Automated Parallelisation of code written in the Bird-Meertens Formalism

Joseph Windows

November 3, 2003

Abstract

The Bird-Meertens Formalism (BMF) is a calculus of higher order functions used to derive functional programs. It shows promise as a basis for a parallel model due to its highly parallelisable building blocks and the ease in which its code can be transformed through equalities.

There is a reasonably large body of work exploring parallel cost models and parallelisation techniques for BMF constructs and of related constructs in related languages. Very little of this work is focused on the development of a constructive framework for parallelising entire programs written in BMF. An actual implementation will need technology developed by such research.

This paper describes a partial implementation of a paralleliser for such programs and the issues encountered while developing it.

Contents

1 Introduction **4**

2 Related Work **6**

3 Preliminaries **7**
- 3.1 Project Context . . . 7
  - 3.1.1 Bird-Meertens Formalism . . . 7
  - 3.1.2 The Adl Language Project . . . 8
- 3.2 Code Transformation . . . 9
  - 3.2.1 Source Code . . . 9
  - 3.2.2 Target Code . . . 9
  - 3.2.3 Process . . . 9
- 3.3 Tools . . . 12
  - 3.3.1 Haskell . . . 12
  - 3.3.2 Simulator . . . 12

4 Implementation **13**
- 4.1 Normalisation . . . 13
  - 4.1.1 Right-Associative Function Compositions . . . 13
  - 4.1.2 The Temporary Sentinel . . . 14
  - 4.1.3 Second Pass: While and If Sweep . . . 14
  - 4.1.4 During parallelisation . . . 19
  - 4.1.5 Post-Parallelisation Cleanup . . . 19
- 4.2 Parallelising Individual Constructs . . . 19
  - 4.2.1 Transpose . . . 19
  - 4.2.2 Zip and Mask . . . 20
  - 4.2.3 Select . . . 20
- 4.3 Alltup / Allvec . . . 20
  - 4.3.1 Compaction . . . 21
  - 4.3.2 Vector Input . . . 21
  - 4.3.3 Following a While/If . . . 22
- 4.4 Address . . . 23
- 4.5 While/If . . . 24

4.5.1 If 24
4.5.2 While 25

5 Results **28**
5.1 Example 1: Maximum Segment Sum 28
5.2 Example 2: Remote Distance 31
5.3 Example 3: While Loop 33

6 Conclusion / Issues **34**
6.1 Compaction 34
6.2 Further Optimisation 35
6.3 Propagating Multiple Split Constructs 35
6.4 Reduction and Redistribution 36
6.4.1 Unwarranted Parallelism 36
6.4.2 Inside While Constructs 37
6.5 Locality 37
6.6 Non-Recursive Parallelisation 38
6.7 Type Information 38

A Paralleliser Code **42**

List of Figures

3.1 Sequential BMF constructs 10
3.2 Parallel BMF constructs 11
3.3 Parallelisation of map 11

4.1 Tuple flattening: finding the tuple structure 17
4.2 Tuple flattening: building the inverse structure 17
4.3 Transpose parallelisation techniques 20
4.4 Processing an alltup with vector input 22
4.5 Processing an alltup with tuple input 23
4.6 Parallelisation of an if construct 25
4.7 Parallelisation of a while construct 26

5.1 Execution times of MSS for various data sizes 29
5.2 Execution trace of MSS on 32 nodes 30
5.3 View of the MSS communications for the parallelised scan . . 31
5.4 Speedup of Remote Distance parallelisations 32
5.5 Execution times of the while loop example 33

Chapter 1

Introduction

The incentive for using parallelism is speedup; if we use N processors, we might expect to be able to execute a task N times faster than when using one processor. Of course, this is the ideal scenario, but the idea stays the same: speeding up execution time. Parallel software has not yet become normal practice because there is currently no established method for creating it. There are, however, a few possible approaches.

New languages can be constructed that are aimed specifically at parallel computing. Or parallel extensions can be added to existing languages. These methods would involve specifying programs directly with parallel components.

To make the process more automated, an intelligent parallelising compiler is needed. Automatic parallelisation has proven to be a difficult task [5].

The Adl language project's research includes exploring practical techniques for the automatic parallelisation of functional programming languages [2]. Intrinsic to the approach is the use of Bird-Meertens Formalism (BMF) as an intermediate form.

The end result of parallelising a sequential program maintains the original meaning of the code, but adds as much parallelism as possible. BMF code can be simply transformed using equalities, which greatly assists with parallelisation. The code can also be easily parallelised manually, which assists in testing and debugging.

There has been a significant amount of research exploring strategies for the parallelisation of BMF constructs [1, 15, 10] and of related constructs in related languages. Most of the parallelisation strategies for BMF have thus far been only manually applied, although some work towards an automated implementation is currently in progress [10].

A prototype needed to be developed to assess how effectively these techniques could be implemented. This paper describes a partial implementation of a paralleliser for BMF code, mainly using the techniques described in [1].

Section 2 lists work related to this topic. Section 3 expands on the concepts of the Bird-Meertens Formalism and the research of the Adl Project. It also introduces the general process to be implemented. The implementation is explained in section 4. Section 5 discusses experiments carried out to evaluate the performance of the paralleliser. Issues that arose during the process are discussed in a conclusion to the project in section 6.

The full code for the paralleliser is in appendix A.

Chapter 2

Related Work

The context for this work relates back to the early nineties with Hu and Sun [7, 20] discussing functional and algebraic approaches to synthesising systolic arrays – arrays of processors in which data flows synchronously between neighbours. Hu et al [8, 9] went on to discuss list homomorphisms – those functions on finite lists that promote through list concatenation – and their application to optimisation and parallelism.

Onoue, Hu, et al [22] constructed the calculational fusion system, HYLO, for deriving efficient programs. Fusion is where separate pieces of programs are fused together, in this case, transforming complex recursive definitions to linear recursive definitions, leading to an efficient program without intermediate data structures.

Roe [15] was first to investigate the use of BMF to derive efficient parallel programs for a variety of machines. Skillicorn [18] refined the process in terms of categorical data structures. With Cai [19], he developed a cost model for parallel functional programming.

Banerjee and Walinsky [21] created a data-parallel compiler for FP – a close relative to BMF, although less theoretical. They also worked towards eliminating intermediate data structures. There was less emphasis on optimisation in the FP form, however. Parallelisation occurred after transformation to imperative form.

The Adl Project [2], on the other hand, took the approach of transforming the code while still in functional form. BMF was used as an intermediate form for the efficient compilation of their own parallel functional language. This language was influenced primarily by SISAL [14]. One of the reasons SISAL itself wasn't used was that compilation by program transformation was difficult.

Martinaitas [13] created a simulator to measure the performance of sequential and parallel BMF code.

Chapter 3

Preliminaries

Section 3.1 discusses the motivation behind this project, discussing BMF and the Adl project. Section 3.2 describes the constructs of BMF, and the general notion of how to implement parallelisation. Section 3.3 describes the tools used during the project.

3.1 Project Context

Here I describe BMF and its application to parallel programming. I then expand on the research of the Adl project, of which, this paper will be a contribution.

3.1.1 Bird-Meertens Formalism

Bird-Meertens Formalism (BMF), or Squiggol, is a calculus for deriving and expressing functional programs. It consists of a set of higher-order functions that operate mainly on lists.

BMF embraces the transformational approach towards programming. That is, starting with the most obvious and concise initial solution to a given specification, without concern for efficiency, successive transformations can create a more and more efficient program. These transformations are based on equalities, so are such that the meaning of the program is guaranteed not to change.

Being much easier to transform than most other programming notations [3, 4], it is not only very useful for exploring program optimisation techniques, but also shows promise as a basis for a parallel model. In the process of parallelising programs, an obvious and concise initial solution, without concern for parallelism, is transformed into a parallel version [4, 15].

BMF is based on a computational model of categorical data types and their accompanying operations. The model is suited to developing parallel software. It is architecture-independent – programs are simply compositions

of operations on values of data types – and it is abstract – details of the operations are hidden. The theory of this model and its parallel application have been realised [6, 10], so it now needs to become concrete [17].

The basic building blocks of BMF, such as the functions map and reduce, have very fast parallel implementations. The sequential map applies a function to each element of a list. It is inherently parallel, as the function applications can be performed independently on different processors. The function reduce converts a list to a single value via an associative binary operator, and can be evaluated on a tree-like structure. It consequently allows significant parallelism [16].

There has been a significant amount of research exploring strategies for the parallelisation of BMF constructs [1, 10, 15]. Most of these strategies have thus far been only manually applied, although some work towards an automated implementation is currently in progress [10].

3.1.2 The Adl Language Project

This research is a contribution to a body of work that is exploring the efficient compilation of data-parallel, functional-programming languages to distributed-memory, parallel architectures [2]. The research of the Adl Language Project has embodied:

- the design and formal specification of Adl, a small, polymorphic, non-recursive, data-parallel language,
- a mathematical formulation of the language's translation to BMF and optimisation to more efficient BMF,
- the design and construction of a parallel, distributed-memory abstract machine to which the compilation process is targeted,
- the development of an executable, natural-semantic description of the translation into BMF, and an optimiser for the resulting code,
- the development of parallelisation strategies for BMF code.

Adl was implemented purposely as a small functional language to ensure that BMF code produced from Adl source could be easily manipulated as a mathematical expression. Its other main characteristic is its support of data parallelism.

Some built-in higher-order functions are provided, including map, reduce and scan and variations of reduce and scan with no base value.[1] Through these functions, Adl is primarily designed for the efficient application of operations over its vector data type; a possibly nested, one-dimensional list

[1]The base value for a function is the value returned when the input vector is empty.

supporting fast random access. Adl also provides the standard selection and repetition constructs, if and while.

The semantic description of the translation into BMF, followed by optimisation, results in BMF code that, in many cases, is close in efficiency to hand coded BMF programs. These automatically generated programs are in sequential BMF.

The topic of this project addresses one of a number of areas where interesting research remains to be carried out. Strategies for the parallelisation of BMF code have been developed and tested, but this is only the preliminary research [1]. Substantial work needs to be undertaken to make these strategies constructive, that is, embedded in a working implementation.

3.2 Code Transformation

This section introduces the sequential and parallel constructs of the Bird-Meertens Formalism. It goes on to explain the general process of parallelisation and the desired outcome.

3.2.1 Source Code

Programs to be parallelised are submitted in a functional form of sequential BMF. The sequential constructs are described in figure 3.1. The source program should be optimised [1] before being parallelised.

3.2.2 Target Code

The resulting parallelised code will comprise a combination of sequential and parallel BMF constructs in functional form. The parallel constructs are described in figure 3.2. Note that this implementation does not currently make use of $\mathsf{splitpair}_p$.

split, denoted $\lhd|\rhd_p$, is the principal distribution construct. It creates a vector, distributed over p nodes, whose elements are sub-vectors of the input vector.

parallel reduce by concatenation, $+\!\!+/^{\|}$, is the principal reduction construct. It concatenates distributed sub-vectors onto one node. It is the inverse of $\lhd|\rhd_p$.

3.2.3 Process

The parallelisation process uses the data parallel model. The aim is to parallelise vector data wherever possible. The other allowed data structures are scalar values and tuples. Scalars are obviously not parallelised. Tuples are not directly parallelised in this implementation. If a tuple structure contains vectors at some level, parallelisation of those vectors will be attempted.

Construct	*Description*
$f*$	`B_map` applies f to all elements of a vector.
$\oplus/$	`B_reduce` applies $\oplus$ to each pair of vector elements.
$\oplus/\!/$	`B_scan` produces a vector of cumulative reductions.
$g \cdot h$	`B_comp` (composition) produces a composite expression.
$\#$	`B_length` returns the length of a vector.
!	`B_index` returns the i^{th} element of a vector.
Υ	`B_zip` combines two vectors element-wise.
$\mathsf{transpose}_{(a,b)}$	`B_transpose` transposes dimensions a and b of a nested vector.
${}^{n}\pi_i$	`B_addr` (address) returns the i^{th} element of a tuple.
$(f_1, \ldots, f_n)^\circ$	`B_alltup` applies f_1 to f_n to copies of the input, producing a tuple of the results.
$[f_1, \ldots, f_n]^\circ$	`B_allvec` Applies f_1 to f_n producing a vector of results.
distl	`B_distl` combines a value with every element of a vector, producing a vector of tuples.
repeat	`B_repeat` produces a vector containing a specified number of copies of a value.
select	`B_select` returns elements of one vector, indexed by the scalar elements of a second.
mask	`B_mask` returns elements of one vector based on the boolean elements of a second.
priffle	`B_priffle` interleaves two vectors based on the boolean elements of a third.
id	`B_id` is the identity function.
iota	`B_iota` produces a vector containing integers from 0 to one less than its input.
if (p) then (c) else (a)	`B_if` applies either c or a based on the evaluation of p.
while (p) do (f)	`B_while` applies f until p evaluates to false.
c	`B_con` denotes a constant value: `B_int` i, `B_real` r, `B_true` or `B_false`.
$\oplus$	`B_op` denotes an operator. Some of the above are operators, and others include `B_conc`, `B_plus` and `B_eq`.

Figure 3.1: Sequential BMF constructs

Construct	*Description*
$\lhd\|\rhd_p$	`P_split` breaks a vector into p distributed sub-vectors.
$\mathsf{splitpair}_p$	`P_splitpair` applies $\lhd\|\rhd_p$ to two vectors symmetrically.
$\mathsf{distl}^{\|}$	`P_distl` broadcasts a value to the elements of a distributed vector.
$\mathsf{repeat}^{\|}$	`P_repeat` distributes a specified number of copies of a value.
$f*^{\|}$	`P_map` applies f to distributed sub-vectors.
$\oplus/^{\|}$	`P_reduce` reduces distributed sub-vectors with $\oplus$.
$\oplus/\!/^{\|}$	`P_scan` scans distributed sub-vectors with $\oplus$.
$\mathsf{select}^{\|}$	`P_select` distributes elements of one distributed vector, indexed by the scalar elements of a second.
$\mathsf{distpriff}^{\|}$	`P_distpriff` produces p distributed vectors of 3-tuples, containing sub-vectors of 3 input vectors.
$\oplus$	`B_op` There are also parallel operators, such as `P_conc`.

Figure 3.2: Parallel BMF constructs

$$
\begin{array}{ll}
\quad f* & \{f* = \mathsf{id} \cdot f*\} \\
\Rightarrow \mathsf{id} \cdot f* & \{\mathsf{id} = +\!\!+/^{\|} \cdot \lhd|\rhd_p\} \\
\Rightarrow +\!\!+/^{\|} \cdot \lhd|\rhd_p \cdot f* & \{\lhd|\rhd_p \cdot f* = (f*)*^{\|} \cdot \lhd|\rhd_p\} \\
\Rightarrow +\!\!+/^{\|} \cdot (f*)*^{\|} \cdot \lhd|\rhd_p &
\end{array}
$$

Figure 3.3: Parallelisation of map

Starting at the end[2] of the program, the aim is to parallelise each construct in turn. This is done by first appending the identity $+\!\!+/^{\|} \cdot \lhd|\rhd_p$ to constructs with vector output [1].

The aim now is to propagate the $\lhd|\rhd_p$ through to the start of the program, using identities to parallelise sequential constructs. To illustrate, figure 3.3 shows the parallelisation of a lone map.

The ideal outcome is for the $\lhd|\rhd_p$ to move through the entire program, resulting in an entirely parallelised program. In some cases this is difficult, if not impossible, to achieve. Some expressions *require* all data to be reduced to one node. The aim is to parallelise as much of a program as possible, even if that means distributing data more than once during the program's execution

[2]The end refers to the left-hand side of the code as it is written down. Similarly, the start is the right-hand side.

3.3 Tools

Given a functional representation of BMF code, the aim is to match each construct – or composition of constructs – and transform it into it's parallel version from a specification.

3.3.1 Haskell

I used the functional language Haskell for the parallelisation of BMF constructs [11]. Specifically, I used the Hugs 98 Interpreter [12].

As Haskell is a functional language, the matching is done by evaluation of expressions. The following expression performs the parallelisation on the sequential code `<seq_code>`:

```
pl <procs> <seq_code>
```

where `<procs>` is the number of processors to be used At this stage that is all the input needed from the user.

Pretty Printer I also developed a simple pretty printer using Haskell and Perl to convert Haskell expressions into regular BMF strings, viewable as HTML.

For example, the parallel version of map is, in Haskell:

```
B_comp
   (P_reduce (B_op B_concat))
   (B_comp (P_map (B_map F)) (P_split (B_num 4) (B_num 0)))
```

after conversion using the Haskell pretty printer:

```
"++/''.(F)**''.split%4%"
```

completed by Perl:

```
++<font face=symbol>/</font>'' . <font face=symbol>(</font>F
   <font face=symbol>)**</font>'' . split<font size=1>4</font>
```

and viewed in HTML:
++/" . (F)**" . split$_4$

3.3.2 Simulator

I had access to a parallel simulator for BMF code, developed by Paul Martinaitas [13]. I used it to measure the performance of the resulting parallel code against the original sequential program. It produces a simulation and subsequent visualisation of parallel BMF code on a virtual architecture. The detailed simulator models various network topologies with configurable bandwidth and latency values. It returns raw timings and highly detailed trace messages.

Section 5 contains the results of experiments using the simulator.

Chapter 4

Implementation

The method involves proceeding through the program from left to right, parallelising each construct. The more complex functions – aggregates (alltup and allvec), selection (if) and repetition (while) – are parallelised recursively. The techniques for these functions are more involved than the others and consequently require deeper description.

Section 4.1 describes the process of normalising the code before parallelisation begins. Section 4.2 explains the parallelisation of the majority of the constructs.

Section 4.3 describes the techniques for alltup and allvec. Section 4.4 discusses the need to match and consequently evaluate addresses during parallelisation. Section 4.5 describes the techniques for while and if.

4.1 Normalisation

Code can be written in an exponential number of ways. This creates a problem when trying to match constructs in a certain pattern. Normalisation reduces this number by making the code predictable. There is no effect on the meaning of the code, because, as per usual with BMF, normalisation is a transformation using identities. I normalise the code so that there is only one (or at most two) expressions that each rule applies to.

4.1.1 Right-Associative Function Compositions

The sequence of expressions is controlled by the binary composition operator. Because the operator is itself an expression, it is recursive, and can consequently be nested. The following pieces of code are all identical in meaning:

```
B_comp (B_comp A B) (B_comp C x)
B_comp (B_comp A (B_comp B C)) x
B_comp A (B_comp B (B_comp C x))          (4.1)
```

where `A`, `B` and `C` each represent a single BMF function and `x` represents any BMF expression.[1]

I would have to deal with all three scenarios if the code wasn't normalised. So, before I attempt to parallelise the code, I nest the composition constructs to the right, as in (4.1). With this normalisation, if I am trying to match B composed with C, I only have to consider the simple expression:

```
B_comp B (B_comp C x)
```
(4.2)

4.1.2 The Temporary Sentinel

Another case has to be considered separately:

```
B_comp B C
```
(4.3)

This occurs when there are no more constructs following `C`. To avoid having to add a separate rule for this case, a temporary construct `B_TEMP` is composed to the start of the program. The expression (4.3), now becomes:

```
B_comp B (B_comp C B_TEMP)
```

which matches (4.2).

The `B_TEMP` construct is removed after parallelisation during a sweep of the code. This process is discussed further in section 4.1.5.

4.1.3 Second Pass: While and If Sweep

After normalising compositions, I make a second pass of the code searching for while and if constructs. I also want to make them as predictable as possible, as they are the only constructs which impose additional constraints upon their type.

For while the condition and the body of the loop must accept the same data type, on the initial and consequent iterations. This demands that the body of the loop must also output data of this type.

For if constructs, the consequent, alternative and predicate must all accept the same data type. The consequent and the alternative must also output data of a corresponding type, which may or may not be the same as their input.

The method I used to make the constructs predictable was restricting their possible input and output data types to either a scalar, vector or a flat[2] tuple. I restrict tuples because dealing with one simplified case greatly decreases the complexity of propagating parallelism. This will become clear after reading this section, and is further discussed in section 6.7.

[1] An expression may be a composition of functions. From here on, these definitions for upper and lower-case letters will be used.

[2] One that isn't nested.

Identities The following procedure and other parts of this implementation use the following identities:

$$(f_1, \ldots, f_n)^\circ \cdot x \Rightarrow (f_1 \cdot x, \ldots, f_n \cdot x)^\circ$$

$${}^n\pi_i \cdot (f_1, \ldots, f_n)^\circ \Rightarrow f_i$$

The first identity applies the alltup to the construct to which it is composed. This is called compaction as it compacts two constructs into one. The second identity simply evaluates the address operator, ${}^n\pi_i$, returning the i^{th} element of the tuple.

Tuple Flattening I want to flatten tuples that are nested. Take the nested alltup:

$$((f,\, g)^\circ,\, h)^\circ$$

I append the following identity to the left of the alltup:

$$(({}^3\pi_1\,, {}^3\pi_2)^\circ, {}^3\pi_3)^\circ \cdot ({}^2\pi_1 \cdot {}^2\pi_1\,, {}^2\pi_2 \cdot {}^2\pi_1\,, {}^2\pi_2)^\circ \tag{4.4}$$

name it the suffix and denote it $S_1 \cdot S_2$. Leaving S_1, I compact S_2 with the original alltup, and evaluate the addresses:

$$(({}^3\pi_1\,, {}^3\pi_2)^\circ, {}^3\pi_3)^\circ \cdot (f,\, g,\, h)^\circ \tag{4.5}$$

which is the desired result.

S_1 is simply an alltup with identical structure to the original construct. Its elements are an enumeration of the addresses ${}^n\pi_1$ to ${}^n\pi_n$, where n is the total number of expressions in the original construct: 3 in the example.

Compaction I first normalise the elements of the while/if as described in 4.1. Consequently the process described here may be executed recursively. I then compact any compositions of an alltup with any other construct into the alltup:

As I recursively compact an element I also use some identities: removing redundant id constructs, and evaluating addresses.

Neither of these identities should need to be used with the original, optimised code, but the process of compaction can produce the need for them. For example:

$$(f,\ \mathsf{id})^\circ \cdot x \Rightarrow (f \cdot x,\ \mathsf{id} \cdot x)^\circ \Rightarrow (f \cdot x,\ x)^\circ$$

Suffix for if The method for producing the suffix is different for the two constructs. The if suffix is produced here, but the while suffix is produced later. The reason for this is explained later in the section.

As previously stated, the structure of S_1 is the same as that of the original alltup expression. It may seem sufficient to copy the structure of either the consequent or alternative expression, however they may not convey the same information. For example:

$$\begin{aligned} \mathit{Consequent} &: ((f \cdot {}^2\pi_1,\ g \cdot {}^2\pi_2)^\circ,\ \mathsf{id})^\circ \\ \mathit{Alternative} &: (\mathsf{id},\ (x \cdot {}^2\pi_1,\ y \cdot {}^2\pi_2)^\circ)^\circ \end{aligned} \tag{4.6}$$

They both output data of the same type, but the same information cannot be gained simply by observing their nested alltups. Their respective structures would be observed as:

$$((*,*),*) \text{ and } (*,(*,*))$$

I need to acquire a unifier for the most general structure that can be retrieved from combining the two expressions. The output structure of the example is:

$$((*,*),(*,*))$$

and the identity can be produced as in (4.4). It is then appended to the start of both expressions. The consequent – and the alternative similarly – becomes:

$$(({}^4\pi_1\,,{}^4\pi_2)^\circ,({}^4\pi_3\,,{}^4\pi_4)^\circ)^\circ \cdot (f \cdot {}^2\pi_1\,, g \cdot {}^2\pi_2\,, {}^2\pi_1\,, {}^2\pi_2)^\circ$$

To acquire the unifier, I use the following `Structure` data type:

```
Structure = Addr | Alltup Int [Structure]
```

which is analogous to the tuple structure. The integer value is included for the latter process of counting how many elements are in the structure. It is initially set to 0.

I observe the two elements simultaneously, by evaluating an expression that takes them as arguments of a tuple. If both components are an alltup, the expression creates an `Alltup` structure and recursively maps itself to the alltup elements. If only one of the components is an alltup, a dummy vector of `B_NULL` values is created for the other, and it proceeds as above. If neither component is an alltup, an `Addr` is returned. The process is illustrated in figure 4.1 for the previous example. `s1` represents the expression to be evaluated. Due to the symmetrical behaviour of the structure, the right-hand side is not expanded completely for brevity.

The final result has the structure of the unifier. I then populate the integer count at each level. This is done by again recursing through the

$$
\begin{aligned}
&\texttt{s1 (}\ ((f \cdot {}^2\pi_1\,, g \cdot {}^2\pi_2)^\circ, \mathsf{id})^\circ\texttt{ ,}\ (\mathsf{id}\,, (x \cdot {}^2\pi_1\,, y \cdot {}^2\pi_2)^\circ)^\circ\ \texttt{)}\\
\Rightarrow\ &\texttt{Alltup 0 [s1 (}(f \cdot {}^2\pi_1\ \texttt{,}g \cdot {}^2\pi_2)^\circ, \mathsf{id}\texttt{) ,s1 (}\mathsf{id}, (f \cdot {}^2\pi_1\,, g \cdot {}^2\pi_2)^\circ\texttt{)]}\\
\Rightarrow\ &\texttt{Alltup 0 [Alltup 0 [s1 (}f\texttt{,NULL) ,s1 (}g, \mathsf{NULL}\texttt{)],[...]]}\\
\Rightarrow\ &\texttt{Alltup 0 [Alltup 0 [Addr,Addr], Alltup 0 [Addr,Addr]]}
\end{aligned}
$$

Figure 4.1: Tuple flattening: finding the tuple structure

$$
\begin{aligned}
&\mathsf{s2}\,((^2\pi_1, {}^2\pi_2)^\circ, (^2\pi_1, {}^2\pi_2)^\circ)^\circ\\
\Rightarrow\ &+\!\!+/\,[\,(\cdot\,{}^2\pi_1) * \mathsf{s2}\,(^2\pi_1, {}^2\pi_2)^\circ,\ (\cdot\,{}^2\pi_2) * \mathsf{s2}\,(^2\pi_1, {}^2\pi_2)^\circ\,]\\
\Rightarrow\ &+\!\!+/\,[\,(\cdot\,{}^2\pi_1) * \cdot +\!\!+/\,[\mathsf{s2}\ {}^2\pi_1, \mathsf{s2}\ {}^2\pi_2],\ (\cdot\,{}^2\pi_2) * \cdot +\!\!+/\,[\mathsf{s2}\ {}^2\pi_1, \mathsf{s2}\ {}^2\pi_2]\,]\\
\Rightarrow\ &+\!\!+/\,[\,(\cdot\,{}^2\pi_1) * \cdot +\!\!+/\,[[{}^2\pi_1], [{}^2\pi_2]],\ (\cdot\,{}^2\pi_2) * \cdot +\!\!+/\,[[{}^2\pi_1], [{}^2\pi_2]]\,]\\
\Rightarrow\ &+\!\!+/\,[\,(\cdot\,{}^2\pi_1) * [{}^2\pi_1, {}^2\pi_2],\ (\cdot\,{}^2\pi_2) * [{}^2\pi_1, {}^2\pi_2]\,]\\
\Rightarrow\ &+\!\!+/\,[\,[{}^2\pi_1 \cdot {}^2\pi_1, {}^2\pi_2 \cdot {}^2\pi_1],\ [{}^2\pi_1 \cdot {}^2\pi_2, {}^2\pi_2 \cdot {}^2\pi_2]\,]\\
\Rightarrow\ &[\,{}^2\pi_1 \cdot {}^2\pi_1, {}^2\pi_2 \cdot {}^2\pi_1,\ {}^2\pi_1 \cdot {}^2\pi_2, {}^2\pi_2 \cdot {}^2\pi_2\,]
\end{aligned}
$$

Figure 4.2: Tuple flattening: building the inverse structure

structure, but leaving an expression at each level to evaluate the count. This allows the count to be propagated from the peaks back to the base of the structure. Each count sums the elements nested inside it: 1 for each `Addr`, and the corresponding `i` for each `Alltup`.

S_1 is then created from the `Structure`. Each `Addr` is replaced with a ${}^n\pi_i$ with i one greater than the last. n is the count of the outer `Alltup`. The counts on the inner levels keep track of i. In the example, i starts at 3 in the second element, because the first element had a count of 2.

S_2 is created by recursing through S_1. The following is mapped to each element i, of the outer alltup of S_1. If the element is an alltup itself, the process is again mapped to the elements, but with the expression $\{\cdot\,{}^n\pi_i\}$ mapped to the result. If the element is not an alltup, a vector of one element is returned containing ${}^n\pi_i$. The resulting vectors are concatenated.

Figure 4.2 illustrates the process, again, for the previous example, using BMF notation. `s2` is the expression to be evaluated. The result is a single vector, which is returned as the argument of a single alltup.

Prefix Appending the suffix flattened the output of the construct. I also want to flatten the input. The process is the same for both while and if. I

find the structure of the input by observing the expressions that act upon it. The expressions access the structure of the input through compositions of addresses.

To illustrate, take the consequent from (4.6). The first element contains a ${}^{2}\pi_1$, hence the input is a tuple of length 2. No other information can be gathered from the consequent or the alternative. Suppose the predicate of the if was:

$$< \cdot \, (\, 0 \, , {}^{3}\pi_1 \cdot {}^{2}\pi_2 \,)^{\circ}$$

The ${}^{3}\pi_1$ provides additional information. The structure becomes:

$$(\, * \, , (\, * \, , * \, , * \,) \,)$$

and the identity for the input is:

$$({}^{4}\pi_1 \, , ({}^{4}\pi_2 \, , {}^{4}\pi_3 \, , {}^{4}\pi_4)^{\circ})^{\circ} \cdot ({}^{2}\pi_1 \, , {}^{3}\pi_1 \cdot {}^{2}\pi_2 \, , {}^{3}\pi_2 \cdot {}^{2}\pi_2 \, , {}^{3}\pi_3 \cdot {}^{2}\pi_2)^{\circ}$$

Because this comes before the construct in question I call it the prefix and denote it $P_1 \cdot P_2$.

I use the `Structure` type again to create P_1. Before looking at each expression, I reverse the associativity of the compositions inside them: normalising from right to left. This is because I am interested in the right-hand side. If the expression ends in an alltup, I check its elements one by one. If I observe a address, ${}^{n}\pi_i$, I create an `Alltup` – if I have not already created one – with n elements: each comprising an Addr. If another address, ${}^{m}\pi_j$, is composed to the first, I repeat the above process inside the j^{th} element of the existing `Alltup`.

I then create P_1 and P_2 using the same process that was used to create the if suffix.

Suffix for while The reason I did not create the suffix for a while construct at the same time as for an if is that the former has matching input and output types. Consequently the suffix is identical to the prefix, and it is redundant to produce them individually.

Completion For if – and while – I now have:

- $\mathsf{if}\,(p)\,\mathsf{then}\,(S_1 \cdot S_2 \cdot c)\,\mathsf{else}\,(S_1 \cdot S_2 \cdot a) \cdot P_1 \cdot P_2$
- $\mathsf{while}\,(p)\,\mathsf{do}\,(S_1 \cdot S_2 \cdot f) \cdot P_1 \cdot P_2$

To complete the process, I move S_1 outside and P_1 inside the construct:

- $S_1 \cdot \mathsf{if}\,(p \cdot P_1)\,\mathsf{then}\,(S_2 \cdot a \cdot P_1)\,\mathsf{else}\,(S_2 \cdot c \cdot P_1) \cdot P_2$
- $S_1 \cdot \mathsf{while}\,(p \cdot P_1)\,\mathsf{do}\,(S_2 \cdot y \cdot P_1) \cdot P_2$

Removing S_1 leaves the flat $S_2 \cdot x$ as in (4.5). Appending P_1 to the front leaves the input flattened and consequently the while/if is now predictable.[3]

[3] As defined at the beginning of this section.

4.1.4 During parallelisation

The complex BMF constructs can contain compositions in their arguments. I normalise their arguments as I come across them during parallelisation, not beforehand. The main reason for this is that the parallelisation process may cause expressions to be added to these arguments. There is no assurance that these expressions are composed in the same, predictable way. Hence, they need to be normalised before continuing. So normalising at the beginning is redundant.

Also, leaving the `B_TEMP` construct at the end of each composition sequence can result in undesirable side effects; for example when combining alltup expressions (4.3).

4.1.5 Post-Parallelisation Cleanup

After the parallelisation process is completed for the whole program, or a complex construct, I 'sweep' through, cleaning up the parallelised code. This process simply involves removing the `B_TEMP` construct that was appended during the normalisation process, and also any instance of the expression $+\!\!+/^{\|} \cdot \lhd|\rhd_p$ that has not been separated and is still in the code.

4.2 Parallelising Individual Constructs

Most of the BMF constructs are parallelised individually using techniques described in [1]. These constructs are map, reduce, scan, length, transpose, zip, select, distl, repeat, mask, priffle and index. For example, figure 3.3 shows the technique for map.

Sections 4.2.1 to 4.2.3 explain techniques which are non-trivial. These apply to transpose, zip, mask and select.

4.2.1 Transpose

Two techniques are given for parallelising transpose. The first cannot be used if the number of nodes is greater than the length of the input vector. The second can only be used in the scenario where both dimensions of the vector to be swapped are greater than 0.

I use the second where possible as no communication is involved. The condition can easily be tested by observing the arguments of the construct.

If the condition fails, I attempt to use the first method. This condition relies on the input to the program, and consequently must be tested during run-time. If both conditions fail, the construct is not parallelised.

transposeis not processed in isolation. The placement of $+\!\!+/^{\|}$ and split constructs depend on whether the parallelisation process continues before and after the transpose.

$$+\!\!+/^{\parallel} \cdot (\mathsf{transpose}_{(0,1)}) *^{\parallel} \cdot \lhd|\rhd_p \qquad (a)$$

$$\mathsf{if}\,(< \cdot (p\,, \#)^{\circ})\,\mathsf{then}\,(\mathsf{transpose}_{(0,1)})\,\mathsf{else}\,(x \cdot \lhd|\rhd_p) \qquad (b)$$

$$\mathsf{if}\,(< \cdot (p\,, +/^{\parallel} \cdot (\#)*^{\parallel})^{\circ})\,\mathsf{then}\,(\mathsf{transpose}_{(0,1)} \cdot +\!\!+/^{\parallel})\,\mathsf{else}\,(x) \cdot (f*) *^{\parallel} \cdot \lhd|\rhd_p \qquad (c)$$

$$\text{where } x = +\!\!+/^{\parallel} \cdot ([\mathsf{id}]^{\circ}) *^{\parallel} \cdot \mathsf{transpose}_{(0,1)}{}^{\parallel} \cdot (! \cdot (\mathsf{id}, 0)^{\circ})*^{\parallel}$$

Figure 4.3: Transpose parallelisation techniques

Figure 4.3 illustrates the process for (a) $\mathsf{transpose}_{(1,2)}$, (b) $\mathsf{transpose}_{(0,1)}$ and (c) $\mathsf{transpose}_{(0,1)} \cdot f*$.

4.2.2 Zip and Mask

The code resulting from the parallelisation of both zip and mask assumes that the two input vectors were of the same length. It also assumes that, for some p, multiple applications of $\lhd|\rhd_p$ will distribute a vector of length N symmetrically. This is because the parallel versions of zip (below) and mask contain two separate applications of $\lhd|\rhd_p$.

$$+\!\!+/^{\parallel} \cdot \Upsilon^{\parallel} \cdot (\Upsilon) *^{\parallel} \cdot (\lhd|\rhd_p \cdot {}^2\pi_1\,,\, \lhd|\rhd_p \cdot {}^2\pi_2)^{\circ}$$

The vectors need to be distributed symmetrically so that the parallel map, $\Upsilon^{\parallel}$, is evaluated correctly [1].

The splitpair construct would achieve this symmetry, but using it increases the complexity of the parallelisation process. To propagate the parallelism, I would have to keep track of two vectors, while trying to move the splitpair through the code.

4.2.3 Select

The technique for parallelising select is valid, but not ideal. The whole source vector is broadcast to every node before the selection commences. The cost of this would tend to outweigh the gains made by parallel computation. Future techniques will probably use primitives from the underlying architecture [1].

4.3 Alltup / Allvec

alltup and allvec constructs can act upon any data type. If the input is a vector, parallelism may be propagated to the end of each element individ-

ually, and ultimately out of the construct through common sub-expression elimination. Section 4.3.2 describes the process.

It is not as simple to eliminate common sub-expressions when they are arbitrarily nested. This is the case for tuples. The only constructs[4] that can output a tuple which may contain parallelisable vectors are while, if and alltup.

As explained in section 4.1.3, tuple output from while/if constructs is flat. Propagating parallelism with a flat tuple as input uses the process as for vector input, applied to each element. This is described in section 4.3.3.

That leaves only the alltup able to produce arbitrarily nested tuples. Section 4.3.1 explains the method of compaction which avoids the need for the more complex propagation.

4.3.1 Compaction

alltups that are composed together are automatically compacted into a single alltup expression:

$$(x_1, \ldots, x_n)^\circ \cdot (y_1, \ldots, y_n)^\circ \Rightarrow (x_1 \cdot (y_1, \ldots, y_n)^\circ, \ldots, x_n \cdot (y_1, \ldots, y_n)^\circ)^\circ$$

This is done to simplify the process of propagating the parallisation. Because the alltup constructs are compacted, I don't have to propagate parallelism between them. Issues resulting from this and possible solutions are discussed in section 6.1.

The same applies for an allvec composed to an alltup:

$$[x_1, \ldots, x_n]^\circ \cdot (y_1, \ldots, y_n)^\circ \Rightarrow [x_1 \cdot (y_1, \ldots, y_n)^\circ, \ldots, x_n \cdot (y_1, \ldots, y_n)^\circ]^\circ$$

4.3.2 Vector Input

I attempt to parallelise the elements of both alltups and allvecs individually. Once I can propagate the parallelism no further, the aim, for vector input, is to try and propagate the splits outside of the construct.

This can only happen if at least one of the elements starts with a split. If none do, either the input type isn't a vector, or none of the elements could be parallelised sufficiently. Figure 4.4 illustrates the process.

To begin, I parallelise each element. I then check whether any of them now start with a split, by recursing to the inner most composition. If they do, I want to move them out of the construct. While doing so, I must make sure that each element is expecting an element of the same distribution. I move the split outside by performing the replacements in the following table to the elements (where *con* is an arbitrary constant):

[4] id and ${}^n\pi_i$ can as well, but they are not parallelised directly.

$$
\begin{aligned}
& (\mathsf{id}\,,\ \underline{\lhd|\rhd_p \cdot g*})^\circ \\
\Rightarrow\ & (\underline{\mathsf{id}}\,,\ (g*) *^{\|} \cdot \lhd|\rhd_p)^\circ \\
\Rightarrow\ & (+\!\!+/^{\|} \cdot \underline{\lhd|\rhd_p}\,,\ (g*) *^{\|} \cdot \underline{\lhd|\rhd_p})^\circ \\
\Rightarrow\ & (+\!\!+/^{\|},\ (g*)*^{\|})^\circ \cdot\ \lhd|\rhd_p
\end{aligned}
$$

Figure 4.4: Processing an alltup with vector input

Ends with	Replacement
con	*con*
$X \cdot \lhd\|\rhd_p$	x
$\lhd\|\rhd_p$	id
x	$x \cdot +\!\!+/^{\|}$

This replacement is an identity as follows. For the fourth case in the table,[5] I skip appending the identity $+\!\!+/^{\|} \cdot \lhd|\rhd_p$. This corresponds to the third line in figure 4.4. Every element now starts with split. I then use common sub-expression elimination to bring them out of the construct. The split can now be propagated through the next construct (if it exists).

I propagate the parallelism dependant on only one element ending in splitȦdvantages and disadvantages of this are discussed in section 6.3.

4.3.3 Following a While/If

Parallelising an alltup/allvec that follows a while or an if, similar to that for vector input. I want to propagate splits into the while/if. Because these constructs can output both flattened tuples or vectors,[6] I need to check whether the elements start with either of the following:

1. $\lhd|\rhd_p$
2. $\lhd|\rhd_p \cdot {}^n\pi_i$

I want to produce a predictable construct to propagate into the while/if: a split or an alltup containing simple elements. If one of the elements of the alltup/allvec starts with a split I proceed as in section 4.3.2. While checking for the second case, if an element starts with an alltup/allvec, I recursively check its elements as well. This is not necessary in the first case, as any split would already have been propagated.

[5] To be strictly correct the $+\!\!+/^{\|} \cdot \lhd|\rhd_p$ would also be appended in the first case. But this is trivial as $x \cdot c = c$ for all constants c.

[6] See section 4.1.3.

$$
\begin{aligned}
&([\underline{\lhd|\rhd_p \cdot f*} \cdot {}^2\pi_1 \, , {}^2\pi_1]^\circ, {}^2\pi_2)^\circ \\
\Rightarrow\ &([(f*) *^{\|} \cdot \underline{\lhd|\rhd_p \cdot {}^2\pi_1} \, , \underline{{}^2\pi_1}]^\circ, \underline{{}^2\pi_2})^\circ \\
\Rightarrow\ &([(f*) *^{\|} \cdot {}^2\pi_1 \, , +\!\!+/^{\|} \cdot {}^2\pi_1]^\circ, {}^2\pi_2)^\circ \cdot (\lhd|\rhd_p \cdot {}^2\pi_1, {}^2\pi_2)^\circ
\end{aligned}
$$

Figure 4.5: Processing an alltup with tuple input

Figure 4.5 is an example of the process for the second case. The input is a tuple of length n; of which at least one element is a vector. While checking each element I record each time I come across an instance of $\lhd|\rhd_p \cdot {}^n\pi_i$, adding each unique i to a set I. I now repeat the same process as for the vector input above, $\forall\ i \in I$. That is, for each i, for any element starting with ${}^n\pi_i$, I perform the following replacements:

Ends with	Replacement
$\lhd\|\rhd_p \cdot {}^n\pi_i$	${}^n\pi_i$
${}^n\pi_i$	$+\!\!+/^{\|} \cdot {}^n\pi_i$

I now compose an alltup whose elements are either $\lhd|\rhd_p \cdot {}^n\pi_i$ or just ${}^n\pi_i$. If i is in I it is the former, otherwise it is the latter.

This is the same process as with the vector input, except that I am doing it for each element of the input tuple.

After propagating a construct via one of the two cases above, it can be incorporated into the while/if. See section 4.5.

4.4 Address

As explained in 4.1.3, the process of compaction can produce code that needs addresses to be evaluated. If code is produced such as the following:

$$
{}^n\pi_i \cdot {}^m\pi_j \cdot ((x, y)^\circ,\ z)^\circ
$$

I cannot simply evaluate the right-hand address ${}^m\pi_i$, because I am trying to match the whole expression. The example could be extended to include 10 address constructs, and an alltup with corresponding nesting. Therefore every time I come across a address in the program, I recurse to the right-most address to see whether it needs to be evaluated.

To do this I create a list to contain the addresses matched. I then check for either of the following:

1. the list has only one element; the previous construct was a $\lhd|\rhd_p$; and the next construct is a while or an if,

2. the next construct is an alltup.

For the first case, I create an alltup to be incorporated into the while/if, as in section 4.3.3:

$$\lhd|\rhd_p \cdot {}^n\pi_i \Rightarrow {}^n\pi_i \cdot ({}^n\pi_1, \ldots, \lhd|\rhd_p \cdot {}^n\pi_i, \ldots, {}^n\pi_n)^\circ$$

The $\lhd|\rhd_p$ can then be propagated through the while/if. See section 4.5.

For the second case, I evaluate the last address that was added. If the list is still not empty, I repeat the process with the next element.

If none of the cases are matched, and the list has become empty, I compose the previous construct to the next construct and continue with the parallelisation. If it is not empty, I replace the addresses and start the parallelisation afresh at the next construct.

4.5 While/If

Every while and if in the code has predictable input and output types (see section 4.1.3). I therefore only need to consider three cases when matching while/if:

1. $\lhd|\rhd_p \cdot$ while/if
2. $(f_1, \ldots, f_n)^\circ \cdot$ while/if
3. while/if

The alltup in the second case is the result of the process in either section 4.3 or section 4.4. Therefore both the first two cases signify that parallelisation has continued up to here. The construct on the left is incorporated into the while/if to propagate the parallelism. The third case signifies that the parallelisation has to (re)start here.

4.5.1 If

The aim is to parallelise the if elements, move splits to the right, and, if necessary, any $+\!\!+/^{\|}$ to the left. The third stage may seem unnecessary, but if the if is inside a while, for example, it is required. The process for the third case is demonstrated in figure 4.6: a vector is copied into a tuple, of which, one element is processed depending on the length of the vector.

For the first two cases above, the construct on the left is appended to both the consequent and alternative. In all three cases, I now follow the process described in section 4.3.3. The only difference is that the set I is produced by considering all three components. If an alltup or split is produced, it is moved outside of the if construct.

The final stage of the process is moving any $+\!\!+/^{\|}$ constructs back out of the if. If either the consequent or the alternative ends with $+\!\!+/^{\|}$, I move it out of the if by performing the following replacements to the two elements:

$$
\begin{array}{l}
\quad \underline{(\lhd|\rhd_p \cdot {}^2\pi_1 \,,\, {}^2\pi_2)^\circ} \cdot \\
\quad \underline{\mathsf{if}\,(> \cdot(\#, 5)^\circ)\,\mathsf{then}\,((\mathsf{id}\,,\ f*)^\circ)\,\mathsf{else}\,((g*,\ \mathsf{id})^\circ)} \\
\Rightarrow \mathsf{if}\,(> \cdot(\#, 5)^\circ)\,\mathsf{then}\,((\lhd|\rhd_p \cdot {}^2\pi_1 \,,\, {}^2\pi_2)^\circ \cdot (\mathsf{id}\,, f*)^\circ) \\
\quad \mathsf{else}\,((\lhd|\rhd_p \cdot {}^2\pi_1 \,,\, {}^2\pi_2)^\circ \cdot (g*,\ \mathsf{id})^\circ) \\
\Rightarrow \mathsf{if}\,(> \cdot(+/^{\|} \cdot (\#) *^{\|} \cdot \underline{\lhd|\rhd_p}\,, \underline{5})^\circ)\,\mathsf{then}\,((\underline{\lhd|\rhd_p}\,, +\!\!+/^{\|} \cdot (f*) *^{\|} \cdot \underline{\lhd|\rhd_p})^\circ) \\
\quad \mathsf{else}\,(((g*) *^{\|} \cdot \underline{\lhd|\rhd_p}\,, \underline{\mathsf{id}})^\circ) \\
\Rightarrow \mathsf{if}\,(> \cdot(+/^{\|} \cdot (\#)*^{\|}, 5)^\circ)\,\mathsf{then}\,((\mathsf{id}\,, \underline{+\!\!+/^{\|}} \cdot (f*)*^{\|})^\circ)\,\mathsf{else}\,(((g*)*^{\|}\,, \underline{+\!\!+/^{\|}})^\circ) \cdot \\
\quad \lhd|\rhd_p \\
\Rightarrow ({}^2\pi_1 \,, +\!\!+/^{\|} \cdot {}^2\pi_2)^\circ \cdot \\
\quad \mathsf{if}\,(> \cdot(+/^{\|} \cdot (\#)*^{\|}, 5)^\circ)\,\mathsf{then}\,((\mathsf{id},\ (f*)*^{\|})^\circ)\,\mathsf{else}\,(((g*)*^{\|}\,, \mathsf{id})^\circ) \cdot \\
\quad \lhd|\rhd_p
\end{array}
$$

Figure 4.6: Parallelisation of an if construct

Ends with	Replacement
$+\!\!+/^{\|} \cdot x$	x
$+\!\!+/^{\|}$	id
x	$\lhd\|\rhd_p \cdot x$

If the consequent and alternative end with an alltup, I check whether any of the elements end with a $+\!\!+/^{\|}$. For each element that does, I proceed the same as for the vector output above.

4.5.2 While

The process for a while is mainly the same as that applied to an if. A difference arises because distribution of the input and output types must be the same. If splits are propagated out of the while, then corresponding $+\!\!+/^{\|}$ constructs must be taken back out to the left. The process for the first case is illustrated in figure 4.7, where a while performs a map on its second argument while its first argument is greater than 0.

In the first two cases, the construct on the left is composed to the while loop. This process must be reversed later to maintain the identity of the code, because:

$$\lhd|\rhd_p \cdot \mathsf{while}\,(x)\,\mathsf{do}\,(y) \not\equiv \mathsf{while}\,(x)\,\mathsf{do}\,(\lhd|\rhd_p \cdot y)$$

In all three cases, if no split has propagated to the start of the while loop after parallelisation, it has not been parallelised sufficiently. The while is therefore reverted back to its original form.

$$
\begin{aligned}
&\mathsf{while}\,((-\cdot\ ({}^2\pi_1, 1)^\circ, f* \cdot {}^2\pi_2)^\circ)\,\mathsf{do}\,(> \cdot ({}^2\pi_1, 0)^\circ)\\
\Rightarrow\ &\mathsf{while}\,((-\cdot\ (\underline{{}^2\pi_1}, 1)^\circ, +\!\!+/^{\|} \cdot (f*)*^{\|} \cdot \underline{\lhd|\rhd_p \cdot {}^2\pi_2})^\circ)\,\mathsf{do}\,(> \cdot (\underline{{}^2\pi_1}, 0)^\circ)\\
\Rightarrow\ &\mathsf{while}\,((-\cdot\ ({}^2\pi_1, 1)^\circ, \underline{+\!\!+/^{\|}} \cdot (f*)*^{\|} \cdot {}^2\pi_2)^\circ)\,\mathsf{do}\,(> \cdot ({}^2\pi_1, 0)^\circ)\cdot\\
&({}^2\pi_1\,, \underline{\lhd|\rhd_p} \cdot {}^2\pi_2)^\circ\\
\Rightarrow\ &({}^2\pi_1\,, +\!\!+/^{\|} \cdot {}^2\pi_2)^\circ\cdot\\
&\mathsf{while}\,((-\cdot\ ({}^2\pi_1, 1)^\circ, (f*)*^{\|} \cdot {}^2\pi_2)^\circ)\,\mathsf{do}\,(> \cdot ({}^2\pi_1, 0)^\circ)\cdot\\
&({}^2\pi_1\,, \lhd|\rhd_p \cdot {}^2\pi_2)^\circ
\end{aligned}
$$

Figure 4.7: Parallelisation of a while construct

If at least one has, the split or alltup is moved outside of the construct as for if.

In the first case, the act of moving the split outside the loop is the inverse of composing the split to the loop, therefore the identity is maintained:

$$
\begin{aligned}
&\lhd|\rhd_p \cdot \mathsf{while}\,(x)\,\mathsf{do}\,(y)\\
\Rightarrow\ &\mathsf{while}\,(x)\,\mathsf{do}\,(\lhd|\rhd_p \cdot y)\\
\Rightarrow\ &\mathsf{while}\,(x' \cdot \lhd|\rhd_p)\,\mathsf{do}\,(y' \cdot \lhd|\rhd_p)\\
\Rightarrow\ &\mathsf{while}\,(x')\,\mathsf{do}\,(y') \cdot \lhd|\rhd_p
\end{aligned}
$$

In the second case, there is no assurance that the alltup that was brought into the loop was identical to the one moved out after parallelisation. Therefore I compare the two. I denote them F and G respectively. I perform replacements on the start of the loop elements if the corresponding elements of F and G differ:

F	G	Starts with	Replacement
$\lhd\|\rhd_p \cdot {}^n\pi_i$	${}^n\pi_i$	$\lhd\|\rhd_p \cdot x$	x
$\lhd\|\rhd_p \cdot {}^n\pi_i$	${}^n\pi_i$	x	$+\!\!+/^{\|} \cdot x$
${}^n\pi_i$	$\lhd\|\rhd_p \cdot {}^n\pi_i$	$+\!\!+/^{\|} \cdot x$	x
${}^n\pi_i$	$\lhd\|\rhd_p \cdot {}^n\pi_i$	x	$\lhd\|\rhd_p \cdot x$

The third replacement occurs when the element was completely parallelised in the while. The other three were only partially parallelised – not the ideal scenario (see section 6.4.2 for further discussion).

To complete the process an alltup is appended to the left of the while with elements corresponding to the third column in the table: replacing x with ${}^n\pi_i$, the new alltup consists of the removed constructs.

In the third case, if the input is a vector, the $+\!\!+/^{\|}$ and split are taken from each end and appended outside of the while.

If the input is a tuple, the process is mainly the same as for the second case. Let G be previously denoted. $\forall\ g \in G$, if $g = \lhd|\rhd_p \cdot {}^n\pi_i$, the following replacement is made to the i^{th} element of the loop:

Starts with	Replacement
$+\!\!+/^{\|} \cdot x$	x
x	$\lhd\|\rhd_p \cdot x$

Again, to complete the process, an alltup is appended to the left of the while. It is identical to G, except any split is replaced with a $+\!\!+/^{\|}$.

Chapter 5

Results

This section presents examples comparing the execution times of sequential BMF programs and the corresponding code resulting from parallelisation. The simulator created by Martinaitis [13] was used to produce timings and visualisations.

The simulator can be configured to model communications on a variety of distributed architectures. In this instance I used a crossbar topology[1] comprising 32 nodes.

Bandwidth and latency were configured to have values of $3ns$ per byte and $5000ns$ respectively. These values are realistic, currently, for an average speed multiprocessor machine, or a high speed cluster.

5.1 Example 1: Maximum Segment Sum

The first example tests a program which computes the maximum segment sum (MSS) of a vector. The MSS is the maximum of the sums of any of its contiguous segments.

The sequential source code has the form:

$$({}^4\pi_1) * \cdot \oplus /\!\!/ \cdot f*$$

The code comprises three constructs with vector input and output, all of which can be parallelised individually. Therefore the parallel code comprises the parallelised version of map twice and of scan. Consequently, the resulting code is the same as the code produced through manual parallelisation.

I simulated the execution of both the sequential and parallel programs for various data sizes: 256, 576, 1664 and 3264. Figure 5.1 compares the time taken to execute the two versions. The sequential time is that for one processor.

[1] A crossbar network has full connectivity: all processors can communicate with each other simultaneously.

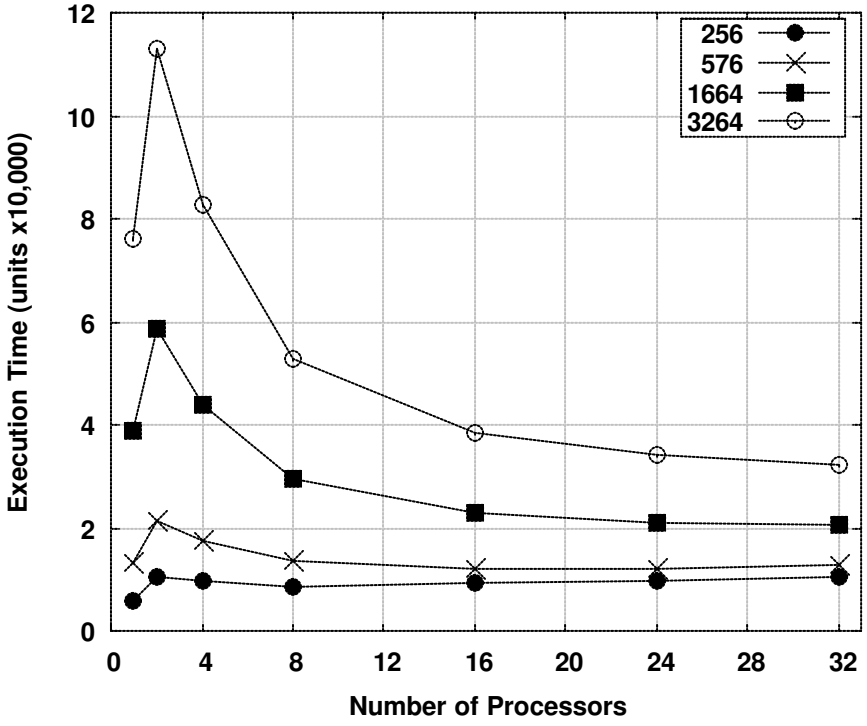

Figure 5.1: Execution times of MSS for various data sizes

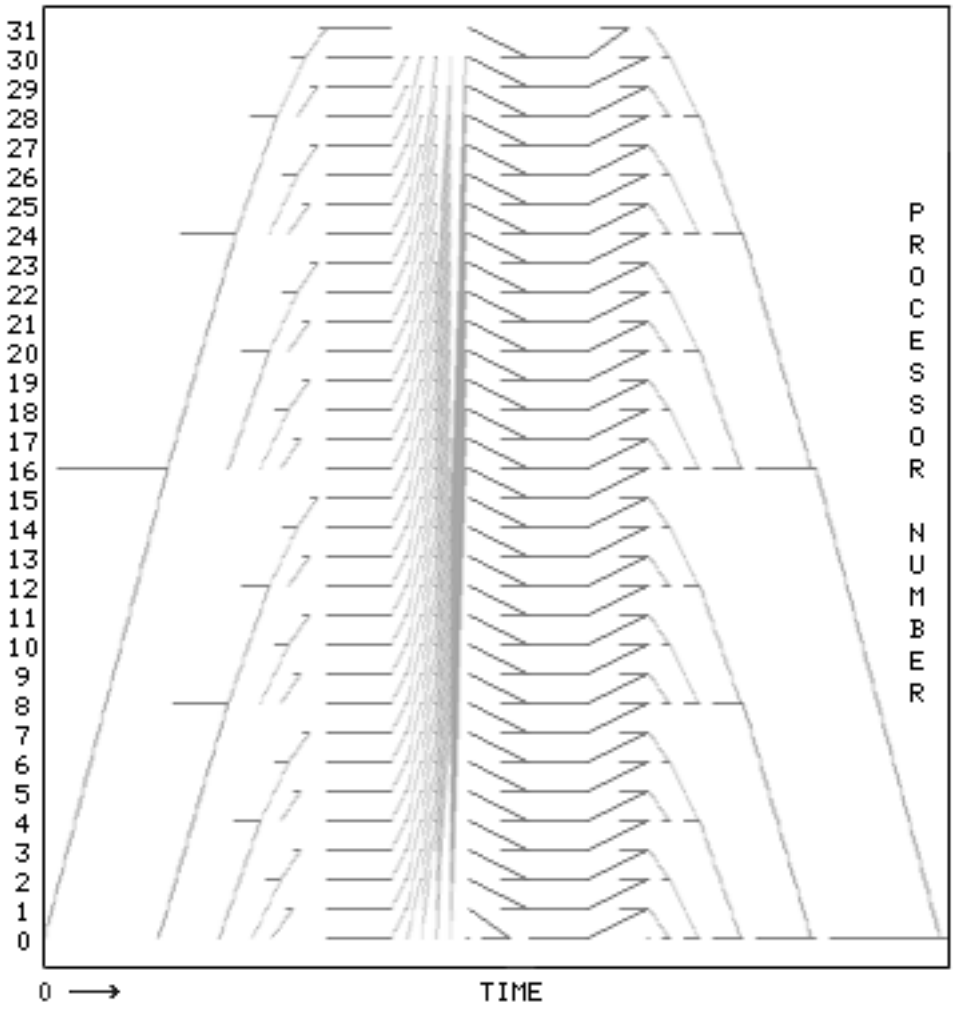

Figure 5.2: Execution trace of MSS on 32 nodes

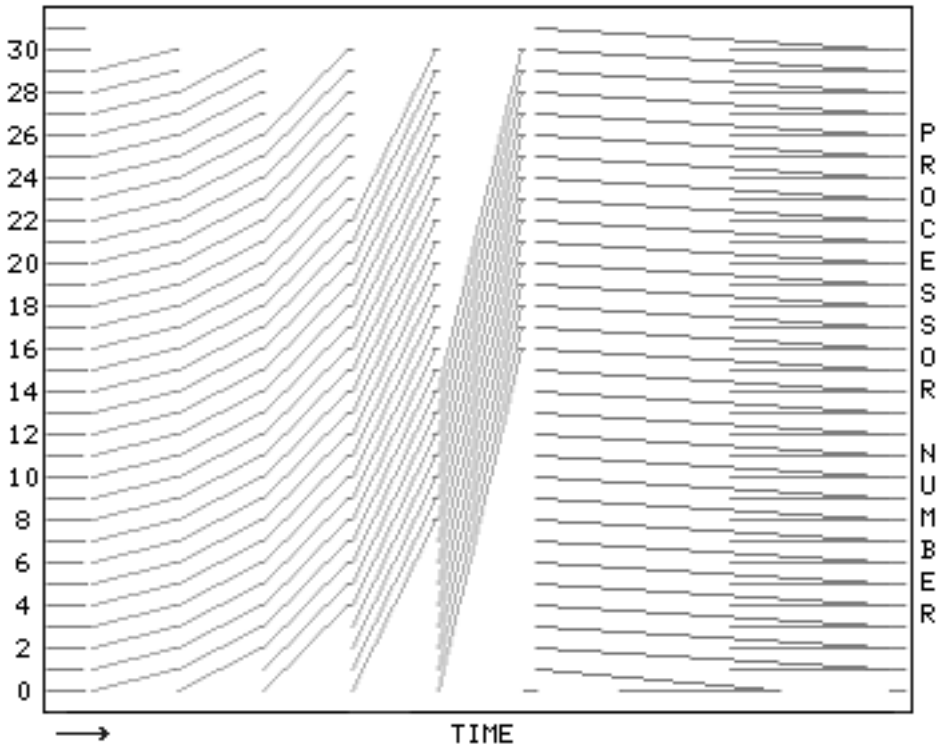

Figure 5.3: View of the MSS communications for the parallelised scan

The resultant code comprises parallel select constructs, used in the parallelisation of scan. Hence there is non-trivial communication involved between nodes. This is displayed in figure 5.2, a space-time diagram of the execution of the parallel code on 32 nodes, with a data size of 3264. Figure 5.3 is a magnification of the scan section of the program, displaying details of the communication.

Even with this communication, the graph shows that the use of parallelisation can be worthwhile.

5.2 Example 2: Remote Distance

The second example tests a program which calculates the remote distance for each element of a vector. That is, for each integer element, it returns the aggregate of the absolute differences between itself and the other elements.

The sequential program has the form:

$$f * \cdot \Upsilon \cdot (\mathsf{id}, \mathsf{repeat} \cdot (\mathsf{id}, \#)^\circ)^\circ$$

There are no constructs before the length operator, #, so the resulting code

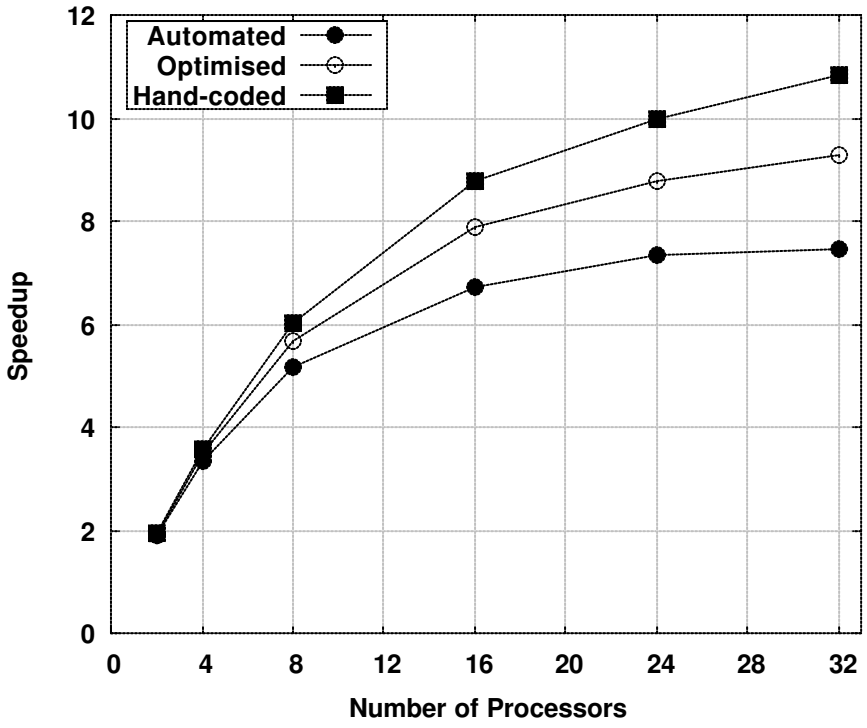

Figure 5.4: Speedup of Remote Distance parallelisations

includes its parallel version:

$$+/^{\|} \cdot (\#) *^{\|} \cdot \triangleleft|\triangleright_p$$

I presumed that this piece of code would not be efficient due to the small amount of work done between distribution and reduction. I therefore simulated the execution of the following:

1. the sequential program,
2. the automatically parallelised code,
3. the automatically parallelised code with the length construct deparallelised, which should result in an optimised version.
4. a hand-coded parallel version.

Figure 5.4 displays the speedup gained for the parallelised versions, with an input vector of size 320.

The graph shows that this implementation will not always produce the best parallelised code. Even with the slight optimisation, the automated code is slower than the hand-written program. It also shows that the parallelisation of a single construct can be inefficient. These points are discussed further in sections 6.5 and 6.4.1 respectively.

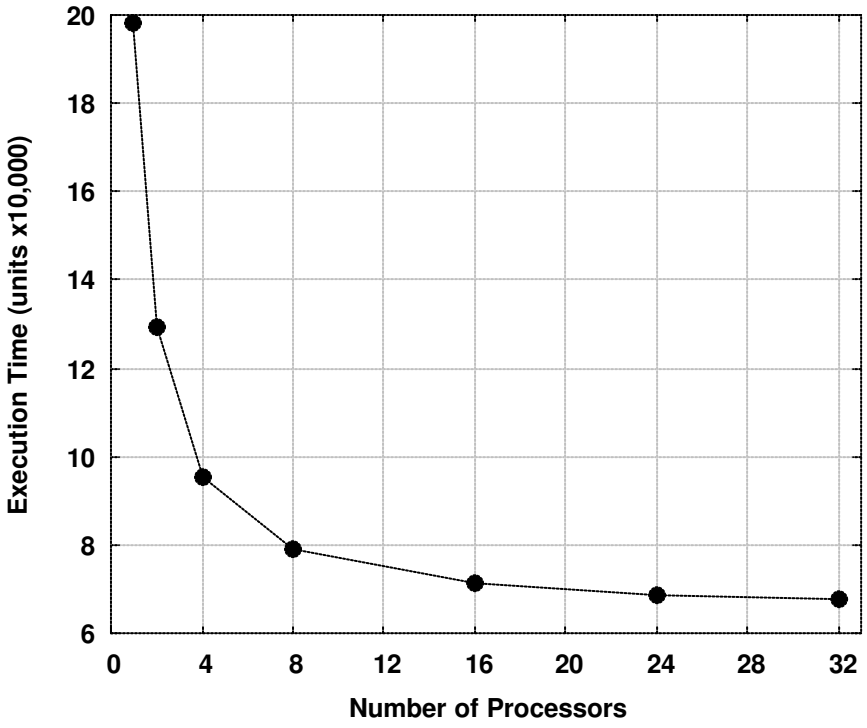

Figure 5.5: Execution times of the while loop example

5.3 Example 3: While Loop

For the third example, I created an arbitrary sequential program comprising a simple while loop which repeated 100 times:

$$\mathsf{while}\,(< \cdot ({}^{2}\pi_2,\, 100)^{\circ})\,\mathsf{do}\,(((+ \cdot (\mathsf{id},\, 2)^{\circ}) * \cdot {}^{2}\pi_1, +(\mathsf{id}, 1)^{\circ} \cdot {}^{2}\pi_2)^{\circ}) \cdot (\mathsf{id},\, 1)^{\circ}$$

I again simulated its execution time, along with the parallel version. Figure 5.5 shows the results. The input vector had length 1664.

This shows the speedup that can be gained from parallelising the while construct.

Chapter 6

Conclusion / Issues

The implementation of techniques for individual constructs from [1] was reasonably straightforward. Complexity was introduced by the constructs that were not directly parallelised: alltup, allvec, address, while and if. The techniques devised for these constructs, while fairly involved, return simple and predictable code.

The final product is only a prototype, but is still fairly effective. In many cases, the automatic parallelisation could produce code that was the same as a hand-coded program. Through the changes prescribed in sections 6.1 to 6.4, future versions should be able to parallelise more programs to the same standards.

As well as these changes, there are various areas for possible future work: section 6.5 discusses the need for knowing the context of a construct; section 6.6 considers the possibility of nested parallelisation; and section 6.7 discusses the use of type information.

6.1 Compaction

alltup constructs that are composed together are automatically compacted (4.3.1). In retrospect, this has undesirable side effects. Consider the simple example:

$$(\mathsf{id}\,, \mathsf{id})^\circ \cdot (f*, \mathsf{id})^\circ$$

where the second alltup simply creates a copy of its input tuple. Unfortunately in this implementation, the mapping is processed twice as a result of compaction:

$$((f*, \mathsf{id})^\circ, (f*, \mathsf{id})^\circ)^\circ \tag{6.1}$$

This is even more prevalent when an alltup is composed to an arbitrary construct inside a while or an if (see section 4.1.3).

A better implementation would leave the alltups separate so no repetition occurs. However, the current implementation may still be acceptable. The

speedup gained by distributing vector data onto multiple processors may be greater than the overhead produced by alltup compaction. The experiments in section 5 show that speedup can still be achieved.

A further sweep of the code to eliminate common sub-expressions could also be implemented. After parallelisation (6.1) would look like this:

$$((+\!\!+/^{\|} \cdot (f*)*^{\|}, +\!\!+/^{\|})^{\circ}, (+\!\!+/^{\|} \cdot (f*)*^{\|}, +\!\!+/^{\|})^{\circ})^{\circ} \cdot \lhd|\rhd_p$$

The inverse compaction sweep could identify the redundancy in the outer alltup and return:

$$(\mathsf{id}\,, \mathsf{id})^{\circ} \cdot (+\!\!+/^{\|} \cdot (f*)*^{\|}, +\!\!+/^{\|})^{\circ} \cdot \lhd|\rhd_p$$

and the problem would be avoided.

6.2 Further Optimisation

The code resulting from the parallelisation process is not optimised. In essence, the process parallelises each construct individually. A further transformation that optimises the parallel code would be analogous to the optimisation techniques in [1].

For example, one simple optimisation is the combination of parallel maps that may end up composed together. This will occur in the following example where a reduce is compose to a map:

$$\oplus/ \cdot f* \Rightarrow \oplus/^{\|} \cdot (\oplus/)*^{\|} \cdot (f*)*^{\|} \cdot \lhd|\rhd_p$$

An optimised solution would be:

$$\oplus/^{\|} \cdot (\oplus/ \cdot f*)*^{\|} \cdot \lhd|\rhd_p$$

6.3 Propagating Multiple Split Constructs

I propagate parallelism through alltup/allvec constructs dependant on only one element ending in split. This may mean that $n-1$ copies and reductions are applied for little or no gain. Take the example below:

$$(f*, \mathsf{id})^{\circ}$$

The input is copied for the two elements, but only one of the copies is processed. This implementation produces the code:

$$(+\!\!+/^{\|} \cdot (f*)*^{\|}, +\!\!+/^{\|})^{\circ} \cdot \lhd|\rhd_p$$

The problem here is that the process is not technically eliminating common split constructs, because the split is not common! If this was the complete

program, the copy and extra reduction is completely overhead. In this case a better result would be to simply to leave the split inside the alltup.

However, this is a case by case problem. If the alltup was composed to an initial map, the overhead may be offset by the parallel computation:

$$(+\!\!+/^{\|} \cdot (f*)*^{\|}, +\!\!+/^{\|})^{\circ} \cdot (g*) *^{\|} \cdot \lhd|\rhd_p$$

During a further optimisation stage, it would be beneficial to remove such overheads. Any instance where a $+\!\!+/^{\|}$ and a $\lhd|\rhd_p$ are separated by the end of an alltup should be removed.

The more complex example where there are more than two elements:

$$(+\!\!+/^{\|} \cdot x, +\!\!+/^{\|} \cdot y, +\!\!+/^{\|})^{\circ} \cdot \lhd|\rhd_p$$

could be solved easily using a second alltup:

$$(+\!\!+/^{\|} \cdot x \cdot {}^2\pi_1, +\!\!+/^{\|} \cdot y \cdot {}^2\pi_1, {}^2\pi_2)^{\circ} \cdot (\lhd|\rhd_p, \mathsf{id})^{\circ}$$

6.4 Reduction and Redistribution

If data is unavoidably reduced to one node for some reason, the parallelisation can restart at the next construct. Consequently, data may be distributed again.

6.4.1 Unwarranted Parallelism

The experiment discussed in section 5.2 deals with a program which starts with the alltup:

$$(\mathsf{id}, \mathsf{repeat} \cdot (\mathsf{id}, \#)^{\circ})^{\circ}$$

The parallelisation of this code results in the expression:

$$(\mathsf{id},\ x \cdot (y \cdot +\!\!+/^{\|}, \lhd|\rhd_p \cdot \mathsf{iota} \cdot +/^{\|} \cdot (\#)*^{\|})^{\circ})^{\circ} \cdot \lhd|\rhd_p$$

First of all, the code introduces unnecessary overheads as described in section 6.3. Assuming that this overhead was removed, the code will contain the expression:

$$+/^{\|} \cdot (\#) *^{\|} \cdot \lhd|\rhd_p$$

The parallelisation of the single construct, length, is not beneficial. The results from the aforementioned experiment show that leaving the construct to be processed on one node is faster.

In other circumstances it might be perfectly reasonable to parallelise length, when the corresponding vector is already distributed. So we cannot ignore the parallelisation of length altogether.

Besides, the fact that there is a parallel map of length is not the complete issue here. The problem is that there is not enough computation between the reduce and the split to warrant parallelism.

One solution could involve the user placing flags around expressions that they did not feel warranted parallelism. For example: $X_1 \cdot \# \cdot X_2$. The implementation could ignore the expression inside the constructs X_1 and X_2. This may, however, need the user to have knowledge of some of the parallelisation process.

Another solution could take place during optimisation after parallelisation. The process could observe the type of computation between each reduce and split and decide whether it justifies parallelism. If it doesn't, the expression could be re-sequentialised[1]. This would be especially appropriate if the expression was a single map, such as above, because the parallelisation was simple and consequently its inverse would be as well.

6.4.2 Inside While Constructs

If a while construct looked something like:

$$\mathsf{while}\,(x)\,\mathsf{do}\,(\mathsf{iota} \cdot \oplus/)$$

it would be parallelised as:

$$+\!\!+/^{\parallel} \cdot \mathsf{while}\,(x')\,\mathsf{do}\,(\lhd|\rhd_p \cdot\ \mathsf{iota} \cdot \oplus/^{\parallel} \cdot (\oplus/)*^{\parallel}) \cdot \lhd|\rhd_p$$

because the loop expression started with a $\lhd|\rhd_p$. Every time around the loop, the data is reduced and then redistributed. This example is arbitrary and obviously would not be implemented, but it shows that parallelising only part of the loop will lead to very inefficient code. The first solution presented in section 6.4.1 would solve this problem.

6.5 Locality

The techniques presented in [1] were applied to constructs in isolation. Knowledge of the locality of a construct may help to optimise the resultant code. For example, the parallelised form of repeat includes the expression:

$$\lhd|\rhd_p \cdot \mathsf{iota} \cdot {}^2\pi_2 \tag{6.2}$$

whose soul purpose is to create a distributed vector of length equal to the number of copies to be made. The elements of the vector are arbitrary, as they are not accessed. Example 3 in section 5.2, which is also discussed in section 6.4.1, comprises the sequential code:

$$\mathsf{repeat} \cdot (\mathsf{id}, \#)^\circ \tag{6.3}$$

[1] Returned to its original form.

The expression corresponding to (6.2) in the resulting parallel program is:

$$\lhd|\rhd_p \cdot \mathsf{iota} \cdot +/^{\|} \cdot (\#) *^{\|} \cdot \lhd|\rhd_p$$

The code finds the length n of the input vector, so it can create a vector of size n. Obviously the process is redundant and the expression can be replaced with:

$$\lhd|\rhd_p$$

However, to implement this replacement in the paralleliser, there must be an attempt to match the code in (6.3). But how often does this case occur? There are probably many more special cases like this, some that are rare, and some that occur frequently. It would be impossible to add them all to the paralleliser. I suggest that only those that occur with the highest frequency are added, and the others would have to be ignored.

6.6 Non-Recursive Parallelisation

Programs that comprise nested vector operations can exploit nested parallelism. The program from in section 5.2 contains such an operation:

$$(+/ \cdot (g*) \cdot \mathsf{distl})*$$

The outer map is simply parallelised. The mapped expression can also be parallelised, because it too, is applied to vector input. Alexander [1] shows that there are benefits to exploiting both levels of parallelism.

For simplicity, this version of the paralleliser implements non-recursive parallelism. Future implementations can incorporate multiple levels of parallelism to attempt to increase performance.

6.7 Type Information

The paralleliser currently uses no type information, other than that which can be derived from observing the constructs that are applied. To parallelise the majority of constructs, this is all that is needed.

However, I thought that it *could* be beneficial to incorporate type information into the program, to help with the constructs that aren't directly parallelised. The reason for compacting alltup/allvec constructs and flattening the input and output of while and if, was to make them predictable: to make it simple to propagate parallelism. I felt that techniques to propagate distributed vectors at arbitrary levels of a tuple would be a lot higher in complexity than those that I eventually implemented.

Having type information may be a useful feature when designing these techniques. Instead of having to work out the structure of the data, it would already be known.

However, my implementation *does* show that it is not necessary. In this case, I think that any benefits are outweighed by the overhead created by adding it.

There is scope to use type information if a stronger need arises in later versions.

Bibliography

[1] Alexander, B., *Mapping a Functional Language to a Data-Parallel Model of Computation*, Chapters 5-6. To be published.

[2] Alexander, B., Engelhardt, D., Wendelborn, A., *An Overview of the Adl Language Project.* In *Proceedings Conference on High Performance Functional Computing*, pp. 73-82, April 1995.

[3] Backhouse, R., *An Exploration of the Bird-Meertens Formalism.* Technical Report CS 8810, Department of Computer Science, Groningen University, 1988.

[4] Gibbons, J., *An Introduction to the Bird-Meertens Formalism.* In *Proceedings of the New Zealand Formal Program Development Colloquium*, Nov.1994.

[5] Gorlatch, S., *Abstraction and Performance in the Design of Parallel Programs.* In *Acta Informatica*, Vol 36, No 9/10, pp. 761-803, 2000.

[6] Gorlatch, S., *Towards Formally-Based Design of Message Passing Programs.* In *Software Engineering*, Vol 26, No 3, pp. 276-288, 2000.

[7] Hu, Z., Sun, Y., *Functional Approach to the Synthesis of Systolic Arrays* In *Proceedings of 2nd International Conference for Young Computer Scientists*, 1991.

[8] Hu, Z., Iwasaki, H., Takeichi, M., *Construction of List Homomorphisms by Tupling and Fusion.* In *21st International Symposium on Mathematical Foundations of Computer Science*, LNCS 1113, pp. 407-418, 1996.

[9] Hu, Z., Iwasaki, H., Takeichi, M., *Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms.* In *ACM Transactions on Programming Langauges and Systems*, 19(3), pp. 444-461, 1997.

[10] Hu, Z., Takeichi, M., Chin, W.N., *Parallelization in Calculational Forms.* In *Symposium on Principles of Programming Languages*, pp. 316-328, 1998.

[11] Jones, S.P., *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[12] Jones, M.P., Peterson, J.C., *Hugs 98: A functional programming system based on Haskell 98*. Oregon Graduate Institute of Science and Technology, 1999.

[13] Martinaitas, P., *The Simulation and Visualisation of Parallel BMF Code*. Department of Computer Science, University of Adelaide, 1998.

[14] McGraw, J., *Sisal: Streams and Iteration in a Single Assignment Language. Language Reference Manual*, Lawrence Livermore National Laboratory.

[15] Roe, P., *Derivation of Efficient Data Parallel Programs*. In *Proceedings of the 17th Australasian Computer Science Conference*, pp. 621–628, 1994.

[16] Skillicorn, D.B., *The Bird-Meertens Formalism as a Parallel Model*. In *Software for Parallel Computation*, Vol 106, 1993

[17] Skillicorn, D.B., *Questions and Answers about Categorical Data Types*. In *Proceedings of Meeting on Bulk Data Types for Architecture Independence*, May 1994.

[18] Skillicorn, D.B., *Foundations Of Parallel Programming*. Cambridge University Press, 1994.

[19] Skillicorn, D.B., Cai, W., *A Cost Calculus for Parallel Functional Programming*. In *Journal of Parallel and Distributed Computing*, Vol 28 No 1, pp. 65-83, 1995.

[20] Sun, Y., Hu, Z., *Algebraic Approach to the Synthesis of Systolic Arrays*. In *Proceedings of 2nd German-Chinese Electronics Congress*, Germany, 1991.

[21] Walinsky, C., Banerjee, D., *A Data-Parallel FP compiler*. In *Journal of Parallel and Distributed Computing*, Vol 22, No 2, pp. 138-153, 1994.

[22] Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, *A Calculational Fusion System HYLO*. In *Algorithmic Languages and Calculi*, pp. 76-106, 1997.

Appendix A

Paralleliser Code

```
module BMF where

data B_exp = B_comp B_exp B_exp | B_TEMP | B_NULL |
             F | G | H | X | Y | Z |        -- arbitrary fns
             B_op Op | B_id |
             B_map B_exp | B_reduce B_exp B_exp | B_scan B_exp |
             B_alltup [B_exp] | B_allvec [B_exp] |
             B_con Con | B_num Int | B_addr B_exp B_exp |
             B_if B_exp B_exp B_exp | B_while B_exp B_exp |

             P_map B_exp | P_reduce B_exp | P_scan B_exp |
             P_split B_exp B_exp | P_splitpair B_exp B_exp |
             P_distpriff B_exp B_exp      deriving Show;

data Op =  B_index | B_transpose Int Int | B_repeat |
           B_plus | B_minus | B_times | B_divide |
           B_and | B_or | B_eq | B_neq | B_gt | B_lt | B_ge |
           B_iota | B_myinitseg | B_conc |
           B_length | B_uminus | B_neg | B_distl | B_select |
           B_zip B_exp | B_mask | B_priffle |

           P_transpose Int Int | P_zip B_exp | P_distl |
           P_conc | P_repeat | P_select
           deriving Show;


data Con = B_int Int | B_real Float | B_true | B_false
           deriving Show;

-- data type for finding structure (int is to count elements)
data Structure = Addr | Alltup Int [Structure] deriving Show;
```

```
-- INITIALISE --
----------------

pl :: Int -> B_exp -> B_exp
pl s x = pll (B_num s,B_num 0) x

pll :: (B_exp,B_exp) -> B_exp -> B_exp
pll d x = sweep (append d (norm x))

-- append: start parallisation
append :: (B_exp,B_exp) -> B_exp -> B_exp
append d B_TEMP = B_TEMP

    -- while and if
append d (B_comp (B_while lp chk) x)
    = insideWhIf d B_NULL [pll d lp] [chk,lp] x
append d (B_comp (B_if cnd thn els) x)
    = insideWhIf d B_NULL (map (pll d) [thn,els]) [cnd,thn,els] x

    --  these operators are not parallelised at all
append d (B_comp(B_op B_plus) x) = B_comp(B_op B_plus)(append d x)
append d (B_comp(B_op B_minus) x) = B_comp(B_op B_minus)(append d x)
append d (B_comp(B_op B_times) x) = B_comp(B_op B_times)(append d x)
append d (B_comp(B_op B_divide) x)=B_comp(B_op B_divide)(append d x)
append d (B_comp(B_op B_and) x) = B_comp(B_op B_and) (append d x)
append d (B_comp(B_op B_or) x) = B_comp(B_op B_or)(append d x)
append d (B_comp(B_op B_eq) x) = B_comp(B_op B_eq)(append d x)
append d (B_comp(B_op B_neq) x) = B_comp(B_op B_neq) (append d x)
append d (B_comp(B_op B_gt) x) = B_comp(B_op B_gt)(append d x)
append d (B_comp(B_op B_lt) x) = B_comp(B_op B_lt)(append d x)
append d (B_comp(B_op B_uminus) x)=B_comp(B_op B_uminus)(append d x)
append d (B_comp(B_op B_neg) x) = B_comp(B_op B_neg)(append d x)
append d (B_comp(B_op B_myinitseg) x)
    = B_comp (B_op B_myinitseg) (append d x)

    -- not parallelised directly
append d (B_comp (B_addr a b) x) = p d (B_comp (B_addr a b) x)
append d (B_comp (B_alltup fs) x) = p d (B_comp (B_alltup fs) x)
append d (B_comp (B_allvec fs) x) = p d (B_comp (B_allvec fs) x)
append d (B_comp (P_split s t) x) = p d (B_comp (P_split s t) x)

    -- transpose has its own setup (with if)
append d (B_comp (B_op (B_transpose a b)) x)
= transPll d True (a,b) x

    -- the rest can have the identity appended!
append d x = B_comp redConc (p d (B_comp (split d) x))
```

```
-- normalise

norm x = whIfSwp (n x)

n :: B_exp -> B_exp
n (B_comp (B_comp x y) z) = n (B_comp x (B_comp y z))
n (B_comp x y) = B_comp x (n y)
n B_TEMP = B_TEMP
n x = B_comp x B_TEMP


-- BODY --
----------

p :: (B_exp,B_exp) -> B_exp -> B_exp

-- con

p d (B_comp x (B_comp (B_con a) y)) = B_comp x (B_con a)

-- addr

p d (B_comp (B_addr l i) x) = addrSwp d B_NULL (B_addr l i) x
p d (B_comp x (B_comp (B_addr l i) y)) = addrSwp d x (B_addr l i) y

-- id

p d (B_comp x (B_comp B_id y)) = p d (B_comp x y)
p d (B_comp B_id (B_comp x y)) = p d (B_comp x y)


-- map

p d (B_comp (P_split s t) (B_comp (B_map a) x))
    = B_comp (P_map (B_map a)) (p d (B_comp (P_split s t) x))

-- reduce

p d (B_comp x (B_comp (B_reduce a b) y))
  = B_comp
        x
    (B_comp
        (P_reduce a)
    (B_comp
        (P_map(B_reduce a b))
        (p d (B_comp (split d) y))))
```

```
-- scan

p d (B_comp (P_split s t) (B_comp (B_scan a) x))
    =   B_comp
            (g a)
        (B_comp
            (B_alltup[
                B_comp (P_scan a) (B_comp myinit (P_map mylast)),
                B_id])
        (B_comp
            (P_map(B_scan a))
            (p d (B_comp (P_split s t) x))))

-- length

p d (B_comp x (B_comp (B_op B_length) y))
    =   B_comp
            x
        (B_comp
            (P_reduce (B_op B_plus))
        (B_comp
            (P_map (B_op B_length))
            (p d (B_comp (split d) y))))

-- transpose

p d (B_comp (P_split s t) (B_comp (B_op (B_transpose a b)) x))
    = transPll d False (a,b) x

-- zip

p d (B_comp (P_split s t) (B_comp (B_op (B_zip a)) x))
    =     B_comp
            (P_map (B_op (B_zip a)))
        (B_comp
            (B_op (P_zip a))
            (p d (B_comp (addAT (P_split s t) 2) x)))

-- select

p d (B_comp (P_split s t) (B_comp (B_op B_select) x))
    = B_comp (P_map (B_op B_select)) (sOrD d x)

-- distl

p d (B_comp (P_split s t) (B_comp (B_op B_distl) x))
    = B_comp (P_map (B_op B_distl)) (sOrD d x)
```

```
-- repeat

p d (B_comp (P_split (B_num s) t) (B_comp (B_op B_repeat) x))
    =   B_comp
            (P_map (B_comp
                        (B_op B_repeat)
                        (B_alltup[
                            ad 2 1,
                            B_comp (B_op B_length) (ad 2 2)])))
        (B_comp
            (B_op (P_zip zipArg))
            (p d (B_comp
                    (B_alltup[
                        B_comp
                            (B_op P_repeat)
                            (B_comp
                                (B_alltup[B_id,B_con (B_int s)])
                                (ad 2 1))
                        ,
                        B_comp
                            (split d)
                            (B_comp
                                (B_op B_iota)
                                (ad 2 2))])
                    x)))

-- mask

p d (B_comp (P_split s t) (B_comp (B_op B_mask) x))
    = B_comp (P_map (B_op B_mask))
        (B_comp
            (B_op (P_zip zipArg))
            (p d (B_comp (addAT (P_split s t) 2) x)))

-- index

p d (B_comp x (B_comp (B_op B_index) y))
  = B_comp
          x
    (B_comp
        (B_op B_index)
    (B_comp
        (B_alltup[
            B_comp (B_op B_index) (B_alltup[ad 2 1, jgen]),
            kgen])
    (p d (B_comp
        (B_alltup[B_comp (split d) (ad 2 1), ad 2 2])
        y))))
```

```
-- priffle

p d (B_comp (P_split s t) (B_comp (B_op B_priffle) x))
    = B_comp
        (P_map (B_op B_priffle))
        (p d (B_comp (P_distpriff s t) x))

-- alltup

    -- compact consecutive alltups
p d (B_comp (B_alltup fs) (B_comp (B_alltup gs) x))
    = p d (B_comp (B_alltup (map (revComp (B_alltup gs)) fs)) x)

    -- look at next construct
p d (B_comp (B_alltup fs) x) = rejAT (insideA d (map (pll d) fs) x)

-- allvec

    -- compact allvec with alltup
p d (B_comp (B_allvec fs) (B_comp (B_alltup gs) x))
    = p d (B_comp (B_allvec (map (revComp (B_alltup gs)) fs)) x)

    -- look at next construct
p d (B_comp (B_allvec fs) y) = rejAV (insideA d (map (pll d) fs) y)


-- if
    -- parallelise if
p d (B_comp x (B_comp (B_if cnd thn els) y))
    = insideWhIf d x
                (map (spn d) (map (B_comp x) [thn,els]))
                [cnd,thn,els] y

-- while
    -- attempt to parallelise while
p d (B_comp x (B_comp (B_while lp chk) y))
    = insideWhIf d x [spn d (B_comp x lp)] [lp,chk] y


-- REPEAT & CLEANUP --
-----------------------

-- multiple parallel sections (redistributions)
p d (B_comp (P_split s t) (B_comp x (B_comp y z)))
    = B_comp (P_split s t) (B_comp x (append d (B_comp y z)))
p d (B_comp (P_split s t) x) = B_comp (P_split s t) x
p d (B_comp x y) = append d (B_comp x y)
p d x = x
```

```
-- The temporary function is removed

sweep (B_comp (P_reduce a) (B_comp (P_split s t) x)) = sweep x
sweep (B_comp x B_TEMP) = x
sweep B_TEMP = B_id
sweep (B_comp x y) = B_comp x (sweep y)
sweep x = x

-- EXTRA METHODS --
--------------------

-- Convenience functions
ad l i = B_addr (B_num l) (B_num i)
split (s,t) = P_split s t
redConc = P_reduce (B_op B_conc)
zipArg = B_alltup[ad 2 1, ad 2 2]

sOrD d x
 = (B_comp
      (B_op P_distl)
      (p d (B_comp (B_alltup[ad 2 1,B_comp (split d) (ad 2 2)]) x)))

    -- create general alltup with specified extra construct
addAT x l = B_alltup (addAT2 x (map (ad l) [1..l]))
addAT2 B_NULL y = y
addAT2 x y = map (B_comp x) y

    -- normalise, parallelise and sweep - no append
spn d f = sweep (p d (norm f))

-- transpose functions --

    -- check whether number of nodes < length of input
transIf (B_num s, B_num t)
    = B_comp (B_op B_lt) (B_alltup[B_con(B_int s),B_op B_length])

    -- check whether split was propagated further,
    -- if not put it back in alternative
checkDist d (B_comp iff (B_comp (P_split s t) x))
    = B_comp iff x
checkDist d (B_comp (B_if cnd thn els) x)
    = B_comp
        (B_if (remSplit (pll d cnd))
              (B_comp thn redConc)
              (remSplit els))
        x
    -- and remove reduce from consequent
remThnConc (B_comp splt (B_comp trans conc)) = B_comp splt trans
```

```
    -- does pll start with the transpose?
tPllStart True x y = y
tPllStart False x y = B_comp x y

    -- transpose
transPll d add (a,b) x
 | m==0  = checkDist d
            (B_comp
              (B_if
                (transIf d)
                (tPllStart add
                    (split d)
                    (B_op (B_transpose a b)))
                (tPllStart (not add)
                    redConc
                    (B_comp
                        (P_map (B_allvec[B_id]))
                    (B_comp
                        (B_op (P_transpose a b))
                    (B_comp
                        (P_map (B_comp (B_op B_index)
                              (B_alltup[B_id,B_con (B_int 0)])))
                        (split d))))))
              (p d (B_comp (split d) x)))

 | m>0   = tPllStart add
              redConc
              (B_comp
                 (P_map (B_map (B_op (B_transpose (a-1) (b-1)))))
                 (p d (B_comp (split d) x)))

    where m=minimum[a,b]


-- move splits from inside alltups/allvecs/if/while to ouside --
-----------------------------------------------------------------

    -- rejoining expressions
rejAT (fs,x) = B_comp (B_alltup fs) x
rejAV (fs,x) = B_comp (B_allvec fs) x

rejoin d (fs,B_NULL) x = (fs, append d x)
rejoin d (fs,splt) x = (fs, p d (B_comp splt x))

rejWhIf :: ([B_exp],B_exp) -> B_exp
rejWhIf ([chk,lp],x) = B_comp (B_while lp chk) x
rejWhIf ([cnd,thn,els],x) = B_comp (B_if cnd thn els) x
```

```
    -- match while/if
insideA d fs (B_comp (B_while lp chk) x) = atWhIf d fs [chk,lp] x
insideA d fs (B_comp (B_if cnd thn els) x)
= atWhIf d fs [cnd,thn,els] x
insideA d fs x = rejoin d (insideA2 d False fs) x

    -- did we have a split?
insideA2 d cont pfs = condMoveA d cont (endSplits pfs) pfs
condMoveA d cont rem fs
    | rem      = (map remSplit fs,split d)   -- yes
    | not cont = (fs,B_NULL)                 -- no
    | cont     = condMoveA2 d (endAS fs) fs  -- no, split.addr?

    -- did we have split.addr?
condMoveA2 d [] fs = (fs,B_NULL)                         -- no
condMoveA2 d bs fs = (map (remAS bs) fs, compAS d bs)   -- yes

    -- look for split.addr
atWhIf d fs attr x = atWhIf2 d (insideA2 d True fs) attr x

    -- anything to propagate?
atWhIf2 d (fs,B_NULL) attr x = (fs, p d (rejWhIf (attr,x))) -- no
atWhIf2 d (fs,add) attr x = (fs, preWhIf d add attr x)     -- yes

    -- evaluate address
addrAT (B_num i) fs y = n (B_comp (fs !! (i-1)) y)

    -- push split into whif, creating vector with one true
addrWhIf d l i attr x = B_comp(B_addr l i)(addrWhIf2 d l i attr x)
addrWhIf2 d (B_num l) (B_num i) attr x
    = preWhIf d (compAS d (setTrue i (replicate l False))) attr x

    -- add to the inner elements & push through
preWhIf d add attr x
    = insideWhIf d add (map (spn d) (map (B_comp add) (tail attr))) attr x

    -- insideWhIf: check whether splits have been pushed through
insideWhIf d add pAttr attr x
    = condMoveWhIf d add
                (insideA2 d True (pll d (head attr):pAttr)) attr x

condMoveWhIf d add (pAttr,B_NULL) attr x      -- was not pushed through
    = add1 add (rejWhIf (attr, append d x))

condMoveWhIf d B_NULL (pAttr,pushd) attr x    -- pushed through, no suffix
    = noSuffMoveAT d pAttr pushd (p d (B_comp pushd x))

condMoveWhIf d add (pAttr,B_alltup gs) attr x -- pushed through with alltup
    = suffMoveAT d add pAttr gs (p d (B_comp (B_alltup gs) x))
```

```
condMoveWhIf d add (pAttr,splt) attr x        -- pushed through with split
    = rejWhIf (pAttr, p d (B_comp splt x))


noSuffMoveAT d [chk,lp] (B_alltup gs) x       -- pushed with alltup, while
    = B_comp
        (B_alltup (map add2 gs))
        (B_comp (B_while (remConcAT d gs lp) chk) x)

noSuffMoveAT d [chk,lp] splt x                -- pushed with split, while
    = B_comp redConc (B_comp (B_while (remConc d (compNorm lp)) chk) x)

noSuffMoveAT d (cnd:thnels) pushd x           -- pushed, if
    = noSuffIf d cnd(map compNorm thnels) x


    -- see whether i can move reduce\concat out the front

noSuffIf d cnd [B_alltup thn,B_alltup els] x  -- have tuple input
    = noSuffIfAT d cnd (map B_alltup [thn,els])
        (unzip (map (tupOr (length thn))
            (zip3 (ifConcs thn) (ifConcs els) [1..(length thn)])))
        x

noSuffIf d cnd thnels x                       -- have vector input
    = noSuffIfVec d cnd thnels (and (map ifConc thnels)) x

    -- moved concs?
noSuffIfAT d cnd thnels (bs,concs) x
    | b        = B_comp (B_alltup concs)
                 (rejWhIf ((cnd:map (remConcAT d concs) thnels),x))
    | not b = rejWhIf ((cnd:thnels),x)
    where b=(or bs)

    -- moved conc?
noSuffIfVec d cnd thnels b x
    | b        = B_comp redConc (rejWhIf (cnd:map (remConc d) thnels,x))
    | not b = rejWhIf((cnd:thnels),x)

    -- it is the same either way for if
    -- (doesn't matter if stuff was brought in or not)
suffMoveAT d add [cnd,thn,els] gs x
    = noSuffMoveAT d [cnd,thn,els] B_NULL x

    -- while: fs is tuple brought in, gs is the one to be taken out
suffMoveAT d (B_alltup fs) [chk,lp] gs x
    = suffMoveAT2 d (unzip3 (map unifyATs (zip fs gs))) chk lp x
```

```
    -- complete concs are out, splits are out.
suffMoveAT2 d (bs,bs2,gs) chk (B_alltup lp) x
 | b      = B_comp (B_alltup gs)
             (rejWhIf([chk,B_alltup(map(doConcs d)(zip3 bs bs2 lp))],x))
 | not b  = rejWhIf ([chk,B_alltup lp],x)
 where b=(or bs)

    -- for if: end in reduce/conc?
ifConcs fs = map ifConc fs
ifConc (B_comp (P_reduce (B_op B_conc)) x) = True
ifConc (P_reduce (B_op B_conc)) = True
ifConc x = False

    -- or operator for tuple
tupOr l (a,b,i)
    | aorb        = (True,B_comp redConc (ad l i))
    | not aorb    = (False,ad l i)
    where aorb=(a||b)

    -- match up concs and splits going out for while
unifyATs (B_comp (P_split s t) x , B_comp (P_split u v) y)
        = (False,False,x)
unifyATs (B_comp (P_split s t) x , y)
        = (True,True,B_comp (P_split s t) x)
unifyATs (x , B_comp (P_split u v) y)
        = (True,False,B_comp redConc x)
unifyATs (x,y) = (False,False,x)

    -- create list of concs\splits to be appended
doConcs d (False,f, x) = x
doConcs d (t,True, B_comp (P_split u v) x) = x
doConcs d (t,True, x) = B_comp redConc x
doConcs d (t,False, B_comp (P_reduce (B_op B_conc)) x) = x
doConcs d (t,False, x) = B_comp (split d) x

    -- conditional composition
add1 B_NULL x = x
add1 x B_NULL = x
add1 add x = B_comp add x
    -- another conditional composition used for appending concs
add2 (B_comp s g) = B_comp redConc g
add2 g = g

    -- remove concs from alltup
remConcAT d gs (B_alltup fs)
    = B_alltup (map (remConcATs d) (zip gs fs))
remConcAT d gs x = x
remConcATs d (B_comp s g, f) = remConc d f
remConcATs d (g, f) = f
```

```
    -- remove single conc
remConc d (P_reduce (B_op B_conc)) = B_id
remConc d (B_comp (P_reduce (B_op B_conc)) x) = x
remConc d x = B_comp (split d) x

    -- check for split
endSplits :: [B_exp] -> Bool
endSplits fs = endSplt2 fs False

endSplt2 [] q = q
endSplt2 (P_split s t :fs) q = endSplt2 fs True
endSplt2 (B_comp x y :fs) q = endSplt2 (y:fs) q
endSplt2 (B_addr l i :fs) q = False
endSplt2 (f:fs) q = endSplt2 fs q

    -- remove split
remSplit :: B_exp -> B_exp
remSplit B_id = redConc
remSplit (B_con a) = B_con a
remSplit (P_split a b) = B_id
remSplit (B_comp x (P_split a b)) = x
remSplit (B_comp x y) = B_comp x (remSplit y)
remSplit x = B_comp x redConc

    -- check for split.addr
endAS fs = endAS2 fs []
endAS2 [] bs = bs
endAS2 (B_comp (P_split s t) (B_addr (B_num l) (B_num i)) :fs) bs
    | bs/=[]    = endAS2 fs (setTrue i bs)
    | bs==[]    = endAS2 fs (setTrue i (replicate l False))
endAS2 (B_comp x y :fs) bs = endAS2 (y:fs) bs
endAS2 (B_alltup gs :fs) bs = endAS2 (gs++fs) bs
endAS2 (B_allvec gs :fs) bs = endAS2 (gs++fs) bs
endAS2 (x:fs) bs = endAS2 fs bs

    -- remove split.addr
remAS bs (B_comp (P_split s t) (B_addr l i)) = B_addr l i
remAS bs (B_comp x y) = B_comp x (remAS bs y)
remAS bs (B_addr l (B_num i))
    | b     = B_comp redConc (B_addr l (B_num i))
    | not b    = B_addr l (B_num i)
    where b = bs !! (i-1)
remAS bs (B_alltup gs) = B_alltup (map (remAS bs) gs)
remAS bs (B_allvec gs) = B_allvec (map (remAS bs) gs)
remAS bs x = x
```

```
    -- create alltup with splits where appropriate
compAS d bs = B_alltup (map (compAS2 d (length bs))
                            (zip bs [1..(length bs)]))
compAS2 d l (b,i)
    | b        = B_comp (split d) (ad l i)
    | not b    = ad l i

    -- sets idx element of vector to true
setTrue :: Int -> [Bool] -> [Bool]
setTrue idx (b:bs)
    | idx==1    =    [True]++bs
    | idx> 1    =    [b]++(setTrue (idx-1) bs)


-- while/if: parallelisation --
-- need to normalise, add TEMP and sweep

whIfSwp (B_comp (B_while lp chk) x)
    = whIfSwp2 (map compNorm [lp,chk]) x
whIfSwp (B_comp (B_if cnd thn els) x)
    = ifSwp1 (map compNorm [cnd,thn,els]) x
whIfSwp (B_comp x y) = B_comp x (whIfSwp y)
whIfSwp x = x

    -- produce if suffix
ifSwp1 attr x = ifSwp2 (suffix (tail attr)) attr x
ifSwp2 suff attr x = add1 suff (ifSwp3 (prefix suff) attr x)
ifSwp3 pref (a:attr) x
    = whIfSwp2 (a: map compNorm (map (add1 pref) attr)) x

    -- get structure of input
whIfSwp2 attr x = whIfSwp3 (map revNorm attr) x
whIfSwp3 attr x = whIfSwp4 (getStruct attr) attr x

    -- put prefix inside construct
whIfSwp4 str attr x
    = whIfSwp5 str (map revSwp (map (repStruct str) attr))
                       (add1 (prefix str) x)

    -- while loop still to be flattened - complete!
whIfSwp5 str [lp,chk] x = add1 str (rejWhIf ([chk, flatten lp],x) )
whIfSwp5 str attr x = rejWhIf (attr,x)
```

```
-- compact and normalise

compNorm lp = sweep (compact (norm lp))

    -- remove id
compact (B_comp x (B_comp B_id y)) = compact (B_comp x y)
compact (B_comp B_id (B_comp x y)) = compact (B_comp x y)

    -- compact!
compact (B_comp (B_alltup fs) x)
    = B_alltup (map compNorm (map (revComp x) fs))

    -- evaluate addresses
compact (B_comp (B_addr l i) x) = addrComp B_NULL (B_addr l i) x
compact (B_comp x (B_comp (B_addr l i) y))
    = addrComp x (B_addr l i) y

compact (B_comp x y) = B_comp x (compact y)
compact x = x


    -- this is a subset of the code for the address sweep
    -- (during parallelisation)

addrComp x addr y = addrCompRec x [addr] y

addrCompRec x (B_addr l i :addrs) (B_comp (B_alltup fs) y)
    = addrCompRec x addrs (addrAT i fs y)
addrCompRec x addr (B_comp (B_addr l i) y)
    = addrCompRec x (B_addr l i:addr) y
addrCompRec x [] y = add1 x y
addrCompRec x as (B_comp B_id y) = addrCompRec x as y
addrCompRec x as y = add1 x (addrCompRejoin (reverse as) y)
addrCompRejoin [] y = y
addrCompRejoin (a:as) y = B_comp a (addrCompRejoin as y)


-- create structure for if suffix

suffix [fs,gs] = struc2bmf (countStr (getSuffix (fs,gs)))

getSuffix (B_alltup fs, B_alltup gs) = atSuffix fs gs
getSuffix (B_alltup fs, y) = repNull fs
getSuffix (x, B_alltup gs) = repNull gs
getSuffix (x, y) = Addr

repNull xs = atSuffix xs (replicate (length xs) B_NULL)
atSuffix fs gs = Alltup 0 (map getSuffix (zip fs gs))
```

```
-- getStruct: create structure for prefix (and substitutions)

getStruct :: [B_exp] -> B_exp
getStruct attr = struc2bmf (countStr (struct2 attr Addr))

struct2 :: [B_exp] -> Structure -> Structure
struct2 [] str = str
struct2 (a:attr) str = struct2 attr (struct3 a str)

struct3 :: B_exp -> Structure -> Structure
struct3 (B_comp x (B_addr l (B_num i))) (Alltup n ss)
    = Alltup n (subStruct x i ss)
struct3 (B_comp x (B_addr (B_num l) (B_num i))) Addr
    = struct3 (B_comp x (ad l i)) (Alltup 0 (replicate l Addr))
struct3 (B_comp x (B_alltup fs)) str = struct2 fs str
struct3 x str = str

    -- get appropriate element
subStruct :: B_exp -> Int -> [Structure] -> [Structure]
subStruct x idx (s:ss)
    | b        = ( struct3 x s : ss )
    | not b    = ( s : subStruct x (idx-1) ss)
    where b=(idx<=1)


-- count elements at each level of structure

    -- get to the top
countStr :: Structure -> Structure
countStr (Alltup i fs) = addStr (Alltup i (map countStr fs))
countStr Addr = Addr

    -- sum from top to bottom
addStr (Alltup i fs) = Alltup (sum (map addStrs fs)) fs
addStrs (Alltup i fs) = i
addStrs Addr = 1


-- struc2bmf - create BMF expression from structure

struc2bmf :: Structure -> B_exp
struc2bmf (Alltup l fs) = B_alltup (s2bmf2 l 1 fs)
struc2bmf Addr = B_NULL

s2bmf2 n idx [] = []
s2bmf2 n idx (Alltup l fs :gs)
    = (B_alltup (s2bmf2 n idx fs) : s2bmf2 n (idx+l) gs)
s2bmf2 n idx (Addr :gs) = (ad n idx : s2bmf2 n (idx+1) gs)
```

```
-- repStruct: replace addr's with extended structure

repStruct :: B_exp -> B_exp -> B_exp

repStruct str (B_comp x (B_alltup fs))
    = add1 x (B_alltup (map (repStruct str) fs))
repStruct (B_alltup fs) (B_comp x (B_addr l (B_num i)))
    = repStruct (fs !! (i-1)) x
repStruct y x = add1 x y


-- flatten: removes nested alltups from lp expression

flatten (B_alltup fs) = B_alltup (flatten2 (B_alltup fs))
flatten x = x

flatten2 :: B_exp -> [B_exp]
flatten2 (B_alltup fs) = concat (map flatten2 fs)
flatten2 x = [x]


-- prefix - create 2nd part of identity - a flattened structure

prefix (B_alltup str) = B_alltup (prefix2 str)
prefix x = x

prefix2 str
    = concat (map (prefix3 (length str)) (zip [1..(length str)] str))

prefix3 n (i,B_alltup fs) = map (revComp (ad n i)) (prefix2 fs)
prefix3 n (i,x) = [ad n i]

revComp x y = B_comp y x


-- reverse normalise - including inside alltups

revNorm x = revNorm3 (revNorm2 x)

revNorm2 (B_comp x (B_comp y z)) = revNorm2 (B_comp (B_comp x y) z)
revNorm2 (B_comp x y) = B_comp (revNorm2 x) y
revNorm2 x = B_comp B_TEMP x

revNorm3 (B_comp x (B_alltup fs))
    = B_comp x (B_alltup (map revNorm fs))
revNorm3 x = x
```

```
-- remove the reverse B_TEMP - including from alltups

revSwp (B_comp x (B_alltup fs))
    = revSwp2 (B_comp x (B_alltup (map revSwp fs)))
revSwp x = revSwp2 x

revSwp2 (B_comp B_TEMP x) = x
revSwp2 (B_comp x y) = B_comp (revSwp2 x) y
revSwp2 x = x



-- addr sweep - create list of addresses --
-------------------------------------------

addrSwp d x addr y = addrSwpRec d x [addr] y

    -- apply to alltup if possible
addrSwpRec d x (B_addr l i :addrs) (B_comp (B_alltup fs) y)
    = addrSwpRec d x addrs (addrAT i fs y)

    -- try to push a split into a while/if
addrSwpRec d (P_split s t) [B_addr l i] (B_comp (B_while lp chk) x)
    = addrWhIf d l i [chk,lp] x
addrSwpRec d (P_split s t) [B_addr l i] (B_comp (B_if cnd thn els) x)
    = addrWhIf d l i [cnd,thn,els] x

    -- found another address
addrSwpRec d x addr (B_comp (B_addr l i) y)
    = addrSwpRec d x (B_addr l i:addr) y

    -- remove id
addrSwpRec d x as (B_comp B_id y) = addrSwpRec d x as y

    -- no more addresses in the list or the program
addrSwpRec d B_NULL [] y = append d y
addrSwpRec d x [] y = p d (B_comp x y)

    -- addresses in list - replace them recursively
addrSwpRec d x as y = add1 x (addrSwpRejoin d (reverse as) y)

addrSwpRejoin d [] y = append d y
addrSwpRejoin d (a:as) y = B_comp a (addrSwpRejoin d as y)
```

```
-- replacements for scan and index parallelisations --
-------------------------------------------------------

g a = B_comp
        (B_op P_conc)
        (B_alltup[
            B_comp
                (P_split (B_num 1)(B_num 0))
                (B_comp
                    (B_op B_index)
                    (B_alltup[
                        ad 2 2,
                        B_con (B_int 0)])),
            B_comp
                (P_map (odot a))
                (B_comp
                    (B_op (P_zip zipArg))
                    (B_alltup[
                        ad 2 1,
                        B_comp
                            (mytail)
                            (ad 2 2)]))])

odot a = B_comp
        (B_map a)
     (B_comp
        (B_op (B_distl))
        (B_alltup[ad 2 1,ad 2 2]))

mytail =
    B_comp
        (B_op P_select)
        (B_alltup[
            B_id
            ,
            B_comp
                (B_map(B_comp
                        (B_op (B_plus))
                        (B_alltup[
                            B_con (B_int 1),
                            B_id])))
            (B_comp
                (B_op (B_iota))
            (B_comp
                (B_op (B_minus))
                (B_alltup[
                    B_op (B_length),
                    B_con (B_int 1)]))))])
```

```
myinit = B_comp
        (B_op P_select)
        (B_alltup[
            B_id,
            B_comp
                (B_op (B_iota))
            (B_comp
                (B_op (B_minus))
                (B_alltup[B_op(B_length),B_con(B_int 1)]))])

mylast = B_comp
        (B_op B_index)
        (B_alltup[
            B_id
            ,
            B_comp
                (B_op (B_minus))
                (B_alltup[B_op(B_length),B_con(B_int 1)])])

jgen
 =  B_comp
         (ad 2 1)
    (B_comp
        (P_reduce
         (B_if (B_comp (ad 2 1) (ad 2 2))(ad 2 1)(ad 2 2)))
    (B_comp
        (B_op (P_zip zipArg))
    (B_comp
        (B_alltup[
            B_comp
                (P_map(
                    B_comp
                        (B_op B_minus)
                        (B_alltup[B_id,B_con (B_int 1)])))
            (B_comp
                (P_scan (B_op B_plus))
                (P_map (B_con (B_int 1)))),
            B_id])
    (B_comp
        (P_map (B_if (B_op B_lt)(B_con B_true)(B_con B_false)))
    (B_comp
        (B_op P_distl)
        (B_alltup[
            ad 2 2,
            B_comp
                (P_scan (B_op B_plus))
            (B_comp
                (P_map (B_op B_length))
                (ad 2 1))])))))))
```

```
kgen
 =    B_comp
         (P_reduce (B_if
                     (B_comp
                         (B_op B_lt)
                         (B_alltup[(ad 2 2),B_con (B_int 0)]))
                     (ad 2 1)
                     (ad 2 2)))
      (B_comp
          (P_map (B_op B_minus))
      (B_comp
          (B_op P_distl)
          (B_alltup[
              (ad 2 2)
              ,
              B_comp
                  (B_op P_conc)
              (B_comp
                  (B_alltup[
                      B_comp
                          (P_map (B_comp (B_op B_index)
                              (B_alltup[B_id,B_con (B_int 0)])))
                      (B_comp
                         (P_split (B_num 1) (B_num 0))
                         (B_allvec[B_con (B_int 0)]))
                     ,
                     myinit
                      ])
              (B_comp
                  (P_scan (B_op B_plus))
              (B_comp
                  (P_map (B_op B_length))
                 (ad 2 1))))])))
```