Mapping Parallel BMF Constructs to a Parallel Machine

Dean Philp, B.Sc (Ma. & Comp.Sc.)

November 2003

Department of Computer Science,
University of Adelaide

Supervisors: Brad Alexander and Dr Andrew Wendelborn

Submitted in partial fulfilment of the requirement for the
Masters Degree in Computer Science

Abstract

The Von-Neumann computational model and single processor architecture is the predominant computation and architecture coupling used throughout the sequential computing world. In the world of parallel computing there is no widely accepted model of computation and there are a large number of parallel architectures. If parallel computing is to gain similar acceptance to that of sequential computing then there needs to be more research towards finding a unified model of computation.

The goal of the Adl project is to provide an efficient implementation of a data-parallel language in the framework of a distributed memory architecture. Adl boasts implicit parallelism and architecture independence, which are desirable features of a parallel model of computation. Implicit parallelism is achieved by defining operations on aggregate data-structures, and architecture independence through algebraic transform ation of an intermediate form BMF. This project has developed a back-end parallel implementation of Adl by defining a translation from BMF code to C/MPI code. Initial experiments with code produced by the translator demonstrate promising levels of speedup.

Acknowledgements

I would like to thank my supervisors Brad Alexander and Andrew Wendelborne, for their guidance and support.

Thanks to my fellow masters students especially Sam Moskwa and Thomas Papadopopoulos for your help and friendship this year.

Finally, a warm thank you to Emily Moskwa for your moral support throughout the year.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

1.1.1 Parallelism

The Von-Neumann computational model and single processor architecture is the predominant computation and architecture coupling used throughout the sequential computing world. Given this model, a sequential program can run on different architectures with predictable performance. However, the Von-Neumann model enforces an ordering and sequencing on instructions more strict than that required by most programs. Instructions can often be executed in *parallel* (using multiple processors) without loss of correctness. Parallel execution of a program can be natural for many applications and can decrease execution time dramatically.

Aside from the opportunity for increased performance there are a four important reasons why parallelism is attractive, as discussed in [1]. First, performance increases are bounded in sequential computing due to physical factors such as the speed of light, though it is not known how close sequential architectures are to this limit. Second, it is incredibly expensive to research and build new generations of sequential processors. Therefore, in terms of performance per dollar, it is cheaper to combine older sequential processors, as long as the interconnection network costs are reasonable. Third, sometimes we cannot afford to wait for faster single processors because certain applications *need* immediate performance increases. And fourth, distributed memory parallel computers provide increased cache and memory, which is pivotal for many data-intensive scientific, and data-mining applications.

1.1.2 Current Problems

Because there are many scientific and commercial applications looking to exploit parallelism, and with cheaper high-performance parallel computers, parallel computing has become increasingly accessible in recent times. A number of issues still need to be addressed however, in relation to both software and hardware.

Today, there are two predominant types of parallel architectures, shared memory machines and distributed machines. Shared memory computers are relatively easy to program but do not scale to large numbers of processors. Distributed memory machines are hard to program but are scalable. A lack of an accepted parallel model of computation means that programs end up being customised to a particular parallel architecture. Furthermore, due to constant improvements in interconnection network technology and uni-processor clock-

rates doubling every 18 months [2,3], parallel architectures can become outdated within 5 years. Therefore, because of its tight coupling with architectures, parallel software also becomes obsolete.

Parallelism is therefore faced with a multitude of problems that affect programmer's willingness to explore it, and consequently the widespread acceptance of parallel computing is compromised. Given the aforementioned issues, it seems obvious that there is a requirement for a widely accepted parallel model of computation that abstracts details of parallel architectures.

1.2 Objective

The goal of the Adl project is to provide an efficient implementation of a data-parallel language in the framework of a distributed memory architecture. Adl boasts implicit parallelism and architecture independence, which are desirable features of a parallel model of computation. Implicit parallelism is achieved by defining operations on aggregate data-structures, and architecture independence through algebraic transformation of an intermediate form BMF.

The aim of this project is to provide a back-end implementation of Adl by defining a translation from BMF code to C/MPI code. This involves the development of two new compiler components, a translator and its target code implementation. Consequently, the Adl language project could provide a usable functional programming language with automatic (or user guided) parallel performance, which may contribute to the widespread adoption of parallel computing in the future.

1.3 Thesis Outline

Chapter 2 describes the skeletal parallel model of computation and the related functional notation BMF, both closely coupled to the scope of this thesis. Chapter 3 starts by describing the Adl project and the constituent components completed to date. It then progresses to outline the new Adl compiler components developed by this project, which provides the necessary foundation to understand the form of its source and target code.

Chapter 4 describes the implementation of the translator and its target code, leaving complete details for appendices B and C. Chapter 5 then demonstrates the success of the implementation by providing the translation of example programs and corresponding speedup and efficiency tests. In addition, it discusses the findings and difficulties that arose throughout the development of the project. This provides the stimulation for the conclusions and future work presented in Chapter 6.

2 Related Work

2.1 Skeletons

One promising model of parallel computation is skeletons [1]. Parallel computation is defined using the skeletal approach by composing a number of 'building blocks', where the implementation of these blocks is predefined. The following table assesses skeletons with respect to six criteria listed by Skilicorn and Talia in measuring the worthiness of a parallel model of computation.

Criteria	Supported
Programmability	Yes
Development methodology	Not Entirely
Parallel architecture independence	Not Entirely
Intuitiveness	Yes
Efficiently implementable	Yes
Provides cost measures	Yes

Table 1: Skeletons as a parallel model [1]

As the table shows, skeletons fulfill many desired qualities of a parallel model of computation. Programs are sequentially composed with predefined parallel skeletons, without concern with difficulties relating to communication and partitioning, and thus are both implicitly parallel and relatively easy to program. Skeletons are intuitive since programmers need only understand the results obtained from a skeletal construct and not the inner complexities involved in its construction. Each skeleton can be implemented and tuned for performance once for each type of parallel architecture and therefore are efficiently implementable. Cost measures can be provided for each parallel skeleton because communication patterns for each construct are encapsulated. However, issues of software development methodology and parallel architecture independence are not entirely supported by skeletons when used in isolation.

Herbert Kuchen has conducted various research related to parallel programming using skeletons. In particular he developed a polymorphic skeleton library in C++/MPI, allowing both task and data parallelism with higher-order functions and partial applications [4]. The motivation for this approach is programmers do not have to learn a new language, and low-level communications and deadlock problems are abstracted over. This library approach is not implicitly parallel, and therefore differs to the Adl project. Furthermore, polymorphism is not required in the Adl's parallel C/MPI implementation since type information is available.

Another skeleton related project is the P3L project (Pisa Parallel Programming Language), which includes data-parallel (map, reduce and scan), task-parallel

(farm and pipe) and control parallel (loop and seq) skeletons [5]. P3L (based on the syntax of C) uses a library of 'implementation templates', targeted to a parallel architecture with associated cost models. The current compiler, ANACLETO, generates C/MPI code based on the attributes of each implementation template. The P3L project differs from the Adl project since it targets skeletons in the imperative paradigm, whereas Adl is concerned with functional programming. Furthermore, Adl achieves architecture independence through algebraic transformation of an intermediate from BMF, which is a different approach.

2.2 BMF

BMF (the Bird-Meertens Formalism) is a functional notation based on categorical data-types and operations on them [6]. While BMF is not solely intended to be a parallel model computation it does define its computation in a skeletal style (section 2.1). BMF does, however, provide many important features of a parallel model of computation such as parallelism, architecture independence and a well-defined software development methodology [7].

An important feature of BMF is its support for massive parallelism through domain decomposition of data. Massive parallelism is important for many applications, such as weather forecasting, and is therefore a desirable feature of a parallel language. Furthermore, because computation is defined in a skeletal way, any parallel complexity is hidden, and therefore BMF programs can be regarded as implicitly parallel.

The BMF theory (for each type) has associated equations for incrementally transforming one version of a BMF program to another. It is therefore possible for a compiler, given enough information, to automatically (using these equations) transform an inefficient program, on some parallel architecture, to an efficient one[1], therefore achieving architecture independence [7]. This equational transformation process is also the view of software development in BMF where programs are incrementally transformed to a solution and then optimised for efficiency [7]. Furthermore, because programs are transformed by algebraic means, semantics are preserved.

One categorical data-type heavily used in BMF is lists, with operations such as map, reduce, scan, select and zip. Some informal definitions of how these constructs work are provided below.

$$f * [x_0, x_1, x_2, ..., x_{n-1}] = [f(x_0), f(x_1), f(x_2), ..., f(x_{n-1})]$$

Definition 1: map (*) a function f over a list

[1] Determining which version is more efficient is a cost modelling issue [8].

$$\oplus/[x_0, x_1, x_2, ..., x_{n-1}] = x_0 \oplus x_1 \oplus x_2 .... \oplus x_{n-1}$$

Definition 2: reduce (/) inserts binary function between all elements of list

$$\oplus//[x_0, x_1, x_2, ..., x_{n-1}] = [x_0, x_0 \oplus x_1, ..., x_0 \oplus x_1 \oplus ... \oplus x_{n-1}]$$

Definition 3: scan (//) = Reduce but storing cumulative results

96.36 709.04032t(sv, [i/Span <</MCID 37 >>BDCB4/TT2 1 Tf-0.0011 Tc9

3 Context Of This Work

3.1 The Adl Project

3.1.1 Outline

The purpose of the Adl language project, described in [12], is to show that an efficient data-parallel functional programming language can be developed in the framework of a distributed memory architecture. Adl itself is a simple polymorphic, non-recursive, strictly evaluated functional language where most computation is defined using high-level operations on aggregate data-structures, such as lists. Adl supports common operations on aggregate structures such as indexing and length operations on vectors and pattern matching to access elements of tuples. It also provides nested scooping and access to global variables. The four broad stages of the Adl project are depicted in figure 1 below.

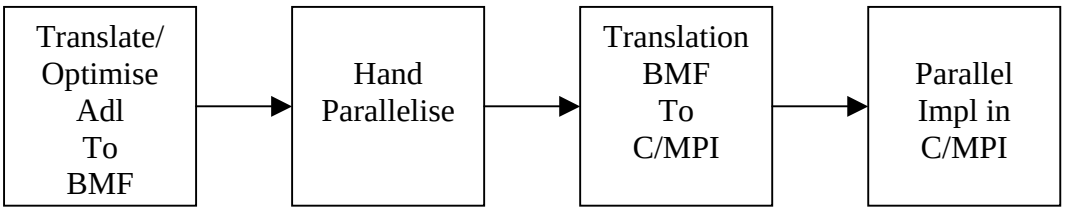


Figure 1: The Adl complier

The first stage of compilation of an Adl program is to translate it to sequential BMF. Section 2.2 described how BMF defines computation over aggregate data-structures such as lists, trees, bags etc. Similarly, Adl also defines most computation over aggregate data structures and therefore it is relatively easy to define a systematic (and automatic) translation scheme from Adl to sequential BMF [13]. As [13] describes, the fundamental difference between Adl and BMF is that BMF transports function values to all parts of the program where they are in scope, whereas Adl values are accessed via naming variables. This language difference means that the raw translated BMF code is quite inefficient and so an automated data-movement optimisation process has been developed [14], with resultant optimised constructs including all those described in section 2.2.

The next stage of compilation is to parallelise the optimised BMF code. One of the advantages of BMF, in terms of a parallel model of computation, is that it can be regarded as *implicitly* parallel. The *explicit* parallelisation stage of the Adl project effectively breaks this BMF property, which is not necessarily a bad thing. Explicit parallelisation allows us to use analysis that can exploit knowledge of machine boundaries, which can have significant performance advantages [15]. In any case, the Adl language itself *is* implicitly parallel, from

the programmer's point of view. It is envisaged that the parallelisation step in the Adl project will eventually be automated, but currently code is hand parallelised using a set of prototype strategies [16,17]. The following example gives a general strategy for parallelising a sequential BMF program, bearing in mind that the sequential optimisation process will have already taken place before any parallelisation occurs.

Starting with a vector argument:

$$f_0 \cdot f_1 \cdot f_2 .... \cdot f_{n-1} \qquad (1)$$

Insert the BMF code:

$$++/^{||}\ split_p \qquad (2)$$

At the rear of the equation (1), above, to obtain:

$$++/^{||} \cdot split_p \cdot f_0 \cdot f_1 \cdot f_2 .... \cdot f_{n-1} \qquad (3)$$

where ++ denotes concatenation, and $/^{||}$ denotes parallel reduction. $Split_p$ divides and input vector into p individual sub-vectors and distributes them over p processing nodes. Note that equation (2) denotes an identity function on lists and does not change the semantics of the program. Also note that BMF programs are written right-to-left. The parallelisation continues from this point by pushing the $split_p$ construct as far through the program as possible.

So the next step will yield the program:

$$++/^{||} \cdot f'_0 \cdot split_p \cdot f_1 \cdot f_2 \cdot .... \cdot f_{n-1} \qquad (4)$$

where f'_0 represents the parallelised version of f_0. We now continue in this fashion until we obtain the program:

$$++/^{||} \cdot f'_0 \cdot f'_1 \cdot f'_2 \cdot .... \cdot f'_{n-1} \cdot split_p \qquad (5)$$

So that the entire program has been (attempted to be) parallelised. For more information regarding the parallelisation process and the Adl language itself, refer to [16,17].

It is important to note that the Adl project fulfills some important requirements, in terms of a parallel model of computation, presented by skeletons in section 2.1. A clean software development methodology is clearly provided in the form of a high-level non-recursive functional language and equational transformation after BMF translation. Parallel architecture independence is achieved by only mapping a small set of parallel BMF primitives to a parallel machine. It is likely that this approach will only require small modifications to the implementation to achieve maximum performance on different parallel architectures, but more investigation is needed to confirm this.

3.1.2 Adl Example

Before moving on to the detailed scope of this project, consider the following example of Adl code to compute the sum of squares of a vector of numbers:

```
main a: vof int :=
```

```
            let
                 add (x,y) := x+y;
                 square x := x*x;
            in
                 reduce(add,0,map(square,a))
            endlet
       $INPUT
```

Where $INPUT refers to a vector of integers. The earlier stages of the Adl compiler then perform translation to sequential BMF, optimisation and parallelisation. These steps yield the following parallel BMF equivalent:

```
(B_program
  (B_comp
     (P_reduce (B_op (B_plus)))
     (B_comp
        (P_map (B_map (B_comp (B_op (B_times))
                              (B_alltup [B_id,B_id]))))
     (P_split (B_num 8) (B_num 0)))))
(IntList (16,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]))
```

In this trivial example we see that the resultant parallel BMF code is quite similar to its original Adl form, this will not generally be the case. The final stage(s) of the Adl compilation process is the objective of this project and involves mapping parallel (or otherwise) BMF constructs to a target parallel architecture.

3.2 The Compiler

The previous sections have outlined the current status of the Adl project and provided an example of Adl and its resultant parallel BMF form. The scope of this thesis (and the final stage(s) of the Adl project) is the mapping of parallel BMF constructs to a target parallel architecture. The shaded stages in figure 2 below clearly demonstrate the work this thesis is concerned with.

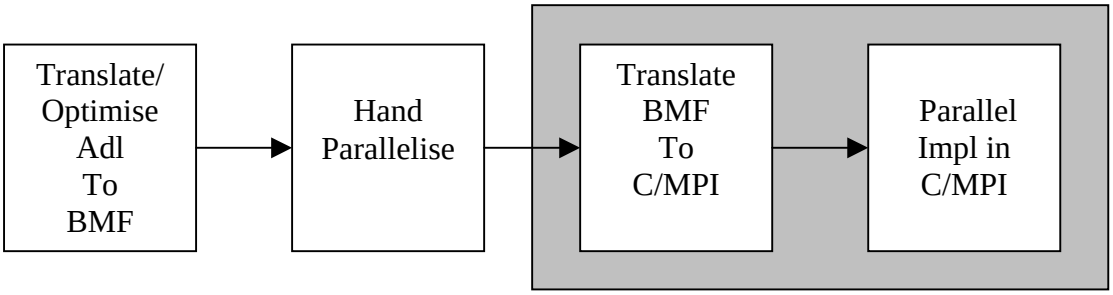


Figure 2: The scope of this thesis (stages 1-4)

The following sections provide information required to gain understanding of the implementation stages described in chapter 4.

3.2.1 Source Language

An outline of the translation process from Adl to BMF, and how subsequent parallelisation is performed has been described. The next step is for the compiler to define a framework for translating BMF code to the target language, which will ultimately run on a parallel machine. Clearly the compiler will require a well-defined process for converting parallel BMF operations to the target language, which can be implemented in C/MPI on a target parallel architecture

The mechanism for which parallel BMF code is translated to the target language is directly analogous to code generation in a conventional compiler. The figure below is a recursive type definition, written in Miranda™, used to define the syntax (or form) of parallel BMF, followed by a short description of each of its components.

```
b_exp ::=
     B_id |
     B_con b_con |
     B_comp b_exp b_exp |
     B_if b_exp b_exp b_exp |
     B_alltup [b_exp] |
     B_allvec [b_exp] |
     B_map b_exp |
     B_op b_op |
     B_reduce b_exp b_exp |
     B_scan b_exp b_exp |
     B_addr b_num b_num |
     B_zip b_exp |
     B_distl |
     B_repeat |
     P_map b_exp |
     P_reduce b_exp |
     P_scan b_exp |
     P_split b_num b_num |
     P_zip b_exp |
     P_repeat b_exp |
     P_distl |
     P_project |
     B_program b_exp inputType
```

Figure 3: Parallel BMF syntax

Note for below: definitions 1,2,3,4 and 5 can be found in section 2.2.

- `B_id`: identity function.
- `B_con`: constant function; integers, reals and booleans.

- `B_comp`: function composition; `(B_comp e2 e1)` means e1 is evaluated before e2.
- `B_if`: if predicate, then {consequent}, else {alternative}.
- `B_alltup`: Apply every function of [b_exp] to a copy of the input, creating a tuple of output values.
- `B_allvec`: Apply every function of [b_exp] to a copy of the input, creating a vector of output values.
- `B_map`: Sequentially apply a function to all elements of a list (defn 1).
- `B_op`: BMF operators; length, less-than, indexing, etc.
- `B_reduce`: Sequential reduce (definition 2).
- `B_scan`: Sequential scan (definition 3).
- `B_addr`: Address an element of a B_alltup.
- `B_zip`: Sequential zip (definition 5).
- `P_map`: Apply a function in parallel to all elements of a list (defn 1).
- `P_reduce`: Parallel reduce (definition 2).
- `P_scan`: Parallel scan (definition 3).
- `P_split`: Distribute (split) a list among processors.
- `P_zip`: Parallel zip (definition 5).
- `P_repeat`: Repeat a value a number of times over processors, creating a distributed list.
- `P_distl`: Distribute a value over a distributed list, creating a distributed list of pairs.
- `P_project`: Parallel select (definition 4).
- `B_program`: A parallel BMF program. Composed of a b_exp and input 'data' (inputType).

The role of the b_exp type definition (figure 3) and the details of the translation system will be discussed in more detail in section 4.1.

3.2.2 Target Language

The previous section described the parallel BMF source language and figure 3 depicted its syntax. The translation system effectively provides a mapping between parallel BMF constructs and target parallel language code:

BMF Construct ⇔ Parallel Executable Code

This section describes the parallel notation used to implement parallel BMF constructs, and the mechanism by which it is provided.

Compilation of the parallel BMF code to a parallel architecture hinges on two factors, its predictable efficiency and portability. Therefore, the choice of parallel language is an important for the Adl language project to be a success. For these reasons the C programming language together with MPI is used. Two versions of MPI were used, Sun's HPC cluster tools [18] and MPICH [19]. A

small amount of porting was required to migrate between the two versions, but not much.

The implementation consists of a suite of C functions with parameters specifying values and type information (which is embedded into the parallel BMF code). Each of these functions will correspond to a parallel (or otherwise) BMF construct (see figure 3). Using C/MPI, and providing the implementation in this way, means that the translator can generate code to call these functions when attempting to translate the parallel BMF program to a (semantically) equivalent C/MPI program. So long as the parallel library is comprehensively tested will provide relatively predictable efficiency. A summary of the target code and their corresponding parallel BMF equivalents is given in the table below. Section 4.2 describes the implementation of the target code in more detail.

Method	BMF construct
start()	-
finish()	-
initialise()	-
toGlobal()	-
toLocal()	-
nSplit()	P_split
hSplit()	P_split
sMerge()	P_reduce(conc)
hMerge()	P_reduce(conc)
operate()	-
seqReduce()	B_reduce
seqScan()	B_scan
seqZip()	B_zip
seqRepeat()	B_repeat
seqDistl()	B_distl
myReduce()	P_reduce
fmyReduce()	P_reduce
reduceConcat()	P_reduce(conc)
reduce()	P_reduce
myScan()	P_scan
scanConcat()	P_scan(conc)
scan()	P_scan
zip()	P_zip
repeat()	P_repeat
distl()	P_distl
pSelect()	P_project

Figure 4: Target methods and the corresponding parallel BMF

3.2.3 Target Architecture

Since this project is concerned with building a MPI based parallel implementation, an appropriate parallel computer is needed for both development and testing. A parallel computer called Hannah, located in the Computer Science department in Adelaide University, is used for development. Hannah is a 16-node cluster of single-processor machines using Sun HPC cluster tools [18]. The supercomputer used to gain the test results reported in chapter 5 is called Hydra, in addition some experiments were conducted on Orion. The characteristics of both Hydra and Orion will now be described.

Hydra is an IBM eServer 1350 Linux (Redhat) cluster with 128 nodes; it is managed by the South Australian Partnership for Advanced Computing (SAPAC) and is located at Adelaide University. Each of the 128 nodes uses dual 2.4GHz Intel Xenon processors. Each node has 2GB RAM and each processor has 512KB of L2 cache memory. Parallel MPI jobs are handled by MPICH, Miranet's implementation of MPI. MPICH is used exploit the low-latency and high-bandwidth of Hydra's Miranet interconnection network [19]. Hydra's theoretical peak performance is 1.2 Teraflops, or 1.2 trillion floating-point operations per second. In June 2003, Hydra recorded 682Gflops on the Linpack Benchmark [20], ranking 106th in the June 2003 top 500 list [21], and the fastest computer in Australia (when installed). This information was extracted from the SAPAC web site; see [22] for more information.

Orion is a Sun Technical Compute Farm composed of a cluster of 40 E420R workstations, connected by 100Mbit/s switched fast Ethernet and Myrinet. The E420R's have four 450MHz UltraSPARC II processors. Each node has 4GB of RAM and each processor has 4MB of L2 cache. The machine therefore has 160GB of RAM and 640MB of cache. The peak speed of the machine is 144Gflops.

Because the cost of computer hardware with reasonable clock rates is relatively low nowadays, and with interconnection network advances, it is relatively affordable to build a cluster-type parallel computer. Hydra and Orion are typical clusters built with commodity components and therefore should provide a suitable platforms for performance assessments.

4 Implementation

So far, description of Adl project and the compiler components already completed has been given. A trivial example of Adl code and its resultant parallel BMF code has also been presented. We have described that the mapping of parallel BMF constructs to a parallel machine requires two extra compiler components, a translation system and a parallel implementation. This chapter is devoted to describing how these new compiler components have been developed. Section 4.1 describes the translation system and section 4.2 describes the parallel target code implementation.

4.1 Translation System

The goal of the translation system is to map a textual representation of a parallel BMF program to a textual representation of an equivalent C/MPI program that utilises the target code (parallel implementation) described in section 4.2. The translator has been exclusively implemented in Miranda using recursive-rewrite rules, similar in structure to those described in [23]. Miranda was chosen because the recursive structure of a parallel BMF program is easily defined in this language using a few type definitions. Furthermore, Miranda's pattern matching makes writing recursive re-write rules simple.

The translator is broken into three components, the types, translator and auxiliary components. The types component is described in section 4.1.1. The main translation rules are defined in the translator component, and are described in section 4.1.2. The auxiliary component is described in appendix B.1.3.

4.1.1 The types Component

This section describes the important definitions contained in the Miranda literate script called transTypes.m. The role of this literate script is threefold. First, to provide the building blocks required in recognising BMF programs. Second, to define an abstract type (and associated operations) called state that stores information about the state of the translation at any given point. And third, to provide functions that handle conversions between BMF types (and operations) to semantically equivalent C/MPI ones.

4.1.1.1 Defining Parallel BMF

Before the translation mechanism can translate a parallel BMF program into an equivalent C/MPI program it necessarily needs to define the form of a parallel BMF program. The purpose of the b_exp (figure 3) type definition is to both define the recursive form of a parallel BMF program and alleviate parsing problems normally encountered in a compiler. The input of the translator is a

file containing a textual description of a parallel BMF program. The constructors defined by b_exp (e.g. P_split) provide a mapping from this text to an (exactly equivalent) internal representation of the program being processed.

There is a strong correspondence between the constructors of b_exp and the parallel BMF constructs described in [16]. The input code of the translator is assumed to be syntactically and semantically correct parallel BMF and therefore the translator performs no checking of this constraint. Furthermore, the input data of a parallel BMF program is assumed to be type annotated so that the corresponding C/MPI values can be easily generated.

4.1.1.2 Parallel BMF Input

The b_exp type definition (figure 3) showed that a parallel BMF program is formed using the constructor rule B_program b_exp inputType. The third component of this constructor rule is inputType, which corresponds to the input 'data' of the parallel BMF program being translated. The Miranda definition of inputType is given below:

```
tuple == (num,[inputType])
inputType ::=
          Int num | Real num | Bool bool |
          Tuple tuple |
          IntList (num,[num]) | RealList (num,[num]) |
          BoolList (num,[bool]) |
          NestList (num,[[inputType]]) |
          TupleList (num,[tuple])
```

Figure 5: The inputType definition

The role of the inputType[3] definition in the translator is twofold. First, to define the form in which input 'data' of a parallel BMF program must be defined. And second, to enforce that precise type information is embedded in the definition of that input 'data'. The latter is necessary because the target language (C/MPI) is not type polymorphic, and therefore sufficient type information is required in generating code. The BoolList (num,[bool]) constructor, for example, means that there is a list of length==num boolean values.

Although the translator could conceivably cope with less type annotation, the C/MPI code is easier to generate in this form. Furthermore, the translator presently only deals with a subset of the types specified by inputType. NestList, for example, has not been handled because the target code implementation does not presently handle nested lists. Future versions of the translator and target code implementation would need to handle all cases.

[3] inputType and tuple are mutually recursive definitions

4.1.1.3 Translator State

The most important structure any compiler contains is a symbol table, which keeps track of variable/function attributes. The translator (being a component of the Adl compiler) also maintains a mini symbol table structure, defined by this abstract type[4] called state. In parallel BMF, a function being evaluated at any point of execution can only refer to variables defined by its input value. This effectively means that the translators' symbol table will (usually) only need to record information about variables created at each construct translation point (and disregard any others). The Miranda definition of the abstract type state and an explanation of each of its components follow.

```
abstype
     state
with
     type_of :: state -> types
     name_of :: state -> names
     size_of :: state -> sizes
     parallel_of :: state -> parallel
     stmts_of :: state -> statements
     create :: types -> names -> sizes ->
          parallel -> statements-> state
     empty :: state
state == (types,names,sizes,parallel,statements)
```

Figure 6: The abstype state

- `types`: This component is an array of string elements describing the C/MPI types corresponding to the input/output values of the parallel BMF construct currently being translated. For example, if a function produces a distributed vector (of tuples) this component might contain ["DataInfo","alltup"], where alltup is the name of the previously generated tuple type. Because we are currently only dealing with a subset of types this mechanism is sufficiently general for now.
- `names`: This component is an array of string elements describing the C/MPI variable identifiers assigned to the input/output values of the parallel BMF construct currently being translated. For example, the P_reduce function might assign this component to ["reduceResult"].
- `sizes`: This component is an array of string elements describing the number of elements contained in the vectors specified by the input/output values of the parallel BMF construct currently being translated. If the input/output values are singular values (e.g. integers) this component contains ["1"]. For example, the P_repeat function (which outputs a distributed list of length==copies) might assign this component to ["copies"]. The strings in this component may be either constant integers or references to variables containing constant integers.

[4] Abstract types encapsulate the details of the implementation

- `parallel`: This component is a boolean value that specifies if the target code being produced is currently being run in parallel. This basically tracks whether or not the P_split function has been encountered.
- `statements`: This component of the translators state is the most important. It is a string containing the C/MPI statements representing the program currently being translated. It is also the output of the translator. For example, the B_con (B_real 33.5) function would set this component to "double temp = 33.5;\n".

The state definition also provides methods for accessing components of a state ($_of functions), state creation from individual components (create function) and for getting an initial empty state (empty function). The next section will give a more in-depth understanding of how the translators' state is updated throughout the translation of a parallel BMF program.

4.1.2 The translator Component

This section describes the translation rules defined in the Miranda literate script called translator.m. The translator literate script provides two main function definitions. One, a function called trans (discussed in section 4.1.2.1) that both initialises and finishes the translation of an input BMF program. And two, a function called gen (discussed in section 4.1.2.2) which performs the core of the translation an input BMF program in recursive re-write style.

4.1.2.1 The Translation Function

The goal of the translation system is to map a textual representation of a parallel BMF program to a textual representation of an equivalent C/MPI program that utilises the methods described in section 4.2.2. The role of the trans function is to both start and finish the translation of a program. Accordingly, the trans function will be invoked from the Miranda interpreter as follows:

```
>CMPIProgram = trans BMFProgram
```

It therefore seems obvious for the trans function to perform the following functions:

1. Concatenate C/MPI #include information to the output
2. Concatenate the start() method to the output
3. Concatenate the input data initialisation code to the output
4. Set answer = gen input_program
5. Concatenate the statement string of the answer to the output
6. Concatenate the finish() method to the output
7. Concatenate any unclosed brackets to the output
8. Return the resultant output string

Step 4 above makes reference to a function called gen, this performs a recursive re-write style translation of the input program and returns the final state of the translation. The next section describes the many rules required to translate a parallel BMF program into an equivalent C/MPI program.

4.1.2.2 The Code Generation Function

The goal of the gen function is to define a translation rule for every parallel BMF construct (described by b_exp). The Miranda type definition for the input and output of the gen function is as follows:

```
gen :: b_exp -> input_state -> output_state
```

Any rule of the gen function will therefore input a parallel BMF expression and a translation state and output a translation state. This means that each rule of the gen function will generate code based on the current parallel BMF expression and the input translation state. The output of each rule will be a translation state describing the code that was generated, since this may be useful in translating the next parallel BMF expression.

A few of the interesting rules defined within the translator are described below, refer to appendix B for a complete description. The first rule described below is actually physically the last rule defined by the gen function, it is described here first because it gives a good basis for understanding how the translation of a b_exp proceeds. Each rule below references $CODE_GENERATED, this refers to the string of C/MPI code under the *Code Generated* heading of that rule.

`gen (B_comp e1 e2) input`
-State Input-
Any state appropriate for input to e1 and e2.

-Code Generated-
```
e2_statements;
e1_statements;
```
Where:

- *e2_statements*: the code generated by the function e2.
- *e1_statements*: the code generated by the function e1.

-State Output-
```
typeof = type_of y
name = name_of y
size = size_of y
parallel = or [parallel_of x, parallel_of y]
stmts = $CODE_GENERATED
```

Code is generated such that the function e2 is executed *before* the function e1. This rule is the most general code generation rule; it is the starting point for most parallel BMF programs.

`gen (P_map function) input`

-State Input-
A state describing the distributed vector to be mapped over.

-Code Generated-
```
function_statements;
distrib.mydata = mappedList;
```
Where:

- *function_statements:* code generated by calling gen the function.
- *distrib:* the name of the input distributed vector.
- *mappedList:* the name of the non-distributed vector output from calling gen on the function.

-State Output-
```
type_of = ["DataInfo",hd (tl (type_of mapped))]
name = name_of input
size = size_of mapped
parallel = True
statements = $CODE_GENERATED
```

Note that mapped is the resulting translation state after calling gen function input. The result of the parallel map is a distributed vector with the same name as the input distributed vector. This is the most general parallel map; mapping a B_reduce function is a special case (due to translation problems) and is not described here. The gen rule for sequential map is given in the appendix.

`gen (P_reduce op) input`

-State Input-
A state describing both a distributed input vector and a locally computed reduce value.

-Code Generated-
```
type result= *(type*)reduce(&list,local,TYPE,operation);
```
Where:

- *type*: the output type corresponding to this reduce operation.
- *list*: the name of the distributed input vector.
- *local*: the name of the locally computed reduce input value.
- *TYPE*: the 'type descriptor' corresponding to this reduce operation. E.g. Integer or IntegerTuple.
- *operation*: the C/MPI reduce operation descriptor corresponding to the BMF operation specified by op. E.g. (b_op B_plus) is SUM.

-State Output-
A state describing the non-distributed value produced by the reduce function.

```
typeof = type_of input
name = [“reduceResult”]
size = [“1”]
parallel = True
stmts = $CODE_GENERATED
```

The rule for reduce with concatenate is given in the appendix.

gen (P_scan op) input

-State Input-
A state describing the distributed vector used as input to this scan function.

-Code Generated-
```
DataInfo sresult = scan(&list,TYPE,operation);
```
Where:

- *list*: the name of the distributed input vector (of type DataInfo).
- *TYPE*: the ‘type descriptor’ corresponding to this scan operation. E.g. Integer or IntegerTuple.
- *operation*: the C/MPI scan operation descriptor corresponding to the BMF operation specified by op. E.g. (b_op B_plus) ⇔ SUM.

-State Output-
A state describing the distributed vector produced by this scan function:

```
typeof = [“DataInfo”,listtyp] ||[distribtyp,elemtyp]
name = [“sresult”]
size = size_of input
parallel = True
stmts = $CODE_GENERATED
```

The rule for scan with concatenate is given in the appendix.

gen (B_alltup list) input

-State Input-
Any state appropriate for input to all components of the B_alltup list.

-Code Generated-
```
statements_1;
statements_2;
      .
      .
      .
```

```
statements_26;
typedef struct{
     type1 *a;
     type2 *b;
          .
          .
          .
     type26 *z;
}alltup;
alltup mytup;
mytup.a = name1;
mytup.b = name2;
     .
     .
     .
mytup.z = name26;
```

Where:

- *statements_i*: code generated from calling the i'th component of the B_alltup.
- *alltup*: the corresponding C typedef to the B_alltup.
    - $type_i$: the output type of the i'th component of the B_alltup
    - *a,b,…*,z: the maximum number of components of a B_alltup. Only 26 components are allowed for ease of naming.
- *mytup*: a variable of type *alltup* to hold the results output from corresponding components of the B_alltup.
    - $name_i$: the name of the output value of the i'th component of the B_alltup. See *alltup* for B_alltup restrictions.

-State Output-

A state describing a tuple of results obtained from code generation of the components of B_alltup.

```
typeof = ["alltup"]++typs ||typs=all tup component types
name = ["mytup"]
size = concat [size_of e | e <- results]
parallel = or para
stmts = $CODE_GENERATED
```

A translation state is generated from calling gen on each component of the B_alltup. This information needs to be encoded in the output state of this function so that it may be later referenced by the B_addr function (rule). Accordingly, `typs` refers to a list of all the output types obtained from calling gen on all components of the B_alltup. The `size` component is a list of all output sizes obtained from calling gen on all components to the B_alltup. The `para` refers to a list of booleans obtained from calling gen on all components of the B_alltup.

`gen (B_addr (B_num size) (B_num offset)) input`

-State Input-
A state describing B_alltup, which is being referenced by this B_addr operation.

-Code Generated-
none

-State Output-
A state describing the value referenced by this B_addr function:

```
typeof = [(arrayIndex (offset-1) tuptypes)]
name = [tupname++”.”++[(arrayIndex (offset-1) alphabet]]
size = [(arrayIndex (offset-1) tupsizes)]
parallel = parallel_of input
stmts = $CODE_GENERATED
```

This B_addr function inputs a state describing a B_alltup (previously described) and outputs a state describing the component of the B_alltup being referenced (offset). Therefore, the typeof component is assigned to tuptypes[offset-1] which is extracted from the information recorded by the input B_alltup. The name component is assigned to tupname.X, where tupname is also extracted from the input B_alltup and X is variable name at (offset-1) in the alphabet (“abcd…xyz”). The size component is derived similarly to the typeof component.

`gen (B_if pred cons alt) input`

-State Input-
A state appropriate for input to the pred, cons and alt components of the B_if function.

-Code Generated-

```
pred_statements;
type ifresult;
if (predname){
     consq_statements;
     ifresult = consqname;
}else{
     alter_statements;
     ifresult = altername;
}
```

Where:

- *pred_statements*: any code generated to acquire the boolean predicate.
- *type*: the type of the output values of consq and alter (both the same).
- *predname*: the name of the output variable (of type boolean) generated by the pred component of the B_if function.

- *consq_statements*: code generated by the consq component of the B_if function (corresponds to if-part statements).
- *alter_statements*: code generated by the alter component of the B_if function (corresponds to else-part statements).

-State Output-

```
typeof = type_of c          ||c = gen cons input
name = ["ifresult"]         ||result of this if statement
size = size_of c
parallel = parallel_of c
stmts = $CODE_GENERATED
```

4.2 Target Code Implementation

The goal of the target code implementation component of the compiler is to provide a library of parallel methods, using C and MPI, which implement parallel (or otherwise) BMF constructs. The implementation is provided in this way so that each 'method' is analogous to an 'instruction' to the translation system when generating target code. The library encapsulates all parallel complexities such as communication and deadlock making it simple for a client program to use.

The parallel implementation has two components. The Parallel Types component provides definitions required in keeping track of distributed (or otherwise) data, which is described in section 4.2.1. The main component is the Construct Library that provides sequential and parallel C/MPI functions that operate on both non-distributed and distributed data, which is described in section 4.2.2. Both section 4.2.1 and 4.2.2 do not attempt to provide complete descriptions of all definitions/functions that have been implemented. The reader can to refer to Appendix B and C for complete detail on the implementation.

4.2.1 Parallel Types

This section describes the various types (and operations on them) required to keep track of parallelism, distributed data and non-distributed data.

4.2.1.1 Parallelsim

When a user runs an MPI program they specify a command-line argument telling the MPI runtime system how many processors should be used. An MPI program typically records this information in two ways, an integer specifying the number of processors and an integer specifying a unique processor ID. The C declaration of these variables, used by parallel types, follows:

```
int rank;       /*processor ID*/
```

```
int np;         /*number of processors*/
```

4.2.1.2 Non-Distributed Data

A requirement of the parallel types component is to define a structure for describing the equivalent of a parallel BMF list. The information needed is the length, element type size (in bytes) and the starting address of the array. The C declaration for a non-distributed parallel BMF list defined by parallel types follows:

```
typedef struct{
     void *list;     /*starting address*/
     int tSize;      /*element size*/
     int length;     /*array length*/
}ListInfo;
```

The parallel types component also defines a function called addrOf(), which allows indexing elements of an array using the byte size of elements in that array. This method is required since lists are represented as void* types.

Operations such as zip and distl require elements which are ‘paired’ together; a tuple of length two. The associated list of pairs (a ListInfo) will store the type size (in bytes) of the combined pair elements, therefore a pair need only contain the memory address where the values can be found. The C declaration used by parallel types to represent a pair is:

```
typedef struct{
     void *pi1;      /*fst element address*/
     void *pi2;      /*snd element address*/
}pair;
```

Explicitly declaring a tuple of length two in the implementation is an optimisation and is used primarily for convenience.

4.2.1.3 Distributed Data

Sequential C programs using arrays typically need only record the length and starting address of arrays to completely describe them. A parallel C/MPI program usually needs to have some (implicit or explicit) representation of a distributed data-structure in order to coordinate computation between multiple processors operating on that data-structure. The C declaration, used by parallel types, to completely describe a distributed array/vector follows:

```
typedef struct{
     ListInfo global;
     ListInfo mydata;
     int pieces;
```

```
        int *sizes;
        int each;
        int master;
        int slave1;
        int slave2;
        int s1Size;
        int s2Size;
    }DataInfo;
```

The following gives a brief description of the attributes this structure and which processors make use of them. The parallel types component also defines get/set functions for accessing/assigning attributes of a DataInfo structure (not described since they are trivial).

- `global`: The non-distributed global data. Only the processor with rank==0 maintains a copy of this attribute.
- `mydata`: This processors portion on the distributed data.
- `pieces`: The number processors the global data is distributed across.
- `sizes`: An array of sizes, where sizes[i]==processor(i).mydata.length.
- `each`: All processors data size, except rank==np-1.
- `master`: This processors parent node in the binary-tree processor hierarchy.
- `slave1`: This processors left-leaf node in the binary-tree processor hierarchy.
- `slave2`: This processors right-leaf node in the binary-tree processor hierarchy.
- `s1Size`: Processor==slave1 data size, plus sizes of all processors below slave1 in the binary-tree processor hierarchy.
- `s2Size`: Processor==slave2 data size, plus sizes of all processors below slave1 in the binary-tree processor hierarchy.

4.2.2 Construct Library

The previous section described the definitions required to keep track of parallelism, distributed and non-distributed data. This section describes a few of the interesting implementations of the target C/MPI library of methods. The table below gives a summary of the methods that have been implemented, their type, which parallel BMF construct they implement and if they are described in the following sections. A complete description of all methods implemented (including ones not described in this section) can be found in appendix C.1.

Method	Type	Implements	Described
start()	M	-	NO
finish()	M	-	NO
initialise()	M	-	NO
toGlobal()	DDI	-	NO
toLocal()	DDI	-	NO
nSplit()	DD	P_split	NO
hSplit()	DD	P_split	YES
sMerge()	DR	P_reduce(conc)	NO
operate()	M	-	NO
seqReduce()	SC	B_reduce	NO
seqScan()	SC	B_scan	NO
seqZip()	SC	B_zip	YES
seqRepeat()	SC	B_repeat	NO
seqDistl()	SC	B_distl	NO
myReduce()	PR	P_reduce	NO
fmyReduce()	PR	P_reduce	NO
reduceConcat()	PR	P_reduce(conc)	YES
reduce()	PR	P_reduce	NO
myScan()	PS	P_scan	YES
scanConcat()	PS	P_scan(conc)	NO
scan()	PS	P_scan	NO
zip()	DDR	P_zip	NO
repeat()	DDR	P_repeat	YES
distl()	DDR	P_distl	NO
pSelect()	DDR	P_project	YES

Table 2: All sequential and parallel methods implemented

Type

M Miscellaneous. Used by parallel Implementation
DDI Distributed Data Indexing
DD Data Distribution
DR Data Re-coalescing
SC Sequential Computation
PR Parallel Reduction
PS Parallel Scan
DDR Distributed Data Re-structuring

Parameters
(1) *info*: a pointer to a structure to be used in splitting the data among all available processors. The structure will be updated with some splitting information.
Returns
void

This method performs a hierarchical distribution of the currently non-distributed input vector described by the info parameter. The communication is conducted in a binary tree with processors at the nodes. Consider the following example:

Vector = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16];
Processors = 8;

Then the following communication tree would be constructed:

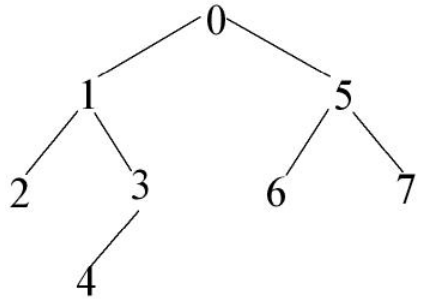


Figure 7: An 8 processor split hierarchy

This means that the processor with rank==0 sends portions of the non-distributed structure described by the input parameter (info) to processors 1 and 5 (its child nodes). From this point the processors located in the left and right hand sub-trees proceed with communication *parallel*. I.e. processors 1 and 5 become masters of their respective sub-trees, independently continuing the split operation[5]. Each processor records its master and slaves (if any) for later use by reduction operations[6]. Barrier synchronisation is performed before returning control to the calling program.

This method of splitting is superior, in terms of performance, to the card-dealing style performed by the nSplit() operation. Note that parallelism increases as the algorithm progresses.

[5] This algorithm is performed by helper function called splitData(), which takes an integer indicating the number of processors to be involved in the split operation, in the case of hierarchical split all processors are utilised.
[6] The recording of master and slaves at each processor (node) means the *implicit* processor hierarchy (figure 7) is recorded. This means reduction operations can use this tree to evaluate sub-tree reduction results (in parallel) from the bottom up.

4.2.2.2 Sequential Zip (γ)

Definition

$\gamma([x_0,x_1,\ldots,x_n],[y_0,y_1,\ldots y_k]) = [(x_0,y_0),(x_1,y_1),\ldots]$

Method Signature

```
ListInfo seqZip(ListInfo *left, ListInfo *right);
```

Parameters

(1) *left*: a pointer to a structure describing the non-distributed vector to be used in computing the sequential zip.
(2) *right*: a pointer to a structure describing the non-distributed vector to be used in computing the sequential zip.

Returns

void

This method inputs two non-distributed vectors of arbitrary values (both vectors of the same type). Corresponding elements from the two vectors are combined into a single vector of pairs, and the resulting non-distributed vector is returned (encapsulated in a ListInfo structure). If the input vectors have different lengths, the longer ones extra values are ignored.

A parallel equivalent of seqZip, called zip(), has also been implemented. The parallel algorithm is quite similar, however the method operates distributed data (see appendix C.1).

4.2.2.3 Parallel Reduce (/)

Definition

$\oplus/[x_0,x_1,x_2,.....,x_{n-1}] = x_0 \oplus x_1 \oplus x_2 .... \oplus x_{n-1}$

Method Signature

```
int myReduceI(DataInfo *info, int operation, int local);
```
[7]

Parameters

(1) *info*: a pointer to a structure describing the distributed vector relating to this reduction operation.
(2) *operation*: an integer representing the reduce operation currently being performed; allowable operations are: sum, product, min and max.
(3) *local*: an integer representing this processors already-computed local result in relation to the reduce operation being performed.

Returns

An integer representing the result of this reduction operation.

[7] This method only operates on distributed vectors of integer values, another method called myReduceD() operates on distributed vectors of double values (see appendix C.1).

This method provides a custom implementation of the MPI_Reduce operation (see reduce() in appendix C.1). It inputs the type of operation being performed and the already computed local result and returns the result of the reduction to the processor with rank==0. The algorithm used hinges on the split hierarchy information recorded during the previously executed hSplit() method. Repeating the diagram from hSplit():

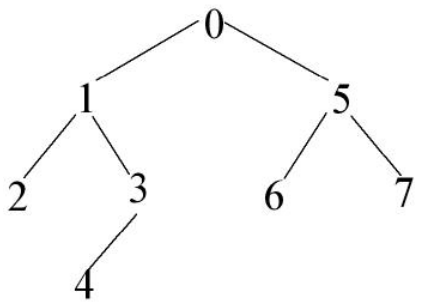


Figure 8: An 8 processor split hierarchy (again)

Given this split hierarchy, the algorithm is obvious. All processors (at each node) asynchronously receive a result from their child processors. Upon receiving a result, processors call the operateI() method to obtain an updated local result, and again upon receiving the second result (if any). All processors (except the one with rank==0) then send the updated result to their master processor (node above). When all communication is completed the result will reside on the node (processor) with rank==0 (at the top of the tree). This algorithm has significant performance advantages because much of the communication (and computation) is carried out in *parallel*. Note that the parallelism decreases as the algorithm progresses.

A custom implementation of MPI_Reduce() has been provided for two reasons: to compare its efficiency with the MPI native method and because MPI_Reduce() does *not* provide a concatenate binary operation[8]. The concatenate operation is required because the parallel BMF construct P_reduce (which this function is implementing) allows an operator called B_conc. MPI allows custom operator/type definitions to cater for operations other than provided but it seems to enforce a restriction with makes a concatenate operation difficult define. Consider the following prototype definitions used to provide a new MPI_Op:

```
typedef void MPI_User_function(void *invec,
                               void *outvec,
                               int *len,
                               MPI_Datatype *datatype);
```

[8] A method called reduceConcat() actually implements reduce with the concatenate operation, see appendix C.1.

```
int MPI_Op_Create(MPI_User_function *function,
                  int commute,
                  MPI_Op *op);
```

These prototype definitions seem to enforce that both the invec and outvec must occupy the same amount of memory. Defining a concatenate operation using MPI_User_function is not possible with this restriction, and therefore this implementation provides a custom function to perform it.

4.2.2.4 Parallel Scan (//)

Definition

$\oplus//[x_0,x_1,x_2,.....,x_{n-1}] = [x_0,x_0 \oplus x_1,....,x_0 \oplus x_1 \oplus ... \oplus x_{n-1}]$

Method Signature

```
DataInfo myScanI(DataInfo *info, int operation);
```

Parameters

(1) *info*: a pointer to a structure describing the distributed vector relating to this scan operation.
(2) *op*: a integer representing the scan operation currently being performed; allowable operations are: sum, product, min and max.

Returns

A pointer to a structure describing the distributed vector produced by this scan operation.

This method provides a custom implementation of the MPI_Scan operation (see scan() appendix C.1). It inputs a scan operation type and a distributed vector describing a list to perform the operation on. The processor with rank==0 starts by performing the seqScanI() method on its portion of the distributed data. Rank==0 then starts the communication by sending its total accumulated value to the processor with rank==1. All other processors synchronously receive a accumulated value from their neighbor (processor rank-1), execute the seqScanI() method with initial value==neighbors value, and synchronously send their total accumulated value (except rank==np-1) to the next processor (rank+1). When all communication is finished scan results of the input operation lie on each processor node, which are updated with the input distributed vector. Barrier synchronisation is performed before returning the results (and control) to the calling program.

The above algorithm will work ok for small numbers of processors; a second (well known) algorithm has also been implemented. Figure 8 below shows the pattern of communication required for the simple algorithm, and figure 9 shows the pattern for the second.

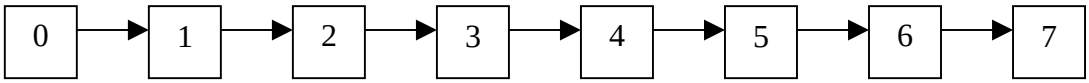


Figure 9: Parallel Scan 1

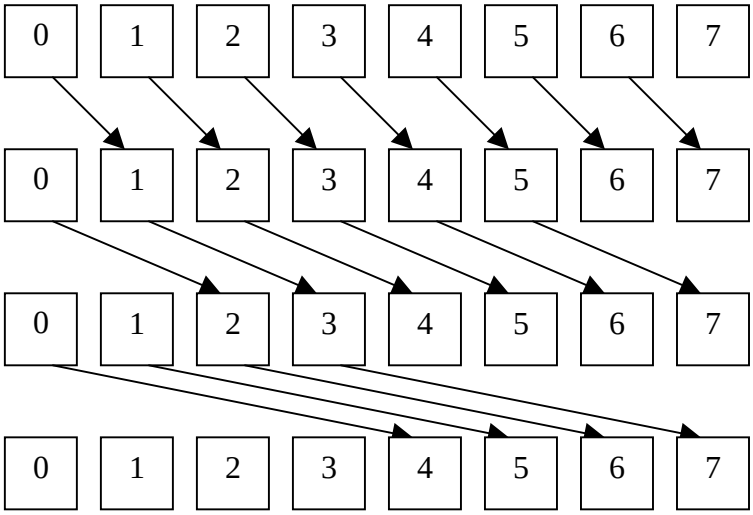


Figure 10: Parallel Scan 2

The first algorithm (figure 8) requires np-1 distinct messages, 7 messages if np==8. The second algorithm (figure 9) has ceil(log(np)/log(2)) steps. Each processor sends a message to the processor with rank==myrank+2^(step-1). However, all communication in each step is carried out in *parallel* which means there are really only 3 messages when np==8. The general case should therefore see the second algorithm outperform the first.

4.2.2.5 Parallel Repeat

Definition
repeat(a,p) = [a,a,a,a,…,a] = a vector with p copies of a

Method Signature
```
DataInfo repeat(void *value, int tSize, int copies);
```

Parameters
(1) *value*: a pointer to a non-distributed value.
(2) *tSize*: an integer representing the size (in bytes) of the value parameter.
(3) *copies*: an integer specifying the number of times value is to be repeated.
Returns
A pointer to a structure describing the resultant distributed vector after the application of this repeat operation.

This method takes a non-distributed value==value and an integer p==copies and forms a (new) distributed vector over p processors where each processors portion of the distributed data is a copy of the value. This method is useful for easily creating distributed vectors where all elements are the same.

The processor with rank==0 first creates a global vector of length p==copies where each element is a copy of the non-distributed value. All processors then execute the splitData() method using only p==copies processors, resulting in a (new) distributed vector. All processors then participate in barrier synchronisation before returning control to the calling program. Note: the splitData() method is the same method called by hSplit() to perform a hierarchical split operation. However, hSplit() utilises all available processors instead of just p==copies.

The repeat() method uses the splitData() method to get the resultant distributed vector for two reasons. The first is obviously because the splitData() method is already implemented, resulting in code reuse. The second is because a calling program may later call the reduceConcat() method to re-coalesce its data. The reduceConcat() method makes heavy use of split hierarchy information recorded during the hSplit() operation, therefore if repeat() does not use splitData() for communicating the input value errors will occur when calling reduceConcat(). The disadvantage to this approach is that if the object to be repeated is very large the processor with rank==0 will perform lots of memory copying.

4.2.2.6 Parallel Select

Definition
select (sv,[i_0,i_1,…,i_{p-1}] = [sv!i_0,sv!i_1,…,sv!i_{p-1}]

Method Signature
```
DataInfo pSelect(DataInfo *sv, int indexes[], int size);
```

Parameters
(1) *indexes*: a non-distributed vector of integers describing how the distributed vector sv is to be re-arranged.
(2) *size*: the size of the indexes vector.
(3) *sv*: a pointer to a structure describing the distributed vector to be re-arranged.
Returns
A pointer to a structure describing the resultant distributed vector after the application of this select operation.

This method takes a distributed source vector==sv and a non-distributed index vector==indexes and produces a distributed vector==sv re-arranged by the non-distributed index vector. Consider the following example:

processors = 4;
dist = [[1, 2], [3, 4], [5, 6], [7, 8]]; //nesting indicates distribution
ind = [1, 0, 3, 2];

Before the select operation, the machine state will contain:

processor 0: [1, 2]
processor 1: [3, 4]
processor 2: [5, 6]
processor 3: [7, 8]

After the execution of the select operation with distributed vector==dist and indexes==ind, the machine state will contain:

processor 0: [3, 4]
processor 1: [1, 2]
processor 2: [7, 8]
processor 3: [5, 6]

So the index vector is a specification on how the distributed data specified by the vector==sv should be re-arranged on the machine.

This parallel select operation requires a subset of the functionality of non-uniform many-to-many personalised communication, and therefore a general communication pattern is required to avoid deadlock problems. The processor with rank==0 first broadcasts the index vector to all other processors, giving them a local copy. Each processor then executes the following pseudo code:

```
for proc in indexes loop
     if (rank==proc) and (rank!=indexes[proc]) then
          syncRecv(indexes[proc],recvbuff);
     else if (rank==indexes[proc]) and (rank!=proc) then
          syncSend(proc,mydata);
     end if;
end loop;
```

Each processor iterates through the indexes array and at index==rank that processor will receive data from the processor with rank==indexes[proc]. If this processors rank==indexes[proc] then this processor must send its data to the processor with rank==proc. Furthermore, processors make sure they do not try to send or receive data to/from themselves. After executing this loop all processors participate in barrier synchronisation before returning control to the calling program.

The performance of this algorithm is highly dependant on the re-distribution index array. The worst case is when a single processor must send its data to every other processor. The best case is when all processors send their data to their neighboring processors, during a shift operation. Parallel select is useful for distributed data re-arrangements.

5 Results

The previous chapter covered the implementation of the final two stages of the Adl compiler, the translator and its parallel C/MPI target code implementation. This chapter will present a number of example parallel BMF programs, their resultant C/MPI code (after translation) and a range of parallel speedup/efficiency tests to demonstrate performance characteristics of code produced by the translator. Absolute performance figures of each of the parallel BMF constructs will also be presented. Finally, the difficulties and findings that arose throughout the development of this project will be discussed.

5.1 Example Programs

5.1.1 SRZ; Split Repeat Zip

Figure 11 shows the parallel BMF program under consideration (remember this program has already been optimised and parallelised). This program inputs a vector of tuples, splits the data among all available processors, repeats the input vector over all processors, and finally zips the two resultant distributed vectors elements together.

```
B_program
   (B_comp
      (P_zip
       (B_alltup[B_addr (B_num 2) (B_num 1),
            (P_repeat(B_alltup[(B_addr(B_num 2)(B_num 2)),
           (B_comp (B_op (B_length))
                    (B_addr (B_num 2) (B_num 1)))]))
                ]))
   (B_alltup[(P_split (B_num 4 (B_num 0)),B_id]))
(TupleList (8,[((3,[Int 2,Bool False,Real 2.3])),
                ((3,[Int 2,Bool False,Real 2.3])),
                ((3,[Int 2,Bool False,Real 2.3])),
                ((3,[Int 2,Bool False,Real 2.3])),
                ((3,[Int 2,Bool False,Real 2.3])),
                ((3,[Int 2,Bool False,Real 2.3])),
                ((3,[Int 2,Bool False,Real 2.3]))]))
```

Figure 11: SRZ; Split, Repeat then Zip

When this program is fed into the translation system the resultant C/MPI code will be generated (with formatting changes):

```
#include "ParallelConstructs.h"
int main(int argc, char **argv){
  int dummy = start(argc,argv);
  {
    typedef struct{
      int a;
      boolean b;
```

```
      double c;
    }ttype;
    ttype input[8] = {{2,false,2.3},{2,false,2.3},
                      {2,false,2.3},{2,false,2.3},
                      {2,false,2.3},{2,false,2.3},
                      {2,false,2.3},{2,false,2.3}};
    DataInfo distrib = initialise(&input[0],sizeof(ttype),8);
    hSplit(&distrib);
    {
      typedef struct {
         DataInfo *a;
         ttype *b;
      }alltup;
      alltup mytup;
      mytup.a = &(distrib);
      mytup.b = &(input[0]);
      {
         int len = (*(mytup.a)).mydata.length;
        DataInfo repeated =
            repeat(mytup.b,8*sizeof(ttype),len);
         DataInfo zipped = zip(mytup.a,&(repeated));
         finish();
      }
    }
  }
}
```

Figure 12: Code generated from SRZ

5.1.2 SumDistl

Figure 13 shows a parallel BMF program to compute the sum of a list of real numbers and pairs the result with all elements of the original distributed list. The program starts by splitting the input list among available processors. Next, it creates a tuple with the first element the distributed list and the second element the sum of the elements in the distributed list. The result of the summation is then paired with elements of the distributed list (using P_distl) and the resultant distributed vector merged to get the final result.

```
B_program
   (B_comp
      (B_comp
         (P_reduce (B_op (B_conc)))
         (B_comp
            (P_distl)
            (B_alltup[B_addr(B_num 2)(B_num 2),
                      B_addr(B_num 2)(B_num 1)])))
      (B_comp
         (B_alltup[B_id,
                   (B_comp
                      (P_reduce (B_op (B_plus)))
                      (P_map (B_reduce (B_op (B_plus))
                                       (B_con (B_int 0))))))]))
```

```
          (P_split (B_num 8) (B_num 0))))
   (RealList (8,[7.0,6.0,5.0,4.0,5.0,6.0,7.0,8.0]))
```

Figure 13: SumDistl

When this program is fed into the translation system the resultant C/MPI code will be generated (with formatting changes):

```
#include "ParallelConstructs.h"
int main(int argc, char **argv) {
  int dummy = start(argc,argv);
  double input[8] = {7.0,6.0,5.0,4.0,3.0,2.0,1.0,0.0};
  DataInfo distrib = initialise(&input[0],sizeof(double),8);
  hSplit(&distrib);
  {
    double brlocal = seqReduceD(&(distrib.mydata),SUM);
    {
       double reduceResult =
            *((double*)reduce(&distrib,&brlocal,Double,SUM));
      typedef struct {
        DataInfo *a;
        double *b;
      } alltup;
      alltup mytup;
      mytup.a = &(distrib);
      mytup.b = &(reduceResult);
      {
        DataInfo dist = distl(mytup.b,sizeof(double),mytup.a);
        {
          ListInfo merged = reduceConcat(&distrib);
          finish();
        }
      }
    }
  }
}
```

Figure 14: Code generated from SumDistl

5.1.3 Remote

This section works though a non-trivial example for a program called remote.Adl. This program takes a one-dimensional list of points and, for each point makes a vector of the distance from itself to each of the other points (including itself). The resultant distances are summed to get a measure of average remoteness. Figure 15 shows the Adl source code.

```
main a: vof int :=
  let
      f x :=
        let
            add(x ,y) := x+y;
            abs x := if x<0 then -x else x endif;
```

```
            dist y := abs(x-y)
        in
            reduce(add, 0, map(dist, a))
         endlet
  in
     map(f, a)
  endlet
?
```

Figure 15: remote.Adl

Earlier stages of the Adl compiler translate/optimise this program to sequential BMF and subsequently parallelise it, to produce the following parallel BMF code.

```
B_program
 (B_comp
   (P_reduce (B_op (B_conc)))
   (B_comp
     (B_comp
       (B_comp
         (P_map
           (B_map(B_comp
                  (B_comp
                    (B_reduce (B_op(B_plus)) (B_con (B_int 0)))
                    (B_map(B_if(B_comp(B_op(B_lt))
                           (B_alltup[
                            B_op(B_minus),
                            B_con (B_int 0)]))
                          (B_comp(B_op(B_uminus))
                            (B_op(B_minus)))
                          (B_op(B_minus)))))
                (B_op(B_distl)))))
        (P_map (B_zip(B_alltup[
                  B_addr(B_num 2)(B_num 1),
                  B_addr(B_num 2)(B_num 2)])))))
      (P_map (B_alltup[
           B_addr(B_num 2)(B_num 1),
           B_comp(B_op(B_repeat))
                 (B_alltup[
                   B_addr(B_num 2)(B_num 2),
                   B_comp(B_op(B_length))
                         (B_addr(B_num 2)(B_num 1))])])))))
    (B_comp
          (B_comp (P_zip (B_alltup[B_addr(B_num 2)(B_num 1),
                                   B_addr(B_num 2)(B_num 2)]))
              (B_alltup[
                B_addr(B_num 2)(B_num 1),
                B_comp(P_repeat (B_num 0))
                       (B_alltup[
                          B_addr(B_num 2)(B_num 2),
                          B_comp(B_op(B_length))
                                (B_addr(B_num 2)(B_num 1))])]))
          (B_alltup[P_split (B_num 8) (B_num 0), B_id])))))
  (IntList (16,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]))
```

Figure 16: Parallel BMF of remote.Adl

The current version of the translator can handle individual parts of this program, but due to some small problems is not able to entirely translate this program. Since this program utilises many of the parallel constructs implemented, it is interesting enough to show a *hand coded* direct C/MPI program form. The program was written to follow the exact algorithm specified by the parallel BMF form of remote.Adl, and as such should have similar performance characteristics. The program listing is quite long and can be found in appendix D.1.

5.2 Efficiency Tests

The previous section presented three example parallel BMF programs and their resultant C/MPI code after translation. This section is divided into two parts. First, parallel speedup and efficiency results on both Hydra and Orion for the three C/MPI programs from the previous section. And second, absolute performance figures of all parallel constructs implemented.

5.2.1 Test Program Performance

This section presents performance figures of C/MPI code for the SRZ, SumDistl and Remote example programs. All speedup and efficiency results reported for Hydra were gained by taking wall-clock measurements from the RedHat version of time, and similarly on Orion (which runs Solaris). In-code measurements were not used because they do not accurately reflect the amount of time a user waits for his/her program to execute. Speedup was measured relative to single-processor parallel performance.

The first two programs (SRZ and SumDistl) are necessarily abstract to demonstrate a particular aspect of the implementation. The Remote example is less trivial and therefore contains a more comprehensive analysis.

5.2.1.1 SRZ; Split Repeat Zip

The aim of this test is to demonstrate that even the data-distribution and re-arrangement operations can obtain speedup. The C/MPI code produced by the translator for SRZ (figure 12) was run on Orion for vector lengths 6000, 7000 and 8000 with 1,2,4 and 8 processors. The resulting speedup graph is given in below.

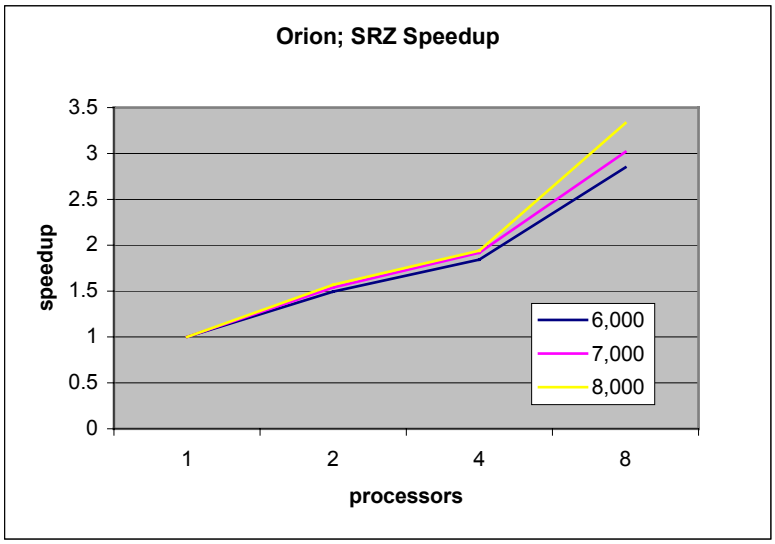


Figure 17: Orion; SRZ speedup

The graph shows the speedup obtained by partitioning the repeat and zip operations over multiple processors. The memory copying performed in parallel by repeat and zip is the source of the speedup in this example. Parallel efficiency is similar for most data-sizes, ranging between 35% for 8 (length=6000) processors to 78% for 2 processors (length=8000). The general trend seems to indicate that increasing the data-size slowly increases speedup and efficiency.

5.2.1.2 SumDistl

The aim of this test is to show how variation in the communication to computation ratio can affect speedup. The C/MPI code produced by the translator for SumDistl (figure 14) was run on Orion for vector lengths 8,9 and 10 million with 1,2,4 and 8 processors. The resulting speedup graph is given below.

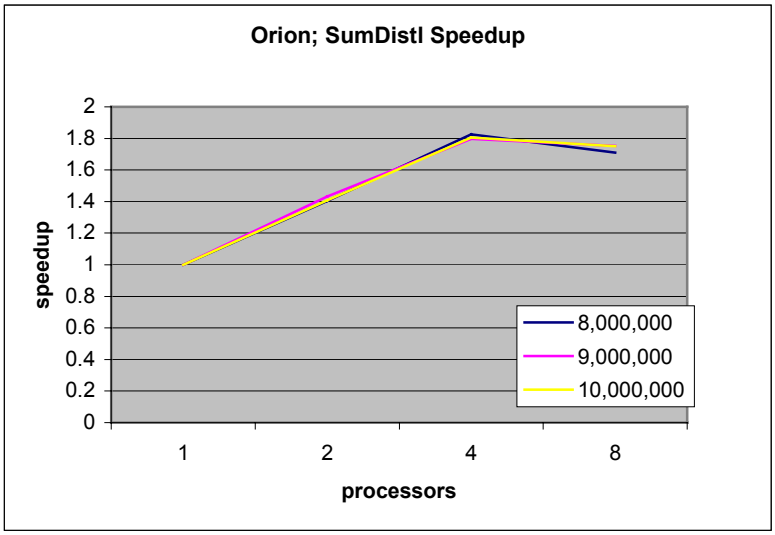


Figure 18: Orion; SumDistl speedup

The graph shows the speedup obtained from partitioning the sum and distl operations over multiple processors. Interestingly, all vector lengths tested showed differing performance but almost identical parallel speedup and efficiency. This is because increasing vector lengths (for these sizes) causes a constant increase in time to execute the reduce(sum) and distl operations in parallel. Also, the communication (during split and distl) to computation (during reduce) ratio becomes to high above 8 processors resulting in a decline in speedup for all data-sizes. This *probably* means that the communication to computation ratio remains constant with data-size for this particular problem. Parallel efficiency ranges from 70% for 2 processors to 20% for 8 processors, which is quite reasonable.

5.2.1.3 Remote

The aim of this test is to demonstrate that promising parallel speedup and efficiency can be obtained using a non-trivial example that utilises a reasonable number of the parallel constructs implemented. While the translator did not automatically produce the C/MPI code used in this example (see section 5.1.3), it does directly implement the computational pattern specified by the parallel BMF generated from the remote.Adl program, and therefore provides a reasonable basis for comparison.

The C/MPI code for remote.Adl (appendix D.1) was run on Hydra with integer lists of length 1000-10000 but only results for 3000-7000 are presented. The first two graphs below show speedup and efficiency results recorded on Orion for list lengths 3000-6000 with 1,2,4 and 8 processors.

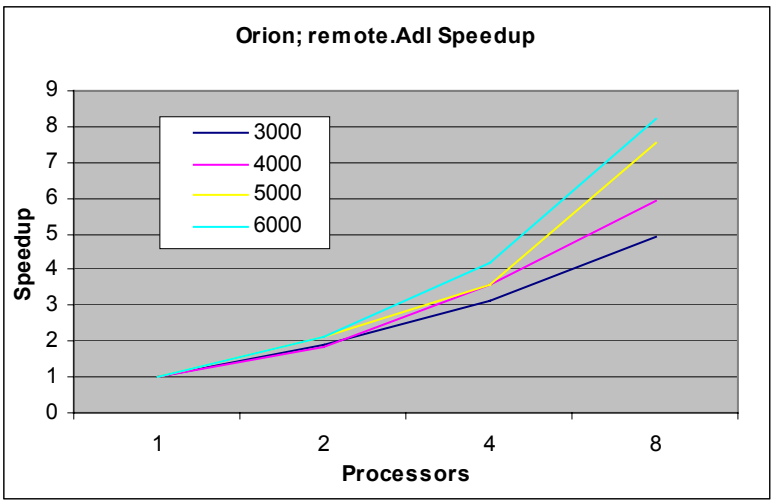


Figure 19: Orion; remote.Adl speedup

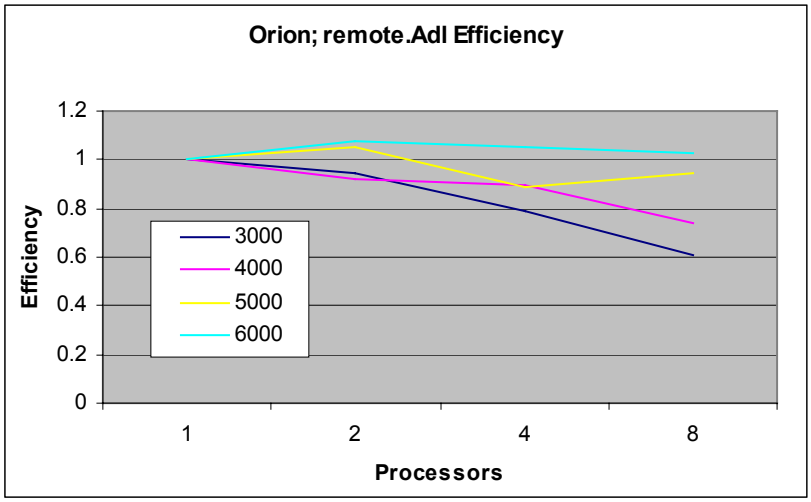


Figure 20: Orion; remote.Adl efficiency

Speedup and efficiency look quite promising on Orion for all vector lengths tested. For an input vector of 6000 integers, all numbers of processors even exhibited superlinear speedup (and consequently over 100% parallel efficiency). Interestingly, vectors of length 5000 had 105% efficiency for 2 processors, 88% efficiency for 4 processors and then 94% for 8 processors. This is probably the cause of one (or more) of the MPI processes being switched by the OS during execution (non-dedicated use of the machine). Nonetheless, this example seems to exhibit excellent speedup and efficiency.

The next two graphs were obtained on Hydra with the same data-sizes but with 1,2,4,8,16 and 32 processors (with the same number of nodes in each case).

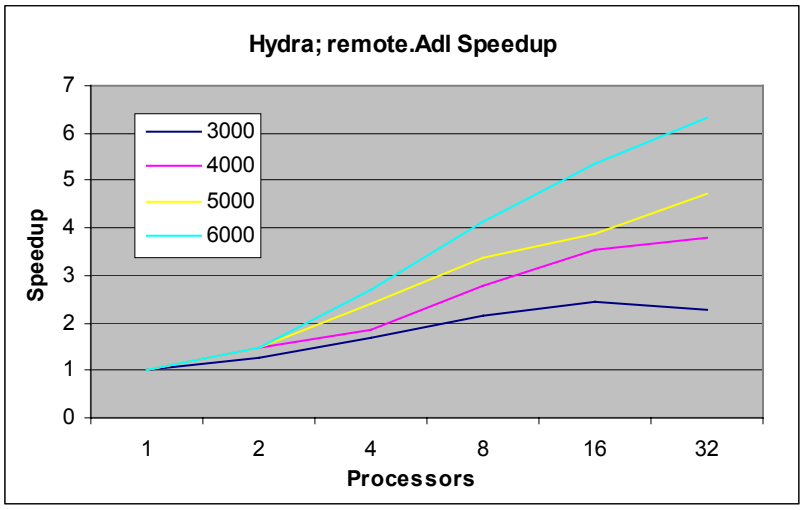


Figure 21: Hydra; remote.Adl speedup

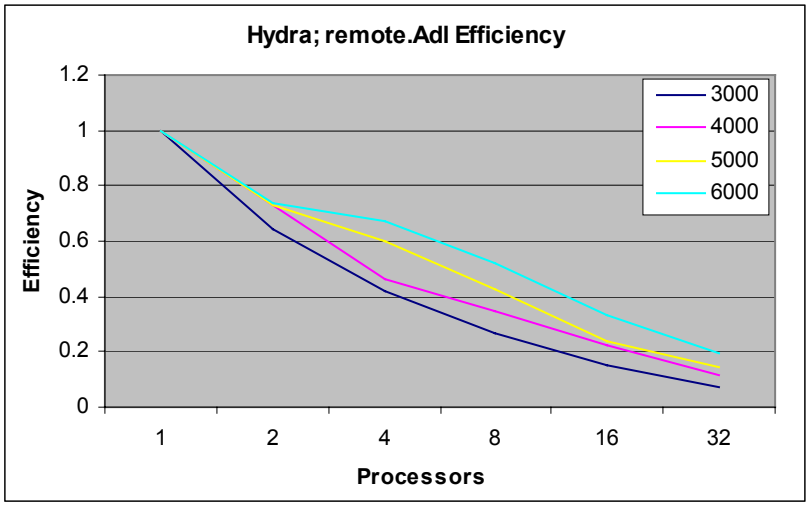


Figure 22: Hydra; remote.Adl efficiency

Speedup and efficiency seem to tell a different story on Hydra, as apposed to the good results obtained from Orion. For vectors of length 3000 and 4000 the efficiency for 2-4 processors is about 45-75%, compared with 78-94% on Orion. Also, for vectors of length 6000 Hydra records speedups of 1.5, 2.6 and 4.2 for 1,2 and 8 processors while Orion recorded speedups of 2.1, 4.2 and 8.2 for the same processor numbers. There could be a few different causes to this difference. The bandwidth and/or latency characteristics of Hydra could be worse than Orion causing greater communication overhead and therefore less speedup. Alternatively, algorithms used in the MPICH implementation of MPI could be causing communication bottlenecks, which would certainly limit speedup. A simple latency/bandwidth test to measure the time taken to send and receive different data-sizes on both Hydra and Orion might reveal worse communication characteristics on Hydra than Orion. The speedup graph below shows a further test conducted on Hydra with a vector size of 7000.

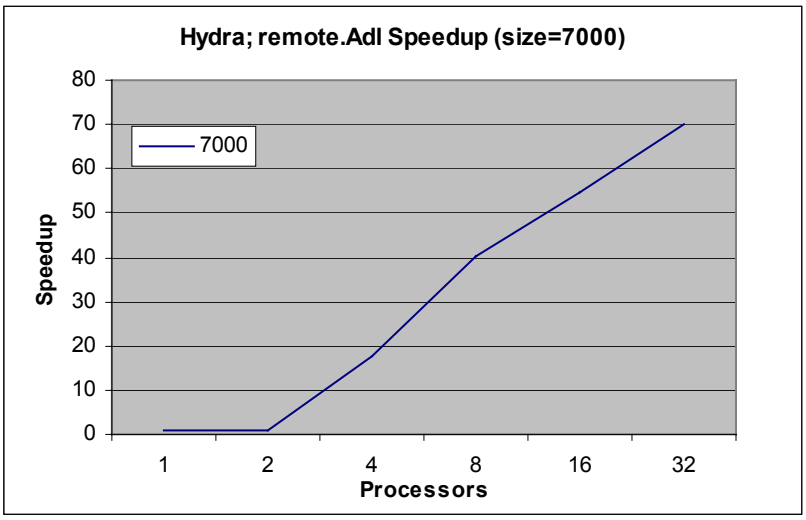


Figure 23: Hydra; remote.Adl Speedup (size=7000)

The execution time for 1 and 2 processors in this example went over a minute, which probably means that data was also being paged as well as causing cache/memory misses. When the number of processors is increased over 4 this effect was gone (more nodes = more memory) and speedup therefore skyrocketed to about 17 for 4 processors, 440% efficiency!.

To demonstrate the performance difference between functional code and imperative code, a sequential C version to compute the same result as remote.Adl was implemented and timings presented below. The results here were obtained using the standard RedHat time utility on the Hydra machine.

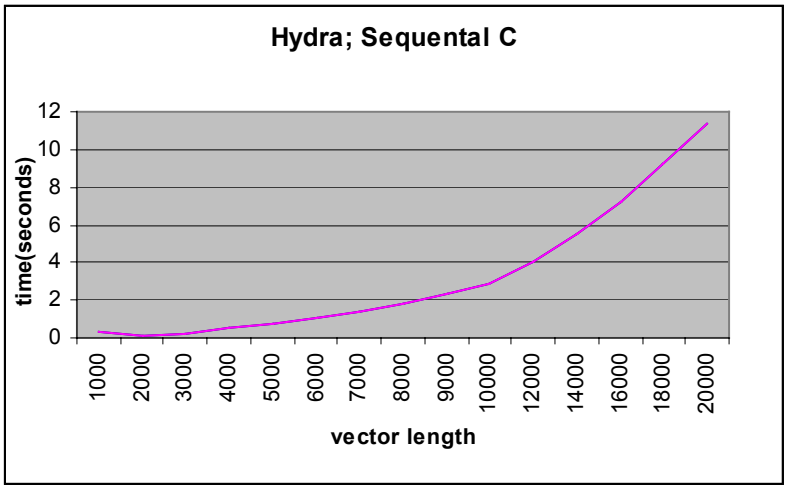


Figure 24: Time for C code to compute remoteness

All performance figures from this program are far superior to any of the parallel results presented earlier. This is because the C/MPI code produced by the translator (or otherwise) is an imperative implementation of a functional algorithm and sequential C is a pure imperative implementation using an imperative algorithm. This means that sequential C results should *not* be used to measure speedup for the C/MPI code of remote.Adl because they are written for different programming paradigms. To most accurately measure speedup, the C/MPI code for a sequential BMF version of remote.Adl should be used.

To get the C/MPI code produced by the translator competing with the sequential C algorithm, implementation optimisations need to be performed. These could include update-in-place or global analysis of programs. Another option might be transformation to an intermediate imperative form before translation to C/MPI to allow better optimisations.

5.2.2 Absolute Performance

This section presents the absolute performance of the parallel BMF constructs implemented. A suit of short programs was written (one per construct) to call the parallel methods implemented. Timing information was gathered in-code

using MPI_Wtime(). Three sets of experiments were conducted on Hydra with data-sizes of 10000, 100000 and 1000000 with 1,2,4,8,16,32 and 64 processors (using 32 nodes). The following table corresponds to the data-size of ten thousand, sizes of 10000 and 1000000 can be found in appendix C.4.

	1	2	4	8	16	32	64
MPI_Barrier()	0.000006	0.009977	0.016245	0.019906	0.022949	0.027006	0.062553
nSplit()	0.000976	0.016589	0.023166	0.026673	0.032136	0.190252	0.165801
hSplit()	0.00001	0.016286	0.061832	0.032849	0.046272	0.166111	0.14958
sMerge()	0.000964	0.007262	0.006909	0.006993	0.007194	0.007591	0.007839
fmyReduceI()	0.000607	0.000357	0.000323	0.000385	0.000463	0.000543	0.000833
myReduceI()	0.000295	0.000397	0.00034	0.000387	0.000449	0.000549	0.000649
reduceConcat()	0.000007	0.006491	0.010112	0.012819	0.012937	0.01587	0.015427
reduce()	0.000311	0.000294	0.000203	0.000224	0.000244	0.000262	0.000307
myScanI()	0.0004	0.000515	0.00068	0.001012	0.001645	0.002857	0.00556
scanConcat()	0.000008	0.006576	0.016223	0.036517	0.07365	0.154496	0.293299
scan()	0.008991	0.005144	0.002622	0.001423	0.000963	0.000702	0.000594
zip()	0.02383	0.012557	0.006722	0.003323	0.001715	0.000954	0.00062
repeat()	0.000986	0.025286	0.062412	0.134383	0.285706	0.599811	1.126089
distl()	0.01581	0.009917	0.005807	0.003148	0.001875	0.001304	0.001133
pSelect()	0.000005	0.006383	0.0043	0.005845	0.003225	0.002352	0.002056

Table 3: All constructs; length=100000

The results for hSplit() and nSplit() show similar performance times for all numbers of processors (and sizes) except around (and possibly above) 64 processors. This is interesting because it may suggest that hSplit() has a bug in it or executes an imperfect algorithm (nSplit() is supposed to be slower than hSplit()). A similar effect seems to occur when comparing sMerge() and reduceConcat(), which both perform a merge operation.

Performance figures for the custom implementation of MPI_Scan, myScan, show faster times for small numbers of processors but slower for larger numbers of processors. A simple algorithm was used in the custom implementation of MPI_Scan and thus gets out-performed with an increase in processors. A second (better) algorithm was also implemented and explained section earlier. Similar effects can be seen when comparing the custom implementation of MPI_Reduce, perhaps due to another naive algorithm.

5.3 Difficulties and Findings

This section discusses some of the difficulties/findings that were encountered throughout the development of both the translation system and target code implementation.

5.3.1 Platform Differences

One major problem the parallel implementation faced is obtaining consistent speedup/efficiency results on different parallel architectures. Specifically, different results were recorded on Hydra to that on Orion. Conducting further experiments to investigate these issues and on different parallel architectures is expected to be the subject of future work.

Nodes on the Hydra machine run RedHat with MPICH installed but nodes on Orion run Solaris with SunHPC cluster tools to run MPI. There are few small interface differences between the MPICH implementation and the Sun implementation, which caused minor porting problems. The two compilers, mpicc and mpcc, also have quite different compiler options. Most of these platform differences where resolved without much effort.

5.3.2 Variables

Variable declaration of code produced by the translator turned out to be a problem because mpcc and mpicc do not allow declaration of variables anywhere within a program. This was a particularly annoying problem because new versions of the gcc compiler allow for this C++ declaration style. At present, the translator just enters a new scope level for each variable declared and remembers how many brackets need closing at the end of the program. This strategy could fall over for code generated for large programs because mpcc and mpicc probably impose a limit to the amount of nesting.

Another interesting problem encountered in developing the translator was value references. The parallel BMF style of value reference only allows reference to input values of a function. A B_alltup contains multiple output values of functions and therefore a function that inputs a B_alltup can reference multiple values. The translator presently generates a C struct with pointers to output values of the corresponding functions to mimic this referencing style. This implementation is simple and fits with the functional form of BMF but a better mechanism may be possible that avoids extra allocation of structures just for the purpose of variable reference.

5.3.3 Barrier Synchronisation

Message passing programs often require barrier synchronisation after a method that executes in parallel or contains communication. This means that the translator must analyse whether or not two functions are ‘parallel’ and insert a MPI_Barrier() between them. The simple solution currently used in the target code implementation is: ‘every parallel method performs barrier synchronisation before returning control to the calling program’. This effectively means that all performance figures for parallel methods reported in chapter 4 include the cost of barrier synchronisation in them. This problem is partly due to the lack of a P_comp function for composing parallel functions in the source code.

5.3.4 Translation Information

Like most compilers, the translator carries around a lot of information about the code it is generating such as lengths, names and types of values. An important observation is that it does not really matter how long the translation takes to generate the code. Specifically, if more type (or otherwise) information about the program being generated is available, and this information is vital to the efficient code generation of the program, then it should be provided. The motivation for this is that the ultimate user of the Adl compiler is mainly interested in gaining parallel speedup. Investigation into the use of extra program information (or extra translation stages) for greater generated-code efficiency is a subject of future work.

5.3.5 Garbage Collection

The current target code implementation naively ignores issues relating to garbage collection. The implementation is written in C/MPI, which does not provide any form of automatic collection. A fixed allocation and de-allocation scheme needs to be implemented to stop large programs chewing up lots of memory. The obvious approach is to adopt the policy of de-allocate input data and allocate for output data. It is important to note that this will not make the implementation more efficient; it will probably make it slower. A related issue is the implementation of update-in-place analysis, which is expected to have some performance advantages similar to found in NESL [24]. Both garbage collection and update-in-place analysis are subjects of future work.

5.3.6 Type Polymorphism

A lack type polymorphism or even function overloading in C caused a considerable amount of unnecessary code repetition (e.g. myReduceI() and myReduceD()). There are two possible solutions to this problem. Macros could obviously be used to allow a single method to work with data of multiple types. Another solution is to perhaps port the implementation to C++ and use template classes. The latter solution may be a better idea because then other ‘nice’ features such as exceptions and objects provided by C++ would clean up the code and therefore make it more readable and robust. Investigating these possibilities and their resultant advantages and possible performance decreases (due to use of ‘rich’ language features) might be the subject of future work.

5.3.7 Nested-Data Types

The heavily nested data-types allowable by parallel BMF are somewhat difficult to accurately (and efficiently) express in C, in some situations. For example, a zip operation combines corresponding elements of distributed lists into a single distributed list of pairs. The ‘elements’ of the original distributed

lists may themselves be distributed lists, and can thus be arbitrarily nested in this way. A clean scheme for representation, allocation and traversal of such structures has not yet been implemented, and is therefore the subject of future work. It is, however, important to note that most parallel BMF programs are not likely to cause to many difficulties for the translator/target code implementation.

6 Conclusions

6.1 Summary

There are a range of reasons why parallelism is attractive, including the opportunity for increased performance, bounded sequential processor speeds, economic forces and a strong application demand. While research into parallelism has received considerable attention, it is not without its drawbacks. A variation in parallel architectures, combined with a lack of computational model, has seen parallel programs being customised to particular machines. Furthermore, continual advancements in interconnection network technology and Moore's law mean that parallel computers quickly become obsolete, dragging their customised software with them.

Parallelism's attractiveness is therefore masked by a barrage of difficulties thereby affecting programmers enthusiasm towards it, and consequently the widespread acceptance of parallel computing is compromised. The aforementioned issues, combined with the difficulty of programming distributed memory computers, necessitates more research into finding a unified parallel model of computation with implicit parallelism and architecture abstraction.

The goal of the Adl project is to provide an efficient implementation of a data-parallel language in the framework of a distributed memory architecture. Adl boasts implicit parallelism and architecture independence, which are desirable features of a parallel model of computation. Implicit parallelism is achieved by defining operations on aggregate data-structures, and architecture independence through algebraic transformation of an intermediate form BMF.

This project has developed a back-end implementation of Adl by defining a translation from parallel BMF code to C/MPI code. A description of the two new compiler components, the translator and target code implementation, has been given and a subsequent speedup/efficiency analysis, with example programs, demonstrating promising results was provided. Consequently, the Adl project is closer to providing a usable functional language with automatic (or user guided) parallel performance, which may contribute to the widespread adoption of parallel computing in the future.

The implementation Adl's new compiler components' has uncovered a number of issues, which need to be addressed. Different parallel machine characteristics and corresponding software platforms can cause both minor porting problems and variation in parallel speedup. A lack of features such as type polymorphism and overloading in the target code (C/MPI) can cause unnecessary code repetition. And, the expression and implementation of arbitrarily nested data-types in C/MPI is non-trivial and requires completion. Finally, additional effort is needed to investigate both algorithm choices and

optimisation in the target code implementation for maximum efficiency. Section 6.2 on future work offers some directions for addressing these, and other, issues.

6.2 Future Work

To spite the significant portion of work conducted in this project, there is still tremendous scope for future directions of work. This thesis has mentioned various avenues for which this work could proceed, which will now be summarised.

6.2.1 Improve and Complete

To be of maximum use to the Adl project, both the translation system and its target code implementation need to be improved and completed. There are various possible implementation algorithms for all of the parallel BMF constructs implemented, which have varied performance. Due to time constraints, this project could not explore more than a couple of these choices and future versions should endeavor to do so. The parallel BMF constructs implemented include: split, merge, map (through translation), reduce, scan, zip, repeat, distl and select. There are various other constructs defined by parallel BMF, which need both translation and implementation.

As mentioned in chapter 4, a general mechanism for defining, allocating and traversing arbitrarily nested data-structures has not been fully implemented. To be completely general, more work needs to be done on both the translation rules and target code implementation in this area. Furthermore, due to the lack of polymorphism and overloading in C/MPI, there is some code repetition in the parallel implementation for different types. Future versions may investigate the use of macros in C or template classes upon porting to C++.

6.2.2 Parallel Architectures

Chapter 5 presented various speedup/efficiency results for some example programs. The Hydra supercomputer was found to exhibit considerably less speedup/efficiency than Orion. Conducting experiments, such as latency/bandwidth tests or examining the MPICH software, to discover the reason for this is expected to be the subject of future analysis. Furthermore, it would also be wise to investigate the implementations efficiency on a range of other parallel architectures to discover any other problems.

6.2.3 Optimisation

The task of obtaining speedup has been completed. The next step is to optimise the translation/implementation. Section 5.3.5 mentioned a few opportunities for doing so. First and foremost is to address the issue of garbage collection, since

a fixed allocation/de-allocation scheme was not implemented. Another avenue is to explore the implementation of update-in-place analysis, since this has proven performance advantages in other functional languages. Also, providing more information during translation, or conversion to an intermediate imperative form for optimisation before translation to C/MPI would be an interesting research path. Other approaches also exist, such as optimising sequential BMF.

6.2.4 Nested Parallelism

A final interesting research direction for this work is to explore the feasibility and resultant efficiency of implementing nested-parallelism using MPI. Nested parallelism allows the parallel application of a parallel function to multiple tasks. This is probably a somewhat challenging problem, however there are a number of applications that exhibit nested parallelism [25].

7 References

[1] D. B. Skillicorn, D. Talia
Models and Languages for Parallel Computation
ACM Computing Surveys, 1996

[2] G. E. Moore
Cramming More Components onto Integrated Chips
Electronics, Vol 38, Num 8, April 19th 1965.

[3] Intel
Intel following Moore's law
www.intel.com/research/silicon/mooreslaw.htm

[4] H. Kuchen
A Skeleton Library
Technical Report 6/02-I, Department of Information Systems,
University of Münster, 2002

[5] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi and S. Pelagatti
ANACLETO: A Template-Based P3L Compiler
Proceedings of the Seventeenth Parallel Computing Workshop,
(PCW '97) Canberra, August 1997

[6] R. Bird.
A calculus of functions for program derivation
Technical Report 64, Programming Research Group,
Oxford University, 1987

[7] D.B. Skillicorn
The Bird-Meertens Formalism as a parallel model
Software for Parallel Computation, volume 106 of NATO ASI
Series F, pages 120-133. SpringerVerlag, 1993

[8] C. B. Jay
Costing Parallel Programs as a Function of Shapes
Science of Computer Programming,
Vol 37, Num1—3, Pages 207-224, 2000

[9] J. Backus.
Can Programming be liberated from the Von-Neumann style? A
functional style and its algebra of programs.
Communications of the ACM, Vol 21, No. 8, Pages 613-641, August 1978

[10] C. Walinsky and D. Banerjee
A Data-Parallel FP Compiler
Journal of Parallel and Distributed Computing,
Vol 22, Pages 138-153, 1994

[11] P. Rao, C. Walinsky
An Equational Language for Data-Parallelism
Proceedings of the fourth ACM SIGPLAN symposium on Principles
and Practice of Parallel Programming, San Diego-California,
Pages 112-118, 1993

[12] B. Alexander, D. Engelhardt and A. Wendelborn

An Overview of the Adl Language Project.
In Proceedings Conference on High Performance Functional Computing, Denver, Colorado, April 1995

[13] B. Alexander
Mapping Adl to the Bird Meertens Formalism.
Technical Report 94-18, The University of Adelaide
September 1994

[14] Brad Alexander
Data Movement Optimisation in Adl
Technical Report, The University of Adelaide

[15] P. Roe
Derivation of Efficient Data Parallel Programs
Proceedings of 17th Australian Computer Science Conference,
Vol 16, No. 1, Pages 621-628, January 1994.

[16] B. Alexander
Pending Thesis,
The University of Adelaide

[17] Joseph Windows
Parallelisation of Code Written in Bird-Meertens Formalism
Pending Thesis,
The University of Adelaide

[18] Sun HPC Cluster Tools
http://www.sun.com/software/hpc/

[19] MPICH and Miranet
http://www.myri.com/scs/

[20] Linpack Benchmark
http://www.top500.org/lists/linpack.php

[21] Top 500 Fastest Computers
http://www.top500.org

[22] Hydra Supercomputer; SAPAC
http://www.sapac.edu.au

[23] S. Jones
The Implementation of Functional Programming Languages
Prentice-Hall International Series in Computer Science, 1987

[24] G.E. Blelloch, S. Chatterjee, J.C. Hardwick, J.Sipelstein and M. Zagha
NESL: Implementation of a Portable Nested Data-Parallel Language
Journal of Parallel and Distributed Computing,
Vol 21, No 1, Pages 4-14, 1994
NESL: Implementation of a Portable Nested Data-Parallel Language

[25] G. E. Blelloch
Vector Models for Data-Parallel computation
Cambridge, Mass: MIT Press, C1990

Appendix A

A.1 Definitions and Acronyms

- **BMF:** Bird-Meertens Formalism
- **Adl**: The functional source language of the Adl language project.
- **C/C++:** Imperative (C), and Object Oriented (C++) languages.
- **Miranda:** A functional programming language.
- **MPI:** Message Passing Interface. An interface used with a programming language to communicate between processes of a parallel program.
- **HPC:** High Performance Computing
- **BMF:** Bird-Meertens Formalism.
- **SIMD:** Single-Instruction Multiple-Data. A classification of a parallel computer.
- **MIMD:** Multiple-Instruction Multiple-Data. A classification of a parallel computer.
- **TeraFlops:** 1 Trillion floating-point operations per second.
- **RAM:** Random Access Memory.
- **mpcc:** Sun's C/MPI compiler.
- **mpicc:** MPICH's MPI compiler.

Appendix B

B.1 Translation System

This appendix provides a full description of the important function definitions and type declarations defined within the translation system. Some of the following sections will contain information already described in the main body of this thesis; the aim of this appendix is to provide a dictionary-like description of the translator's components. Section B.1.1 describes the types component, section B.1.2 describes the translator component and section B.1.3 describes the auxiliary component.

B.1.1 The types Component

This section elaborates important Miranda definitions in the transTypes.m literate script. Section B.1.1.1 repeats the definition used to define parallel BMF. Section B.1.1.2 provides the definitions used to define the form of BMF numbers, constants and operators. Section B.1.1.3 repeats the definition to define the form of parallel BMF input. Section B.1.1.4 provides some important definitions used to convert between BMF and C/MPI types and values. And finally, section B.1.1.4 gives the full definition of the abstract type used to describe a translation state.

B.1.1.1 Defining Parallel BMF

```
b_exp ::=
      B_id |
      B_con b_con |
      B_comp b_exp b_exp |
      B_if b_exp b_exp b_exp |
      B_alltup [b_exp] |
      B_allvec [b_exp] |
      B_map b_exp |
      B_op b_op |
      B_reduce b_exp b_exp |
      B_scan b_exp b_exp |
      B_addr b_num b_num |
      B_zip b_exp |
      B_distl |
      B_repeat |
      P_map b_exp |
      P_reduce b_exp |
      P_scan b_exp |
      P_split b_num b_num |
      P_zip b_exp |
      P_repeat b_exp |
      P_distl |
      P_project |
```

```
B_program b_exp inputType
```

The following describes the meaning of individual components of the b_exp type definition used to define the syntax/form of parallel BMF. Note for below: definitions 1,2,3,4 and 5 can be found in section 2.2.

- `B_id`: identity function.
- `B_con`: constant function; integers, reals and booleans.
- `B_comp`: function composition; `(B_comp e2 e1)` means e1 is evaluated before e2.
- `B_if`: if predicate, then {consequent}, else {alternative}.
- `B_alltup`: Apply every function of [b_exp] to a copy of the input, creating a tuple of output values.
- `B_allvec`: Apply every function of [b_exp] to a copy of the input, creating a vector of output values.
- `B_map`: Sequentially apply a function to all elements of a list (defn 1).
- `B_op`: BMF operators; length, less-than, indexing, etc.
- `B_reduce`: Sequential reduce (definition 2).
- `B_scan`: Sequential scan (definition 3).
- `B_addr`: Address an element of a B_alltup.
- `B_zip`: Sequential zip (definition 5).
- `P_map`: Apply a function in parallel to all elements of a list (defn 1).
- `P_reduce`: Parallel reduce (definition 2).
- `P_scan`: Parallel scan (definition 3).
- `P_split`: Distribute (split) a list among processors.
- `P_zip`: Parallel zip (definition 5).
- `P_repeat`: Repeat a value a number of times over processors, creating a distributed list.
- `P_distl`: Distribute a value over a distributed list, creating a distributed list of pairs.
- `P_project`: Parallel select (definition 4).
- `B_program`: A parallel BMF program. Composed of a b_exp and input ‘data’ (inputType).

B.1.1.2 Numbers, Constants and Operators

```
b_num ::= B_num num
b_con ::= B_int num | B_real num | B_true | B_false
b_op ::=  B_index |
          B_plus | B_times | B_minus | B_divide |
          B_min | B_minloc | B_max | B_maxloc |
          B_and | B_or | B_eq | B_neq | B_gt | B_lt |
          B_length | B_uminus | B_neg |
          B_project | B_iota | B_conc
```

- `B_num num`: Numbers, mostly used for parameterising functions.
- `B_int num`: Constant integer.
- `B_real num`: Constant real number.
- `B_true, B_false`: Constant Boolean.
- `B_index`: List indexing operator.
- `B_plus, B_times etc`: Binary arithmetic operators.
- `B_min, B_maxloc etc`: MPI operators. MPI_MIN etc. Mostly used to test the implementation reduce and scan functions.
- `B_and, B_gt, etc`: Binary comparison operators.
- `B_length`: List length operator.
- `B_uminus`: Urinary minus operator.
- `B_neg`: Negation operator.
- `B_project`: Sequential project operator.
- `B_iota`: List creation operator.
- `B_conc`: List concatenation operator.

B.1.1.3 Parallel BMF Input

```
tuple==(num,[inputType])
inputType ::=
     Int num | Real num | Bool bool |
     Tuple tuple |
     IntList (num,[num]) | RealList (num,[num]) |
     BoolList (num,[bool]) |
     NestList (num,[[inputType]]) |
     TupleList (num,[tuple])
```

- `Int num, Real num, Bool bool`: Defines the form of singular input values for integers, real numbers and booleans.
- `Tuple tuple`: Defines the form of a BMF tuple, with embedded type information on its length and individual components.
- `IntList (num,[num]) etc`: Defines the form of a list of integers, reals and boolens so that length and type information is known.
- `NestList (num,[[inputType]])`: Defines the form of a nested list, explicitly representing the number of nested lists and the length and type of each of nested lists.
- `TupleList (num,[tuple])`: Defines the form of a list of tuples, with all type information is embedded.

B.1.1.4 Translator State

```
types==[string]
names==[string]
sizes==[string]
parallel==bool
```

```
statements==string

abstype
  state
with
  type_of :: state -> types
  name_of :: state -> names
  size_of :: state -> sizes
  parallel_of :: state -> parallel
  stmts_of :: state -> statements
  create :: types -> names -> sizes ->
            parallel -> statements-> state
  empty :: state
state == (types,names,sizes,parallel,statements)

type_of (t,n,s,p,stmts) = t
name_of (t,n,s,p,stmts) = n
size_of (t,n,s,p,stmts) = s
parallel_of (t,n,s,p,stmts) = p
statements_of (t,n,s,p,stmts) = stmts
```

The first section of code defines type synonyms so that the definition of the abstract type is more readable. The second section of code is a prototype definition for ‘public’ functions that are provided by the type definition. The actual definition of the state is a tuple of types, names, sizes, parallel and statements. Descriptions of the components the state tuple were given in the main body of text, but are repeated here. The last section of code defines functions to access components of a state; their definitions are self-explanatory.

- `types`: This component is an array of string elements describing the C/MPI types corresponding to the input/output values of the parallel BMF construct currently being translated. For example, if a function produces a distributed vector (of tuples) this component might contain [“DataInfo”,”alltup”], where alltup is the name of the previously generated tuple type. Because we are currently only dealing with a subset of types this mechanism is sufficiently general for now.
- `names`: This component is an array of string elements describing the C/MPI variable identifiers assigned to the input/output values of the parallel BMF construct currently being translated. For example, the P_reduce function might assign this component to [“reduceResult”].
- `sizes`: This component is an array of string elements describing the number of elements contained in the vectors specified by the input/output values of the parallel BMF construct currently being translated. If the input/output values are singular values (e.g. integers) this component contains [“1”]. For example, the P_repeat function (which outputs a distributed list of length==copies) might assign this component to [“copies”]. The strings in this component may be either constant integers or references to variables containing constant integers.

- `parallel`: This component is a boolean value that specifies if the target code being produced is currently being run in parallel. This basically tracks whether or not the P_split function has been encountered.
- `statements`: This component of the translators state is the most important. It is a string containing the C/MPI statements representing the program currently being translated. It is also the output of the translator. For example, the B_con (B_real 33.5) function would set this component to “double temp = 33.5;\n”.

B.1.1.5 Conversion Functions

```
showValue :: inputType -> string
```

This function inputs an inputType value and outputs a string, which represents an equivalent C/MPI value (assuming the value is being assigned to a variable of that type). All constructors of inputType can be converted to their corresponding representation. A helper function called showTuple handles conversion of tuples, and is mutually recursive to showValue. The following is an example of a call to showValue:

```
>showValue Tuple(2,[Real 3.5,Bool True])
{3.5,True}
```

```
constType :: b_con -> string
```

This function inputs a b_con expression (appendix B.1.1.2) and outputs a string that represents an equivalent C/MPI type. The following is an example of some calls to constType:

```
>constType (B_true)
boolean
>constType (B_int 3)
int
```

```
constValue :: b_con -> string
```

This function inputs a b_con expression (section B.1.1.2) and outputs a string that represents and equivalent C/MPI value. The following is an example of some calls to constValue:

```
>constValue (B_false)
False
>constValue (B_real 4.4)
4.4
```

```
showOp :: b_op -> string
```

This function inputs a b_op expression (section B.1.1.2) and outputs a string that represents its corresponding C/MPI operation. The following is an example of a call to showOp:

```
>showOp (b_op B_plus)
+
>showOp (b_op B_times)
*
```

```
opString :: b_op -> string
```

This function inputs a b_op expression (section B.1.1.2) and outputs a string that represents its corresponding MPI reduction/scan operation. Only the B_plus, B_times, B_min, B_minloc, B_max and B_maxloc constructors can be converted to this representation. The strings produced reference the global constants, specifying an MPI operation. The following is an example of a calls to opString:

```
>opString (b_op B_times)
PROD
>opString (b_op B_plus)
>SUM
```

B.1.2 The translator Component

This section describes the translation rules defined in the Miranda literate script called translator.m. The translator literate script provides two main function definitions. One, a function called trans (discussed in section B.1.2.1) that both initialises and finishes the translation of an input BMF program. And two, a function called gen (discussed in section B.1.2.2) which performs the core of the translation an input BMF program in recursive re-write style.

B.1.2.1 The Translation Function

The goal of the translation system is to map a textual representation of a parallel BMF program to a textual representation of an equivalent C/MPI program that utilises the methods described in section C.1. The role of the trans function is to both start and finish the translation of a program. Accordingly, the trans function will be invoked from the Miranda interpreter as follows:

```
>CMPIProgram = trans BMFProgram
```

It therefore seems obvious for the trans function to perform the following functions:

1. Concatenate C/MPI #include information to the output
2. Concatenate the start() method to the output
3. Concatenate the input data initialisation code to the output

4. Set answer = gen input_program
5. Concatenate the statement string of the answer to the output
6. Concatenate the finish() method to the output
7. Concatenate any unclosed brackets to the output
8. Return the resultant output string

Step 4 above makes reference to a function called gen, this performs a recursive re-write style translation of the input program and returns the final state of the translation. The next section describes the many rules required to translate a parallel BMF program into an equivalent C/MPI program.

B.2.1.2 The Code Generation Function

The goal of the gen function is to define a translation rule for every parallel BMF construct (described by b_exp). The Miranda type definition for the input and output of the gen function is as follows:

```
gen :: b_exp -> input_state -> output_state
```

Any rule of the gen function will therefore input a parallel BMF expression and a translation state and output a translation state. This means that each rule of the gen function will generate code based on the current parallel BMF expression and the input translation state. The output of each rule will be a translation state describing the code that was generated, since this may be useful in translating the next parallel BMF expression.

The rules defined within the translator are described below. The first rule described below is actually physically the last rule defined by the gen function, it is described here first because it gives a good basis for understanding how the translation of a b_exp proceeds. Each rule below references $CODE_GENERATED, this refers to the string of C/MPI code under the *Code Generated* heading of that rule.

`gen (B_comp e1 e2) input`
-State Input-
Any state appropriate for input to e1 and e2.

-Code Generated-
```
e2_statements;
e1_statements;
```
Where:
- *e2_statements*: the code generated by the function e2.
- *e1_statements*: the code generated by the function e1.

-State Output-
```
typeof = type_of y
name = name_of y
```

```
size = size_of y
parallel = or [parallel_of x, parallel_of y]
stmts = $CODE_GENERATED
```

Code is generated such that the function e2 is executed *before* the function e1. This rule is the most general code generation rule; it is the starting point for most parallel BMF programs.

`gen (P_split (B_num n1) (B_num n2)) input`

-State Input-
A state describing the non-distributed vector used as input to the split function.

-Code Generated-
```
DataInfo distrib = initialise(&name,sizeof(type),length);
hSplit(&distrib);
```
Where:

- *name*: the name of the non-distributed input value/vector.
- *type*: the C/MPI type of the elements of the non-distributed input value/vector.
- *length*: the number of elements in the non-distributed input value/vector.

-State Output-
A state describing the distributed vector generated by this split function:

```
typeof = ["DataInfo",inptype]  ||[distribType,elemType]
name = ["distrib"]
size = size_of input
parallel = True
stmts = $CODE_GENERATED
```

`gen (P_map function) input`

-State Input-
A state describing the distributed vector to be mapped over.

-Code Generated-
```
function_statements;
distrib.mydata = mappedList;
```
Where:

- *function_statements:* code generated by calling gen the function.
- *distrib:* the name of the input distributed vector.
- *mappedList:* the name of the non-distributed vector output from calling gen on the function.

-State Output-
```
type_of = ["DataInfo",hd (tl (type_of mapped))]
```

```
name = name_of input
size = size_of mapped
parallel = True
statements = $CODE_GENERATED
```

Note that mapped is the resulting translation state after calling gen function input. The result of the parallel map is a distributed vector with the same name as the input distributed vector. This is the most general parallel map; mapping a B_reduce function is a special case (due to translation problems) and is not described here.

gen (P_reduce (B_op B_conc)) input

-State Input-

A state describing the distributed vector used as input to the reduce function.

-Code Generated-

```
ListInfo merged = reduceConcat(&name);
```

Where:

- *name*: the name of the distributed input vector (with type DataInfo).

-State Output-

A state describing the non-distributed vector generated by this reduce function:

```
typeof = [“ListInfo”,listtyp]   ||[listType,elemType]
name = [“merged”]
size = size_of input
parallel = True
stmts = $CODE_GENERATED
```

Many of the gen functions translation rules are similar to that of P_reduce (e.g. P_scan). Most input a translation state describing a distributed vector and output a translation state describing the code generated to compute the result of the function.

gen (P_reduce op) input

-State Input-

A state describing both a distributed input vector and a locally computed reduce value.

-Code Generated-

```
type result= *(type*)reduce(&list,local,TYPE,operation);
```

Where:

- *type*: the output type corresponding to this reduce operation.
- *list*: the name of the distributed input vector.
- *local*: the name of the locally computed reduce input value.

- *TYPE*: the 'type descriptor' corresponding to this reduce operation. E.g. Integer or IntegerTuple.
- *operation*: the C/MPI reduce operation descriptor corresponding to the BMF operation specified by op. E.g. (b_op B_plus) is SUM.

-State Output-
A state describing the non-distributed value produced by the reduce function.

```
typeof = type_of input
name = ["reduceResult"]
size = ["1"]
parallel = True
stmts = $CODE_GENERATED
```

`gen (P_scan (B_op B_conc)) input`

-State Input-
A state describing the distributed vector used as input to this scan function.

-Code Generated-
```
DataInfo scanResult = scanConcat(&list);
```
Where:
- *list*: the name of the distributed input vector (of type DataInfo).

-State Output-
A state describing the distributed vector produced by this scan function:

```
typeof = ["DataInfo",listtyp] ||[distribType,elemType]
name = ["scanResult"]
size = size_of input
parallel = True
stmts = $CODE_GENERATED
```

`gen (P_scan op) input`

-State Input-
A state describing the distributed vector used as input to this scan function.

-Code Generated-
```
DataInfo sresult = scan(&list,TYPE,operation);
```
Where:
- *list*: the name of the distributed input vector (of type DataInfo).
- *TYPE*: the 'type descriptor' corresponding to this scan operation. E.g. Integer or IntegerTuple.
- *operation*: the C/MPI scan operation descriptor corresponding to the BMF operation specified by op. E.g. (b_op B_plus) ⇔ SUM.

-State Output-
A state describing the distributed vector produced by this scan function:

```
typeof = ["DataInfo",listtyp] ||[distribType,elemType]
name = ["sresult"]
size = size_of input
parallel = True
stmts = $CODE_GENERATED
```

gen (P_repeat (B_alltup[addr,copies])) input

-State Input-
A state appropriate for input to the addr and copies components of the B_alltup.

-Code Generated-
```
addr_statements;
copies_statements;
DataInfo repeated = repeat(size*sizeof(type),
                           value,copies);
```
Where:

- *addr_statements*: code generated by the addr component of the B_alltup.
- *copies_statements*: code generated by the copies component of the B_alltup.
- *size*: the number of elements in the value/vector specified by *value*.
- *type*: the type of the elements of the value/vector specified by *value*.
- *value*: the value to be repeated, generated the addr component of the B_alltup.
- *copies*: the constant/variable value specifying the number of times to repeat the *value,* generated by the copies component of the B_alltup.

-State Output-
A state describing the distributed vector produced by this repeat function:

```
typeof = ["DataInfo",reptyp]  ||[distribType,elemType]
name = ["repeated"]
size = [times]
parallel = True
stmts = $CODE_GENERATED
```

gen (P_zip (B_alltup[list1,list2])) input

-State Input-
A state appropriate for input to the list1 and list2 components of the B_alltup.

-Code Generated-
```
list1_statements;
list2_statements;
DataInfo zipped = zip(left,right);
```

Where:

- *list1_statements*: code generated by the list1 component of the B_alltup.
- *list2_statements*: code generated by the list2 component of the B_alltup.
- *left*: the name of the distributed vector generated by the list1 component of the B_alltup.
- *right*: the name of the distributed vector generated by the list2 component of the B_alltup.

-*State Output*-
A state describing the distributed vector produced by this zip function:

```
typeof = ["DataInfo","pair"]  ||[distribType,elemType]
name = ["zipped"]
size = min [size_of left, size_of right]
parallel = True
stmts = $CODE_GENERATED
```

```
gen (B_comp (P_project)
            (B_alltup[distrib,indexes])) input
```

-*State Input*-
A state appropriate for input to the distrib and indexes components of the B_alltup.

-*Code Generated*-

```
distrib_statements;
indexes_statements;
DataInfo selected = pSelect(source,indexes,size);
```

Where:

- *distrib_statements*: code generated by the distrib component of the B_alltup.
- *Indexes_statements*: code generated by the indexes component of the B_alltup.
- *source*: the name of the distributed vector generated by the distrib component of the B_alltup.
- *indexes*: the name of the re-arrangement (indexing) array, generated by the indexes component of the B_alltup.
- *size*: the length of the *indexes* array, generated by the indexes component of the B_alltup.

-*State Output*-
A state describing the distributed vector produced by this project function:

```
typeof = ["DataInfo",listtyp] ||[distribType,elemType]
name = ["selected"]
size = size_of sv             ||sv=gen distrib input
parallel = True
```

```
stmts = $CODE_GENERATED
```

gen (B_comp (P_distl) (B_alltup[x,y])) input

-State Input
A state appropriate for input to the x and y components of the B_alltup.

-Code Generated-
```
x_statements;
y_statements;
DataInfo dist = distl(value,sizeof(type),list);
```
Where:

- *x_statements*: code generated by the x component of the B_alltup.
- *y_statements*: code generated by the y component of the B_alltup.
- *value*: the name of the value to be distributed across the *list*. The name is extracted from the x component of the B_alltup.
- *type*: the type of the *value* to be distributed, generated by the x component of the B_alltup.
- *list*: the name of the list involved in the distl operation, generated by the y component of the B_alltup.

-State Output-
A state describing the distributed vector produced by this distl operation:

```
typeof = ["DataInfo","pair"]  ||[distribType,elemType]
name = ["dist"]
size = size_of dist           ||dist=gen y input
parallel = True
stmts = $CODE_GENERATED
```

gen (B_id) input = input

-State Input-
Any state.

-Code Generated-
none

-State Output-
The B_id function corresponds to a no-op in the implementation, so this rule turns the input state into the output state. The statements of the input are not passed on because this might cause them to be printed twice.

```
typeof = type_of input
name = name_of input
size = size_of input
parallel = parallel_of input
```

```
stmts = ""     ||don't repeat statements, just need state
```

gen (B_alltup list) input

-State Input-
Any state appropriate for input to all components of the B_alltup list.

-Code Generated-
```
statements_1;
statements_2;
     .
     .
     .
statements_26;
typedef struct{
     type1 *a;
     type2 *b;
          .
          .
          .
     type26 *z;
}alltup;
alltup mytup;
mytup.a = name1;
mytup.b = name2;
     .
     .
     .
mytup.z = name26;
```
Where:

- *statements_i*: code generated from calling the i'th component of the B_alltup.
- *alltup*: the corresponding C typedef to the B_alltup.
    - $type_i$: the output type of the i'th component of the B_alltup
    - *a,b,…*,z: the maximum number of components of a B_alltup. Only 26 components are allowed for ease of naming.
- *mytup*: a variable of type *alltup* to hold the results output from corresponding components of the B_alltup.
    - $name_i$: the name of the output value of the i'th component of the B_alltup. See *alltup* for B_alltup restrictions.

-State Output-
A state describing a tuple of results obtained from code generation of the components of B_alltup.

```
typeof = ["alltup"]++typs ||typs=all tup component types
name = ["mytup"]
size = concat [size_of e | e <- results]
```

```
parallel = or para
stmts = $CODE_GENERATED
```

A translation state is generated from calling gen on each component of the B_alltup. This information needs to be encoded in the output state of this function so that it may be later referenced by the B_addr function (rule). Accordingly, `typs` refers to a list of all the output types obtained from calling gen on all components of the B_alltup. The `size` component is a list of all output sizes obtained from calling gen on all components to the B_alltup. The `para` refers to a list of booleans obtained from calling gen on all components of the B_alltup.

gen (B_con const) input

-State Input-
Any state.

-Code Generated-
`type temp = value;`
Where:

- *type*: the type of the constant, determined by const in (B_con const).
- *val*ue: the constant value assigned to the variable temp. Constants can be int's, doubles or booleans.

-State Output-
A state describing the constant value produced by the B_con function:

```
typeof = [consttype const] ||e.g:consttyp(B_int i)=”int”
name = [“temp”]
size = [“1”]
parallel = parallel_of input
stmts = $CODE_GENERATED
```

gen (B_op (B_iota)) input

-State Input-
A state which outputs a constant/variable with typeof component=`[“int”]`.

-Code Generated-
`type *mylist = (type*)calloc(number,sizeof(type));`
Where:

- *type*: the type of the vector/array being created, usually 'int'.
- *number*: the number of elements in the vector/array being created, specified by the input constant/variable.

-State Output-
A state describing the vector/array produced by the B_iota function:

```
typeof = type_of input
name = [“mylist[0]”]
size = name_of input     ||variable contains length
parallel = parallel_of input
stmts = $CODE_GENERATED
```

gen (B_reduce op init) input

-State Input-
A state describing the non-distributed vector (or a local portion of a distributed vector) used as input to the sequential reduce function.

-Code Generated-
```
type brlocal = seqReduce{I/D}(&list,operation);
```
Where:

- *type*: the type of the result corresponding to this reduce operation.
- *list*: the name of the non-distributed input vector the sequential reduce will be performed over.
- *operation*: the C/MPI reduce operation descriptor corresponding to the BMF operation specified by op. E.g. (b_op B_plus) is SUM.

-State Output-
A state describing the non-distributed singular value produced by the B_reduce function:

```
typeof = [typ,hd (type_of input)]  ||remember vector type
name = [“brlocal”,inpname]         ||remember vector name
size = [“1”]
parallel = parallel_of input
stmts = $CODE_GENERATED
```

gen (B_scan op init) input

-State Input-
A state describing the non-distributed vector (or local portion of a distributed vector) used as input to the sequential scan function.

-Code Generated-
```
seqScan{I/D}(&(list.mydata),operation);
```
Where:

- *list*: the name of the non-distributed input vector the sequential scan will be performed over.
- *operation*: the C/MPI scan operation descriptor corresponding to the BMF operation specified by op. E.g. (b_op B_plus) is SUM.

-State Output-

A state describing the non-distributed vector produced by the B_scan function:

```
typeof = [typ,hd (type_of input)]  ||remember vector type
name = name_of input               ||remember vector name
size = size_of input
parallel = parallel_of input
stmts = $CODE_GENERATED
```

gen (B_map function) input

The translator does not currently implement this rule correctly. It is likely that it will generate code that looks something like:

```
{
  int i;
  for(i = 0; i < name.mydata.length; i++){
     type curr = *(type*)addrOf(name.mydata.tSize,
                                name.mydata.list,i);
     mapped_statements;
     ((type*)(name.mydata.list))[i] = mapped_name;
  }
}
```

Where *type* is the name of the type of elements in the local portion of the distributed vector and name is the *name* of the distributed input vector. *Mapped_statements* is the code generated from the function being mapped. *Mappded_name* is the name of the output value of the mapped function. This strategy will only work for mapping functions that map a vector of type t to a vector of type t (of the same length).

gen (B_addr (B_num size) (B_num offset)) input

-State Input-
A state describing a B_alltup, which is being referenced by this B_addr operation.

-Code Generated-
none

-State Output-
A state describing the value referenced by this B_addr function:

```
typeof = [(arrayIndex (offset-1) tuptypes)]
name = [tupname++"."++[(arrayIndex (offset-1) alphabet]]
size = [(arrayIndex (offset-1) tupsizes)]
parallel = parallel_of input
stmts = $CODE_GENERATED
```

This B_addr function inputs a state describing a B_alltup (previously described) and outputs a state describing the component of the B_alltup being referenced (offset). Therefore, the typeof component is assigned to tuptypes[offset-1], which is extracted from the information recorded by the input B_alltup. The name component is assigned to tupname.X, where tupname is also extracted from the input B_alltup and X is variable name at (offset-1) in the alphabet ("abcd…xyz"). The size component is derived similarly to the typeof component.

gen (B_if pred cons alt) input

-State Input-
A state appropriate for input to the pred, cons and alt components of the B_if function.

-Code Generated-
```
pred_statements;
type ifresult;
if (predname){
     consq_statements;
     ifresult = consqname;
}else{
     alter_statements;
     ifresult = altername;
}
```
Where:

- *pred_statements*: any code generated to acquire the boolean predicate.
- *type*: the type of the output values of consq and alter (both the same).
- *predname*: the name of the output variable (of type boolean) generated by the pred component of the B_if function.
- *consq_statements*: code generated by the consq component of the B_if function (corresponds to if-part statements).
- *alter_statements*: code generated by the alter component of the B_if function (corresponds to else-part statements).

-State Output-
```
typeof = type_of c        ||c = gen cons input
name = ["ifresult"]       ||result of this if statement
size = size_of c
parallel = parallel_of c
stmts = $CODE_GENERATED
```

gen (B_comp (B_op op) (B_alltup[B_id,B_id])) input

-State Input-
A state appropriate for input to the oper1 and oper2 components of the B_alltup.

-Code Generated-

```
oper1_statements;
oper2_statements;
type comp = lname OP rname;
```

Where:

- *oper1_statements*: code generated by the oper1 component of the B_alltup.
- *oper2_statements*: code generated by the oper2 component of the B_alltup.
- *lname*: the name of the variable generated by oper1.
- *OP*: the C/MPI operation corresponding to the BMF (B_op op). E.g. '+' corresponds to (B_op B_plus).
- *rname*: the name of the variable generated by oper2.

-State Output-

```
typeof = ["type"]          ||type of input
name = ["comp"]
size = ["1"]
parallel = parallel_of input
stmts = $CODE_GENERATED
```

`gen (B_comp (B_op (B_length)) addr) input`

-State Input-

A state appropriate for input to the addr component of the B_length function.

-Code Generated-

```
int len = list.mydata.length;
```

Where:

- *list*: the name of a distributed vector, generated by the addr component of this B_length function. I.e. `list.mydata` is a non-distributed vector.

-State Output-

A state describing a non-distributed singular integer (the length of an vector):

```
typeof = ["int"]
name = ["len"]
size = ["1"]
parallel = parallel_of input
stmts = $CODE_GENERATED
```

Future versions of this rule should check if the type of the input is a ListInfo, because this would cause the following code to be generated:

```
int len = list.length;
```

`gen (B_comp (B_op op) (B_alltup[oper1,oper2])) input`

-State Input-
A state appropriate for input to the oper1 and oper2 components of the B_alltup.

-Code Generated-
```
oper1_statements;
oper2_statements;
boolean comp = lname OP rname;
```
Where:

- *oper1_statements*: code generated by the oper1 component of the B_alltup.
- *oper2_statements*: code generated by the oper2 component of the B_alltup.
- *lname*: the name of the variable generated by oper1.
- *OP*: the C/MPI operation corresponding to the BMF (B_op op). E.g. '<' corresponds to (B_op B_lt).
- *rname*: the name of the variable generated by oper2.

-State Output-
```
typeof = ["boolean"]
name = ["comp"]
size = ["1"]
parallel = parallel_of input
stmts = $CODE_GENERATED
```

This rule is not actually correct; the code generated should be the same type as its operands. This would be quite easy to fix for future versions of this rule.

B.1.3 The auxiliary Component

This section describes the auxiliary functions defined in the Miranda literate script called transAux.m. The role of the function definitions described in this section is to abstract some BMF to C/MPI translation complexities away from the gen rule described in the previous section. More specifically, many of the functions provided in the transAux.m script return the $CODE_GENERATED value referred to in the previous section (B.2.1.2). The following sections describe the important function definitions in transAux.m and their purpose.

The interpretation of the Miranda function definitions in the following sections should take into consideration the following type synonyms:

```
name==string
data==string
typ==string
size==string
val==string
```

```
number==string
dim1==num
dim2==num
list==string
init==string
local==string
copies==string
value==string
op==(B_op b_op)
ind==string
sv==string
left==string
right==string
x==string
xtype==string
y==string
```

All strings quoted in the following sections *directly substitute* in parameters. I.e. the actual implementation uses string (list) concatenation to substitute in parameters.

B.1.3.1 distribute Function

```
>distribute :: name -> data -> typ -> size -> string
```
This function returns the string of C/MPI code to call the initialise() method:

"DataInfo name = initialise(&data,sizeof(typ),size);\n"

B.1.3.2 initialise Function

```
>initialise :: inputType -> (string,state)
```
This function returns a tuple with the first element representing the string of C/MPI code required to declare/initialise the inputType instance, and the second element the initial state of the translation. For example:

```
>myinput = IntList (4,[1,2,3,4])
>initialise myinput
("int input[4] = {1,2,3,4};\n",
 (["int"],["input[0]"],["4"],False,""))
```

B.1.3.3 declareType function

```
>declareType :: tuple -> string
```
This function returns a string of the C/MPI type definitions corresponding to the types of the elements in the input tuple. This function is used by the initialise function in declaring/assigning tuples (or lists of tuples). For example:

```
>mytup = Tuple (2,[Int 1,Bool True])
>declareType mytup
```

```
"int a;\n boolean b;\n"
```

B.1.3.4 declare Function

```
>declare :: typ -> name -> val -> string
```
This function returns a string of the C/MPI code required to declare a variable==name, of type==typ and with initialisation value==val:

" typ name = val;\n"

B.1.3.5 allocate Function

```
>allocate :: name -> typ -> number -> string
```
This function returns a string of the C/MPI code required to allocate memory:

" typ *name = (typ*)calloc(number,sizeof(typ));\n"

B.1.3.6 psplit Function

```
>psplit :: name -> dim1 -> dim2 -> string
```
This function returns a string of C/MPI code to call the hSplit() method:

"hSplit(&name);\n"

B.1.3.7 breduce Function

```
>breduce :: name -> list -> typ -> op -> init -> string
```
This function returns a string of C/MPI code to call the seqReduceI() or seqReduceD() method, depending on the typ parameter:

"typ name = seqReduceI(&list,(opString op));\n"

OR

"typ name = seqReduceD(&list,(opString op));\n"

Where opString is a function defined in transTypes.m.

B.1.3.8 preduce Function

```
>preduce :: name -> list -> typ -> local -> op -> string
```
This function returns a string of C/MPI code to call the reduce() method:

"typ name = *((typ*)reduce(&list,&local,(showType typ),(opString op)));\n"

Where showType and opString are functions defined in transTypes.m

B.1.3.9 mypreduce Function

```
>mypreduce :: name ->list -> typ -> local -> op -> string
```
This function returns a string of C/MPI code to call the myReduceI() or myReduceD() method, depending on the typ parameter:

"typ name = myReduceI(&list,(opString op));\n"

OR

"typ name = myReduceD(&list,(opString op));\n"

Where opString is a function defined in transTypes.m.

B.1.3.10 bscan Function

```
>bscan -> list -> typ -> op -> init -> string
```

This function returns a string of C/MPI code to call the seqScanI() or seqScanD() method, depending on the typ parameter:

"seqScanI(&list,init(opString op));\n"

OR

"seqScanD(&list,init(opString op));\n"

Where opString is a function defined in transTypes.m.

B.1.3.11 pscan Function

```
>pscan :: name -> list -> typ -> op -> string
```

This function returns a string of C/MPI code to call the scan() method:

"DataInfo name = scan(&list,(showType typ),(opString op));\n"

Where showType and opString are functions defined in transTypes.m

B.1.3.12 mypscan Function

```
>mypscan :: name -> list -> typ -> op -> string
```

This function returns a string of C/MPI code to call the myScanI() or myScanD() method, depending on the typ parameter:

"DataInfo name = myScanI(&list,(opString op));\n"

OR

"DataInfo name = myScanD(&list,(opString op));\n"

Where opString is a function defined in transTypes.m.

B.1.3.13 prepeat Function

```
>prepeat ::name ->value -> typ ->size -> copies -> string
```

This function returns a string of C/MPI code to call the repeat() method:

"DataInfo name = repeat(value,size*sizeof(type),copies);\n"

B.1.3.14 pzip Function

```
>pzip :: name -> left -> right -> string
```

This function returns a string of C/MPI code to call the zip() method:

"DataInfo name = zip(left,right);\n"

B.1.3.15 pproject Function

```
>pproject :: name -> sv -> ind -> size -> string
```

This function returns a string of C/MPI code to call the pSelect() method:

"DataInfo name = pSelect(sv,ind,size);\n"

B.1.3.16 pdistl Function

```
>pdistl :: name -> x -> xtype -> y
```

This function returns a string of C/MPI code to call the distl() method:

"DataInfo name = distl(x,sizeof(xtype),y);\n"

B.1.3.17 Wrapper Functions

```
>headers :: string
>start :: string
>finish :: string
```

The headers function returns a string of C/MPI code showing #include and main function information:

"#include <stdlib.h>\n"++
"#include "ParallelConstructs.h"\n"++
"int main(int argc, char **argv) {\n\n"

The start function returns a string of C/MPI code to call the start() method:

"int dummy = start(argv,argc);\n"

The finish function returns a string of C/MPI code to call the finish() method:

"finish()\n"

Appendix C

C.1 Construct Implementation

This appendix provides a description of all parallel (or otherwise) methods implemented in the construct implementation file (ParallelConstructs.h).

C.1.1 start()

```
void start(int argc, char** argv);
```

Parameters
(1) *argc*: an integer representing the number of arguments to the calling program.
(2) *argv*: an array of strings representing the arguments to the calling program.
Returns
void

This method starts the parallel MPI execution of the calling program with the arguments supplied by the argc and argv parameters. It calls the MPI_Init, MPI_Comm_rank and MPI_Comm_size operations to initialise the rank and np variables.

C.1.2 finish()

```
void finish();
```

Parameters
none
Returns
void

This method finalises the parallel processing of the program that originally called the start method. All executed code beyond calling this method will be carried out sequentially.

C.1.3 initialise()

```
DataInfo initialise(void *input, int tSize, int length);
```

Parameters
(1) *input*: a non-distributed value (contained the processor with rank==0), the input value of the calling program.
(2) *tSize*: an integer representing the size in bytes of the type of x.
(3) *length*: an integer representing the length of the input value.

Returns
A pointer to a structure describing a distributed vector of the input parameter (not-yet actually distributed).

This method creates the structure describing the distribution information required by various operations in ParallelConstructs.h. It performs allocation of each processors data and calls the partition() method which will be later used by the nsplit() or hsplit() methods.

C.1.4 to_global()

```
int to_global(DataInfo *info, int rank, int local);
```

Parameters
(1) *info*: a pointer to a structure representing the distribution of the vector being indexed in this method.
(2) *rank*: an integer representing the id of the processors data being indexed from.
(3) *local*: an integer representing the local processor index.
Returns
An integer representing the index corresponding to the parameter==local into the global vector.

This method calculates the index in the global vector that the parameter specified by local corresponds to. This is useful when a processor is examining a distributed portion of data and needs to calculate its position relative to other portions of distributed data. This method is intended to be used exclusively by the implementation itself in the hSplit() and sMerge() methods. If the distributed data has been restructured by the pSelect() method, for example, the result of this method may be incorrect.

C.1.5 to_local()

```
void to_local(DataInfo *info, int global, int *rank,
              int *local);
```

Parameters
(1) *info*: a pointer to a structure representing the distribution of the vector being indexed from by this method.
(2) *global*: an integer representing the index of the global vector being addressed at by this method.
(3) *rank*: an output integer representing the rank of the processor containing the global data at index == global.
(4) *local*: an output integer representing the index into the processors data (identified by rank) in which the data value at index==global (in the global vector) can be found.

Returns
void

This method calculates both the rank of the processor that holds the data value corresponding to the global parameter, and its local index into its distributed portion of the data where it can be found. This method is intended to be used exclusively by the implementation itself in the hSplit() and sMerge() methods. If the distributed data has been restructured by the pSelect() method, for example, the result of this method may be incorrect.

C.1.6 partition()

```
void partition(DataInfo *part);
```

Parameters
(1) *part*: a pointer to a structure to be updated with various data distribution information.
Returns
void

This method constructs a simple partitioning map over all available processors of the currently non-distributed data described by the part parameter. If np divides the length of the vector then there is perfect load balance and all processors have equally sizes chunks of data. Otherwise, there is load-imbalance and each processor gets allocated (length/np)+1 data values except the last processor (np-1), which is allocated a (smaller) portion of the remaining data. In cases where ((length/np)+1)*np > length, the allocation is length/np for all processors except the last processor (np-1), which is allocated the (larger) remaining portion of the remaining data.

C.1.7 nSplit()

```
void nSplit(DataInfo *info);
```

Parameters
(1) *info*: a pointer to a structure to be used in splitting the data specified by it among all available processors.
Returns
void

This method performs a simple distribution of a currently non-distributed input vector described by the info parameter. The processor with rank==0 asynchronously sends the data to all available processors. The size of the distributed portions being received by the processors was calculated when the initialise() method called the partition() method. Barrier synchronisation is performed before returning control to the calling program.

C.1.8 hSplit()

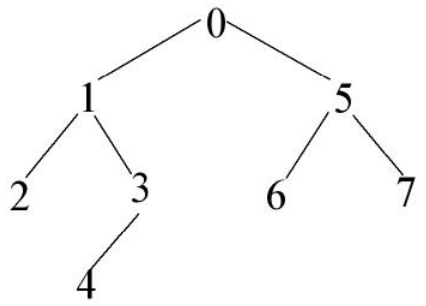
```
void hSplit(DataInfo *info);
```

Parameters
(1) *info*: a pointer to a structure to be used in splitting data among all available processors. The structure will be updated with some splitting information.
Returns
void

This method performs a hierarchical distribution of the currently non-distributed input vector described by the info parameter. The communication is conducted in a binary tree with processors at the nodes. Consider the following example:

Vector = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16];
Processors = 8;

Then the following communication tree would be constructed:



An 8 processor split hierarchy

This means that the processor with rank==0 sends portions of the non-distributed structure described by the input parameter (info) to processors 1 and 5 (its child nodes). From this point the processors located in the left and right hand sub-trees proceed with communication *parallel*. I.e. processors 1 and 5 become masters of their respective sub-trees, independently continuing the split operation[9]. Each processor records its master and slaves (if any) for later use by reduction operations[10]. Barrier synchronisation is performed before returning control to the calling program.

[9] This algorithm is performed by helper function called splitData(), which takes an integer indicating the number of processors to be involved in the split operation, in the case of hierarchical split all processors are utilised.
[10] The recording of master and slaves at each processor (node) means the *implicit* processor hierarchy (figure 7) is recorded. This means reduction operations can use this tree to evaluate sub-tree reduction results (in parallel) from the bottom up.

This method of splitting is superior, in terms of performance, to the card-dealing style performed by the nSplit() operation. Note that parallelism increases as the algorithm progresses.

C.1.9 sMerge()

```
void* sMerge(DataInfo *info);
```

Parameters
(1) *info*: a pointer to a structure to be used in splitting data among all available processors. The structure will be updated with some splitting information.
Returns
A pointer to array of values representing all processors data combined into a single array, the result will reside on the processors node with rank==0

This method provides a simple means of re-coalescing a distributed vector into the processor with rank==0. This method basically just uses the inverse of the algorithm used in the nSplit() method. The processor with rank==0 asynchronously receives the distributed data from all other processors, waits for all communication to complete and returns a pointer to a memory location where the re-coalesced results are stored. Barrier synchronisation is performed before returning control to the calling program.

Future versions of this function should return this rule as a ListInfo structure, so that the client program can access type and length details about the returned list.

C.1.10 operateI()

```
int operateI(int op, int left, int right);
```

Parameters
(1) *op*: an integer representing the operation to be performed using the left and right operand parameters; allowable operations are sum, product, min and max.
(2) *left*: an integer representing the left operand of the operation to be performed.
(3) *right*: an integer representing the right operand of the operation to be performed.
Returns
An integer value representing the result of the operation.

This method is used by various reduction operations in the implementation, such as myReduceI(). The op parameter has the same semantics as in the applyI() method. This method therefore returns the result of the input operation applied to the operands left and right. For example:

left = 40;
right = 60;
operateI(SUM,left,right) == 100;

The purpose of this method is to update a result received from another processor with the locally computed result.

C.1.11 operateD()

```
double operateD(int op, double left, double right);
```

Parameters

(1) *op*: a integer representing the operation to be performed using the left and right operand parameters; allowable operations are sum, product, min and max.
(2) *left*: a double value representing the left operand of the operation to be performed.
(3) *right*: a double value representing the right operand of the operation to be performed.
Returns
A double value representing the result of the operation.

This method executes the same algorithm as used the operateI() method. However it operates on and returns a double value, instead of an integer.

C.1.12 seqReduceI()

```
int seqReduceI(ListInfo *info, int operation);
```

Parameters
(1) *info*: a pointer to a structure describing the vector to be used in computing the sequential reduction.
(2) *op*: an integer representing the reduce operation to be applied to the input vector; allowable operations are sum, product, min and max.
Returns
An integer value representing the result of the operation.

This method basically computes a local reduction result, which is either called directly by a client program or called by implementation methods such as fmyReduce(). The computation operates on info.list. This method assumes that the memory pointed to by info.list contains info.length integers. For example, if the operation parameter is equal to SUM then this method returns the sum of the integers in the info.list vector.

C.1.13 seqReduceD()

```
double seqReduceD(ListInfo *info, int operation);
```

Parameters
(1) *info*: a pointer to a structure describing the vector to be used in computing the sequential reduction.
(2) *op*: an integer representing the reduce operation to be applied to the input vector; allowable operations are sum, product, min and max.
Returns
A double value representing the result of the operation.

This method executes the same algorithm as used the seqReduceI() method. However, it operates on vectors of doubles and returns a double result.

C.1.14 seqScanI()

```
void seqScanI(ListInfo *info, int initial,
              int operation);
```

Parameters
(1) *info*: a pointer to a structure describing the non-distributed vector to be used in computing the sequential scan.
(2) *op*: an integer representing the scan operation to be applied to the input vector; allowable operations are sum, product, min and max.
Returns
void

This method basically computes a local reduction result, which is either called directly by a client program or called by implementation methods such as myScanI(). The computation operates on info.list. This method assumes that the memory pointed to by info.list contains info.length integers. For example, if the operation parameter is equal to SUM then this method performs an accumulative sum operation on the info.list vector (and updates it). The following example provides some pseudo-code on execution of the seqScanI() method:

vector = [1,2,3,4,5,6,7,8,9,10];
seqScanI(vector,0,SUM);
vector = [1,3,6,10,15,21,28,36,45,55];

C.1.15 seqScanD()

```
void seqScanD(ListInfo *info, double initial,
              int operation);
```

Parameters
(1) *info*: a pointer to a structure describing the vector to be used in computing the sequential scan.

(2) *op*: an integer representing the scan operation to be applied to the input vector; allowable operations are sum, product, min and max.
Returns
void

This method executes the same algorithm as used the seqScanI() method. However, it operates on vectors of doubles.

C.1.16 seqZip()

```
ListInfo seqZip(ListInfo *left, ListInfo *right);
```

Parameters
(1) *left*: a pointer to a structure describing the non-distributed vector to be used in computing the sequential zip.
(2) *right*: a pointer to a structure describing the non-distributed vector to be used in computing the sequential zip.
Returns
void

This method inputs two non-distributed vectors of arbitrary values (both vectors of the same type). Corresponding elements from the two vectors are combined into a single vector of pairs, and the resulting non-distributed vector is returned (encapsulated in a ListInfo structure). If the input vectors have different lengths, the longer ones extra values are ignored.

C.1.17 seqRepeat()

```
ListInfo seqRepeat(void *value, int tSize, int copies);
```

Parameters
(1) *value*: a pointer to the value to be sequentially repeated.
(2) *tSize*: an integer describing the type size of the input value.
(3) *copies*: an integer describing the number of times the value will repeat.
Returns
void

This method inputs an arbitrary non-distributed value and an integer==copies, and produces a non-distributed vector of length==copies where each element is a copy of the input value. The result is returned to the calling program encapsulated in a ListInfo structure.

C.1.18 seqDistl()

```
ListInfo seqDistl(void *value, int tSize,
                  ListInfo *vector);
```

Parameters
(1) *value*: a pointer to the value to become the first element in the output vector of pairs.
(2) *tSize*: an integer describing the type size of the input value.
(3) *vector*: a pointer to a structure describing the vector of values which will become the second elements of the output vector.
Returns
void

This method inputs an arbitrary non-distributed value and a non-distributed vector and returns a non-distributed vector of pairs where the first elements are a copy of the input value and the second elements are elements of the input vector. The result is returned to the calling program encapsulated in a ListInfo structure.

C.1.29 myReduceI() (/)

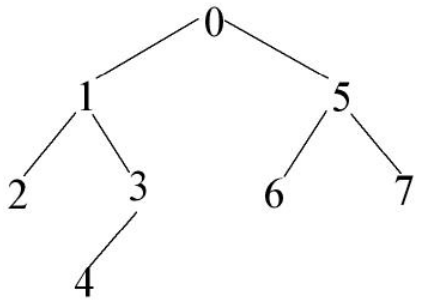
Definition
$\oplus/[x_0,x_1,x_2,.....,x_{n-1}] = x_0 \oplus x_1 \oplus x_2 .... \oplus x_{n-1}$

```
int myReduceI(DataInfo *info, int operation, int local);
```
[11]

Parameters
(1) *info*: a pointer to a structure describing the distributed vector relating to this reduction operation.
(2) *operation*: an integer representing the reduce operation currently being performed; allowable operations are: sum, product, min and max.
(3) *local*: an integer representing this processors already-computed local result in relation to the reduce operation being performed.
Returns
An integer representing the result of this reduction operation.

This method provides a custom implementation of the MPI_Reduce operation (see reduce() in appendix C.1). It inputs the type of operation being performed and the already computed local result and returns the result of the reduction to the processor with rank==0. The algorithm used hinges on the split hierarchy information recorded during the previously executed hSplit() method. Repeating the diagram from hSplit():

[11] This method only operates on distributed vectors of integer values, another method called myReduceD() operates on distributed vectors of double values (see appendix C.1).



An 8 processor split hierarchy (again)

Given this split hierarchy, the algorithm is obvious. All processors (at each node) asynchronously receive a result from their child processors. Upon receiving a result, processors call the operateI() method to obtain an updated local result, and again upon receiving the second result (if any). All processors (except the one with rank==0) then send the updated result to their master processor (node above). When all communication is completed the result will reside on the node (processor) with rank==0 (at the top of the tree). This algorithm has significant performance advantages because much of the communication (and computation) is carried out in *parallel*. Note that the parallelism decreases as the algorithm progresses.

A custom implementation of MPI_Reduce() has been provided for two reasons: to compare its efficiency with the MPI native method and because MPI_Reduce() does *not* provide a concatenate binary operation[12]. The concatenate operation is required because the parallel BMF construct P_reduce (which this method is implementing) allows an operator called B_conc. MPI allows custom operator/type definitions to cater for operations other than provided but it seems to enforce a restriction with makes a concatenate operation difficult define. Consider the following prototype definitions used to provide a new MPI_Op:

```
typedef void MPI_User_function(void *invec,
                               void *outvec,
                               int *len,
                               MPI_Datatype *datatype);
int MPI_Op_Create(MPI_User_function *function,
                  int commute,
                  MPI_Op *op);
```

These prototype definitions seem to enforce that both the invec and outvec must occupy the same amount of memory. Defining a concatenate operation using MPI_User_function is not possible with this restriction, and therefore this implementation provides a custom function to perform it.

[12] A method called reduceConcat() actually implements reduce with the concatenate operation, see appendix C.1.

C.1.20 myReduceD() (/)

Definition

$\oplus/[x_0,x_1,x_2,.....,x_{n-1}] = x_0 \oplus x_1 \oplus x_2 .... \oplus x_{n-1}$

```
double myReduceD(DataInfo *info, int operation,
                 double local);
```

Parameters

(1) *info*: a pointer to a structure describing the distributed vector relating to this reduction operation.
(2) *op*: an integer representing the reduce operation currently being performed; allowable operations are: sum, product, min and max.
(3) *local*: a double value representing this processors already-computed local result in relation to the reduce operation being performed.

Returns

A double value representing the result of this reduction operation.

This method executes the same algorithm as used in the myReduce() method. However, it operates on a distributed vector of double values rather than integers.

C.1.21 reduceConcat() (/)

```
ListInfo reduceConcat(DataInfo *info);
```

Parameters

(1) *info*: a pointer to a structure describing the distributed vector relating to this reduction operation.

Returns

A pointer to array of values representing all processors data combined into a single array, the result will reside on the processors node with rank==0.

This method provides an implementation of reduce with concatenate. This method has been specialised (re-named) because reduce(concat) is the only form of reduce which need operate on data of arbitrary type. The general algorithm is closely related to that used in the myReduceI() method, with a few necessary changes.

All processors reallocate their distributed portion of data to accommodate for their slaves data (because a larger portion of memory will be needed upon receiving their slaves data). Processors with slaves synchronously receive data into appropriate memory location(s) (updated in-place). Once all processors have received any data from their slaves, they send this updated data to their master (if any). This will propagate partial results up the processor hierarchy until the result resides on the processor with rank==0. Note that the parallelism

decreases as this algorithm progresses. Further, calling reduceConcat() is semantically equivalent to performing the inverse of a hierarchical split operation, i.e. a hierarchical merge.

C.1.22 fmyReduceI() (/)

```
int fmyReduceI(DataInfo *info, int op);
```

Parameters
(1) *info*: a pointer to a structure describing the distributed vector relating to this reduction operation.
(2) *op*: an integer representing the reduce operation currently being performed; allowable operations are: sum, product, min and max.
Returns
An integer representing the result of this reduction operation.

The myReduceI() method provides a custom implementation of the MPI_Reduce operation. This method provides an alternative algorithm that is superior in both performance and intuitiveness to the myReduceI() method. The reason being that this method (fmyReduceI()) takes advantage of latency hiding techniques by overlapping the computation of the local reduction with the communication required to propagate the results up the processor hierarchy. To be more precise, each processor executes the seqReduceI() method while waiting for the result from its slave(s) to arrive. All other aspects of the algorithm are identical to that given in the description of the myReduceI() method. This method is clearly more intuitive because a client program need only provide the distributed vector and the operation to be performed without the necessity to perform the local reduction itself.

C.1.23 fmyReduceD() (/)

```
double fmyReduceD(DataInfo *info, int op);
```

Parameters
(1) *info*: a pointer to a structure describing the distributed vector relating to this reduction operation.
(2) *op*: an integer representing the reduce operation currently being performed; allowable operations are: sum, product, min and max.
Returns
A double value representing the result of this reduction operation.

This method executes the same algorithm as used in the fmyReduceI() method. However, it operates on a distributed vector of double values rather than integers.

C.1.24 reduce() (/)

```
void* reduce(DataInfo *info, void *local, int type,
             int operation);
```

Parameters
(1) *info*: a pointer to a structure describing the distributed vector relating to this reduction operation.
(2) *local*: a pointer this processors already-computed local result in relation to the reduce operation being performed.
(3) *type*: an integer representing the type of the local parameter; allowable types are Integer, Double, IntegerTuple and DoubleTuple.
(4) *operation*: an integer representing the reduce operation currently being performed; allowable operations are: sum, product, min, max, min_loc, max_loc.
Returns
A pointer to a value representing the result of this reduction operation.

This method uses the MPI_Reduce operation to perform the specified reduction operation on the specified distributed vector. The type parameter allows for both allocation of the result and mapping to the appropriate MPI_Datatype to be passed to the MPI_Reduce operation. A client program can perform any of the sum, product, min, or max operations by first calling the seqReduceI() or seqReduceD() methods and providing the address of the result as the local parameter to this method. A local reduction result using the {min/max}_loc operations can be provided in the following steps.

Using the example of min_loc on a vector of integers:
(1) int min = seqReduceI(&(list.mydata), Min);
(2) integerPair local = { min, rank };

The address the local variable can then be directly passed to this method.

C.1.25 myScanI() (//)

Definition
$\oplus//[x_0,x_1,x_2,.....,x_{n-1}] = [x_0,x_0 \oplus x_1,....,x_0 \oplus x_1 \oplus ... \oplus x_{n-1}]$

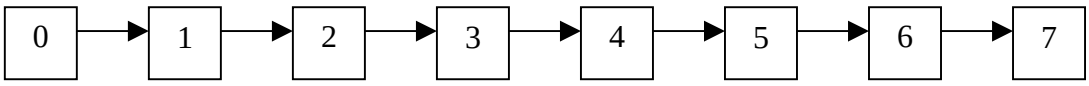
```
DataInfo myScanI(DataInfo *info, int operation);
```

Parameters
(1) *info*: a pointer to a structure describing the distributed vector relating to this scan operation.
(2) *op*: an integer representing the scan operation currently being performed; allowable operations are: sum, product, min and max.

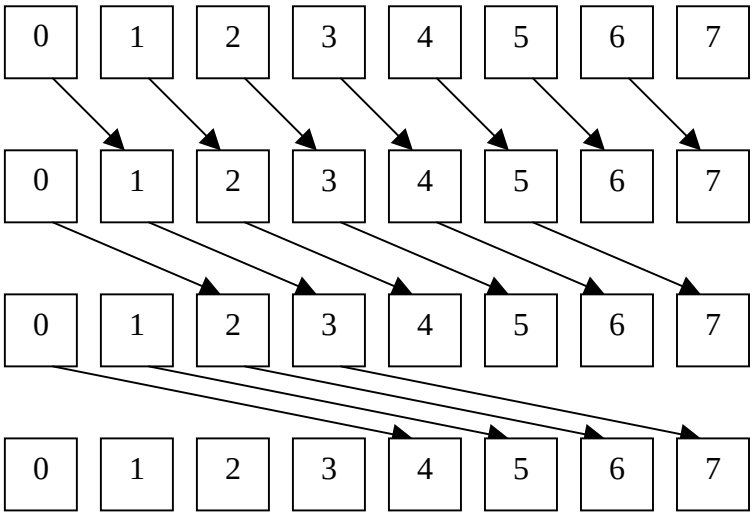
Returns
A pointer to a structure describing the distributed vector produced by this scan operation.

This method provides a custom implementation of the MPI_Scan operation (see scan() appendix C.1). It inputs a scan operation type and a distributed vector describing a list to perform the operation on. The processor with rank==0 starts by performing the seqScanI() method on its portion of the distributed data. Rank==0 then starts the communication by sending its total accumulated value to the processor with rank==1. All other processors synchronously receive an accumulated value from their neighbor (processor rank-1), execute the seqScanI() method with initial value==neigbours value, and synchronously send their total accumulated value (except rank==np-1) to the next processor (rank+1). When all communication is finished scan results of the input operation lie on each processor node, which are updated with the input distributed vector. Barrier synchronisation is performed before returning the results (and control) to the calling program.

The above algorithm will work ok for small numbers of processors, a second (well known) algorithm has also been implemented. The first Figure below shows the pattern of communication required for the simple algorithm, and the second shows the pattern for the second algorithm.



Parallel Scan 1



Parallel Scan 2

The first algorithm (figure 8) requires np-1 distinct messages, 7 messages if np==8. The second algorithm (figure 9) has ceil(log(np)/log(2)) steps. Each processor sends a message to the processor with rank==myrank+2^(step-1). However, all communication in each step is carried out in *parallel* which means there are really only 3 messages when np==8. The general case should therefore see the second algorithm outperform the first.

C.1.26 myScanD() (//)

```
DataInfo myScanD(DataInfo *info, int operation);
```

Parameters
(1) *info*: a pointer to a structure describing the distributed vector relating to this scan operation.
(2) *op*: an integer representing the scan operation currently being performed; allowable operations are: sum, product, min and max.

Returns
A pointer to a structure describing the distributed vector produced by this scan operation.

This method executes the same algorithm as used in the myScanI() method. However, it operates on a distributed vector of double values rather than integers.

C.1.27 scanConcat() (//)

```
DataInfo scanConcat(DataInfo *info);
```

Parameters
(1) *info*: a pointer to a structure describing the distributed vector relating to this scan operation.
Returns
A pointer to a structure describing the distributed vector produced by this scan operation.

This method provides an implementation of scan with concatenate. This method has been specialised (re-named) because scan(concat) is the only form of scan that need operate on data of arbitrary type. The general algorithm is closely related to that used in the myScanI() method, with a few necessary changes.

All processors reallocate their portion of the distributed data to accommodate for all data contained on processors with lesser ranks. All processors then move their data in memory to the right to allow space for the received data because this data physically belongs before this processors data. The processor with

rank==0 starts the communication by sending its portion of the distributed data to the processor with rank==1. All other processors synchronously receive the accumulated data from their neighbor (processor rank-1) into the appropriate memory location (updated in-place). All processors then synchronously send their updated (accumulated) data to the next processor (processor rank+1). This process leaves accumulated data vectors on each processor node, which together forms a (new) distributed vector. Barrier synchronisation is performed before returning the results (and control) to the calling program.

For clarity, below is an example of a scan operation with concatenate (i.e. accumulateData) over a vector of integers (in pseudo code):

processors = 4;
distributed vector = [[1], [2], [3], [4]];
scanConcat() = [[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]];

In this example the initial vector was distributed with one value per processor. After the execution of accumulateData() each processor also contains the data values from processors with lesser ranks.

C.1.28 scan() (//)

```
DataInfo scan(DataInfo *info, int rtype, int operation);
```

Parameters
(1) *info*: a pointer to a structure describing the distributed vector relating to this scan operation.
(3) r*type*: an integer representing the return type of this scan operation; allowable types Integer, Double, IntegerTuple and DoubleTuple.
(4) *operation*: an integer representing the scan operation currently being performed; allowable operations are: sum, product, min, max, min_loc, max_loc along with any other MPI operations.
Returns
A pointer to a structure describing the distributed vector produced by this scan operation.

This method uses the MPI_Scan operation to perform the specified scan operation on the specified distributed vector. The rtype parameter allows for both allocation of the result and mapping to the appropriate MPI_Datatype to be passed to the MPI_Scan operation. First, all processors perform the seqReduce() method (based on rtype and operation) to compute their local accumulated result. Second, all processors participate in the MPI_Scan operation to obtain a global accumulated results. Third, all processors execute the seqScan() method with initial value==the ‘difference’ between the local accumulated result and the global accumulated result. Here, ‘difference’ depends on the operation currently being performed:

1. SUM => initial = abs(local–global)
2. PROD => initial = max(local,global) / min(local,global)

If the operation is MIN_LOC or MAX_LOC, all processors create a list of the global accumulated value (all elements), using the seqRepeat() method.

As with the myScanI() method, the input distributed vector==info is updated and the results can be accessed using the mydata component of the returned distributed vector. All processors participate in barrier synchronisation before returning control to the calling program.

C.1.29 zip() (γ)

Definition

γ ($[x_0,x_1,\ldots,x_n],[y_0,y_1,\ldots y_k]$) = $[(x_0,y_0),(x_1,y_1),\ldots]$

```
DataInfo zip(DataInfo *left, DataInfo *right);
```

Parameters

(1) *left*: a pointer to a structure describing a distributed vector.
(2) *right*: a pointer to a structure describing a distributed vector.

Returns

A pointer to a structure describing the resultant distributed vector after the application of this zip operation.

This method inputs two distributed vectors containing arbitrary data-values (of the same type) and combines corresponding elements from each to form a (new) distributed vector of pairs. The corresponding elements must lie on the same processor node.

A new distributed vector is first created. All processors then allocate for their new portion of the distributed data (knowing the type sizes). All processors then iterate through their portion of the distributed data creating a new pairs with copies of the corresponding values from the distributed vectors. If the distributed vectors have different lengths the excess values from the longer one are disregarded.

C.1.30 repeat()

Definition

repeat(a,p) = [a,a,a,a,…,a] = a vector with p copies of a

```
DataInfo repeat(void *value, int tSize, int copies);
```

Parameters

(1) *value*: a pointer to a non-distributed value.
(2) *tSize*: an integer representing the size in bytes of the type of value.

(3) *copies*: an integer specifying the number of times value is to be repeated.
Returns
A pointer to a structure describing the resultant distributed vector after the application of this repeat operation.

This method takes a non-distributed value==value and an integer p==copies and forms a (new) distributed vector over p processors where each processors portion of the distributed data is a copy of the value. This method is useful for easily creating distributed vectors where all elements are the same.

The processor with rank==0 first creates a global vector of length p==copies where each element is a copy of the non-distributed value. All processors then execute the splitData() method using only p==copies processors, resulting in a (new) distributed vector. All processors then participate in barrier synchronisation before returning control to the calling program. Note: the splitData() method is the same method called by hSplit() to perform a hierarchical split operation. However, hSplit() utilises all available processors instead of just p==copies.

The repeat() method uses the splitData() method to get the resultant distributed vector for two reasons. The first is obviously because the splitData() method is already implemented, resulting in code reuse. The second is because a calling program may later call the reduceConcat() method to re-coalesce its data. The hMerge() method makes heavy use of split hierarchy information recorded during the hSplit() operation, therefore if repeat() does not use splitData() for communicating the input value errors will occur when calling reduceConcat(). The disadvantage to this approach is that if the object to be repeated is very large the processor with rank==0 will perform lots of memory copying.

C.1.31 distl()

Definition
distl $(x,[y_0,y_1,\ldots,y_n]) = [(x,y_0),(x,y_1),\ldots,(x,y_n)]$

```
DataInfo distl(void *x, int tSize, DataInfo *y);
```

Parameters
(1) *x*: a pointer to a non-distributed value.
(2) *tSize*: an integer representing the size in bytes of the type of x.
(3) *y*: a pointer to a structure describing a distributed vector.
Returns
A pointer to a structure describing the resultant distributed vector after the application of this distl operation.

This method takes a non-distributed value==x and a distributed vector==y and produces a distributed vector of pairs $(\prod_1, \prod_2)$ where $\prod_1$ is a copy of x and $\prod_2$

is the element of y originally occupying that node. This method is clearly useful for broadcasting a non-distributed value over a distributed vector.

This implementation of distl has three phases. Phase one: a copy of the distributed vector==y is made and updated with information required to accommodate the vector of pairs, which involves all processors. Phase two: the non-distributed value==x is efficiently broadcasted to all processors involved in the distribution of the distributed vector==y by using the processor split hierarchy recorded during the splitting of that vector (see the hSplit() method). And finally, phase three: all processors iterate through their portion of the (new) distributed data combining x and y_i into pairs ($\prod_1$, $\prod_2$), where y_i denotes elements of the copied distributed vector from phase two. All processors then participate in barrier synchronisation before returning control to the calling program.

C.1.32 pSelect()

Definition
select (sv,[i_0,i_1,…,i_{p-1}] = [sv!i_0,sv!i_1,…,sv!i_{p-1}]

```
DataInfo pSelect(DataInfo *sv, int indexes[], int size);
```

Parameters
(1) *indexes*: a non-distributed vector of integers describing how the distributed vector sv is to be re-arranged.
(2) *size*: the size of the indexes vector.
(3) *sv*: a pointer to a structure describing the distributed vector to be re-arranged.
Returns
A pointer to a structure describing the resultant distributed vector after the application of this select operation.

This method takes a distributed source vector==sv and a non-distributed index vector==indexes and produces a distributed vector==sv re-arranged by the non-distributed index vector. Consider the following example:

```
processors = 4;
dist = [[1, 2], [3, 4], [5, 6], [7, 8]];  //nesting indicates distribution
ind = [1, 0, 3, 2];
```

Before the select operation, the machine state will contain:

```
processor 0: [1, 2]
processor 1: [3, 4]
processor 2: [5, 6]
processor 3: [7, 8]
```

After the execution of the select operation with distributed vector==dist and indexes==ind, the machine state will contain:

processor 0: [3, 4]
processor 1: [1, 2]
processor 2: [7, 8]
processor 3: [5, 6]

So the index vector is a specification on how the distributed data specified by the vector==sv should be re-arranged on the machine.

This parallel select operation requires a subset of the functionality of non-uniform many-to-many personalised communication, and therefore a general communication pattern is required to avoid deadlock problems. The processor with rank==0 first broadcasts the index vector to all other processors, giving them a local copy. Each processor then executes the following pseudo code:

```
for proc in indexes loop
     if (rank==proc) and (rank!=indexes[proc]) then
          syncRecv(indexes[proc],recvbuff);
     else if (rank==indexes[proc]) and (rank!=proc) then
          syncSend(proc,mydata);
     end if;
end loop;
```

Each processor iterates through the indexes array and at index==rank that processor will receive data from the processor with rank==indexes[proc]. If this processors rank==indexes[proc] then this processor must send its data to the processor with rank==proc. Furthermore, processors make sure they do not try to send or receive data to/from themselves. After executing this loop all processors participate in barrier synchronisation before returning control to the calling program.

The performance of this algorithm is highly dependant on the re-distribution index array. The worst case is when a single processor must send its data to every other processor. The best case is when all processors send their data to their neighboring processors, during a shift operation. Parallel select is useful for distributed data re-arrangements.

Appendix D

D.1 Example Programs

D.1.1 C/MPI code for remote.Adl

```
#include "ParallelConstructs.h"
int main(int argc, char **argv){
  int i,j;
  int length = atoi(argv[1]);
  int dummy = start(argc,argv);
  int *input = makeInput(length); //initialise list
  DataInfo distrib =
initialise(&input[0],sizeof(int),length);
  int mysize = distrib.mydata.length;
  void *distled;
  int *mappedif;
  int *reduced;
  DataInfo repeated;
  DataInfo zipped;
  distled =
     calloc(mysize,(sizeof(pair)+2*sizeof(int))*length);
  mappedif = (int*)calloc(mysize,length*sizeof(int));
  reduced = (int*)calloc(mysize,sizeof(int));
  //(1)split
  hSplit(&distrib);
  //(2)repeat
  repeated = repeat(&(input[0]),length*sizeof(int),np);
  //(3)zip
  //  ----- hack -----
  distrib.mydata.length = 1;
  distrib.mydata.tSize = distrib.mydata.tSize*mysize;
  //  --- end hack ---
  zipped = zip(&distrib,&repeated);
  {
    //(4)map brepeat
    pair element = *(pair*)(zipped.mydata.list);
    ListInfo rep =
seqRepeat(element.pi2,length*sizeof(int),mysize);
    (*((pair*)zipped.mydata.list)).pi1 = element.pi1;
    (*((pair*)zipped.mydata.list)).pi2 = rep.list;
    zipped.mydata.tSize =
     sizeof(pair)+mysize*sizeof(int)+rep.tSize*mysize;
  }
  {
    //(5)map bzip
    pair element = *(pair*)(zipped.mydata.list);
    ListInfo left,right,dist;
    left.list = element.pi1;
```

```
    left.tSize = sizeof(int);
    left.length = mysize;
    right.list = element.pi2;
    right.tSize = length*sizeof(int);
    right.length = mysize;
    dist = seqZip(&left,&right);
    zipped.mydata.list = dist.list;
    zipped.mydata.tSize = dist.tSize;
    zipped.mydata.length = mysize;
  }
  {
    //(6)map bmap bdistl
    int newtSize = (sizeof(pair)+2*sizeof(int))*length;
    for(i = 0; i < zipped.mydata.length ; i++){
       void *addr = addrOf(newtSize,distled,i);
       pair element =
*(pair*)addrOf(zipped.mydata.tSize,zipped.mydata.list,i);
       int value = *(int*)element.pi1;
       ListInfo vector,result;
       vector.list = element.pi2;
       vector.tSize = sizeof(int);
       vector.length = length;
       result = seqDistl(&value,sizeof(int),&vector);
       addr =
     memcpy(addr,result.list,result.tSize*result.length);
    }
    zipped.mydata.list = distled;
    zipped.mydata.tSize = newtSize;
    zipped.mydata.length = mysize;
  }
  {
    //(7)map bmap ifstatement
    int newtSize = length*sizeof(int);
    int upto = 0;
    for(i = 0; i < zipped.mydata.length ; i++){
       void *pairlist =
addrOf(zipped.mydata.tSize,zipped.mydata.list,i);
       for(j = 0; j < length ; j++){
          pair element =
*(pair*)addrOf(sizeof(pair)+2*sizeof(int),pairlist,j);
          int left = *(int*)(element.pi1);
          int right = *(int*)(element.pi2);
          int result = left-right;
          if (result < 0)
             mappedif[upto] = abs(result);
          else
             mappedif[upto] = result;
          upto++;
       }
    }
    zipped.mydata.list = (void*)&(mappedif[0]);
```

```
    zipped.mydata.tSize = newtSize;
    zipped.mydata.length = mysize;
  }
  {
    //(8)map bmap breduce(+)
    int newtSize = sizeof(int);
    for(i = 0; i < zipped.mydata.length ; i++){
      void *addr =
addrOf(zipped.mydata.tSize,zipped.mydata.list,i);
      ListInfo toreduce;
      toreduce.list = addr;
      toreduce.tSize = sizeof(int);
      toreduce.length = length;
      reduced[i] = seqReduceI(&toreduce,SUM);
    }
    zipped.mydata.list = (void*)&(reduced[0]);
    zipped.mydata.tSize = newtSize;
    zipped.mydata.length = mysize;
  }
  {
    //(9)reduce conc
    ListInfo result = reduceConcat(&zipped);
  }
  finish();
  return 0;
}
```

Appendix E

E.1 Efficiency Tests

This section shows some of the other efficiency tests (other vector lengths) conducted on all parallel (or otherwise) constructs implemented.

	1	2	4	8	16	32	64
MPI_Barrier()	0.000003	0.001143	0.004142	0.006911	0.007869	0.052521	0.032514
nSplit()	0.000067	0.002038	0.002794	0.022503	0.008194	0.01199	0.101452
hSplit()	0.000008	0.001969	0.03702	0.00517	0.007616	0.011428	0.083078
sMerge()	0.000085	0.000721	0.000796	0.000932	0.001086	0.001498	0.002343
fmyReduceI()	0.000027	0.000139	0.000222	0.000337	0.000432	0.000547	0.000665
myReduceI()	0.000027	0.000138	0.000224	0.000344	0.000433	0.000539	0.00067
reduceConcat()	0.000007	0.000699	0.001321	0.001657	0.001855	0.002198	0.002525
Reduce()	0.000028	0.000079	0.000109	0.00016	0.000219	0.000254	0.000303
myScanI()	0.000043	0.000154	0.000362	0.000698	0.001406	0.002727	0.005362
scanConcat()	0.000006	0.000718	0.00189	0.004184	0.00889	0.017128	0.034635
scan()	0.000914	0.000706	0.000429	0.000342	0.000348	0.000393	0.000467
zip()	0.002383	0.001279	0.000709	0.000431	0.000293	0.000253	0.000287
repeat()	0.000107	0.002928	0.00451	0.010664	0.031116	0.064627	0.128036
distl()	0.001559	0.001089	0.000774	0.000628	0.000676	0.000777	0.00086
pSelect()	0.000005	0.000759	0.001152	0.000674	0.001016	0.000869	0.001073

Table 4: All constructs; length=1000000

	1	2	4	8	16	32	64
MPI_Barrier()	0.000007	0.098035	0.14474	0.185089	0.186763	0.252978	0.203599
nSplit()	0.009671	0.163631	0.21515	0.242985	0.260588	0.265572	0.272595
hSplit()	0.000013	0.161016	0.279784	0.299179	0.340364	0.351073	0.439814
sMerge()	0.009501	0.068817	0.07027	0.067163	0.067903	0.068703	0.073958
fmyReduceI()	0.002789	0.002129	0.001381	0.000985	0.000786	0.000719	0.000761
myReduceI()	0.002736	0.002128	0.001434	0.001042	0.000766	0.000725	0.000731
reduceConcat()	0.000008	0.059201	0.094033	0.113442	0.134356	0.149553	0.158847
reduce()	0.002738	0.002027	0.001246	0.00081	0.000528	0.000409	0.000358
myScanI()	0.004488	0.004659	0.004783	0.004906	0.005297	0.006446	0.008985
scanConcat()	0.000009	0.06244	0.1508	0.327681	0.698456	1.402378	2.854036
scan()	0.091995	0.051212	0.026313	0.013566	0.006705	0.00357	0.002064
zip()	0.238677	0.124238	0.067726	0.033532	0.016871	0.008492	0.004277
repeat()	0.009734	0.250721	0.578145	1.231322	2.660968	5.444215	11.35441
distl()	0.158276	0.084156	0.056427	0.028267	0.013059	0.007269	0.003974
pSelect()	0.000006	0.121002	0.086666	0.042758	0.033021	0.019243	0.012721

Table 5: All constructs; length=1000000