approach, and nested data parallelism is an interesting parallelisation technique. One problem associated with this approach though is that as the nested arrays are strict [12], referencing one element from such an array means that the whole array must be evaluated, and this may not always be necessary. This is a big difference to the standard Haskell lists, which are evaluated element-wise by need.

Another drawback to using DPH is that it requires careful separation of code that may be 'vectorised' - conversion of nested data into flat data - and code that cannot be vectorised into different modules. GHC - the Haskell compiler - can only either vectorise all or none of the code in a module, and vectorisation is only applicable to pure Haskell functions [11].

2.2.3 Skeletal Programming

Skeletons, due to Cole [16], are well recognised common patterns of parallel programming that can be efficiently mapped to parallel architectures without involving the developer of a program that uses them. These primitive functions can be used to assist the development of parallel versions of sequential programs, and may be viewed as the building blocks of a parallel program from which all parallelism is obtained [18]. These building blocks should be viewed as higher-order functions, each of which have a known efficient parallel implementation. The skeletons *map* and *reduce* are shown in Figure 2.5 and are very important skeletons which are used in a number of parallelisation techniques.

Given a function f, and a list xs, *map* applies f to each element of xs. *map* has a very obvious parallel implementation as each application of f is independent of all others and can be computed in parallel.

Given a binary function g, and a list xs *reduce* reduces the list to a single value using g. While the parallel implementation of *reduce* is not as obvious as that of *map*, as long as g is associative, *reduce* can be efficiently evaluated in parallel using a *divide-and-conquer* strategy by recursively splitting xs in half and applying *reduce* to each half of the list.

$$
\begin{array}{lcl}
map & = & \lambda f.\ \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \qquad [] \qquad\ \rightarrow [] \\
 & & \qquad (x : xs) \rightarrow f\ x : map\ f\ xs \\
 & & \\
reduce & = & \lambda g.\ \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \qquad [] \qquad\ \rightarrow [] \\
 & & \qquad (x : xs) \rightarrow g\ x\ (reduce\ g\ xs)
\end{array}
$$

Figure 2.5: Example skeletons

Within skeletal programming, given a set of known skeletons, the developer of a program uses at least one of these to implement their parallel program and an efficient parallel implementation can be derived from the use of the selected skeletons. The use of skeletal programming is beneficial as the skeletons show how the program can be decomposed into parallel parts and describe the strategy for parallelising the program. However, one problem with skeletons is that as parallel architectures differ, so do their underlying implementations. This means that as architectures change the skeletons themselves may need new implementations, however the developer need not be aware of these changes.

Despite the advantages of the skeletal approach, developing efficient parallel programs still remains a big challenge for two reasons. Firstly, as is pointed out in [71, 54], it is hard to choose appropriate parallel skeletons for a particular problem, especially when the given problem is complicated. This is due to the gap between the simplicity of parallel skeletons and the complexity of the algorithms to be parallelised.

Secondly, as is pointed out in [55], although a single skeleton can be efficiently implemented, a combination of skeletons may not execute efficiently, as parallel programs defined in terms of multiple skeletons tend to introduce intermediate data structures which are used for communication between skeletons. In order to solve this problem, further program transformation may need to be applied to the skeletal program to eliminate the use of intermediate data-structures.

Another issue with using skeletons is that a given skeleton may not work well on a given architecture. Darlington et al. [18] present a technique by which a program defined in terms of a skeleton that may not work well on a given architecture can be transformed into another skeleton that does.

However, the presented technique is simply an overview of such a system and presents no concrete transformation process.

Matsuzaki et al. [55] present a framework for designing and implementing skeletal programming libraries, that uses a *shortcut-deforestation* [22] transformation technique to eliminate the use of intermediate data between skeletons defined on *join*-lists. Using information about how each skeleton produces and consumes data and *fusion* allows for efficient combinations of skeletons to be used.

Using the associative operators $\oplus$ and $\otimes$, and the functions g, p and q, Matsuzaki et al.'s approach is to define skeletons in terms of three functions, *acc*, denoted $[\![g, (p, \oplus), (q, \otimes)]\!]$, *cataJ*, denoted $(\!|\oplus, p, e|\!)$, and *buildJ*, as shown in Figure 2.6.

$$
\begin{array}{lcl}
[\![g, (p, \oplus), (q, \otimes)]\!]\ []\ e & = & g\ e \\
[\![g, (p, \oplus), (q, \otimes)]\!]\ (a : x)\ e & = & p\ (a, e) \oplus [\![g, (p, \oplus), (q, \otimes)]\!]\ x\ (e \otimes q\ a) \\
 & & \\
(\!|\oplus, p, e|\!)\ [] & = & e \\
(\!|\oplus, p, e|\!)\ (a : x) & = & p\ a \oplus (\!|\oplus, p, e|\!)\ x \\
 & & \\
buildJ\ gen & = & gen\ (+\!\!+)\ [\cdot]\ []
\end{array}
$$

Figure 2.6: *acc*, *cataJ* and *buildJ* Definitions

By restricting the definition of skeletons to the forms shown in Figure 2.6, Matsuzaki et al. define fusion rules for the different combinations of these forms. By applying fusion to these combinations, the use of intermediate data structures between combinations of skeletons can be eliminated, resulting in an efficient parallel implementation.

While this technique for defining skeletons is applicable to a broad class of programs it is a manual technique, and an automated technique would be more desirable. It is also restrictive as the operators $\oplus$ and $\otimes$ must be known to be associative and it requires that skeletons be defined in a restrictive form, as shown in Figure 2.6. The skeletons are defined across *join*-lists and require that their inputs be partitioned, yet no methodology is shown to generate these *join*-lists.

González-Vélez et al. [23] presents a thorough review of existing skeletal parallel programming frameworks that can be used to enable developers to make use of high-level parallel programming techniques, in which the au-

thors make the observation that skeletons may be broken down into three categories: data-parallel, task-parallel and resolution. Data-parallel skeletons may be used to parallelise bulk data and are defined by the data they evaluate e.g. the *map* skeleton, task parallel skeletons may be used to parallelise tasks and are defined according to the interactions between these tasks, e.g. pipelined parallelisation of tasks and lastly, resolution skeletons can be used to solve a family of problems and are defined by the solution to that family of problems, such as the divide-and-conquer skeleton.

In addition to this, González-Vélez et al. further classify and group skeletal techniques according to the following four categories: *co-ordination* in which a high-level language is used to describe and co-ordinate the algorithm, and a host language is used to interact with the environment. Many examples, such as Single Assignment C [27] enhance an existing language with a means to describe skeletal behaviour at a high level, which can then be translated into the host language. The *functional* category of skeletal frameworks have been enabled through the use of both language extensions, such as the Eden Haskell extension [49], and have also been defined as part of an existing language, such as the Concurrent Clean Haskell library [35]. Object-oriented languages can have skeletal frameworks embedded through the use of classes and objects which enable parallelism, such as the SkeTo project [54]. Imperative skeletal frameworks are the final classification, which can be added to a procedural language through the use of APIs which expose parallel skeletal behaviour to the developer, such as the ASSIST programming model [84].

2.3 Program Parallelisation by Transformation

2.3.1 Parallelisation via Fold/Unfold Methods

Fold/unfold transformations are not only useful when applied in a sequential context, they can also be used to automate the parallelisation process. Chin et al. [78] present a semi-automatic fold/unfold based approach that converts programs defined on *cons*-lists into parallel versions defined on *join*-lists.

Chin et al.'s technique is quite powerful as it can handle a broad range of programs, including those that can be difficult to parallelise [78], such as programs with accumulating parameters, non-linear recursion and conditional constructs. At the core of the algorithm are three steps: (1) The

derivation of a pair of equations in a sequential *pre-parallel* form from the initial function. (2) This pair of pre-parallel functions are then generalised, the results of which may contain undefined auxiliary functions. (3) The final step is to then define these auxiliary functions.

The result of the first step should be two pre-parallel functions in which all function calls on both sides are associative or distributive and the depth of recursive variables from the root of both sides should be as shallow as possible. As an example, consider the simple *sum* function, which calculates the sum of the numbers contained in a *cons*-list:

$$\begin{array}{ll} sum\ [] & = 0 \\ sum\ (x : xs) & = x + (sum\ xs) \end{array}$$

Two acceptable pre-parallel functions for *sum* are:

$$\begin{array}{lll} sum\ ([x] +\!\!+ xs) & = x + (sum\ xs) & (\mathbf{1}) \\ sum\ (([x] +\!\!+ [y]) +\!\!+ xs) & = x + y + (sum\ xs) & (\mathbf{2}) \end{array}$$

After deriving two sequential pre-parallel functions, a second-order generalisation is applied, the result of which may contain calls to undefined auxiliary functions. If there are undefined auxiliary functions, an inductive derivation step is then applied to provide definitions for these functions. As part of the second order generalisation step, upon examination of the left-hand-side of both equations, $sum\ ([x] +\!\!+ xs)$ and $sum(([x] +\!\!+ [y]) +\!\!+ xs)$ respectively, a conflict between both is detected, where the first argument mismatches in both, being $[x]$ for the first and $[x] +\!\!+ [y]$ for the second. Such a difference can be generalised in both expressions by simply replacing the two conflicting sub-terms with a new variable ys.

After resolving the conflicts between the left-hand-sides of both expressions, their right-hand-sides must also be examined for conflicting contexts. Another conflict between both is detected in the sub-term before the recursive call to *sum*, being x in the first and $x + y$ in the second. This conflict can be resolved by generalising the conflicting terms and introducing a new function, while preserving the context of the original function. The result of performing generalisation on the functions (**1**) and (**2**) can be seen below:

$$sum'\ (ys +\!\!+ xs) \quad = (h\ ys) + (sum\ xs)$$

The result of this generalisation contains a call ($h\ ys$) to an undefined auxiliary function, which can be defined via inductive derivation, resulting in the following definition of h:

$$\begin{array}{ll} h\ [] & = 0 \\ h\ (x : xs) & = x + (h\ xs) \end{array}$$

In cases where auxiliary functions such as h are introduced, it may be necessary to apply the parallelisation algorithm to these functions. In the case of the above example, h is the same as the initial definition of sum and the call to h can therefore just be replaced by an appropriate call to sum', as can the remaining call to sum:

$$sum'\ (ys +\!\!+\ xs) \quad = (sum'\ ys) + (sum'\ xs)$$

It should be obvious from this definition, that provided there are no data dependencies between ys and xs, then the evaluation of both $sumList\ ys$ and $sumList\ xs$ can be completed in parallel. While such an approach is indeed powerful and does allow complex functions such 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. As such, the technique is *semi-automatic* and a fully automatic technique would be more desirable.

While [78] presents an informal overview of the technique, a more concrete version was specified by Chin. et al. [13, 15]. However, these more formal versions are still only semi-automatic and are defined for a first-order language and require that the associativity of operators be specified. They also produce functions defined on $join$-lists but do not specify a means to convert the original $cons$-list into an appropriate $join$-list.

The techniques are applied to *list-paramorphisms* [56] and while these encompass 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 parallelised.

2.3.2 Parallelisation via Calculational Methods

Like fold/unfold based transformations, calculational methods based transformations are also useful when parallelising sequential programs. At the core of these parallelising transformations lies the notion of a homomorphism defined on a *join*-list (*join-homomorphism*).

2.3.2.1 Join-Homomorphisms

The use of join-homomorphisms in the parallelisation process was first suggested by Skillicorn [69].

Definition 15 (Join-Homomorphism) A function f is defined as a join-homomorphism if it matches the following definition, where $\oplus$ is an associative binary operator.

$$\begin{aligned} f\ [x] &= g\ x \\ f\ (xs +\!\!+ ys) &= (f\ xs) \oplus (f\ ys) \end{aligned}$$

$(\!|g, \oplus|\!)_j$ denotes the unique function f [39, 42].

It should be obvious from this definition that the computations of $(f\ xs)$ and $(f\ ys)$ are independent and may be completed in parallel [25]. According to the join-homomorphism lemma, shown in Lemma 1, any join-homomorphism may be defined as a composition of a *map* and a *reduce*, both of which are join-homomorphisms themselves [5] and have known highly efficient parallel implementations [7, 70].

Lemma 1 (Join-Homomorphism) *Any function f is a join-homomorphism, with respect to $(+\!\!+)$, iff, for some function g, and a binary associative operator $\oplus$, the following holds:*

$$f = (\!|g, \oplus|\!)_j = (reduce\ \oplus) \circ (map\ g)$$

As join-homomorphisms are quite general and are definable in terms of the skeletons *map* and *reduce*, they have a large amount of potential parallelism built in. The methodology presented by Skillicorn is quite restrictive, as while it does provide numerous functions, these are all defined in terms of *map* and *reduce*, and as with list-homomorphisms, there are a significant number of programs that cannot be expressed using join-homomorphisms [36]. As a result of this there is also a large amount of work on extending the power of join-homomorphisms [24, 25, 26, 39, 40, 44].

2.3.2.2 Distributable-Homomorphisms

Gorlatch [25] attempts to expand the class of parallelisable functions by introducing a new type of homomorphism, *distributable-homomorphisms* and an efficient and correct parallel schema for implementing these functions.

Definition 16 (Distributable Homomorphism) *A function f is a distributable-homomorphism, for given associative operators $\oplus$ and $\otimes$, if its definition matches that of dist below:*

$$\begin{aligned} dist\ \oplus\ \otimes\ [x] &= [x] \\ dist\ \oplus\ \otimes\ (xs +\!\!+ ys) &= combine\ \oplus\ \otimes\ (dist\ \oplus\ \otimes\ xs)\ (dist\ \oplus\ \otimes\ ys) \\ \\ combine\ \oplus\ \otimes\ xs\ ys &= (zipWith\ \oplus\ xs\ ys) +\!\!+ (zipWith\ \otimes\ xs\ ys) \end{aligned}$$

Distributable-homomorphisms, defined in Definition 16, attempt to solve two of the main problems with using join-homomorphisms to guide the parallelisation process: that of finding a suitable $\oplus$, the combination operator for the join-homomorphism, and efficiently implementing the reduction part in parallel [25]. However, the form of distributable-homomorphisms is quite restrictive and it is unrealistic to force developers to define their functions in such a form. The associativity of both $\oplus$ and $\otimes$ also needs to be specified by the developer.

2.3.2.3 Third Homomorphism Theorem

Another method for deriving join-homomorphisms was also presented by Gorlatch [24, 26] which builds on the work of Gibbons on the third homomorphism theorem [21].

Definition 17 (Leftwards & Rightwards Functions) The function f is leftward with respect to $\oplus$ iff for all elements x and all lists xs the following holds:

$$f\ ([x] +\!\!+ xs) = x \oplus f\ xs$$

The function g is rightward with respect to $\otimes$ iff for all lists xs and all elements x the following holds:

$$g\ (xs +\!\!+ [x]) = g\ xs \otimes x$$

The third homomorphism theorem states that if a function is both leftward and rightward, as defined in Definition 17, then it is a join-homomorphism. Gorlatch builds upon this theorem by generating join-homomorphisms via generalisation of both leftward and rightward functions, that for some non-homomorphic functions can provide embedding into join-homomorphisms. By generalising both a leftward and rightward (*cons*-list and *snoc*-list) version of a function, into another expression that defines a combination operation $\ominus$ which is associative, then the original function is a join-homomorphism, and its combination operation is $\ominus$. While this technique is certainly powerful, requiring that a user define two versions of their program in the restrictive leftward and rightward forms is quite a pitfall.

2.3.2.4 Join-Mutumorphisms

Hu. et al. [39] further previous work by presenting a powerful transformation technique which is applicable to a broad class of primitive recursive functions and is more powerful than previous work [69, 25, 26]. This parallelisation technique makes use of tupling [36, 38] and join-mutumorphisms, defined in Definition 18, as part of the parallelisation process.

Definition 18 (Join-Mutumorphism) The functions $mutu_1, \ldots, mutu_n$ are join-mutumorphisms if they are mutually defined in the following form:

$$
\begin{aligned}
&mutu_j\ [a] &&= k_j\ a \\
&mutu_j\ (xs +\!\!+ ys) &&= ((\Delta_1^n\ mutu_j)\ xs) \oplus ((\Delta_1^n\ mutu_j)\ ys) \\
&\textbf{where} \\
&\Delta_1^n\ mutu_i = (\!|\Delta_1^n k_i, \Delta_1^n \oplus_i |\!)_j
\end{aligned}
$$

There are three steps to the parallelisation algorithm in [39], the first of which is to make the associative operators explicit [68]. Then the body of the function is normalised via abstraction and fusion. The final step is application of the presented parallelisation theorem and optimisation via the tupling calculation [36, 38] to eliminate multiple evaluation of data.

The technique presented in [39] is quite a powerful technique, as it applies to a very general form of recursive program. It can handle a broader class of program than previous work, such as non-linear recursion, accumulating parameters and conditional statements. Similar to previous techniques, it does require that input programs are defined in a restricted form.

2.3.2.5 Diffusion

Hu et al. [40] present a technique, which makes use of the algorithm presented in [39], called *diffusion* which generalises the join-homomorphism lemma to allow for accumulating parameters. As a result diffusion is applicable to quite a broad class of recursive functions, and when combined with the normalisation technique in [39] allows for an even broader class of program.

At the core of this technique lies the *diffusion theorem* which is applicable to three classes of program: those that can be simply *diffused* to *reduce*, those that can be diffused to a combination of *map* and *reduce*, and those that can be diffused to a combination of *map*, *reduce* and *scan*. This latter class of program is the most interesting, as the others have been dealt with before. This type of diffusion focuses on functions with an accumulating parameter that match the following form:

$$\begin{array}{ll} f\ []\ c & = g_1\ c \\ f\ (x : xs)\ c & = k\ (x, c)\ \oplus\ (f\ xs\ (c\ \otimes\ g_2\ x)) \end{array}$$

By precomputing all values of the accumulating parameter, Hu et al. can eliminate the problems associated with it, leading to the diffusion theorem:

Definition 19 (Diffusion Theorem) *Using the above recursive form, and the fact that $\oplus$ and $\otimes$ are associative and have units, a recursive function, f, is diffusible into the following:*

$$\begin{array}{lll} f\ x\ c = & \textbf{let} & cs' +\!\!+ [c'] = map\ (c\ \otimes)\ (scan\ \otimes\ (map\ g_2\ x)) \\ & & ac = zip\ x\ cs' \\ & \textbf{in} & (reduce\ \oplus\ (map\ k\ ac)) \oplus (g_1\ c') \end{array}$$

Using the diffusion theorem and the recursive function definition above, Hu et al. define the diffusion parallelisation algorithm as:

1. Linearize recursive calls.
2. Identify associative operators.
3. Apply the diffusion theorem.
4. Optimise operators.

While their recursive definition applies to quite a broad class of program, a lot of functions that can be diffused do not match this form initially and must be transformed into a diffusible form. As with the technique presented in [39], there is a need to derive associative operators.

2.3.2.6 The *accumulate* Skeleton

Making use of the diffusion technique, Iwasaki et al. [44] present the *accumulate* skeleton, which is applicable to functions that are diffusible and has a known efficient parallel implementation. The accumulate skeleton aims to solve some of the main problems associated with working with skeletons: the selection of an appropriate skeleton for a given problem and the combination of skeletons in an efficient manner [71, 55, 54].

As the *accumulate* skeleton is based on the diffusion theorem, it is applicable to the same broad class of programs that are diffusible, and is more descriptive than existing skeletons such as *scan*. The implementation of the *accumulate* skeleton, defined in Definition 20, uses *fusion* [22, 42] to eliminate intermediate data and multiple accesses to data and as a result can decrease communication across processors.

Definition 20 (Accumulate Skeleton) *Where g, p, q are functions and $\oplus$ and $\otimes$ are associative operators, the* accumulate *skeleton, denoted $[\![g, (p, \oplus), (q, \otimes)]\!]$, is defined as:*

$$\begin{aligned} &accumulate\ []\ c &&= g\ c \\ &accumulate\ (x : xs)\ c &&= p\ (x, c) \oplus\ accumulate\ xs\ (c \otimes q\ x) \end{aligned}$$

As a result of diffusion, this can be written in terms of reduce, map, scan and zip as follows:

$$\begin{aligned} [\![g, (p, \oplus), (q, \otimes)]\!]\ xs\ c\ = \mathbf{let}\ &bs +\!\!+ [b] = map\ (c \otimes)\ (scan\ \otimes\ (map\ q\ xs)) \\ &as\ \ \ \ \ \ \ \ = zip\ xs\ bs \\ \mathbf{in}\ &reduce\ \oplus\ (map\ p\ as)\ \oplus g\ b \end{aligned}$$

The *accumulate* skeleton abstracts the use of a number of lesser skeletons: *map*, *reduce*, *scan* and *zip*, with no need for a developer to be aware of how to combine these skeletons, the developer only needs to provide g, p, $\oplus$, q and $\otimes$. As an example of the use of this skeleton consider a function that eliminates the smaller elements of a list, *smaller*, defined below:

$$
\begin{array}{ll}
smaller\ []\ c & = [] \\
smaller\ (x : xs)\ c & = \textbf{case}\ x < c\ \textbf{of} \\
 & \quad True\ \rightarrow smaller\ xs\ c \\
 & \quad False \rightarrow [x] +\!\!+ smaller\ xs\ x
\end{array}
$$

The result of applying diffusion to *smaller* is shown below:

$$
\begin{array}{lll}
smaller\ xs\ c = & \textbf{let} & bs +\!\!+ [b] = map\ (c \otimes)\ (scan\ \otimes\ (map\ q\ xs)) \\
 & & as = zip\ xs\ bs \\
 & & c \otimes x = \textbf{case}\ x < c\ \textbf{of} \\
 & & \quad True\ \rightarrow c \\
 & & \quad False \rightarrow x \\
 & & p\ (x, c) = \textbf{case}\ x < c\ \textbf{of} \\
 & & \quad True\ \rightarrow [] \\
 & & \quad False \rightarrow [x] \\
 & & g\ c = [] \\
 & & q\ x = x \\
 & \textbf{in}\ (reduce\ (+\!\!+)\ (map\ p\ as)) +\!\!+ (g\ b) &
\end{array}
$$

This can then be rewritten using the *accumulate* skeleton as:

$$
\begin{array}{lll}
smaller\ xs\ c = & \textbf{let} & c \otimes x = \textbf{if}\ x < c\ \textbf{then}\ c\ \textbf{else}\ x \\
 & & p\ (x, c) = \textbf{if}\ x < c\ \textbf{then}\ []\ \textbf{else}\ [x] \\
 & & g\ c = [] \\
 & & q\ x = x \\
 & \textbf{in}\ [\![g, (p, +\!\!+), (q, \otimes)]\!]\ xs\ c &
\end{array}
$$

The benefits of using the *accumulate* skeleton should be obvious, as it allows quite a general class of program to be parallelised, and is quite intuitive. One issue inherent in the definition of *accumulate* is that it makes use of a combination of skeletons, and combining skeletons efficiently can be difficult in general. However, due to the specific skeletons used in *accumulate*, it can be efficiently implemented in parallel, as the authors show using MPI, a well-known message-passing parallel library. Fusion [37] is used to merge many of the combinations of skeletons used in the definition of *accumulate*, removing intermediate data-structures and allowing for a much more efficient implementation. Defining programs in terms of *accumulate* should be a reasonable process, as the developer need only identify the parameters needed by $[\![g, (p, \oplus), (q, \otimes)]\!]$.

A difficulty associated with this would lie with the need to find suitable associative operators for $\oplus$ and $\otimes$, however the context preservation transformation in [15] can provide a solution for this.

2.3.2.7 Zippers

Morihata et al. [60] present another parallelisation technique based upon the third homomorphism theorem in which the theorem is generalised from lists to trees. Rather than using the tree itself to define a parallel algorithm, this approach uses *zippers* [43] to represent the path from the root of a tree to a node where a downward function will evaluate a path in a downward fashion, and an upward function will evaluate a path in an upward fashion. Obviously, an efficient divide-and-conquer algorithm should split a path in half and evaluate each half in parallel.

A zipper is a list containing the contexts left after walking a tree [60]. When walking a tree, if a step is down-right from a node its left child is added to the zipper and if a step is down-left from a node then its right child is added to the zipper. As an example, a zipper for a tree containing numbers at its nodes is shown below:

$$
\begin{array}{lcl}
data\ Either\ a\ b & ::= & L\ a \\
 & | & R\ b \\
 & & \\
data\ Tree & ::= & Leaf \\
 & | & Node\ Int\ Tree\ Tree \\
 & & \\
data\ Zipper & ::= & [Either\ (Int, Tree)\ (Int, Tree)]
\end{array}
$$

Consider the function $sumTree$ which calculates the sum of a tree, xs, containing numbers at its nodes, shown below:

$$
\begin{array}{lcl}
sumTree & = & \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Leaf \rightarrow 0 \\
 & & \quad Node\ n\ l\ r \rightarrow n + sumTree\ l + sumTree\ r
\end{array}
$$

Assuming that xs has been converted to a zipper, versions which calculate $sumTree$ in a downward fashion, $sumTree_{\downarrow}$, and an upward fashion, $sumTree_{\uparrow}$ can be defined as shown below:

$$
\begin{aligned}
sumTree_{\downarrow} = \lambda xs.\ \textbf{case}\ xs\ \textbf{of}\\
&[] && \to 0\\
&(x +\!\!+ [L\ (n,l)]) && \to sumTree_{\downarrow}\ x + n + sumTree\ l\\
&(x +\!\!+ [R\ (n,r)]) && \to sumTree_{\downarrow}\ x + n + sumTree\ r
\end{aligned}
$$

$$
\begin{aligned}
sumTree_{\uparrow} = \lambda xs.\ \textbf{case}\ xs\ \textbf{of}\\
&[] && \to 0\\
&([L\ (n,l)] +\!\!+ x) && \to n + sumTree\ l + sumTree_{\uparrow}\ x\\
&([R\ (n,r)] +\!\!+ x) && \to n + sumTree\ r + sumTree_{\uparrow}\ x
\end{aligned}
$$

In order to show the correspondence between functions defined on trees and those defined on zippers, Morihata et al. introduce the notion of *path-based computations.*

Definition 21 (Path-Based Computations) A function $h' :: Zipper \to B$ is a path-based computation of $h :: Tree \to A$ if there exists a function $\psi :: B \to A$ and the following holds:

$$
\begin{aligned}
&\psi \circ h' = h \circ z2t\\
&\textbf{where}\\
&z2t \quad = \quad \lambda xs.\ \textbf{case}\ xs\ \textbf{of}\\
&\qquad\qquad [] && \to Leaf\\
&\qquad\qquad ([L\ (n,l)] +\!\!+ x) && \to Node\ n\ l\ (z2t\ x)\\
&\qquad\qquad ([R\ (n,r)] +\!\!+ x) && \to Node\ n\ (z2t\ x)\ r
\end{aligned}
$$

In order to divide a binary-tree for parallelisation Morihata et al. perform division based on one-hole contexts, where a one-hole context is a zipper representing a walk of a binary-tree to some arbitrary leaf. At each division a node is selected along the path from the root to the hole and at this point the tree is split into an upper part, a lower part and a part containing the tree at the selected node. The upper and lower parts may then be divided again.

Following from this, in order to parallelise computations on binary-trees there are three required functions: ϕ which takes the result of a full tree and merges that with its parent, $\odot$ which merges the results of two one-hole contexts and ψ which evaluates a full tree using the result of a one-hole context. As one-hole contexts are described using zippers, these functions can be described using path-based computations.

Definition 22 (Decomposition of Binary-Trees) A decomposition of the function $h :: Tree \to A$ is a triple $(\phi, \odot, \psi)$ consisting of the associative

function $\odot :: B \to B \to B$ and the functions $\phi :: Either\ (Int, A)\ (Int, A) \to B$ and $\psi :: B \to A$ such that:

$$
\begin{array}{lll}
h' = \lambda xs.\ \textbf{case}\ xs\ \textbf{of} & & \\
\qquad [] & \to & \iota_{\odot} \\
\qquad [L\ (n,t)] & \to & \phi\ (L\ (n, h\ t)) \\
\qquad [R\ (n,t)] & \to & \phi\ (R\ (n, h\ t)) \\
\qquad (x +\!\!+ y) & \to & h'\ x \odot h'\ y
\end{array}
$$

If a decomposition of h exists in a form matching Definition 22, then h is said to be *decomposable*. Decomposable functions can be efficiently evaluated in parallel using *tree-contraction* [57, 58, 1] techniques. If each of $(\phi, \odot, \psi)$ is applicable in constant time, then h can be evaluated for a tree of n nodes on p processors in $O(n/p + log\ p)$ time.

As an example, consider the $sumTree$ function shown earlier. Assuming the binary-tree $sumTree$ is defined on has been converted to a zipper, $sumTree$ is decomposable as shown by the following function:

$$
\begin{array}{lll}
sumPara = \lambda xs.\ \textbf{case}\ xs\ \textbf{of} & & \\
\qquad [] & \to & 0 \\
\qquad [L\ (n,t)] & \to & n + sumTree\ t \\
\qquad [R\ (n,t)] & \to & n + sumTree\ t \\
\qquad (x +\!\!+ y) & \to & sumPara\ x + sumPara\ y
\end{array}
$$

While $sumPara$ is easy to derive due to the associativity of $+$, this is not always the case. The *third homomorphism theorem on trees* attempts to solve this problem. According to the theorem, a function is decomposable if and only if there exists a path-based computation of h that is both upwards and downwards [60]. Once such a path-based computation exists, then the necessary operators can be derived according to Lemma 2.

Lemma 2 Given a function h' which is a path-based computation of h such that $\psi \circ h' = h \circ z2t$, there exists a decomposition $(\phi, \odot, \psi)$ such that the following holds:

$$
\begin{array}{lll}
\phi\ (L\ (v, h\ t)) & = & h'\ [L\ (v,t)] \\
\phi\ (R\ (v, h\ t)) & = & h'\ [R\ (v,t)]
\end{array}
$$

$$
a \odot b = h'\ (h'^{\circ}\ a +\!\!+ h'^{\circ}\ b)
$$

While this approach to parallelisation is powerful, it is not without its problems. It presents an interesting approach to partitioning the data contained within a binary tree, however the partitioning technique is quite complicated and relies upon zippers and presents no concrete methodology for generating zippers from a binary-tree, assuming that the developer has provided such a function.

The parallelisation technique relies upon a zipper generated by walking a tree from its root to some arbitrary leaf, yet presents no methodology to select a leaf that will generate an optimal zipper. A technique for generating an optimal zipper is a necessity as each one-hole context generated by walking the binary-tree will contain one of the sub-trees that is a child of that node, and this will then be evaluated using the original function. If these sub-trees are badly partitioned, then the resulting parallel program may not be efficient.

It also requires that the user specify both upward and downward solutions to a given problem which must be equivalent. The form of both upward and downward functions is quite restrictive and defined in terms of zippers, so it is not realistic to expect a developer to define their programs in such forms.

2.4 Conclusion

To conclude, there are many existing transformation techniques which are capable of optimising and parallelising functional programs, such as fold/unfold based transformations, based on the work of Burstall & Darlington [10]. The surveyed techniques defined using fold/unfold methods are quite powerful. Deforestation [88] and positive-supercompilation are transformations capable of eliminating the use of intermediate data in functional programs and obtaining a linear increase in efficiency. Distillation is another, more powerful fold/unfold program transformation technique, which can obtain a super-linear increase and as such is an ideal candidate as the transformation technique to be used as part of the proposed solution.

Chin et al.'s [78, 13, 15] fold/unfold based work on parallelisation can be applied to complex functions such as those with accumulating parameters, nested recursion and conditional statements. However, it also requires that associativity and distributivity be specified for primitive functions by the developer and is only applicable to *list-paramorphisms* [56], and it is unrealistic

to expect developers to define functions in this form.

In addition to fold/unfold transformations there are also calculational methods based transformations. At the core of many such transformations is the *list*-homomorphism [42] which excludes a large class of programs. *List*-mutumorphisms are a more powerful approach, capable of describing all primitive list functions. These and their derivative works [70, 69, 7, 25, 26, 38, 39, 40, 41] can be used as part of a parallelisation process but require that they are supplied with data in the form of a *cons*-list, and also require the specification of associative/distributive operators to be used as part of the parallelisation process. While these techniques are quite powerful, the forms that their inputs are required to hold are quite restrictive and it is unrealistic to expect developers to define their functions in such forms.

These techniques are only applicable to lists, excluding the large class of programs that are defined on trees. One approach to parallelising trees is that of Morihata et al.'s [60] redefinition of the third homomorphism theorem [21] which is generalised to apply to trees. 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 requires that the user specify two functions in upward and downward form [60] which is quite restrictive so it is not realistic to expect a developer to define their programs in such a manner.

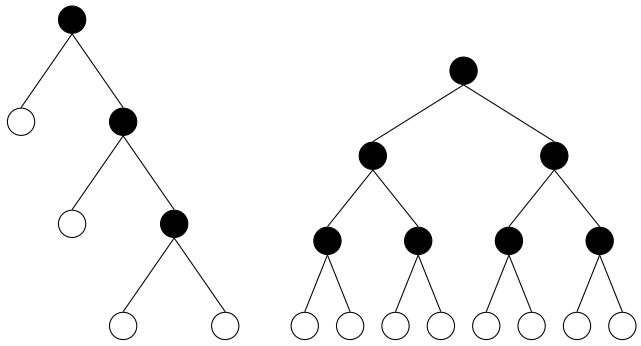
As a result of this survey, there is a clear need for an automatic parallelisation technique which is applicable to data of any type. This technique should not require or assume the definition of a function to convert the data used by the input program into a well-partitioned form, but should derive it itself. In addition to this, there should be no requirement for the developer to specify the associativity/distributivity of functions; these should also be derived automatically where necessary.

GpH has been selected as the explicit parallelisation technique to be used as part of the proposed solution 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.

Chapter 3

Automatic Partitioning

3.1 Introduction



(a) Poorly Partitioned (b) Well-partitioned

Figure 3.1: Examples of Partitioned Data

As mentioned in the introductory chapter, Chapter 1, at the core of any program to be parallelised lies the data it evaluates in order to produce its result. The distribution or partitioning of this data across parallel processes is a problem that presents itself to many developers, as if there is a significant imbalance between parallel processes, this can result in a parallel program with little to no gains in efficiency.

Consider again the partitioned data shown in Figure 1.1, reproduced in Figure 3.1. If a developer is parallelising a program in which the data is poorly partitioned, as shown in Figure 3.1a, and at each branch creates parallel processes to evaluate both the left and right child, then this will

result in a large imbalance in work between the parallel processes and an inefficient parallel program. However, if the data in the program can be restructured into a well-partitioned form, as shown in Figure 3.1b, and the resulting program is parallelised in a similar fashion, then this should result in parallel processes with a more even distribution of work and as a result, a more efficient parallel program.

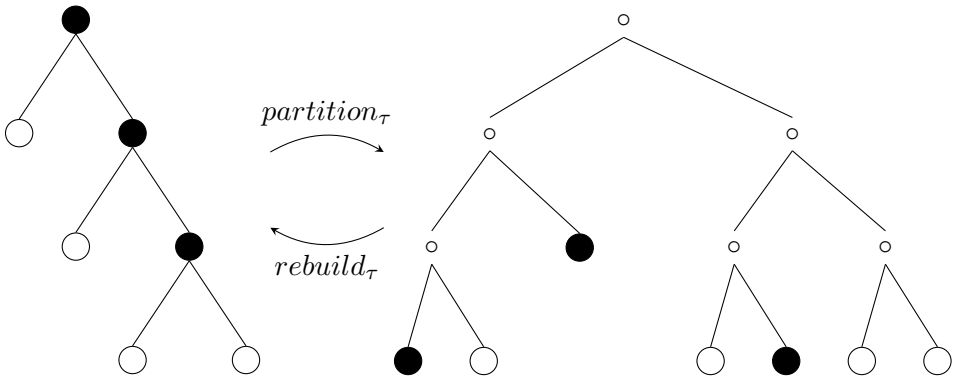


Figure 3.2: Data Partitioning Functions

To solve this problem, this chapter presents a novel data-type transformation which, given a sequential program defined using the language presented in Chapter 1 can be used to automatically redefine that program into one defined on well-partitioned data, so facilitating the automatic parallelisation of such programs. To achieve this, the technique automatically derives functions, $partition_\tau$ and $rebuild_\tau$, as shown in Figure 3.2, to convert the data that the given program evaluates into an efficiently partitioned form for parallelisation. At a high-level, the $partition_\tau$ function essentially serialises, in a depth-first fashion, potentially poorly-partitioned data and then creates a well-partitioned tree containing the serialised data. Conversely, the $rebuild_\tau$ function can be used to reconstruct the original data using a depth-first traversal of the well-partitioned data. Using these conversion functions, the technique then redefines the given program into one defined on well-partitioned data.

A *join*-list, as defined in Figure 3.3, is a data-type which enables the data it contains to be well-partitioned and is used at the core of many existing parallelisation techniques [70, 69, 7, 36, 39, 40, 41]. Each of these techniques require that their input programs are defined using a *cons*-list, for which

$$
\begin{array}{lcl}
data\ JList\ a & ::= & Singleton\ a \\
& | & Join\ (JList\ a)\ (JList\ a)
\end{array}
$$

Figure 3.3: *join*-List Type Definition

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 and this approach does not extend easily to programs defined using other data-structures. The material presented in this chapter allows data to be converted between any given type τ and well-partitioned *join*-lists using the functions $partition_\tau$ and $rebuild_\tau$, as shown in Figure 3.2.

A high-level overview of the automatic partitioning technique is presented in Figure 3.4. By combining the $rebuild_\tau$ function with distillation it is possible to automatically convert a given program into one defined on well-partitioned data. At a high level, the technique consists of three steps:

1. Given a program, f, compose f with $rebuild_\tau$ to create a function defined on *join*-lists.

2. Distill this composition to create a more efficient program equivalent to f, f_{wp}.

3. Partition the input data of f using $partition_\tau$ to generate a correct input for f_{wp}.

Obviously, any transformation which manipulates the data that a given program evaluates must ensure that there are no negative side-effects on that data. To ensure this, the transformations defined in this chapter observe the *data-preservation property*, as defined in Property 1.

Property 1 (Data-Preservation Property) For any unpartitioned data of type τ, to which the automatic partitioning technique is applied, the following holds: $rebuild_\tau \circ partition_\tau = id$

The remainder of this chapter is structured as follows: Section 3.2 describes the derivation of necessary data-types which are used as part of the partitioning and rebuilding functions. Section 3.3 describes the derivation of a function which will convert any data into a well-partitioned form.

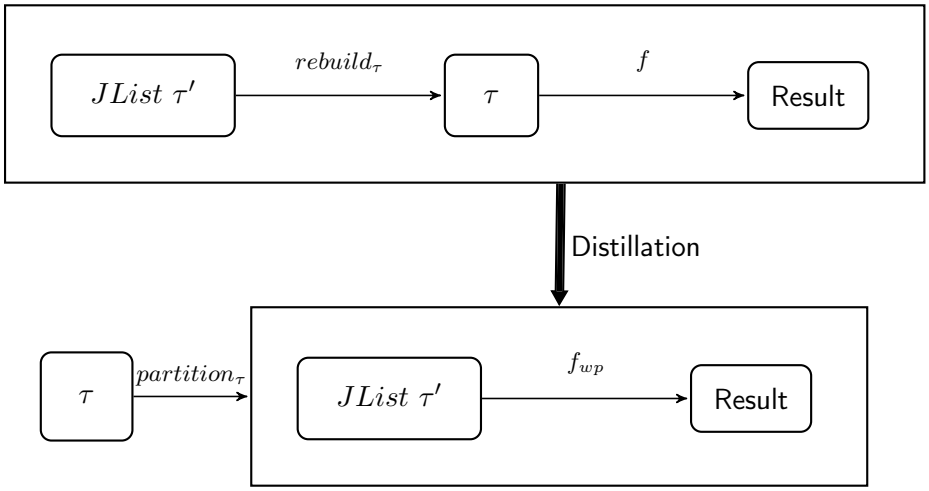


Figure 3.4: Distilling Programs on Well-Partitioned Data

Section 3.4 describes the derivation of a function which will convert well-partitioned data back into its original form. Section 3.5 describes how these techniques can be combined with distillation in order to derive programs defined on well-partitioned data automatically from ones defined on any data. Section 3.6 compares the material presented in this chapter to existing techniques. Finally, Section 3.7 presents conclusions drawn from the material presented in this chapter.

3.2 Defining Flattened Data-Types

In order to partition data of a given data-type, τ, the first step is to define a corresponding data-type, τ', which contains the non-recursive components of τ. The non-recursive components of τ are those that are not of type τ. The new data-type, τ', can be defined according to the rules shown in Figure 3.5. Removal of the recursive components of τ allows for an instance of τ to be converted into a flattened *cons*-list, which can then be converted to a well-partitioned *join*-list, $JList\ \tau'$. While in some cases the parallel processing of all data that a program is defined on may be desirable, this is not always the case. For example, it may make sense to parallelise the processing of a tree but it may not make sense to parallelise the processing of an integer. In order to allow for this, the presented technique allows the developer to specify a set of *parallelisable-types*, γ, where instances of each

parallelisable-type can be processed in parallel. All other types are referred to as *sequential-types*.

$$\mathcal{N}_\tau = \begin{cases} JList\ \tau' & \textbf{if } \tau \in \gamma \\ \tau & \textbf{otherwise} \end{cases}$$

where

τ is defined by:

$$\begin{array}{lll} \textbf{data } T & ::= & c_1\ t_{1_1}\ \ldots\ t_{1_n} \\ & \vdots & \\ & | & c_m\ t_{m_1}\ \ldots\ t_{m_n} \end{array}$$

τ' is a new data-type defined by:

$$\begin{array}{lll} \textbf{data } T' & ::= & c'_1\ \mathcal{N}_{t'_{1_1}}\ \ldots\ \mathcal{N}_{t'_{1_k}} \\ & \vdots & \\ & | & c'_m\ \mathcal{N}_{t'_{m_1}}\ \ldots\ \mathcal{N}_{t'_{m_k}} \end{array}$$

where

c'_i **is fresh**

$$\forall i \in \{1, \ldots, m\} : \langle t'_{i_1}, \ldots, t'_{i_k} \rangle = \begin{cases} removeAll_\tau \langle t_{i_1}, \ldots, t_{i_n} \rangle & \textbf{if } \tau \in \gamma \\ \langle t_{i_n}, \ldots, t_{i_n} \rangle & \textbf{otherwise} \end{cases}$$

$$removeAll_\tau\ \langle t_1, \ldots, t_n \rangle = \langle t \mid t \in \langle t_1, \ldots, t_n \rangle \wedge t \neq \tau \rangle$$

Figure 3.5: Transformation Rule for Defining τ' From τ

At a high level, if τ is a parallelisable type, then the rule $\mathcal{N}_\tau$ creates a new data-type, τ' which corresponds to τ. For each constructor in τ a corresponding constructor is created in τ' in which any recursive components have been removed, using the function $removeAll_\tau$ which removes all occurrences of τ from a given sequence. Given a program defined on a data-type τ, the rule $\mathcal{N}_\tau$, as defined in Figure 3.5, is applied as follows: If τ is not a parallelisable-type, then it is left unchanged. If τ is a parallelisable-type, then the data-type τ' is created, where, for each constructor, c_i, in the definition of τ, a new constructor, c'_i, is added to τ'. If τ is a parallelisable-type then any recursive components of c_i are not added to c'_i. Each of the components of c'_i are then transformed according to the rule $\mathcal{N}$. As an example, consider a program defined on a tree of integers, $TreeInt$, where $TreeInt$ is defined as shown below:

$$\begin{array}{lcl} data\ TreeInt & ::= & Leaf\ Int \\ & | & Node\ TreeInt\ TreeInt \end{array}$$

In order to facilitate the flattening of the data that is processed by the program, if $\gamma = \{TreeInt\}$, then $TreeInt'$ is defined according to $\mathcal{N}_{TreeInt}$, the result of which is shown below:

$$\begin{array}{lcl} \textbf{data}\ TreeInt' & ::= & Leaf'\ Int \\ & | & Node' \end{array}$$

In order to generate this definition of $TreeInt'$, $\mathcal{N}_{TreeInt}$, given the definition of $TreeInt$, replaces occurrences of $TreeInt$ with $(JList\ TreeInt')$, as $TreeInt$ is a parallelisable-type. In order to define $TreeInt'$ the constructors of $TreeInt$, $Leaf$ and $Node$ are examined as follows:

- The definition of $Leaf$ results in a corresponding constructor, $Leaf'$, being added to the definition of $TreeInt'$.
  - As $Leaf$ only contains one non-recursive data-component, Int, which is not a parallelisable-type, $Leaf'$ also has a data-component of type Int. This is also transformed according to $\mathcal{N}_{Int}$, but as it is not a parallelisable-type, it is not modified.
- The definition of $Node$ results in a corresponding constructor, $Node'$, being added to the definition of $TreeInt'$.
  - As $Node$ contains two recursive data-components of type $TreeInt$ (a parallelisable-type) these are not added to the definition of $Node'$ which contains no data-components as a result.

3.3 Partitioning Data Using *Join*-Lists

In order to allow for the partitioning of any data, a partitioning function, $partition_\tau$, is defined to convert any data into a well-partitioned *join*-list, according to the rules shown in Figure 3.7. Given a program defined on a data-type, τ, the rule $\mathcal{F}_\tau$, which generates a partitioning function is applied to τ as follows: if τ is not a parallelisable-type it is left unchanged. If τ is a parallelisable-type, then the functions $partition_\tau$ and $flatten_\tau$ are generated. At a high-level, as shown in Figure 3.6, $partition_\tau$ uses $flatten_\tau$

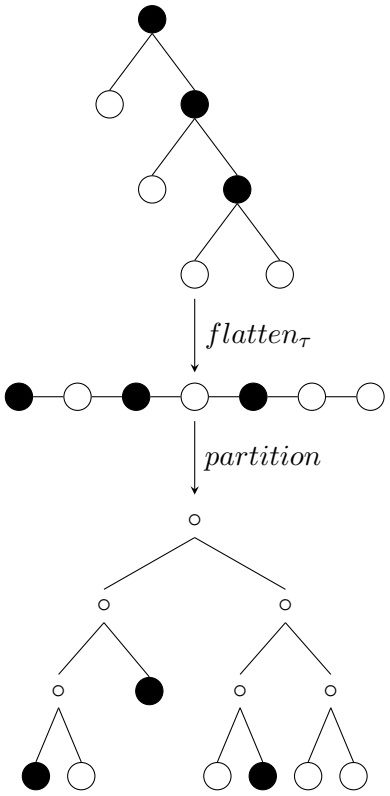


Figure 3.6: Automatic Partitioning Example

to convert data of type τ into a flattened *cons*-list, and then applies the function *partition* to the output of this to generate a well-partitioned *join*-list.

The function $flatten_\tau$ looks at the outermost constructor, c_i, of the given data. The components of c_i are split into those that are non-recursive data-components and those that are recursive components. A singleton *cons*-list containing an application of c_i' is then generated using c_i's non-recursive data-components. The recursive components of c_i, if any, are then flattened using $flatten_\tau$ and appended to this singleton *cons*-list. Each component of c_i' is also partitioned. Once this flattened *cons*-list has been generated, it is partitioned using the function *partition* which, given a *cons*-list, creates a well-partitioned *join*-list from that *cons*-list, using the function *split*, defined in Section 1.4.1.

The *split* function plays an important part in ensuring that the resulting *join*-list is well-partitioned. As the *split* function defined as part of the *hop*

$$\mathcal{F}_\tau = \begin{cases} partition_\tau & \textbf{if } \tau \in \gamma \\ id & \textbf{otherwise} \end{cases}$$

$$partition_\tau = partition \circ flatten_\tau$$

$$\left.\begin{array}{l} flatten_\tau = \lambda x.\ \textbf{case } x \textbf{ of} \\ \qquad c_i\ x_{i_1} \ldots x_{i_n} \rightarrow [c_i'\ (\mathcal{F}_{t_{i_1}}\ s_{i_1}) \ldots (\mathcal{F}_{t_{i_j}}\ s_{i_j})] +\!\!+ flatten_\tau\ p_{i_1} +\!\!+ \ldots +\!\!+ flatten_\tau\ p_{i_k} \\ \qquad\quad \textbf{where}: \ \{(s_{i_1} :: t_{i_1}), \ldots, (s_{i_j} :: t_{i_j})\} = \{(x :: t) \in \{x_{i_1}, \ldots, x_{i_n}\} \mid t \neq \tau \vee t \notin \gamma\} \\ \qquad\qquad\qquad\ \ \{(p_{i_1} :: t_{i_1}), \ldots, (p_{i_k} :: t_{i_k})\} = \{(x :: t) \in \{x_{i_1}, \ldots, x_{i_n}\} \mid t = \tau \wedge t \in \gamma\} \end{array}\right\} \quad \forall i \in \{1, \ldots, m\}$$

$$\begin{array}{l} partition = \lambda xs.\ \textbf{case } xs \textbf{ of} \\ \qquad [x] \qquad \rightarrow Singleton\ x \\ \qquad (y : ys) \rightarrow\ \textbf{case } split\ xs \textbf{ of} \\ \qquad\qquad\qquad Pair\ l\ r \rightarrow Join\ (partition\ l)\ (partition\ r) \end{array}$$

where
τ is defined by:

$$\begin{array}{l} \textbf{data } T\ T_1 \ldots T_k ::= c_1\ t_{1_1}\ \ldots\ t_{1_n} \\ \qquad\qquad\qquad\quad \vdots \\ \qquad\qquad\qquad\quad \mid\ c_m\ t_{m_1}\ \ldots\ t_{m_n} \end{array}$$

Figure 3.7: Transformation Rules for Partitioning Data

language splits a *cons*-list in half, it is reasonable to define a well-partitioned *join*-list as one in which the children of a $Join$ may only differ in size by a maximum of 1. This is referred to as the *well-partitioned property*, as defined in Property 2.

Property 2 (Well-Partitioned Property) A *join*-list is said to be well-partitioned iff for each child of a $Join$, the property *wpp* holds:

$$\begin{aligned} wpp\ (Join\ l\ r) &= |size\ l - size\ r| \leq 1 \wedge wpp\ l \wedge wpp\ r \\ wpp\ (Singleton\ x) &= True \end{aligned}$$

Though a well-partitioned binary tree resulting from the presented partitioning technique will uphold the well-partitioned property, it is worth noting that as a consequence of the flattening technique, the resulting tree may be one in which the left hand side of the *join*-list mainly consists of singletons containing constructors with no data-component(s). Such a situation arises when the data-type in the given program contains a constructor, c, which has no data-component(s), only recursive components. When flattened, such a constructor will result in a list where the head is a constructor, c', that has no data-component and the tail of the list is the result of flattening the recursive components of c. A consequence of this may be that, when evaluated in a parallel context, though the *join*-list is itself well-partitioned, there may still be an imbalance in the work distributed across parallel processes.

As an example, consider again a program defined on a tree of integers, $TreeInt$, where $\gamma = \{TreeInt\}$. In order to partition the data that is processed by the program, the function $partition_{TreeInt}$ is generated using $\mathcal{F}_{TreeInt}$, the result of which is shown below:

$$partition_{TreeInt} = partition \circ flatten_{TreeInt}$$

$$\begin{aligned} flatten_{TreeInt} = \lambda x.\ &\textbf{case}\ x\ \textbf{of} \\ &Leaf\ x_1 \rightarrow [Leaf'\ x_1] \\ &Node\ x_1\ x_2 \rightarrow [Node'] +\!\!+ flatten_{TreeInt}\ x_1 \\ &\qquad\qquad\qquad\quad +\!\!+ flatten_{TreeInt}\ x_2 \end{aligned}$$

Given the definition of $TreeInt$, $\mathcal{F}_{TreeInt}$ generates the necessary definitions of $partition_{TreeInt}$ and $flatten_{TreeInt}$ as $TreeInt$ is a parallelisable-type. $partition_{TreeInt}$ is simply a composition of *partition* and

$flatten_{TreeInt}$. In order to define $flatten_{TreeInt}$ the constructors of $TreeInt$ ($Leaf$ and $Node$) are examined as follows:

- If the data being converted by $flatten_{TreeInt}$ is a $Leaf$, this results in a singleton *cons*-list containing a constructor application of $Leaf'$ being created.
  - As $Leaf$ only contains one non-recursive data-component, Int, which is not a parallelisable-type, the variable bound to this component is applied as an argument to the $Leaf'$ constructor application. The data bound by this variable is also partitioned according to $\mathcal{F}_{Int}$, but as it is not of a parallelisable-type, no partitioning is necessary and it is left as-is.
- If the data being converted by $flatten_{TreeInt}$ is a $Node$, this results in a singleton *cons*-list containing a constructor application of $Node'$ being created.
  - As the only data-components of a $Node$ are two recursive data-components of type $TreeInt$ (a parallelisable-type) these are flattened using $flatten_{TreeInt}$ and appended to the singleton *cons*-list.

Figure 3.8 provides an example of how a sample $TreeInt$ is converted into a well-partitioned *join*-list containing instances of $TreeInt'$ using $partition_{TreeInt}$.

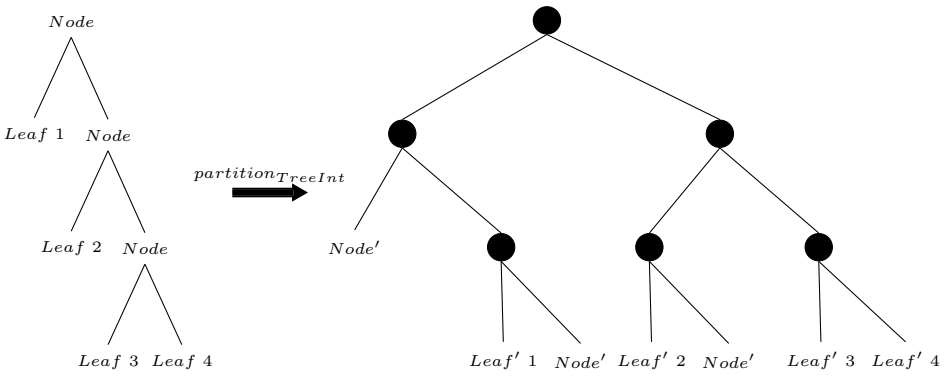


Figure 3.8: Example of Partitioning Using $partition_{TreeInt}$

3.4 Rebuilding Original Data from Well-Partitioned *Join*-Lists

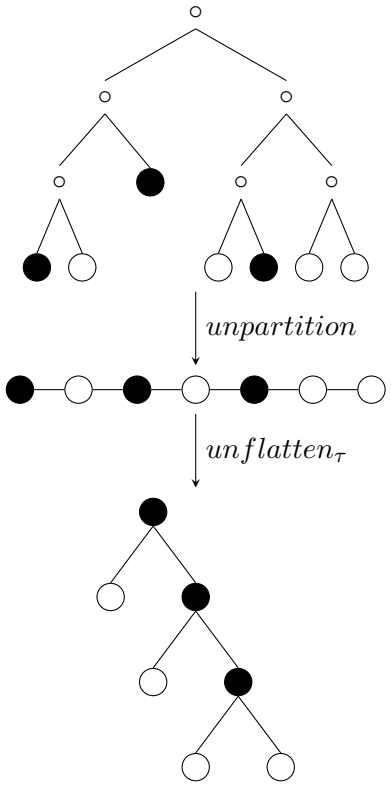


Figure 3.9: Automatic Rebuilding Example

As well as defining a partitioning function, a function, $rebuild_\tau$, which rebuilds an instance of type τ from a well-partitioned *join*-list, as shown in Figure 3.9 containing instances of τ' is defined according to the rules shown in Figure 3.10. Given a program defined on a data-type, τ, the rule $\mathcal{R}_\tau$, which generates a rebuilding function is applied to τ as follows: if τ is not a parallelisable-type it is left unchanged. If τ is a parallelisable-type, then the functions $rebuild_\tau$ and $unflatten_\tau$ are generated. At a high-level, $rebuild_\tau$ uses the function *unpartition* to convert a well-partitioned *join*-list into a flattened *cons*-list. Following this, $unflatten_\tau$ is used to convert the flattened *cons*-list into an instance of τ.

Given a *cons*-list whose first element is an instance of τ', the function $unflatten_\tau$ looks at its outermost constructor, c'_i. The arguments of c'_i cor-

$$
\begin{array}{lll}
\mathcal{R}_\tau & = \begin{cases} rebuild_\tau & \textbf{if } \tau \in \gamma \\ id & \textbf{otherwise} \end{cases} & \\
\\
rebuild_\tau & = fst \circ unflatten_\tau \circ unpartition & \\
\\
unflatten_\tau & = \lambda xs.\textbf{case } xs \textbf{ of} & \\
 & \quad (x : xs) \rightarrow \textbf{case } x \textbf{ of} & \\
 & \qquad c'_i\ s_{1_i} \ldots s_{j_i} \rightarrow \textbf{case } unflatten_\tau\ xs \textbf{ of} & \\
 & \qquad\qquad (p_{1_i}, ps_{1_i}) \rightarrow \textbf{case } unflatten_\tau\ ps_{1_i} \textbf{ of} & \\
 & \qquad\qquad\qquad \vdots & \\
 & \qquad\qquad\qquad \textbf{case } unflatten_\tau\ ps_{k_i - 1} \textbf{ of} & \\
 & \qquad\qquad\qquad (p_{k_i}, ps_{k_i}) \rightarrow (\mathcal{U}_{c_i}\ (\mathcal{R}_{t_{1_i}}\ s_{1_i}) \ldots (\mathcal{R}_{t_{j_i}} s_{j_i})\ p_{1_i} \ldots p_{k_i},\ ps_{k_i}) & \Big\} \forall i \in \{1, \ldots, m\} \\
 & \qquad \textbf{where}: \{(s_{1_i} :: t_{1_i}), \ldots, (s_{j_i} :: t_{j_i})\} = \{(x :: t) \in \{x_{1_i}, \ldots, x_{n_i}\} \mid t \neq \tau \vee t \notin \gamma\} & \\
 & \qquad\qquad\ \ \{(p_{1_i} :: t_{1'_i}), \ldots, (p_{k_i} :: t_{k_i})\} = \{(x :: t) \in \{x_{1_i}, \ldots, x_{n_i}\} \mid t = \tau \wedge t \in \gamma\} & \\
\\
\mathcal{U}_{c_i} & = \lambda s_1 \ldots s_j . p_1 \ldots p_k .\ c_i\ x_1 \ldots x_n & \\
 & \quad \textbf{where}\ \ \{(x :: t) \in \{x_1, \ldots, x_n\} \mid t \neq \tau \vee t \notin \gamma\} = \{s_1, \ldots, s_j\} & \\
 & \qquad\qquad\ \ \{(x :: t) \in \{x_1, \ldots, x_n\} \mid t = \tau \wedge t \in \gamma\} = \{p_1, \ldots, p_k\} & \\
\\
unpartition & = \lambda x.\ \textbf{case } x \textbf{ of} & \\
 & \qquad Singleton\ x \rightarrow [x] & \\
 & \qquad Join\ l\ r \rightarrow (unpartition\ l) +\!\!+ (unpartition\ r) &
\end{array}
$$

Figure 3.10: Transformation Rules for Rebuilding Data

respond to the non-recursive data-components of the constructor c_i. As τ is a parallelisable-type, the remainder of the list contains the flattened instances of τ' which are the recursive components of c_i and can be rebuilt using $unflatten_\tau$ where for each recursive component of c_i, there will be a nested **case** in which the selector is a call to $unflatten_\tau$. An application of c_i is then created according to the rule $\mathcal{U}_{c_i}$, by placing the components in their correct location, where each data-component of the constructor c_i' has been rebuilt. The input to $unflatten_\tau$ can be generated by applying the function *unpartition* to a well-partitioned *join*-list.

The result of applying *unpartition* to a well-partitioned *join*-list is a *cons*-list that is completely consumed in the rebuilding of the original data. It is therefore reasonable to expect that once the original data has been rebuilt, every component of that *cons*-list will have been consumed. This is observed by the presented rebuilding technique as the *rebuilding property*, as defined in Property 3.

Property 3 (Rebuilding Property) Given a well-partitioned *join*-list resulting from $partition_\tau$, when the original data has been rebuilt, the following property must hold:

$$snd \circ unflatten_\tau \circ unpartition = []$$

As an example, consider again a program defined on a tree of integers, $TreeInt$, where $\gamma = \{TreeInt\}$. In order to rebuild the data contained in a well-partitioned *join*-list containing instances of τ', the function $rebuild_{TreeInt}$ is generated, the result of which is shown below:

$$
\begin{array}{l}
rebuild_{TreeInt} \quad = fst \circ unflatten_{TreeInt} \circ unpartition \\
\\
unflatten_{TreeInt} = \\
\lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
\quad (x : xs) \rightarrow\ \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad Leaf'\ x_1 \rightarrow (Leaf\ x_1,\ xs) \\
\qquad\qquad Node' \quad \rightarrow\ \textbf{case}\ unflatten_{TreeInt}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad (x_1,\ xs_1) \rightarrow \textbf{case}\ unflatten_{TreeInt}\ xs_1\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad (x_2,\ xs_2) \rightarrow (Node\ x_1\ x_2,\ xs_2)
\end{array}
$$

Given the definition of $TreeInt$, $\mathcal{R}_{TreeInt}$ generates the necessary definitions of $rebuild_{TreeInt}$ and $unflatten_{TreeInt}$ as $TreeInt$ is a parallelisable-type. $rebuild_{TreeInt}$ is simply a composition of $unflatten_{TreeInt}$ and *unpartition*, which returns a $Pair$ containing the rebuilt instance of $TreeInt$ and a *cons*-list. The function fst is also used to return the first element of this pair, which is the rebuilt $TreeInt$. In order to define $flatten_{TreeInt}$ the constructors of $TreeInt$ ($Leaf$ and $Node$) are examined as follows:

- If the first element of the flattened *cons*-list being converted by $unflatten_{TreeInt}$ is a $Leaf'$, this results in an application of the $Leaf$ constructor being created.
  - As $Leaf$ only contains one non-recursive data-component, Int, which is not a parallelisable-type, $Leaf'$ also contains a data-component of type Int, and the variable bound to this component is applied as an argument to the $Leaf$ constructor application. The data bound by this variable is also rebuilt according to $\mathcal{R}$, but as it is not of a parallelisable-type, no rebuilding is necessary and it is left as-is.
- If the first element of the flattened *cons*-list being converted by $unflatten_{TreeInt}$ is a $Node'$, this results in an application of the $Node$ constructor being created.
  - As the only data-components of a $Node$ are two recursive data-components of type $TreeInt$ (a parallelisable-type) this means that they have been flattened and are contained in the remainder of the list being converted by $unflatten_{TreeInt}$. As a result, the first of these components is obtained by applying $unflatten_{TreeInt}$ to the remainder of the list, which returns a $Pair$ containing the first data-component and the remainder of the list left after rebuilding the first data-component. The second data-component is then obtained in the same way and these two components are applied as arguments to the $Node$ constructor application.

Figure 3.11 provides an example of how a sample *join*-list containing instances of $TreeInt'$ is converted into an instance of $TreeInt$ using $rebuild_{TreeInt}$.

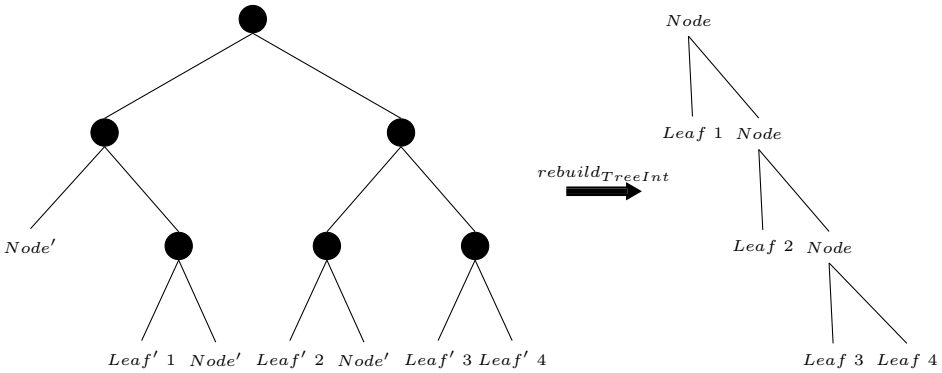


Figure 3.11: Example of Rebuilding Using $rebuild_{TreeInt}$

3.5 Distilling Programs defined on Well-Partitioned Data

As shown in Figure 3.12, given a sequential program, f, defined using a data-type, τ, once both $partition_\tau$ and $rebuild_\tau$ have been defined, it is possible to automatically convert f into an equivalent program, f_{wp}, which is defined using well-partitioned data. By applying distillation, denoted $\mathcal{D}$, to the composition of f and $rebuild_\tau$ ($\mathcal{D}[\![f \circ rebuild_\tau]\!]$) f_{wp} can be automatically derived.

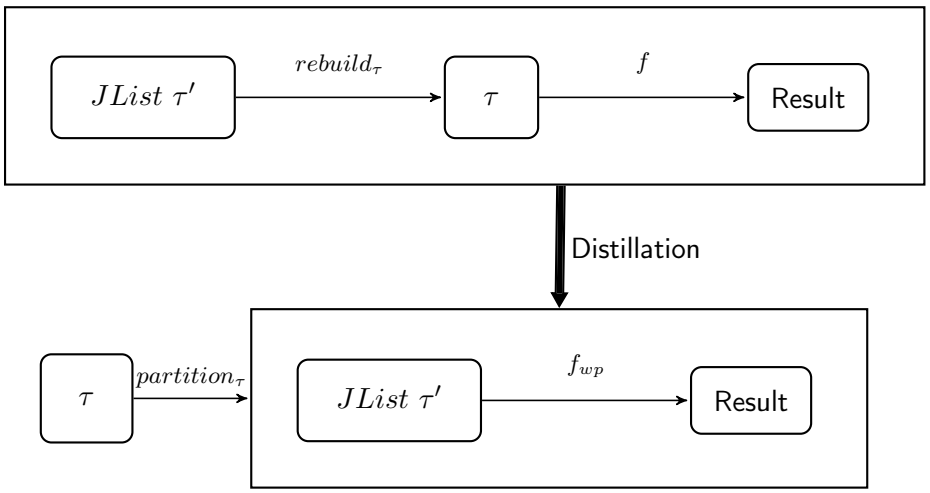


Figure 3.12: Distilling *Well*-Partitioned Programs

In addition to generating f_{wp}, a well-partitioned input to f_{wp} must also be

generated. The function $partition_\tau$ can be used to generate an appropriate input to f_{wp} using the original input to f. As an example, consider the function $sumTree$, which calculates the sum of the numbers contained in a $Tree$, as shown below:

$$\begin{array}{lcl} sumTree & = & \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\ & & \quad Leaf\ x \quad\ \ \rightarrow x \\ & & \quad Node\ x_1\ x_2 \rightarrow sumTree\ x_1 + sumTree\ x_2 \end{array}$$

By composing $sumTree$ with $rebuild_{TreeInt}$, defined in Section 3.4, and applying distillation to this composition, a new function, $sumTree_{wp}$, can be derived. $sumTree_{wp}$, shown below, is a program equivalent to $sumTree$ but is defined using well-partitioned data.

$$\begin{array}{lcl} sumTree_{wp} & = & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\ & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\ & & \qquad\qquad\qquad Leaf'\ x \rightarrow x \\ & & \quad Join\ l\ r \quad\ \ \rightarrow\ \textbf{let}\ r' = sumTree_{wp}\ r \\ & & \qquad\qquad\qquad \textbf{in}\ sumTree'_{wp}\ l\ r' \end{array}$$

$$\begin{array}{lcl} sumTree'_{wp} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\ & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\ & & \qquad\qquad\qquad Leaf'\ x \rightarrow x + n \\ & & \qquad\qquad\qquad Node' \quad\ \rightarrow n \\ & & \quad Join\ l\ r \quad\ \ \rightarrow \textbf{let}\ r' = sumTree'_{wp}\ r\ n \\ & & \qquad\qquad\qquad \textbf{in}\ sumTree'_{wp}\ l\ r' \end{array}$$

Consider again the well-partitioned *join*-list that results from partitioning an instance of $TreeInt$, as shown in Figure 3.8. If that *join*-list were to be supplied as an argument to $sumTree_{wp}$, the result of its evaluation would be as shown in Figure 3.13. At each $Join$ that is encountered, the *join*-list is split in half and the sum is calculated for the left child, followed by the sum of the right child.

3.6 Related Work

There are many existing works that aim to resolve the same problem that the presented transformation does: mapping potentially badly partitioned data into a form that can be efficiently parallelised. Some work, such as

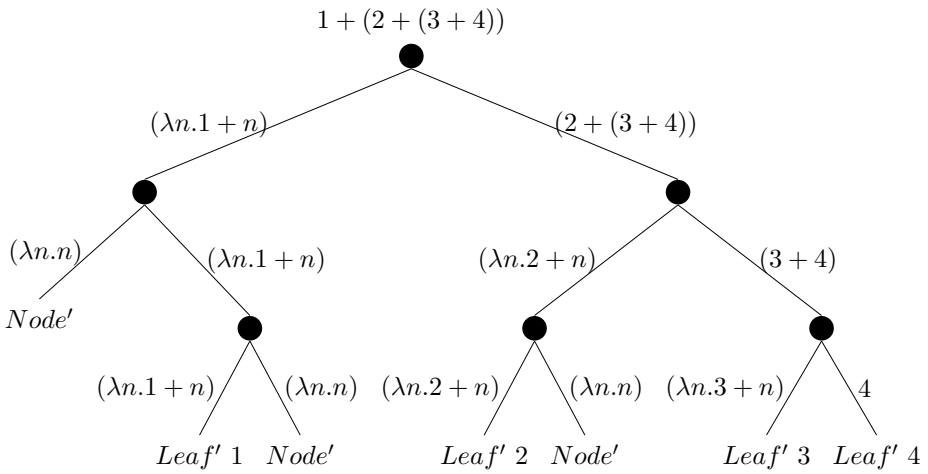


Figure 3.13: Evaluation of $sumTree_{wp}$ on a Sample *Join*-List

list-homomorphisms [5, 4, 50, 19, 3] and their derivative works [70, 69, 7, 36, 39, 40, 41] and Chin et. al's [78, 13, 15] work on parallelisation via context-preservation simply assume 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.

One approach to parallelising trees is that of Morihata et al.'s [60] redefinition of the third homomorphism theorem [21] which is generalised to apply to trees. This approach makes use of zippers [43] to partition the path from the root of a tree to an arbitrary leaf. A zipper contains a list of contexts left after walking a tree. When walking a tree, if a step is down-right from a node its left child is added to the zipper and if a step is down-left from a node then its right child is added to the zipper. While this work presents an interesting approach to partitioning the data contained within a *binary*-tree, it is not without its problems. It presents no concrete methodology for generating zippers from *binary*-trees and assumes that the developer has provided such a function. In addition it presents no methodology to select a leaf that will generate an optimal zipper. A technique for generating an optimal zipper is a necessity as each one-hole context generated by walking the *binary*-tree will contain one of the sub-trees that is a child of that node, and this will then be evaluated using the original sequential function. If these sub-trees are poorly-partitioned, then the resulting parallel program may be

no more efficient than the sequential program. As this technique requires the developer to provide a function to generate a zipper from a tree it is not fully automatic, and is only applicable to *binary*-trees. The automated partitioning technique presented in this chapter requires no input from the developer beyond an input program, and is also applicable to trees of any shape and size.

3.7 Conclusion

In conclusion, this chapter has presented a novel data partitioning technique which, given a program defined on any data-type, automatically partitions data of that type, and redefines the given program into one defined on the well-partitioned data, in order to facilitate the parallelisation of functional programs. The presented data-type transformation system forms the data-partitioning component of the proposed solution.

It is worth noting that the presented partitioning technique may have merit in other fields. For example, within cloud and distributed computing, often, several tasks or jobs may be split and completed concurrently on distributed machines or nodes, be they physical or virtualised. As with the presented partitioning technique, the *poorly*-partitioned data would first be serialised via the flattening transformation, and then partitioned to suit the underlying distribution technique. To that end, though the presented technique partitions data using a *join*-list, it is straightforward to redefine it to create well-partitioned data with a different structure, as long as the data-preservation, well-partitioned and rebuilding property are defined and observed for that structure.

The automated partitioning technique presented in this chapter provides an answer to R.Q. 1 which asks: given any data-type can a corresponding data-type be defined which will allow for efficient partitioning of the data? The transformation rules presented in this chapter allow for the extraction of the data belonging to any data-type into a form that allows for it to be well-partitioned automatically, as shown in Chapter 6 and Chapter 7. This chapter also provides an answer to R.Q. 2 which asks: can program transformation be used to automatically redefine a program defined on any data-type into an equivalent program defined on well-partitioned data? By defining a technique which allows the extraction of the data belonging to a data-type

into a well-partitioned form, and combining these techniques with distillation this chapter shows that program transformation can be used to convert a given program into one defined on well-partitioned data automatically.

Chapter 4

Automatic Parallelisation

4.1 Introduction

As the transformation technique in Chapter 3 provides a means to convert any data into well-partitioned data, its output is an ideal candidate for the design of an automatic parallelisation technique. In addition to this, as the automatic partitioning technique also provides a program defined in terms of this well-partitioned data, it makes sense to design a parallelisation technique which can automatically parallelise expressions in the resulting program which operate on well-partitioned data.

By defining a technique by which a developer can automatically convert a sequential program defined on any data-type into an equivalent program defined on a *join*-list and then parallelising this program, the difficulties associated with the parallelisation 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.

Given a derived program that evaluates well-partitioned *join*-lists, it makes sense to target expressions evaluating *join*-lists for parallelisation. Due to the nature of distillation, and its generalisation phase, most expressions which evaluate *join*-lists will be extracted using a series of nested **let** statements, the depth of which will vary between expressions. As such, the parallelisation technique presented in this chapter targets these nested **let** statements and converts any series of nested **let** statements which evaluate *join*-lists in to a series of nested parallel **let** statements. The result of this

is that as these **let** statements are being traversed each individual **let** that evaluates a *join*-list is sparked for parallel evaluation.

This chapter presents a novel program transformation technique which automatically converts a given sequential functional program, f, defined on any data-type into an equivalent explicitly parallel program defined on well-partitioned data. A high level overview of this technique is shown in Figure 4.1. Once the given sequential program has been converted into one defined on well-partitioned data, f_{wp}, using the techniques presented in Chapter 3, an explicit parallelisation transformation is applied to the program in order to derive an equivalent program, f_{par}, in which the evaluations of functions operating on well-partitioned *join*-lists have been sparked in parallel.

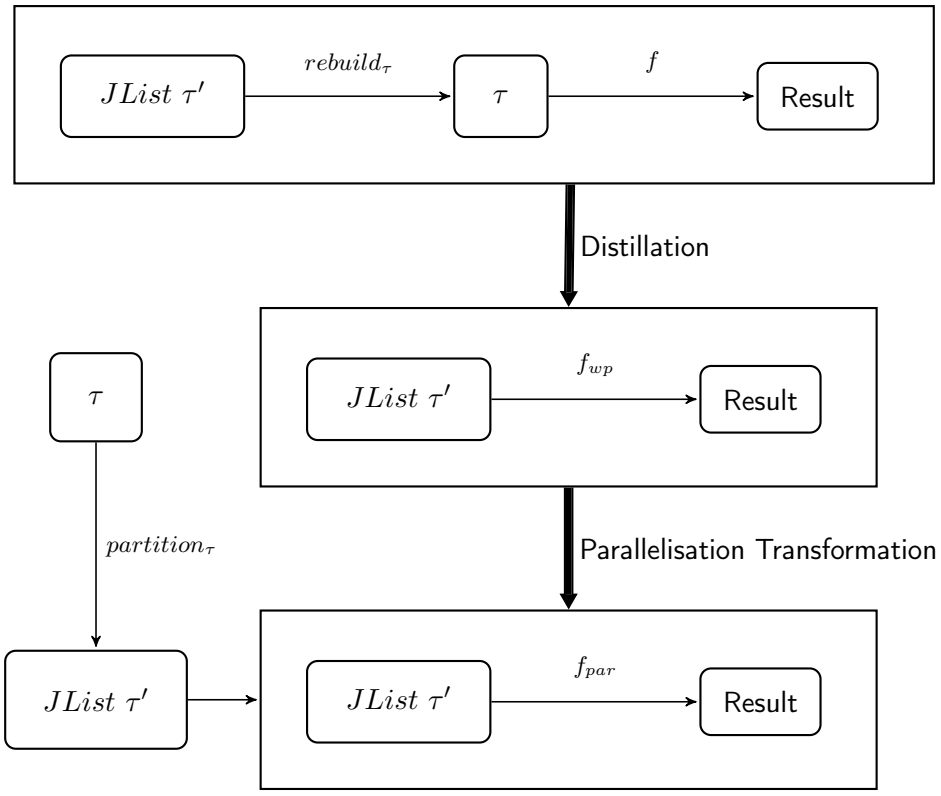


Figure 4.1: Parallelisation Process

The remainder of this chapter is structured as follows: Section 4.2 describes the process by which a program defined on well-partitioned data can be explicitly parallelised. Section 4.3 presents an example application of the automatic parallelisation technique to the *sumTree* running example.

Section 4.4 presents conclusions drawn from the material presented in this chapter.

4.2 Explicit Parallelisation of Functional Programs

This section defines a fully automatic transformation which can explicitly parallelise a functional program defined on well-partitioned data. Given a sequential program, f_{wp}, defined on well-partitioned data which has been derived from a program, f, defined on any data-type, the final step of the parallelisation technique is to apply another transformation, $\mathcal{P}$, in order to explicitly parallelise expressions operating on this well-partitioned *join*-list, resulting in an equivalent parallel program, f_{par}. The transformation rules for $\mathcal{P}$ are defined as shown in Fig. 4.2.

In addition to $\mathcal{P}$, another set of transformation rules, $\mathcal{P}_{br}$, are presented which are used to enable explicit parallelism within the branches of case expressions. As the transformation rules $\mathcal{P}$ are applied to the output of distillation, which is in the let-normal form of the *hoplet* language, as defined in Section 1.4.1, the transformation rules are defined using this language. Within both sets of transformation rules, use is made of a local parameter ψ, which contains the set of variable names that are bound to well-partitioned *join*-lists. The elements of ψ are referred to as *parallelisable variables*, and upon application of $\mathcal{P}$ to a program, ψ is initially empty, and is expanded as the program is traversed by the transformation rules.

As the original program f takes a parameter (x, say) of type τ, and f_{par} is defined using a *join*-list containing instances of τ', x must first be converted into a well-partitioned *join*-list. This can be completed by applying $partition_{\tau}$ to x.

These rules are applied in a top-down manner and the majority should be self explanatory: variables, abstraction variables, **case** selectors and constructor names are left unmodified. The bodies of constructors and abstractions and function definitions are transformed according to the parallelisation rules. Essentially, as the expression is traversed and **let** statements are encountered which operate on *join*-lists, these are converted into parallelised **let** statements via the use of the *par* and *pseq* strategies.

When a **case** statement is encountered, its patterns are examined, via $\mathcal{P}_{br}$, for the presence of $Join$ constructors. If a $Join$ constructor is found

$$
\begin{array}{ll}
\mathcal{P}[\![x]\!]\ \psi & = x \\
\mathcal{P}[\![c\ de_1\ \ldots\ de_n]\!]\ \psi & = c\ (\mathcal{P}[\![de_1]\!]\ \psi)\ \ldots\ (\mathcal{P}[\![de_n]\!]\ \psi) \\
\mathcal{P}[\![f]\!]\ \psi & = f \\
\mathcal{P}[\![\lambda x.de]\!]\ \psi & = \lambda x.\mathcal{P}[\![de]\!]\ \psi \\
\mathcal{P}[\![de\ x]\!]\ \psi & = (\mathcal{P}[\![de]\!]\ \psi)\ x \\
\multicolumn{2}{l}{\mathcal{P}[\![\textbf{case}\ x\ \textbf{of}\ p_1 \to de_1 \mid \ldots \mid p_k \to de_k]\!]\ \psi} \\
\multicolumn{2}{l}{\quad = \textbf{case}\ x\ \textbf{of}\ (\mathcal{P}_{br}[\![p_1 \to de_1]\!]\ \psi) \mid \ldots \mid (\mathcal{P}_{br}[\![p_k \to de_k]\!]\ \psi)} \\
\multicolumn{2}{l}{\mathcal{P}[\![\textbf{let}\ x = de_0\ \textbf{in}\ de_1]\!]\ \psi} \\
\multicolumn{2}{l}{\quad = \begin{cases}
\textbf{let}\ x = \mathcal{P}[\![de_0]\!]\ \psi & \textbf{if}\ (\psi \cap fv(de_0)) \neq \varnothing \\
\textbf{in}\ x\ \text{`}par\text{`}\ (\mathcal{P}[\![de_1]\!]\ \psi) & \wedge\ (\psi \cap fv(de_1)) \neq \varnothing \\
\\
\textbf{let}\ x = \mathcal{P}[\![de_0]\!]\ \psi & \textbf{if}\ \ (\psi \cap fv(de_0)) \neq \varnothing \\
\textbf{in}\ x\ \text{`}pseq\text{'}\ (\mathcal{P}[\![de_1]\!]\ \varnothing) & \wedge\ (\psi \cap fv(de_1)) = \varnothing \\
\\
\textbf{let}\ x = \mathcal{P}[\![de_0]\!]\ \varnothing & \\
\textbf{in}\ \mathcal{P}[\![de_1]\!]\ \varnothing & \textbf{otherwise}
\end{cases}} \\
\multicolumn{2}{l}{\mathcal{P}[\![f\ x_1\ \ldots\ x_n\ \textbf{where}\ f = \lambda x_1\ \ldots\ x_n.de]\!]\ \psi} \\
\multicolumn{2}{l}{\quad = f\ x_1\ \ldots\ x_n\ \textbf{where}\ f = \lambda x_1\ \ldots x_n.(\mathcal{P}[\![de]\!]\ \psi)} \\
\\
\mathcal{P}_{br}[\![Join\ x_1\ x_2 \to de]\!]\ \psi & = Join\ x_1\ x_2 \to \mathcal{P}[\![de]\!]\ (\psi \cup \{x_1, x_2\}) \\
\mathcal{P}_{br}[\![p \to de]\!]\ \psi & = p \to \mathcal{P}[\![de]\!]\ \psi
\end{array}
$$

Figure 4.2: Transformation Rules for Parallelisation

then the pattern variables representing its children are added to the set of parallelisable variables in order to allow for the explicit parallelisation of expressions in which they are used at a later point in the transformation.

Perhaps the most interesting of the presented transformation rules are the three which attempt to parallelise **let** statements. In the first, at least one parallelisable variable is used in the extracted expression and at least one parallelisable variable is also used in the **let** body. As both the extracted expression and the **let** body contain parallelisable variables it makes sense that the evaluation of the extracted expression, once it has been parallelised according to the transformation rules, should be sparked for parallel evaluation using the *par* strategy. Following this, the body of the **let** statement is also further parallelised according to the transformation rules.

In the second rule, there is at least one parallelisable variable contained in the extracted expression and no parallelisable variables are contained in the **let** body. As this rule may be reached upon traversing a series of nested

let statements whose extracted expressions have been sparked for parallel evaluation, it makes sense to mark the extracted expression for strict evaluation using the *pseq* strategy, after it has been parallelised according to the transformation rules. This may allow any previously sparked expressions to be evaluated to weak-head normal form, before evaluating the body of the **let** statement. As parallelisable variables are added to ψ upon encountering a *Join* constructor, by $\mathcal{P}_{br}$, it may be the case that the body of the **let** statement contains currently unknown parallelisable variables. As such, the parallelisation transformation is also applied to the body, in order to parallelise expressions which operate on currently unknown parallelisable variables.

In the final rule, neither the extracted expression nor the **let** body contain any parallelisable variables. The parallelisation algorithm only attempts to parallelise expressions based upon a known set of parallelisable variables, constructed while traversing an expression. As such, it makes sense to continue attempting to parallelise both the extracted expression and the body of the **let** expression, as they may contain further instances of well-partitioned data which are unknown at the current point.

These rules also allow for a situation in which the extracted expression contains no parallelisable variables, but the **let** body does: in such a situation, by the final rule, the parallelisation technique is applied to the the **let** body. This allows for expressions operating on known parallelisable variables in the **let** body to be extracted, as well as for the identification and parallelisation of expressions operating on currently unknown parallelisable variables.

It is worth noting that the presented transformation rules only introduce parallelism in **let** statements which operate upon *join*-lists, which are introduced as part of the generalisation phase of distillation. However, it may be the case, depending on the input program being distilled, that distillation may not produce a program containing **let** statements if no generalisation is required. In such a case, application of the parallelisation technique would result in no parallelism being added to the distilled program.

4.3 Example of Automatic Parallelisation

As an example of the application of this parallelisation technique, consider again the program $sumTree_{wp}$ defined in Chapter 3. Applying $\mathcal{P}$ to $sumTree_{wp}$ and $sumTree'_{wp}$ results in the functions $sumTree_{par}$ and $sumTree'_{par}$, the full derivations of which are shown in Figure 4.3, where any modifications to the program have been highlighted in red.

$$
\begin{array}{lcl}
sumTree_{par} & = & \mathcal{P}[\![\lambda x.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x \\
 & & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
 & & \Downarrow \\
sumTree_{par} & = & \lambda x.\ \mathcal{P}[\![\textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x \\
 & & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
 & & \Downarrow \\
sumTree_{par} & = & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad \mathcal{P}_{br}[\![Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
 & & \Downarrow \\
sumTree_{par} & = & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \mathcal{P}[\![\textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
 & & \Downarrow \\
sumTree_{par} & = & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad \mathcal{P}_{br}[\![Leaf'\ x \rightarrow x]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing
\end{array}
$$

Figure 4.3: Automatic Parallelisation of $sumTree_{par}$

While parallelising *sumTree*, the parallelisation technique is applied in a top-down fashion and first encounters a **case** expression containing two branches: one which matches with a *Singleton* and one which matches with

$$
\begin{array}{lcl}
sumTree_{par} & = & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad \mathcal{P}_{br}[\![Leaf' x \rightarrow x]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
sumTree_{par} & = & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow \mathcal{P}[\![x]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
sumTree_{par} & = & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
 & & \qquad \Downarrow \psi = \psi \cup \{l, r\} \\
sumTree_{par} & = & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x \\
 & & \quad Join\ l\ r \quad \rightarrow \mathcal{P}[\![\textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \{l, r\} \\
 & & \qquad\qquad \Downarrow \\
sumTree_{par} & = & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x \\
 & & \quad Join\ l\ r \quad \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ r'\ `par`\ sumTree'_{par}\ l\ r'
\end{array}
$$

Figure 4.2 (cont.): Automatic Parallelisation of $sumTree_{par}$

a *Join*. Though the *Singleton* contains no parallelisable variables itself, the parallelisation technique still attempts to parallelise its branch expression, as this may contain an instance of a currently unknown *join*-list whose evaluation can be parallelised. However, as no new parallelisable variables are identified while traversing the branch expression, it is left unmodified by the parallelisation algorithm.

The more interesting branch is the one that matches with a *Join* constructor, highlighted in red, with the data-components l and r. When parallelising this branch, as both l and r are instances of a *join*-list, they are added

$$
\begin{array}{lcl}
sumTree'_{par} & = & \mathcal{P}[\![\lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Leaf'\ x \rightarrow x + n \\
& & \qquad\qquad Node' \quad \rightarrow n \\
& & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
& & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r'[\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
sumTree'_{par} & = & \lambda x.\ \mathcal{P}[\![\lambda n.\ \textbf{case}\ x\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Leaf'\ x \rightarrow x + n \\
& & \qquad\qquad Node' \quad \rightarrow n \\
& & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
& & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r'[\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \mathcal{P}[\![\textbf{case}\ x\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Leaf'\ x \rightarrow x + n \\
& & \qquad\qquad Node' \quad \rightarrow n \\
& & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
& & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r'[\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
& & \quad \mathcal{P}_{br}[\![Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Leaf'\ x \rightarrow x + n \\
& & \qquad\qquad Node' \quad \rightarrow n[\![\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
& & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r'[\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \mathcal{P}[\![\textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Leaf'\ x \rightarrow x + n \\
& & \qquad\qquad Node' \quad \rightarrow n[\![\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
& & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r'[\![\ \varnothing
\end{array}
$$

Figure 4.1 (cont.): Automatic Parallelisation of $sumTree_{par}$

to the set of parallelisable variables (and are henceforth highlighted in red) so that expressions in which they are evaluated may be sparked for parallel evaluation. Following this, the branch expression is transformed according to the parallelisation algorithm. The branch expression consists of a **let** statement in which both the extracted expression and body use a parallelisable

$$
\begin{array}{lcl}
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \mathcal{P}[\![\textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Leaf'\ x \rightarrow x + n \\
& & \qquad\qquad Node' \quad \rightarrow n]\!]\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
& & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
& & \qquad\qquad \Downarrow \\
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad \mathcal{P}_{br}[\![Leaf' x \rightarrow x + n]\!]\ \varnothing \\
& & \qquad\qquad \mathcal{P}_{br}[\![Node' \quad \rightarrow n]\!]\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
& & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
& & \qquad\qquad \Downarrow \\
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Leaf'\ x \quad \rightarrow \mathcal{P}[\![x + n]\!]\ \varnothing \\
& & \qquad\qquad \mathcal{P}_{br}[\![Node' \rightarrow n]\!]\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
& & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
& & \qquad\qquad \Downarrow \\
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Leaf'\ x \quad \rightarrow x + n \\
& & \qquad\qquad \mathcal{P}_{br}[\![Node' \rightarrow n]\!]\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
& & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
& & \qquad\qquad \Downarrow \\
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Leaf'\ x \rightarrow x + n \\
& & \qquad\qquad Node' \quad \rightarrow \mathcal{P}[\![n]\!]\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
& & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing
\end{array}
$$

Figure 4.0 (cont.): Automatic Parallelisation of $sumTree_{par}$

variable. As such, the appropriate **let** rule is applied to the **let** statement and the **let** statement is therefore converted into a parallel **let** statement by sparking its extracted expressions for parallel evaluation using the *par* strategy (highlighted in red). The parallelisation algorithm attempts to further parallelise both the extracted expression and the body of the **let** statement

$$
\begin{array}{lcl}
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x + n \\
 & & \qquad\qquad Node' \quad \rightarrow \mathcal{P}[\![n]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x + n \\
 & & \qquad\qquad Node' \quad \rightarrow n \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
 & & \qquad\qquad \textbf{in}\ sumTree'\ l\ r']\!]\ \varnothing \\
 & & \quad \Downarrow \psi = \psi \cup \{l, r\} \\
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x + n \\
 & & \qquad\qquad Node' \quad \rightarrow n \\
 & & \quad Join\ l\ r \quad\ \rightarrow \mathcal{P}[\![\textbf{let}\ r' = sumTree'_{par}\ r\ n \\
 & & \qquad\qquad\ \ \textbf{in}\ sumTree'_{par}\ l\ r']\!]\ \{l, r\} \\
 & & \qquad\qquad \Downarrow \\
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Leaf'\ x \rightarrow x + n \\
 & & \qquad\qquad Node' \quad \rightarrow n \\
 & & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
 & & \qquad\qquad \textbf{in}\ r'\ `par`\ sumTree'_{par}\ l\ r'
\end{array}
$$

Figure 4.-1 (cont.): Automatic Parallelisation of $sumTree_{par}$

but they do contain any further expressions which should be parallelised. The parallelisation of $sumTree'$ is performed in an identical manner to that of $sumTree$.

Consider again supplying the well-partitioned *join*-list shown in Figure 3.13 as an argument to $sumTree_{par}$; the result of its evaluation would be as shown in Figure 4.0. At each $Join$ that is encountered, the *join*-list is split in half and the sum of the right child is calculated in parallel, using the *par* strategy, with the evaluation of the left child. As the sums of the left and right children are calculated separately, it makes sense that these calculations are parallelised.

It is worth noting again that the successful application of the paralleli-

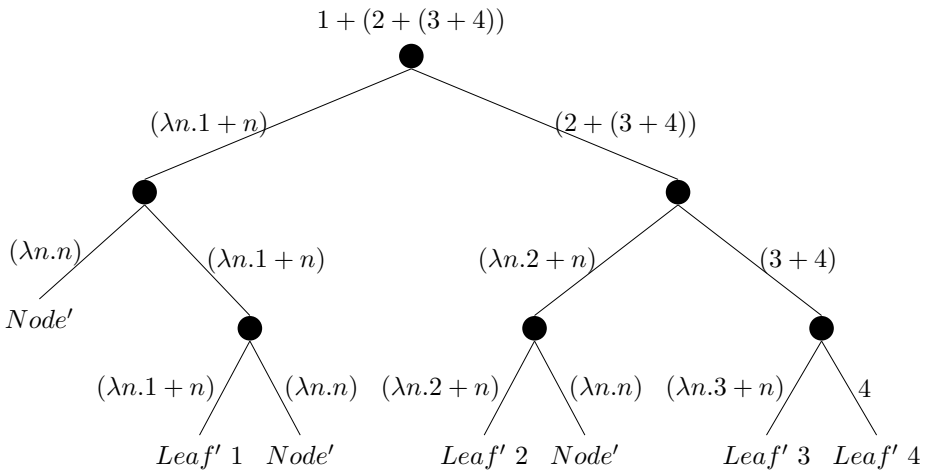


Figure 4.0: Evaluation of $sumTree_{par}$ on a Sample *Join*-List

sation technique is dependent on the output of the distillation step in the automatic partitioning technique, specifically that there are **let** statements introduced as part of a distillation generalisation step. For example the $sumTree_{par}$ function could be defined as follows:

$$\begin{aligned} sumTree_{par} = \lambda x.\ \textbf{case}\ x\ \textbf{of}& \\ Singleton\ x &\rightarrow \textbf{case}\ x\ \textbf{of} \\ & \qquad Leaf'\ x \rightarrow x \\ Join\ l\ r &\rightarrow sumTree'_{par}\ l\ (sumTree_{par}\ r) \end{aligned}$$

If the output of the automatic partitioning step was in this form, though it is defined for a well-partitioned *join*-list, the presented automatic parallelisation technique would not add any parallelism, and further transformation would be required to convert the given program into a form suitable for automatic parallelisation, or the parallelisation rules would need to be modified to suit such a form. In the above example, a further pre-parallelisation transformation could extract the components of the expression into a form that matches the output currently provided by the automatic partitioning transformation. However, the let-normal form output of distillation, as presented in Section 1.4.1, ensures that expressions such as the one above do not result from distillation.

4.4 Conclusion

In conclusion, this chapter has presented a novel, fully automatic parallelisation technique for functional programs defined on any data-type. Using the techniques presented in Chapter 3 in combination with the material presented in this chapter, the presented automatic parallelisation technique will take a program defined over a well-partitioned *join*-list, which as a consequence of distillation is in **let**-normal form, and convert it into an explicitly parallel program using GpH. The presented technique is therefore applicable to the full range of programs that distillation is applicable to; however, the technique may not prove worthwhile if the output of distillation does not include any **let** statements in which the extracted expression operates on a *join*-list. These **let** statements are generated as part of the generalisation stage of distillation, and this risk is partly alleviated due to the output form of distillation. Assuming that the output of distillation does contain **let** statements which evaluate *join*-lists, the presented parallelisation technique automatically converts these **let** statements into parallel **let** statements, by sparking their parallel evaluation. By defining a technique with which a developer can automatically parallelise programs, the difficulties associated with the parallelisation process have been removed from the developer, who can now develop software within the comfortable sequential paradigm and have equivalent parallel software derived automatically as needed. The presented automatic parallelisation technique forms the parallelising component of the proposed solution.

There are numerous parallelisation techniques which aim to solve the same problem as the presented technique. For example, Chin et al.'s [78, 13, 15] work on parallelisation via context-preservation also makes use of *join*-lists as part of its parallelisation process. Given a program, this technique derives two programs in *pre-parallel* form, which are then generalised. The resulting generalised 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 [78] presents an informal overview of the technique, a more concrete version was specified by Chin. et al. in [13, 15]. 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* [56] 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 parallelised.

Where existing automated parallelisation techniques [70, 69, 7, 25, 26, 36, 39, 40, 41, 60] are restrictive with respect to the form of their input programs and the types they are defined on, the presented parallelisation technique holds no such restrictions due to its use of the 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 parallelisation technique that is applicable to programs defined on any data-type.

The automated parallelisation technique presented in this chapter provides an answer to R.Q. 3 which asks: given a program defined over any data, can program transformation be used to automatically parallelise that program? By using the techniques presented in Chapter 3 and applying an explicit parallelisation transformation to their outputs, this chapter has shown that program transformation can be used to automatically parallelise functional programs defined over any data-type.

Chapter 5

Thresholding

5.1 Introduction

One obvious problem with the transformation rules, $\mathcal{P}$, as they are presented in Fig. 4.2, is that as any **let** bound expression containing parallelisable variables will be sparked for parallel evaluation, it may be the case that the parallelisation algorithm reaches such a fine level of granularity 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. The use of GpH can transparently improve the granularity of a parallelised program in which many sparks are created due to its underlying evaluate-and-die [66] evaluation model and the lazy thread creation [59] that this allows. However, providing an explicit means of granularity control can improve the behaviour of an ill-performing automatically parallelised program.

A solution to this problem is to control the level of granularity via the use of thresholding to govern the creation of new parallel processes. Such an approach can be quite effective when dealing with parallelised programs defined over inductive data-structures such as *join*-lists. Using thresholding to limit parallelisation in such a way means that once a certain threshold value, whether it be size, depth, height or some other value, has been reached, no further parallel processes will be created. Along with preventing the sparking of spurious processes, extending the automatic parallelisation technique to support such a thresholding approach will also give the developer a measure

of control over the parallelism obtained in the output program, as it makes sense to allow the developer to define the threshold.

This chapter presents an extension to the automatic parallelisation technique which allows for the number of sparks created by the parallelisation technique to be governed via the use of a user-specified thresholding function. The remainder of this chapter is structured as follows: Section 5.2 presents the modifications to both the automatic partitioning and automatic parallelisation techniques that are required in order to enable thresholding. Section 5.3 presents an example application of the automatic parallelisation technique with thresholding to the *sumTree* running example. Section 5.4 presents conclusions drawn from the material presented in this chapter.

5.2 Extending the Automatic Parallelisation Technique for Thresholding

Property 4 (Size Property) Given a well-partitioned *join*-list, the number of singletons it contains is referred to as its *size*.

The presented parallelisation technique can be extended to support thresholding by allowing the developer to supply a thresholding function to the parallelisation technique which will be used by the resulting parallel program to determine whether or not to create parallel sparks. The thresholding function will take three integer parameters: the first represents the size of the current *join*-list, as defined in Property 4, the second represents the current depth from the root of the *join*-list and the final parameter is the number of parallel cores available.

Based on these three parameters, the threshold function will return $True$ if parallelisation is to continue and $False$ if not. Defining the size property in such away will allow developers to relate their thresholding function to the size of the original data-structure. For example, consider again the $IntTree$ data-type, if an instance of $IntTree$ has a total size of 1000, then a well-partitioned *join*-list created by automatically partitioning that $IntTree$ will have a size of 1000. Once the result of the thresholding function for a given *join*-list becomes $False$, no more parallel sparks will be created beyond that point. If a thresholding function is not specified it is assumed that the developer simply wants as much parallelism from the program as possible.

Enabling such an approach requires several modifications to both the automatic partitioning technique and the automatic parallelisation technique. With respect to the automatic partitioning technique, the *join*-list data-type used so far in this thesis needs to be modified with an additional data-component in the *Join* constructor which will be used to determine whether or not to create parallel sparks for the children of that join. As a result of this, both the *partition* and *unpartition* functions defined in Chapter 3 will need to be redefined to suit this modified *join*-list. In addition to these modifications, the automatic parallelisation technique also needs to be modified to reflect the modified *join*-list and to use its extra data-component to govern the creation of sparks.

The remainder of this section is structured as follows: Section 5.2.1 presents the modifications that are required in the automatic partitioning technique and Section 5.2.2 presents the modifications that are required in the automatic parallelisation technique.

5.2.1 Modifications to the Automatic Partitioning Technique

$$
\begin{array}{lcl}
data\ JList\ a & ::= & Singleton\ a \\
 & | & Join\ Bool\ (JList\ a)\ (JList\ a)
\end{array}
$$

Figure 5.1: Modified *join*-list for Thresholded Programs

In order to enable thresholding, the first modification that must be made to the automatic partitioning technique is to the *join*-list that is used at its core. This modification requires the addition of a data-component, referred to as the *thresholding component*, to the *Join* constructor. The thresholding component is a boolean value which represents whether or not parallel sparks should be created at that point in a *join*-list. If the thresholding component is true, then any expressions that operate on its children should be sparked for parallel evaluation. If it is false, then no further sparks should be created for its children. The modified *join*-list data-type is defined as shown in Figure 5.1.

Assuming that the developer has supplied a thresholding function, *threshold*, which takes three integer parameters representing the size and depth of a *join*-list, as well as the number of available cores, this function

$$
\begin{aligned}
&partition\ = \lambda xs.\ partition'\ xs\ 1\ p \\
&\\
&partition' = \lambda x.\ \lambda d.\ \lambda p.\ \textbf{case}\ x\ \textbf{of} \\
&\qquad\qquad [x] \qquad\ \rightarrow Singleton\ x \\
&\qquad\qquad (y : ys) \rightarrow \textbf{case}\ split\ xs\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Pair\ l\ r \rightarrow \textbf{let}\ len = length\ xs \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad d' \ = d + 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad t \ = threshold\ len\ d\ p \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad l' \ = partition'\ l\ d'\ p \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad r' \ = partition'\ r\ d'\ p \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ Join\ t\ l'\ r'
\end{aligned}
$$

Figure 5.2: Modified *partition* Function Definition

can be used to calculate the value of the thresholding component for each *Join* that is created. As the size of a well-partitioned *join*-list is defined based on the number of singletons it contains, and this is the same as the size of the list created as a result of applying $flatten_\tau$ to a given data-type, it makes sense to calculate the value of the thresholding component when constructing the well-partitioned *join*-list. To allow for this, the *partition* function must be modified, both to suit the modified *join*-list data-type and to calculate the threshold value as it is constructing the *join*-list.

The definition of the modified version of *partition* is shown in Figure 5.2, where p represents the number of available processors. Rather than just partitioning the given *cons*-list into a *join*-list, *partition* now calls *partition*$'$, passing in the given list, an initial depth of 1 along with the number of available processors. At each recursive call of *partition*$'$ the value of the thresholding component is calculated using the user-defined thresholding function using the supplied size, depth, and the number of available processors. The depth of its children is then calculated and passed as an argument to the next recursive calls to *partition*$'$. The result of this function is a modified *join*-list where every join now contains a thresholding component which can be used to determine whether or not to create parallel sparks for expressions which operate on the join's children.

As an example of this, consider a user-defined thresholding function which creates parallel sparks up to a depth of 2 for a $TreeInt$. An example of the result of creating a modified *join*-list using this thresholding function is shown in Figure 5.3. For each join with a depth that is 2 or

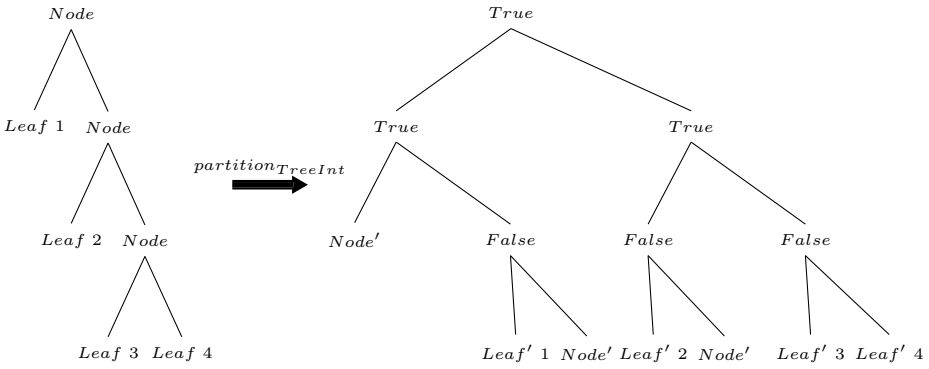


Figure 5.3: Example of Modified Partitioning for $TreeInt$

less, the thresholding component is true, implying that any expression that operates on the children of that join should be sparked for parallel evaluation. For each join with a depth greater than 2 the thresholding component is false, implying that expressions operating on the children of those joins should not be evaluated in parallel.

$$
\begin{aligned}
\mathit{unpartition} = \lambda x.\ &\textbf{case}\ x\ \textbf{of} \\
&\mathit{Singleton}\ x \rightarrow [x] \\
&\mathit{Join}\ t\ l\ r \ \ \rightarrow (\mathit{unpartition}\ l) +\!\!+ (\mathit{unpartition}\ r)
\end{aligned}
$$

Figure 5.4: Modified *unpartition* Function Definition

In addition to modifying the definition of *partition*, *unpartition* must be also be updated to suit the modified *join*-list data-type. This is a straightforward change and is shown in Figure 5.4. The modified definition of *unpartition* is almost identical to the previous version, except that upon encountering a join with a thresholding component, the thresholding component is ignored and the children of that join are recursively unpartitioned.

5.2.2 Modifications to the Automatic Parallelisation Technique

Along with modifying the automatic partitioning technique, the automatic parallelisation technique must also be updated to reflect the modified *join*-list data-type, and make use of the thresholding component to govern the creation of parallel sparks. In fact, only one rule needs to be updated to suit the modified *join*-list: the rule that parallelises the branches of a **case**

expression, $\mathcal{P}_{br}$.

$$
\begin{array}{lll}
\mathcal{P}_{br}[\![Join\ t\ l\ r \rightarrow de]\!]\ \psi = & & \\
\qquad Join\ t\ l\ r \rightarrow & \textbf{case}\ t\ \textbf{of} & \\
& True\ \rightarrow \mathcal{P}[\![de]\!]\ (\psi \cup \{l, r\}) & \\
& False \rightarrow \mathcal{P}[\![de]\!]\ \psi & \\
 & & \\
\mathcal{P}_{br}[\![p \rightarrow de]\!]\ \psi & = p \rightarrow \mathcal{P}[\![de]\!]\ \psi &
\end{array}
$$

Figure 5.5: Transformation Rules for Parallelisation with Thresholding

The modification to the parallelisation rules is quite straightforward and is as shown in Figure 5.5. Where the initial parallelisation rules simply attempted to extract as much parallelisation from the given program as possible, the modified parallelisation rules now use the thresholding component of the modified *join*-list to determine whether or not to spark an expression for parallel evaluation. This is accomplished by generating an expression in which the children of that join have been added to the set of parallelisable variables along with an expression in which they have not. These expressions are then embedded as the branch expressions of a **case** statement which performs a runtime test against the thresholding component to determine which branch to select. This test is performed at runtime as it is not possible to assume knowledge of the data ahead of runtime, and the well-partitioned *join*-list is not created until runtime.

In contrast with the previous parallelisation rules, upon encountering a *Join*, rather than just creating an expression where its children are automatically parallelised, the modified parallelisation technique now embeds a further **case** statement into the program being parallelised. At runtime, the thresholding component is checked and if true, the branch expression will be evaluated in which expressions operating on the children of that join will be sparked for parallel evaluation. If the thresholding component is false, then the branch expression will be evaluated in which expressions operating on the children of that join will not be sparked for parallel evaluation.

5.3 Example of Automatic Parallelisation with Thresholding

As an example of the application of the modified parallelisation technique, consider again the program $sumTree_{wp}$ defined in Chapter 3. Assuming the user has supplied a thresholding function which is used to set the thresholding component of each node of the well-partitioned *join*-list, then applying the modified version of $\mathcal{P}$ to $sumTree_{wp}$ and $sumTree'_{wp}$ results in the functions $sumTree_{par}$ and $sumTree'_{par}$, as shown in Figure 5.6.

$$
\begin{array}{lcl}
sumTree_{par} & = & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad Leaf'\ x \rightarrow x \\
 & & \quad Join\ t\ l\ r \quad \rightarrow \textbf{case}\ t\ \textbf{of} \\
 & & \qquad True\ \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ r'\ \text{`}par\text{`}\ sumTree'_{par}\ l\ r' \\
 & & \qquad False \rightarrow \textbf{let}\ r' = sumTree_{par}\ r \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r'
\end{array}
$$

$$
\begin{array}{lcl}
sumTree'_{par} & = & \lambda x.\ \lambda n.\ \textbf{case}\ x\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad Leaf'\ x \rightarrow x + n \\
 & & \qquad Node' \quad \rightarrow n \\
 & & \quad Join\ t\ l\ r \quad \rightarrow \textbf{case}\ t\ \textbf{of} \\
 & & \qquad True\ \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
 & & \qquad\qquad \textbf{in}\ r'\ \text{`}par\text{`}\ sumTree'_{par}\ l\ r' \\
 & & \qquad False \rightarrow \textbf{let}\ r' = sumTree'_{par}\ r\ n \\
 & & \qquad\qquad \textbf{in}\ sumTree'_{par}\ l\ r'
\end{array}
$$

Figure 5.6: Automatically Parallelised Version of $sumTree_{par}$ with Thresholding

The main difference between the version of $sumTree_{par}$ resulting from the modified parallelisation technique and the original technique is in how it handles branches where the pattern is a modified join. Upon encountering a $Join$, the automatically parallelised version of $sumTree$ now first checks the thresholding component of that $Join$. If the thresholding component is true, then the branch is evaluated in which expressions operating on its children (as in the version resulting from the original parallelisation technique) will be sparked for parallel evaluation. If the thresholding component is false, then the branch is evaluated in which expressions operating on the children

of the join are not sparked for parallel evaluation.

By allowing the developer to specify a thresholding function, and then using that function to generate the thresholding components of a modified *join*-list while partitioning, the modified parallelisation technique has allowed for a parallelised program to be derived in which thresholding is now used to govern the creation of parallel sparks. As the developer specifies the thresholding function, this allows the developer to have some measure of control over the automatically parallelised program and the granularity of the parallelism it creates. Up until the point that the thresholding function would return false for a *join*-list, expressions which operate on the children of that *join*-list will be sparked for parallel evaluation. From the point that the thresholding function returns false, no more parallel sparks are created.

5.4 Conclusion

In conclusion, this chapter has presented an extension to the automatic parallelisation technique presented in Chapter 4 which allows the developer to control the granularity of the automatically generated parallelism, in order to prevent the creation of trivial parallel sparks. This should allow a developer to tune the parallel behaviour of their automatically parallelised program, via the thresholding function which they must supply in order to avail of thresholding. Totoo et al [79] presents a comprehensive overview of different thresholding strategies that could be used by a developer when defining their thresholding function, as well as a heuristic based thresholding approach.

The thresholding function is defined over the size, depth of a *join*-list and number of available cores and as such allows the developer a significant degree of control over the point at which parallel processes stop being created. Another approach would be to remove all control from the developer and enable the parallelisation technique to govern spark creation based on some arbitrary thresholding value without any input from the developer. For example, it would also be possible to threshold based upon some generic cut-off point, such as if the height, width or size of the *join*-list currently being evaluated falls below or exceeds some arbitrary value.

However, selecting or generating such an arbitrary value is a complex problem, as what works best for one parallel program may not work well for another program. Even if the data that is evaluated by the program were

known by the parallelisation technique, it would be very complicated to measure the actual work involved in evaluating one *join*-list versus another and generate an appropriate thresholding value based upon this measurement. By allowing the developer to specify the thresholding function to be used, the technique presented in this chapter allows the developer to control the parallelism involved and to tailor the evaluation of the parallel program to suit the program and data involved in order to obtain the best performance.

Chapter 6

Examples of Automatic Parallelisation

Benchmark Name	**Data-Type**
Leftmost Odd Number	*Cons*-List
Sum of Bigger Numbers	*Cons*-List
Sum of Squares	*Cons*-List
Power Tree	Binary Tree
Maximum Prefix Sum	*Cons*-List
Maximum Segment Sum	*Cons*-List

Table 6.1: Benchmark Programs

This chapter presents examples of the application of the automatic parallelisation technique to the benchmark programs which will be used to evaluate the technique. Each benchmark program is defined using a *cons*-list of numbers, with the exception of Power Tree which is defined on a binary tree, and is presented in its sequential form along with the results of applying the automatic parallelisation process to the sequential form.

The remainder of this chapter is structured as follows: Section 6.1 - Section 6.6 present the application of the parallelisation technique to the each of the benchmark programs, as defined in Table 6.1. Finally, Section 6.7 presents conclusions drawn from the material presented in this chapter.

6.1 Leftmost Odd Number

Given a data-structure containing an ordered set of numbers, the function $leftMostOdd$ returns the leftmost odd number occurring in that set. A version defined for a list of integers, $ListInt$ is shown in Figure 6.1.

$$
\begin{array}{l}
leftMostOdd\ xs \\
\textbf{where} \\
leftMostOdd\ =\ \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \qquad\quad \rightarrow Nothing \\
\qquad\qquad\qquad\qquad Cons\ x\ xs \rightarrow \textbf{case}\ odd\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad True\ \ \rightarrow Just\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad False \rightarrow leftMostOdd\ xs
\end{array}
$$

Figure 6.1: Sequential Definition of $leftMostOdd$ Operating on Unpartitioned Data

Application of the automatic parallelisation technique to the definition of $leftMostOdd$ results in the parallel program, $leftMostOdd_{par}$, shown in Figure 6.2.

$$
\begin{array}{l}
leftMostOdd_{par}\ xs\ Nothing \\
\textbf{where} \\
leftMostOdd_{par}\ =\ \lambda xs.\ \lambda o.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Nil' \qquad \rightarrow o \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Cons'\ x \rightarrow \textbf{case}\ odd\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad True\ \ \rightarrow Just\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad False \rightarrow o \\
\qquad\qquad\qquad\qquad Join\ l\ r \qquad \rightarrow \textbf{let}\ r' = leftMostOdd_{par}\ r\ o \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ r'\ \text{`}par\text{`}\ leftMostOdd_{par}\ l\ r'
\end{array}
$$

Figure 6.2: Parallelised Version of $leftMostOdd$ Operating on *Well-Partitioned* Data

Automatically parallelising the sequential version of $leftMostOdd$ has resulted in an equivalent parallel program, defined on a well-partitioned *join*-list. The resulting parallel program creates a parallel spark when evaluating the right child of a $Join$, which may be evaluated in parallel with the sequential evaluation of the left child. However, though there is an opportunity created for the parallel evaluation of the right child, there is a dependency

between the evaluations of the left and right child, as the evaluation of the left child requires the result of the evaluation of the right. As a result, the evaluation of the right child may be performed as part of the sequential evaluation of the left before it has been evaluated in parallel, which could cause the created spark to become fizzled, though this is unlikely to cause problems as the accumulating parameter is not needed until the end of the recursion.

6.2 Sum of Bigger Numbers

$$
\begin{array}{l}
sumBig\ xs \\
\textbf{where} \\
sumBig \ = \ \lambda xs.\ biggers\ xs\ (sum\ xs) \\
\\
biggers \ = \ \lambda xs.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \qquad\quad \rightarrow 0 \\
\qquad\qquad\qquad\qquad Cons\ x\ xs \rightarrow \textbf{case}\ x > (s - x)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad True\ \ \rightarrow x + biggers\ xs\ (s - x) \\
\qquad\qquad\qquad\qquad\qquad\qquad False \rightarrow biggers\ xs\ (s - x)
\end{array}
$$

Figure 6.3: Sequential Definition of *sumBig* Operating on Unpartitioned Data

Given a data-structure containing an ordered sequence of numbers, the function *sumBig* calculates the sum of the *bigger* numbers in that sequence, where a bigger number is defined as one that is larger than the sum of the numbers following it in a sequence. A version defined for a list of integers, *ListInt* is shown in Figure 6.3.

Automatically parallelising the sequential version of *sumBig* results in the definition of $sumBig_{par}$, as defined in Figure 6.4, which evaluates well-partitioned data. As part of the automatic parallelisation process, distillation is used to transform the original program into one defined on well-partitioned data. While performing this transformation, distillation also optimises the resulting program, and as part of these optimisations, *common sub-expression elimination* has been performed on *sumBig*, where the multiple uses of $(s - x)$ have been extracted to a parameter that is passed on each recursive call to $sumBig_{par}$. Common sub-expression elimination is a

```
sumBig_par xs 0 0
where
sumBig_par =
   λxs. λs. λb. case xs of
                  Singleton x → case x of
                                  Nil'      → b
                                  Cons' x → case x > s of
                                              True  → x + b
                                              False → b
                  Join l r      → let r'= sum r s
                                  in r' `par` let r''= sumBig_par r s b
                                              in r'' `par` sumBig_par l r' r''

sum           = λxs. λs. case xs of
                  Singleton x → case x of
                                  Nil'      → s
                                  Cons' x → x + s
                  Join l r      → let r'= sum r s
                                  in r' `par` sum l r'
```

Figure 6.4: Parallelised Version of *sumBig* Operating on *Well*-Partitioned Data

commonly used optimisation that is used to eliminate multiple calculations of the same expression.

During the evaluation of the parallelised program, on encountering a join, both the functions $sumBig_{par}$ and *sum* create parallel sparks when evaluating the right child of that join. These sparks may be evaluated in parallel with the evaluation of the left child of the join. However, as with $leftMostOdd_{par}$, there are also dependencies between the evaluations of the left and right child. As a result, in both functions, the evaluation of the right child may be performed while sequentially evaluating the left child, rather than in parallel, which could result in the created sparks fizzling. Again, this shouldn't be an issue as this dependency isn't required until the end of the recursive evaluation.

```
sumSquares xs
where
sumSquares = λxs. case xs of
                    Nil       → 0
                    Cons x xs → x * x + sumSquares xs
```

Figure 6.5: Sequential Definition of *sumSquares* Operating on Unpartitioned Data

6.3 Sum of Squares

Given a data-structure containing a set of numbers, the function *sumSquares* calculates the sum of the squares of all the numbers in that set. A version defined for a list of integers, *ListInt*, is shown in Figure 6.5. An automatically parallelised version of this, $sumSquares_{par}$, is shown in Figure 6.6.

```
sumSquares_par xs 0
where
sumSquares_par = λxs. λs. case xs of
                            Singleton x → case x of
                                            Nil'     → 0
                                            Cons' x  → x * x + s
                            Join l r    → let r'= sumSquares_par r s
                                          in r' `par` sumSquares_par l r'
```

Figure 6.6: Parallelised Version of *sumSquares* Operating on *Well-Partitioned Data*

The result of applying the automatic parallelisation technique to *sumSquares* is shown in Figure 6.6, in which the evaluation of the right child of a join is sparked in parallel with that of the left child. Again, like the previous examples, there is a dependency between these evaluations, which may cause some of the created sparks to fizzle.

6.4 Power Tree

Unlike the other example programs, given a *binary-tree* containing a set of numbers which are placed at its leaves, the function *powerTree* calculates

```
powerTree xs
where
powerTree = λxs. case xs of
                   Leaf x   → x^3
                   Node l r → powerTree l + powerTree r
```

Figure 6.7: Sequential Definition of *powerTree* Operating on Potentially Unpartitioned Data

the sum of the cubes of all the numbers in that set, as shown in Figure 6.7. An automatically parallelised version of this, $powerTree_{par}$, is shown in Figure 6.8.

```
powerTree_par xs
where
powerTree_par = λxs. case xs of
                       Singleton x → case x of
                                       Leaf' x → x^3
                       Join l r    → let r' = powerTree_par r
                                     in r' `par` powerTree'_par l r'

powerTree'_par = λxs. λn. case xs of
                            Singleton x → case x of
                                            Leaf' x → x^3 + n
                                            Node'   → n
                            Join l r    → let r' = powerTree'_par r n
                                          in r' `par` powerTree'_par l r'
```

Figure 6.8: Sequential Definition of *powerTree* Operating on *Well-Partitioned* Data

The use of distillation as part of the parallelisation technique has resulted in a definition of *powerTree* which uses a tail-recursive auxiliary function to calculate its result. On encountering a join, this automatically parallelised version of *powerTree* creates parallel sparks to evaluate the right child. These sparks may then be evaluated in parallel with the sequential evaluation of the left child. As with previously parallelised examples, this results in dependencies between these two evaluations, which can contribute to the number of fizzled sparks that occur in a parallel run of this program.

6.5 Maximum Prefix Sum

```
mps xs
where
mps    = λxs. case xs of
                Nil         → 0
                Cons x xs → mps' xs x

mps'   = λxs. λn. case xs of
                  Nil         → n
                  Cons x xs → max n (max (n + x) (mps' xs (n + x)))
```

Figure 6.9: Sequential Definition of *mps* Operating on Unpartitioned Data

Given a data-structure containing an ordered set of numbers, the function *mps* calculates the maximum sum of all prefix subsets. A version defined for a list of integers, *ListInt* is shown in Figure 6.9. This benchmark was also parallelised in [61]. Application of the automatic parallelisation process to the definition of *mps* results in the parallel program, mps_{par}, shown in Figure 6.10.

```
mps_par xs 0
where
mps_par = λxs. λn. case xs of
                     Singleton x → case x of
                                     Nil'     → n
                                     Cons' x → max x (n + x)
                     Join l r    → let r'= mps_par r n
                                   in r' `par` mps_par l r'
```

Figure 6.10: Parallelised Version of *mps* Operating on *Well*-Partitioned Data

On encountering a join, the automatically parallelised version of *mps* creates parallel sparks to evaluate the right child. These sparks may then be evaluated in parallel with the sequential evaluation of the left child. It is worth noting that where the original definition of *mps* is defined in terms of the function *mps'*, the parallelised version only requires one function definition. Once again, this modification is due to distillation and the transformations and optimisations that it performs. As a result of this, distillation

has produce a single function definition, as the original version of *mps* can be defined in terms of *mps′* as *mps′* 0 *xs*, and can therefore the need for a separate function has been eliminated.

6.6 Maximum Segment Sum

```
mss xs
where
mss      = λxs. maxList (map sum (segments xs))

maxList  = λxs. case xs of
                  Nil       → 0
                  Cons x xs → maxList′ x xs

maxList′ = λm. λxs. case xs of
                      Nil       → m
                      Cons x xs → maxList′ (max m x) xs

sum      = λxs. case xs of
                  Nil       → 0
                  Cons x xs → x + sum xs

inits    = λxs. case xs of
                  Nil       → Cons Nil Nil
                  Cons y ys → Cons xs (inits (init xs))

tails    = λxs. case xs of
                  Nil       → Cons Nil Nil
                  Cons y ys → Cons xs (tails ys)

segments = λxs. concat (map inits (tails xs))
```

Figure 6.11: Sequential Definition of *mss* Operating on Unpartitioned Data

Given a list containing a set of numbers, the function *mss* calculates the sum of the contiguous subset whose elements have the largest sum amongst all such subsets. A version defined for a list of integers, *ListInt* is shown in Figure 6.11, where the function *concat* concatenates a list of lists and the function *init*, given a *cons*-list, returns a *cons*-list containing all but the last element of the given *cons*-list. This program was also parallelised as part of the parallelisation techniques presented in [17, 41].

Automatically parallelising the sequential version of *mss* has resulted in an equivalent parallel program, defined for well-partitioned data and con-

```
mss_par xs 0 0
where
mss_par =     λxs. λm. λs. case xs of
                             Singleton x → case x of
                                             Nil'      → m
                                             Cons' x → max (s + x) m
                             Join l r      → let r' = mss_par r m s
                                             in r' `par` let r'' = f1 r s
                                                         in r'' `par` mss_par l r' r''

f1      = λxs. λm. case xs of
                     Singleton x → case x of
                                     Nil'      → m
                                     Cons' x → max x (x + m)
                     Join l r      → let r' = f1 r m
                                     in r' `par` f1 l r'
```

Figure 6.12: Parallelised Version of *mss* Operating on *Well*-Partitioned Data

sisting of two functions, mss_{par} and $f1$, as shown in Figure 6.12. A full working of the parallelisation of *mss* is shown in Appendix D. It is obvious when comparing the original and parallelised version of *mss* that the automatic parallelisation process has had a drastic impact on how *mss* is calculated. All of the functions that were required to calculate *mss* originally have either had their computation removed or inlined into the definition of mss_{par}. Where the original definition had a cubic run-time complexity, the version result from the automatic parallelisation technique has a linear run-time complexity, which shows the power of using distillation as part of the parallelisation technique.

6.7 Conclusion

In conclusion this chapter has presented examples of the application of the automatic parallelisation technique to several programs. Each sequential program was presented along with the resulting parallel program automatically generated by the parallelisation technique. For each example program, the parallelisation technique has derived an equivalent parallel program defined on well-partitioned data in which expressions that operate on well-partitioned data are evaluated in parallel. In a several of these programs, the use of the distillation to transform the original program into one which evaluates well-partitioned data has resulted in some interesting optimisa-

tions in the resulting program, such as common sub-expression elimination, expression inlining and the introduction of accumulating parameters and tail-recursive function definitions. Each of these optimisation offers efficiency improvements over the original program, even before parallelisation has been added.

Though, the parallelisation technique has automatically parallelised each of the benchmark programs, each of the programs does have dependencies between the parallel sparks that it will create. A consequence of such dependencies is that some parallel sparks may be evaluated sequentially before they have been evaluated in parallel, causing these sparks to become fizzled. However, as these dependencies are generally not required until then end of the recursive calls in which they are present, they should not have much of an impact of the parallel evaluation of the benchmark programs.

Chapter 7

Evaluation

This chapter presents a thorough empirical evaluation of the speedups gained by the automatic parallelisation technique presented in this thesis. In order to evaluate the merits of the parallelisation technique, a number of benchmark programs have been identified, and are as defined in Chapter 6. The automatically parallelised versions of these benchmarks use a well-partitioned *join*-list as their data-type which has been automatically defined according to the technique defined in Chapter 3.

Once the benchmark programs have been hand-parallelised and had the automatic parallelisation technique successfully applied, they have been evaluated in a parallel environment, as defined in Section 1.4.2, with respect to their parallel performance. The results of evaluating the sequential versions of the benchmark programs are used as a baseline for comparison with those defined using the automatic parallelisation process. The execution times of the sequential versions of the benchmarks are used to determine the *absolute* speedups obtained by both hand parallelised and automatically parallelised versions. The obtained absolute speedups are then used to compare both techniques. Absolute speedups are used in this evaluation to compare the *total* execution times of the parallelised programs with the *total* execution times of their sequential counterparts, including overheads such as reading inputs from file amongst others. This allows for an evaluation of the actual speedups that would be observed in a real-world situation, something which comparison of the *relative* speedups (the speedups optained by comparing just the parallelised code with its sequential counterpart), would not show.

The main focus of this evaluation is the differences between hand paral-

lelised and automatically parallelised versions of the benchmark programs. This comparison has allowed for an in-depth analysis of the behaviour of programs derived using the automatic parallelisation technique when compared to their hand parallelised counterparts.

In addition to these benchmarks, a version of the automatically parallelised version which uses a thresholding function to govern spark creation is also benchmarked. A common practice that many developers use is to constrain the number of threads their parallel programs create to be the number of cores on their machine, and divide the work they perform between these threads. In a case where constant work may not be guaranteed for each core, it is again common practice to create slightly more threads than there are cores to provide work to processors that become idle. The thresholding function used in this evaluation is an approximation of this technique.

It is worth noting at this point, that as this chapter evaluates the overall performance of the automatic parallelisation technique, any speedups resulting from the parallelisation technique will include those due to the optimisations that distillation performs. In addition to this, it is worth mentioning that this evaluation does not isolate the speedups garnered by replacing the data-type used in the sequential program with a well-partitioned *join*-list. This is a result of the evaluation being focused on the absolute speedups garnered by the automatic parallelisation technique as a whole.

The remainder of this chapter is structured as follows: Section 7.1 describes the process involved in benchmarking and obtaining the necessary metrics related to a benchmark program. Section 7.2 presents an evaluation of the results of the benchmarking process. Section 7.3 presents conclusions drawn from the material presented in this chapter.

7.1 Benchmark Process

Given a sequential benchmark program, there are several steps which must be followed in order to complete a benchmark:

1. Define an equivalent hand-parallelised version.
2. Derive an equivalent automatically parallelised version using the techniques presented in this thesis.
3. Generate an appropriate input for the benchmark program.

4. Execute the sequential version and record appropriate metrics.
5. Execute the hand-parallelised version and record appropriate metrics.
6. Execute the automatically parallelised version and record appropriate metrics.
7. Execute the automatically parallelised version using a thresholding function and record appropriate metrics.

The material presented in this thesis, specifically Chapter 3 and Chapter 4 describes the process by which a given benchmark program can be automatically parallelised. Section 7.1.1 describes the process by which a given benchmark program is parallelised by hand. Section 7.1.2 describes the process by which an input is generated for each benchmark program. Section 7.1.3 describes the thresholding function that is used when evaluating automatically parallelised programs that limit the number of sparks that are created. Section 7.1.4 describes the process by which a sequential program is benchmarked and how metrics related to the sequential behaviour of the benchmark are obtained and recorded. Section 7.1.5 describes the process by which parallel programs, both hand parallelised and automatically parallelised, are benchmarked and how metrics related to the parallel behaviour of the benchmark are obtained and recorded.

7.1.1 Defining Hand-Parallelised Benchmark Programs

In order to parallelise a given sequential program by hand, each function used by the program is examined individually, using a pattern based approach, to determine whether it can be defined as a *reduction* or a *mapping*, two commonly used patterns of parallel programming. While this may not always yield optimal parallel programs it is expected that users of the presented parallelisation technique will not be experts in the field of parallel software and this approach is an approximation of what such a user might define by hand themselves.

In the case of the benchmark programs used as part of this evaluation, a reduction will typically take a list and use a reduction operator to reduce the values in the list into a final result. A mapping however, will take a list and apply a function to each value in the list and return a new list containing the results of these applications.

For example, the function *sum*, shown below, is a reduction which takes a list and uses the reduction operator + to add the first element of the list to the sum of the remainder of the list.

$$
\begin{aligned}
sum = \lambda xs.\ &\textbf{case}\ xs\ \textbf{of} \\
&Nil \rightarrow 0 \\
&Cons\ x\ xs \rightarrow x + sum\ xs
\end{aligned}
$$

Reductions typically lend themselves well to a *divide-and-conquer* parallel implementation, in which a given list is split in half and a result is calculated for each half of the list. Upon calculating the result for each half of the list, the reduction operator is used to calculate the correct final result for the given list. A *divide-and-conquer* implementation of *sum* is shown in Figure 7.1, in which the list xs is split in half and the sum of each half is calculated recursively in parallel, before being added together to calculate the total sum of xs.

$$
\begin{aligned}
sum = \lambda xs.\ &\textbf{case}\ xs\ \textbf{of} \\
&Nil \rightarrow 0 \\
&Cons\ x\ xs \rightarrow \textbf{case}\ split\ xs\ \textbf{of} \\
&\qquad Pair\ l\ r \rightarrow \textbf{let}\ r' = sum\ r \\
&\qquad\qquad \textbf{in}\ r'\ `par`\ \textbf{let}\ l' = sum\ l \\
&\qquad\qquad\qquad \textbf{in}\ l'\ `pseq`\ x + l' + r'
\end{aligned}
$$

Figure 7.1: Hand-Parallelised Version of *sum* Using Divide-And-Conquer Task Parallelism

As another example, the function *squareList*, shown below, is a mapping, in which each element of a list is squared and used to create a list containing the squares of the numbers in the original list.

$$
\begin{aligned}
squareList = \lambda xs.\ &\textbf{case}\ xs\ \textbf{of} \\
&Nil \rightarrow Nil \\
&Cons\ x\ xs \rightarrow Cons\ (x * x)\ (squareList\ xs)
\end{aligned}
$$

Mappings, by their nature, are well suited to being evaluated using a data-parallel implementation, in which each application of the given function to an element of the list is performed in parallel and the results of these parallel evaluations are used to construct the new list which is returned as

a result. As a result a data-parallel implementation of *squareList* can be defined as shown in Figure 7.2, in which each element of the list is squared in parallel before being used to construct a new list. This is done by sparking the evaluation of the computation on the head of the list in parallel with the computation of the mapping on the remainder of the list. Evaluation order is constrained by using *pseq* to allow the application of the mapping to each element to be sparked for parallel evaluation before constructing the resultant list.

```
squareList = λxs. case xs of
                    Nil         → Nil
                    Cons x xs → let x'= x * x
                                in x' `par` let xs'= squareList xs
                                    in xs' `pseq` (Cons x'xs')
```

Figure 7.2: Hand-Parallelised Version of *squareList* using Data-Parallelism

Given a program to hand-parallelise, any functions it contains which can be defined as reductions are parallelised in a similar fashion to the definition of *sum*. Any further functions a program may contain that are similar to mappings are parallelised using an approach similar to that of *squareList*. The hand parallelised versions of the benchmark programs are presented in Appendix C.

7.1.2 Generating Inputs For Benchmark Programs

An important part of the benchmarking process is ensuring that each version of a given benchmark evaluates the same input as the others on any particular run in order to ensure fairness. Also, each benchmark should be evaluated over varying input sizes in order to ensure that an accurate description of the its behaviour, in both sequential and parallel forms, is obtained. To allow for this, as part of the benchmarking suite, for each run of a benchmark for a given input size, n, a file is generated containing n randomised data points appropriate to the benchmark, be they integers or floating point numbers. This file is then duplicated in the locations of the sequential, hand parallelised and automatically parallelised versions of the given benchmark.

Each benchmark version has been set up to allow for this, and at run-time the inputs to the various benchmark versions are read from this file and converted into an appropriate format for the program being benchmarked.

For sequential and hand parallelised benchmark programs, this means that the data in the input file is converted into a *cons*-list or binary tree. For automatically parallelised benchmark programs, this means that the data in the input file is converted into an appropriate well-partitioned *join*-list, using the partitioning techniques defined in Chapter 3.

7.1.3 Evaluation Threshold Function

When evaluated without a threshold, programs resulting from the automatic parallelisation technique simply attempt to create as much parallelism as possible. However, as pointed out in Chapter 4, this may have a negative impact on the efficiency of these programs as too many sparks may be introduced at too fine a granularity to result in an optimal parallel program. In an attempt to solve this, a thresholding technique was introduced in Chapter 5 as an extension to the automatic parallelisation technique which allows a developer to specify a thresholding function in order to govern the creation of sparks, which should result in improved parallel behaviour.

$$threshold = \lambda size.\ \lambda depth.\ \lambda numcores.\ depth \leq (\lceil \log numcores \rceil + 1)$$

Figure 7.3: Thresholding to Approximately the Number of Available Cores

A common practice used by many developers when developing parallel programs is to create as many threads as their are cores on their machines. Sometimes, as a consequence of this, some cores may not have as much work as other cores and become idle upon completing their workload. In such cases, slightly more parallel processes can be created than there are threads to offset the loss in productivity due to idle cores. The thresholding function that is benchmarked as part of this evaluation is an approximation of such an approach. At a high level, it create slightly more sparks than there are cores available. This is accomplished by determining the depth of a tree that corresponds, roughly, to the number of available cores, according to the thresholding function presented in Figure 7.3.

7.1.4 Benchmarking Sequential Programs

Sequential programs are compiled using the following command, where `PROGRAM_NAME` is a placeholder for the name of the benchmark program: `ghc --make -fforce-recomp -rtsopts PROGRAM_NAME`. The use of

the `-fforce-recomp` compilation flag ensures that the compiled executable is always based on the latest version of the benchmark code. The use of the `-rtsopts` compilation flag allows the benchmarking suite to instruct the Haskell runtime to print statistics about the run of each benchmark program such as its execution time.

```
55,936 bytes allocated in the heap
 3,912 bytes copied during GC
36,592 bytes maximum residency (1 sample(s))
12,560 bytes maximum slop
1 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0  0 colls,     0 par    0.00s    0.00s     0.0000s    0.0000s
Gen  1  1 colls,     0 par    0.00s    0.00s     0.0002s    0.0002s

INIT    time    0.00s  (  0.00s elapsed)
MUT     time    0.00s  (  0.00s elapsed)
GC      time    0.01s  (  0.01s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    0.02s  (  0.02s elapsed)

%GC     time       6.6%  (22.0% elapsed)

Alloc rate    766,246,575 bytes per MUT second

Productivity  88.2% of total user, 204.9% of total elapsed
```

Figure 7.4: Example of Haskell Runtime Statistics for Sequential Programs

Once the necessary executable has been compiled, the benchmarking suite then generates an appropriate input for the given benchmark. Following this, the sequential version is then executed using the following command: `PROGRAM_NAME +RTS -sstderr`. The use of the `-sstderr` runtime flag instructs the Haskell runtime to print statistics about the behaviour of the program to `stderr`. An example of the statistics output by the Haskell runtime for sequential programs is shown in Figure 7.4, where the statistics that are recorded as part of the benchmarking process are highlighted in red. As can be seen from Figure 7.4, the Haskell runtime produces many statistics of which the total elapsed run-time (`0.02s`) is of particular interest.

At the end of a successful benchmark of a sequential program, these statistics are recorded, along with the size of the input and are stored in a database for evaluation and comparison. The total elapsed run-time of the

sequential version will be used as the baseline for evaluating the run-time performance of the parallelised versions of the given benchmark. Comparing the total elapsed run-time of a sequential program with that of an automatically parallelised program will allow the merits or failings of the automatic parallelisation technique to be shown.

7.1.5 Benchmarking Parallel Programs

Benchmarking the parallel behaviour of a given program is performed in a similar fashion to that of the sequential, except that it is applied to both the hand parallelised and automatically parallelised versions of the benchmark. These parallel programs are compiled using the following command: `ghc --make -fforce-recomp -rtsopts -threaded PROGRAM_NAME`. The addition of the `-threaded` compilation flag ensures that the compiled executable can be run in a parallel environment.

Once the necessary executables have been compiled the benchmarking suite generates an appropriate input. Following this, the parallel versions are then executed in a way that makes use of all possible numbers of cores on the benchmark machine, from 2 cores to n cores, where n represents the total number of cores on the machine, using the following command: `PROGRAM_NAME +RTS -sstderr -Nt`. The use of the `-Nt` runtime flag instructs the Haskell runtime to use `t` cores for the parallel execution of the benchmark run. Again, the runtime flag `-sstderr` is used to print statistics about the behaviour of the program to `stderr`. An example of the statistics output by the Haskell runtime for parallel programs is shown in Figure 7.5, where the statistics that are recorded as part of the parallel benchmarking process are highlighted in red. As can be seen from Figure 7.5, the Haskell runtime now produces the same statistics as for a sequential program but adds many extra statistics related to the parallel behaviour of the program, including the number of sparks created and converted, amongst others.

Converted sparks are those that have been converted into parallel work. Dud sparks are those which have already been evaluated to weak-head normal-form (WHNF). Garbage collected (GC'd) sparks are those which have been discarded as they were unnecessary for the evaluation of the program. Overflowed sparks are those which have been discarded due to the spark-pool being full. Fizzled sparks are those which have been evaluated in between being sparked for parallel evaluation and being converted into

```
127,312 bytes allocated in the heap
26,664 bytes copied during GC
62,448 bytes maximum residency (1 sample(s))
23,568 bytes maximum slop
2 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0  0 colls,     0 par    0.00s    0.00s     0.0000s    0.0000s
Gen  1  1 colls,     0 par    0.00s    0.00s     0.0015s    0.0015s

Parallel GC work balance: 10% (serial 0%, perfect 100%)

TASKS: 6 (1 bound, 5 peak workers (5 total), using -N2)

SPARKS: 10 (3 converted, 0 overflowed, 0 dud, 2 GC'd, 5 fizzled)

INIT    time    0.00s  (  0.01s elapsed)
MUT     time    0.00s  (  0.00s elapsed)
GC      time    0.00s  (  0.00s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    0.00s  (  0.01s elapsed)

Alloc rate    0 bytes per MUT second

Productivity  67.2% of total user, 11.1% of total elapsed

gc_alloc_block_sync: 0
whitehole_spin: 0
gen[0].sync: 0
gen[1].sync: 0
```

Figure 7.5: Example of Haskell Runtime Statistics for Parallel Programs

parallel work [47, 52].

At the end of a successful benchmark of a parallel program, the statistics relating to that program's execution time and parallel spark information including the number of sparks created, converted, overflowed, dud, garbage collected and fizzled are recorded, along with the size of the input, number of cores used and the benchmark run identifier. These metrics are then stored in a database for evaluation and comparison with the performance of the sequential program. The additional statistics relating to sparks that are recorded will allow for an evaluation to be performed between the hand parallelised and automatically parallelised programs. For example, comparing the percentage of sparks converted by each parallel version will determine

whether a hand parallelised version makes better use of the sparks it creates compared to an automatically parallelised version. Comparing the execution times of both the hand parallelised and automatically parallelised versions of a given benchmark will determine whether or not the automatic parallelisation technique represents an improvement over hand parallelisation techniques.

7.2 Evaluation of Benchmark Results

The results of the benchmarking process are presented in this section along with an evaluation of speedups and parallel behaviour for each benchmark program. The execution times of both the automatically and hand parallelised versions of each benchmark program are presented in Appendix B. The individual results and evaluation for each benchmark program are presented, both for with and without thresholding, as follows:

- The average absolute speedup of the benchmark program is shown with respect to the *number of cores* used by the parallel versions, averaged across all inputs. The sequential version is used as a baseline, shown as a constant across all cores, against which the results of both the automatically parallelised and hand parallelised versions are plotted. A result above the baseline indicates an improvement in efficiency over the sequential version. Absolute speedup graphs are used to show whether or not the parallelised benchmark program benefits from the addition of more cores.

- The *scalability*, or average speedup of the benchmark program, is shown with respect to the *input size* benchmarked, averaged across the numbers of cores. The sequential version provides a baseline against which the speedups of the automatically parallelised and hand parallelised versions are plotted. A result above the baseline indicates an improvement in efficiency over the sequential version. Scalability graphs are used to show whether or not the speedup increases for larger inputs.

- The parallel behaviour of the automatically parallelised version of the benchmark program is shown with respect to the *input size* benchmarked and presents the average numbers of sparks created, converted, dud, overflowed, gc'd and fizzled. A parallelised program exhibiting a

high number of converted sparks and low numbers of dud, overflowed, gc'd and fizzled sparks indicates good parallel behaviour. The percentage of each created spark which finishes in each of these states is also presented in order to better show its parallel behaviour as input size increases.

- A cost centre analysis of the automatically parallelised version each the benchmark program is presented in order to better understand its execution profile. This is used to show how much of the execution time is used by sequential and parallel operations.

The remainder of this section is structured as follows: Section 7.2.1 presents and discusses the speedups and scalability achieved by each of the benchmark programs. Section 7.2.2 presents and discusses the behaviour of the automatically parallelised programs with respect to their spark profiles. Section 7.2.3 presents a cost centre analysis of each of the automatically parallelised programs.

7.2.1 Absolute Speedups & Scalability

Benchmarking the sequential, automatically parallelised and hand parallelised versions of the benchmark programs results in absolute speedups as shown in Figure 7.6. This presents the average absolute speedup of the automatically parallelised version of each benchmark program plotted against its hand-parallelised counterpart, using the execution time of the sequential version as a baseline. As can be seen from these results, for each benchmark program the automatic parallelisation technique delivers a positive speedup when compared to both the sequential version and hand parallelised version, though in the case of $sumBig$, it is only a slight improvement when using at least 8 cores.

The results which were used to generate the absolute speedup graphs for each benchmark program are presented in more detail in Appendix A, which presents the absolute speedups obtained by both the automatically parallelised version and the hand parallelised version, based on the average execution times for each benchmark program, as shown in Appendix B. Examining the individual speedup results per core and per input size reveals a broader picture of the speedup behaviour of both parallelised versions.

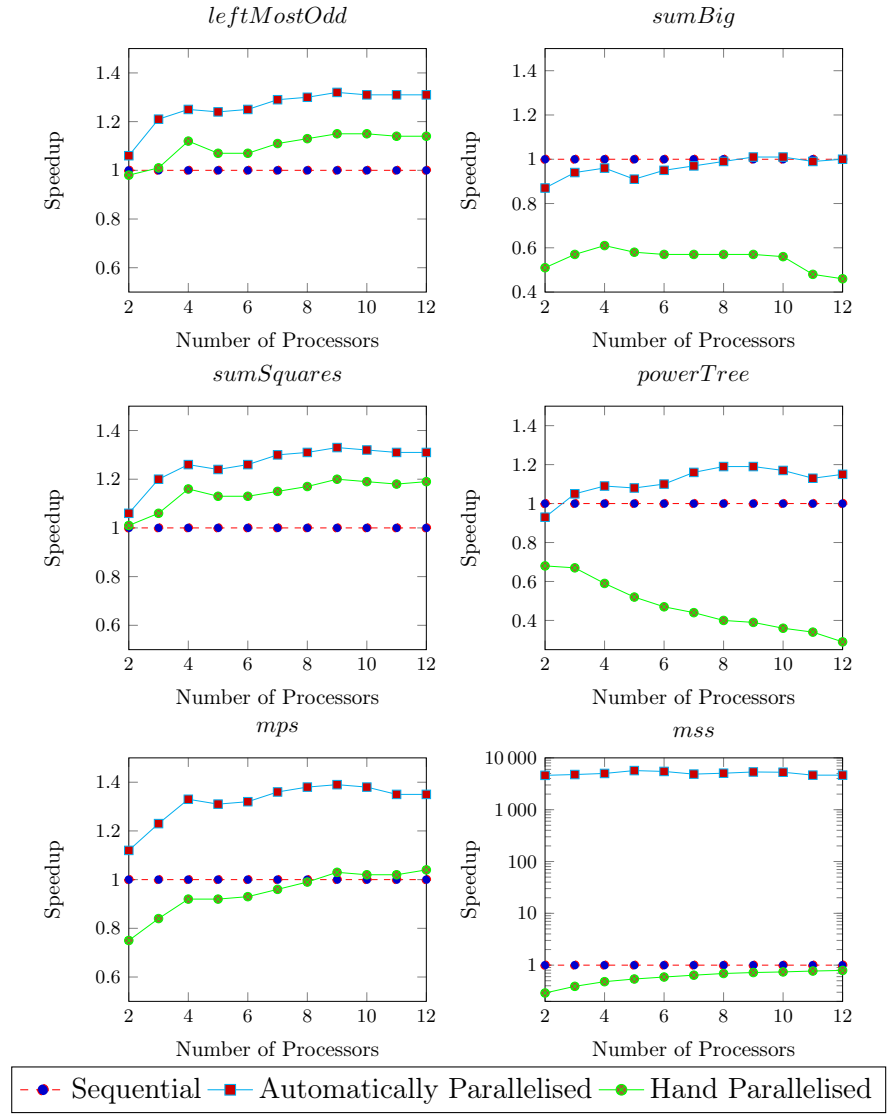


Figure 7.6: Average Absolute Speedups

The maximum segment sum benchmark presents the largest speedup over its sequential counterparts of all benchmark programs. Interestingly, the absolute speedup graph for *mss* appears to show an almost constant speedup across all cores. This near constant speedup is due to the optimisations that distillation performs due to its use as part of the partitioning process. Due to the impact that distillation has had on the average run-time of the automatically parallelised version, there isn't much of an opportunity for parallelisation to provide much positive effect on run-time performance.

Indeed, examination of the observed execution times for *mss*, presented in Table B.11, shows an average runtime of 0.01 seconds for the automatically parallelised version, compared to an average runtime of 102.84 seconds for the sequential version, with a maximum average speedup of 5685 on 5 cores.

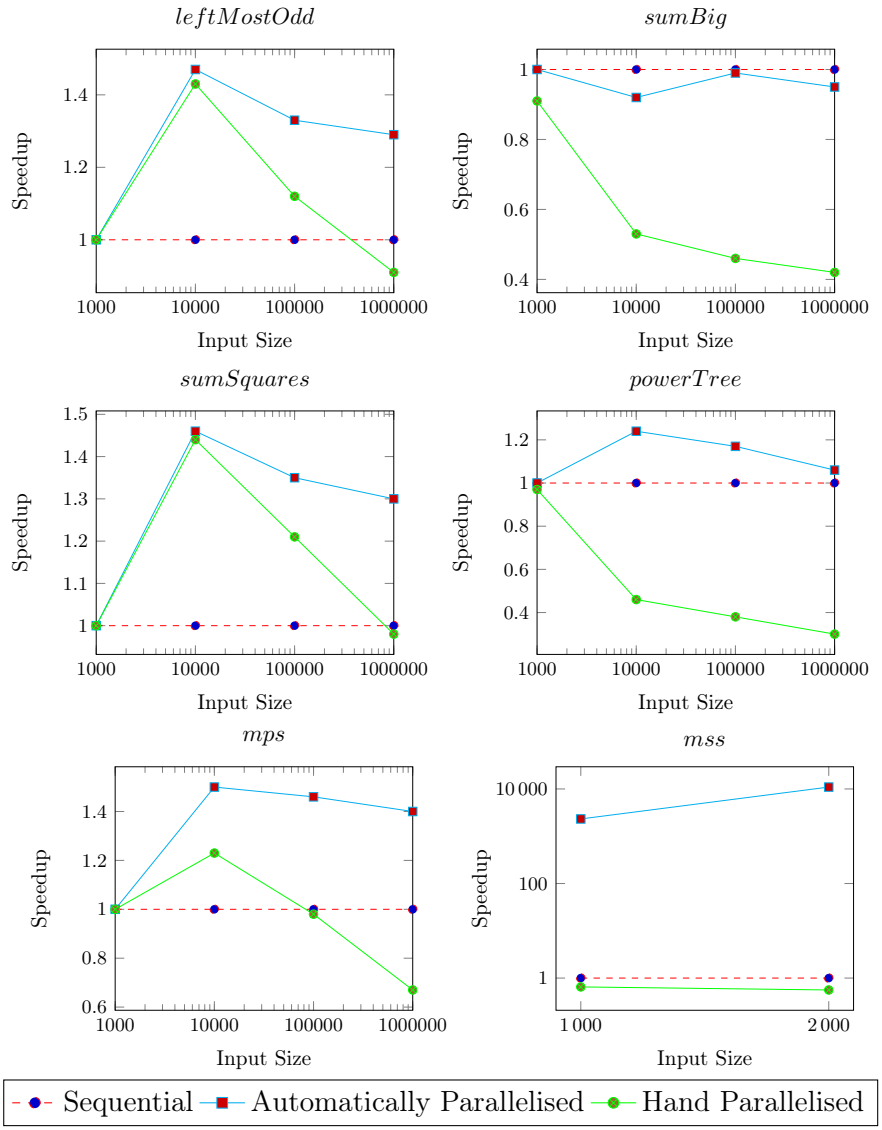


Figure 7.7: Scalability Graph for Benchmark Programs

In contrast to *mss*, the automatically parallelised version of the *sumBig* benchmark presents the worst speedup when compared to both its sequential counterparts with a minimum average speedup of 0.87, the lowest of all automatically parallelised benchmarks. Even with this, it still outper-

forms its corresponding hand parallelised benchmark, and it does offer a small improvement over the sequential version when executed on at least 8 cores. Like *mss*, this is one of the more interesting benchmark programs, as the remaining programs exhibit roughly the same behaviour across all cores. This is likely due to the fact that the automatically parallelised version of *sumBig* has the potential to create many more sparks than the others (excluding *mss*), as can be seen from its definition (Figure 6.4): unlike $leftMostOdd$, *sumSquares* and *mps* which all contain a single function which creates sparks while traversing a *join*-list, *sumBig* contains two functions: *sumBig* and *sum*, both of which create sparks while traversing their *join*-list argument. Each evaluation of *sumBig* creates a spark to evaluate the sum of its right child, as well as applying itself in parallel to that child. As a result of this, *sumBig* creates a much larger amount of sparks than the others for an input of the same size.

Further to the absolute speedup information presented above, Figure 7.7 presents scalability graphs for the benchmark programs. These show the parallel behaviour of both the automatically parallelised and hand parallelised versions of the benchmark programs with respect to input size, specifically whether or not their performance scales with input size, using the performance of the sequential programs as a baseline.

The information in the presented scalability graphs confirms that, with the exception of *sumBig*, the automatically parallelised versions of the benchmark programs outperform both their sequential and hand parallelised counterparts. In can be seen that, in general, the parallelised versions of the benchmark programs achieve their greatest speedup for an input size of 10000, after which the observed speedups decrease as input size increases, though this decrease tends to be much more rapid for the hand parallelised versions of the benchmark programs. In addition to this, it can be surmised that the hand parallelised programs do not scale well with input size, as can be seen from the scalability graphs: as the input size increases to 1000000, the average performance of the hand parallelised programs is always poorer than that of the sequential version, with the speedup obtained by *powerTree* closest to that of the sequential runtime of all benchmarked programs with an average speedup of 1.06 at this point.

The decline in scalability shown for automatically parallelised programs is likely caused, in part, by the optimistic parallelisation approach used in

the parallelisation technique: without thresholding, it simply attempts to parallelise any extracted expression which operates upon a *join*-list. As a result of this, the number of sparks created is a function of its input size and it stands to reason that for large input sizes, automatically parallelised programs are simply creating too much parallelism at too fine a level of granularity, and the performance of the resulting program suffers due to the overheads associated with this.

Examining the scalability graphs further, *mss* and *sumBig* again present the most interesting graphs, indeed the scalability of *mss* seems to avoid the trend of the other benchmark programs and appears to scale reasonably well with input size, though it is worth stating again that due to the non-performant natures of the sequential and hand parallelised versions of *mss*, it was only possible to benchmark *mss* for two reasonably sized inputs. As a result, though the scalability graph of *mss* shows promising results, these can not be taken as conclusive. As with the absolute speedups observed for *mss*, its apparent scalability is likely due, again, to the use of distillation as part of the parallelisation process: both the sequential and hand parallelised versions of the *mss* benchmark make heavy use of intermediate data in their evaluations, which has been removed from the automatically parallelised version. As a consequence of this, the automatically parallelised version creates far fewer sparks than the hand parallelised version which has an impact on its observed behaviour.

Again, the automatically parallelised version of *sumBig* shows performance less than that of the sequential version, showing poorer performance for inputs with a size larger than 1000, though it does come close to being as performant for an input size of 100000. This is also the worst scaling of the hand parallelised programs, showing poorer performance than even that of the hand parallelised version of *mss*. As shown previously, the automatically parallelised version of *sumBig* creates far more sparks than any other automatically parallelised program, the overheads of which bear a negative effect on its performance.

In general, from the speedup graphs shown in Figure 7.6 and the scalability graphs shown in Figure 7.7, it can be seen that for all benchmark programs the automatic parallelisation technique does result in parallel programs which have a reduced execution time when compared to the sequential versions, slight as this reduction may be in some cases. Though this does

validate the hypothesis that transformation techniques can be used to generate parallel programs that are at least as efficient as their hand parallelised counterparts, the speedups offered are quite small.

Given a sequential program and an equivalent parallel version running on n cores, an observed speedup of close to n would be desirable. However, in the case of the automatically parallelised programs, with the exception of mss, the maximum speedup across all numbers of available cores falls far short of this. For example, in the case of $leftMostOdd$, running the automatically parallelised version on 12 cores, a speedup of close to 12 would be desirable, however the actual observed speedup is a maximum of 1.5 at an input size of 10000.

As the number of sparks created by each of the automatically parallelised programs is a function of its input size, where the number of sparks created is at least the same as the total size of the *join*-list representing the input. It may be the case that each of these programs is simply creating too many sparks at too fine a granularity, and the obtained speedups are negatively affected by the overheads associated with this potential parallelism.

Thresholding Thresholding is a commonly used technique in the presence of excessive amounts of parallelism, and was added as an extension of the automatic parallelisation technique in order to limit such unwanted parallel behaviour by allowing the developer a measure of control over the number of parallel sparks which are created. Enabling thresholding for the automatically parallelised versions of the benchmark programs, using the thresholding strategy defined in Section 7.1.3, results in the speedup graphs shown in Figure 7.8 and the scalability graphs shown in Figure 7.9. The expectation when using thresholding is that, as the number of created sparks is reduced, the overheads associated with parallelism are likewise reduced. These graphs aim to show whether or not this is the case and compare the average speedups and scalability obtained by each thresholded program with that of its non-thresholded counterpart, again using the sequential program as a baseline.

With respect to the average absolute speedups obtained by the automatically parallelised benchmarks using thresholding, as shown in Figure 7.8, it can be seen that in each benchmark program the use of thresholding has resulted in an improvement in performance, though this is typically only seen

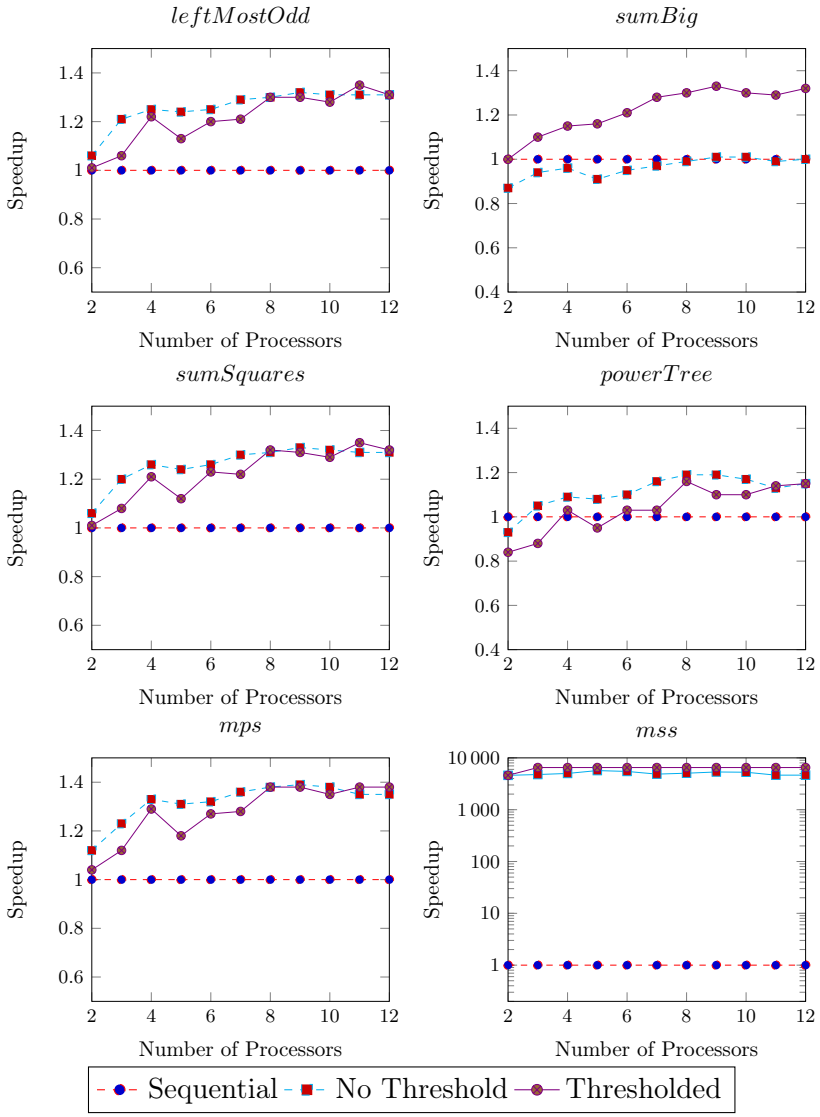


Figure 7.8: Average Absolute Speedups for Automatically Parallelised Benchmarks with Thresholding

when using more than 10 cores. Perhaps the most immediate difference when compared with the automatically parallelised benchmarks that do not use thresholding is that every benchmark program is always at least as efficient as the sequential version, with the exception of *powerTree*, which only becomes as efficient as its sequential counterpart when using 6 or more cores. It is worth noting at this point, that as the application of the automatic parallelisation technique to *powerTree* has garnered speedups over its se-

quential counterpart, though they are slightly lower, they are still largely in line with the *cons*-list based benchmarks. This shows that the application of the automatic parallelisation technique works for programs other than those solely defined on *cons*-lists.

The general parallel behaviour of the automatically parallelised programs validates, to some degree, the theory that without thresholding the automatic parallelisation technique may result in parallel programs that create too many sparks, and that excessive spark creation can have a detrimental effect on the observed speedups as a result.

As with the benchmarks evaluated without thresholding, mss yet again offers the most significant speedups over the sequential version. Again the obtained speedups are roughly constant, though as they have improved this would imply that the addition of optimistic parallelisation may decrease the efficiency of the version of mss resulting from distillation. The constant nature of the speedups obtained by mss with and without thresholding, supports this: the constant speedup implies that very little useful work is being performed in parallel: as the average execution times of the automatically parallelised versions are so low (0.01 seconds), there isn't much opportunity for parallelisation to offer an improvement. Following from this, the fact that introduction of thresholding and the reduction in parallelism creation it brings has improved its efficiency implies that the overheads associated with parallelisation do have a negative impact on its execution times. The use of thresholding has improved the speedups obtained by automatically parallelising mss which now offers an average speedup of 6508.17 when using 3 or more cores.

The most significant difference in parallel behaviour of the individual benchmarks is that of $sumBig$: without thresholding the automatically parallelised version is mostly less efficient than the sequential version, only becoming as efficient when evaluated using at least 8 cores. When thresholding is enabled, the automatically parallelised version is immediately as efficient as the sequential version, with its obtained speedups increasing as more cores are added, reaching a maximum average speedup of 1.33 on 9 cores compared to a maximum of 1.01 on 9 cores without thresholding. As stated previously, due to its definition, $sumBig$ creates many more sparks than $leftMostOdd$, $sumSquares$ and mps. This is still true, even with thresholding enabled: as the threshold function defined for this evaluation is based on the depth

of the *join*-list being evaluated, if the same *join*-list is supplied as an argument to both *sumBig* and *sum* and its depth is less than the threshold, sparks will be created in both of these functions. It is worth noting that where *sumBig* shows an improvement across all cores due to thresholding, $leftMostOdd$, *sumSquares* and *mps* do not. This indicates that perhaps the number of sparks created by these functions is too low for lower numbers of cores, but the extra sparks generated by *sumBig* is enough to result in a significant improvement. The speedups observed by *sumBig* validate the theory that thresholding can be used in combination with the automatic parallelisation technique to obtain improved behaviour in parallelised programs. Though this supports the use of thresholding, as mentioned, the enabling of thresholding does not always result in better parallel behaviour.

In the case of $leftMostOdd$, *sumSquares* and *mps*, up to 10 cores, the average speedups obtained are less than those obtained without thresholding, though they still represent an improvement in efficiency over their sequential counterparts. Beyond 10 cores, the use of thresholding in these benchmarks does offer moderate improvements when compared to those evaluated without thresholding: for example, in the case of $leftMostOdd$ a new maximum speedup of 1.35 is obtained on 11 cores compared to the previous maximum of 1.32 on 9 cores. The poorer speedups obtained for lower numbers of cores in these benchmarks is due to the number of sparks that each of these functions create: consider *sumBig*, which creates many more sparks than these functions and obtains a substantial improvement than its non-thresholded counterpart. Perhaps if the threshold used for $leftMostOdd$, *sumSquares* and *mps* were larger, similar improvements would be observed for these benchmarks.

Further to the absolute speedups examined above, Figure 7.9 presents the scalability graphs resulting from the use of thresholding in the benchmark programs. In general, with the exception of *mss* and *sumBig*, these show that with respect to input size, the use of thresholding has resulted in parallel programs which exhibit weaker scaling, though they do still outperform their sequential counterparts. As without thresholding, with the exception of *mss*, the benchmarks offer their best speedup for an input size of 10000, after which the observed speedup declines, with the speedup gained by *powerTree* falling below that of the sequential for an input size of 1000000. However, without thresholding this decline is much slower. This

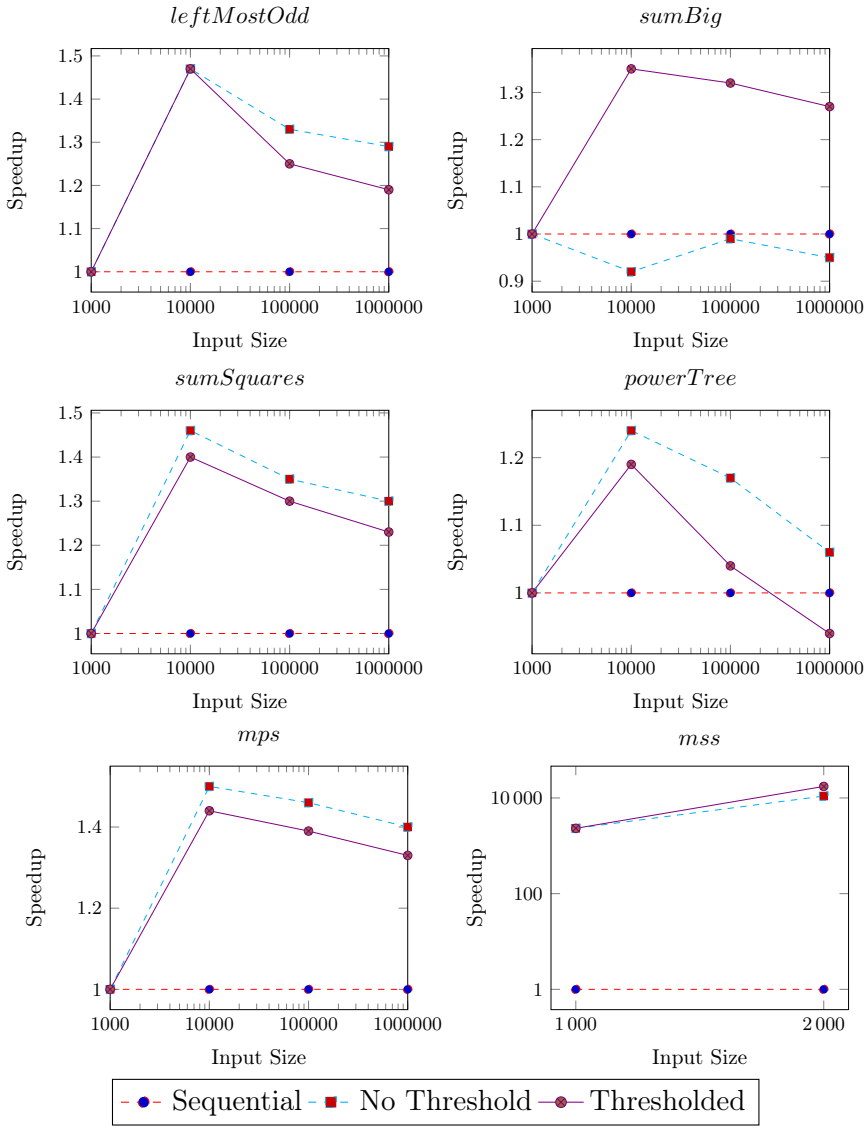


Figure 7.9: Scalability Graph for Automatically Parallelised Benchmarks with Thresholding

behaviour has a straightforward explanation: as the threshold function used in this evaluation is defined based on the depth of the *join*-list being evaluated, the number of sparks created is constant (with respect to the number of cores used), regardless of the overall size of the *join*-list. As a result of this, the automatically parallelised version with thresholding creates parallelism that is perhaps too coarse grained, and as a result the scalability of the resulting program is negatively impacted. This result furthers the notion

that, though the use of thresholding still results in parallel programs with an improved execution time over the sequential versions, there is no one ideal threshold value for every parallel program, as expected.

The differing efficiencies of the thresholded benchmark programs is not totally unexpected: each program is different, and it is unreasonable to expect that application of the same threshold will affect every program in the same way. Though small in some cases, the improvements in efficiency gained through the use of thresholding shows that the addition of thresholding to the automatic parallelisation technique can result in improved parallel behaviour. Perhaps the thresholding technique could be used as part of a heuristics based search for an ideal, or close to ideal, threshold for each individual program, however such work is beyond the scope of this thesis.

Examining the presented scalability information further, the *mss* and *sumBig* benchmarks exhibit the best scaling profiles, particularly when compared to their non-thresholded counterparts. As with the non-thresholded version, *mss* presents the best observed speedup, scaling reasonably well with input size. Again, it is worth acknowledging that it was only possible to benchmark *mss* using two reasonably sized inputs, due to the poor performance of the sequential and hand-parallelised versions, and as a result its scalability graph cannot be taken as conclusive. The use of thresholding in this benchmark presents an improvement in scalability, with a maximum average speedup of 17416.77 for an input size of 2000, compared to the 10968.02 obtained without the use of thresholding. This further confirms that the addition of parallelism to the distilled version of *mss* has a negative impact on its efficiency. As stated previously, the removal of the use of intermediate data in the distilled version of *mss* results in far fewer sparks being created when compared to the sequential version, resulting in a decrease in associated overhead costs. Through the use of thresholding, the amount of sparks created has been further reduced resulting in a 60% improvement in observed speedup.

The *sumBig* benchmark again presents the biggest difference in behaviour between its thresholded and non-thresholded version. As with the speedup information presented in Figure 7.8, without thresholding, this benchmark was mostly less efficient than its sequential counterpart. However, with the addition of thresholding it becomes immediately as efficient as the sequential version, with its obtained speedups peaking for an input

size of 10000 at 1.35 and declining slowly after that to 1.27 for an input size of 1000000. This behaviour observes the general trend of the automatically parallelised programs: a peak for an input size of 10000 followed by a steady decline. As with the other benchmarks, it would appear that as the threshold value is constant depending on the number of cores used, that as the input size grows the parallelism created is now of too coarse a granularity. In the case of *sumBig*, this decline is more gradual likely owing to the fact that it does still create more sparks than the other thresholded versions.

7.2.2 Parallel Behaviour

While the results presented in Section 7.2.1 show that, in general, with respect to both absolute speedup and scalability, the automatic parallelisation technique presents better parallel behaviour than the hand parallelised version, in addition to a run-time improvement over the sequential version, the speedups garnered by the automatically parallelised version do leave a lot to be desired, and further evaluation is required to determine the cause(s) of this poor offering. This behaviour can be expected to some degree, as the automatic parallelisation technique is not aware of the characteristics of the program being parallelised. However, since the run-time system is designed to deal with large numbers of sparks, this is partly delegated to the run-time system. The following discusses how to improve the automatic parallelisation to reduce the burden of spark management for the run-time system. The graphs presented in Figure 7.10 present the the parallel behaviours of each automatically parallelised program, using a log-log scale, with respect to the number of sparks created and at their status upon completion of the benchmark.

The amount of sparks *created* as part of the evaluation is important as this represents the amount of potential parallelism created by the program. The number of *converted* sparks represents the number of sparks that have *actually* been evaluated in parallel. The amount of *overflowed* sparks represents the number of sparks that were added to the spark pool when it was already full. While an overflowed spark will not be evaluated in parallel, this is not necessarily a negative, as sparking is optional, and the result of that spark will be evaluated regardless, although it will be evaluated sequentially. While a high number of overflowed sparks runs the risk of losing potentially useful parallelism, this is less problematic as the user can tune the spark pool

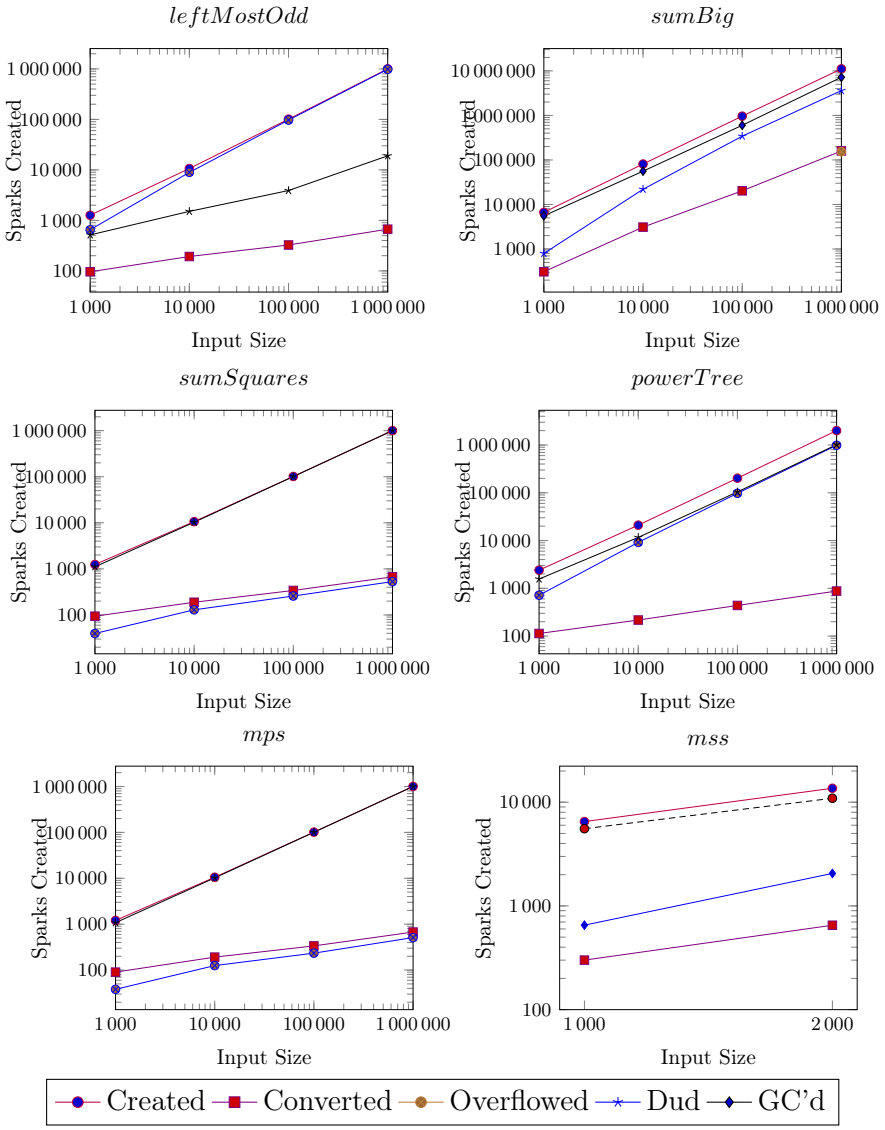


Figure 7.10: Spark Profiles for Automatically Parallelised Benchmarks

size. The amount of *dud* sparks represents the number of sparks that have already been evaluated to *weak-head normal form (WHNF)*. The number of *fizzled* sparks represents the amount of sparks whose results have already been calculated by another parallel process. The number of *garbage collected (GC'd)* sparks represents the amount of sparks that were created, but not actually required. Figure 7.11 presents the average breakdown of sparks which finish in each of these states as a percentage of the total number of sparks created for each benchmark program.

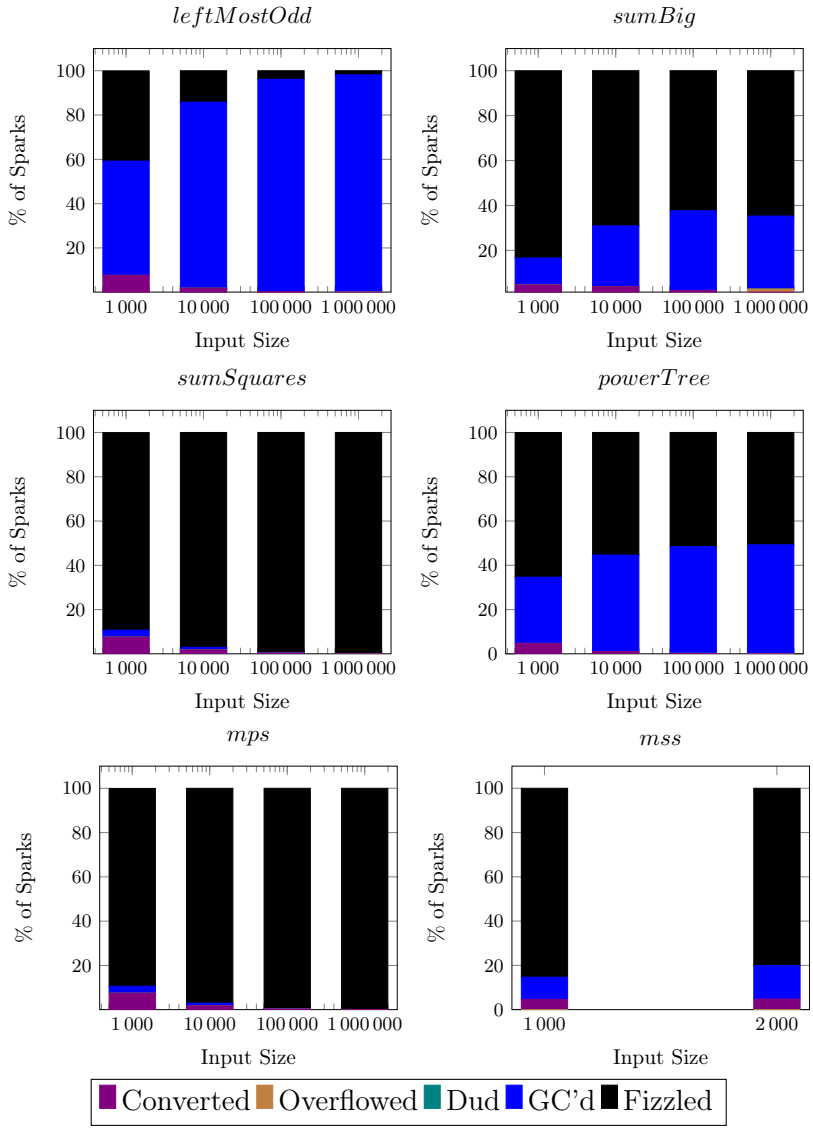


Figure 7.11: Automatically Parallelised Spark States

It can be seen from the spark profiles presented in Figure 7.10 and Figure 7.11, that the number of sparks created by the automatically parallelised programs shows at least linear growth with respect to input size, showing that the automatically parallelised programs do indeed create a massive amount of potential parallelism. While having a large amount of potential parallelism is good, an excessive amount can be detrimental, as the parallel program has to bear the costs of creating that parallelism as well as its elimination. The amount of sparks that are actually converted to parallel

work are a good indication of whether or not the potential parallelism is excessive. As can be seen from Figure 7.11, across all input sizes, the majority of sparks end up either garbage collected or fizzled, show that, though each benchmark does create a massive amount of potential parallelism, the majority of this is unnecessary. The result of this is that the automatically parallelised programs then have to bear the costs of managing this unused parallelism, with small gains from actual parallelism.

In addition to this, it can be seen from Figure 7.11 that as the input size increases, the relative percentage of converted sparks diminishes, though this can be expected to some degree as the number of sparks created is relative to the input size. This means that even though as input size increases there is more potential parallelism, a consequence of this is that there is also a decrease in the relative percentage of actual parallelism. This isn't necessarily a bad outcome, as the thread subsumption performed by GpH will result in fizzled sparks whose work is performed either sequentially or by other sparks that have been converted into useful parallelism. Indeed the generally high numbers of fizzled sparks further confirm that automatically parallelised programs do create far too many sparks at too fine a granularity. The amount of dependencies created by the automatically parallelised versions of the benchmark programs, as shown in Chapter 6 is likely a significant contributor to this issue, as the nature of these dependencies means that some sparks may be evaluated sequentially before they have been evaluated in

Of the presented spark profiles, perhaps $leftMostOdd$, $sumBig$ and mss present the most interesting profiles: $leftMostOdd$ presents a high number of garbage collected sparks, $sumBig$ presents a high number of both garbage collected and fizzled sparks and mss presents a very low percentage of sparks which are converted into useful work. In the case of the automatically parallelised version of $leftMostOdd$, though the amount of converted sparks increases with input size, it is far from keeping with the growth of the created sparks: from a conversion rate of 7.6% for an input size of 1000, it falls to 0.07% for an input size of 1000000. This low conversion rate, in combination with the high creation rate represents a poor showing from the automatic parallelisation technique as it means that the majority of created sparks perform no useful parallel work.

The spark profile presented in Figure 7.11 does offer some information as to what becomes of the remaining created sparks. As the input size increases,

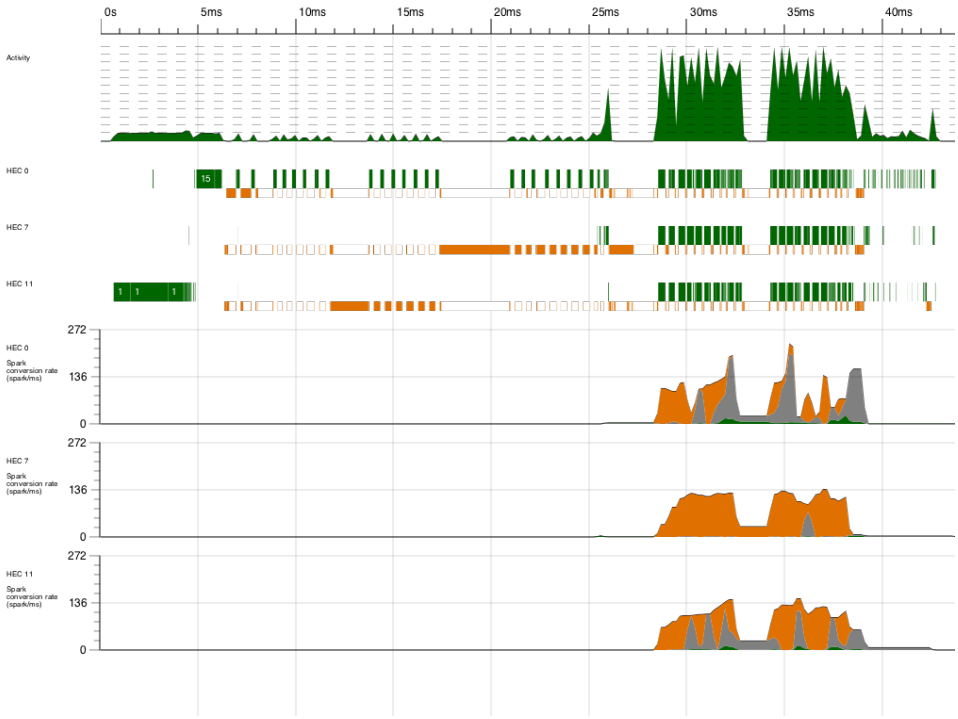


Figure 7.12: Automatically Parallelised $leftMostOdd$ Threadscope Profile

so does the number of sparks that are garbage collected: the amount of sparks garbage collected grows from 51.59% for an input size of 1000 to 98.04% for an input size of 1000000. This is an obviously undesirable behaviour, as garbage collection has a negative impact on the execution time of the parallel program, due to the overhead costs it bears. In addition to the high garbage collection rate, a significant number of the created sparks become fizzled, though this decreases as input size increases: from 40.75% at an input size of 1000 to 1.9% for an input size of 1000000. While, for a smaller input the percentage of fizzled sparks is quite high, this is not necessarily bad behaviour, as GpH's thread subsumption will naturally result in an amount of fizzled sparks.

Due to the high spark creation rate, the low conversion rate and the high garbage collection rate, it is obvious that for $leftMostOdd$, the automatic parallelisation technique has resulted in a program which simply creates an excessive amount of potential parallelism at too fine a granularity. Indeed, its spark profile supports this: for every input size, the majority of the created sparks end up being garbage collected, with very few being converted into

useful parallel work. This evaluation is supported by the *threadscope* [47] profile generated for the automatically parallelised version of $leftMostOdd$, shown in Figure 7.12, which was generated using an input size of 10000 and executed using 12 HECs. Due to space constraints, the presented profile does not include profiles for every HEC used, just the three most significant ones. The threadscope profile presents an activity profile for each of the cores, separated into active (green), blocked/garbage collected (orange) and fizzled (grey) with respect to time (x-axis) and cores (y-axis).

It can be seen from this threadscope profile, that while the sparks are being evaluated, the most significant behaviours are those of the sparks that are being garbage collected or fizzled. Very few sparks are actually converted into useful work, in fact HEC 7 converts virtually no sparks into useful work, it just spends its time either garbage collecting or fizzling sparks. The same can be said for HEC 11, though this does convert some sparks into useful work.

On examination of the threadscope profile as a whole, the most obvious point of interest is the amount of time that is spent performing garbage collection. This is highly undesirable as parallel evaluation is halted while the garbage collector is run, as can be seen from the troughs in the activity profile, though an increase in the total amount of garbage collection is to be expected due to the increase in heap size associated with increased parallelism. This increase will trigger more, potentially more expensive garbage collections as a consequence. Additionally, when considering the spark conversion rates, it is obvious that the majority of the parallel evaluation is spent not only performing garbage collection, but fizzling sparks. The amount of time spent either fizzling sparks or garbage collecting represents a significant overhead cost to the automatically parallelised program and the time spent satisfying this overhead could be better spent performing useful parallel evaluation. This further supports the conclusion that the automatic parallelisation technique, while offering a speedup over both sequential and hand parallelised version, generates a poorly performing parallel program for $leftMostOdd$. In addition to this conclusion, it is worth noting, that more than two thirds of the execution time of the program is spent in using only one HEC. This represents the sequential part of the automatically parallelised program, and includes the time spent partitioning the data the program operates on into a well-partitioned form, as well as the IO costs

associated with reading the inputs from file, as shown in Section 7.2.3.

As with $leftMostOdd$, the amount of sparks created by the automatically parallelised version of $sumBig$ grows linearly with input size though $sumBig$ creates far more sparks than the other benchmarks, with the exception of mss. As stated previously, the reason for this is that the definition of $sumBig$ relies upon more functions than the other benchmarks, and each of these functions creates sparks for the $join$-lists that they evaluate. As an example of this, where for an input size of 1000, $leftMostOdd$ creates an average of 1265 sparks, $sumBig$ creates an average of 6647. As would be expected, this large increase in created sparks does result in overflowed sparks for larger input sizes, with an average of 152966.91 overflowed sparks for an input size of 1000000, though this does not have an overly detrimental impact on the evaluation of the benchmark as the results of these overflowed sparks are still calculated as part of its evaluation.

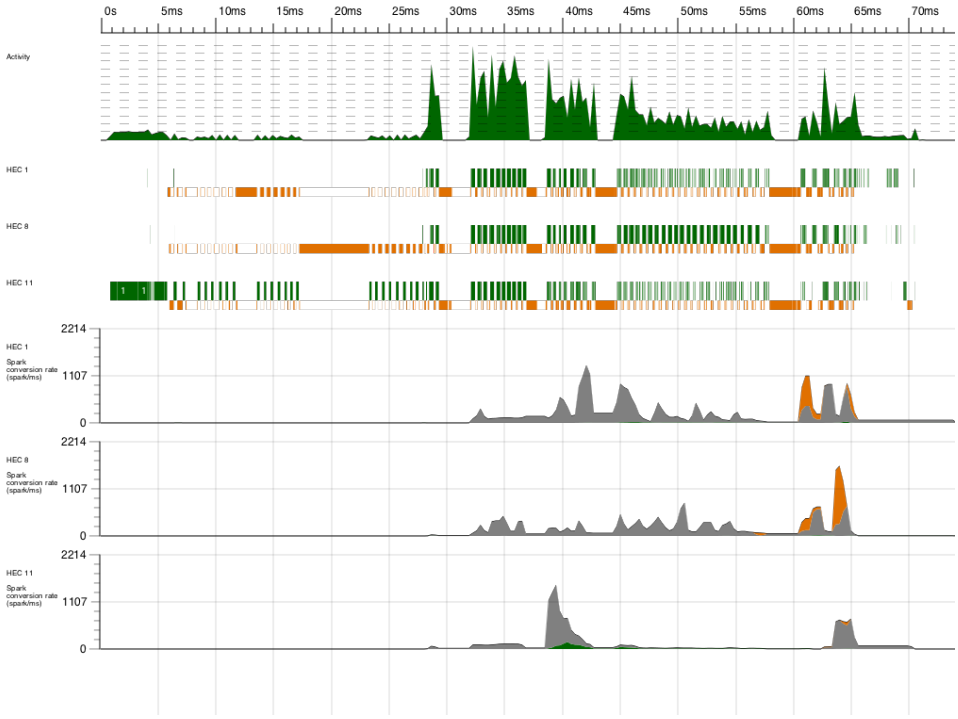


Figure 7.13: Automatically Parallelised $sumBig$ Threadscope Profile

Delving deeper into the spark profiles presented for $sumBig$ in Figure 7.10 and Figure 7.11, it can be seen that like $leftMostOdd$, as input size increases, the relative percentage of sparks which are converted into useful

parallel work diminishes, from 4.63% for an input size of 1000 to 1.43% for an input size of 1000000, offering further proof that even though a massive amount of parallelism is created by the automatically parallelised program, the majority of this does not prove useful. Unlike $leftMostOdd$, however, the majority of the remaining sparks become fizzled. As stated previously, this does not necessarily represent a negative impact on the parallel programs behaviour as GpH's thread subsumption lends itself to the presence of fizzled sparks.

Though the number of fizzled sparks in the automatically parallelised versions of both *sumBig* and $powerTree$ decreases with input size, the number of garbage collected sparks tends to increase. As mentioned previously, when sparks become garbage collected, this represents a negative impact on the runtime of the benchmark program, as it adds to the amount of time spent performing garbage collection across all cores. The combination of the high number of created sparks and the high proportion of both fizzled and garbage collected sparks, as well as the occurrence of overflowed sparks present in the spark profiles for the *sumBig* benchmark all further confirm that the automatic parallelisation technique results in parallel programs that contain a significant amount of potential parallelism but a low amount of actual, useful parallelism. This evaluation is supported by the threadscope profile generated for *sumBig* using an input size of 10000 and 12 HECs, as shown in Figure 7.13.

It can be seen from the presented threadscope profile that the most significant spark behaviour is that of the sparks which become fizzled. Following this, the next most significant behaviour is that of sparks which become garbage collected, which is highly undesirable as when garbage collection is being run all parallel evaluation is halted until garbage collection is completed. As with $leftMostOdd$, the amount of time that is spent performing either garbage collection or fizzling represents a significant overhead in the performance of the parallel program. In addition to this, it can be seen that very few sparks are actually evaluated in parallel and while these are being evaluated the majority of the cores have poor utilisation, with HEC 1 and HEC 8 converting very few sparks into actual work and spending most of their time fizzling sparks. This poor utilisation of the available cores is a significant contributor to the poor parallel performance of the *sumBig* benchmark program, particularly for lower numbers of cores.

In addition to this, like $leftMostOdd$, a significant portion of the program ($\sim 33\%$) is spent in sequential evaluation, prior to any parallel evaluation taking place. Again, this is the time spent reading the benchmark input from file and partitioning it into a well-partitioned form. Of this, the time spent reading from file is a cost that is also borne by the sequential version and the time spent partitioning is an unavoidable overhead associated with the automatic parallelisation technique.

Considering all of these factors together, this further lends itself to the conclusion that while the automatically parallelised program does result in a version of *sumBig* that does offer a slight speedup when compared to its sequential counterpart, it does so with a poorly performing parallel program. It would appear that perhaps one of the main contributors to this performance is the number of sparks created by the automatic parallelisation technique and the resulting overhead penalties that the parallel program suffers.

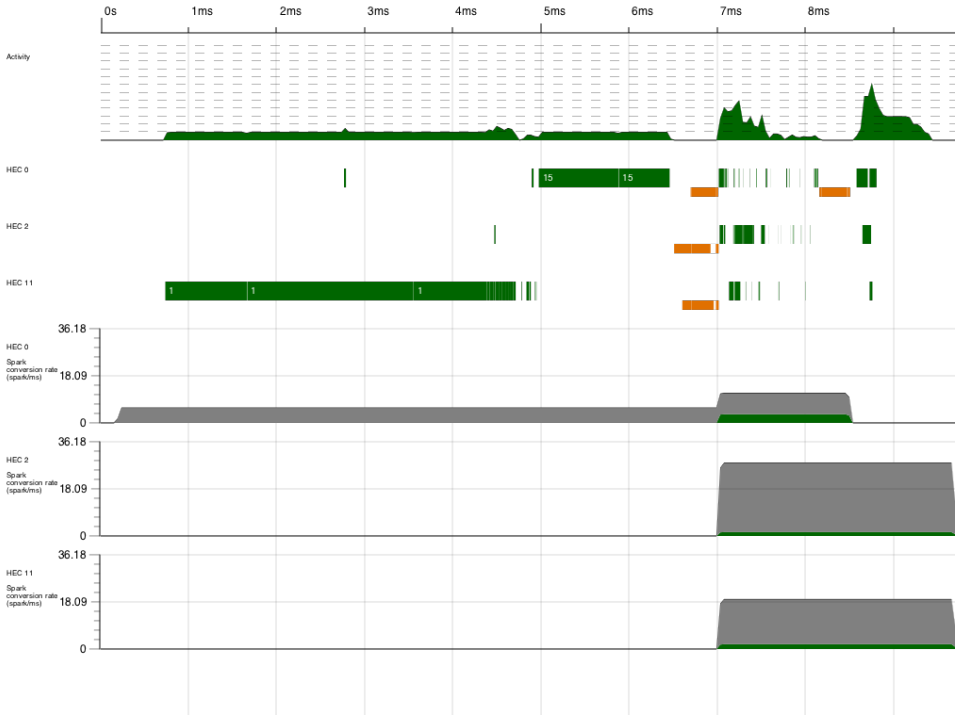


Figure 7.14: Automatically Parallelised *mss* Threadscope Profile

As with the *sumBig* benchmark, the evaluation of the *mss* benchmark results in a spark creation rate that is linear with respect to its input, though it does create far more sparks than the other benchmarks, excluding *sumBig*,

due to the number of functions it also uses in its evaluation, each of which creates sparks for the *join*-list that it evaluates, resulting in the creation of an average of 6505 sparks for an input size of 1000 and 13625 sparks for an input size of 2000.

Of the sparks that *mss* creates, relatively few become converted into useful work, with conversion rates of 4.67% for an input size of 1000 and 4.78% for an input size of 2000. Even with this poor conversion rate, *mss* still obtains the best speedup results, with a maximum obtained speedup of 6508.17 when evaluated using 3 or more cores. As stated previously, these speedups are due to the optimisations which distillation has performed on the resulting program. Indeed, the threadscope profile generated for *mss* using an input size of 100 using 12 cores, as shown in Figure 7.14 would appear to support this conclusion.

As can be seen from its threadscope profile, the majority (75%) of the execution time of *mss* is spent in sequential evaluation. Like the other benchmarks, this is the time associated with reading the benchmark input from file and converting into a well-partitioned form. Following this a period of parallel activity can be seen where, like *sumBig*, there is an obvious under-utilisation of the available HECs. Of the sparks that are converted into useful work the activity graph presented in the threadscope profile indicates that there is very little work for these sparks, resulting in this under-utilisation. Examining the behaviour of the sparks during this parallel period shows that the majority of the time is spent fizzling the created sparks across all HECs, with a very low rate of sparks being converted into useful parallel work. Again, it is worth mentioning that distillation has a significant impact on the complexity of the benchmark program and it would appear that the majority of its evaluation is performed sequentially with a very small portion being performed in parallel.

This shows that regardless of the large amount of potential parallelism created, very few of the created sparks prove worthwhile, and the overheads associated with the remaining sparks have a negative effect on the benchmarks execution time. Of the remaining sparks, the majority (85.35% for an input size of 1000 and 80.12% for an input size of 2000) become fizzled. As stated previously, a large number of fizzled sparks does not necessarily represent a decrease in speedup due to thread subsumption, however the remaining 15-20% of sparks which become garbage collected does. The over-

heads associated with the garbage collection present a problem, with respect to execution time, as any time spent performing garbage collection is time that would be better spent performing useful parallel evaluation.

The combination of these results appear to indicate that, like previous benchmarks, in the case of *mss* the automatic parallelisation algorithm creates a vast amount of potential parallelism, the majority of which does not prove useful. A significant percentage of the created sparks end up being garbage collected, the result of which is an increase in execution time, which is undesirable. However, the *mss* benchmark does present significant speedups with respect to its sequential counterpart, though this evaluation has furthered shown that the addition of parallelism to the distilled version offers little improvement. In fact, due to the garbage collection overheads associated with this parallelisation, it may have a negative impact on the efficiency of the distilled version.

Considering all of the benchmarks together, it is obvious that the automatic parallelisation technique creates an overly large number of sparks, the majority of which offer no significant benefit to the efficiency of the parallelised programs. In each benchmark, a significant number of the created sparks become garbage collected, the result of which is extra time spent in garbage collection. This is an undesirable result as, obviously, this time would be better spent performing useful evaluation. In addition to this, generally, the majority of sparks become fizzled, though this isn't always an undesirable behaviour due to GpH's thread subsumption. Never the less, there is an overhead associated with the creation and management of these sparks that it would be better to avoid. These observations are supported by the presented threadscope profiles which show that the majority of the cores spend their time either performing garbage collection or fizzling of sparks. Further to this, it is obvious in the presented threadscope profiles that significant portions of the execution times of the benchmark programs are spent in sequential evaluation due to the overheads associated with reading their inputs from file and converting these inputs into a well-partitioned form. However, these overheads are unavoidable and the sequential versions of the benchmarks must also the bear the costs of reading their inputs from file.

Thresholding In order to avoid the creation of massive amounts of potential parallelism at too fine a granularity, a thresholding extension was de-

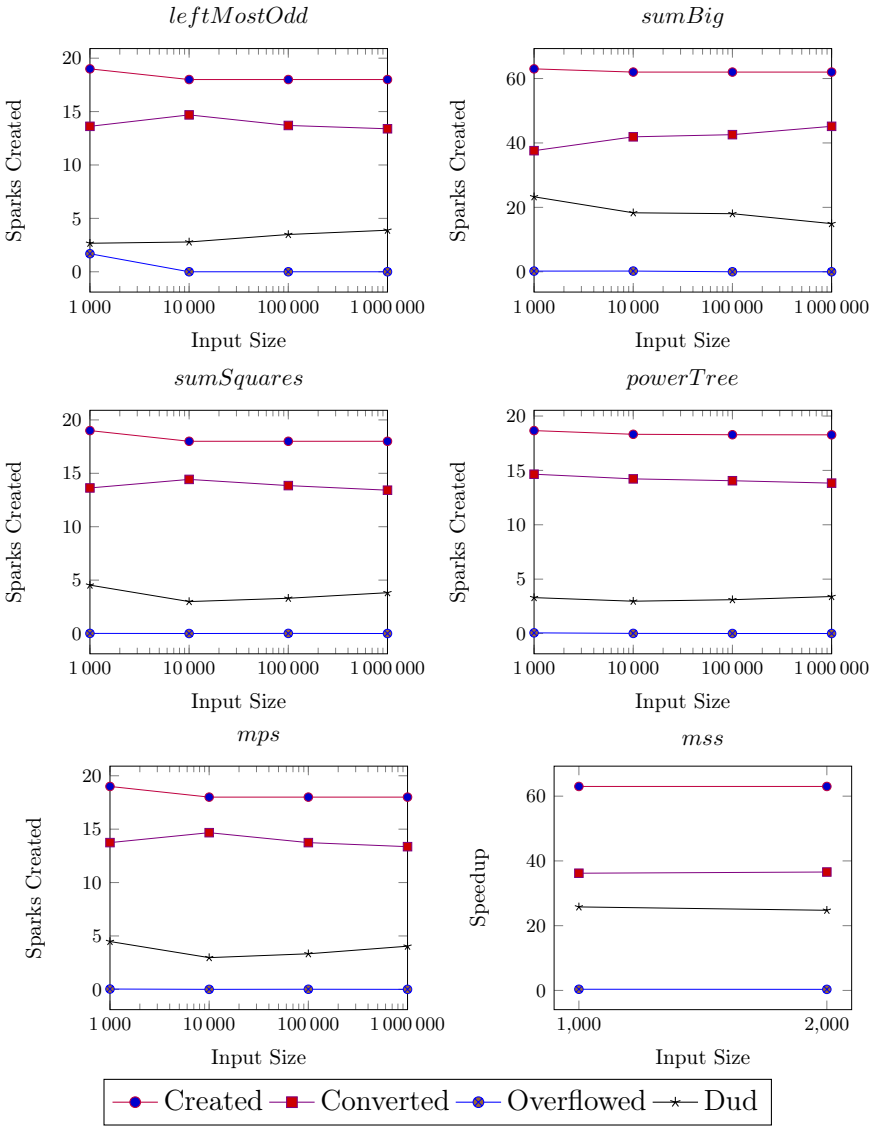


Figure 7.15: Spark Profiles for Automatically Parallelised Benchmarks with Thresholding

fined for the automatic parallelisation technique in Chapter 5. The graphs presented in Figure 7.15 present the the parallel behaviours of each automatically parallelised program using thresholding, with respect to the number of sparks created and at their status upon completion of the benchmark. Figure 7.16 presents the average breakdown of sparks which finish in each of these states as a percentage of the total number of sparks created for each thresholded benchmark program.

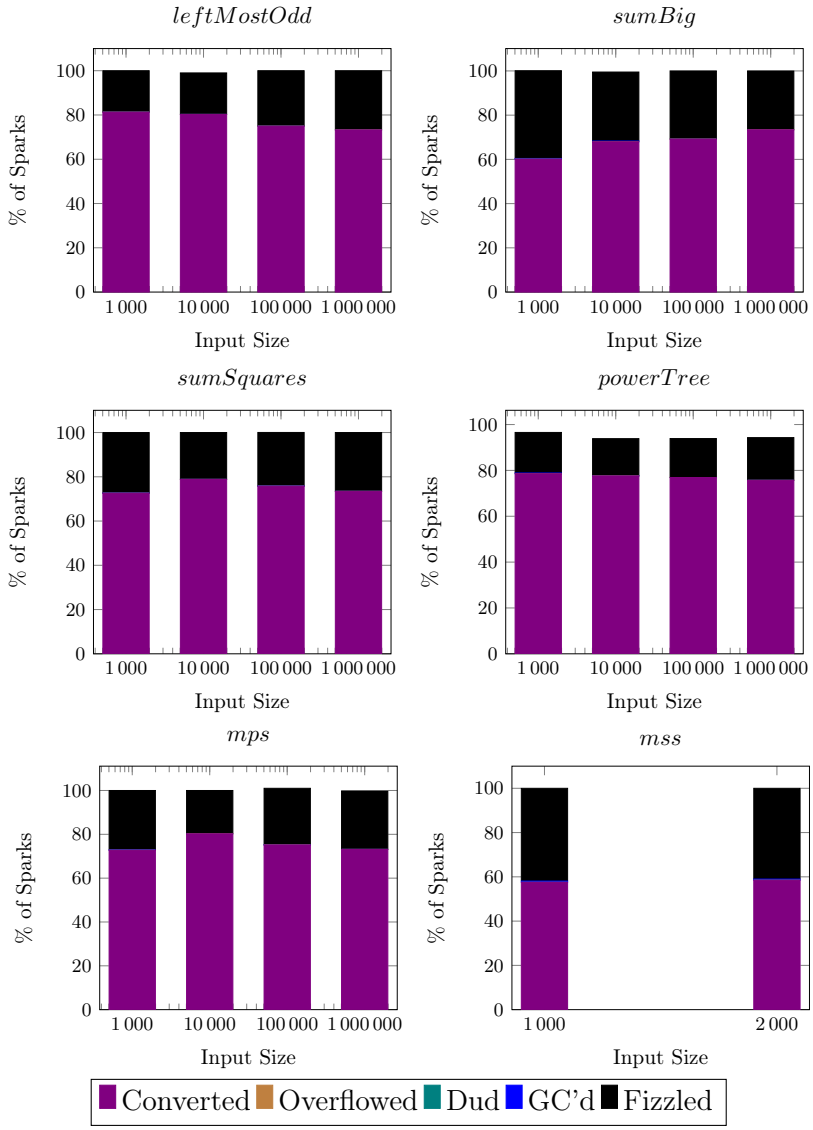


Figure 7.16: Automatically Parallelised Spark States with Thresholding

As can be seen from the graphs presented in Figure 7.15, the addition of thresholding, using the thresholding strategy defined in Section 7.1.3, to the automatically parallelised programs has resulted in a vast reduction in the amount of sparks created, as expected. Their parallel behaviours, with respect to the number of sparks which end up in each of the various states, are now much more consistent: for each benchmark program, across all input sizes, the majority of created sparks become converted into useful parallelism, with little to no sparks becoming garbage collected, a significant

improvement when compared to the behaviours of the benchmarks without thresholding. In addition to this, where previously the majority of the sparks in each of the benchmark programs were those that fizzled, the number of fizzled sparks resulting from thresholding is vastly reduced, as would be expected. The fact that there still exist some fizzled sparks should not be seen as a flaw in the parallelised program as, again, GpH's thread subsumption will naturally result in a number of fizzled sparks.

Further to this, with respect to the percentage of sparks that become converted, fizzled, overflowed, dud or garbage collected, the graphs presented in Figure 7.16 also show much better parallel behaviour resulting from the use of thresholding. It is immediately obvious from these graphs that the majority of the sparks created, when using thresholding, become converted into useful parallelism, with the remainder becoming fizzled. When compared with the corresponding profiles presented in Figure 7.11 resulting from parallel programs without thresholding, this shows a significant improvement in parallel behaviour.

Though the use of thresholding has resulted in better observed parallel behaviour, it is worth bearing in mind that, generally, the use of thresholding in the benchmarks only offers speedups over those without thresholding when using more than 8 cores, as presented in Figure 7.6. This is an interesting result, as it seems to suggest that where, previously, too many sparks were created at too fine a granularity, the use of this particular thresholding function may have resulted in the creation of too few sparks, with too coarse a granularity for lower numbers of cores.

The results of using thresholding in the automatically parallelised programs, as whole, show that it does offer improved speedups over those without, and it does appear to present an improvement in parallel behaviour. Indeed, the profiles presented in Figure 7.16 do present very desirable behaviour: high conversion rates with a lower rate of fizzling and no garbage collected, dud or overflowed sparks. Considering these results together, it seems obvious that while thresholding can offer an improvement in execution time and does offer improved behaviour, the selection of an appropriate thresholding function for each benchmark is an important process: what works well for one, like $sumBig$, may not offer the same gains to other automatically parallelised programs, as shown in Figure 7.8. As suggested previously, perhaps an extension to the thresholding technique to use a heuristic

search to determine a more optimal threshold function for each benchmark could be defined, however, such work is outside the remit of this thesis.

Considering the data presented in this section as a whole, there are several obvious conclusions which can be drawn: the automatic parallelisation technique, without thresholding, can result in parallel programs that create far too many sparks, the result of which can be high rates of thread fizzling and high levels of garbage collection, the overheads of which have a detrimental impact on the efficiency of the parallelised program. The use of thresholding can be used to limit the creation of too many sparks, though it is not without its own problems: mainly that the thresholding function which is used has a very significant impact on the efficiency of the resulting program, and as a result, what works for one may not work well for others.

7.2.3 Cost Centre Analysis

As described in Section 7.2.2, a significant portion of the execution time of each automatically parallelised program is spent performing sequential evaluation. This time is spent performing the tasks of reading the benchmark input from file and then converting that input into a well-partitioned form. The overheads associated with each of these functions are unavoidable, and the sequential version of each benchmark also bears the cost of reading its input from file. Examination of the costs of each of these functions may provide further insights into the behaviour of programs resulting from the automatic parallelisation technique.

A cost centre analysis of each of the automatically parallelised benchmark programs, which were generated from a sample run using an input size of 10000 (with the exception of mss, which used an input size of 1000), is presented in Figure 7.17. From this it can clearly be seen that the most significant cost in the evaluation of each benchmark is that of IO bound operations, mainly the reading of the inputs from the input file. In addition to this, for each benchmark, the partitioning function clearly adds a significant overhead to the evaluation though this tends to decrease as more cores are added: in the case of $leftMostOdd$, it decreases from 11.6% on 2 cores to 2.3% on 12 cores.

This is unfortunately an unavoidable overhead introduced by the parallelisation process, though perhaps a redefinition of the partitioning technique could reduce this cost and offer an increase in observed speedup to automat-

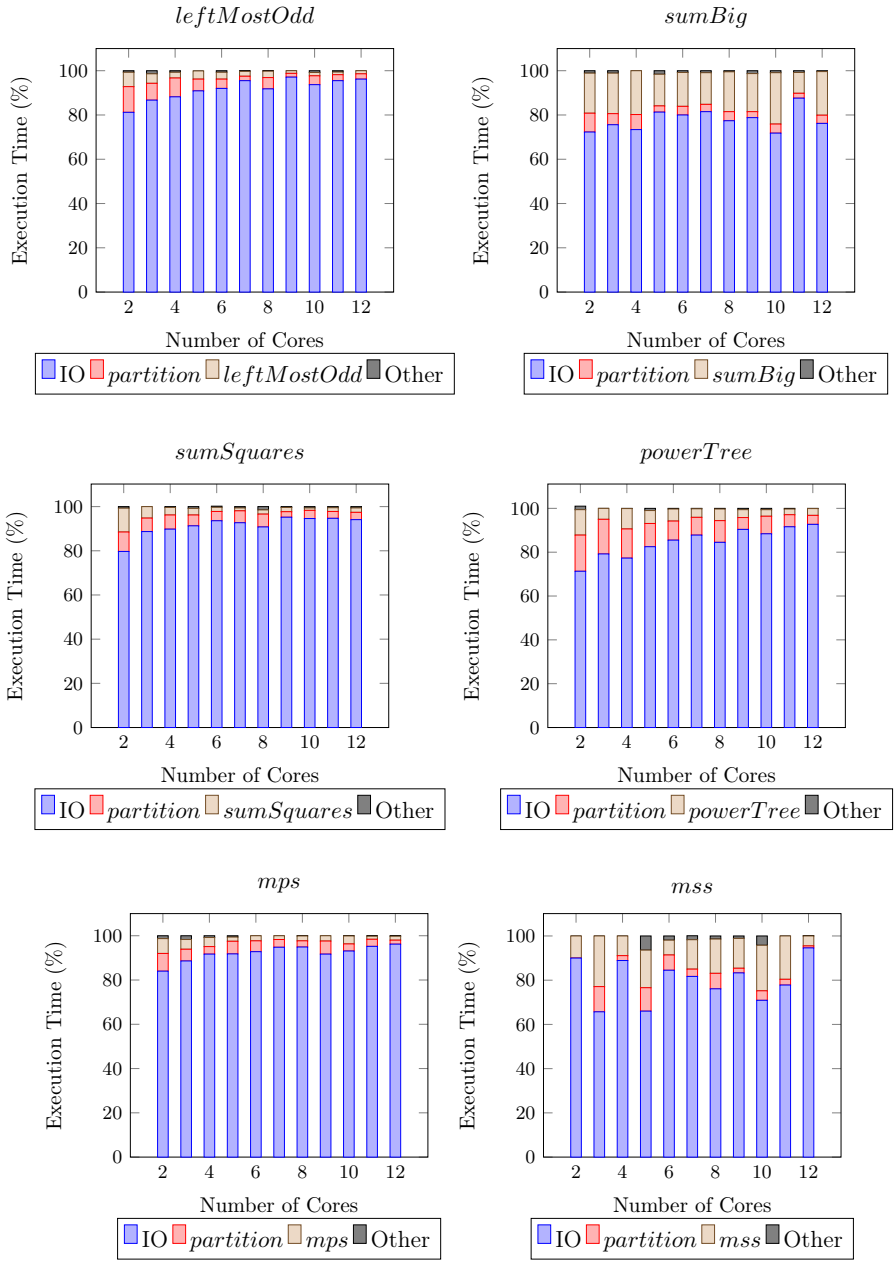


Figure 7.17: Cost-Centre Profile for Automatically Parallelised Benchmarks

ically parallelised programs. It is encouraging to see that, in general, the actual evaluation of the parallelised version of the problem does show a reduction in the cost of the actual benchmark as more cores are made available to it, though it does suffer from the problems described previously.

Of the presented cost centre graphs, *sumBig* and *mss* again present the

most interesting graphs. As stated previously, the definitions of each of these functions make use of further functions in their evaluation, increasing their complexity. For example, in the case of *sumBig*, at each point that a *Join* is encountered, the sum of the right child is calculated, in addition to the application of *sumBig* to both children. This increase in complexity is evident in their cost centre graphs, as their evaluations exhibit a significantly higher cost than the others. It can also be seen that while Figure 7.6 shows that the speedup garnered by *sumBig* increases slightly as more cores are added, this does not appear to have much impact on each of the cost centres. On the other hand, where Figure 7.6 shows that *mss* exhibits a roughly constant speedup over the sequential version, regardless of the number of cores used, the addition of extra cores does not seem to have a consistent impact on its cost centres. This is, again, likely due to the effect that distillation has had on the parallelised program: as it evaluates so quickly (less than 0.01s), its behaviour appears erratic.

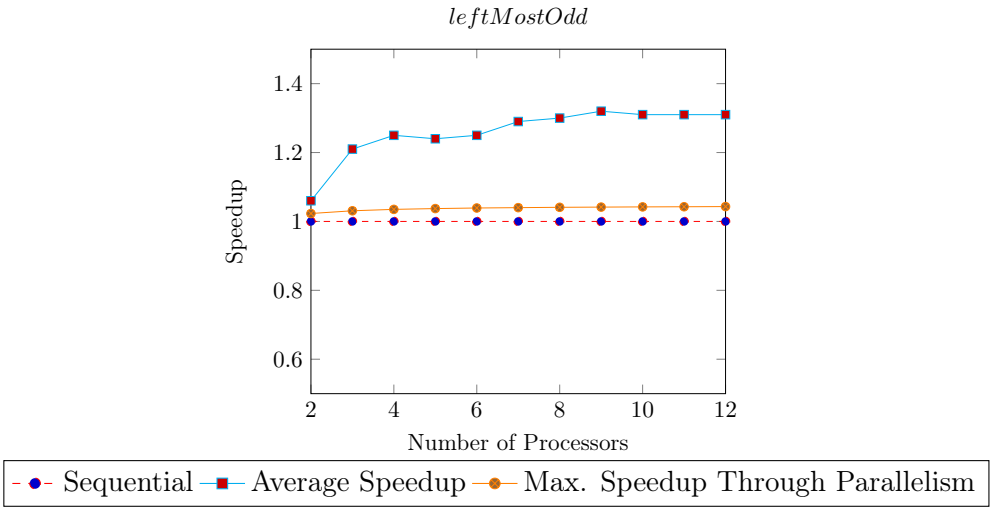


Figure 7.18: Average absolute speedups compared to expected parallel speedups using Amdahl's Law to show the expected speedup in *leftMostOdd* due to parallelism with respect to the amount of sequential work it requires.

It is worth considering Amdahl's Law when examining the presented cost centre analysis and the impact that the large amount of sequential work present in the benchmark programs has on the potential speedups obtainable by the automatically parallelised versions. For example, consider the *leftMostOdd* benchmark program: its theoretical speedup is as shown in

Figure 7.18, generated using a cost centre analysis for its sequential version for an input size of 10000. From this it can be seen that under Amdahl's law, due to the excessive amounts of sequential work, the theoretical speedup that can be obtained is quite low, with a maximum theoretical speedup of 1.043 on 12 cores. This is due to the fact that in the sequential version, the actual $leftMostOdd$ function takes only 4.5% of its 0.09 second runtime.

Though the theoretical speedup for $leftMostOdd$ is quite low, even with its excessive creation of potential parallelism, the automatic parallelisation technique has resulted in a parallel program which exceeds the expected theoretical speedup. This, as with the behaviour of mss, is likely mostly due to the optimisations that distillation performs while redefining the sequential program into one defined on well-partitioned data: its removal of intermediate data and redefinition of the given function into one that is tail-recursive.

The low potential speedup observed by $leftMostOdd$ is present in each of the benchmark programs, as they each have significant sequential costs. Obviously, a higher potential speedup would be desirable, but this requires further work to reduce the amount of sequential work performed by the automatically parallelised programs, such as that in the partitioning function as a start.

Considering the presented graphs as a whole, it is obvious that the most significant cost to the benchmark program is that of IO, though this cannot be helped. The next most significant cost across all benchmark programs is that of the partitioning function: perhaps a redefinition of the partitioning technique can result a reduction in its associated costs. Finally, the lowest cost is that of the actual benchmark itself: generally the cost of this tends to decrease as more cores are added, which would be expected. However, the costs observed by mss and $sumBig$ both buck this trend, an issue likely resulting from their use of auxiliary functions. Their use of auxiliary functions results from the distillation process and is likely unavoidable, though it does have a significant impact; perhaps distillation could be extended, or another transformation defined which limits the use of such functions.

7.3 Conclusion

In conclusion, this chapter has presented a thorough evaluation of the automatic parallelisation technique. It has described the methods by which

sequential, hand parallelised and automatically parallelised programs are compiled and executed, along with how appropriate inputs are generated and how metrics related to the benchmark programs are recorded. For each benchmark program defined in Chapter 6, sequential, hand parallelised and automatically parallelised versions were benchmarked. Following this a thorough analysis of each benchmark program was presented, initially without any thresholding of spark creation, following which the results of benchmarking the automatically parallelised programs using an example threshold function were presented, allowing for an in-depth evaluation of each program.

In the cases of programs without thresholding, the results are promising; in all benchmark programs, application of the automatic parallelisation technique has resulted in an improvement in efficiency with respect to both the numbers of cores made available to the program and to the input size used, though these efficiency improvements are limited by excessive parallelism. These increases in efficiency are obtained over both the sequential and hand-parallelised equivalents of the automatically parallelised program. In the case of the sum of the bigger numbers problem, this speedup is only obtained on average on 8 or more cores. Enabling thresholding to govern the number of sparks created by the automatically parallelised program results in a significant decrease in the numbers of sparks created as part of the parallel evaluation of all benchmark programs. As a consequence of enabling thresholding, each benchmark program also attained further speedups over their sequential counterparts, showing that thresholding the automatic parallelisation technique can result in a further improvement in efficiency.

Though there are improvements in efficiency gained from the automatic parallelisation technique, over both sequential and hand parallelised programs the speedups realised by the technique are not as optimal as they could be. For example, evaluating a parallel program on 12 cores (the maximum available in the benchmark environment), a speedup of close to 12 would be desirable. With the exception of the parallelised version of the maximum segment sum problem, which owes its drastic improvement in efficiency to the distillation technique, the automatically parallelised programs never come close of a speedup of 2, regardless of the number of cores that are used.

There are several reasons for this. As shown from the parallel behaviour of the automatically parallelised programs, if thresholding is not used, a high

percentage of the created sparks can end up garbage collected, which has a negative impact on the efficiency of the program due to the time it takes. Additionally, as the results of many of the sparks that are created are required by the sequential evaluation of the program before being evaluated in parallel, a significant percentage of the created sparks end up fizzled. Again this results in a loss of potential parallelism in the resulting program, though fizzled sparks can be expected to some degree due to GpH's thread subsumption. Additionally, as more sparks are created as input size increases, the likelihood that they become fizzled increases and the likelihood that they will be converted into parallel work decreases.

One of the most important parallelism tuning techniques is that of thresholding; enabling thresholding in the automatically parallelised programs appears to make their parallel behaviour more consistent and in some cases, like the sum of the bigger numbers problems, can result in an improvement in efficiency over the sequential and hand-parallelised versions, as well as the automatically parallelised version that makes no use of thresholding. However, this improvement is typically only seen above 10 cores. Whereas, without thresholding, the automatic parallelisation technique simply creates as much potential parallelism as possible, the use of the given thresholding function restricts the amount of potential parallelism, and this seems to be detrimental on lower numbers of cores. This prompts the obvious conclusion that there is no one threshold function that will result in optimal behaviour for all of the benchmark programs: each benchmark likely requires its own specific thresholding function to achieve this. Perhaps the thresholding extension of the parallelisation technique could be extended to generate an approximation of such a function using a heuristic search, however, such work is outside the scope of this thesis.

In general, the automatic parallelisation technique is capable of improving the efficiency of each benchmark program and results in average execution times less than those of both the sequential and hand parallelised versions. Unfortunately, there are some problems with the technique, and significant further work may be required in order to address these problems. One of the improvements that should be made, if possible, is to remove the dependencies between potentially parallelisable expressions, as this would allow more work to be sparked for parallel evaluation. As the resulting parallelised expressions would be independent this should result in a decrease in

the number of fizzled sparks and result in an improvement in parallel efficiency. Additionally, it may be the case that the *join*-list used as part of the parallelisation technique and the divide-and-conquer parallelism that results from its use may not present an optimal parallelisation approach and perhaps the use of a data-parallel approach for parallelisation may create better parallel performance.

Chapter 8

Conclusion

This chapter presents conclusions drawn from the material presented in this thesis, and describes its contributions to the field as well as discussing the results of the evaluation and suggesting improvements to the presented techniques. The work presented in this thesis has sought to prove the research hypothesis presented in Chapter 1 which states:

> **Research Hypothesis**
> *Program transformation techniques can be used to automatically transform a given sequential program into an equivalent efficient parallel program which makes use of well-partitioned data and has performance comparable to that of a hand-parallelised version of the given program.*

As a result of the research undertaken to prove this hypothesis, a *novel* fully automatic parallelisation technique has been presented which enables the parallelisation of functional programs defined on any data-type. Along with the definition of the parallelisation technique and its underlying theory, a thorough empirical evaluation was also presented, as was required to validate the thesis. At a high-level, this automatic parallelisation technique consists of four core components:

1. An automatic partitioning component which, given a program defined on arbitrary unpartitioned data, automatically generates functions which allow that data to be converted to and from a well-partitioned form, according to the *well-partitioned property* defined in Chapter 3.

2. The *distillation* program transformation system [30, 33] which is used in combination with the automatically generated partitioning functions to convert the given program into an equivalent one which is defined on well-partitioned data, in which expressions operating on well-partitioned data are typically extracted using **let** statements.

3. An automatic parallelisation transformation, simple by design, which takes the distilled program that is defined on well-partitioned data and converts it into an equivalent explicitly parallel program. Expressions in the resulting program which operate on well-partitioned data are evaluated in parallel, through the introduction of parallel **let** statements, albeit with very little constraints placed on evaluation order.

4. A means to allow a developer to define a thresholding function in order to govern the creation of parallel sparks by the parallelisation technique was also defined. Using this, the developer can use the size of the well-partitioned data, its depth and the number of available cores to determine whether or not a spark should be created at a certain point in the well-partitioned data. Through the use of the *size property*, defined in Chapter 5, the developer is able to relate the well-partitioned data to their original data when determining whether or not to create a parallel spark.

The remainder of this chapter is structured as follows: Section 8.1 discusses the research hypothesis and questions and how the work in this thesis has contributed to proving the hypothesis and answering the research questions. Section 8.2 discusses the contributions made to the field by the presented techniques. Finally, Section 8.3 presents a review of problems and difficulties facing the research, and suggestions for future work and improvements to the material presented in this thesis.

8.1 Research Hypothesis & Research Questions

The research presented in this thesis was undertaken in an attempt to prove the research hypothesis which was developed in Chapter 1 in order to solve the problems that a developer faces when developing parallel programs. By using program transformation techniques to automatically derive an equivalent, efficient parallel program from a sequential one, the difficulties as-

sociated with parallel development can be removed from the development process, as developers would then be able to continue to develop in the comfortable sequential paradigm and automatically generate parallel programs from their sequential ones at no additional cost. By proving the research hypothesis, the time and resources required to develop parallel programs can be greatly reduced as well as allowing for the issues that are currently placed upon the developers of such programs to be removed.

In order to guide research related to this hypothesis, several research questions were identified in Chapter 1. The research presented in Chapter 3 provided a positive answer to R.Q. 1 which asked: *given any data-type can a corresponding data-type be defined which will allow for efficient partitioning of the data?* By removing the recursive components from a given data-type, the data in an instance of that type can then be converted into a flattened list which can then be partitioned. Chapter 3 also provided a positive answer to R.Q. 2 which asked: *can program transformation be used to automatically redefine a program defined on any data-type into an equivalent program defined on well-partitioned data?* Using the derived data-type defined in answer to R.Q. 1, functions to convert any data into a well-partitioned form can be automatically defined. Distillation can then be used in combination with these functions in order to convert a given program into an equivalent one defined on well-partitioned data. The result of using distillation to handle this transformation is a program in which expressions operating on well-partitioned data are typically extracted using **let** statements.

The research presented in Chapter 4 provided a positive answer to R.Q. 3 which asked: *given a program defined over any data, can program transformation be used to automatically parallelise that program using well-partitioned data?* Using the output of the partitioning technique allowed for the definition of a set of automatic transformation rules to be developed which will convert a sequential program defined on well-partitioned data into an equivalent explicitly parallel one. The program resulting from distillation is automatically parallelised by targeting any extracted expressions it contains that operate on well-partitioned data, essentially converting them into parallel **let** statements, using Glasgow parallel Haskell. Glasgow parallel Haskell was selected for use in the explicit parallelisation pass due to its conceptual simplicity, its semantic transparency and its correctness preserving properties.

Further to these initial questions, in order to restrict the amount of sparks that are created as part of the automatic parallelisation technique, a refinement of the automatic parallelisation technique that allows for a user to define a thresholding function was presented in Chapter 5. This allows the user to define a function based on the size of the current *join*-list, its depth and the number of available processors, which is used by the parallelisation technique to determine whether or not to create parallel sparks at a certain point while traversing a *join*-list.

The evaluation of the automatic parallelisation technique in Chapter 7 provided an answer to R.Q. 4 which asked: *are the resulting automatically parallelised programs efficient with respect to the performance of a hand-parallelised version of the input program?* As the evaluation has shown, in most cases, the automatically parallelised programs are efficient with respect to their hand-parallelised counterparts.

Collectively, the answers to the research questions prove the hypothesis that program transformation techniques can be used to automatically transform a given sequential program into an equivalent efficient parallel program which makes use of well-partitioned data and has performance comparable to that of a hand-parallelised version of the given program.

8.2 Research Contributions

This thesis has presented an end-to-end implementation and evaluation of a fully automatic parallelisation technique. The research presented in this thesis has made contributions mainly to the fields of program parallelisation and program transformation. A *novel* fully automatic parallelisation technique has been presented which is capable of taking a sequential functional program and transforming it into an equivalent parallel program defined on well-partitioned data without the need for additional information from the user. Though the parallelisation technique is defined using Glasgow parallel Haskell, it could be redefined to use other explicit parallelisation approaches and this should not present too difficult a task.

The automatic parallelisation technique is significant as no other work, of which the author is aware, is capable of performing such a task without requiring additional information from the user or restricting the form of the input program. While there are many existing parallelisation techniques, as

described in Chapter 2, that are automated to varying degrees, most require that the user assert various laws about the operators in their programs, such as the associativity and distributivity of operators. The presented parallelisation technique does not require any such information from the user. In addition to this, where existing techniques are only defined for programs that exist in some specific form defined using specific data-types that are easily partitioned, the presented technique is applicable to programs defined on *any* data-type, which is a significant improvement over existing techniques.

As part of the presented parallelisation technique a *novel*, fully automatic partitioning technique was presented which allows data of any type to be converted into a well-partitioned form using *join*-lists. The automatic partitioning technique can be easily extended to support the use of other data-types that allow the data they contain to be well-partitioned. Such a change would require the introduction of a new data-type in place of the *join*-list that is used for partitioning. In addition to this, a modified partitioning technique would also require suitable *partition* and *unpartition* functions to replace those presented in Chapter 3 and modification of the parallelisation rule that is applied to the branches of case expressions in order to identify appropriate parallelisable variables. This modification is required, as though the partitioning technique is capable of *well*-partitioning any data-structure, it does so using a *join*-list as its output type. If its output type is modified, then the technique must be updated to reflect this new output type.

The partitioning technique is a significant contribution as it is due to this partitioning technique that the parallelisation technique places no restrictions on the data-types that are used by the input programs. In addition to this, no known work exists which will automatically redefine any data into a well-partitioned form. The automatic partitioning technique uses the distillation transformation system to redefine the original program into one defined on well-partitioned data. Other transformation techniques such as positive-supercompilation could be used to enable this part of the partitioning transformation, however distillation was selected as it is a more powerful technique. The results of the partitioning technique could potentially be used to extend existing parallelisation techniques based on *join*-lists to allow them to be applicable to programs defined on any data-type, however this would likely require a significant amount of work and would not remove any of the other issues inherent in such techniques.

In addition to the automatic partitioning technique, and parallelisation technique, a technique to allow for the governing of created parallelism, based on a user defined function was also presented in Chapter 5. The initial version of the parallelisation technique simply attempted to create as much parallelism as possible, however, this can be counter-productive, as shown in Chapter 7. Through the use of a threshold function, the user can constrain the creation of parallelism, with the goal of obtaining an automatically parallelised program with better parallel behaviour. Though the identification of an optimal thresholding value is outside the scope of this thesis, it is worth noting that this thresholding approach could be used as part of a heuristic based approach to determining such a value. For example, a simple approach would be to iteratively execute automatically parallelised programs using a range of threshold functions in a given environment, allowing for the identification of which of those threshold functions that provide the best behaviour for that environment. However, this would likely need to be performed on a per-environment basis. The definition of a means to automatically determine an optimal, or even close to optimal, thresholding function for any data-type would likely require significant further work.

8.3 Future Work

This section discusses the merits and downfalls of the presented research and presents potential solutions to the identified problems and improvements to the system as a whole. The parallelisation technique can convert a given sequential program into one defined on well-partitioned data and then parallelise expressions operating on well-partitioned data. The resulting parallel programs are generally efficient with respect to their hand-parallelised and sequential counterparts. However, the technique is unfortunately not without its issues and this section suggests two potential directions for future work: Section 8.3.1 presents a modification to the distillation technique which may remove some of the dependencies between parallelised expressions and Section 8.3.2 presents a modification to the partitioning and parallelisation techniques which would allow data-parallel programs to be generated.

8.3.1 Removing Dependencies Between Sparked Expressions

One of the main problems with the presented parallelisation technique is that in the distilled program defined on well-partitioned data there are often dependencies between expressions which operate on *join*-lists. These dependencies are introduced as part of the distillation generalisation step. As a result of this, many of the expressions that are sparked for parallel evaluation can end up being evaluated as part of the sequential evaluation of the program, as evidenced by the parallel behaviours of the benchmark programs that were evaluated and their high numbers of both garbage collected and fizzled sparks. This is problematic as, if the dependencies are removed from these expressions, more opportunities for independent parallel evaluation can be exposed to the parallelisation algorithm. Such an enhancement could have a positive effect on the execution times of automatically parallelised programs. In order to allow for the dependencies between expressions operating on *join*-lists to be reduced, the parallelisation algorithm will need to be modified.

The issue of dependencies between expressions operating on well-partitioned *join*-lists arises as a result of the generalisation process performed by distillation, which introduces extracted expressions using **let** statements after applying its generalisation rules [33]. The problem arises where the generalisation process extracts an expression which already contains a generalisation variable, as this creates a dependency between expressions which are potentially parallelisable and as a result this later limits the number of expressions which can be evaluated in parallel under the current parallelisation technique.

The introduction of these dependencies exposes a conflict between the goals of distillation and the definition of the presented parallelisation technique: the former aims to remove unnecessary computations, which can result in dependencies between expressions, and the latter aims to parallelise programs without dependencies. In order to solve this problem, the generalisation rules of the distillation system need to be modified in order to allow restructuring of the expressions being generalised. For example, consider the following expressions as they are encountered by distillation:

$$e_1 = sum\ l + sum\ r$$
$$e_2 = sum\ l' + (sum\ r' + sum\ r)$$
$$e_3 = sum\ l'' + (sum\ r'' + (sum\ r' + sum\ r))$$

Generalisation of e_2 with respect to e_1 results in the expression $e'_2 = sum\ l' + v$, where v is a generalisation variable representing the extracted expression $sum\ r' + sum\ r$. Generalisation of e_3 with respect to e_2 results in the expression $e'_3 = sum\ l'' + (sum\ r'' + v)$ and generalisation of e'_3 with respect to e'_2 results in the expression $e''_3 = sum\ l'' + v'$ where v' is a generalisation variable representing the extracted expression $sum\ r'' + v$. As v' refers to v, there exists a data-dependency between these generalised expressions and as a consequence of this, expressions which are sparked for parallel evaluation by the parallelisation technique may be evaluated sequentially instead of in parallel.

However, if the generalisation stage of distillation could be modified to exploit the associativity of the operators it has encountered, then this could be used to restructure the expressions which are to be generalised. It is worth noting that distillation is capable of proving the associativity of an operator itself: for any binary operator g, if the distillation of $g\ x\ (g\ y\ z) = g\ (g\ x\ y)\ z$ results in an expression which always evaluates to $True$, then the operator g is associative. Utilising this in the above example, distillation could prove the associativity of the + operator, and then restructure the expression e'_3 to be $e'_3 = (sum\ l'' + sum\ r'') + v$ prior to generalisation. Generalising the modified version of e'_3 with respect to e'_2 would result in the expression $e''_3 = v' + v$ where v' is a generalisation variable representing the expression $sum\ l'' + sum\ r''$ and contains no reference to any other generalisation variable. As a result, there will be no dependencies between parallel processes created by the parallelisation technique as a result.

As an example of how this modification can result in an improved parallel program, consider the definition of $sumSquares_{par}$ as shown in Figure 8.1, where the dependency between potentially parallelisable expressions is highlighted in red. The definition of $sumSquares_{par}$ has been parallelised according to the current parallelisation technique. The expression which calculates the sum of the right child of a $Join$ has been sparked for parallel evaluation and its result is required in the expression calculating the sum of the left child. However, if the suggested modification to the generalisation rules of distillation is implemented then it should produce an alternative

$$
\begin{aligned}
sumSquares_{par} &= f\ xs\ 0 \\
\\
f &= \lambda xs.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
&\qquad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
&\qquad\qquad Nil' \rightarrow 0 \\
&\qquad\qquad Cons'\ x \rightarrow x * x + s \\
&\qquad Join\ l\ r \rightarrow \textbf{let}\ r' = f\ r\ s \\
&\qquad\qquad \textbf{in}\ r'\ `par`\ f\ l\ r'
\end{aligned}
$$

Figure 8.1: Current Automatically Parallelised Version of *sumSquares*

to the current version of $sumSquares_{par}$ with no dependencies between the expressions calculating the sums of the children, potentially allowing more work to be completed in parallel.

$$
\begin{aligned}
sumSquares_{mod} = \lambda xs.\ &\textbf{case}\ xs\ \textbf{of} \\
&Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
&\qquad Nil' \rightarrow 0 \\
&\qquad Cons'\ x \rightarrow x * x \\
&Join\ l\ r \rightarrow \textbf{let}\ l' = sumSquares_{mod}\ l \\
&\qquad \textbf{in}\ l'\ `par`\ \textbf{let}\ r' = sumSquares_{mod}\ r \\
&\qquad\qquad \textbf{in}\ r'\ `pseq`\ l' + r'
\end{aligned}
$$

Figure 8.2: Automatically Parallelised Version of *sumSquares* with Modified Generalisation Rules

An alternative definition of $sumSquares_{par}$ is shown in Figure 8.2, $sumSquares_{mod}$, which is the result of enabling the suggested modification to the generalisation stage of distillation and then applying the parallelisation transformation to the program defined on well-partitioned data which is generated as a result. The changes to the program produced by the modifications to the distillation technique are highlighted in blue. The modified version now contains no dependencies between potentially parallelisable expressions and as a result, the sum of each child can be calculated independently where the evaluation of the left child is sparked in parallel with the strict evaluation of the right child, which should give the spark evaluating the left child a chance to complete before its result is needed. This could provide an improvement in parallel performance as a result.

8.3.2 Data Chunking Approach

Though the suggested modification to the generalisation rules of distillation should improve the performance of the derived parallel programs, it is entirely possible that too many sparks are created at too fine a granularity to provide much benefit. In addition to this, as more expressions are sparked for parallel evaluation it is also possible that the problem of overflowed sparks may be introduced, if the number of created sparks is high enough to exceed the size of the spark pool.

The thresholding technique presented as part of Chapter 5 could be used to prevent this occurring, however, like the unmodified parallelisation technique it is not without issue itself. As part of the thresholding technique, run-time checks to determine whether or not a threshold function has been breached are introduced at every recursive call to the *partition* function. As an example of how this can be improved, consider some input data with a size of n where under the current technique, a developer sets the threshold function to restrict the creation of sparks to expressions evaluating *join*-lists with a size greater than some value t. Rather than partitioning the data using a *join*-list and checking whether or not the threshold has been breached at each $Join$ that is encountered, a better approach may be to simply flatten the input data into a *cons*-list and then partition that *cons*-list into n/t chunks and parallelise the evaluation of expressions operating on each of these chunks. Such an approach goes above and beyond the simple divide-and-conquer thresholding presented in Chapter 5 and evaluated in Chapter 7, and makes the parallelisation garnered by such a technique potentially applicable not just to the multi-core machines currently targeted, but to other architectures too, like GPUs, though supporting these would require further work.

$$data\ ChunkList\ a ::= ChunkList\ (List\ a)_1\ \ldots\ (List\ a)_{n/t}$$

Figure 8.3: *Chunk*-List Data-Type Definition

Such an approach requires modification of the current partitioning technique as the *join*-list that is currently used does not allow for an implementation of this approach. A list-like structure containing the chunks that are to be evaluated in parallel is an intuitive approach. Such lists are referred to henceforth as *chunk*-lists and are defined as shown in Figure 8.3. Expressions

evaluating the chunks of a *chunk*-list should be evaluated in parallel using the *par* strategy, apart from those evaluating the last chunk which should be evaluated using the *pseq* strategy in order to provide an opportunity for any parallel sparks to finish their parallel evaluation before their results are needed.

$$
\begin{array}{l}
sumSquares_c = \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad ChunkList\ c_1\ \ldots\ c_n \rightarrow \textbf{let}\ c_1' = sumSquares_c'\ c_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ c_1'\ `par` \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ c_n' = sumSquares_c'\ c_n \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ c_n'\ `pseq`\ c_1' + \ldots + c_n' \\
sumSquares_c' = \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad Nil \qquad\qquad \rightarrow 0 \\
\qquad\qquad Cons'\ x\ xs \rightarrow \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad Nil' \quad \rightarrow 0 \\
\qquad\qquad\qquad\qquad\qquad\quad Con'\ x \rightarrow x * x + sumSquares_c\ xs
\end{array}
$$

Figure 8.4: Parallelised Version of *sumSquares* Operating on a *Chunk*-List

As an example of the effects of such a modification, consider the function $sumSquares_c$ defined in Figure 8.4 which is now defined for a *chunk*-list and has been parallelised according to a modified version of the automatic parallelisation algorithm which uses the modified version of generalisation and has been updated to suit the parallelisation of expressions which operate on the chunks of a *chunk*-list. Modification of the partitioning technique to allow for partitioning using a *chunk*-list has allowed the sum of each chunk to be calculated in parallel using the *par* strategy before the total sum of all chunks in the *chunk*-list is returned. As the number of chunks is controlled by the developer, the granularity of parallelism in this program can be controlled quite tightly and the chunking can be tailored to suit the underlying architecture.

By combining the suggested modification to the generalisation rules used by distillation with the use of a data-type that allows data to be partitioned into chunks, further improvements in the efficiency of the automatically parallelised program may be obtained. The resulting parallel programs should create far less sparks than previous versions, very few of which should become garbage collected or fizzled. However, if the chunk sizes are sufficiently small,

then the issue of sparks becoming garbage-collected and fizzled could present itself again. While the presented parallelisation technique represents a significant contribution to the field and is capable of generating efficient parallel programs, it is not without its faults. However, addition of the suggested improvements to the technique may provide a path to better performance automatically parallelised. The suggested improvements are complicated tasks however, and would require significant further research to determine their merit before being added to the parallelisation technique.

Bibliography

[1] K Abrahamson, N Dadoun, D.G Kirkpatrick, and T Przytycka. A Simple Parallel Tree Contraction Algorithm. *Journal of Algorithms*, 10(2):287 – 302, 1989.

[2] L. Augustsson. Compiling Pattern Matching. *Functional Programming Languages and Computer Architecture*, 1985.

[3] Roland Backhouse. An Exploration of the Bird-Meertens Formalism. Technical report, In STOP Summer School on Constructive Algorithmics, Abeland, 1989.

[4] R. Bird. Constructive Functional Programming. *STOP Summer School on Constructive Algorithmics*, 1989.

[5] 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.

[6] R. S. Bird. Algebraic Identities for Program Calculation. *The Computer Journal*, 32(2):122–126, 1989.

[7] G.E. Blelloch. Scans as primitive parallel operations. *Computers, IEEE Transactions on*, 38(11):1526–1538, Nov 1989.

[8] Guy Blelloch. NESL: A Nested Data-Parallel Language. Technical report, CARNEGIE MELLON UNIVERSITY, 1992.

[9] Christopher Brown, Hans-Wolfgang Loidl, and Kevin Hammond. Paraforming: Forming Parallel Haskell Programs using Novel Refactoring Techniques. *Twelfth Symposium on Trends in Functional Programming*, 2011.

[10] R. M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[11] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Gabriele Keller. Partial Vectorisation of Haskell Programs. In *In DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*. ACM Press, 2008.

[12] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A Status Report. In *In DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*. ACM Press, 2007.

[13] W N Chin, A Takano, Z Hu, Wei ngan Chin, Akihiko Takano, and Zhenjiang Hu. Parallelization via Context Preservation. In *In IEEE Intl Conference on Computer Languages*, pages 153–162. IEEE CS Press, 1998.

[14] Wei-Ngan Chin, Aik-Hui Goh, and Siau-Cheng Khoo. Effective Optimization of Multiple Traversals in Lazy Languages. In Olivier Danvy, editor, *PEPM*, pages 119–130. University of Aarhus, 1999.

[15] Wei-Ngan Chin, Siau-Cheng Khoo, Zhenjiang Hu, and Masato Takeichi. Deriving Parallel Codes via Invariants. In Jens Palsberg, editor, *Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 75–94. Springer Berlin Heidelberg, 2000.

[16] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. PhD thesis, Department of Computing Science, University of Glasgow, 1989.

[17] Murray Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. Technical report, Proceedings of Parco 93. Elsevier Series in Advances in Parallel Computing, 1993.

[18] John Darlington, A. J. Field, Peter G. Harrison, Paul Kelly, D W N Sharp, Qiang Wu, and R. Lyndon While. Parallel Programming using Skeleton Functions. In *PARLE'93, 5th International PARLE Conference on Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160, June 1993.

[19] M.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.

[20] Vincent W. Freeh and Vincent W. Freeh. A Comparison of Implicit and Explicit Parallel Programming. Technical report, University of Arizona, 1994.

[21] Jeremy 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.

[22] Andrew Gill, John Launchbury, and Simon Peyton Jones. A Shortcut to Deforestation. *FPCA: Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, 1993.

[23] Horacio González-Vélez and Mario Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[24] Sergei Gorlatch. Constructing List Homomorphisms for Parallelism. Technical report, Universitat Passau, 1995.

[25] Sergei 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.

[26] Sergei 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.

[27] Clemens Grelck. Single assignment c (sac) high productivity meets high performance. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School*, volume 7241 of *Lecture Notes in Computer Science*, pages 207–278. Springer Berlin Heidelberg, 2012.

[28] G. W. Hamilton. Higher Order Deforestation. *Proceedings of the Eight International Symposium on Programming, Logics, Implementation and Programs*, 1996.

[29] G. W. Hamilton. Higher Order Deforestation. *Fundamenta Informaticae*, 69(1-2):39–61, July 2005.

[30] G. W. Hamilton. Distillation: Extracting the Essence of Programs. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation*, 2007.

[31] G. W. Hamilton. Extracting the Essence of Distillation. *Proceedings of the Seventh International Andrei Ershov Memorial Conference: Perspectives of System Informatics*, 2009.

[32] G. W. Hamilton and G. Mendel-Gleason. A Graph-Based Definition of Distillation. *Proceedings of the Second International Workshop on Metacomputation in Russia*, 2010.

[33] Geoff Hamilton and N.D. Jones. Distillation and Labelled Transition Systems. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation*, pages 15–24, January 2012.

[34] Geoff Hamilton and N.D. Jones. Proving the Correctness of Unfold/Fold Program Transformations using Bisimulation. *Lecture Notes in Computer Science*, 7162:153–169, 2012.

[35] Z. Horváth, V. Zsók, P. Serrarens, and R. Plasmeijer. Parallel Elementwise Processable Functions in Concurrent Clean. *Mathematical and Computer Modelling*, 38(7–9):865 – 875, 2003. Hungarian Applied Mathematics.

[36] Zhenjiang Hu, Hideya Iwasaki, and Masato Takechi. Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms. *ACM Trans. Program. Lang. Syst.*, 19(3):444–461, May 1997.

[37] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *In ACM SIGPLAN International Conference on Functional Programming*, pages 73–82. ACM Press, 1996.

[38] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling Calculation Eliminates Multiple Data Traversals. In *In ACM SIGPLAN International Conference on Functional Programming*, pages 164–175. ACM Press, 1997.

[39] Zhenjiang Hu, Masato Takeichi, and Wei-Ngan 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.

[40] Zhenjiang Hu, Masato Takeichi, and Hideya Iwasaki. Diffusion: Calculating Efficient Parallel Programs. In *In 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–94, 1999.

[41] Zhenjiang Hu, Tetsuo Yokoyama, and Masato 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.

[42] Zhenjiang Hu, Tetsuo Yokoyama, and Masato Takeichi. Program Optimizations and Transformations in Calculation Form. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 144–168. Springer Berlin Heidelberg, 2006.

[43] Gérard Huet. The Zipper. *J. Funct. Program.*, 7(5):549–554, September 1997.

[44] Hideya Iwasaki and Zhenjiang Hu. A New Parallel Skeleton for General Accumulative Computations. *International Journal of Parallel Programming*, 32:389–414, 2004.

[45] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell, 2008.

[46] D. Knuth, J. Morris, Jr., and V. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[47] Eric Kow. Threadscope tour. `https://www.haskell.org/haskellwiki/File:Spark-lifecycle.png`, December 2014.

[48] Hans-Wolfgang Loidl, Philip W. Trinder, Kevin Hammond, Abdallah Al Zain, and Clement A. Baker-Finch. Semi-Explicit Parallel Programming in a Purely Functional Style: GpH. In Michael Alexander and Bill Gardner, editors, *Process Algebra for Parallel and Distributed Processing: Algebraic Languages in Specification-Based Software Development*, pages 47–76. Chapman and Hall, December 2008.

[49] Rita Loogen, Yolanda Ortega-mallén, and Ricardo Peña marí. Parallel functional programming in eden. *J. Funct. Program.*, 15(3):431–475, May 2005.

[50] Grant 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.

[51] Simon Marlow. *Deforestation for Higher-Order Functional Programs.* PhD thesis, University of Glasgow, 1996.

[52] Simon Marlow. *Parallel and Concurrent Programming in Haskell.* O'Reilly, 2013.

[53] Kiminori Matsuzaki, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. Systematic Derivation of Tree Contraction Algorithms. In *In Proceedings of INFOCOM '90*, pages 321–336, 2005.

[54] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *Proceedings of the 1st international conference on Scalable information systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[55] Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu, and Yoshiki Akashi. A Fusion-Embedded Skeleton Library. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference*, pages 644–653. Springer, 2004.

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

[57] Gary L. Miller and John H. Reif. Parallel Tree Contraction and its Application. In *26th Symposium on Foundations of Computer Science*, pages 478–489, Portland, Oregon, October 1985. IEEE.

[58] Gary L. Miller and John H. Reif. Parallel Tree Contraction Part 1: Fundamentals. In Silvio Micali, editor, *Randomness and Computation*, pages 47–72. JAI Press, Greenwich, Connecticut, 1989. Vol. 5.

[59] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 185–197, New York, NY, USA, 1990. ACM.

[60] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato 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.

[61] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic Inversion Generates Divide-and-Conquer Parallel Programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 146–155, New York, NY, USA, 2007. ACM.

[62] Wei ngan Chin, Siau cheng Khoo, and Tat wee Lee. Synchronisation Analysis to Stop Tupling. In *Lecture Notes in Computer Science*, pages 75–89. Springer LNCS, 1998.

[63] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, July 2013.

[64] Alberto Pettorossi and Maurizio Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Comput. Surv.*, 28(2):360–414, June 1996.

[65] Alberto Pettorossi and Maurizio Proietti. The List Introduction Strategy for the Derivation of Logic Programs. *Formal Aspects of Computing*, 13(3-5):233–251, 2002.

[66] Simon L. Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201, New York, NY, USA, 1989. ACM.

[67] Amr Sabry and Matthias Felleisen. Reasoning About Programs in Continuation-passing Style. *SIGPLAN Lisp Pointers*, V(1):288–298, January 1992.

[68] Tim Sheard and Leonidas Fegaras. A Fold for All Seasons. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 233–242, New York, NY, USA, 1993. ACM.

[69] 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.

[70] David B. Skillicorn. Architecture-Independent Parallel Computation. *Computer*, 23:38–50, December 1990.

[71] David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM COMPUTING SURVEYS*, 30, 1998.

[72] D.B. Skillicorn. *Foundations of Parallel Programming.* Cambridge International Series on Parallel Computation. Cambridge University Press, 2005.

[73] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. *International Logic Programming Symposium*, pages 465–479, 1995.

[74] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 1(1), January 1993.

[75] Jan Sparud. Fixing some space leaks without a garbage collector. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 117–122, New York, NY, USA, 1993. ACM.

[76] Morten Heine Sørensen. Turchin's Supercompiler Revisited - An Operational Theory of Positive Information Propagation, 1996.

[77] Akihiko Takano, Zhenjiang Hu, and Masato Takeichi. Program Transformation in Calculational Form. *ACM Comput. Surv.*, 30(3es), September 1998.

[78] Yong Meng Teo, Wei-Ngan Chin, and Soon Huat 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.

[79] Prabhat Totoo and Hans-Wolfgang Loidl. Lazy data-oriented evaluation strategies. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 63–74, New York, NY, USA, 2014. ACM.

[80] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[81] P.W. Trinder, H.-W. Loidl, and R.F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(5):469–510, 2002.

[82] Valentin F Turchin. *The Language Refal: The Theory of Compilation and Metasystem Analysis.* Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.

[83] Valentin F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, June 1986.

[84] Marco Vanneschi. The Programming Model of ASSIST, an environment for Parallel and Distributed Portable Applications. *Parallel Computing*, 28(12):1709 – 1732, 2002.

[85] Philip Wadler. Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile Time. *ACM Symposium on Lisp and Functional Programming*, 1984.

[86] Philip Wadler. Listlessness is Better than Laziness ii: Composing Listless Functions. *Proceedings of the Workshop on Programs as Data Objects*, 1985.

[87] Philip Wadler. Efficient Compilation of Pattern Matching. In Simon Peyton Jones, editor, *The Implementation of Functional Programming Languages.*, pages 78–103. Prentice-Hall, 1987.

[88] Philip Wadler. Deforestation: Transforming Programs to Eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

Appendices

Appendix A

Absolute Speedups for Benchmark Programs

A.1 Leftmost Odd Number

Table A.1: Automatically Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	1.2	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.49	1.49
100000	1.05	1.23	1.3	1.28	1.31	1.39	1.42	1.45	1.43	1.39	1.41
1000000	1.0	1.17	1.24	1.22	1.24	1.32	1.35	1.41	1.39	1.41	1.42
G. Mean	1.06	1.21	1.25	1.24	1.25	1.29	1.3	1.32	1.31	1.31	1.31
Min	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Max	1.2	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.49	1.49

Table A.2: Hand Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	1.2	1.2	1.5	1.43	1.43	1.5	1.5	1.5	1.5	1.5	1.5
100000	0.93	1.03	1.12	1.06	1.07	1.11	1.2	1.21	1.2	1.18	1.2
1000000	0.82	0.86	0.92	0.87	0.88	0.91	0.92	0.96	0.96	0.94	0.95
G. Mean	0.98	1.01	1.12	1.07	1.07	1.11	1.13	1.15	1.15	1.14	1.14
Min	0.82	0.86	0.92	0.87	0.88	0.91	0.92	0.96	0.96	0.94	0.95
Max	1.2	1.2	1.5	1.43	1.43	1.5	1.5	1.5	1.5	1.5	1.5

A.2 Sum of Bigger Numbers

Table A.3: Automatically Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	0.69	0.93	0.92	0.9	0.92	0.92	0.96	0.98	1.0	0.97	0.97
100000	0.95	0.9	0.96	0.92	0.97	1.0	1.02	1.06	1.04	1.02	1.04
1000000	0.87	0.94	0.97	0.84	0.92	0.98	0.99	0.99	0.98	0.99	0.98
G. Mean	0.87	0.94	0.96	0.91	0.95	0.97	0.99	1.01	1.01	0.99	1.0
Min	0.69	0.9	0.92	0.84	0.92	0.92	0.96	0.98	0.98	0.97	0.97
Max	1.0	1.0	1.0	1.0	1.0	1.0	1.02	1.06	1.04	1.02	1.04

Table A.4: Hand Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.55	0.5
10000	0.49	0.56	0.6	0.55	0.54	0.52	0.54	0.52	0.5	0.49	0.49
100000	0.39	0.46	0.5	0.46	0.45	0.46	0.47	0.47	0.46	0.45	0.45
1000000	0.36	0.42	0.45	0.44	0.43	0.44	0.43	0.44	0.43	0.42	0.41
G. Mean	0.51	0.57	0.61	0.58	0.57	0.57	0.57	0.57	0.56	0.48	0.46
Min	0.36	0.42	0.45	0.44	0.43	0.44	0.43	0.44	0.43	0.42	0.41
Max	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.55	0.5

A.3 Sum of The Squares

Table A.5: Automatically Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	1.2	1.4	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.49	1.47
100000	1.06	1.25	1.33	1.29	1.35	1.41	1.45	1.49	1.44	1.41	1.42
1000000	1.01	1.18	1.26	1.23	1.26	1.34	1.36	1.42	1.4	1.41	1.42
G. Mean	1.06	1.2	1.26	1.24	1.26	1.3	1.31	1.33	1.32	1.31	1.31
Min	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Max	1.2	1.4	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.49	1.47

Table A.6: Hand Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	1.2	1.23	1.5	1.49	1.47	1.5	1.5	1.5	1.5	1.5	1.5
100000	1.0	1.11	1.21	1.14	1.15	1.21	1.29	1.31	1.3	1.28	1.29
1000000	0.88	0.93	1.0	0.94	0.95	0.98	0.99	1.05	1.04	1.01	1.02
G. Mean	1.01	1.06	1.16	1.13	1.13	1.15	1.17	1.2	1.19	1.18	1.19
Min	0.88	0.93	1.0	0.94	0.95	0.98	0.99	1.0	1.0	1.0	1.0
Max	1.2	1.23	1.5	1.49	1.47	1.5	1.5	1.5	1.5	1.5	1.5

A.4 Power Tree

Table A.7: Automatically Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	1.0	1.2	1.2	1.2	1.22	1.32	1.41	1.41	1.29	1.23	1.2
100000	0.91	1.05	1.13	1.13	1.18	1.22	1.26	1.27	1.27	1.2	1.25
1000000	0.83	0.96	1.02	1.02	1.04	1.1	1.12	1.13	1.14	1.11	1.15
G. Mean	0.93	1.05	1.09	1.08	1.1	1.16	1.19	1.19	1.17	1.13	1.15
Min	0.83	0.96	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Max	1.0	1.2	1.2	1.2	1.22	1.32	1.41	1.41	1.29	1.23	1.25

Table A.8: Hand Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.63
10000	0.72	0.64	0.54	0.48	0.44	0.41	0.39	0.38	0.36	0.36	0.35
100000	0.55	0.58	0.5	0.4	0.36	0.34	0.32	0.31	0.28	0.27	0.26
1000000	0.52	0.56	0.43	0.37	0.3	0.27	0.21	0.2	0.16	0.14	0.13
G. Mean	0.68	0.67	0.59	0.52	0.47	0.44	0.4	0.39	0.36	0.34	0.29
Min	0.52	0.56	0.43	0.37	0.3	0.27	0.21	0.2	0.16	0.14	0.13
Max	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.63

A.5 Maximum Prefix Sum

Table A.9: Automatically Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	1.25	1.35	1.56	1.56	1.56	1.56	1.56	1.56	1.55	1.5	1.45
100000	1.14	1.33	1.44	1.41	1.45	1.54	1.58	1.59	1.56	1.48	1.5
1000000	1.1	1.26	1.38	1.35	1.36	1.44	1.47	1.51	1.49	1.49	1.51
G. Mean	1.12	1.23	1.33	1.31	1.32	1.36	1.38	1.39	1.38	1.35	1.35
Min	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Max	1.25	1.35	1.56	1.56	1.56	1.56	1.58	1.59	1.56	1.5	1.51

Table A.10: Hand Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	0.89	1.04	1.25	1.25	1.25	1.27	1.28	1.34	1.34	1.29	1.37
100000	0.7	0.83	0.92	0.92	0.94	1.0	1.07	1.1	1.09	1.1	1.12
1000000	0.5	0.57	0.62	0.62	0.63	0.68	0.69	0.75	0.75	0.75	0.77
G. Mean	0.75	0.84	0.92	0.92	0.93	0.96	0.99	1.03	1.02	1.02	1.04
Min	0.5	0.57	0.62	0.62	0.63	0.68	0.69	0.75	0.75	0.75	0.77
Max	1.0	1.04	1.25	1.25	1.25	1.27	1.28	1.34	1.34	1.29	1.37

A.6 Maximum Segment Sum

Table A.11: Automatically Parallelised Absolute Speedup

	Speedup per Number of Cores						
	2	3	4	5	6	7	8
1000	2321.38	2321.38	2321.38	2321.38	2321.38	2321.38	2321.38
2000	9123.07	9798.85	10798.73	13924.68	12905.8	10175.73	11023.71
G. Mean	4601.97	4769.37	5006.79	5685.46	5473.51	4860.22	5058.68
Min	2321.38	2321.38	2321.38	2321.38	2321.38	2321.38	2321.38
Max	9123.07	9798.85	10798.73	13924.68	12905.8	10175.73	11023.71

	Speedup per Number of Cores			
	9	10	11	12
1000	2321.38	2321.38	2321.38	2321.38
2000	12305.53	12025.86	9283.12	9283.12
G. Mean	5344.7	5283.62	4642.16	4642.16
Min	2321.38	2321.38	2321.38	2321.38
Max	12305.53	12025.86	9283.12	9283.12

Table A.12: Hand Parallelised Absolute Speedup

	Speedup per Number of Cores										
	2	3	4	5	6	7	8	9	10	11	12
1000	0.32	0.43	0.52	0.58	0.65	0.69	0.74	0.78	0.79	0.83	0.85
2000	0.27	0.36	0.44	0.5	0.54	0.59	0.63	0.67	0.69	0.72	0.73
G. Mean	0.29	0.39	0.48	0.54	0.59	0.64	0.69	0.72	0.74	0.77	0.79
Min	0.27	0.36	0.44	0.5	0.54	0.59	0.63	0.67	0.69	0.72	0.73
Max	0.32	0.43	0.52	0.58	0.65	0.69	0.74	0.78	0.79	0.83	0.85

Appendix B

Execution Times of Parallelised Programs

B.1 Leftmost Odd Number

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.05	0.04	0.04	0.04	0.04	0.04
100000	0.55	0.53	0.45	0.42	0.43	0.42	0.4
1000000	5.51	5.54	4.72	4.46	4.53	4.43	4.18
Average	1.53	1.53	1.3	1.23	1.25	1.23	1.16

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.04	0.04	0.04	0.04	0.04
100000	0.55	0.39	0.38	0.39	0.4	0.39
1000000	5.51	4.08	3.9	3.97	3.9	3.87
Average	1.53	1.13	1.08	1.1	1.09	1.08

Table B.1: Automatically Parallelised Execution Times

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.05	0.05	0.04	0.04	0.04	0.04
100000	0.55	0.59	0.54	0.49	0.52	0.52	0.5
1000000	5.51	6.7	6.41	5.97	6.33	6.29	6.08
Average	1.53	1.84	1.75	1.63	1.73	1.71	1.66

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.04	0.04	0.04	0.04	0.04
100000	0.55	0.46	0.46	0.46	0.47	0.46
1000000	5.51	6.01	5.71	5.75	5.85	5.82
Average	1.53	1.63	1.55	1.57	1.59	1.58

Table B.2: Hand Parallelised Execution Times

B.2 Sum of Bigger Numbers

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.09	0.06	0.07	0.07	0.07	0.07
100000	0.64	0.67	0.71	0.67	0.7	0.66	0.64
1000000	6.49	7.43	6.88	6.67	7.73	7.09	6.61
Average	1.8	2.05	1.92	1.85	2.13	1.96	1.83

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.06	0.06	0.06	0.06	0.06
100000	0.64	0.63	0.61	0.62	0.63	0.62
1000000	6.49	6.55	6.55	6.61	6.58	6.63
Average	1.8	1.81	1.81	1.82	1.82	1.83

Table B.3: Automatically Parallelised Execution Times

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.12	0.11	0.1	0.11	0.11	0.12
100000	0.64	1.65	1.39	1.29	1.39	1.42	1.41
1000000	6.49	18.06	15.5	14.36	14.75	15.03	14.7
Average	1.8	4.96	4.25	3.94	4.06	4.14	4.06

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	0.01	0.01	0.01	0.01	0.02	0.02
10000	0.06	0.11	0.12	0.12	0.12	0.12
100000	0.64	1.37	1.37	1.39	1.42	1.43
1000000	6.49	15.14	14.86	15.16	15.5	15.73
Average	1.8	4.16	4.09	4.17	4.26	4.32

Table B.4: Hand Parallelised Execution Times

B.3 Sum of The Squares

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.05	0.04	0.04	0.04	0.04	0.04
100000	0.59	0.56	0.47	0.44	0.45	0.44	0.42
1000000	5.9	5.83	5.01	4.69	4.8	4.68	4.41
Average	1.64	1.61	1.38	1.3	1.33	1.29	1.22

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.04	0.04	0.04	0.04	0.04
100000	0.59	0.41	0.4	0.41	0.42	0.41
1000000	5.9	4.33	4.15	4.23	4.18	4.14
Average	1.64	1.2	1.15	1.17	1.16	1.15

Table B.5: Automatically Parallelised Execution Times

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.05	0.05	0.04	0.04	0.04	0.04
100000	0.59	0.59	0.53	0.49	0.51	0.51	0.49
1000000	5.9	6.69	6.32	5.91	6.24	6.23	6.02
Average	1.64	1.83	1.73	1.61	1.7	1.7	1.64

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.04	0.04	0.04	0.04	0.04
100000	0.59	0.46	0.45	0.45	0.46	0.46
1000000	5.9	5.98	5.62	5.69	5.85	5.78
Average	1.64	1.62	1.53	1.55	1.59	1.57

Table B.6: Hand Parallelised Execution Times

B.4 Power Tree

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.06	0.05	0.05	0.05	0.05	0.05
100000	0.58	0.64	0.55	0.51	0.52	0.49	0.47
1000000	5.83	6.98	6.07	5.7	5.73	5.59	5.28
Average	1.62	1.92	1.67	1.57	1.58	1.54	1.45

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	0.01	0.01	0.01	0.01	0.01	
10000	0.04	0.04	0.05	0.05	0.05	
100000	0.46	0.46	0.46	0.48	0.46	
1000000	5.22	5.14	5.11	5.24	5.07	
Average	1.43	1.41	1.41	1.45	1.4	

Table B.7: Automatically Parallelised Execution Times

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.08	0.09	0.11	0.12	0.14	0.15
100000	0.58	1.05	1.01	1.16	1.44	1.6	1.72
1000000	5.83	11.14	10.35	13.45	15.79	19.57	21.97
Average	1.62	3.07	2.87	3.68	4.34	5.33	5.96

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	0.01	0.01	0.01	0.01	0.02	
10000	0.16	0.16	0.16	0.17	0.17	
100000	1.8	1.9	2.05	2.16	2.24	
1000000	27.36	29.54	35.93	40.43	45.06	
Average	7.33	7.9	9.54	10.69	11.87	

Table B.8: Hand Parallelised Execution Times

B.5 Maximum Prefix Sum

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.05	0.05	0.04	0.04	0.04	0.04
100000	0.64	0.56	0.48	0.45	0.46	0.44	0.42
1000000	6.53	5.93	5.18	4.74	4.84	4.81	4.54
Average	1.81	1.64	1.43	1.31	1.34	1.33	1.25

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.04	0.04	0.04	0.04	0.04
100000	0.64	0.41	0.4	0.41	0.44	0.43
1000000	6.53	4.44	4.33	4.37	4.38	4.33
Average	1.81	1.22	1.2	1.21	1.22	1.2

Table B.9: Automatically Parallelised Execution Times

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	0.01	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.07	0.06	0.05	0.05	0.05	0.05
100000	0.64	0.92	0.78	0.7	0.7	0.68	0.65
1000000	6.53	13.04	11.37	10.48	10.51	10.35	9.65
Average	1.81	3.51	3.05	2.81	2.82	2.77	2.59

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	0.01	0.01	0.01	0.01	0.01	0.01
10000	0.06	0.05	0.05	0.05	0.05	0.05
100000	0.64	0.6	0.59	0.59	0.59	0.58
1000000	6.53	9.53	8.69	8.74	8.66	8.5
Average	1.81	2.55	2.33	2.35	2.33	2.28

Table B.10: Hand Parallelised Execution Times

B.6 Maximum Segment Sum

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	23.21	0.01	0.01	0.01	0.01	0.01	0.01
2000	182.46	0.02	0.02	0.02	0.01	0.01	0.02
Average	102.84	0.02	0.02	0.02	0.01	0.01	0.02

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	23.21	0.01	0.01	0.01	0.01	0.01
2000	182.46	0.02	0.01	0.02	0.02	0.02
Average	102.84	0.02	0.01	0.02	0.02	0.02

Table B.11: Automatically Parallelised Execution Times

	Sequential	Runtime per Number of Cores					
		2	3	4	5	6	7
1000	23.21	73.56	54.45	44.33	39.98	35.89	33.72
2000	182.46	665.25	513.32	413.22	365.62	339.6	310.05
Average	102.84	369.4	283.89	228.78	202.8	187.75	171.88

	Sequential	Runtime per Number of Cores				
		8	9	10	11	12
1000	23.21	31.16	29.89	29.25	28.05	27.25
2000	182.46	288.87	272.26	263.05	254.47	250.79
Average	102.84	160.02	151.07	146.15	141.26	139.02

Table B.12: Hand Parallelised Execution Times

Appendix C

Hand Parallelised Benchmark Programs

C.1 Leftmost Odd Number

```
leftMostOdd xs
where
leftMostOdd =
λxs. case xs of
     Nil          → Nothing
     Cons x xs → case split xs of
                   Pair l r → let r'= leftMostOdd r
                              in r' `par` let l'= leftMostOdd l
                                          in l' `pseq` case odd x of
                                                         True  → Just x
                                                         False → case l' of
                                                                   Just x  → l'
                                                                   Nothing → r'
```

Figure C.1: Hand Parallelised Definition of *leftMostOdd* Operating on Unpartitioned Data

C.2 Sum of Bigger Numbers

```
sumBig xs
where
sumBig = λxs. biggers xs (sum xs)

biggers =
λxs. λs. case xs of
         Nil          → 0
         Cons x xs → case split xs of
                       Pair l r → let r'= biggers r (s - x - sum l)
                                  in r' `par` let l'= biggers l (s - x)
                                              in l' `pseq` case x > (s - x) of
                                                             True  → x + l' + r'
                                                             False → l' + r'
```

Figure C.2: Hand Parallelised Definition of *sumBig* Operating on Unpartitioned Data

C.3 Sum of The Squares

```
sumSquares xs
where
sumSquares =
λxs. case xs of
     Nil          → 0
     Cons x xs → case split xs of
                   Pair l r → let r'= sumSquares r
                              in r' `par` let l'= sumSquares l
                                          in l' `pseq` x * x + l' + r'
```

Figure C.3: Hand Parallelised Definition of *sumSquares* Operating on Unpartitioned Data

C.4 Power Tree

```
powerTree xs
where
powerTree = λxs. case xs of
                   Leaf x   → x^3
                   Node l r → let r'= powers r
                              in r' `par` let l'= powers l
                                          in l' `pseq` l' + r'
```

Figure C.4: Hand Parallelised Definition of *powerTree* Operating on Unpartitioned Data

C.5 Maximum Prefix Sum

```
mps xs
where
mps    =  λxs. case xs of
                 Nil          → 0
                 Cons x xs → mps′ xs x

mps′   =
λxs. λn. case xs of
       Nil          → n
       Cons x xs → case split xs of
                     Pair l r → let r′= mps r
                                in r′ `par` let l′= sum l
                                    in l′ `par` let l″= mps l
                                        in l″ `pseq`  max n (n + (max x
                                                      (x + max l″ (l′ + r′))))
```

Figure C.5: Hand Parallelised Definition of *mps* Operating on Unpartitioned Data

C.6 Maximum Segment Sum

```
mss xs
where
mss      = λxs. maxList (map sum (segments xs))

maxList  = λxs. case xs of
                  Nil        → 0
                  Cons x xs  → maxList' x xs

maxList' = λm. λxs. case xs of
                      Nil        → m
                      Cons x xs  → case split xs of
                                     Pair l r → let r'= maxList r
                                                in r' `par` let l'= maxList l
                                                            in l' `pseq` max m (max x (max l' r'))

map      = λf. λxs. case xs of
                      Nil        → Nil
                      Cons x xs  → let x'= f x
                                   in x' `par` let xs'= map f xs
                                               in xs' `pseq`Cons x' xs'

sum      = λxs. case xs of
                  Nil        → 0
                  Cons x xs  → case split xs of
                                 Pair l r → let r'= sum r
                                            in r' `par` let l'= sum l
                                                        in l' `pseq` x + l' + r'

inits    = λxs. case xs of
                  Nil        → Cons Nil Nil
                  Cons y ys  → Cons xs (inits (init xs))

tails    = λxs. case xs of
                  Nil        → Cons Nil Nil
                  Cons y ys  → Cons xs (tails ys)

segments = λxs. concat (map inits (tails xs))
```

Figure C.6: Hand Parallelised Definition of *mss* Operating on Unpartitioned Data

Appendix D

Parallelisation of Maximum Segment Sum

$$
\begin{array}{lcl}
mss_{par} & = & \mathcal{P}[\![\lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ (s+x)\ m \\
 & & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r''[\![\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \mathcal{P}[\![\lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ (s+x)\ m \\
 & & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r''[\![\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \mathcal{P}[\![\lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ (s+x)\ m \\
 & & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r''[\![\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \mathcal{P}[\![\textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ (s+x)\ m \\
 & & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r''[\![\ \varnothing
\end{array}
$$

Figure D.1: Automatic Parallelisation of mss_{par}

$$
\begin{array}{lcl}
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \mathcal{P}[\![\textbf{case}\ xs\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad Nil' \quad \rightarrow m \\
& & \qquad Cons'\ x \rightarrow max\ (s+x)\ m \\
& & \quad Join\ l\ r \quad \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
& & \qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
& & \qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r''[\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
& & \quad \mathcal{P}_{br}[\![Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad Nil' \quad \rightarrow m \\
& & \qquad Cons'\ x \rightarrow max\ (s+x)\ m]\!]\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \quad \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
& & \qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
& & \qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r''[\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \mathcal{P}[\![\textbf{case}\ x\ \textbf{of} \\
& & \qquad Nil' \quad \rightarrow m \\
& & \qquad Cons'\ x \rightarrow max\ (s+x)\ m]\!]\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
& & \qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
& & \qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r''[\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad \mathcal{P}_{br}[\![Nil' \quad \rightarrow m]\!]\ \varnothing \\
& & \qquad \mathcal{P}_{br}[\![Cons'\ x \rightarrow max\ (s+x)\ m]\!]\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
& & \qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
& & \qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r''[\![\ \varnothing
\end{array}
$$

Figure D.0 (cont.): Automatic Parallelisation of mss_{par}

$$
\begin{array}{lcl}
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad \mathcal{P}_{br}[\![Nil' \quad\ \rightarrow m]\!]\ \varnothing \\
 & & \qquad\qquad \mathcal{P}_{br}[\![Cons'\ x \rightarrow max\ (s+x)\ m]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r'']\!]\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow \mathcal{P}[\![m]\!]\ \varnothing \\
 & & \qquad\qquad \mathcal{P}_{br}[\![Cons'\ x \rightarrow max\ (s+x)\ m]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r'']\!]\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad \mathcal{P}_{br}[\![Cons'\ x \rightarrow max\ (s+x)\ m]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r'']\!]\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow \mathcal{P}[\![max\ (s+x)\ m]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r'']\!]\ \varnothing
\end{array}
$$

Figure D.-1 (cont.): Automatic Parallelisation of mss_{par}

$$
\begin{array}{lcl}
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow \mathcal{P}[\![max\ (s+x)\ m]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r'']\!]\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ (s+x)\ m \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad \textbf{in}\ mss_{par}\ l\ r'\ r'']\!]\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ (s+x)\ m \\
 & & \quad Join\ l\ r \quad\ \rightarrow \mathcal{P}[\![\textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad\ \ \textbf{in}\ \ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\quad\ \textbf{in}\ mss_{par}\ l\ r'\ r'']\!]\ \{l,r\} \\
 & & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ (s+x)\ m \\
 & & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ r'\ \text{`}par\text{`}\ \mathcal{P}[\![\textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\qquad\qquad\ \textbf{in}\ mss_{par}\ l\ r'\ r'']\!]\ \{l,r\} \\
 & & \qquad\qquad \Downarrow \\
mss_{par} & = & \lambda xs.\ \lambda m.\ \lambda s.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ (s+x)\ m \\
 & & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = mss_{par}\ r\ m\ s \\
 & & \qquad\qquad \textbf{in}\ r'\ \text{`}par\text{`}\ \textbf{let}\ r'' = f1\ r\ s \\
 & & \qquad\qquad\qquad\qquad\ \textbf{in}\ r''\ \text{`}par\text{`}\ mss_{par}\ l\ r'\ r''
\end{array}
$$

Figure D.-2 (cont.): Automatic Parallelisation of mss_{par}

$$
\begin{array}{lcl}
f1 & = & \mathcal{P}[\![\lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Nil' \quad\ \rightarrow m \\
& & \qquad\qquad Cons'\ x \rightarrow max\ x\ (x+m) \\
& & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = f1\ r\ m \\
& & \qquad\qquad \textbf{in}\ f1\ l\ r'][\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \mathcal{P}[\![\lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Nil' \quad\ \rightarrow m \\
& & \qquad\qquad Cons'\ x \rightarrow max\ x\ (x+m) \\
& & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = f1\ r\ m \\
& & \qquad\qquad \textbf{in}\ f1\ l\ r'][\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \lambda m.\ \mathcal{P}[\![\textbf{case}\ xs\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Nil' \quad\ \rightarrow m \\
& & \qquad\qquad Cons'\ x \rightarrow max\ x\ (x+m) \\
& & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = f1\ r\ m \\
& & \qquad\qquad \textbf{in}\ f1\ l\ r'][\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
& & \quad \mathcal{P}_{br}[\![Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Nil' \quad\ \rightarrow m \\
& & \qquad\qquad Cons'\ x \rightarrow max\ x\ (x+m)][\![\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = f1\ r\ m \\
& & \qquad\qquad \textbf{in}\ f1\ l\ r'][\![\ \varnothing \\
& & \qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
& & \quad Singleton\ x \rightarrow \mathcal{P}[\![\textbf{case}\ x\ \textbf{of} \\
& & \qquad\qquad Nil' \quad\ \rightarrow m \\
& & \qquad\qquad Cons'\ x \rightarrow max\ x\ (x+m)][\![\ \varnothing \\
& & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = f1\ r\ m \\
& & \qquad\qquad \textbf{in}\ f1\ l\ r'][\![\ \varnothing
\end{array}
$$

Figure D.-1: Automatic Parallelisation of $f1$

$$
\begin{array}{lcl}
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \mathcal{P}[\![\textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ x\ (x+m)]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = f1\ r\ m \\
 & & \qquad\qquad\qquad\qquad \textbf{in}\ f1\ l\ r']\!]\ \varnothing \\
 & & \qquad\qquad\qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad \mathcal{P}_{br}[\![Nil' \quad\ \rightarrow m]\!]\ \varnothing \\
 & & \qquad\qquad\qquad\qquad \mathcal{P}_{br}[\![Cons'\ x \rightarrow max\ x\ (x+m)]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = f1\ r\ m \\
 & & \qquad\qquad\qquad\qquad \textbf{in}\ f1\ l\ r']\!]\ \varnothing \\
 & & \qquad\qquad\qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad Nil' \quad\ \rightarrow \mathcal{P}[\![m]\!]\ \varnothing \\
 & & \qquad\qquad\qquad\qquad \mathcal{P}_{br}[\![Cons'\ x \rightarrow max\ x\ (x+m)]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = f1\ r\ m \\
 & & \qquad\qquad\qquad\qquad \textbf{in}\ f1\ l\ r']\!]\ \varnothing \\
 & & \qquad\qquad\qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad\qquad\qquad \mathcal{P}_{br}[\![Cons'\ x \rightarrow max\ x\ (x+m)]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = f1\ r\ m \\
 & & \qquad\qquad\qquad\qquad \textbf{in}\ f1\ l\ r']\!]\ \varnothing \\
 & & \qquad\qquad\qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad\qquad\qquad Cons'\ x \rightarrow \mathcal{P}[\![max\ x\ (x+m)]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = f1\ r\ m \\
 & & \qquad\qquad\qquad\qquad \textbf{in}\ f1\ l\ r']\!]\ \varnothing
\end{array}
$$

Figure D.-2 (cont.): Automatic Parallelisation of $f1$

$$
\begin{array}{lcl}
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow \mathcal{P}[\![max\ x\ (x+m)]\!]\ \varnothing \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = f1\ r\ m \\
 & & \qquad\qquad \textbf{in}\ f1\ l\ r']\!]\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ x\ (x+m) \\
 & & \quad \mathcal{P}_{br}[\![Join\ l\ r \rightarrow \textbf{let}\ r' = f1\ r\ m \\
 & & \qquad\qquad \textbf{in}\ f1\ l\ r']\!]\ \varnothing \\
 & & \qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ x\ (x+m) \\
 & & \quad Join\ l\ r \quad\ \rightarrow \mathcal{P}[\![\textbf{let}\ r' = f1\ r\ m \\
 & & \qquad\qquad\ \ \textbf{in}\ f1\ l\ r']\!]\ \{l, r\} \\
 & & \qquad\qquad \Downarrow \\
f1 & = & \lambda xs.\ \lambda m.\ \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \quad\ \rightarrow m \\
 & & \qquad\qquad Cons'\ x \rightarrow max\ x\ (x+m) \\
 & & \quad Join\ l\ r \quad\ \rightarrow \textbf{let}\ r' = f1\ r\ m \\
 & & \qquad\qquad \textbf{in}\ r'\ \texttt{`}par\texttt{`}\ f1\ l\ r'
\end{array}
$$

Figure D.-3 (cont.): Automatic Parallelisation of $f1$