Parallel Functional Programming by Partitioning

Frank Stephen Taylor

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Engineering of the University of London and the Diploma of Imperial College, 10th September, 1996
Abstract

Caliban is a declarative language which addresses the area of static distributed memory parallel computing. It is an annotation language that allows the programmer to partition a functional program and data amongst the computational resources available. It is integrated into the source language so that the full power of the host language can be used to express the partitioning of the program. Partial evaluation is used to determine a complete version of the annotation at compile time. Program transformation is then used to make the parallelism explicit.

This thesis describes the Caliban language and its pilot implementation. It then continues by presenting extensions and improvements to the basic language. Implementation techniques for the improved language are discussed in relation to an implementation on the Fujitsu AP1000 distributed memory multiprocessor computer. Two application case studies together with some performance results are presented. Finally, there is a critical appraisal of the language and its approach.

Caliban has good support for general data and computation partitioning. It also aids software reuse with its ability to abstract common computational structures into higher order forms which are concretised at compile time by partial evaluation. However, there do remain some open issues relating to evaluation order control. Finally, Caliban can be implemented reasonably efficiently on standard parallel hardware.
Acknowledgements

First and foremost I would like to thank Paul Kelly, my supervisor, for his encouragement, support and direction. It was his work that formed the genesis of the work presented here and it is with his supervision that I have journeyed through the world of parallel programming as an undergraduate, research assistant and PhD student. I also thank my second supervisor Susan Eisenbach for her help.

Many thanks go to the members of the Advanced Languages and Architectures Section for providing a stimulating and friendly environment to work in. Also my work with the Imperial College/Fujitsu Parallel Computing Research Centre has provided not only a parallel machine to work on, but also a view of how parallelism is really used. Many thanks to the centre and its users.

On the technical side, I am indebted to Koen Langendoen, Pieter Hartel and all on the FAST project for the use of their functional language compiler and runtime system. I hope I didn’t ask too many stupid questions.

On the pastoral side, I would like to thank all my musical friends for the opportunities to make music together. There are far too many of them to name, but they should know who they are. Music has helped to keep me sane throughout all my work!

Finally, there are three people to whom I offer my deepest heartfelt thanks and love; my parents who have given me support, both fiscal and moral, during my studies, and to Ruth who has been with me all through my PhD and is now my wife.

This work was supported by an EPSRC research studentship, number 92R/0003633.
Contents

1 Introduction ......................................................... 17
   1.1 Contribution of this thesis ................................. 18
       1.1.1 Statement of originality .......................... 19
   1.2 Thesis Layout ............................................... 20
   1.3 Literature Survey ........................................... 21
       1.3.1 Parallelism ........................................... 22
       1.3.2 Parallelism: Hardware ............................ 27
       1.3.3 Parallelism: Software ............................ 31
       1.3.4 Conclusions ........................................ 47
   1.4 Caliban ....................................................... 49

2 Basic Caliban ...................................................... 51
   2.1 The Caliban Philosophy .................................... 52
   2.2 Basic Caliban ............................................... 53
       2.2.1 Building up networks ............................. 57
       2.2.2 Summary of Basic Caliban ....................... 58
       2.2.3 Expanding NFOs .................................. 60
   2.3 Example programs .......................................... 66
       2.3.1 Raytracing ........................................ 71
   2.4 Conclusions ................................................. 79

3 The Implementation of Basic Caliban .............................. 81
   3.1 Integrating Caliban with HASKELL- ....................... 82
   3.2 Simplification ............................................. 85
3.2.1 The implementation .................................. 87
3.2.2 Simplification Recapitulation ......................... 102
3.3 The Network Extractor ................................... 102
   3.3.1 The Process Placement Rule ....................... 105
   3.3.2 The Details of Network Extraction .................. 105
   3.3.3 The code ......................................... 108
3.4 Runtime System ........................................ 112
   3.4.1 The host runtime system .......................... 113
   3.4.2 The cell runtime system .......................... 115
3.5 Performance Evaluation ................................. 117
3.6 Conclusions ........................................... 119

4 Advanced Caliban .................................... 121
   4.1 An evaluation of Basic Caliban ....................... 122
   4.2 Multiple Output Processes .......................... 123
      4.2.1 Example: Selector functions and friends .......... 123
      4.2.2 Example: Load balancing ........................ 125
      4.2.3 Possible solutions .............................. 128
      4.2.4 Bundles ........................................ 129
   4.3 Building Better NFOs ................................ 134
      4.3.1 Multiple Moreovers ............................. 135
   4.4 More Efficient Processor Usage ...................... 137
      4.4.1 Phasing in Advanced Caliban ..................... 138
      4.4.2 Problems relating to multiple moreovers .......... 139
   4.5 Recapitulation ...................................... 140

5 Implementation of Advanced Caliban ..................... 143
   5.1 The Advanced Caliban Frontend ....................... 143
      5.1.1 Multiple Output Processes ....................... 144
      5.1.2 Multiple moreover clauses ........................ 151
      5.1.3 Phasing ......................................... 167
      5.1.4 Integrating multiple moreover clauses and phasing . 174
5.2 The Advanced Caliban Runtime System .................................. 180
  5.2.1 The software architecture ............................................ 180
  5.2.2 FCG .......................................................... 183
  5.2.3 The AP1000 ...................................................... 186
  5.2.4 The Non-Preemptive Runtime System .............................. 190
  5.2.5 Extensions for preemption ......................................... 201
5.3 Conclusions .......................................................... 202

6 Using Advanced Caliban .................................................. 205
  6.1 Raytracing .......................................................... 205
    6.1.1 Raytracer performance ........................................... 207
  6.2 Jacobi Relaxation .................................................... 223
    6.2.1 The problem .................................................... 223
    6.2.2 Local neighbourhood operations ............................... 225
    6.2.3 Sequential Jacobi Relaxation .................................. 226
    6.2.4 Parallel Jacobi Relaxation ..................................... 229
    6.2.5 Performance of Parallel Jacobi Relaxation ..................... 244
  6.3 Advanced Caliban — An Analysis ..................................... 244
    6.3.1 An Advanced Caliban overview ................................. 245
    6.3.2 Program Structure .............................................. 246
  6.4 Conclusions .......................................................... 254

7 Conclusions ............................................................ 255
  7.1 Summary of thesis .................................................... 255
  7.2 Related Work ....................................................... 261
    7.2.1 HPF ........................................................ 261
    7.2.2 Skeletons ...................................................... 262
    7.2.3 Coordination/Configuration languages .......................... 262
    7.2.4 Jacobi iteration .............................................. 263
  7.3 Contribution of this Thesis ........................................ 264
  7.4 Future Work ......................................................... 265
  7.5 Final Thoughts ...................................................... 266
## List of Figures

2.1 A simple program, with annotation .............................. 54
2.2 A pipeline ...................................................... 55
2.3 A fan-in process network ......................................... 55
2.4 The pipe NFO .................................................... 57
2.5 The fan NFO ...................................................... 57
2.6 A parallel database search process network ...................... 61
2.7 Parallel database search ......................................... 61
2.8 After a simple term-re-writing evaluation ....................... 62
2.9 A pipeline annotated with computational loads ............... 67
2.10 A hand-balanced pipeline ....................................... 67
2.11 chain — apply a sequence of functions to an input ........... 68
2.12 Building and annotating a pipe in one go ..................... 68
2.13 A load balancing pipe NFO .................................... 69
2.14 A binary folding tree .......................................... 70
2.15 The treefold annotation ........................................ 71
2.16 A linear fold NFO ............................................. 71
2.17 Merging using parinsert ........................................ 71
2.18 Partition each element ......................................... 73
2.19 Partition to a given number of contiguous groups ........... 74
2.20 Partition to a given number of groups, round-robin style ..... 74
2.21 A simple two-slave Farm ....................................... 75
2.22 The transformed parallel ray-tracer ............................ 79
3.1 The frontend compiler for the Caliban system .................. 82
3.2 Definitions to integrate Caliban into Haskell.............................. 84
3.3 Definitions for moreover......................................................... 84
3.4 Definition of ANF................................................................. 86
3.5 Simple unsugared λ-calculus.................................................. 88
3.6 Definitions for schemes......................................................... 88
3.7 Simple λ-expression to Term translation................................. 89
3.8 Some useful auxiliary functions used in the schemes............... 92
3.9 The WHNF evaluator............................................................. 93
3.10 Copying for λ-expression instantiation................................. 94
3.11 Instantiation for letrec usage............................................. 95
3.12 The ANF evaluator............................................................... 97
3.13 A simple program to simplify.............................................. 97
3.14 Three snapshots of the simplification process...................... 99
3.15 The simple example program graph after simplification........ 100
3.16 An example procnet call, with generated network................. 103
3.17 Building the process network on the host.......................... 114
3.18 Reading values from a stream............................................. 115
3.19 Output transport processes supplying the output stream...... 117
3.20 Results for a 12×12 image grid with 6 scene objects............... 118

4.1 The required structure of a farm.......................................... 123
4.2 The Basic Caliban code for farm.......................................... 123
4.3 The farm network that results from the code in Figure 4.2........ 124
4.4 An alternative coding of farm in Basic Caliban.................... 124
4.5 The process network generated by Figure 4.4........................ 125
4.6 A simple tree structure....................................................... 125
4.7 A sample use and definition of treefold.............................. 126
4.8 Load balancing a tree by its subtrees.................................. 127
4.9 Load balancing a tree by depth............................................ 127
4.10 A simple program and its network...................................... 130
4.11 A bundled process network and code.................................. 131
4.12 Farming with B
dles ........................................... 132
4.13 Subtree blocking using B
dle for treefold ............................. 133
4.14 Layer-wise grouping for treefold ................................. 133
4.15 The pipe NFO .............................................. 135
4.16 Multiple moreover version of pipe ................................ 136
4.17 Subtree blocking treefold using multiple moreover ............... 136
4.18 Multiple moreover version of farm ................................ 137
4.19 A program with three sections ..................................... 137
4.20 A program with two distinct computations .......................... 139
4.21 Annotating distinct computations as phases ....................... 140
4.22 Advanced Caliban assertions and connectives ...................... 141

5.1 The Bundle extended ANF evaluator ................................. 145
5.2 The Advanced Caliban parallel runtime system interface .......... 148
5.3 Equivalences for moreover clauses .................................. 153
5.4 Meta-annotation extended definitions for programs ................. 155
5.5 A scheme to meta-annotate a program .............................. 157
5.6 Updated versions of A and B for multiple moreover .............. 159
5.7 The searching simplifier scheme (part a) ............................ 160
5.8 The searching simplifier scheme (part b) ............................ 161
5.9 Schemes to drive the searching simplification ....................... 162
5.10 A sample Caliban program using two NFOs .......................... 165
5.11 Equivalences for the With connective .............................. 168
5.12 Two phases .................................................. 169
5.13 The complexity of the interface communication elimination match-
ing algorithm .................................................. 172
5.14 Changes to S' and A required for Annot ............................. 176
5.15 A sample Caliban program using two phased NFOs ............... 177
5.16 Partitions of the program graph ................................... 181
5.17 The layout of the backend compiler ................................ 183
5.18 FCG’s cell layout .............................................. 185
5.19 The cell layout of aggregate data in FCG .......................... 185
5.20 The AP1000 hardware architecture ............................... 187
5.21 The data structures used by the runtime system to manage com- 
munication and computation ........................................ 192
5.22 Pseudo code for the control loop ................................. 196

6.1 Farmed raytracing in Advanced Caliban ............................ 206
6.2 A pipeline raytracer in Advanced Caliban .......................... 207
6.3 Simple raytracer performance figures for three problem sizes ... 210
6.4 Execution-time plot of the 2 slave static raytracer .............. 211
6.5 Execution-time plot of the 15 slave static raytracer ............ 211
6.6 Execution-time plot of the 30 slave static raytracer ............ 212
6.7 Farmed raytracing with variable grain size control ............ 213
6.8 Different job sizes for a 10 slave farm raytracer, 100x100 scene... 214
6.9 Execution-time for a 10 slave farm raytracer, 100x100 scene, 160 
ray work packet. .................................................. 215
6.10 Execution-time for a 10 slave farm raytracer, 100x100 scene, 160 
ray work packet and compute ahead of two. ...................... 215
6.11 Different degrees of compute-ahead for a 35 processor farm ray- 
tracer (100x100) .................................................. 216
6.12 A static farm with compute ahead of two. ......................... 217
6.13 A static farm with compute ahead of two and blocking factor of 40.218
6.14 Pipeline farm with control over the number stages ............ 219
6.15 The basic pipeline raytracer for three problem sizes ............ 220
6.16 The pipeline raytracer with grain size control ................. 221
6.17 Different job sizes for a 4 stage pipeline raytracer, 40×40 scene. 221
6.18 Different compute ahead values for a 40×40 scene pipeline raytracer. 222
6.19 A 40×40 scene pipeline raytracer, with compute ahead of two and 
block size of 10. ................................................. 222
6.20 Requirement stencil for the transformed Laplace equation .... 224
6.21 LNO functions for 1D and 2D arrays ............................ 226
<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.22</td>
<td>A 2D LNO in Haskell</td>
<td>227</td>
</tr>
<tr>
<td>6.23</td>
<td>A simple sequential implementation of Jacobi iteration</td>
<td>227</td>
</tr>
<tr>
<td>6.24</td>
<td>Adding convergence to the sequential Jacobi iteration</td>
<td>228</td>
</tr>
<tr>
<td>6.25</td>
<td>A streamed sequential version of the Jacobi iteration</td>
<td>230</td>
</tr>
<tr>
<td>6.26</td>
<td>A naive parallel implementation of Jacobi Iteration</td>
<td>231</td>
</tr>
<tr>
<td>6.27</td>
<td>Dividing the plane into patches and patch communication</td>
<td>233</td>
</tr>
<tr>
<td>6.28</td>
<td>Unfolding a patch ready for computation</td>
<td>234</td>
</tr>
<tr>
<td>6.29</td>
<td>Streamed patch LNO</td>
<td>236</td>
</tr>
<tr>
<td>6.30</td>
<td>Parallel Jacobi Iteration using streamed patches</td>
<td>238</td>
</tr>
<tr>
<td>6.31</td>
<td>Adding convergence to the patched parallel Jacobi Relaxation program</td>
<td>239</td>
</tr>
<tr>
<td>6.32</td>
<td>Parallel folding of arrays of streams</td>
<td>241</td>
</tr>
<tr>
<td>6.33</td>
<td>Broadcasting using a snake</td>
<td>242</td>
</tr>
<tr>
<td>6.34</td>
<td>Using the broadcast mechanism to distribute control information</td>
<td>243</td>
</tr>
<tr>
<td>6.35</td>
<td>The full, unphased process network generated by the Jacobi Relaxation program</td>
<td>244</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Writing parallel programs is hard as parallelism adds several new layers of complexity to the task of programming such as

- problem partitioning,
- scheduling and placement and
- temporal dependencies (e.g. updating shared variables and messages racing each other) — potential for non-determinism.

Traditional parallel programming languages have not helped by being little more than libraries bolted onto existing programming languages to access the new features of the parallel machine.

New methods of abstracting detail are needed. Slowly new languages are being developed that hide various complexities of parallel programming.

Functional programming languages have often been proposed and used for parallel programming as they are semantically clean and offer a non von Neumann abstraction of computation [3, 20, 72]. The whole range of parallel programming styles have been tried using functional languages, from message passing libraries (e.g. SML with message passing libraries [50]) through to automatic parallelisation (e.g. Goldberg’s Buckwheat and Alfalfa [37]), with varying degrees of success. Most success has been with shared memory parallel computers.
The problems lie not in discovering parallelism, but in exploiting useful sized parallelism and organising it on the parallel machine. The problems are exacerbated on a distributed memory machine as the overheads arising from process startup and communication latency are higher than on a shared memory machine, so the parallelism used has to be larger in grain size than for a shared memory machine to compensate for the higher management costs.

In an attempt to solve these problems, most researchers have retreated from the goal of automatic parallelisation for distributed memory computers. Programmer directives are used to prompt the system to make reasonable choices about the individual parallel components.

\section{Contribution of this thesis}

This thesis explores the parallel functional language called Caliban \cite{caliban}. It is an annotation language, introduced by Kelly, that allows the programmer to specify how a sequential functional program is to be partitioned into a parallel functional program.

The language has the following interesting features:

- It is declarative.
- Annotations are formed in the same language as the application program.
- Annotations can interact with the application program, making decisions based on the information in the program.
- It can specify relationships between parallel components.
- All partitioning decisions are made at compile time.

There are two main areas of interest when looking at Caliban. Firstly, how it is compiled and secondly what sort of programs can be written in it. Both of these questions are addressed in this thesis.
1.1.1 Statement of originality

The original contributions of this thesis are as follows:

- Techniques used to compile a simple form of Caliban, Basic Caliban. The techniques fall into two major sections:
  - **Simplification**: partial evaluation of the program to extract the annotation.
  - **Network Extraction**: program transformation to partition the program according to the annotation.

- The first Caliban compiler implementation.

- Advanced Caliban: an enhanced language definition, compared to Basic Caliban, to overcome some of the inadequacies of Basic Caliban.

- Updated compilation techniques to compile Advanced Caliban:
  - **Simplification**: added ability to extract multiple nested annotations.
  - **Phasing**: folding sub-networks on top of each other to improve program performance.
  - **Network Extraction**: a much improved scheme for partitioning programs.

- A design for an Advanced Caliban runtime system.

- An implementation of Advanced Caliban, based on the Fujitsu AP1000 distributed memory parallel computer.

- Two example applications implemented in Advanced Caliban.

- Some performance figures.

- A detailed analysis of the potential and problems of Advanced Caliban, especially relating to the integrated nature of the annotation and application language.
1.2 Thesis Layout

In the remainder of this chapter important related work is surveyed. The survey is broken into three sections, the first examining the nature and prevalence of parallelism, the second introducing parallel computing architectures and the final section describing the other attempts at building parallel programming systems.

Chapter 2 introduces Basic Caliban. It is the subject of this thesis as a whole. Caliban is a declarative parallel programming language based on Haskell. The language is introduced concept by concept and some small example programs are presented. A raytracer, which is used as an example throughout the rest of the thesis, is described. Different parallelisations of it are presented, including the use of program transformation to verify that two implementations are equivalent. The chapter aims to introduce the basic philosophies of the language and its approach.

Chapter 3 shows how Basic Caliban has been implemented on the Meiko CS-1. The major, novel, phases of the compiler are described:

- simplification,
- network extraction and
- the runtime system.

Simplification is the use of partial evaluation to convert the programmer’s high level description of the required parallelism into a lower level description usable by the rest of the compiler. The issues surrounding incomplete information and non-termination of the process are discussed. A schema notation for the partial evaluator is also introduced. Network extraction is the process by which the original program is transformed into a network of communicating tasks according to the programmer’s transformed annotation. In the initial implementation the program is transformed into a set of functions that represent the subcomputations. The pilot runtime system is also described.

Chapter 4 starts by analysing Basic Caliban’s faults and inadequacies. It leads to the definition of Advanced Caliban, an extended version of Basic Caliban that
improves on the original in three important areas:

- process partitioning,
- annotation encapsulation and
- process network usage.

Chapter 5 presents implementation methods for Advanced Caliban. It takes each of the enhancements in turn and describes their implementation. The whole implementation is based on a new target machine and functional runtime system. These are described and the new Caliban runtime system and interface are presented.

Chapter 6 shows how two different application programs are developed and implemented in Caliban, the major new program being a Jacobi Relaxation solver. Some performance figures are presented and analysed. The chapter finishes with an analysis of Advanced Caliban which deals with programming methodology and some of the problems presented by the approach.

Finally, Chapter 7 summarises the thesis, describes the similarities and differences between Caliban and some related work and concludes.

Appendix A shows the final Jacobi Relaxation program from Chapter 6.

1.3 Literature Survey

In the following sections the field of parallel computing will be surveyed. Three different areas will be examined:

1. Abstract notions of parallelism.
2. Parallel computing hardware.
3. Parallel computing programming languages.
1.3.1 Parallelism

This section surveys the nature of parallelism, where it is available, how we view it and its limits.

**Intra- and inter-processor parallelism**

There are two routes into parallelism, one is through intra-processor parallelism (e.g. replicating functional units within a processor) and the other is through inter-processor parallelism (replicating processors). The advantage of intra-processor parallelism is that it does not require a shift in programming style: the computer still looks like a normal von Neumann machine. The processor uses internal concurrency to boost performance, e.g. multiple instructions can be started at each clock cycle, or at the bit level by operating on each bit of a word concurrently. To use this form of the parallelism only the compilers need be modified so that instructions in the sequential program are scheduled to avoid the processor stalling.

The alternative to intra-processor parallelism is inter-processor parallelism. Here, multiple processors are combined using some interconnection network to form a parallel computer. The combined processors either have local per-processor memories or one large shared memory. Either way, each processor can be viewed as a single computer. The big advantage of parallel computers is that the component complexity is kept low. Collections of simpler processors are easier and cheaper to build than one very complicated processor. Increasingly the two approaches are being combined as intra-processor parallelism finds its way onto the standard commodity microprocessors from which most multiprocessors are built.

The rest of this thesis is concerned with multiprocessors, rather than intra-processor parallelism.

**Parallel paradigms**

There are two basic approaches to parallel programming:

1. process parallel and
2. data parallel.

The process parallel approach divides the tasks that are required by the application into subtasks that can be performed in parallel. The system then schedules these subtasks for execution according to the available computing power and any dependencies between the subtasks. Inherently, it is the processes that are parallelised and the data distribution that has to conform to whatever pattern is required of it.

With data parallel systems the data is the focus of the parallelisation. Either the system or the programmer partitions the data amongst the available computing resources and this forces the computation to be shared out amongst the processors. Here, each processor executes the same set of instructions, but only on subsets of the data. Parallelism is gained by extracting concurrency from operations that are applied to multiple data items. In other words, the algorithm is specified as a single control thread that can perform concurrent operations on the program’s aggregate data. This approach relies on large, regular data sets to achieve significant parallelism [43].

In order to reason formally about parallel program behaviour, formal models of synchronisation and communication are needed. In the most general models, programs are described as a set of interacting sequential processes (with encapsulated control and state). The interfaces between the sequential processes are a well defined set of communication channels down which information messages can be sent. The sequential processes cannot access any part of the other processes’ state except by sending messages down the communication channels and receiving results in a programmed exchange. Each task can be viewed as a black box with input channels and output channels defining its interface.

There are several formalised models based on message passing with distinct sequential tasks. These include Hoare’s CSP [45] and Milner’s $\pi$-calculus [70]. The $\pi$-calculus (the most recent of the two) is based on the notion of naming and name hiding. Using a name is equivalent to receiving a message; names represent communication channels. Hidden names are used to keep information
local to a process or a group of processes. Another feature of the $\pi$-calculus is its ability to specify dynamic process systems. Formal languages like the $\pi$-calculus have been so successful in specifying behaviour that they are used to notate the semantics of other languages, e.g. generalised object oriented languages [86] and Darwin [24]. The advantage of providing a $\pi$-calculus semantics is that once a program has been translated into the calculus, various proof techniques developed for the calculus are applicable, thus providing a route to formal proofs without developing a theory for the language itself.

Neither of the above approaches provide a good general mapping from software to hardware. Bulk-synchronous parallelism (BSP) is an attempt to provide a model that is as general for parallel computers as the von Neumann model of computation is for sequential computers [83, 84]. The model is characterised in the following way. A BSP machine consists of a number of processor and memory units, an interconnection medium that links these units and a facility to synchronise the components at regular intervals. Computation proceeds in supersteps where some controller hands out tasks to each processor in the network which then start to execute them. After a period of time, $L$, the processors are synchronised. The processors then execute for another time period, $L$. If all the processors have finished their task when one of the synchronisation points is reached, then the superstep has finished and a new one can start. Notice that there is no synchronisation between the processors, other than at the prescribed global synchronisation points. Messages can be exchanged between processors during timesteps.

BSP sits between the hardware and software as a bridging model. It does not mandate the type of hardware required to implement it or the programming style needed to express programs in it. Both data parallel and process parallel style programs can be compiled for the BSP model, making it general in the sense that the von Neumann model is general for serial processing.
Limits of parallelism

Is there a limit to the amount of parallelism that can be extracted from a program? The first answer to this question was given by Gene Amdahl, an IBM engineer, in the late sixties. His argument was that the parallelism available is crucially restricted by the inherently sequential sections that almost all programs possess. The argument can be formulated into a simple set of equations, starting with speedup:

\[ S = \frac{T(1)}{T(N)} \]  

(1.1)

where \( S \) is the speedup achieved given that \( T(i) \) is the time taken to solve the problem with \( i \) processors. It can be reasoned that a program can be divided into those sections that are inherently sequential and those that are possible to implement in parallel. If the sequential fraction of the program is \( \alpha \), the parallel portion must be \( (1 - \alpha) \). The sequential portion of the program therefore takes \( \alpha T(1) \) to execute. Given that the parallel section of code exhibits perfect speedup, in other words \( T(N) = \frac{T(1)}{N} \), it takes \( (1 - \alpha) \frac{T(1)}{N} \) on \( N \) processors. This gives the total execution time on \( N \) processors as:

\[ T(N) = \alpha T(1) + (1 - \alpha) \frac{T(1)}{N} \]  

(1.2)

This equation can be substituted into the speedup equation to give the following representation of Amdahl’s Law:

\[ S = \frac{N}{\alpha N + (1 - \alpha)} \]  

(1.3)

This view of the amount of parallelism is very pessimistic. For example if \( \alpha \) for a particular program is 0.1, then the maximum speedup that is possible for that program with 10 processors is only 5.3. When \( N \) is infinity then the law becomes:

\[ S = \frac{1}{\alpha} \]  

(1.4)

This makes the example program’s maximum speedup to be 10, even on an
infinitely large and powerful machine. This is not a good omen for parallel computing.

In the late eighties two researchers, John Gustafson and Ed Barsis, produced a program with a speedup of around 1000 for a program which had a $\alpha$ of about 0.004 [38]. According to Amdahl’s law, the maximum speedup of this program should have been 250. Gustafson and Barsis argued that there is a link between the $\alpha$ and $N$ and that the parallel part of the code can be used to “fit” the parallel machine in question. This means that if a single processor is used to solve the problem, it must compute both the sequential and parallel parts (note that in the following equations execution time is normalised to one):

$$T(1) = \alpha + (1 - \alpha)$$

(1.5)

Because the parallel part of the problem has been scaled to the machine size, the time to solve the problem scaled for $N$ processors on one processor is:

$$T(1) = \alpha + N(1 - \alpha)$$

(1.6)

Substituting these new timing equations into the speedup equation gives (Gustafson-Barsis Law):

$$S = N - (N - 1)\alpha$$

(1.7)

Using the Gustafson-Barsis speedup law, the program with $\alpha$ of 0.1 has a maximum speedup of 9.1 using 10 processors, and as $N$ tends to infinity, so will the maximum speedup. This is of fundamental importance to the parallel programming world. It means that we can extract as much parallelism out of certain problems as we like, not by reducing the execution time, but by increasing the amount of work a program does in a given time. A consequence of this is that certain programs are not amenable to high degrees of parallelism because they have fixed amounts of work. To make a program achieve a high degree of parallelism one must make the sequential part disappear compared to the parallel part.
There is still much debate about the usefulness of Amdahl’s law, especially its predictive powers. Gustafson argues that when a problem is scaled to fit a larger machine, the relationship between the serial and parallel components of the program is lost and can only be discovered by running the program at the larger size. Of course, once the program is run there is no need to use Amdahl’s law to discover the speedup as it will be plain from the results [39].

### 1.3.2 Parallelism: Hardware

Flynn produced a taxonomy of computer systems based on the notion of data and instruction streams [26]. A standard uniprocessor computer is classified as a SISD (Single Instruction stream, Single Data stream) machine. From his classification, two important parallel computer organisations have remained:

1. **SIMD** (Single Instruction stream, Multiple Data streams)
2. **MIMD** (Multiple Instruction streams, Multiple Data streams)

It can be seen that SIMD closely matches the data parallel model of computing and MIMD matches the task parallel model. It is, of course, possible to implement data parallel programs on a MIMD machine by making each processor perform the same sequence of operations. This hybrid style is called SPMD (Single Program, Multiple Data streams).

These classifications say nothing of the organisation of the memory for the computer. There are two ends to the possible memory organisation spectrum:

1. **Fully shared**: All processors share a single pool of memory. For example, the Sequent Symmetry machines [63, 71].

2. **Fully distributed**: Each processor has its own local memory and there is no direct access to a remote processor’s memory. For example, the Fujitsu AP1000 [49] and the Meiko CS-1 [68] (both used for experimental work in this thesis).

This thesis is concerned with the distributed memory model of hardware. The main distinguishing factor between different distributed memory computers
is the design of the communication network that connects the processors together. There are many different schemes including:

- **Fully connected:** all processors are connected to all other processors. This scheme is only feasible for small numbers of processors.

- **Ring:** processors are connected to two neighbours only, forming a ring.

- **Mesh:** each processor is connected to its four neighbours in a 2D plane or its six neighbours for a 3D cube.

- **Hypercube:** for an $n$-dimension hypercube each processor has $n$ links. The interconnections are given by a simple recursive definition:
  1. A $0$-dimension hypercube is a single processor.
  2. A $(d + 1)$-hypercube is constructed by connecting the corresponding processors of two $d$-dimension hypercubes.

- **Multistage:** multiple layers of switching nodes are placed between the processors such that a processor can communicate with all the other processors by selecting a route through the switching stages.

- **Bus:** all the processors lie on a bus, only one processor can be sending at one time.

Each network type has its advantages and disadvantages. The larger the number of connections the more expensive it is to build, but the better message passing performance it will provide. There are two aspects to message passing performance:

1. message distance and

2. contention.

When a network is not fully connected, messages have to be routed via intermediate nodes for them to reach their destination. The more nodes that have to be traversed the longer the message will take to arrive at its destination. When a
network shares connections between various sets of processors then it is possible that two senders will wish to use the same connection at the same time, which is not possible, so one will have to wait. This effect is called contention. The various design choices above suffer from various levels of the two performance effects.

**Message routing**

Another major influence on message passing performance is the method used to forward messages from processor to processor. Traditionally messages were passed using the *store and forward* method. The complete message is sent from one processor to its neighbour, then the neighbour sends it to one of its neighbours and so on until the message arrives at its destination. Machines such as the Meiko CS-1 use this method. This means that the time taken to transmit a message from source to destination is proportional to the message distance (number of hops), effectively it is the product of message size and message distance \([2]\). Another factor is whether the processors have separate logic to deal with messages or whether the main processor has to be interrupted. On the Meiko, the latter is the case. This means that the work of intermediate processors is interrupted whilst they route messages to their destinations.

One major improvement is to use *cut-through* routing \([56]\). Here, the message is divided into small packets that are forwarded to the next processor in the path as soon as they arrive. Effectively the messages snakes its way across the network to its destination; the technique is sometimes called *wormhole* routing. Cut-through routing requires hardware support, which in itself improves the performance. The real gains come because of the parallelism achieved by the pipeline nature of the transmission. The message transmission time is now effectively the sum of message length and the message distance. The Fujitsu AP1000 is an example of a machine that uses cut-through routing. In \([49]\), Ishihata et al quote the message passing time for the AP1000 as: \(160 \times (D + 4N + 1)\) nano seconds, where \(D\) is the message distance and \(N\) is the size of the message in bytes. Each extra hop is merely equivalent to an extra quarter of a byte in message size. With
store and forward, each extra hop is equivalent to the whole of the message!

There is a drawback, the message time equation given here is assuming that there is no contention. When a worm is in the process of being sent, each stage in the route is blocked and cannot transmit other worms. Although cut-through routing gives better performance (throughput and latency) because it is using pipeline parallelism, it suffers from more contention. The nature of this contention is hard to predict and is beyond the scope of this thesis. The general effect is that as message distance or size increase so does the effect of contention.

One conclusion of this section is that message distance is becoming less important. It is now no longer necessary for two processors to be adjacent in order to achieve good communications performance. However, message distance increases the possibility of contention, rather than directly increasing communication time. This contention can reduce the message passing performance of the machine. It may be possible to organise the communication so that there is no contention, even for multi-hop transfers.

**Dominance of distributed memory architectures**

Small scale shared memory computers can be implemented by placing multiple processors onto a single memory bus. This design does not scale very well; as more processors are added, so the contention for access to the single memory bus increases until it becomes a bottleneck by causing processors to stall waiting for data from memory. Caching can be added between each processor and the main memory. Once caches are added, special protocols are needed so that when memory cells are written to, copies of the old data are flushed out of the other caches.

As processor numbers increase still further, the bus based shared memory model becomes less attractive because of the contention for use of the bus. Large scale shared memory computers are now implemented using an architecture that looks like a distributed memory computer with each processor having a local memory and messages passing between processors to keep the global memory up to date. A distributed memory machine has hardware and software added to make
it appear like a shared memory machine. This has the advantage over distributed memory systems in that it is easier to program. The big disadvantage is that data placement becomes very important as the costs of remote data fetching are large. Also, accidentally migrating data away from a processor that is using it can cause serious performance problems.

Finally, systems have been developed that have no global memory. Caches usually store copies of global data. However with cache only memory architectures (COMA) there is no basic global memory, instead the local memories of each CPU are managed as cache and data is stored entirely in these caches and passes between them. A reference to an address may cause a message to be sent to retrieve the latest version of the cell. The KSR1 from Kendall Square Research is an example of a COMA machine [54].

This leads to the conclusion that distributed memory and shared memory models are converging towards a single architecture. The big issue of both models is now the same: how to partition data and computation so as to optimise performance.

1.3.3 Parallelism: Software

It has been known for some time that software has been lagging behind hardware in terms of performance and usability. Great effort and expertise are needed in order to be able to produce efficient parallel programs. Part of the reason for this is the ingrained sequential model of computing that attempts to form parallel solutions in terms of old serial solutions. Another problem lies in the extra level of complexity of parallel programming in the requirement of managing multiple processors. Not only do the application specific problems have to be solved but also the partitioning of the solution and the interaction between the parallel components of the system have to be managed.

What follows is a tour of some of the language solutions to parallel programming. There is a bias towards systems that cater for distributed memory (or distributed shared memory) as these are the systems of the future.
Plain message passing

The simplest systems (for the system implementor) are those that place message passing libraries on top of existing languages. The programmer is required to write a set of communicating programs which are loaded onto the processors. The normal arrangement of the system is to have a host program and a set of processor programs. The host program is the first to start, it coordinates the placement and connections of the processor programs which do the actual work of the application. Often the process configuration is fixed at load time by the host program. With these systems the programmer has to deal with all the complexity of partitioning and scheduling explicitly as the programs must be written to fit the desired partition and schedule.

An example of the simple libraries approach is the Fujitsu AP1000 implementations for C and Fortran [32, 33]. They provide message passing, including automatic routing, and synchronisation operations for all the processors and in addition they provide facilities to load and organise the parallel machine for the host process.

One problem with the library method is that each manufacturer has produced a slightly different set of operations for their own machines. The basic core of functionality is the same across the platforms, but the naming or approaches vary. This is where the standards groups have entered. There are two main attempts at a message passing library standard; PVM (Parallel Virtual Machine) [35] and MPI (Message Passing Interface) [69]. Both systems provide an abstract notion of the parallel machine, allowing groups of processors to form sub-machines (process groups). Additionally, PVM provides dynamic process creation. MPI provides collective operations that help SPMD programs to be written.

Although these systems do not provide high levels of abstraction, programmers do like them because they are based on languages that they already know. Also, once a program is written in PVM or MPI it can be easily ported to other parallel machines that have an implementation of the these standard libraries. Finally, these systems are usually efficient because they sit very close to the
original machine.

**Data parallel languages**

Like the message passing libraries, there have been many attempts at extending imperative languages with data parallel capabilities. Examples include C* [76], VPP Fortran [34] and High Performance Fortran (HPF) [43].

HPF provides several extensions over and above Fortran 90. These include:

- Data distribution directives
- Explicit parallel statements

The data distribution directives provide hints to the compiler as to where data should be placed. Arrays can also be aligned with respect to each other to improve locality during computation. Data can be redistributed and realigned during the computation to improve access for each computational phase. HPF uses the *Owner Computes* rule to decide where the computation for each element of the distributed array should happen. Essentially, the computation happens at the processor where the result will reside. In other words, the data partitioning drives the computation scheduling.

Explicit parallelism can be specified in two ways:

1. **FORALL** is an extension of the vector assignments of the Fortran 90.

2. **INDEPENDENT** specifies that iterations of a **DO** loop are independent and can therefore be scheduled in parallel.

In essence the HPF directives allow the programmer to specify data placement and evaluation order. Computation partitioning is extracted from the data placement information using the “owner computes” rule. At present HPF is limited to parallelising array based computations only.

**Functional programming**

Functional languages are declarative programming languages that do not use state or assignment and therefore do not admit side effects. Programs written
in functional languages are formed as sets of function definitions and a top level expression that defines what the result of the program should be. Examples of sequential functional languages are Haskell [48], Gofer [51], Miranda\(^1\) [82] and Hope [10].

Reduction is the process by which functional programs are evaluated. A functional program starts off as a top level expression and a set of function definitions. An expression consists of subexpressions, some of which can be evaluated because all the information that they need is present. These evaluable expressions are called redexes. Functional evaluation is the process of choosing redexes and evaluating them to leave new expressions. The resulting expressions may or may not include redexes. If they don’t then the expression is said to be in normal form. Which redex is chosen to be evaluated next is defined by the reduction order. Program evaluation consists of evaluating the top level expression (and any resulting subexpressions) until it is in normal form.

The most important property of functional language evaluation is the Church-Rosser Theorem which states that all possible terminating reduction orders for an expression will produce the same result. Some evaluation orders cause certain programs not to terminate, therefore according to the Church-Rosser Theorem, different evaluation order will either produce the same result or no result at all.

There are two common reduction orders:

1. **Strict** (applicative order or leftmost innermost)
   
   Expressions are evaluated before they are passed into functions as arguments. This is similar to the way normal imperative languages work.

2. **Non-strict** (normal order or leftmost outermost)
   
   Expressions are only evaluated when needed. This generally means that only the minimum amount of work needed to produce a result is performed. When non-strict reduction order is combined with full sharing of the results of common subexpression evaluation, the result is lazy evaluation.

---

\(^1\)Miranda is a trademark of Research Software Ltd.
Normal order reduction is fully normalising, i.e. if a program could terminate, it will terminate with normal order reduction.

Another important property of functional languages is referential transparency. Because functional programs do not allow assignment, the result of any function call depends only on the values of its arguments. To demonstrate this effect a small example is needed. Assume a one argument function called \texttt{foo()}, written in C. The application \texttt{foo(1)} may return 2 the first time it is called, but it could return 3 the next time. This is because the body of the function may use some static, updatable data in computing the result of the function. In a functional language, however, this cannot happen as no assignment, and therefore no updating, is allowed. If a Haskell expression \texttt{(foo 1)} were evaluated twice, the same result would be produced both times, no matter how the function is defined.

This has great implications for the compiler designer and programmer. The most important is the ability to perform program transformation without the need for global information. For example, the expression \texttt{((foo 1) + (foo 1))} can always be transformed into the expression:

\begin{verbatim}
let
  a = foo 1
in a + a
\end{verbatim}

where \texttt{let} introduces a name for a value. Conversely, the similar C statement \texttt{result = foo(1) + foo(1)} cannot be transformed into:

\begin{verbatim}
a = foo(1);
result = a + a;
\end{verbatim}

without global information about the definition of \texttt{foo()} and the transitive closure of all the functions/variables it references to ensure that no updatable state that is used is changed.

One of the major advantages given for functional languages (and all pure declarative languages) is their ability to be manipulated using simple correctness
preserving rules. One example of this the Unfold/Fold system developed by Burstall and Darlington [11]. The system has six rules that can be used to perform equational transformation on target programs. No matter which rules are applied, the program is guaranteed to maintain at least partial correctness. Total correctness cannot be guaranteed as derived programs may not terminate on all the inputs with which the original program terminated. One use of such systems is in program optimisation, for example by reducing the number of passes over a data structure by combining the functionality of multiple passes into a single pass.

Another example of optimising transformations is their use with data types [53, 25]. Here, the structure of a data type is transformed to suit the underlying evaluation mechanism. For example, a list could be transformed into an array to allow better access complexity, or it could be transformed into a tree to make it more amenable to parallelism. No matter what transformation is used, the programmer still needs to be involved. Usually some insight into the problem is needed which allows the programmer to see the final version before it is reached. This style of transformation cannot, therefore, be fully automated, although there have been suggestions to aid it with transformation support environments. One such example would allow the programmer to build libraries of transformations using a transformation meta-language. These could then be applied at the touch of a button. This would greatly ease the use of transformation as an everyday programming tool [23].

Indirectly related to the data representation problem for parallelism is the similar problem for sequential computing. The representation of a data structure has an effect on the asymptotic behaviour of its operations. By choosing the right representation for a data structure the execution time of a program can be reduced. Chuang and Hwang [15] have recently published some results that show that data representation can indeed be automatically chosen by the system. It may be possible to broaden these results to include data representation for parallel programs.

The Bird-Meertens theory is another type of transformational programming
approach [7]. It is based on providing high level, aggregate operations to standard data types, very similar in concept to Backus’ FP [4]. One example is based on lists. Rather than provide the standard operations of cons, head, tail etc, and build programs from that level, the theory of lists provides second order functions that use lists in an “atomic” way, such as map, reduce, accumulate. When programmed in this high level manner, list programs become amenable to law-based meaning-preserving transformations in a way that standard “low level” recursive programs are not. This leads in turn to a style of programming based on stepwise refinement of a program from initial inefficient, but obviously correct, problem statement to efficient implementation with the desired properties.

Skillicorn went on to show that this programming formalism provided a universal model for parallel programming [79]. Programs expressed in the Bird-Meertens formalism are very high level, they do not make any assumptions about the underlying architecture, nor do they over synchronise algorithms by expressing them as elementwise computations. Because, for example, list operations are specified at a high level using the predefined functionals, rather than as linearly recursive functions, there is a reduction in the synchronisation introduced in the implementation of the algorithm as the system is free to choose an implementation of the high level operation that suites it best. When implemented as a low level elementwise computation, the system can do nothing except follow the programmer’s wishes, with an inherent loss of parallelism. This is because it cannot in general infer the global picture from a specific set of operations. If it could, then operations such as map could be performed in parallel rather than sequentially. The Bird-Meertens formalism gives the programmer the tools to express programs at the high level required to allow the system to organise the parallelism.

Cai and Skillicorn demonstrate that the Bird-Meertens formalism can be used to develop a parallel solution to the maximum segment sum problem from a simple specification [12]. After developing a set of transformations based on observations about Horner’s rule, these can be reapplied to the problem in question, providing a new parallel algorithm. This approach can be generalised to
using mathematical toolkits to help develop programs. Unfortunately the approach requires a high degree of mathematical skill, not only in understanding and applying the transformations, but also in the vision needed to know which transformations to apply.

Parallel functional programming languages

The Church-Rosser property states that no matter which evaluation order is used to evaluate a program, the same result will be produced. This includes using a parallel reduction order. For this reason functional languages are seen as being ideal for parallel programming. Unlike imperative languages, where statements are bound tightly by dependencies that are not obvious because they are hidden by the updatable store (aliasing), functional languages make all their dependencies clear and easily discoverable. This means that any expression can be evaluated in parallel with any other expression, the only restriction being direct data dependencies. In fact, the problem with functional languages is not finding the parallelism, but limiting it to only useful parallelism. An expression is only worth evaluating in parallel if the work that is needed to evaluate it is larger than the overhead incurred by the parallel reduction mechanism.

Goldberg demonstrated that automatic parallelisation of functional programs produced good results for small scale shared memory parallel computers [37]. Unfortunately, results from his distributed memory version of the system were less promising. This was due to the difference in overheads in creating a parallel task between a shared memory system and a distributed memory system.

To improve the situation, annotations can be added to programs to tell the system where useful parallelism exists. The most common annotation is a “spark”. When an expression is marked as sparkable, it means that the system may evaluate the expression in parallel with the rest of the program in order to produce a speedup. Systems that use sparks include \( \langle \nu, G \rangle \)-machine [3], Alice [20] and Grip [40, 74, 81].

Although spark annotations define the partitioning of the program, they do not define the placement and locality. These are important considerations for
distributed memory parallel computing. To allow the placement information to be included Hudak's Para-functional programming language, ParAlfl [46, 47], allowed the programmer to name the processor on which the work was to be carried out. In ParAlfl, parallel tasks are flagged with an annotation that is similar to a spark except that the annotation includes an expression that evaluates to a processor number. This processor number is used to identify where the parallel task should be computed. This allows the programmer to specify exactly how the computation should utilise the machine. Unfortunately, it also means that the programmer has to deal with load balancing and locality issues explicitly. A layer of software can be added that disconnects the application from its exact mapping and improves its portability between different sized and shaped machines.

Other approaches to parallel functional programming have focused on a particular type of parallelism. For example the APERM (Amsterdam Parallel Experimental Reduction Machine) project [57] uses a different method of introducing parallelism. Instead of a general spark annotation they provide the \textit{sandwich} annotation that expresses divide-and-conquer parallelism. This construct removes the need to do low level synchronisation of the program graph between the parallel reducers by evaluating sequentially the expressions that would be shared by the parallel tasks before they start running. This means that all subexpressions shared between parallel tasks are shared "read-only" and therefore without the need for synchronisation.

One system that appears to have yielded good performance results is the functional language Sisal [9, 14]. The language has many features including reduction operations, records, streams (although not recursive streams) and non-strict computation. As it is a functional language, it does not provide mutable variables. It does, however, provide iteration constructs based on arrays and streams. Variables that are carried between iterations can be thought of as arrays that are automatically indexed by the current or previous iteration number (depending on where the variable reference is made). The result of the loop can be folded down to a single value or returned as a stream or array. Pipeline-parallelism is exhibited between stream producers and consumers because streams
are non-strict. Iteration parallelism is exhibited by loops that have no carried dependence.

The goal of Sisal is performance, and this is born out by their results. For a sample set of standard scientific codes, Sisal performed comparatively to or outperformed automatically parallelised Fortran on a Cray Y-MP [13].

**Data parallel functional programming**

The Bird-Meertens formalism uses functions that operate over complete lists. In a similar manner, NESL, an applicative functional programming language, provides sequences as a datatype for parallelism [8]. Sequences are collections of data items that can be manipulated in parallel using a predefined set of operators. Also, functions can be applied over elements of sequences in parallel. What makes this approach very powerful is that the data items that make up the sequences can themselves be sequences and the functions that are applied to the elements of a sequence can themselves contain parallel constructs. This *Nested Parallelism* allows the programmer to specify irregular nested computations and divide and conquer style parallel algorithms. Finally the system provides good PRAM-based asymptotic performance modeling.

Hill takes a slightly different approach with DPHaskell [44]. He introduces Pods, an array-like datatype that is dynamic and unbounded in size. Pod comprehensions allow the programmer to specify parallel computation over the Pods. What is interesting about this work is that the data parallel evaluation is performed with a non-strict semantics. This is achieved by using an evaluation mechanism based on *Aims*. A program can have multiple aims (or points of evaluation). The aims are evaluated in parallel. Only those expressions that are known to be needed are in the set of aims, therefore laziness is maintained whilst still allowing parallelism.

**Skeletons**

Another approach to parallel programming is to notice that there are very few basic algorithms, each program tends to be a variation of one of a small set of
parallel algorithms, e.g. task queue, pipeline, merging network and divide and conquer. In the late 1980’s, Cole realised this and developed the idea of an algorithmic skeleton [17]. He developed and analysed the performance of several basic parallel algorithmic skeletons, including:

- Farm
- Iterative Combination
- Cluster
- Divide and Conquer

In essence skeletons are libraries of parallel code that implement the structure of the parallel program, leaving the programmer to specify the application specific code segments that a particular skeleton requires. Cole’s view of parallel programming was that the user selected a skeleton that most closely matched the algorithmic need of their program, instantiated the application specific code and then the system did the rest. This high level of abstraction and restriction afforded many advantages over conventional general purpose languages. Firstly the machine architecture was not specified or even accounted for. Each skeleton would have several implementations, one for each architecture that the programming system supported, the correct one would be chosen by the system, implementing the algorithm as efficiently as possible for that architecture. Secondly, performance modelling is possible. Each skeleton comes with a performance model: the user simply needs to insert the correct figures for application specific sections of code into the model to get a complete model of their application’s parallel performance.

There are several problems with this approach. Firstly, a large number of skeletons would be needed to be able to code most parallel algorithms. Although many programs are instances of general algorithm forms, they can have many special cases that make them difficult to capture in skeletal form without having many different, but similar, skeletons. Also, specifying more general skeletons, in order to encompass a larger set of parallel programs, would mean that a
performance model could not be produced and efficient implementation could not be guaranteed on all architectures. As the implementors of a skeleton system cannot foresee all the possible requirements of the user this means that there will be certain programs that cannot be expressed using the set of skeletons provided. Because of the lack of generality, this cannot be avoided.

Foster et al describe a different form of skeleton system that they call Algorithmic Motifs [28, 27, 31]. They take the language Strand as the basis for their work. Strand is a single assignment, guarded rule based language that already has support for concurrency. A motif consists of two parts: a transformation schema, $T$, and a library of support code, $L$. The transformation is used to adapt the user’s program to suit the support library. If $A$ is the application program then the result after applying a motif is $T(A) \cup L$. In this way the original application program can be specified in any way that is convenient to the programmer, yet different motifs can be experimented with without large changes to the application code. With a normal skeleton system, the programmer has to restructure the program every time a different skeleton is tried as the program has to conform to each skeleton’s different interface. With motifs, all the programmer has to supply is a pragma identifying the task decomposition and data dependencies. These source to source transformations are easy in Strand as it has a simple term based structure.

Furthermore, new motifs can be constructed out of the composition of other motifs. Because the application of a motif is a source to source transformation, the resulting program can have another motif applied to it: $T_2(T_1(A) \cup L_1) \cup L_2$. An important consequence of this easy composition of motifs is that they can be tailored to a particular application’s need, thus improving the fit of parallelism to the application.

One final benefit that they claim is that because motifs are written in a high level language, they become useful repositories of expertise. Libraries of traditional imperative code do provide useful functionality, but are not good at preserving the expertise that went into creating them because they can be difficult to understand. Foster describes such languages as write-only: it is difficult to
turn the code back into the original algorithm. On the other hand, with high level languages the details of the algorithm are not obscured by the language used to express them.

A slightly different approach was taken by the P$^3$L project at the University of Pisa [5, 21]. Instead of encoding algorithms as skeletons, they encode programming/process structures such as LOOPS, TREES, PIPES, FARMS and GEOMETRIC mesh structures.

The program forms correspond to different types of parallelism, in much the same way as Cole’s skeletons correspond to different parallel algorithms. The programmer is asked to build programs using these structures wherever parallelism is wanted. The programmer does not need to specify the degree of parallelism, just the nature. During compilation, the system works out the degree of parallelism, e.g. the number of workers in a FARM, to be utilised by doing compile time load-balancing. An optimiser transforms the program based on knowledge of the actual target architecture and performance models of the skeletons and the architecture. A mapper is used to select a good mapping for the generated process network onto the target machine from a library of such mappings.

Later work from the same group uses an FP like language to encode the parallel algorithms [22]. Parallel paradigms (skeleton structures) are represented as special second order functionals that have a declarative meaning and a set of runtime templates (or implementations). Which template is chosen depends on the target architecture. FP was chosen as the base language because it has a fixed set of second order functionals that can be combined. This is similar to the idea that there is a fixed set of basic skeletons (or templates) that can be used to build parallel programs. The advantage of this is that the compiler does not have to deal with arbitrary functional structures, some of which can be inherently sequential (e.g. the standard recursive implementation of map).

The main restriction of Cole’s original work was that a parallel program could only be an instantiation of the single skeleton, in other words the whole application structure had to be captured by one skeleton. This works for simple programs, where only a single type of computation is being performed. More
complex programs tend to exhibit more complex algorithmic requirements, for example by being multi-phased (having more than one significant parallel computation), or by having several levels of useful parallelism.

Skeleton work at Imperial College [23, 80] has progressed toward the goal of providing skeletons that are “intelligent libraries”. These skeletons are composable using special glue skeletons to provide more complex programs. The use of glue skeletons rather than general composition operators allows more careful control of the combination of skeletons and interaction at their interfaces. This allows the system to maintain control over what is feasible for maintaining performance.

The work is grounded on the use of two layers of skeletons, primitive skeletons and combining skeletons. The primitive skeletons are implementations of parallel abstract data type (PADT) operators. A PADT is an aggregate data structure and a set of parallel operations performed over this data structure. The observation made is that most highly parallel programs are based on applying operations over large data structures in a data parallel manner. Primitive skeletons are then combined using combining skeletons that are used to capture patterns of control flow. The combining skeletons take skeletons as their parameters and codify the interface matching of their argument skeletons. Combining skeletons are in fact operators in a skeleton composition language which is used to build the structure of a skeleton program. An application therefore has a tree structure, where the internal nodes are combining and primitive skeletons and the leaves of the tree are sequential code segments provided as arguments to the lowest level of skeletons.

A skeleton is described in two ways: its declarative meaning (what the skeleton does) and its set of behaviours (how each of the skeleton implementations does the work). By codifying the behaviour of primitive skeleton implementations, it is possible for a combining skeleton to match the implementations of two skeletons to be combined by matching the interface behaviours and thus improving their fit, for example by maintaining data distribution between a pair composed skeletons. Most skeleton systems that allow composition of skeletons only allow simple interfaces between the skeletons. This work allows a more
general interface structure to be exploited.

Rangaswami has looked specifically at the area of selection of implementation based on some form of least-cost analysis [75]. A program is viewed as a series of distinct phases, each consisting of applications of recognised functions (or skeletons). Each recognised function has a parallel implementation together with a cost function. The system builds a search tree of the costs of parallelising (or not) each of the recognised functions in each of the phases. The least cost route through the tree represents the best parallel implementation.

**Configuration programming**

A final strand of parallel programming languages separates the process of defining the sequential components of a program from the parallel composition of them. Configuration languages allow the user to define sequential “black boxes” of code and then link them together using a special purpose configuration language. The configuration language knows nothing about the units except their interface, all it does is link inputs and outputs, ensuring that links are correctly typed. Some configuration languages can also specify the placement and grouping of the individual components.

New units are built from the composition of existing units in a recursive manner. Once built, they are treated as black box units themselves and are used accordingly. This enables libraries of code to be built and reused.

Examples of imperative configuration languages include Tonic [65] and Darwin [66]. Tonic, a development from Conic [67], was the earlier of the two systems and used a simple configuration language to link units built from a standard imperative programming language extended with the notion of ports. The configuration language also allows the programmer to specify the logical to physical mapping of their program. This is done by defining a special task to represent a processor node. These processor node tasks are linked together to form a model of the connections in the underlying architecture. The physical network interconnection description is compiled into a separate Tonic program using a system called Gin. The Gin program connects the processors in the way desired and loads the
application program onto the processor network. The Gin program then acts as transport layer for the application, routing messages between processors.

Darwin inherits many of the features of Tonic, and adds to them the ability to evolve structure during run time. This is achieved by the application code explicitly calling process creation and linking functions defined in the configuration layer.

In the declarative realm, Schepers developed a system called Graph that allows the user to program graphically [77]. The system consists of two language levels; a lower-level functional language, OREL/2, to define processes, and a process level language that connects the lower functional processes together using streams. Other process level constructs available are non-deterministic merges, loops and dynamic process management.

Lucco and Sharp present an approach called coordination structures [64]. A coordination structure is a structured collection of coordination items, which themselves are notations of dependencies between subtasks in the application program. A parallel program is therefore a pipeline of coordination structures. Their premise is that most programmers start with an already working sequential program before starting a parallel version. All that is then needed is to split the program into subtasks and glue them together. Delirium is a language that allows them to do this. It is a strict functional language that allows the programmer to coordinate subtasks of the application. As with a standard configuration language, their language is embedding, i.e. the coordination code sits on top of the application rather than being spread around within it. Delirium transforms are used to implement coordination structures. A transform connects inputs to outputs performing the structure changes that are needed to match the two, in other words it provides a data routing service. Each of the these routes is a coordination item and therefore a transform represents a coordination structure. Programs can be tuned by moving code in and out of the coordination language, thus changing the degree of parallelism and efficiency of the program.

Transforms can only be used to form regular transformations: this has the advantage that they can be optimised using a technique called summarisation.
Because of the regular structure the system can often, but not always, subdivide the transforms to achieve an optimal grain size of the tasks being linked.

Foster, Olson and Tuecke describe a configuration programming system called PCN in [29]. The system allows existing “dusty decks” of C and Fortran code to be parallelised by embedding them in a PCN (Process Control Notation) program. PCN uses single-assignment, definitional, variables for inter-process communication. The PCN approach involves decomposing the program into many processes which are linked using definitional variables. This forms a logical process network. The programmer can then define a mapping of logical processes to physical processors on the machine.

This mapping stage is similar to Hudak’s Para Functional Programming where a process is annotated with the processor identifier of where it should be be evaluated. With PCN, the programmer can map the application to a virtual topology and provide a mapping function from the virtual topology to the actual parallel machine. This way programs can be developed independently of their underlying architecture. To help build the virtual to real topology mapping meta-functions exist to enquire about the topology of the machine.

Code reuse is achieved through the use of templates. These are higher order functions that define a parallel algorithm based on a virtual topology. The interface to a template is provided by a port array (an array of definitional variables spread evenly across a virtual topology). This approach allows applications to be built by composing templates together.

A second method of achieving code reuse is multilingual programming which is facilitated through an interface to C or Fortran. In this way “dusty deck” code can be parallelised with much less effort than rewriting it completely [30]. The high level parallel language is used to structure the low level imperative program.

1.3.4 Conclusions

This survey has demonstrated that inter-processor parallelism is a practical way forward in the search for more computing power to solve computationally de-
manding problems.

Early in the history of computational parallelism, Amdahl's Law was thought to place severe limits on the availability of parallelism. This, to some extent, held back progress until new results showed that although Amdahl’s law was correct as far as it went, it did neglect vital possibilities for parallel programs, such as scaling problem size.

**Hardware**

Distributed memory computers are shown to be the most likely route to large-scale parallelism for the foreseeable future. Shared memory computers, in an attempt to grow larger, have turned into distributed memory computers with special hardware or extra software. It is important, therefore, to develop programming methodologies that relate directly to the underlying technology if efficient use of these machines is to be made. After all, parallelism is used in the pursuit of computational power.

Through the use of cut-through routing, locality is shown to be far less important than it used to be. This greatly simplifies the job of building parallel applications for distributed memory parallel computers. It also means that applications with very rich communication topologies can now be considered, e.g. \(n\)-body simulations [78].

**Software**

Solutions for parallel programming range from direct support of the underlying machine architecture, e.g. provided by MPI and other “bolt-on” parallelism libraries, to totally new languages designed with parallelism in mind from the start, e.g. \(P^3\)L and Darwin.

Simple “bolt-on” parallelism libraries suffer from their low level of specification. Programmers have to deal with all issues of parallel computing directly: scheduling, partitioning and synchronisation etc.

Functional languages reduce this burden by removing the need to consider synchronisation and to some extent scheduling issues. Functional languages that
use spark annotations are simple to use, but because of their dynamic and non-
regular nature, prove very difficult to parallelise efficiently, especially on dis-
tributed memory machines.

Skeletons look like a good solution for parallel application developers. They
provide a high level programming environment for quick development, coupled
with the promise of efficient and portable implementations due to the restricted
problem domain. Therein lies the issue. Skeleton systems only provide solutions
that the implementors envisaged at the design of the system. This situation can
be mitigated by the provision of low level skeletons and a method for combining
them. When this is done the complexity of the skeleton system increases as
inter-skeleton analysis is required to maintain the high level of knowledge that a
skeleton system has of the application and its performance.

Configuration based systems, on the other hand, provide flexibility and ef-
iciency at the cost of less portability. Using abstraction and modularisation
techniques, configuration systems can capture a lot of what skeleton systems
provide, but also allow for user modification and development of the base system
to provide missing functionality.

1.4 Caliban

Caliban is a declarative parallel functional programming language. It was origi-
nally introduced by Kelly in [53] in which he demonstrates functional program-
ning techniques for parallel programming. The book gives a brief introduction
to the language and a possible implementation.

Caliban is an interesting language for several reasons:

- It is declarative.

Programs written in Caliban specify what is wanted, not how to achieve
it. The declarative nature, together with the underlying referential trans-
parency of the host language allow simple program transformation and
reasoning.
• Annotations are written in the source language. Parallelism is expressed by annotating the program using expressions written in the host language itself. This allows for a powerful and novel style of expression.

• Programs express their own placement. Related to the previous point; annotations can use any information from the application. There is no distinction between annotation domain data and application domain data.

• Programs are parallelised by partitioning. Parallelism is achieved by partitioning the data and processing. The user has full control over this process.

• Reusable parallel components. Parallel structures can be captured in reusable components similar to skeletons. These components can also be extended and modified by the user.

This thesis is concerned with exploring Caliban and its related ideas. It asks if the Caliban approach is viable and if not, what extensions over the initial specification would make it viable. It examines the techniques required to implement such a language, and the restrictions that apply to it.
Chapter 2

Basic Caliban

This chapter introduces a declarative parallel programming language called Caliban. The language was first defined by Kelly in his book, “Functional Programming for Loosely-coupled Multiprocessors” [53]. He used it as a notation for defining stream based programs. The language was then taken up by the FAST project, a collaboration between Imperial College and Southampton University [1], whose aim was to develop a declarative parallel programming system and compilation techniques. The original definition of Caliban was modified during the FAST project — it is this modified version of the language, called Basic Caliban, that is presented here.

The language is the central aspect of this thesis. Its development and implementation form the bulk of the material.

This chapter, and the following one, describe the foundation work for the rest of the thesis. It is hoped that by understanding this, it will give an insight into the rest of the work that otherwise would not have been gained.

Starting with the philosophy of Caliban, the language and its constructs are then introduced. Some examples of its use are presented, finishing with a larger ray tracing example, including the use of program transformation.
2.1 The Caliban Philosophy

Caliban is a completely declarative language defined on top of a functional programming language. Caliban defines aspects of the parallel runtime behaviour of the application program using annotations. The annotations do not affect the result produced by the program, although they can affect the termination properties of the program.

The basic object in a Caliban description is the stream, a sequence of values. A stream is the connection between computations, like a communication link. Caliban annotations partition the application program into processes that are connected by streams. The processes form a directed, (possibly) cyclic graph whose nodes are the processors on the parallel machine and whose arcs are streams of data.

Partitioning is done by naming streams in the application program. In a functional language, naming a stream automatically names the computation that produces it. Streams, i.e. the data, are named rather than functions that compute the data, as this removes all ambiguity about which data is being named. This aspect of Caliban can be confusing — the data itself is being named, not the means of producing that data.

There is no interaction between nodes of the process graph other than through communication via the streams. This makes the language suitable for implementation on distributed memory multiprocessors.

The most important part of the Caliban philosophy is that the programmer knows best. Although reasonable results have been obtained by automatically parallelising compilers for shared memory systems [37], no one has obtained good results for distributed memory machines. This is due to the cost of communication being comparatively high and the difficulty in automatically recognising work packages at the right grain size — too large and you lose parallelism, too small and the overheads of management and communication outweigh the parallel savings. The programmer has a much better intuition about how the program is structured and where the best parallelism comes from.
Finally, Caliban describes static process graphs. The shape or size of the graph cannot depend on runtime information. The static restriction may seem tight, but there are still many useful programs that can be represented.

2.2 Basic Caliban

Basic Caliban, as described here, is defined on top of Haskell\(^-\), a non-strict, higher order functional programming language. Haskell\(^-\) is defined in [1], it is a reduced version of the Haskell programming language, defined in [48]. Haskell\(^-\) doesn’t fully implement some of advanced features of Haskell, such as full type classes, I/O and proper modules, but has the same syntax.

To describe the parallelism in a program, Basic Caliban partitions it into a graph, where the nodes are computed on separate processors. This is done by annotating the program with placement information. The amount of parallelism obtained is then dependent on the program’s data dependencies and the order in which the placed streams are demanded. Later, in section 2.3.1, there is a discussion of how the demand profile can change the degree of parallelism.

Annotations are of type Placement. They are just normal values in the source language, like any other. The moreover keyword introduces an annotation. It is followed by an annotation expression which does the partitioning. The annotation has no effect on the result of the expression that is being annotated. In Basic Caliban the moreover clause needs to be at the top level of the program, annotating the main expression, this is a restriction that will be lifted in chapter 4.

Annotations are made up of assertions that are linked together. Each assertion describes some aspect of the placement of the program. The basic assertion is:

Node x

It places the stream named x on a processing element of its own, therefore allowing it to compute in parallel with the rest of the program. The Process Placement Rule defines what gets computed on each node. It says that “a value is computed locally unless it has been explicitly placed elsewhere.” This means
that all reachable subexpressions from \( x \) are computed at \( x \)'s place unless they have been explicitly placed elsewhere. If they have been placed elsewhere they are computed remotely and imported over an arc.

A stream is actually just a list. This means that any list typed expression can be placed using a \texttt{Node} assertion. An alternative set of constructors was considered to represent streams instead of using normal lists, but they would have made programming too cumbersome. The annotation can describe the placement of any stream that is in scope, not just the placement of the expression that it is attached to. This is because the annotation refers to objects by their name.

Annotations are linked together using the \texttt{And} connective.

\[
a \text{And} b
\]

This takes two annotations and forms them into one. \texttt{And} is associative, so:

\[
(a \text{And} b) \text{And} c \equiv a \text{And} (b \text{And} c)
\]

\[
\text{main} = a \text{ moreover Node } a \text{And} Node b
\]

\[
\text{where}
\]

\[
a = \text{map inc } b
\]

\[
b = \text{generate_numbers}
\]

Figure 2.1: A simple program, with annotation

Some real parallelism can now be specified. Figure 2.1 shows a simple program that generates a streams of numbers and then increments each of them. The annotation is used to partition the program so that the streams \( a \) and \( b \) are computed on different processors and form a producer-consumer pair. Demand is placed for the output, \( a \), by the system, so that the final result is produced. This in turn demands elements on the stream \( b \) in order for it to satisfy the output demand. Of course, more complex structures can be built, Figure 2.2 shows a pipeline and Figure 2.3 shows a fan-in process network. The details of the programs have been removed, just leaving the important data dependencies.
main = d moreover Node a And Node b And
  Node c And Node d
  where
  a = ...
  b = ... a ...
  c = ... b ...
  d = ... c ...

Figure 2.2: A pipeline

main = d moreover Node a And Node b And
  Node c And Node d
  where
  a = ...
  b = ...
  c = ...
  d = ... a b c ...

Figure 2.3: A fan-in process network
Using the data dependencies the compiler can work out the shape of the process network and build it accordingly. Unfortunately the annotations in Figure 2.2 and 2.3 are exactly the same even though the networks they generate are very different. Thus, apart from the placement information, very little is gained from the annotation. To alleviate this situation, Basic Caliban has another assertion:

\texttt{Arc a b}

This documents a data dependency, and therefore communication, between the two named streams. In other words, one of the streams is used in the computation of the other. The direction of the data flow is not specified by the assertion; this is to make the assertion more general. As will be seen, making \texttt{Arc} directional would be disadvantageous to the user. Notice that \texttt{Arc} only documents the expectation of a data dependency so that the compiler can check it. This is similar to the part that type specifications take in the languages that have type inference, like Haskell — the system can work out what the type should be so the programmer only supplies one for verification and documentation purposes. Without the \texttt{Arc} assertions an annotation means very little to the user.

The full annotations for the pipe and fan examples (Figures 2.2 and 2.3) are now:

\begin{verbatim}
moreover Node a And Node b And Node c And Node d And
   Arc a b And Arc b c And Arc c d
\end{verbatim}

for the pipeline and

\begin{verbatim}
moreover Node a And Node b And Node c And Node d And
   Arc a d And Arc b d And Arc c d
\end{verbatim}

for the fan-in network. Note that \texttt{Arc} binds more tightly than \texttt{And} to reduce bracketing.
2.2.1 Building up networks

The annotations in their complete form are large and unwieldy. It would be useful if we could somehow package them for reuse. As annotations are written in the source language, Basic Caliban allows the programmer to use any source language construct to build them. This makes it possible to write a function that will evaluate to an annotation when given the names of the nodes that form the network wanted. Such functions are called Network Forming Operators (NFOs) as they codify common computational structures for reuse. Figures 2.5 and 2.4 show how to write the pipe and fan networks as NFOs. Remember that the Arc assertions do not specify the direction of dependency. This means that either NFO can be used in either direction. To distinguish normal lists from streams, the type Stream is used as a special synonym for [a]. The annotations for Figures 2.2 and 2.3 can now be written as:

\[
\text{d moreover pipe } [a,b,c,d]
\]

for the pipeline and:

\[
\text{d moreover fan d } [a,b,c]
\]

for the fan-in example.

\[
\begin{align*}
\text{pipe} &:: [\text{Stream}] \rightarrow \text{Placement} \\
\text{pipe } [s] & = \text{Node s} \\
\text{pipe } (a:b:rest) & = \text{Node a And Arc a b And pipe (b:rest)}
\end{align*}
\]

Figure 2.4: The pipe NFO

\[
\begin{align*}
\text{fan} &:: \text{Stream} \rightarrow [\text{Stream}] \rightarrow \text{Placement} \\
\text{fan s } [] & = \text{Node s} \\
\text{fan s } (a:as) & = \text{Node a And Arc a s And fan s as}
\end{align*}
\]

Figure 2.5: The fan NFO

The final assertion in Basic Caliban is NoPlace. It is the null assertion, that does not define any placement.
NoPlace And a ≡ a

NoPlace is useful in the base cases of NFOs, e.g. we could rewrite fan as:

fan :: Stream → [Stream] → Placement
fan s [] = NoPlace
fan s (a:as) = Node a And Arc a s And fan s as

The difference between this version and Figure 2.5 is that the input stream is no longer placed, which may be desired in certain circumstances. An annotation consisting of just NoPlace would make the program execute sequentially on a single processor.

2.2.2 Summary of Basic Caliban

Here is a summary of the Basic Caliban language:

<table>
<thead>
<tr>
<th>Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node x</td>
</tr>
<tr>
<td>Arc a b</td>
</tr>
<tr>
<td>NoPlace</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Connective</th>
</tr>
</thead>
<tbody>
<tr>
<td>a And b</td>
</tr>
</tbody>
</table>

Operator precedence:

(function application) > Arc, Node > And

Some useful equivalences:

NoPlace And annot ≡ annot     Null effect of NoPlace
e moreover NoPlace ≡ e         A NoPlace annotation has no effect
Arc a b ≡ Arc b a              Arc is commutative
a And b ≡ b And a              And is associative and commutative
a And b ≡ a, if a ≡ b           Assertions are idempotent

Process Placement Rule:

A value is computed locally unless it has been explicitly placed elsewhere.
The original Caliban

As an aside, the original version of Caliban presented in Kelly’s book [53] had only the Arc assertion. It not only documented the dependency between two stream computations, but also defined the placement. In other words the following equivalence holds:

<table>
<thead>
<tr>
<th>Kelly’s Caliban</th>
<th>Basic Caliban</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arc a b</td>
<td>Node a And Node b And Arc a b</td>
</tr>
</tbody>
</table>

A note on the semantics of Basic Caliban

Caliban attempts to have no effect on the semantics of the application it is annotating. If the annotation is removed and the program run sequentially, the answer produced should be the same as the answer produced when running it in parallel, with the annotation. This is only true when the parallel version terminates. Caliban can change the termination properties of the program. This is because a stream is a head-strict lazy list. Therefore, given a list a:as, the system evaluates a fully before it evaluates any of as. This makes the runtime evaluation order of the program slightly stricter than the lazy, sequential order. If the evaluation of a fails to terminate and values from the tail as are needed to finish the computation, then the program will not terminate. This should not be a problem, as most useful programs do not contain non-terminating expressions. It must be stressed that this termination issue only affects streams that are placed, all other expressions are evaluated lazily. For example, the following program would terminate under normal non-strict, sequential semantics (⊥ is a non-terminating expression):

```haskell
main ls = size
    where
    size = length list
    list = [⊥]
```

If the program were to be annotated thus:
main is = size moreover Node list
    where
    size = length list
    list = [⊥]

it would fail to terminate as the evaluation of the first element of list would fail to terminate. When a stream is placed, it becomes head-strict, in other words the head of the list is fully evaluated before the tail is evaluated, in this case causing the whole list computation to fail (or loop).

### 2.2.3 Expanding NFOs

During the compilation process, the compiler needs to expand the annotation to its full form, i.e. without any NFOs. This process is called simplification, and is similar to evaluation except that it is performed at compile time.

A small parallel searching example will be used to show how an NFO is expanded by the system. Suppose there is a stream of incoming queries to a database of records, each query needs to be looked up in the database to produce a corresponding output record.

As a specification, here is the sequential formulation:

```plaintext
search keys db = map (match db) keys
```

The program assumes the existence of a database inquiry function `match` which finds the record corresponding to a single key, or returns `FAILURE`.

As a simple example of a parallel search implementation, the database will be subdivided into several equal-sized parts, and a search will be performed for each key in each of the sub-databases using the original sequential algorithm. Figure 2.6 shows the process network, with each arc labelled with the expression that calculates it.

This computation is specified by the following definition, where `search'` is the parallel version, and `search` is the original sequential version used to search the sub-databases:
2.2. Basic Caliban

Figure 2.6: A parallel database search process network

\[
\text{search keys } db = \text{merge splitresults}
\]

\[
\text{where}
\]

\[
\text{splitresults} = \text{map (search keys) (split } N \text{ db)}
\]

where \(\text{split } N \text{ db}\) divides the database into \(N\) equal-sized partitions (e.g. \(\text{split 3 db} = [\text{db1, db2, db3}]\)), and \text{merge} selects a successful match, if possible, from each set of \(N\) replies.

Expressing the parallelism

The next step is to add a \textit{moreover} clause, to specify how the computation is to be distributed. The return result is named so that it can be used in the annotation. Then the \textit{fan} NFO is used to describe the distribution and merge the halves of the process network. Figure 2.7 gives the full code for the example.

\[
\text{search keys } db = \text{replies}
\]

\[
\text{moreover fan keys splitresults And fan replies splitresults}
\]

\[
\text{where}
\]

\[
\text{replies} = \text{merge splitresults}
\]

\[
\text{splitresults} = \text{map (search keys) (split } N \text{ db)}
\]

Figure 2.7: Parallel database search

This example illustrates how a function like \textit{fan} captures a family of parallel computation structures, leading to a concise explanation of how the computation is mapped onto the underlying hardware.

It also demonstrates that some care is needed with the naming of expressions: the name \textit{replies} labels the process which performs the \textit{merge} computation. Although slightly counter-intuitive, it is necessary to identify a process by the \textit{value} it computes rather than the \textit{function} it performs to avoid ambiguity.
search keys db
    = replies
    moreover Node (search keys db1) And Arc (search keys db1) keys And
    Node (search keys db2) And Arc (search keys db2) keys And
    Node keys And
    Node (search keys db1) And Arc (search keys db1) replies And
    Node (search keys db2) And Arc (search keys db2) replies And
    Node replies

where
    replies = merge splitresults
    splitresults = map (search keys) (split N db)

Figure 2.8: After a simple term-rewriting evaluation

The compiler now needs to expand the annotation to its canonical form. If a simple term rewriting system were used then the annotation would be expanded, but the computational part of the program would not get updated in step. Figure 2.8 shows the results, assuming \( N \) is two and therefore \( \text{split 2 db} \) returns \([db1, db2]\). Not only does the program not mirror the evaluation that was performed to build the annotation, but the annotation doesn’t match itself internally. Each of the \( \text{search keys db} \) expressions is a different computation that cannot be equated. Caliban needs to be able to determine if two expressions should be the same computation. Fortunately, there is a solution. A proof devised by Barendregt et al. in [6], shows that term-graph rewriting, where \text{letrecs} name expressions to allow sharing, correctly implements term rewriting. Therefore term-graph rewriting can be used to expand the annotation. This maintains full sharing between the annotation evaluation and the program. The proof assures that the program’s I/O behaviour is not changed by the shared expansion process.

An example of how the compiler expands the annotation will now be presented. Only those steps that rewrite the graph are shown, the evaluation order presented is not exactly the same as would be used by the compiler, but is just to make the example clearer to the reader. In the following program samples, the next expression to be reduced is marked with an under-brace.
search’ keys db = replies
    moreover fan replies splitresults And fan keys splitresults
    where
    replies = merge splitresults
    splitresults = map (search keys) (split N db)

Firstly, the second argument of each of the fan calls needs to be examined, in both cases it is splitresults. This will force the evaluation of the split call, yielding:

splitresults = map (search keys) [db1, db2]

splitresults can then be evaluated to [sdb1, sdb2]. This is a two element list of closures representing the two subsearches (search keys db1 and search keys db2).

search’ keys db = replies
    moreover fan replies [sdb1, sdb2] And fan keys [sdb1, sdb2]
    where
    replies = merge [sdb1, sdb2]
    sdb1 = search keys db1
    sdb2 = search keys db2

Notice how the program is also rewritten, all references to a value are updated when it is evaluated because of the nature of term-graph rewriting. Next, the first call to fan in the annotation can be rewritten by one step:

search’ keys db = replies
    moreover Node sdb1 And Arc sdb1 replies And
    fan replies [sdb2] And
    fan keys [sdb1, sdb2]
    where
    replies = merge [sdb1, sdb2]
    sdb1 = search keys db1
    sdb2 = search keys db2

This leaves the recursive call, applied thus:
search’ keys db = replies
     moreover Node sdb1 And Arc sdb1 replies And
        Node sdb2 And Arc sdb2 replies And
        fan replies [] And
        fan keys [sdb1, sdb2]
     where
     replies = merge [sdb1, sdb2]
sdb1 = search keys db1
sdb2 = search keys db2

And finally, for this particular fan call, the base case is evaluated:

search’ keys db = replies
     moreover Node sdb1 And Arc sdb1 replies And
        Node sdb2 And Arc sdb2 replies
        Node replies And
        fan keys [sdb1, sdb2]
     where
     replies = merge [sdb1, sdb2]
sdb1 = search keys db1
sdb2 = search keys db2

This completes the first fan call, the second one is performed in a similar manner, leaving the program as:

search’ keys db = replies
     moreover Node sdb1 And Arc sdb1 replies And
        Node sdb2 And Arc sdb2 replies And Node replies And
        Node sdb1 And Arc sdb1 keys And
        Node sdb2 And Arc sdb2 keys And Node keys
     where
     replies = merge [sdb1, sdb2]
sdb1 = search keys db1
sdb2 = search keys db2

As Node is an idempotent assertion, any repeated annotations can be ignored. Doing this leaves the same annotation as the hand-written version. The annotation is fully evaluated and the program reflects the evaluation. Also note that each process is now a named expression at the top level of the program.
The simplification process is similar to normal program evaluation, therefore like a normal program, it may fail to terminate. This can be caused for several reasons:

1. The process network described is unbounded.

2. The computation performed to build the annotation may loop.

3. The process network’s shape or composition may depend on some runtime argument or input.

The first reason why simplification may fail is demonstrated by the following program:

```haskell
main input = result moreover fan result approx
where
  approxFs = map makeF [1..]
  applyFs (f:fs) i = (f i) : (applyFs fs i)
  approx = applyFs approxFs input
  result = findBestSoln approx
```

Without the annotation the program simply constructs a lazy list of approximation functions, which are in turn applied to the input data until the best solution is found and returned. The construction of `approx` is performed lazily, with only those solutions that are required by `findBestSoln` actually being computed. However, when the annotation is used to place the solution generators, the process network it represents is infinite as there are an infinite number of solution functions. Therefore the simplifier cannot compute it.

An obvious example of the second of these failure conditions is the following NFO:

```haskell
sillyNFO :: [Stream] → Placement
sillyNFO (a:as) = an
  where
    an = Node a And an
```
Here, the annotation is actually a cycle. The simplifier would end up chasing its
tail trying to evaluate an.

The compiler cannot simplify all annotations. If the annotation needs runtime
information, then the simplifier cannot fully expand the annotation and has to
stop with an error message. For example, the shape and size of a process network
may be affected by runtime information:

\begin{verbatim}
main precision = result moreover fan result approx
  where
  approxFs = take numberNeeded (map makeF [1..])
  numberNeeded = howMany precision
  applyFs (f:fs) i = (f i) : (applyFs fs i)
  approx = applyFs approxFs input
  result = findSoln approx
\end{verbatim}

This program is similar to the earlier one, except that the number of solution
functions needed for a certain level of precision can be determined. Only that
number of approximation functions need be constructed. This now places a bound
on the number of functions in the network. Unfortunately, in this example the
precision parameter is a run-time argument and therefore the size of the process
network cannot be determined at compile time. This makes it impossible for the
simplifier to construct the normal form annotation.

2.3 Example programs

This section shows several examples of how Basic Caliban can be used to build
parallel programs. It culminates in a raytracer, showing how different parallelis-
sations of the same program can be achieved.

Load balancing a pipeline

Given a pipeline of computations, it has already been shown how an annota-
tion can be built to form the appropriate process graph. This has not taken load
balancing into account. Figure 2.9 shows a pipeline which is annotated with a
measure of the computational load at each node. The higher the load value, the longer it will take to process a single element on that node. The values ascribed to each node can be worked out in several ways, for example, by analysing the code or by running the program with monitoring and examining execution traces.

The pipe as a whole can only work at the speed of the slowest node within it [16]. It therefore seems a waste of resources to use a whole processor to compute a fast function, like on node H, when it will mostly stand idle waiting for results from earlier sections of the pipeline. The converse is true of nodes that precede the slow node. They would stall because their output buffers would fill up.

A more efficient pipeline is one where the processors all work in lock step, i.e. the processors all calculate their current value at the same time, passing the result on to the next processor concurrently. The closer a pipeline gets to this balance, the more efficient it will be. To improve the efficiency of the pipeline, computations should be grouped so that each processor is doing approximately the same amount of work. This can be done by hand quite easily. To annotate this, just remove items from the call to pipe. Figure 2.10 shows a balancing of the pipeline, together with the new annotation needed to build it. The other pipe stages are automatically coalesced into neighbouring nodes by the Caliban system because they are no longer placed anywhere else. They will be computed on processors that use them.

Figure 2.9: A pipeline annotated with computational loads

Figure 2.10: A hand-balanced pipeline
Building pipelines

Pipelines can be built in much more interesting ways. Until now NFOs have been used to build only the annotations. This can be tedious as access to each of the subcomputations that form the stages of the pipe is needed in the call to pipe. The normal higher order function chain, defined in Figure 2.11, is used to build pipeline-like computations in sequential programs. Unfortunately, building an annotation to place a pipeline formed by chain is impossible as the subcomputations (each of the function applications) are not accessible to the caller of chain.

\[
\text{chain} :: [a \rightarrow a] \rightarrow a \rightarrow a
\]

\[
\text{chain} [] \text{input} = \text{input}
\]

\[
\text{chain} (f:fs) \text{input} = \text{chain} \text{fs} (f \text{input})
\]

Figure 2.11: chain — apply a sequence of functions to an input

The solution to the subcomputation hiding problem is to build another version of the higher-order function that builds an annotation as it is building the computation, so that it has access to the subcomputations. fpipe is an example of just such a function, it is defined in Figure 2.12.

\[
\text{fpipe} :: [\text{Stream} \rightarrow \text{Stream}] \rightarrow \text{Stream} \rightarrow (\text{Stream}, \text{Placement})
\]

\[
\text{fpipe} [] \text{input} = (\text{input}, \text{Node} \text{input})
\]

\[
\text{fpipe} (f:fs) \text{input}
\]

\[
= (\text{next}, \text{annot And Node current And Arc current next})
\]

\[
\text{where}
\]

\[
\text{current} = f \text{input}
\]

\[
(\text{next}, \text{annot}) = \text{fpipe} \text{fs} \text{current}
\]

Figure 2.12: Building and annotating a pipe in one go

The NFO fpipe is the parallel form of chain, it takes a list of functions and builds a pipeline where each function is computed on a separate node. It can be
balanced by hand by composing groups of the argument functions together to balance the computational load.

Interestingly, Caliban can help with the load balancing. The definition of \texttt{fpipe} could be refined so that the computational load of the argument functions is included, then Caliban can balance the pipe automatically. A simple implementation of this idea is shown in Figure 2.13. It works by finding the function with the highest load and trying to group the other functions together to produce pipeline nodes that are nearly as heavily loaded. The functions are grouped together so that each group is as balanced as it can be. The functions within each group are then chained together, to form one function. The list of chained functions is then passed to the original \texttt{fpipe} function.

\begin{verbatim}
loadfpipe :: [(Stream\rightarrow Stream, Load)] \rightarrow Stream \rightarrow (Stream, Placement)
loadfpipe fs input = fpipe nodes input
  where
  max = fold max (map snd fs)
  groups = group max fs
  nodes = map chain groups

group :: [(Stream\rightarrow Stream, Load)] \rightarrow Int \rightarrow [(Stream\rightarrow Stream)]
group [] max = []
group fs max = node : (group max remaining)
  where
  (node, remaining) = groupNeighbours fs max 0

groupNeighbours :: [(Stream\rightarrow Stream, Load)] \rightarrow Int \rightarrow Int \rightarrow
  [(Stream\rightarrow Stream), [(Stream\rightarrow Stream), [(Stream\rightarrow Stream, Load)]]
groupNeighbours ((f,load) : rest) max tot
  | (load+tot) > max = ([], (f,load):rest)
  | otherwise = (f,node, remaining)
  where
  (node, remaining) = groupNeighbours rest max (load+tot)
\end{verbatim}

Figure 2.13: A load balancing pipe NFO

To use this NFO with the example pipeline the call would be:

\begin{verbatim}
loadfpipe [(fA,5),(fB,2),(fC,1), (fD,3),(fE,4),(fF,8),(fG,7),(fH,1)] input
\end{verbatim}
where \( fA \) etc are the functions used to compute each of the pipeline stages. It would produce the same partitioning as the hand balanced example above. More complex strategies are possible, but are beyond the scope of this thesis. This is an example where the decomposition is based on the data. This is an exciting class of programs that allow for great flexibility.

**More computation NFOs**

To finish this section, two more computation NFOs will be presented. Both are fold-like operations, but with very different process graphs. The first is `treefold`, which builds a binary reduction tree, like in Figure 2.14. The input streams are placed and then combined by the binary operator `op`. The NFO that produces this tree is given in Figure 2.15. E.g. To merge four streams

\[
\text{treefold merge } [A,B,C,D]
\]

can be used to build the computation and annotation.

![Figure 2.14: A binary folding tree](image)

The second version of the fold operation is called `insert`. It effectively places the combining operator between adjacent elements of the input list, with this specification:

\[
\text{insert} \oplus [a_1, a_2, a_3 \ldots a_n] = a_1 \oplus (\cdots (a_{n-2} \oplus (a_{n-1} \oplus a_n)) \cdots)
\]

The parallel version of `insert` is given in Figure 2.16. Using `parinsert` instead of `treefold` for the merge example would produce the process graph shown in Figure 2.17. Here the merge operations are performed on the nodes where the
treefold :: (Stream → Stream → Stream) → [Stream] → (Stream, Placement)

\[
\text{treefold } f \ [s]\ =\ (s, \text{Node } s)
\]

\[
\text{treefold } f \text{ streams } =\ (\text{combined, Node } \text{combined And fan combined } [a, b] \text{ And }
\]
\[
\text{annota And annotb})
\]

where

\[
\text{combined } = f\ a\ b
\]
\[
(a, \text{annota}) = \text{treefold } f \text{ suba}
\]
\[
(b, \text{annotb}) = \text{treefold } f \text{ subb}
\]
\[
(\text{suba, subb}) = \text{split streams}
\]

Figure 2.15: The treefold annotation

streams are placed, except for the last stream which is placed on its own. This process graph is similar to a pipeline in that the processes form a single linear structure.

parinsert :: (Stream → Stream) → [Stream] → (Stream, Placement)

\[
\text{parinsert } f \ [s]\ =\ (s, \text{Node } s)
\]

\[
\text{parinsert } f \ (s:ss) = (\text{appl, annot And Node appl And Arc appl next})
\]

where

\[
\text{appl } = f\ s\ \text{next}
\]
\[
(\text{next, annot}) = \text{parinsert } f\ ss
\]

Figure 2.16: A linear fold NFO

parinsert merge [A,B,C,D]

Figure 2.17: Merging using parinsert

2.3.1 Raytracing

To round off this chapter, a simple raytracer is presented. This example comes from [53]. A static farm implementation of it is discussed. Then a manual pro-
gram transformation is described that converts the farm program into a pipeline equivalent.

A simple ray-tracer can be reduced to the following program:

```haskell
ray_trace scene viewpoint = map impact rays
  where
    impact ray = fold earlier impacts
      where
        impacts = map (hit ray) scene
    rays = generate rays viewpoint
```

The function `impact` takes a ray and generates an impact description (either the object name and impact location or the fact that no impact has occurred) for each object in the scene database by mapping `hit` over `scene`. From the list of impacts the program finds the closest one to the viewpoint using `fold earlier` (i.e. the one that will colour the output pixel).

**Farming**

A simple parallel decomposition of the problem would be to *farm* the top level `map` by partitioning the set of rays amongst a set of worker processors. The stream of rays is partitioned and passed to a group of processors who calculate the impact information for each of the rays. The impacts are then collected and reassembled into a single result stream. This can be written as follows:
ray_trace scene viewpoint = result
    moreover farm_ann
    where
      (result, farm_ann) = farm impact rays
      rays = ...
      impact ray = ...

farm :: (a → a) → Stream → (Stream, Placement)
farm func input = (farmed, annot)
    where
      farmed = unpartition slaves
      slaves = partition (map func input)
      annot = fan farmed slaves

When using a static farm to parallelise the application, the number of processors and the data distribution need to be decided at compile time. There is also the assumption that the computation time for each of the elements is similar. If they are not similar then the computation load of each slave would be unbalanced; load balancing this way may cause a problem if all the high computation elements are assigned to the same slave. Randomisation approaches may help here because patterns in the computation load are not repeated on the slaves, the load is jumbled up leading to a smoothing effect. In the code, partition splits the impact list (the list being farmed over) according to some criterion. The exact criteria are up to the programmer. Figures 2.18 – 2.20 show three possible implementations of the partition/unpartition pair of functions.

    partition1 :: Stream → [Stream]
    partition1 ls = map singleton ls
        where
          singleton x = [x]
    unpartition1 :: [Stream] → Stream
    unpartition1 ls = fold (++) ls

Figure 2.18: Partition each element

The three partitioning functions presented have different compile-time requirements. The first two need to be able to work out the size of the impacts
partition2 :: Int → Stream → [Stream]
partition2 n ls = divide size ls
  where
    len = length ls
    size = len div n
    divide size [] = []
    divide size ls = (take size ls) : (divide size (drop size ls))
unpartition2 :: [Stream] → Stream
unpartition2 ls = fold (++) ls

Figure 2.19: Partition to a given number of contiguous groups

dividation3 :: Int → [a] → [[a]]
partition3 n [] = []
partition3 n ls = zipW (::) [] heads tails
  where
    heads = take n ls
    tails = partition3 n (drop n ls)
unpartition3 :: [[a]] → a
unpartition3 ls = heads ++ tails
  where
    heads [] = []
    heads ([]:rest) = heads rest
    heads ((a:as):rest) = a : (heads rest)
    tails [] = []
    tails ([]:rest) = tails rest
    tails ((a:as):rest) = as : (tails rest)

- Lazy version of zipWith with base case specified for second argument
zipW :: (a→b→c) → b → [a] → [b] → [c]
zipW f b [] ls = []
zipW f b (a:as) [] = f a b : (zipW f b [])
zipW f b (a:as) (l:ls) = f a l : (zipW f b ls)

Figure 2.20: Partition to a given number of groups, round-robin style
list at compile time so that all the Node assertions for each process can be given. The final partition function, partition3, generates N processes in a round-robin fashion so needs no information about impacts.

It turns out that at run-time the partition2 version has a (lack of) strictness problem. Figure 2.21 shows a simple farm with two partitions. The boxes above the slaves represent the data items that will be produced by the slave, the values represent the order in which they will be produced. The Caliban system computes one element ahead of known demand (so that the processors do not coroutine), this means that values 5 and 6 from slave2 will not be computed until needed by result. In other words, slave2 will be stalled until all of the values from slave1 have been used, effectively sequentialising slave1 and slave2.

Using partition3/unpartition3 avoids this problem by computing adjacent elements of the output on different processors, slave1 would compute [1,3,5] while slave2 would compute [2,4,6]. Now one element compute-ahead is used to full advantage as values are taken from each slave in turn, in a round-robin fashion.

**Folding**

Instead of parallelising over the rays, the program can be parallelised over the scene objects. This is achieved by transforming the program so that the computation over the rays is a sub-computation of the computation over the scene.

This can be done by pushing the map over rays inside the map over scene. To achieve this two basic steps are needed:

1. Push the outer map inside the fold
2. Swap the two maps. This is like a loop interchange in an imperative language.

The first step is to push the outer map inside the fold. Removing extraneous detail, the basic raytracer looks like:

\[ \text{map } (\lambda r . \text{fold earlier expr}) \text{ rays} \]

It can be shown that this is equivalent to:

\[ \text{fold } (\text{map2 earlier}) (\text{transpose } (\text{map } (\lambda r . \text{expr }) \text{ rays})) \]

Here the outer map has been pushed inside the fold. This has two effects. Firstly earlier now has to deal with streams of impacts, so map2 (a map for two argument functions) is needed. Secondly, to deliver the impacts in the right order the list of lists generated by the map has to be transposed. Remember, expr produces a list.

This transformation is shown to be correct as follows.

\[ \text{map } (\lambda r . \text{fold earlier expr}) [r_1 \cdots r_n] \]

expands to:

\[
[ \\
  \text{fold earlier } [\text{expr}_1 r_1 \cdots \text{expr}_m r_1], \\
  \vdots \\
  \text{fold earlier } [\text{expr}_1 r_n \cdots \text{expr}_m r_n]
\]

Here, \(\text{expr}_i r_n\) is the \(i^{\text{th}}\) value \((1 \leq i \leq m)\) of the list generated from \(r_n\). To show the transformation correct, the transformed version is expanded from:

\[ \text{fold } (\text{map2 earlier}) (\text{transpose } (\text{map } (\lambda r . \text{expr }) [r_1 \cdots r_n])) \]

to give:
fold (map2 earlier) (transpose [ 
    [ expr\textsubscript{1} \cdot \cdot \cdot expr\textsubscript{r\textsubscript{1}}, 
    \ldots 
    [ expr\textsubscript{1} \cdot \cdot \cdot expr\textsubscript{r\textsubscript{n}} ]
]
)

Performing the transpose leaves:

fold (map2 earlier) [ 
    [ expr\textsubscript{1} \cdot \cdot \cdot expr\textsubscript{r\textsubscript{1}}, 
    \ldots 
    [ expr\textsubscript{m} \cdot \cdot \cdot expr\textsubscript{r\textsubscript{n}} ]
]

which finally expands to:

[ 
fold earlier [expr\textsubscript{1} \cdot \cdot \cdot expr\textsubscript{m\textsubscript{1}}], 
\ldots 
fold earlier [expr\textsubscript{1} \cdot \cdot \cdot expr\textsubscript{m\textsubscript{n}}]
]

which is the same as the untransformed version of the expression. With this transformation the raytracer can be rewritten as:

fold (map2 earlier) (transpose (map (λr . (map (λel . hit r el) scene)) rays))

The second stage of the transformation takes the maps that have been pushed together and swaps them, bringing the map over scene to the outside. Just taking the maps:

map (λr . (map (λel . hit r el) scene)) rays

they can be swapped like this:

transpose (map (λel . (map (λr . hit r el) rays)) scene)
Again, a `transpose` is needed to keep the order of the results the same as in the original expression. This transformation can be shown to be correct using extensionality, as before.

\[
\text{transpose} \left( \text{map} \left( \lambda \text{el} . \left( \text{map} \left( \lambda \text{r} . \text{hit r el} \right) [r_1 \cdots r_n] \right) \right) [s_1 \cdots s_m] \right)
\]

expands to:

\[
\text{transpose} \left[ \begin{array}{c}
  \left[ \text{hit } r_1 \ s_1, \ \cdots \ \text{hit } r_n \ s_1 \right], \\
  \vdots \\
  \left[ \text{hit } r_1 \ s_m, \ \cdots \ \text{hit } r_n \ s_m \right], 
\end{array} \right]
\]

Doing the transpose gives:

\[
\left[ \begin{array}{c}
  \left[ \text{hit } r_1 \ s_1, \ \cdots \ \text{hit } r_n \ s_m \right], \\
  \vdots \\
  \left[ \text{hit } r_n \ s_1, \ \cdots \ \text{hit } r_n \ s_m \right], 
\end{array} \right]
\]

which is the same as the original version of the `map`:

\[
\text{map} \left( \lambda \text{r} . \left( \text{map} \left( \lambda \text{el} . \text{hit r el} \right) \text{scene} \right) \right) \text{rays}
\]

Finally, the two transformations can be pulled together, leaving the following program:

\[
\text{fold} \left( \text{map2 earlier} \right) \left( \text{transpose} \left( \text{transpose} \left( \text{map} \left( \lambda \text{el} . \left( \text{map} \left( \lambda \text{r} . \text{hit r el} \right) \text{rays} \right) \right) \text{scene} \right) \right) \right)
\]

Notice that the two `transposes` cancel each other out, so they can be removed leading to the new definition of the ray-tracer as shown in figure 2.22. This is
now amenable to parallelisation over the scene by using a parallel fold operator, like treefold from Figure 2.15.

Using the parinsert folding NFO, Figure 2.16, the process network generated is a pipeline.

\[
\text{ray_trace' scene viewpoint} = \text{impacts}
\]
\[
\text{moreover annot}
\]
\[
\text{where}
\]
\[
(\text{impacts, annot}) = \text{parinsert (map2 earlier) llimps}
\]
\[
\text{llimps} = \text{map imp_for_object scene}
\]
\[
\text{imp_for_object obj} = \text{map (λray . hit ray obj) rays}
\]
\[
\text{rays} = \text{generate}_\text{rays viewpoint}
\]

Figure 2.22: The transformed parallel ray-tracer

## 2.4 Conclusions

This chapter has introduced the basic philosophies of Caliban:

- The programmer is responsible for making decisions about partitioning.
- Expressive power is borrowed from the source language by forming annotations in that language rather than in a meta-language.
- Libraries of useful network forms (NFOs) can be built to aid code reuse.
- Only expressions that are explicitly placed are communicated, all other values are computed locally — again, the programmer knows best.
- Caliban has a global view of the program in that the relationship between parallel components can be expressed (using Arc). This is unlike the systems with a local view, such as the spark annotation, where there is no structure to express relationships between components.

The Basic Caliban language has been introduced, together with some simple parallel programs. Several useful building blocks (NFOs) have been presented,
these will be used again later. Finally, the power of declarative program transformation and reasoning has been shown with the transformation of farm-based raytracer to a pipeline-based version.

In the next chapter, the implementation of the Basic Caliban will be documented. It shows how a Basic Caliban program is compiled into a normal functional programming language which has been extended with a simple parallelism providing primitive.
Chapter 3

The Implementation of Basic Caliban

This chapter will present the pilot implementation of Basic Caliban, produced during the FAST project, a collaboration between Southampton University and Imperial College. [1, 18, 19]. The parallel system is built on top of an optimising sequential functional compiler. This chapter focuses on the extensions required to parallelise the existing compiler. The system is targeted at a Meiko CS-1 consisting of 32 Inmos T800 transputers using the CSTools [68] communication library to provide high level management facilities such as automatic process mapping and routed message support.

This pilot implementation was presented at a meeting of parallel functional programmers held at Heriott-Watt University. The performance of the pilot implementation was very poor. Later in the thesis a new implementation based on an improved language is presented. The material in this chapter is presented to show the starting point for the new implementation.

The novel work in this chapter is the language frontend and the compilation techniques used to compile Caliban programs into the intermediate form ready for the backend compiler. The runtime system had already been implemented.

Figure 3.1 shows the passes which constitute the Basic Caliban compiler frontend. The input to the compiler is a Caliban program (annotated Haskell−)
Figure 3.1: The frontend compiler for the Caliban system

and the output is a transformed program in the FAST intermediate language. This is a simple functional language enhanced with a single parallelism primitive, procnet. The first phase performs type checking and pattern matching removal on the input program, leaving a simple, type correct, pattern free, internal program tree which the Caliban phases can then manipulate. Next, the Caliban annotation is simplified to its normal form. Then the program is restructured by the Network Extractor into a set of functions, one for each node in the process graph, leaving the program as a call to the procnet primitive. Finally the program is lambda-lifted, leaving a normal sequential program that is output in a simple intermediate functional language. The output of the frontend is then compiled by the FAST sequential compiler which generates a binary file that contains the whole parallel program. This program can then be loaded onto the parallel machine for execution.

This chapter will start with a description of the Haskell language on which Caliban is built. It will show how Caliban is integrated with the source language and go on to describe the two frontend phases; Simplification and Network Extraction. Then it will describe the enhancements of the sequential runtime system needed to the support the procnet primitive. Finally it will present a short performance evaluation of the pilot implementation.

3.1 Integrating Caliban with Haskell

As described in the chapter 2, Haskell is a cut-down version of Haskell, the standardised non-strict, higher order, functional programming language [48]. The Haskell committee has used Haskell to introduce several interesting language
features, such as type classes and modules. Although these concepts can greatly improve the expressive power of a language, they are also hard to implement and are not important for parallel programming. As such, they have have been left out of the Haskell definition. Full details of Haskell can be found in [62]. The main difference for parallel programming between Haskell and Haskell is that input/output has been simplified. The function main takes a list from the user and returns a list as the program result.

Caliban needs to be able to represent its annotations at the program level as they are built by evaluating part of the program. Unlike other language systems, where annotations are not at the same language level as the application program (e.g. sparks), Caliban programs can directly manipulate their own annotations, albeit only at compile time. The method used is to make the annotations a normal constructed data type called Placement. Annotations need to refer to the streams to be placed. Streams can be of any type, but the type system will not allow a datatype to include references to free types. One possibility would be to pull the type of the stream out as an argument to Placement like this:

\[
\text{Placement a} = ... \mid \text{Node [a]} \mid ...
\]

Unfortunately, due to the restriction that the type system places on objects in the language this would only allow streams of a particular type for any one program, i.e. each stream would have to be of the same type, e.g. a stream of integers. The solution adopted is to construct a tagged union of all message types called Message. That way annotations can refer to any stream because they all have the same type; [Message]. The disadvantage of this scheme is that the programmer has to explicitly use tags for elements of streams. Although not all types are defined in the union, the Message type is under programmer control and can be extended with any type needed.

Figure 3.2 shows the set of declarations needed to add Caliban to Haskell. This set allows the construction of annotations by the source program. NFOs are just functions that return objects of type Placement. Annotations can be manipulated using any part of the host language.
data Message = CHAR Char | INT Int | FLOAT Float | BOOL Bool

type Stream = [Message]

infixl 2 'And'
data Placement = NoPlace | Node Stream | Arc Stream Stream

| Placement And Placement

Figure 3.2: Definitions to integrate Caliban into HASKELL-

infixl 1 'moreover'
moreover :: a → Placement → a
a moreover b = a

Figure 3.3: Definitions for moreover

Finally, we need to implement the moreover annotation keyword. This is simply implemented as an infix binary function, as in Figure 3.3. Notice that moreover binds less tightly than And so that bracketing is not needed in normal use. Also, And binds less tightly than function application, again to reduce bracketing.

These definitions are all placed in a separate module called Caliban, which can be included with any Caliban program to make it executable with any Haskell compiler/interpreter. The sequential system would just ignore the annotations and execute the programs normally: see the definition of moreover in Figure 3.3. Also, useful NFO definitions can be placed in the Caliban module.

HASKELL’s reduced implementation of the module system is just like C’s #include mechanism. A module is textually included at the point of the import statement. HASKELL also uses a prelude file, containing frequently used definitions and certain system required declarations, which is imported as a set of modules with every program. The prelude file is prepended to the Caliban program, leading to the following program structure:
letrec
    prelude definitions

user’s top level definitions

main l = result
    where
    result = f ...

The standard type checker can perform some of the consistency checking needed by Caliban, e.g. ensuring the placed items are streams. The other checking, e.g. ensuring that there is only one moreover clause and that it is at the top level, is done by the Caliban phases themselves.

## 3.2 Simplification

The programmer can use any source language construct to help build the annotation. It is the simplifier’s job to evaluate the annotation to a form that can be used by the rest of the compiler. The fully evaluated annotation is in Annotation Normal Form, ANF. An annotation in ANF has no NFO applications in it: it is a complete and canonical definition of the process network to be formed. All streams referenced by the annotation are in Weak Head Normal Form, WHNF, so that the compiler can match expressions that are the same computation. If the streams were not in WHNF then there would be an aliasing problem, e.g. stream1 and head [stream1...] are the same stream, but without analysis the compiler cannot discover this. By evaluating both streams to WHNF the compiler can easily test whether they are the same stream. The test for equality turns out to be a simple pointer equality test. A relation to test whether an annotation is in ANF is given in Figure 3.4.

It is important that the evaluation of the annotation does not change the properties of the program in several important respects:

- **Sharing**
  The compiler uses sharing to determine if two expressions are the same
ANF NoPlace
ANF Node a if WHNF a
ANF Arc a b if WHNF a & WHNF b
ANF a And b if ANF a & ANF b

Figure 3.4: Definition of ANF

computation. The simplifier must maintain all sharing that existed in the original program and all that is introduced during the evaluation process.

- **Recomputation**
  Related to the sharing issue. The simplifier must not introduce recomputation of expressions as this would affect the complexity of the resulting program.

- **Structure**
  If there is any hope that the user will be able to relate error messages to their program then the evaluated program must maintain structural and naming similarity with the original.

The database example in chapter 2 shows that sharing is very important and that the way to achieve it is to use term-graph rewriting as the evaluation model for the simplification process. Without term-graph rewriting the program would not be updated during simplification. This would leave old references to computations and would mean that the following phases could not determine which expressions had been annotated.

Finally, because the simplifier is evaluating parts of the program at compile time, certain pieces of information are not known. Any expression that depends on the input stream of the program cannot be evaluated at compile time because the input stream will not be available until runtime. The simplifier needs to be able to deal with this lack of full information.

The simplifier could fail for other reasons too:

- by evaluating non-terminating expressions and
• attempting to build infinite process networks.

The previous chapter showed examples of each of these failure situations.

3.2.1 The implementation

The implemented simplifier is basically a template instantiation graph reducer [25, 73]. A full template instantiation graph reducer would start evaluation by instantiating the top level definitions. This destroys the program’s structure and therefore violates the structure property laid down earlier. A modified version of the graph reducer is used instead. With the modified reducer, environments are used to store the definitions of objects that are in scope of the moreover clause, whilst instantiation is used for definitions of expressions actually being evaluated. With the modified method, only expressions that are evaluated in the simplification process are changed. The simplifier views letrec definitions as naming nodes in the program graph, therefore two uses of the same name (in the same scope) reference the same node and therefore share the result when it is evaluated.

Two evaluation strategies are needed to evaluate the program to ANF. Firstly an evaluator to evaluate annotations to ANF is needed, this works at the Placement level, uncovering assertions. Secondly an extended WHNF evaluator is needed to remove the aliasing problem. Both of these evaluators need to be able to cope with incomplete information. Specifically, if an expression references the argument to main then it cannot be evaluated fully and an error message can be given.

Figure 3.5 shows the structure of programs as they enter the simplifier, i.e. a very simple unsugared \(\lambda\)-calculus. This is translated into a term-graph, as defined in Figure 3.6, by the \(TS\) scheme (Figure 3.7). It is these term-graphs on which the simplification algorithms are based.

Figure 3.6 shows the data structure used to represent the application program\(^1\).

---

\(^1\)This notation is new and was developed after discussions about previous incarnations of it with Ross Patterson.
Data type for simple $\lambda$-calculus expressions:
\[ \textbf{data} \ \text{STerm} = \text{Const} \ \text{Constants} \mid \]
\[ \text{Var Name} \mid \]
\[ \lambda \ \text{Name} \ \text{STerm} \mid \]
\[ \text{App} \ \text{STerm} \ \text{STerm} \mid \]
\[ \text{If} \ \text{STerm} \ \text{STerm} \ \text{STerm} \mid \]
\[ \text{Constr} \ \text{Tag} \ [\text{STerm}] \mid \]
\[ \text{Letrec} \ [\text{Name} = \text{STerm}] \ \text{STerm} \]

Constants = Num Int | Chars Char | Boolean Bool | Prim (STerm $\rightarrow$ STerm) | ...
Name = ... abstract ... – Variable names
Tag = ... abstract ... – Constructor tags for constructed data

Figure 3.5: Simple unsugared $\lambda$-calculus.

Data type for terms:
\[ \textbf{data} \ \text{Term} \ \alpha = \text{Const} \ \text{Constants} \mid \]
\[ \text{Var Name} \mid \]
\[ \lambda \ \text{Name} \ \alpha \mid \]
\[ \text{App} \ \alpha \ \alpha \mid \]
\[ \text{If} \ \alpha \ \alpha \ \alpha \mid \]
\[ \text{Constr} \ \text{Tag} \ [\alpha] \mid \]
\[ \text{Letrec} \ [\text{Name} = \alpha] \ \alpha \mid \]
\[ \text{Ind} \ \alpha \]

Constants = Num Int | Chars Char | Boolean Bool | Prim (Memory $\rightarrow$ Node $\rightarrow$ (Node, Memory, Success)) | ...
Name = ...
Tag = ...
\[ \textbf{data} \ \text{Success} = S \mid F \]

Data type and access functions for memories:
Memory = Node $\rightarrow$ Term Node
Node = ... abstract ... – “Address” of object in the memory
Env = [(Name, Node)]

empty :: Memory
assign :: Memory $\rightarrow$ Node $\rightarrow$ Term Node $\rightarrow$ Memory
new :: Memory $\rightarrow$ Term Node $\rightarrow$ (Node, Memory)

Figure 3.6: Definitions for schemes
- Translate simple λ-expressions into Terms:

\[ \mathcal{T}S :: \text{STerm} \rightarrow \text{Memory} \rightarrow (\text{Node, Memory}) \]

\[ \mathcal{T}S (\text{Const } c) \ m = \text{new } (\text{Const } c) \ m \]

\[ \mathcal{T}S (\text{Var } v) \ m = \text{new } (\text{Var } v) \ m \]

\[ \mathcal{T}S (\lambda \ n \ e) \ m = \text{new } (\lambda \ b \ ep) \ m' \]

where

\[ (ep, m') = \mathcal{T}S e \ m \]

\[ \mathcal{T}S (\text{App } f \ a) \ m = \text{new } (\text{App } fp \ ap) \ m'' \]

where

\[ (fp, m') = \mathcal{T}S f \ m \]

\[ (ap, m'') = \mathcal{T}S a \ m' \]

\[ \mathcal{T}S (\text{If } a \ b \ c) \ m = \text{new } (\text{If } ap \ bp \ cp) \ m''' \]

where

\[ (ap, m') = \mathcal{T}S a \ m \]

\[ (bp, m'') = \mathcal{T}S b \ m' \]

\[ (cp, m''') = \mathcal{T}S c \ m'' \]

\[ \mathcal{T}S (\text{Constr } t \ [a_1, ..., a_n]) \ m = \text{new } (\text{Constr } t \ [ap_1, ..., ap_n]) \ m_n \]

where

\[ (ap_1, m_1) = \mathcal{T}S a_1 \ m \]

\[ \vdots \]

\[ (ap_n, m_n) = \mathcal{T}S a_n \ m_{n-1} \]

\[ \mathcal{T}S (\text{Letrec } \ [n_1 = a_1, ..., n_x = a_x]) b) \ m = \text{new } (\text{Letrec } \ [n_1 = ap_1, ..., n_x = ap_x]) bp) \ m' \]

where

\[ (ap_1, m_1) = \mathcal{T}S a_1 \ m \]

\[ \vdots \]

\[ (ap_x, m_x) = \mathcal{T}S a_x \ m_{x-1} \]

\[ (bp, m') = \mathcal{T}S b \ m_x \]

Figure 3.7: Simple λ-expression to Term translation.
A *Memory* is an abstract type used to represent the term-graph of the user’s application program that needs simplifying. A *Node* is a named “memory” cell, it is like an address (or pointer) in a conventional computer. A *Term* is the datatype that defines what can be placed into a node (or memory cell). Term is parameterised by the type used for naming the cells, in this case Node. A memory is a collection of node to term mappings, defining the current state of the graph. The node indirection is used so that sharing can be expressed in the functional context of the evaluations rules. There are four operations that can be used to build/look at memories:

1. *lookup* a node’s contents,
2. add a *new* node to the memory,
3. *update* the contents of an existing node,
4. build an *empty* memory.

As the memory is represented as a partial function from nodes to terms, the lookup operation simply becomes function application. The contents of node “x” in memory “m” can be found by “m x”.

Terms can be constants, variables, lambda expressions, function applications, conditionals, constructed data, recursive “let”-expressions or indirection nodes. Indirection nodes are used to maintain full sharing, especially when selector functions are used [25, 73]. Note also that primitive functions are constants. Primitive functions are represented as a function that takes an argument node in the current memory and returns a result node in an updated memory. Primitives also return a success value that determines whether they were able to evaluate all their arguments enough to be able to produce a result. Should the primitive call yield a failure, indicating that it was not able to produce a result due to lack of information, then the caller needs to decide if this is important. If the caller is in the process of evaluating a stream to WHNF then the failure is deemed to be unimportant and the evaluator returns (also passing the failure back) as if it has
actually reached WHNF. If, on the other hand, it is attempting to build part of the annotation, then the failure is treated as an error and evaluation is halted.

The caller has to decide whether the success of the call was important or not and deal with failures. Primitive function applications are the only point in the system when lack of runtime information can become apparent. It turns out that the “success” values have to be piped all round the schemes.

The $\mathcal{T}$ scheme directly converts the simple tree structure of its input into the graph structure by copying each node into the graph space. The new function is used to build a new node in the memory.

Figures 3.8 to 3.11 show the definitions for a WHNF evaluator extended with ability to cope without full information. The evaluator is represented as a functional program that manipulates the term-graph structure representing the user’s application program that is being simplified.

Figure 3.8 shows some definitions used by the rest of the schemes. Following that, Figures 3.9 to 3.11 show the schemes themselves. Each sub-scheme is captured as a function that performs different actions according to the type of the arguments passed to it. A case expression is used to match different expression types using patterns. Each pattern may contain several guarded right hand sides and also a local set of recursive definitions, introduced by the where keyword.

Starting with the WHNF scheme, $\mathcal{E}$, in Figure 3.9. The first rule in $\mathcal{E}$ shows that a constant is already in WHNF, so the memory is returned unchanged and successfully. In the second rule, for variables, the variable is looked up in the memory and changed into an indirection to the value found, then evaluated. In the function application case the function is evaluated, if that fails due to lack of information the call is failed straight away, otherwise the $\mathcal{F}$ scheme is used to perform the application. Note that the “skip” function is used to remove all indirections to the function before using $\mathcal{F}$. The “letrec” case instantiates, in place, each of the definitions and the body with references to the definitions, then updates the letrec node with an indirection to the instantiated body. Finally it

\[\text{Figures 3.8 to 3.11 show the definitions for a WHNF evaluator extended with ability to cope without full information. The evaluator is represented as a functional program that manipulates the term-graph structure representing the user's application program that is being simplified.}

\[\text{Starting with the WHNF scheme, } \mathcal{E}, \text{ in Figure 3.9. The first rule in } \mathcal{E} \text{ shows that a constant is already in WHNF, so the memory is returned unchanged and successfully. In the second rule, for variables, the variable is looked up in the memory and changed into an indirection to the value found, then evaluated. In the function application case the function is evaluated, if that fails due to lack of information the call is failed straight away, otherwise the } \mathcal{F} \text{ scheme is used to perform the application. Note that the "skip" function is used to remove all indirections to the function before using } \mathcal{F}. \text{ The "letrec" case instantiates, in place, each of the definitions and the body with references to the definitions, then updates the letrec node with an indirection to the instantiated body. Finally it}

\[\text{Remember that the top levels of the the program are not instantiated, therefore examination of the memory will yield the definition of any variable found.}
- Skip indirection nodes
  
  \[
  \text{skip} :: \text{Memory} \rightarrow \text{Node} \rightarrow \text{Node} \\
  \text{skip} \ m \ n = \text{case } (m \ n) \ \text{of} \\
  \hspace{1em} (\text{Ind} \ o) \Rightarrow \text{skip} \ m \ o \\
  \hspace{1em} x \Rightarrow n
  \]

- Remove a definition from an environment
  
  \[
  \text{remove} :: \text{Var} \rightarrow \text{Env} \rightarrow \text{Env} \\
  \text{remove} \ v \ \text{env} = \text{filter} (\text{match} \ v) \ \text{env} \\
  \hspace{1em}\text{where} \\
  \hspace{2em} \text{match} \ v \ (x, y) = (v \neq x)
  \]

- Check to see if a free instance of a variable is reachable
  
  \[
  \text{freein} :: \text{Var} \rightarrow \text{Node} \rightarrow \text{Memory} \rightarrow \text{Bool} \\
  \text{freein} \ v \ n \ m = \text{case } (m \ n) \ \text{of} \\
  \hspace{1em} (\text{Const} \ c) \Rightarrow \text{False} \\
  \hspace{1em} (\text{Var} \ v') \Rightarrow v = v' \\
  \hspace{1em} (\lambda \ v' \ b) \Rightarrow \text{False} \\
  \hspace{2em} \hspace{1em} \text{if } v' = v \\
  \hspace{2em} \hspace{1em} \Rightarrow \text{freein} \ v \ b \ m' \\
  \hspace{2em} \text{otherwise} \\
  \hspace{1em} (\text{App} \ f \ a) \Rightarrow (\text{freein} \ v \ f \ m) \ \lor (\text{freein} \ v \ a \ m) \\
  \hspace{1em} (\text{Letrec} \ [v_1 = x_1, \ldots, v_k = x_k] \ \text{body}) \\
  \hspace{2em} \Rightarrow \text{False} \\
  \hspace{2em} \hspace{1em} \text{if } v = v_i, \ 1 \leq i \leq k \\
  \hspace{2em} \hspace{1em} \Rightarrow \text{freein} \ v \ \text{body} \ m \ \lor \\
  \hspace{2em} \hspace{2em} (\bigvee_{i=1}^k \text{freein} \ v \ x_i \ m) \ \text{otherwise} \\
  \hspace{1em} (\text{If} \ a \ b \ c) \Rightarrow (\text{freein} \ v \ a \ m) \ \lor (\text{freein} \ v \ b \ m) \ \lor (\text{freein} \ v \ c \ m) \\
  \hspace{1em} (\text{Constr} \ \text{tag} \ [x_1, \ldots, x_k]) \Rightarrow \bigvee_{i=1}^k \text{freein} \ v \ x_i \ m \\
  \hspace{1em} (\text{Ind} \ x) \Rightarrow \text{freein} \ v \ x \ m
  \]

Figure 3.8: Some useful auxiliary functions used in the schemes
- WHNF evaluator

\[ \mathcal{E} :: \text{Node} \rightarrow \text{Memory} \rightarrow (\text{Memory}, \text{Success}) \]

\[ \mathcal{E} \ n \ m = \text{case} \ (m \ n) \ of \]

\[ (\text{Const} \ c) \Rightarrow (m, S) \]

\[ (\text{Var} \ v) \Rightarrow \mathcal{E} \times m' \]

\[ \text{where} \]

\[ x = \text{lookup}_{\text{Var}} \ v \ m \]

\[ m' = \text{assign} \ m \ n \ (\text{Ind} \ x) \]

\[ (\lambda \ v \ b) \Rightarrow (m, S) \]

\[ (\text{App} \ f \ a) \Rightarrow \mathcal{F} \ (\text{skip} \ f) \ n \ a \ m' \]

\[ \Rightarrow (m', \ F) \]

\[ \text{if} \ r = S \]

\[ \Rightarrow (m', \ F) \]

\[ \text{otherwise} \]

\[ \text{where} \]

\[ (m', \ r) = \mathcal{E} \ f \ m \]

\[ (\text{If} \ a \ b \ c) \Rightarrow (m', \ F) \]

\[ \Rightarrow \mathcal{E} \ b \ m' \]

\[ \text{if} \ r = F \]

\[ \Rightarrow \mathcal{E} \ c \ m' \]

\[ \text{otherwise} \]

\[ \text{where} \]

\[ (m', \ r) = \mathcal{E} \ a \ m \]

Boolean \ \text{bool} = m' \ (\text{skip} \ a) \]

\[ (\text{Constr} \ \text{tag} \ \text{args}) \Rightarrow (m, S) \]

\[ (\text{Letrec} \ [v_1 = x_1, ..., v_k = x_k] \ \text{body}) \]

\[ \Rightarrow \mathcal{E} \ \text{body} \ m_{k+2} \]

\[ \text{where} \]

\[ \text{env} = [(v_1, x_1), ..., (v_k, x_k)] \]

\[ m_1 = \mathcal{I} \ x_1 \ m \ \text{env} \]

\[ :: \]

\[ m_k = \mathcal{I} \ x_k \ m_{k-1} \ \text{env} \]

\[ m_{k+1} = \mathcal{I} \ \text{body} \ m_k \ \text{env} \]

\[ m_{k+2} = \text{assign} \ m_{k+1} \ n \ (\text{Ind} \ \text{body}) \]

\[ (\text{Ind} \ x) = \mathcal{E} \ x \ m \]

- Perform a function application

\[ \mathcal{F} :: \text{Node} \rightarrow \text{Node} \rightarrow \text{Node} \rightarrow \text{Memory} \rightarrow (\text{Memory}, \text{Success}) \]

\[ \mathcal{F} \ f \ n \ a \ m = \text{case} \ (m \ f) \ of \]

\[ (\lambda \ v \ b) \Rightarrow \mathcal{E} \ c \ m'' \]

\[ \text{where} \]

\[ (c, m') = \mathcal{C} \ b \ m \ v \ a \]

\[ m'' = \text{assign} \ m' \ n \ (\text{Ind} \ c) \]

\[ (\text{Const} \ (\text{Prim} \ \text{prim})) \]

\[ \Rightarrow \mathcal{E} \ \text{result} \ m'' \]

\[ \text{if} \ r = S \]

\[ (m', \ F) \]

\[ \text{otherwise} \]

\[ \text{where} \]

\[ (\text{result}, m', r) = \text{prim} \ m \ a \]

\[ m'' = \text{assign} \ m' \ n \ (\text{Ind} \ \text{result}) \]

Figure 3.9: The WHNF evaluator
Copy a graph, instantiating the given variable, whilst maintaining sharing
\( \mathcal{C} : \text{Node} \rightarrow \text{Memory} \rightarrow \text{Name} \rightarrow \text{Node} \rightarrow (\text{Node}, \text{Memory}) \)

\[
\mathcal{C} n m v o = (n, m) \quad \text{if not } (\text{freein} v n m)
\]

\[
= \begin{cases} 
(\text{Const} c) \Rightarrow (n, m) \\
(\text{Var} v') \Rightarrow (o, m) & \text{if } v = v' \\
\Rightarrow \text{new } m (\text{Var} v') & \text{otherwise} \\
(\lambda v' b) \Rightarrow (n, m) & \text{if } v = v' \\
\Rightarrow \text{new } m' (\lambda v' b') & \text{otherwise}
\end{cases}
\]

\[
\text{where}
\]

\[
(b', m') = \mathcal{C} b m v o \\
(\text{App } f a) \Rightarrow \text{new } m'' (\text{App } f' a')
\]

\[
\text{where}
\]

\[
(f', m') = \mathcal{C} f m v o \\
(a', m'') = \mathcal{C} a m' v o
\]

\[
(\text{If } a b c) \Rightarrow \text{new } m''' (\text{If } a' b' c')
\]

\[
\text{where}
\]

\[
(a', m') = \mathcal{C} a m v o \\
(b', m'') = \mathcal{C} b m' v o \\
(c', m''') = \mathcal{C} c m'' v o
\]

\[
(\text{Constr tag } [x_1, \ldots, x_k]) \Rightarrow \text{new } m_k (\text{Constr tag } [x'_1, \ldots, x'_k])
\]

\[
\text{where}
\]

\[
(x'_1, m_1) = \mathcal{C} x_1 m v o \\
\vdots
\]

\[
(x'_k, m_k) = \mathcal{C} x_k m_{k-1} v o
\]

\[
(\text{Letrec } [v_1 = x_1, \ldots, v_k = x_k] \text{ body}) \Rightarrow \text{new } m_{k+1} (\text{Letrec } [v_1 = x'_1, \ldots, v_k = x'_k] \text{ body'})
\]

\[
\text{where}
\]

\[
(x'_1, m_1) = \mathcal{C} x_1 m v o \\
\vdots
\]

\[
(x'_k, m_k) = \mathcal{C} x_k m_{k-1} v o \\
\text{(body', } m_{k+1}) = \mathcal{C} \text{ body } m_k v o
\]

\[
(\text{Ind } x) \Rightarrow \text{new } m' (\text{Ind } x')
\]

\[
\text{where}
\]

\[
(x', m') = \mathcal{C} x m v o
\]

Figure 3.10: Copying for \( \lambda \)-expression instantiation
- Instantiate a graph given an environment

\[ \mathcal{I} :: \text{Node} \rightarrow \text{Memory} \rightarrow \text{Env} \rightarrow \text{Memory} \]

\[ \mathcal{I} n \ m \ \text{env} = \text{case} \ (m \ n) \ \text{of} \]

\[ \quad (\text{Const} \ c) \Rightarrow m \]

\[ \quad (\text{Var} \ v) \Rightarrow m \quad \text{if} \ v \neq v_i, \ 1 \leq i \leq k \]

\[ \quad \Rightarrow \text{assign} \ m \ n \ (\text{Ind} \ x_j) \quad \text{if} \ v = v_j, \ 1 \leq j \leq k \]

\[ \quad \text{where} \]

\[ \quad [v_1 = x_1, \ldots, v_k = x_k] = \text{env} \]

\[ \quad (\lambda \ v' \ b) \Rightarrow \mathcal{I} \ b \ m \ \text{env} \]

\[ \quad (\text{App} \ f \ a) \Rightarrow \mathcal{I} \ a \ m' \ \text{env} \]

\[ \quad \text{where} \]

\[ m' = \mathcal{I} \ f \ m \ \text{env} \]

\[ (\text{If} \ a \ b \ c) \Rightarrow \mathcal{I} \ c \ m'' \ \text{env} \]

\[ \quad \text{where} \]

\[ m' = \mathcal{I} \ a \ m \ \text{env} \]

\[ m'' = \mathcal{I} \ b \ m' \ \text{env} \]

\[ (\text{Constr} \ \text{tag} \ [x_1, \ldots, x_k]) \Rightarrow m_k \]

\[ \quad \text{where} \]

\[ m_1 = \mathcal{I} \ x_1 \ m \ \text{env} \]

\[ \vdots \]

\[ m_k = \mathcal{I} \ x_k \ m_{k-1} \ \text{env} \]

\[ (\text{Letrec} \ [v_1 = x_1, \ldots, v_k = x_k] \ \text{body}) \Rightarrow \mathcal{I} \ \text{body} \ m_k \ \text{env}_k \]

\[ \quad \text{where} \]

\[ \text{env}_1 = \text{remove} \ v_1 \ \text{env} \]

\[ \vdots \]

\[ \text{env}_k = \text{remove} \ v_k \ \text{env}_{k-1} \]

\[ m_1 = \mathcal{I} \ x_1 \ m \ \text{env}_k \]

\[ \vdots \]

\[ m_k = \mathcal{I} \ x_k \ m_{k-1} \ \text{env}_k \]

\[ (\text{Ind} \ x) \Rightarrow \mathcal{I} \ x \ m \ \text{env} \]

Figure 3.11: Instantiation for letrec usage
evaluates the body.

The $F$ scheme deals with evaluating function applications. There are two sorts, lambda expressions and primitives. For lambda expressions, the function body is copied and instantiated using the $C$ scheme, Figure 3.10. This scheme only copies subexpressions that contain references to the variable being instantiated, this ensures full sharing of common subexpressions because expressions without references to the instantiating variable are constant between function applications. When the copy is complete the new body is evaluated. If the function is a primitive, it is applied to the argument. A primitive returns three values: the result node, the updated memory and a success value. If the success value is “F” then the result value is undefined. If the primitive failed then the un-updated memory is returned with a failure, otherwise the result from the primitive call is evaluated.

The instantiation schema, $I$ in Figure 3.11, performs an in-place instantiation of a letrec body. It takes an environment of definitions and changes all references to the definitions into indirections to the relevant definition body. The copy and instantiate schema, $C$, is more interesting. It maintains full subexpression sharing by only copying subexpressions that contain a free reference to the variable being instantiated. The guard on the rule says that the graph is only copied if it contains a free occurrence of the variable being instantiated. Because the whole program has been renamed to stop name clashes the letrec rule does not have to test for capture. However, the lambda rule does need to test for capture as the function may have a recursive call and therefore replicate names. The converse is true in the $I$ scheme. Because a lambda expression’s formal parameter cannot be the same as a definition there can be no capture, but there can be capture of copies of the letrec buried deep in the graph.

The $E$ evaluation scheme presented is extended with the ANF scheme, $A$, shown in Figure 3.12. $A$ is used as the top level evaluator to control the whole simplification process. The effect is to evaluate the arguments to the assertions to WHNF and the arguments to And to ANF. The WHNF evaluator is called twice in the $A$ scheme. For the first instance, in the Node rule, the success value
3.2. Simplification

- Evaluate an annotation to ANF \( \mathcal{A} :: \) Node \( \rightarrow \) Memory \( \rightarrow \) Memory

\[
\mathcal{A} \ a \ m = \text{case } (\ a \ m) \ \text{of}
\]

\[
\begin{align*}
(\text{Constr NoPlace}) & \Rightarrow m \\
(\text{Constr Node } [x]) & \Rightarrow m'
\end{align*}
\]

\[
\text{where}
\]

\[
(m', r) = \mathcal{E} \times m
\]

\[
(\text{Constr Arc } [x,y]) \Rightarrow \mathcal{E} \ y \ m' \text{where}
\]

\[
(m', r) = \mathcal{E} \times m
\]

\[
(\text{Constr And } [x, y]) \Rightarrow \mathcal{A} \ y \ m'
\]

\[
\text{where}
\]

\[
m' = \mathcal{A} \times m
\]

\[
(\text{Ind } x) \Rightarrow \mathcal{A} \times m
\]

\[
\text{annot } \Rightarrow \mathcal{A} \ \text{annot } m'
\]

\[
\Rightarrow \text{FAIL}
\]

\[
\text{where}
\]

\[
(m', r) = \mathcal{E} \ \text{annot } m
\]

Figure 3.12: The ANF evaluator

of the call is not important as WHNF is used to stop aliasing of the stream — all references to a particular stream will be the same once the WHNF evaluator has returned. Provided they are all evaluated as far as they can go, they can be equated. In the second case, the catch-all rule, the success value is important. If the WHNF evaluator fails at this point it means that an NFO did not have enough information to decide on a placement, therefore the simplifier cannot carry on.

\[
\text{nodeise } :: \ [\text{Stream}] \rightarrow \text{Placement}
\]

\[
\text{nodeise } = \lambda ps_c. \ \text{if } a \ ps = [] \ \text{then NoPlace}
\]

\[
\text{else (Node (hd ps)) And (nodeise (tl ps))}
\]

\[
\text{main input } = \text{res moreo v er nodeise } [\text{res}]_d
\]

\[
\text{where}
\]

\[
\text{res } = \text{map inc input}
\]

Figure 3.13: A simple program to simplify

To demonstrate the simplifier at work a small example will be presented. The program simply reads the user input and increments each item, Figure 3.13
shows the input program. What follows are the rewrites needed to simplify
the program. In each case the memory argument, \( m \), is omitted for brevity.
Only some node labels are used for the important sharing. Each line shows
a direct rewrite, or a sequence of rewrites to a particular expression, indented
lines show sub applications of the schemas. The semantics brackets, \([\ ]\), enclose
textual representations of the syntax that the “Memory” type of the schemas
represented. Indirection nodes are represented syntactically as \( \nabla \). The subscript
names attached to pieces of syntax are names of the Node where that particular
piece of graph resides. E.g. \([ if_{a} e_{1} \ then \ e_{2} \ else \ e_{3} ]\) means that the given if
term is at node (location) \( a \), and \([ \ n_{b} \ if_{a} \ ...\ ]\) means that node \( b \) contains an
indirection to node \( a \) (which is the if term). To help understand the evaluation
Figure 3.14 shows the state of the memory at each of the labeled points. Line \( A \)
is the initial state of the program. Line \( B \) shows the state of the graph after the
argument to the first call to nodeise has been instantiated. Each reference to the
argument is left pointing to the argument expression. Line \( C \) shows the first half
of the annotation as it is being evaluated to WHNF. In fact this is as far as the
evaluation goes as the value of input is not known.

\[
\begin{align*}
A & \quad \mathcal{A} \ [\text{nodeise} \ [\text{res}_b]] \\
& \quad \mathcal{E} \ [\text{nodeise} \ [\text{res}_b]] \\
& \quad \vdots \\
& \quad \mathcal{F} \ [\lambda \text{ps}_c . \ if_{a} \ \text{ps} = [] \ then \ ...\] \ n \ [[\text{res}_b]] \\
& \quad C [if_{a} \ \text{ps} = [] \ ...\] m \ \text{ps} \ d \\
& \quad \vdots \\
B & \quad \mathcal{E} \ [\nabla_{a} \text{if}_{a} \ [\text{res}_b] \ d \ a = [] \ ...\] \\
& \quad \vdots \ \text{Delta rules for if and =} \\
& \quad \mathcal{E} \ [\text{Node (hd [res}_b]) \ \text{And nodeise (tl [res}_b])]} \\
& \quad \mathcal{A} \ [\text{Node (hd [res}_b]) \ \text{And nodeise (tl [res}_b])]} \\
& \quad \text{Evaluate first argument of And} \\
& \quad A \ [\text{Node (hd [res}_b])]] \\
& \quad \mathcal{E} \ [\text{hd [res}_b]] \\
& \quad \vdots \ \text{Delta rule for hd} \\
& \quad \mathcal{E} \ [\text{res}_b] \\
& \quad \mathcal{E} \ [\text{map inc input}] \\
& \quad \vdots \ \text{Rules for map (definition not shown)}
\end{align*}
\]
A

\[
\text{main input} = \text{letrec}
\]

\[
\text{res} = \text{map inc input}
\]

\[
\text{in res moreover nodeise [res]}
\]

B

\[
\text{main input} = \text{letrec}
\]

\[
\text{res} = \text{map inc input}
\]

\[
\text{in res moreover if} \bullet = [] \text{ then } [] \text{ else (Node (hd \bullet)) And nodeise (tl \bullet))}
\]

C

\[
\text{main input} = \text{letrec}
\]

\[
\text{res} = \text{if m input = [] then [] else ...}
\]

\[
\text{in res moreover } \bullet \text{ And nodeise (tl \bullet))}
\]

Figure 3.14: Three snapshots of the simplification process.

\[
\mathcal{E} \ [i_{m} \text{ input}=[\text{]} \text{ then } [] \text{ else ...}]
\]

\[
:\text{ Delta rule for if leads to }
\]

\[
\mathcal{E} \ [\text{input}=[\text{]}]
\]

\[
:\text{ Delta rule for } = \text{ fails as input is not known}
\]

\[
\leftarrow \text{Return F (fail)}
\]

\[
\leftarrow \text{Return F (fail)}
\]

\[
\leftarrow \text{Return S (success)}
\]

\[
\text{Evaluate second argument of And}
\]

\[
\mathcal{A} \ [\text{nodeise (tl [res_b])}]
\]

\[
\mathcal{E} \ [\text{nodeise (tl [res_b])}]
\]

\[
:\text{ }
\]

\[
\mathcal{E} \ [\text{if } []=[\text{]} \text{ then NoPlace else ...}]
\]

\[
\mathcal{E} \ [\text{NoPlace}]
\]

\[
\leftarrow \text{Return S (success)}
\]

\[
\leftarrow \text{Return S (success)}
\]

\[
\leftarrow \text{Return S (success)}
\]

\[
\leftarrow \text{Return S (success)}
\]

\[
\leftarrow \text{Return S (success)}
\]

Leaving the annotation as:

\[
\text{Node (if}_{m} \text{ input=[} \text{]} \text{ then } [] \text{ else ...) And NoPlace}
\]

The final program graph left by the evaluation is shown in Figure 3.15.
Converting graphs into trees

Once a program has been simplified the graph used to represent it has various properties that preclude it from further processing by the rest of the sequential compiler:

- Sharing, and more importantly cycles
- Indirection nodes.

The front end therefore needs to transform the graph back into a tree. The first stage is to determine a list of shared nodes in the graph. The result of this stage is a list of sharing records which describe which terms share which expressions. Note that all the points at which sharing takes place will be indirection nodes.

The next stage in the process is to create a named definition for each shared expression in the original graph and then change each reference (an indirection node) to the shared expression into a name node with the new name. To take as a running example:

```
ones = (INT 1) : ones
main ls = res moreover map Node [res,ones]
   where
       res = process ones ls
```

After simplification the graph would look like:
With this graph, a sharing list with two elements is constructed, one element representing the sharing of \texttt{res} and the other representing the sharing of \texttt{ones}. For each of these shares a new definition is created, leaving the following tree:

\[
\begin{align*}
\text{ones} &= n1 \\
\text{main ls} &= n2 \text{ moreover Node n2 And Node n1} \\
\text{where} \\
\text{res} &= n2 \\
\text{n2} &= \text{process n1 ls}
\end{align*}
\]

If a shared expression is found to be named already, then the existing name can be used instead of providing a new name. This would simplify the program and result in a form of the annotation that the programmer may have written had \texttt{map} not been used in building the annotations.

The important part of the algorithm is deciding where to place the new definitions. Because of the restricted way in which the program was initially partially evaluated, locating the new definitions is not difficult. There are only two scope levels that pointers to shared objects can originate from; the top level of the program and the letrec inside the \texttt{main} definition. This is because when we evaluate expressions, letrec structures are instantiated into direct pointer sharing so any letrecs that would have “made” more scope levels disappear during evaluation. If an expression was not involved in the evaluation process then it can’t possess pointers to shared objects.

Each node in the evaluated graph has a flag that defines its initial position, before evaluation. The flag can have one of two values:

\textbf{\texttt{OUTER\_YES}} Means that the node came from the outer environment.

\textbf{\texttt{OUTER\_NO}} Means that the node came from the inner environment.

When an expression is given a name it is placed in either the outer or inner letrec according to the flag in the top node in the expression. To avoid any possible capture all the definitions in the outer letrec are renamed.
Simplification termination properties

As mentioned at the beginning of this section, there are two failure modes for the simplifier:

1. **Not enough information**
   
The annotation relied on runtime parameters to the program. This manifests itself as a primitive returning a failure whilst the simplifier is evaluating the annotation to ANF.
   
   If a primitive fails whilst a stream is being evaluated to WHNF, this is not treated as failure.

2. **Non termination**
   
   There are two reasons for non termination:
   
   - The annotation described is infinite.
   - The annotation computation contains an infinite loop.
   
   Because of the nature of the simplifier, these problems cannot be avoided. Wherever a recursive language is available to a programmer to describe a structure or computation, non termination will possible.

### 3.2.2 Simplification Recapitulation

The simplifier is a partial evaluator that targets the annotation expression. It evaluates the annotation to ANF, i.e. the placement assertions are made explicit and all the stream expressions are in WHNF. In the process of doing this the tree structured program representation is made into a graph. To allow the program to continue through the compiler it has to be restructured back into a tree.

### 3.3 The Network Extractor

The FAST system has a primitive called procnet that implements a static process network given a process list and a wiring list; this is demonstrated in figure 3.16.
procnet \([f_1, f_2, f_3, f_4, f_5]\)
\[
\begin{align*}
\&\quad \begin{pmatrix}
(1,1),(2,1), \ (2,1),(3,1), \ (2,1),(4,1), \\
(3,1),(5,1), \ (4,1),(5,2), \ (5,1),(0,1)
\end{pmatrix}
\end{align*}
\]

Figure 3.16: An example \texttt{procnet} call, with generated network.

The first argument to \texttt{procnet} is a list of functions that define the behaviours of each of the processes in the network. Each function has the following type:

\[
\text{type Process} = [\text{Stream}] \rightarrow \text{Stream}
\]

The argument is the list of \texttt{Stream}s that the function consumes, i.e. those streams that the function uses that have been placed elsewhere.

The second argument to \texttt{procnet} is a "wiring diagram" that links processors together. It is formed as a list of links. Each link is specified by a pair of pairs of numbers, \([(a,b),(c,d)]\), where \(a\) and \(c\) are the source and destination processors respectively (they are indexes into the process list). Values \(b\) and \(d\) are ports on the source and destination processors, used to differentiate each input and each output of a processor. It should be noted at this point that each processor has only one output stream because each processor evaluates only one expression.
We can define the functional behaviour of \texttt{procnet} as follows:

\begin{verbatim}
procnet ps wl = f
    where
    f input = output
        where
        [output] = parameters 0
        results = [ (ps !) m) (parameters m)
            | m ← [1..numproc] ]
        parameters m = [ (input : results) !! assoc (m,n) wl
            | n ← [1..(numin m)] ]
        numin proc = max [ n | ((i,j),(m,n)) ← l ]
            where
        l = filter (\(\lambda(c,\_)) . c = proc) wl
        numproc = length ps
        assoc k ((x, k') : xs) = if k = k' then x
            else assoc k xs
\end{verbatim}

Here, "!" denotes list subscripting, and "!!" denotes subscripting a list using the first of a pair of integers. The function \texttt{assoc} is used to find the output corresponding to each of a process's inputs. The function \texttt{parameters} collects all the input streams for the process numbered \(m\). The list \texttt{results} is the result of applying all the input streams to their respective processes. Process 0 is a special input-output pseudo-process; its input is the process network's output, and its output is the process network's input. Therefore, the result of the network is \texttt{parameters 0}.

The \texttt{procnet} function can represent any static, functional process network, although the current implementation can only support processes with a single output. The job of the network extractor is to convert an annotation into a call to \texttt{procnet}. A skeletal example will be used to demonstrate how this is done: (\(f\) and \(g\) represent general functions over their stream arguments)

\begin{verbatim}
main ls = result moreover Node result And Node b
    where
    result = f b
    b = g ls
\end{verbatim}
In order to call procnet we need a wiring list and a process list. The wiring list is easy to construct once we understand the data dependencies of the annotated program. The process list is harder. We must construct one function for each node in the process graph. These functions are of Process type. Each function should take as its arguments those streams that it imports and its result is the stream that the node computes.

### 3.3.1 The Process Placement Rule

Exactly what has to be computed and imported for each node needs to be known in order to perform the transformation to produce the node functions. This is defined by the process placement rule. It states that “a value is computed locally unless it has been explicitly placed elsewhere.” That is to say, everything is computed locally unless it has been the subject of a Node assertion in the annotation. Also, each of these expressions is only referenced through its name. A where clause definition names nodes in the program graph and thus allows sharing and avoids duplication.

The example above would appear something like this after extraction:

```haskell
main ls = procnet [result', b']
    [((0,1),(2,1)), ((2,1),(1,1)), ((1,1),(0,1))]
    where
    result' [b] = f b
    b' [ls] = g ls
```

As you can see the transformation is very similar to λ-lifting: exported streams are lifted out of their contexts and re-formed as parameters.

### 3.3.2 The Details of Network Extraction

There are several assumptions about the form of the program that is presented to the network extractor which are all ensured by previous phases. Firstly, all expressions that are subject to Node assertions are named. Secondly, the names refer to expressions that are in WHNF. Thirdly, each of these expressions has a
unique name. In other words in the annotation "Node a And Node b", a and b must refer to different streams because they have different names.

Several simplifications were made for the pilot implementation, these relate to where the annotation can be placed and how the argument to main is used. The moreover clause can only be in the definition of main, also main cannot be called recursively. To allow moreover to be situated anywhere else would mean that there is a possibility it could be “called” more than once during execution, e.g.

\[
\begin{align*}
f [] &= [] \\
f n &= \text{res moreover Node res} \\
&\quad \text{where} \\
&\quad \text{res} = g n \\
\text{main ls} &= f \text{ls}
\end{align*}
\]

The semantics of Basic Caliban cannot ascribe a meaning to this program. If g calls f, what should that mean? Is a process network constructed each time f is called or is the annotation from only the first call to f used?

The second restriction has to do with the implementation of the rest of the system. The target architecture is a distributed-memory multiprocessor that is hosted by a Unix machine. The backend implementation does not allow any application code to be executed on the host, meaning that all code must be executed on the parallel machine. Therefore, all computation must be placed explicitly. However if the programmer were to write:

\[
\begin{align*}
\text{main ls} &= f \text{ result moreover Node result} \\
&\quad \text{where} \\
&\quad \text{result} = g \text{ ls}
\end{align*}
\]

then the application of f is not specified to be performed on the network. The programmer might expect that the application of f would be performed on the host after it received the result from the network. As the backend implementation cannot cope with this, a transformation is used to move the computation of f onto the network. In other words the program would become:
\[
\text{main} \text{ is } \text{realresult} \quad \text{moreover Node result And Node realresult}
\]
\[
\text{where}
\]
\[
\text{realresult} = f \text{ result}
\]
\[
\text{result} = g \text{ ls}
\]

With these two restrictions in mind the implementation details of the extractor can be given. The program combines a simple data dependency analysis with a stream “lifter”. The data structures maintained are described in the following two sections.

The process list

```c
struct stab_list {
    stab_handle sl_old_name; /* Old name of an expression. */
    int sl_scope; /* Which scope are we defined in. */
    struct new_names *sl_abstraction; /* Abstraction for this name. */
    int sl_output; /* Is the node’s output used? */
    struct s_expression *sl_def; /* The name’s definition point. */
    struct stab_list *sl_next;
};
```

The structure stores what will be turned into the process list in the call to `procnet`. Because each expression that is of interest is uniquely named, we use the name to refer to it rather than its address. This helps when displaying error messages. `sl_old_name` refers to the original name, as used by the user’s program. `sl_scope` is a number that defines the scope depth at which the name was defined. `sl_def` points to the place at which the name was initially defined. Taking this and the name, each named expression can be uniquely identified.

The `sl_abstraction` field is the name of the process formed, together with the newly created arguments, representing the streams that the process imports. The boolean, `sl_output`, is used to emit a warning if a process’s output is not used.
The annotation

```c
enum annot_type {an_arc, an_node};

struct annot {
    enum annot_type an_type;  /* Which annotation. */
    int an_used;  /* Have we used this annotation? */
    stab_handle an_a, an_b;  /* Arguments for the annotation. */

    struct annot *an_next;
};
```

The annotation presented in the *moreover* clause is converted into an internal form, held in this data structure. *NoPlace* annotations are ignored and not translated. Each part of the annotation is flagged with whether it has been used or not. During the extraction we mark each *Arc* annotation component when a link between its two arguments, in either direction, is found. At the end of the extraction process, warning messages for each unmarked *Arc* can be issued. *Node* assertions are used automatically as we use them to drive the extraction.

### 3.3.3 The code

The routine `extract()` performs the extraction. Its first job is to locate the *moreover* clause. It then tries to determine the name of the argument to *main*, if it has one. A pattern is not allowed as the argument to *main*.

The list of names that will appear in the final process list can be constructed from the full annotation. A functional definition of its construction looks like:

```c
NodesOf NoPlace = []
NodesOf (Node a) = [a]
NodesOf (Arc a b) = []
NodesOf (a And b) = uniq ((NodesOf a) ++ (NodesOf b))
```

The function `uniq` removes duplicate names from a list. In other words `NodesOf` produces a list of all streams that are the subject of a *Node* assertion.
The next stage is to call `moreover2procnet()` and actually perform the extraction transformation on the program. This runs through the nodes list extracting each expression named with respect of the other exported expressions. References to exported names are made into arguments for the particular expression. References to that expression are made into a function application of the lifted expression and the objects that are lifted, and the lifting transformation is applied to the new expression to lift out any newly introduced references to exported names. The routine that performs this task is called `rec_abstr()`, which will be described in more detail shortly. The process list structure is updated with information as this transformation proceeds.

A wiring list, as used by the `procnet` call, can be constructed once the program has been lifted. The information from the process list is used and various consistency checks are performed at the same time.

The final operation is to replace the `moreover` block with a call to `procnet`.

**The actual extraction transformation**

The function `rec_abstr()` is at the heart of the network extraction transformation. It effectively defines the semantics of Caliban by deciding what is imported onto a processor and what is recomputed locally.

The function is given a name which it should extract. This name needn’t be a placed stream, it could be an intermediate definition. Firstly, therefore, it must find the definition of this name. It can then recursively run through the definition lifting the uses of exported streams to create what is called an abstraction environment. If this environment is empty then no lifting is needed and it is finished. Otherwise a version of the expression as a function with the imported names lifted out is constructed. For example:

```plaintext
main ls = result moreover Node result And Node a
    where
    result = f a
    a = g ls
```
After extraction becomes:

```
main ls = procnet [result', a'] [ ((0,0),(1,1)), ((1,1),(2,1)),
                     ((2,1),(0,0)) ]

  where
  result' [a] = f a
  a' [ls] = g ls
```

This is adequate for simple examples, but for something slightly more complex, recomputation can be introduced where it needn’t be. For example:

```
main ls = result moreover Node result And Node a
  where
  result = f b b
  b = h a
  a = g ...
```

This translates directly to:

```
main ls = procnet [result’, a] [ ((0,0),(1,1)), ((1,1),(2,1)),
                     ((2,1),(0,0)) ]

  where
  result’ [a] = f (b’ [a]) (b’ [a])
  b’ [a] = h a
  a = g ...
```

Even though b is not exported it must still be extracted and treated like any other extraction. This is an example of why the extractor must be called on newly constructed applications, i.e. b’ [a], so that references to a are bubbled to the top of the process definition. As a result of b being extracted and used twice in the definition of result, recomputation has been introduced, the two calls, b’ [a], are computed separately and not shared.

To achieve the desired sharing the multiple applications are changed into two references to a single named application, giving:
main $ls = \text{procnet} \ [\text{result}', \ a] \ [\ ((0,0),(1,1)), \ ((1,1),(2,1)), \ ((2,1),(0,0))]$

\begin{verbatim}
where
result' [a] = f b b
  where
    b = b' [a]

b' [a] = h a
a = g ...
\end{verbatim}

One further optimisation is not to extract streams from subexpressions that are defined inside the scope of the main expression that is being extracted. In the example above, if $b$ had been defined in a \texttt{where} clause for \texttt{result} then it would not be extracted, thus reducing the plumbing of imported streams. Extraction would have left the following program:

\begin{verbatim}
main $ls = \text{procnet} \ [\text{result}', \ a] \ [\ ((0,0),(1,1)), \ ((1,1),(2,1)), \ ((2,1),(0,0))]$

where
result' [a] = f b b
  where
    b = h a

a = g ...
\end{verbatim}

One final problem remains. If a process references itself, e.g.

\begin{verbatim}
main $ls = \text{res morev Node res}
  where
    res = f $ls res
\end{verbatim}

then the extracted program would look like this:

\begin{verbatim}
main $ls = \text{procnet} \ [\text{res'}] \ [\ ((0,1),(1,1)), \ ((1,1),(1,1)), \ ((1,1),(0,1))]$

where
res' [$ls$, $res] = f $ls res
\end{verbatim}

This is inefficient because the recursive reference is implemented as a communication link.
Chapter 3. The Implementation of Basic Caliban

The self reference should be “inside” the process:

To solution is to spot this case and use a letrec to name the result of the computation so that it can be used directly, leaving the following program:

```haskell
main ls = procnet [res'] [[(0,1),(1,1)), ((1,1),(1,1)),
                          ((1,1),(0,1))]
  where
  res' [ls] = res
  where
  res = f ls res
```

### 3.4 Runtime System

Once a program has passed through the frontend phases of simplification and network extraction, it is compiled using a normal sequential compiler. The code can then be executed on the parallel machine.

This section describes the rest of the compiler, after the frontend. It presents the provision that the runtime system needs to make for the Caliban system and shows how parallelism is achieved.

Firstly, the target architecture of this implementation needs to be described. The pilot implementation was built for the Meiko CS-1 with 32 T800 transputers connected in a 2D mesh. One of the cells on the parallel machine is connected to the host computer, the host is used to build the processing network and to supply it with input data.

Meiko provide a high level programming toolkit called CSTools [68], which is the interface that the system is written in. It provides arbitrary node to node communication on top of the physical 2D mesh using virtual circuits called “transports.” It also provides a high level network building operation called
3.4. Runtime System

cs_load(), which takes a description of the processor network wanted, and loads it onto the transputer surface.

Once network extraction has finished with the user’s program it is in the form of a single call to the procnet primitive. The operation of each node in the intended process network is described by a function in the resulting program. The program is now compiled using the standard sequential FAST compiler [41]. The compiler can use any optimisation on the program to improve its sequential performance. The only difference compared to the normal sequential compilation route is that a modified version of the runtime system is linked with the output of the FAST compiler. In fact there are two versions of the runtime system. One for the host process and one for the cell processes. After, linking the runtime systems there are two executable programs; one for the host and one for the cells, the application code of these two programs is identical.

The rest of this section details the extensions to each of the runtime systems, starting with the host runtime system.

3.4.1 The host runtime system

The host program runs on the host computer of the CS-1. It is responsible for loading the machine with the program, configuring the links between cells, supplying the machine with input and and consuming its output. When the program runs, after initialisation, the evaluator calls the user-defined function main. This is the call to procnet generated by the network extractor. The host version of procnet reads the process list and wiring list and builds the CSTools structure to represent the process network on the machine. The structure describes the connections required between the cells and which executables should be used as the code for which cells, in fact, the same executable is used for every cell. The information about the cell connections is contained directly in the wiring list parameter of procnet.

Once the build structure has been built the host can call the CSTools routine to execute it. Figure 3.17 shows the code used to extract the network from the
/* Build the processor groups */
thisw = snd(red(wireList));
for (i = 0; i < numofWires; i++) {
    connection = _11_HEAD(thisw);
    src = get_src_node(connection);
    dest = get_dest_node(connection);
    if ((src & dest) & necessary(src, dest, connect_array))
        cs_connect(grpArray[src], grpArray[dest]);
    thisw = _11_TAIL(thisw);
}

/* Commit the groups */
cs_option(grpArray[0], "commit", "host");
for (i = 1; i < numofProcs; i++) {
    cs_option(grpArray[i], "commit", "transputer");
}

/* Run the program */
cs_load();

Figure 3.17: Building the process network on the host

wiring list, tell CSTools about it and then execute it. The calls to the reducer are
to ensure that the arguments are in the correct form for the code, they should
be constants. The runtime functions _11_TAIL() and _11_HEAD() are used to
destruct the wiring list evaluated by the reducer.

CSTools will attempt to match the desired interconnections of the program
to physical connections of the machine. Any logical links it cannot map onto
physical links are implemented by software message routing. This is inefficient
on the transputer as the processor has to be interrupted from computation to
forward a message to a neighbouring processor.

Once the network is loaded the host then communicates with the cells on the
network that are designated as the source and sink for the network. One cell
will demand the input to the network and another will produce the output of the
network. In fact, there is another process running on the host that performs the
IO operation as the cs_load() function does not return until the process network
is finished. This second host process is built as part of the build structure. It
uses a third version of procnet that simply connects with the process network and passes data back and forth.

### 3.4.2 The cell runtime system

Each cell runs the same executable. As with the host, the first operation to be performed by a cell is calling procnet. The cell version of procnet firstly reads a number from the host which informs them which cell they are in the process network. They then read their process and wiring list arguments. From the process list they extract the function that represents the work that they are to perform by indexing into the list with the cell identifier that was retrieved from the host. From the wiring list they work out which other processors they need to communicate with. Then, they can set up their input and output streams. Once this initialisation has been performed, control passes to the evaluation loop, which in turn, evaluates and sends results to consuming processors.

**Input streams**

Input to a node is performed using the primitive **ReadStreamItem()**. It reads a stream element from the specified source and builds a cons cell with the new item for the head and a replicated call to itself in the tail. Figure 3.18 shows how this works. During each call to **ReadStreamItem()** a request for the next item is sent to the producing processor. If the program is properly load-balanced, the requested data should arrive at the consumer before **ReadStreamItem()** is next called, thus completely overlapping computation and communication.

![ReadStreamItem Diagram]

**Figure 3.18: Reading values from a stream**
The initial call to proclnet builds a suspension of ReadStreamItem() for each input stream for the process and passes it as one of the arguments to the computation function that the cell is evaluating. In this way each input is built.

Output streams

The output side of communications is slightly more complex. The output stream of a process may be consumed by more than one consumer. Each of the consumers may want to consume the stream at a different rate. Kahn [52] showed that if a node in a graph of sequential functional processes is starved of input (or execution time) then the program may only produce partial output, although the output produced would be correct. This has an implication for processes in a Caliban program. If two processes, A and B, are consuming the output from a third process, C, then the rate at which each consumer consumes must be allowed to be independent. Process A must not have to wait for process B to be ready to receive an element of the stream before it receives it itself. If the receiving rates are not allowed to be independent then deadlock can result.

With these semantic concerns in mind the Basic Caliban runtime system is implemented as follows. Each consumer has a transport allocated to it. A non-preemptively scheduled process is created to serve each of the output transports for a processor. These processes are responsible for calling the evaluator when their consumer, at the other end of the transport they represent, demands output. The evaluator is locked so that only one process can call it at once, therefore the process that demands the next uncomputed value first is the process that calls the evaluator. All other processes that subsequently require the, as yet, uncomputed value have to wait until it is ready. Figure 3.19 shows what happens during a typical evaluation. Each process points to the next value to be transmitted down its transport. In this diagram process 2 is ahead, it is the process that is causing the next head of the stream to be evaluated. The other processes are reading previously computed values from the heap — the heap automatically provides buffering of previously computed values.
Garbage Collection

None of the versions of Basic Caliban ever included a garbage collector. However, because the nodes only communicate by streams their memories are independent, which means that each node could garbage collect independently of the others. All that would be needed is a standard garbage collector with an extended root set, where each output process’ pointer into heap is added.

3.5 Performance Evaluation

The section presents a short description of the performance of the Basic Caliban system. The system was never fully finished for several reasons.

1. the initial performance indicated from the system was very poor,
2. a new and more interesting target machine became available and
3. the source level language was not expressive enough and changes there involved changes to the runtime system which meant a considerable redevelopement.
To demonstrate the performance of the system, the static farm version of the ray tracer from section 2.3.1 of chapter 2 was run. The problem size was restricted to a 12×12 image grid with only six objects in the scene database. Given these parameters, Figure 3.20 shows the performance for various numbers of processors; these figures do not include the configuration or loading time.

The implementation suffered from several performance drawbacks:

1. No garbage collection

2. Very small cell memory

3. An experimental RTS — the code was heavily instrumented by the FAST team to measure various aspects of sequential performance, thus reducing overall performance

4. Huge load times

The cells had four megabytes of memory each, but after the operating system had loaded the user program was left with approximately two megabytes. The executable program for the cell ray tracer was nearly one megabyte in size, therefore
consuming half of the available memory. This meant that the functional program had only one megabyte; without garbage collection this would not be enough for anything but the smallest program — many experiments failed because of lack of memory. The Meiko does not have a broadcast mechanism, so each cell must be individually loaded from the host, leading to load times as high as 10 minutes. The problem could not be made to run in less than six processors because of the lack of garbage collection and the small available memory.

3.6 Conclusions

This chapter has described the implementation of Basic Caliban. It consists of three main stages:

1. Simplification

   - Two different evaluators: \( E \) and \( A \).
   - Dealing with the lack of information.
   - Converting graphs back to trees.

2. Network Extraction

   - Building a function for each process, by lifting stream references.

3. Runtime System

   - \texttt{procn}et interface.
   - Task per consumer, driving evaluation of a single expression.

The important contributions of this chapter are the simplification and network extraction algorithms. In the following two chapters, Caliban will be extended allowing a larger class of problems to be solved efficiently. This starts, in Chapter 4, with an analysis of Basic Caliban leading to the development of an improved language: Advanced Caliban. The Advanced Caliban implementation is split into the same phases as the Basic Caliban implementation. Simplification
for the new language is just an extended version of the Basic Caliban’s simplification. Whereas, network extraction will be completely redesigned to cope with the new requirements.
Chapter 4

Advanced Caliban

Basic Caliban, as presented in Chapter 2, is rather limited. It only allows for the construction of simple process graphs, with few useful features for parallel programming such as computation/communication overlapping and generalised graph contraction. The process of designing and building the prototype language did however demonstrate some relevant techniques, such as partial evaluation for the annotations and the use of annotation programs to make placement decisions at compile time.

This chapter presents an evaluation of Basic Caliban and then continues by describing an enlarged version of Caliban, called Advanced Caliban, motivating the design by example. It has all of the basic features of its predecessor but has three additional areas of functionality that make it a much better parallel programming language:

- Multiple output processes — more than one result stream per processor
- Easier annotation construction and positioning
- Support for phased computation

The chapter starts with a short evaluation of Basic Caliban to uncover its weaknesses. These weaknesses are then individually discussed in detail, motivating a set of extensions which are then described. Finally the new language is summarised.
4.1 An evaluation of Basic Caliban

This section is an analysis of the strengths and weaknesses of Basic Caliban. It describes the positive and negative features of the language as well as what was learned from developing it.

Basic Caliban aimed to be a fully declarative parallel programming language. Its design avoided any impure features. The greatest feature is that the annotations are built up in the source language. This allows the programmer to use objects from the application program to influence the annotation that is produced. It also gives the programmer the same abstraction mechanisms that the source language has to build NFOs and annotations. Common network forms can be built as NFOs and stored in libraries for reuse.

The language generates static process networks at compile time, this avoids all the overheads of dynamic process creation at runtime. Also, because the compiler fully unravels the process network and therefore all possible communications, at compile time, it could do some optimisation on it. E.g. it could minimise the communication distance by embedding the process network onto the target machine carefully.

There are, of course, problems with Basic Caliban. By having the annotations written in the source language there is some difficulty in knowing how much partial evaluation to perform on the program at compile time. It is possible to write programs that are fully evaluated at compile time! In a similar vein, it is possible to write programs that the compiler will loop with. The static nature of the language also leads to a need to recompile the program if anything that affects the parallelism changes.

The language cannot describe all possible process networks. Also, parallelism cannot be encapsulated sufficiently, annotations are dealt with in a very raw fashion.

Having said all this, the language is novel and interesting. With development some of its ills can be cured, leaving a useful programming language. The rest of this chapter addresses the flaws in Basic Caliban and attempts to put them
4.2 Multiple Output Processes

This section will deal with the first extension to the Caliban language, allowing multiple output streams from a single computation. It will start by motivating the need for such extensions, using examples, and finish with the implemented extension.

4.2.1 Example: Selector functions and friends

Basic Caliban only allowed each process to compute a single output stream. It could be sent to more than one consumer, but it was only one stream of data. This led to a problem when trying to set up various parallel programming forms. For example, you cannot write an efficient static farm function that accepts a stream of jobs and hands them out to a set of workers, as seen in Figure 4.1. The Basic Caliban code for a farm is shown in Figure 4.2.

```
farm :: Int → (a → b) → [a] → ([b], Placement)
farm n f jobs = (results, fan results generated)
  where
  results = unpartition generated
  generated = map (map f) partitions
  partitions = partition n jobs
```

Figure 4.1: The required structure of a farm

Figure 4.2: The Basic Caliban code for farm
Figure 4.3: The farm network that results from the code in Figure 4.2

```
farm :: Int -> (a -> b) -> [a] -> ([b], Placement)
farm n f jobs = (results, fan results generated And fan jobs partitions)
    where
    results = unpartition generated
    generated = map (map f) partitions
    partitions = partition n jobs
```

Figure 4.4: An alternative coding of farm in Basic Caliban

This may look fine, until you examine exactly what gets placed where and what is communicated. The annotation as it stands will place the farm workers on separate processors, which will each receive all the jobs and which will each have to execute the partition code to extract the particular jobs that they need to execute. This works by lazily executing the partition application, generating the spine of the generated list and then pulling only on the sublist that they are meant to work on. The full list of jobs they are being supplied with is then used up by extracting every n'th element and discarding the rest. Figure 4.3 shows the process network that results.

You can see that an excess of communication is being performed, as all the jobs are being sent to all processors, and there is duplicated computation of the partition function, with all the processors calling it. If partition were a complex function then this duplication could have serious performance overheads, reducing the performance of the worker. This performance problem could be overcome by placing the partitioning on a separate set of processors, as in Figure 4.4.

Figure 4.4 shows how an extra layer of processes can perform the selection. Each one still receives the complete list of jobs, but the workers have been protected from the extra work of selection at the expense of using more processors.
These problems can be boiled down to a simple observation: Basic Caliban cannot control the replication of work and communication sufficiently well. We should be able to write the `farm` function without replicating the job stream or the selection computation, in other words, produce a process network that looks like Figure 4.1. To do this, it must be possible to define nodes that compute more than one stream.

### 4.2.2 Example: Load balancing

Chapter 2 showed how computations in a pipeline network could be balanced by adding and removing assertions from the annotation. Unfortunately, this is not a general property of Basic Caliban. Load balancing of more general structures is not easily possible. It turns out that Basic Caliban cannot load balance any network that is non-linear (i.e. has a fan-out greater than one). This becomes obvious when you remember that nodes in Basic Caliban process networks can only have a single output stream (although the single stream can be copied to more than one consumer).
Take a tree structure, an example of which can be seen in Figure 4.6. The tree could be a network to calculate residuals for a large plane of values during a Local Neighbourhood Operation (LNO) computation (see section 6.2), a sorting program or any other type of tree structured computation. In the LNO case, the values (residuals from each patch of the plane) would enter at the leaves of the tree and would be combined by higher nodes until a final value would be emitted at the root. A sample section of code to perform this is in Figure 4.7. This style of process network can be balanced successfully by Basic Caliban as the fan-out of nodes is one and communication load down each arc in the network is the same.

\[
\text{main l} = \text{residuals moreover annot}
\]

\[
\begin{align*}
\text{where} \\
(\text{residuals, annot}) &= \text{treefold f (makeResiduals patches)} \\
\text{patches} &= \langle \text{LNO computation} \rangle
\end{align*}
\]

\[
\text{treefold} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow (a, \text{Placement})
\]

\[
\text{treefold f [s]} = (s, \text{Node s})
\]

\[
\text{treefold f streams} \quad = (\text{combined, Node combined And fan combined [a, b] And annota And annotb})
\]

\[
\begin{align*}
\text{where} \\
\text{combined} &= f a b \\
(a, \text{annota}) &= \text{treefold f suba} \\
(b, \text{annotb}) &= \text{treefold f subb} \\
\text{(suba, subb)} &= \text{split streams}
\end{align*}
\]

Figure 4.7: A sample use and definition of \texttt{treefold}

The second example (a tree-sort computation) has the process network being used in the opposite direction. Elements are sent down from the root and routed through the tree as needed to end up at the leaves in their correct “bucket”, from where they can be collected. One may wish to balance the communication of this program so that the new arcs carry the same load as each other. But, as this problem has a fan-out of two, it is not amenable to load balancing with Basic Caliban.

We could also perform a merge sort operation, by sending values in at the
leaves and sorting sublists there, then merging the sublists up the tree until a fully sorted list emerges at the root. Although this network has a fan-out of one, the ideal grouping would be depth-wise, i.e. all nodes at the same depth would be computed by the same processor because the communication and therefore the computation load would be the same (there is an invariant that each layer deals with the same number of elements during each sort).

Figures 4.8 and 4.9 show two different load balancings that may be desirable. The second, by depth, is the one to use for the merge sort example.

If problem-specific details are removed from consideration, it can be seen that there are variables that affect the ability to load balance. Firstly there is the direction of data flow, root to leaves (RL) or leaves to root (LR). Then there is the communication load: does it increase, decrease or stay the same between layers? In other words, if you take a subtree (e.g. nodes A, B and C of Figure 4.6), is the communication done by B and C more, less or the same that done by A.
We always order the relation from the parent to child for consistency. E.g. if the parent splits an input list between its two children then the parent has a higher communication load than the children. Similarly if it copies the elements to both children then the load is the same.

Finally there is the computation load at the layers. This is usually proportional to the communication load, but can be different. The same terminology can be used to refer to the computation load as is used for the communication load.

The only trees that Basic Caliban can properly load balance are those that are merging (LR-trees) and have communication and computation loads that are constant between layers, i.e. the parents have the same communication and computation loads as each of their children. This is because the only reduction operation that can be performed on the tree is to group subtrees together as one node (i.e. removing Node assertions from the annotation) as this is the only operation that maintains a fan-out of one.

### 4.2.3 Possible solutions

Examples of how Basic Caliban cannot cope with all the network structures that might be needed have already been presented. Indeed, some of them are needed quite regularly and are basic to parallel programming. The problem boils down to the fan-out available to Basic Caliban nodes. To be able to implement an efficient splitting function (as used in the farm example), a node that has a fan-out greater than one is needed. To be able to balance a tree computation by layer, a node that has a fan-out greater than one is also needed. To be able to build any directed, cyclic graph, nodes that have arbitrary fan-outs are needed.

Can these problems be solved within the confines of Basic Caliban?

One possibility to solve the fan-out problem is to use tuples of values. The split node, from the farm example, takes a stream of jobs and generates a stream of $n$-tuples of jobs where $n$ is the number of workers. The $i$th worker could then extract the $i$th element of each tuple as their work. This is a typical Caliban
program transformation: a tuple of streams, which is the output wanted from the split node, is converted into a stream of tuples. This solves only one problem, that of redundant computation (the splitting is only performed once now). But, it leaves the problem of redundant communication. All the jobs are still being sent to all the workers. Of course additionally, each worker now needs to perform a simple selection on the tuple for each job.

There are other problems with this solution. The number of elements and rate of output or consumption of each of the output “streams” of the split node needs to be the same because a tuple can only be sent when all of the contributing “streams” can supply a value. In other words this solution could not be used to implement a routing algorithm (as used in the sort above), as the number of elements that are routed to each branch will vary according to the input.

It seems that there are no solutions to the fan-out problem within the confines of Basic Caliban. We need to extend the language to cope with these problems.

The Arc could be extended so that the streams can automatically be selected from the tuple at the generation end and thus generate multiple output streams. This would cure the communication redundancy problems, but would still leave the stream length and rate issue. Also it would mean that the annotation would change the declarative meaning of the program.

A similar solution would be to provide map (sel n) (the operation used to extract items from the stream of tuples) as a primitive. If this were so, the system could automatically push the selection back into the producer and solve the problem that way. Again, this leaves the stream length and rate problem.

4.2.4 Bundles

The basic idea underlying Caliban is the stream of data. It represents progression of computation. When a stream is consumed, computation happens at the remote end to calculate further values for it. The objects that perform the calculations are nodes, they are connected by arcs to other nodes and streams of data flow over the arcs to supply the consumer with the data required. Once connected these
nodes and arcs form a (possibly) cyclic graph. At present there is a restriction that the same data must flow out of all the output streams of a node, i.e. a node can only compute one stream, but several nodes can consume copies of that stream. What is needed is a mechanism that allows multiple different streams of data to be exported from a single node. We could then implement proper splitting functions.

The solution to the problem is to remove the Node assertion from the language and replace it with a new assertion. Bundle \([x]\) places the stream \(x\) on a new processor. This means that

\[
\text{Bundle } [x] \equiv \text{Node } x
\]

Figure 4.10 shows a simple program and its process network. But, Bundle does more than this, it takes a list of streams and places them all on the same processor. If the annotation from Figure 4.10 in changed to bundle the two intermediate computations onto the same processor then the process network will be changed to that shown in Figure 4.11.

\[
\text{main input } = \text{res} \\
\text{moreover Bundle } [\text{res}] \text{ And Bundle } [\text{inter1}] \text{ And} \\
\text{Bundle } [\text{inter2}] \\
\text{where} \\
\text{res } = f \text{ inter1 inter2} \\
\text{inter1 } = g \text{ input} \\
\text{inter2 } = h \text{ input}
\]

Figure 4.10: A simple program and its network

If one of the streams in a bundle is already placed then all the other streams are located on the same processor as that stream, otherwise a new location for the bundle is used. This leads, by transitivity, to the following equivalence:

\[
\text{Bundle } [a,b] \text{ And Bundle } [b,c] \equiv \text{Bundle } [a,b,c]
\]
main input = res

moreover Bundle [res] And Bundle [inter1, inter2]

where
res = f inter1 inter2
inter1 = g input
inter2 = h input

Figure 4.11: A bundled process network and code

This means that the computation of a stream cannot be replicated onto two processors, if two bundles have the stream in common then all streams in both bundles are computed on the same node. Bundle cannot be defined in any other way as streams cannot be replicated as there would be no way of discriminating between the different instantiations. Note that a stream still maintains its identity as a separate stream when it has been bundled. Arc assertions between streams must still be correct and not just between any two streams in the same bundles. With the semantics given here, the programmer has to be careful that a chain is not formed around all the placed streams, linking them together, so that they are all placed on the same processor!

Also, bundles do not change the semantics of the program at all. If a stream is placed somewhere, bundling it with another will not change the result of the program, it will only change the runtime performance of the program. Therefore bundles can be used arbitrarily to reduce the number of PEs a program is using.

Sharing in Bundles

One important property of bundles is that streams within a bundle wherever possible share computation with each other. When two expressions are bundled together on a node they share the results of any shared subexpression. They also share input streams from other processors. If they both demand values from a particular stream, then only one set of demands for that stream will be made.
This sharing is how Bundles solve the splitting function problem. Figure 4.12 shows how farm can be implemented using Bundle. This builds a process network that matches the definition of farm that was originally shown in Figure 4.1. It works by bundling together the selector nodes that were drawn out of the original implementation, shown in Figure 4.4.

\[
\text{farm} :: \text{Int} \to (a \to b) \to [a] \to (\#\text{partition generated And Bundle partitions})
\]

\[
\text{where}
\]

\[
\text{results} = \text{unpartition generated}
\]

\[
\text{generated} = \text{map (map f) partitions}
\]

\[
\text{partitions} = \text{partition n jobs}
\]

Figure 4.12: Farming with Bundles

Load balancing with Bundles

Using Bundle solves the load balancing problem very neatly. Any node can now be bundled with any other node, irrespective of how related they are, e.g. both the groupings, for treefold, in Figures 4.8 and 4.9 can be implemented easily with Bundle, Figures 4.13 and 4.14 respectively show how.

Grouping by subtree is easier to implement as it follows the natural tree structure. All that is needed is an extra parameter to the treefold function to keep track of the tree level. If called at the root of what will become a bundled subtree then the current computation is simply bundled with the node’s children, otherwise the node is placed as normal. This can be generalised to any depth of bundled subtree by making only the bottom layer of a subtree not bundle with its children. The layers above the bottom layer all bundle with their children, and by transitivity all nodes in a subtree will be bundled together.

The layered grouping is slightly harder as it cuts across the tree shape. Here, a list of the nodes that are at a each level of the overall tree is built up (this is a list of lists of streams). Then at the top level place them by mapping Bundle over
\texttt{treefold} :: \((a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow (a, \text{Placement})\)

\(\text{treefold } f \ s = \text{treefold } f \ s \ 1\)

\texttt{treefold'} :: \((a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow \text{Int} \rightarrow (a, \text{Placement})\)

\(\text{treefold'} \ f \ [s] \ d = (s, \text{Bundle } [s])\)

\(\text{treefold'} \ f \ \text{streams } d\)

\(= (\text{combined}, \text{annotation})\)

where

\(\text{annotation} = \text{if } (\text{odd } d) \text{ then Bundle } [a, b, \text{combined}]\)

\(\quad \text{else Bundle } [\text{combined}] \text{ And fan combined } [a, b] \)

\(\text{And annota And annotb}\)

\(\text{combined} = f \ a \ b\)

\((a, \text{annota}) = \text{treefold'} \ f \ \text{suba} \ (d+1)\)

\((b, \text{annotb}) = \text{treefold'} \ f \ \text{subb} \ (d+1)\)

\((\text{suba, subb}) = \text{split streams}\)

\textbf{Figure 4.13: Subtree blocking using Bundle for \texttt{treefold}}

\texttt{treefold} :: \((a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow (a, \text{Placement})\)

\(\text{treefold } f \ s = (\text{result, annotation})\)

where

\((\text{result, nodes}) = \text{treefold'} \ f \ s\)

\(\text{annotation} = \text{foldr1 (And) (map Bundle nodes)}\)

\(\text{treefold'} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow (a, [[a]])\)

\(\text{treefold'} \ f \ [s] = (s, [[s]])\)

\(\text{treefold'} \ f \ \text{streams } d = (\text{combined, nodes})\)

where

\(\text{nodes} = [\text{combined}] : (\text{map2 } (+) \text{ nodesa nodesb})\)

\(\text{combined} = f \ a \ b\)

\((a, \text{nodesa}) = \text{treefold'} \ f \ \text{suba}\)

\((b, \text{nodesb}) = \text{treefold'} \ f \ \text{subb}\)

\((\text{suba, subb}) = \text{split streams}\)

\texttt{map2} :: \((a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]\)

\(\text{map2 } [] \ [] = []\)

\(\text{map2 } f \ (a:as) \ (b:bs) = (f \ a \ b) : (\text{map2 } f \ as \ bs)\)

\textbf{Figure 4.14: Layer-wise grouping for \texttt{treefold}}
the outer list so that each layer (a list of streams) is bundled together. Finally
joining the bundles together, one large annotation is created.

4.3 Building Better NFOs

We have already seen how annotations can be built using the base language and
partial evaluation at compile time. We have also seen how NFOs can be written
that abstract the frequently used placement patterns and allow for easy reuse, e.g.
fan is used in both farm and treefold earlier in this chapter. The NFOs that have
been defined up until now have had a very simple structure. They have taken
all the placed items as arguments (e.g. fan). The next stage is to provide not
just placement abstraction but computation abstraction as well. The beginnings
of this can be seen in NFOs like farm and treefold. Not only do they define the
placement of the computation, they also define the computation itself. They are
higher order functions that organise their own placement. There is a problem with
them though, they need to not only pass back their computational result, but
also an annotation to define their placement. These are usually returned tupled
together. This destroys a lot of the applicability of the NFOs because handling
code is needed to split the annotation and result. Figure 4.7 demonstrates this
tupling technique.

The problem lies in the fact that Basic Caliban needs the annotation to be at
the top level of the program, annotating the main expression. This leaves it up
the programmer to organise getting all the constituent parts of the annotation
up to the top level of the program. This is the well known problem of data
“plumbing” in functional languages. Other examples of this problem occur when
programming with random numbers (passing a list of numbers round), doing file
IO etc.

All NFOs that hide parts of their computational structure internally will
suffer from this problem. This is because the calling environment will have no
handle on the internal subcomputations that need to be placed in order for the
correct placement to be performed. The work around, up until now, has been to
construct the annotation as part of the NFO and return it so that it can be used at the top level. E.g., another NFO that would be useful is pipe. Pipe takes a list of functions and applies them, in order, to a given input stream. Figure 4.15 gives a definition for the pipe NFO. As you can see, some “plumbing” is needed to get the annotation back to the caller of pipe.

\[
\text{pipe :: \([\text{Stream} \to \text{Stream}] \to \text{Stream} \to (\text{Stream}, \text{Placement})\)}
\]

\[
\text{pipe \([\]\) \(s\) \(= (s, \text{NoPlace})\)}
\]

\[
\text{pipe \((f:fs)\) \(s\) \(= (\text{result, annot And Bundle [local] And Arc local s})\)}
\]

\[
\text{where}
\]

\[
(\text{result, annot}) = \text{pipe fs local}
\]

\[
\text{local} = f s
\]

Figure 4.15: The pipe NFO

One possible solution to this is to use monads \([85]\). These are structures that hide a “state” without having to explicitly pass it around. Although this is quite clean, the program does need to be considerably restructured to allow their use.

### 4.3.1 Multiple Moreovers

Another way of removing the plumbing is not to need it in the first place! Advanced Caliban allows multiple \textit{moreover} clauses. That way annotations can be built at the place where the constituent parts are available.

A \textit{moreover} annotation can be placed anywhere in the code. During partial evaluation, the system hunts out the \textit{moreovers} and evaluates them as it would the top level annotation and automatically synthesises the full annotation from the constituent parts.

As before, an annotation is introduced using the \textit{moreover} keyword. It can annotate any expression, anywhere in the program. Once partial evaluation is complete, any remaining annotations are ignored as they will not play any part in running the program. When defining a multi-clause function the programmer need not place \textit{moreover} annotations for every clause, the null assertion \textit{NoPlace} will be assumed for any that are not provided.
The system still needs to build an annotation at the top level, but now it automatically plumbs the constituent annotations up to the top level.

The mechanism by which clean NFOs can be built is now in place. Figures 4.16, 4.17 and 4.18 show the multiple \textit{moreover} versions of the three NFOs used in this chapter so far.

\begin{verbatim}
pipe :: [Stream → Stream] → Stream → (Stream, Placement)
pipe [] s = s
pipe (f:fs) s = pipe fs ours
    moreover Bundle [ours] And Arc ours s
where
    ours = f s

Figure 4.16: Multiple \textit{moreover} version of pipe

\end{verbatim}

\begin{verbatim}
treefold :: (a → a → a) → [a] → a
treefold f s = treefold f s 1

moreover Bundle [ours] And Arc ours s
where
    ours = f s

Figure 4.17: Subtree blocking \textit{treefold} using multiple \textit{moreovers}

\end{verbatim}

This relatively simple addition to the language has opened up a whole new area of ability to Caliban. There has been some research into the area of algorithmic skeletons for parallel programming [5, 17, 23]. That is, providing prebuilt templates that allow the programmer to build parallel applications quickly. These templates include information about the performance of the algorithm they embody as well as its suitability for a particular parallel architecture. The idea is that the program will tailor itself to the machine that it is being run on.
farm :: Int \to (a \to b) \to [a] \to [b]
farm n f jobs = results
    moreover fan results generated And Bundle partitions
    where
    results = unpartition generated
generated = map (map f) partitions
partitions = partition n jobs

Figure 4.18: Multiple moreover version of farm

Now that functions that not only embody an algorithm but also its placement can be built, libraries of such NFOs can also be built in much the same way as with skeletons.

4.4 More Efficient Processor Usage

It is common for computations to consist of several parts that are executed sequentially, even though within themselves they are parallel. With Basic Caliban these program sections would be implemented as separate subgraphs of the final program graph. Figure 4.19 shows a schematic version of a program with three sections. Only the central section is currently active as it had to wait for the first section to finish before it could start. Similarly, the third section has to wait for the second to finish.

Figure 4.19: A program with three sections

Examples of this sectional behaviour are common, e.g. scene rendering (the scene database may need to be transformed before some raytracing can take place). It can lead to quite large processor usage inefficiencies during program execution, with large areas of the machine not in use, just waiting.

These program sections are called phases. Phases usually compute a whole
operation, and can be considered like skeleton instantiations. The result of one of the NFOs, like farm may be a phase. It all depends on how the NFO is used.

Barrier conditions may be imposed at phase boundaries so that all the data from the producer phase has to be produced before the consumer phase is started. With a non-functional system this may be an important issue as updates may still need to take place. However, in a functional system we do not need to do this as the second phase will only start when the data it needs is available, because the process network imposes its own synchronisation. It can be an advantage to have both phases running at the same time. If they are truly disjoint then the consumer tasks will be stalled with the effect of a barrier. If not then the both sets of tasks will be active which will provide more work for the processors to perform, hopefully providing greater parallel slackness.

### 4.4.1 Phasing in Advanced Caliban

It is vital that sets of processors are not left idle when they could be doing useful work. To avoid this waste of resources, two phases (or subnetworks of processes) need to be phased onto one set of processors.

One way to implement this is to use Bundle to pair nodes from the two phases onto the same processor. This is difficult to program directly as it is hard to get references to each of the nodes of a phase so that pairing can be performed. To do this, each phase would have to return a list of placed streams so that the programmer could match them up. As with the depth load balanced tree, matching disparate items in Caliban is difficult, the annotation needs to be built where there is access to the data it is annotating.

One way to solve this is to get Caliban to help. A phase is a subnet of the final process network. A process network is defined by an annotation. We use And to link two annotations (and therefore subnets) together to form one network. Therefore, by providing a new form of connective it is possible to specify that the networks on either side of it should be scheduled onto the same physical processors. The new connective is called With.
A With B

With is associative and commutative and binds less tightly than And. Also, it doesn’t matter which way round the arguments are, it does not enforce an ordering.

What it does is match up nodes from each of the two argument phases and place them onto one set of processors. In doing so, it will try to minimise communication at the interface of the phases and will also try to reduce the fan-in/out of the network as a whole. Nodes from the same phase will not be placed together. The final number of PEs used by the phased network is the same as the number used by the larger of the two subnets.

Because of the complexity of the problem of mapping generalised structures onto each other, a heuristic algorithm is used to implement phasing.

4.4.2 Problems relating to multiple moreover

Unfortunately the second two extensions to Caliban, multiple moreover and phasing, do not work well together in their current form. Figure 4.20 shows a small program that is to be phased. Each of the two computations should be made into a phase, but because the annotations are built inside the NFOs With cannot be used to connect them because there is no handle on them.

```
main ls = result
  where
  result = pipe [f,g] inter
  inter = farm 4 func ls
```

Figure 4.20: A program with two distinct computations

What is needed is a way of getting a handle on the annotation that is generated inside an NFO result. This is done by introducing a meta-function: Annot expr.

This meta-function returns the annotation that was constructed inside expr. This turns out to be the obvious intermediate step between plumbing and the automatic aggregation of the annotations. It was not needed before because the
only connective available to link two annotations was \texttt{And}. Figure 4.21 shows how the new assertion can be used to phase the two computations.

\begin{verbatim}
main Is = result
   moreover Annot result With Annot inter
   where
   result = pipe [f,g] inter
   inter = farm 4 func ls
\end{verbatim}

Figure 4.21: Annotating distinct computations as phases

If \texttt{Annot} is not used to extract the annotation from a subexpression then it is automatically extracted as before. In other words

\begin{verbatim}
result moreover Annot result
   where
   result = (expr)
\end{verbatim}

is equivalent to

\begin{verbatim}
result
   where
   result = (expr)
\end{verbatim}

\section{4.5 Recapitulation}

This chapter has shown then that Basic Caliban is not flexible enough for parallel programming. It could not construct all the process graphs that could be needed, nor could it define proper splitting functions without communication or computation redundancy. Finally, it could not be used to balance the load of network in the general case. To solve these problems the \texttt{Bundle} assertion was introduced. It has the effect of grouping a subgraph of nodes onto one node, with full subexpression sharing between the grouped computations. This solved all the balancing and splitting problems.
### Figure 4.22: Advanced Caliban assertions and connectives

Another goal was to be able to build libraries of skeleton-like functions that didn’t have to pass back their annotations explicitly. This was achieved by allowing the programmer to use multiple *moreover* annotations. The user could then annotate a computation at the point where all the handles to the streams to be placed were available. Also, an extra assertion, *Annot*, was added to retrieve the annotation of a subcomputation, should the programmer wish to place the subcomputation differently from normal, e.g. phasing it with another computation.

Finally, the chapter showed how a computation that consisted of several phases could leave the parallel computer underemployed. It is hard to bundle disparate computations by hand. This problem was solved by introducing a new annotation connective, *With*, which automatically pairs up nodes from two phases.

Figure 4.22 shows the final set of assertions and connectives available in Advanced Caliban. With this new language, useful parallel tools and programs can now be built.

In the following chapter, an implementation of Advanced Caliban will be presented.

| NoPlace | Null assertion |
| Bundle [x,y] | Place x and y on the same processor |
| Annot x | x’s annotation |
| Arc a b | Document a data dependency between a and b |
| a And b | Join annotations a and b |
| a With b | Phase subnets a and b |
Chapter 5

Implementation of Advanced Caliban

This chapter describes the implementation of Advanced Caliban as defined in chapter 4. The implementation is an extension of that presented in chapter 3.

A major change from the pilot implementation is the target machine. The new target is a Fujitsu AP1000, a medium sized (128 processors) distributed memory parallel computer. Details of the machine and its architecture will be given when the runtime system is described.

The chapter starts by describing the changes that are needed to the front end compiler, specifically changes to the simplifier and network extractor. It then continues with details of the AP1000 and FCG, the functional runtime system, on which the parallel runtime system is built. Finally the parallel runtime system is presented.

5.1 The Advanced Caliban Frontend

The Basic Caliban front end forms the basis for the Advanced Caliban implementation. This section describes the changes to the old front end that are needed. Each language feature motivates different changes and they are presented separately, starting with multiple output processes.
5.1.1 Multiple Output Processes

Multiple output processes are nodes in the final computation graph that compute more than one stream. The Bundle annotation is used to place several stream computations onto the same processor. Bundle takes a list of streams and places all their computations on the same node.

Bundle is easily integrated at the source level, by changing the Placement declaration, as described in Chapter 3, to be:

```haskell
data Placement = NoPlace | Bundle [Stream] | Arc Stream Stream
               | Placement And Placement
```

Here, the Node assertion has been removed and in its place Bundle has been added.

As with the previous implementation, the TS scheme is used to convert the simple output of the early compiler phases into the term-graph representation used by the Advanced Caliban phases.

A minor change is needed to allow the simplifier to cope with the new assertion. The ANF scheme, $\mathcal{A}$, is modified by removing the case for Node and replacing it by a new case for Bundle. Because Bundle requires that each item in its argument list is evaluated to WHNF a new sub-schema $\mathcal{B}$ is needed to do this. Figure 5.1 shows the new version of $\mathcal{A}$ and the $\mathcal{B}$ sub-schema. Note that this scheme does not deal with multiple moreover clauses, they will be introduced later in the chapter (section 5.1.2).

Network Extraction

The purpose of the Bundle assertion is to coalesce computations onto one processor in the final program. Network extraction is the process during which the placement of the computations is defined. Previously the phase generated a function for each node in the process graph. These functions computed a single stream. A call to procret would be generated that explicitly linked, using a wiring list, each of the inputs and outputs of the functions. The obvious way
- ANF scheme extended for Bundle $\mathcal{A} :: \text{Node} \to \text{Memory} \to \text{Memory}$

$\mathcal{A} \mathcal{B} = \text{case } (m, n) \text{ of}$

(Constr NoPlace) $\Rightarrow m$

(Constr Bundle streams) $\Rightarrow m' \quad \text{if } r = S$

$\quad \Rightarrow \text{FAIL} \quad \text{otherwise}$

where

$(m', r) = B \text{ streams } m$

(Constr Arc $[a, b]$) $\Rightarrow m''$

where

$(m', \_ ) = \mathcal{E} a m$

$(m'', \_ ) = \mathcal{E} b m'$

(Constr And $[x, y]$) $\Rightarrow A y m'$

where

$m' = A x m$

(Ind x) $\Rightarrow A x m$

annot $\Rightarrow A \text{ annot } m'$

$\quad \Rightarrow \text{FAIL} \quad \text{if } r = S$

$\quad \quad \quad \text{otherwise}$

where

$(m', r) = \mathcal{E} \text{ annot } m$

$B :: \text{Node} \to \text{Memory} \to (\text{Memory, Success})$

$B \mathcal{B} = \text{case } (m, n) \text{ of}$

(Constr Nil) $\Rightarrow (m, S)$

(Constr Cons $[h, t]$) $\Rightarrow B t m'$

where

$(m', \_ ) = \mathcal{E} h m$

(Ind x) $\Rightarrow B x m$

$x \Rightarrow B x m'$

$\Rightarrow (m', S) \quad \text{if } r = S$

$\quad \quad \quad \text{otherwise}$

where

$(m', r) = \mathcal{E} x m$

Figure 5.1: The Bundle extended ANF evaluator
to implement Bundles is to extend this approach so that the functions generated by network extraction return a list of streams, one stream for each computation placed on a particular node. The existing proconet interface already allows for multiple outputs to be expressed, by using port numbers greater than 0 for output streams. Until now, output ports could only have the value 0.

The obvious way to implement this is to allow proconet to build node functions that return a tuple of streams, rather than just a single stream. This introduces a new problem into the scheme: where does the resulting node function definition get placed? When only one stream was evaluated by a node function, there was an obvious location for the node function's definition, i.e. the same place as the stream that the node computes. Now that several (possibly disparate) streams are to be accommodated, there is no obvious location for the definition. There are two main scopes in a simplified program:

1. The top level program

2. The main expression

All streams need to be in scope of the annotation which is in the main expression scope. If the programmer were to bundle a stream defined at the top level and one defined at the main expression level, where could its node function be defined? It could not be at the top level, as the main expression stream could not be referenced by the definition, nor could it be at the main level, as other top level expressions that used the top level stream could not access it.

It turns out that a much better network extraction scheme is possible. To reach this new scheme a step backwards has to be taken, and both the network extraction and runtime systems have to be considered in parallel as it is the interface between them that is important. The problem that needs to be solved is to partition a computation graph across multiple processors. During the evaluation of a particular stream on a particular node, data from another stream may be needed. If this second stream was Bundled with the evaluating stream then the data is to be computed locally and the evaluator can continue as if nothing has happened. If, on the other hand, the second stream was placed remotely, then
the data must be imported from the remote processor for use by the currently evaluating stream.

A new structure for the run-time system

The *procnet* scheme tries to solve this problem by explicitly piping the data from an interface point for each processor (the formal parameters to the generated node functions) to where the data is used. This approach considerably restructures the program by lifting free variables (references to placed streams) out of their use contexts so that the values can flow down them from the top level of the function to where they are used. As has been seen, this can create many problems of correctness and over strictness.

There is a simpler and better scheme that can be employed to partition the computation graph. It works by directing each initial reference to a stream via a “switchboard”. If the stream is found to be local its suspension is returned as the value and it is therefore computed locally, if it is remote then data is imported from the remote processor. The new scheme does not create a function for each node in the process graph. Rather it relies on the fact that all streams are in the scope of the *moreover* clause (and therefore at the top level) so that references to the stream name are enough.

To implement this new scheme, the functions of *procnet* need to be split in two. Firstly, an initialisation function, *setup*, is needed that sets up buffer space for each input and output stream and performs the initial demand for elements of the each input stream. This initialisation function also prepares the processor with the tasks that are going to be computed locally. The second function, *connect*, takes on the role of building the connection between the use of a stream and where the stream is computed, it acts as a distributed switchboard. Effectively, *connect* modifies the environment of the program based on the PE on which it is being evaluated.
A new network extraction scheme

Network extraction now becomes very simple. Each stream definition is replaced by a call to `connect`; when evaluated the data for that stream is either piped from a remote processor or computed locally. More importantly though, the program structure is not modified and no inefficiencies are introduced. Also the decision about placement can be left until later in the program’s life cycle. With the `procnet` scheme, the program transformation fixed the work distribution by rewriting the program into a set of functions. The new scheme does not restructure the program, it merely adds a level of indirection to the definitions of the streams allowing for the possibility that their placement can be decided upon at any point up to runtime. In fact this possibility is not made use of by the current Advanced Caliban implementation.

```hs
data Message = CHAR Char | INT Int | ...
type Stream = [Message]
type PortID = (Int, Int)
type Connection = (PortID, PortID)
type WiringList = [Connection]
type BundleList = [[Stream]]

setup :: BundleList → WiringList → Stream
connect :: Stream → PortID → WiringList → Stream

setup bl wl = (bl !! a) !! b
  where
    [[[a,b), (0,0)]] = filter toSink wl
toSink (_ (i)) = (i == 0)

connect s pid wl = s
```

Figure 5.2: The Advanced Caliban parallel runtime system interface

Figure 5.2 shows the interface for the parallel runtime system. Semantic definitions of the functions themselves are also given. A `PortID` is a tuple of numbers defining the processor address and a port on that processor. With two `PortID`s a `Connection` can be defined between a producer, specified by the first ID, and a consumer, specified by the second. Remember that there are two
namespaces for ports on a processor; one for input ports and the other for output ports. Therefore a connection \(((a,b),(c,d))\) would connect the stream generated on port b of processor a to the input port d of processor c. Finally, a \texttt{WiringList} is a list of connections that defines the whole process graph. These structures are familiar from the Basic Caliban implementation.

The interface requires that the top level expression of a program is a call to \texttt{setup}. It sets up the buffers and control structures for each input and output stream that the local processor uses or is responsible for. Denotationally, it returns the stream that supplies the host with data. This can be worked out from the wiring list, as demonstrated by the definition given in Figure 5.2. Note that only one stream may supply results to the host, so this definition will fail if the wiring list contains more than one connection to the host.

The important environment manipulation is carried out by \texttt{connect}. Its first argument is the stream it is managing. The second argument is the output port \texttt{PortID} for that stream. If the local processor ID matches the port ID’s processor ID then the first argument can be returned directly allowing it to be computed locally. However, if processor IDs do not match, then the stream must be imported as it is computed elsewhere. The port ID given defines where the stream should be computed, but the wiring list is also needed to determine which port the stream should be imported on. Denotationally, \texttt{connect} just returns its first argument.

Network Extraction now becomes a simple eight-step algorithm:

1. **Rename program**
   Make each definition name unique so that each named expression can be uniquely identified by just its name.

2. **Locate moreover clause**

3. **Massage the program**
   Move any computation on the result of the given process network onto the network itself as an extra processor.
4. Data dependence analysis

Trace through the program graph (avoiding looping), determining which stream definitions use which other streams. After this, each stream definition has a list of imports associated with it. As the import list is built, the Arc annotations can be confirmed. A precondition of this data dependence analysis is that streams are named consistently and uniquely. In other words, to share an expression, the expression must be named and all the shared uses of the expression use the same name. Names uniquely label graph nodes.

5. Build wiring list

Using the import lists generated from the previous step, a wiring list can be constructed that captures the process graph defined. This is the same as the Basic Caliban wiring list, except that output port numbers other than 0 may be used.

6. Build connect calls

For each stream, rename the actual definition of the stream and replace the old definition by a call to connect. E.g. if \( \text{rays} \) was to be computed as output port 5 on processor 2, then the definition \( \text{rays} = \text{expr} \) would be replaced by the following pair of definitions:

\[
\text{rays}' = \text{expr} \\
\text{rays} = \text{connect rays}' (2,5) \text{wiringList}
\]

7. Build setup call

This is similar to building the procnet call in the Basic Caliban implementation. The call to setup replaces the main expression. It takes the full representation of the process network (the BundleList and WiringList) and forks off the threads for each of the output streams to be computed for the processor it is run on. The BundleList is generated from the annotation.
8. **Arc report**

Finally, issue a warning for any Arc assertions that were not uncovered by the data dependence analysis of step 4.

The effect of this version of Network Extraction is to leave the program almost completely unchanged, except for stream definitions, which are indirecd through a call to `connect`, and the main expression, which is turned into a call to `setup`. This means that there is no complexity overhead introduced and that correctness can be guaranteed provided that when `connect` fetches data from a remote processor, it fetches the same values as would be computed locally. This is the case as the remote processor is executing exactly the same code.

### 5.1.2 Multiple moreover clauses

Chapter 4 showed that it would be useful if the programmer could place `moreover` clauses at other places apart from just at the top level of the program. Some NFO declarations would be greatly simplified and generalised if this were possible.

An annotation is attached to an expression by the `moreover` keyword. Although an annotation can place any stream that is in scope, not all annotations that are in scope are valid. Only annotations that are attached to expressions that are involved in the runtime evaluation are used. For example in this program:

```plaintext
main ls = result
     where
     result = (f a) moreover Bundle [result]
     a = g ls
     b = (h a) moreover Bundle [a]
```

the stream `a` would not get placed, if `b` is not used in the computation of `result`. This program is equivalent to:
main ls = result moreover Bundle [result]
    where
    result = f a
    a = g ls

An expression carries its annotation along with it. Any subexpression that is used in the final computation, should have its annotation included as part of the global annotation. This is obviously not possible. It is impossible to know, a priori, which subexpressions (and therefore annotations) will be used. Firstly, only those annotations that are reachable could possibly be used in computation. The simplifier has to decide which of the reachable annotations to include by evaluating the program until it is obvious that it will be used, or it definitely will not be used. Each time there is choice, the discriminating expression must be evaluated to decide upon inclusion.

If any annotation in lexical scope were to be used then the program would be incorrectly placed, e.g. in the presence of conditional statements:

main ls = result
    where
    result = if p then a else b
    a = g ls moreover Bundle [a]
    b = h a moreover Bundle [b]
    p = ...

The conditional in this example decides whether the function h is applied to a or not. What is wanted here is for the expression selected by the conditional to be placed. But, both annotations are in the current lexical scope and it is unclear from just looking at them that they are “hidden” by the conditional. If both annotations were used then either b would be placed and not used (if bool was true) or a would be placed when it is not wanted, in other words the computation of b would take two nodes rather than one node as desired.

The obvious reason for this problem is that where is just a syntactic form for describing a graph. The definitions in a where clause are not related to each other in any way except by normal data dependence. As such, any analysis and
transformation used to amalgamate annotations must take this into account. In Caliban, \texttt{where} clauses name nodes in a graph, and do not create any extra connection between sibling definitions. Therefore evaluation is needed to select the right \texttt{moreover} clause.

Figure 5.3 shows the set of equivalences for \texttt{moreover} clauses used by the compiler. They show how \texttt{moreover} clauses can be moved around without changing their meaning. Equivalence 5.4 shows that a function has to be applied for any annotation that is part of the function body to be released. All aggregate data values (arrays, constructors, tuples etc.) are represented by equivalence 5.5. It shows that all annotations from the argument values are included. Equivalence 5.6 shows that when a named node is used, its definition’s \texttt{moreover} clause can be used. In other words, an annotation from a \texttt{letrec} definition is only used when the definition is used. If one of the consequent expressions of a conditional expression contains an annotation, equivalence 5.8, then the boolean expression has to be used to determine which annotation should be used. Finally, equivalence 5.9 shows that two \texttt{moreover} clauses can be concatenated together using \texttt{And}.

\begin{align}
\text{x moreover NoPlace} & \Rightarrow x \quad (5.1) \\
(f \text{ moreover a}) x & \Rightarrow (f x) \text{ moreover a} \quad (5.2) \\
f(x \text{ moreover a}) & \Rightarrow (f x) \text{ moreover a} \quad (5.3) \\
(\lambda v. (x \text{ moreover a})) y & \Rightarrow x[v/y] \text{ moreover a}[v/y] \quad (5.4) \\
\text{constr x}_1 \ldots (x_i \text{ moreover a}) \ldots x_n & \Rightarrow (\text{constr } x_1 \ldots x_n) \text{ moreover a} \quad (5.5) \\
(v = x \text{ moreover a}) v & \Rightarrow \langle v = x \text{ moreover a} \rangle v \text{ moreover a} \quad (5.6) \\
\text{if b moreover a then x else y} & \Rightarrow \langle \text{if b then x else y} \rangle \text{ moreover a} \quad (5.7) \\
\text{if b then x moreover a1 else y moreover a2} & \Rightarrow \langle \text{if b then x else y moreover a1} \rangle \text{ if b then a1 else a2} \quad (5.8) \\
(x \text{ moreover a1}) \text{ moreover a2} & \Rightarrow x \text{ moreover a1 And a2} \quad (5.9)
\end{align}

Figure 5.3: Equivalences for \texttt{moreover} clauses

An obvious approach to implementing multiple \texttt{moreover} annotations is to do what the programmer would do in the same situation: transform annotated expressions to return a tuple of result and annotation. Although this looks at-
tractive it has several difficulties. Firstly, the tupling transformation is very expensive, a single annotation nested somewhere in a program could lead to almost the whole program being “tupled”. Secondly, once simplification has finished, unused tuples would need to be removed. Thirdly, which expressions become tupled? At one end of the spectrum, all expressions in the program are turned into tuples and manipulated as such. At the other end, only the minimal set that actually need to be tupled are. The second approach is harder because of typing restrictions, not only do all expressions that use the annotated, and therefore tupled, value need to accommodate the tupling, but also all sibling values need to be tupled with a null assertion. E.g.

\[
\text{res} = f [a, b \text{ moreover} c]
\]

needs to be transformed to

\[
\text{res} = f [(a,\text{NoPlace}), (b,c)]
\]

Also from this it can be seen that the function \( f \) has to deal with tupled arguments. It is now possible for \( f \) to be called sometimes with tupled arguments and sometimes without, so there has be more than one version of \( f \). But, it is also important that all closed subexpressions of \( f \) are shared between invocations of different versions of \( f \) during simplification. Without this assurance the full sharing property needed by Caliban is not provided, so some parallelism may be lost and some unwanted recomputation may result.

When a programmer uses tupling to allow annotations in a subexpression to be used, the tuples provide the simplifier with a “route” from the top level \texttt{moreover} clause to the actual placement assertions. This works because the simplifier reduces annotations to ANF, which is normal form reduction except for the WHNF reduction of the stream typed arguments. In effect the annotation is pulled out of the program through the plumbing supplied by the user in the form of the tuples. If all expressions were to be tupled by some compiler transformation, then the simplifier would no longer directly know the “route” to the assertions, it would need some other mechanism to direct its evaluation.
The solution to these problems is to modify the simplifier itself. The approach taken is similar to the “modify all expressions” approach mentioned above, except that the program is not transformed at all. The simplifier automatically associates a Caliban annotation with each expression evaluated and uses meta-annotations to direct the evaluation. In this way the program does not need to be transformed to accept tuples and the original sharing in functions is not disturbed.

Extended data type for terms:
\[ \textbf{data} \quad \text{Term} \quad \alpha = \text{Const Constants} \mid \quad \text{Var Name} \mid \quad \vdots \quad \text{Moreover } \alpha \ \alpha \]

Meta-annotated memory definition:
\[ \text{MMemory} = \text{Node } \rightarrow \text{MTerm Node} \]
\[ \text{MTerm } \alpha = (\text{MetaAnnot}, \text{Term } \alpha) \]
\[ \text{MetaAnnot} = \text{Y } | \text{N } | ? \]
\[ \text{VarType} = \lambda | \text{Letrec} \]
\[ \text{lookup } :: \text{Name } \rightarrow \text{MMemory } \rightarrow (\text{Node, VarType}) \]

Annotation environments:
\[ \text{AnnotRec} = \text{Node} \]
\[ \text{AnnotEnv} = \text{Node } \rightarrow (\text{Success, AnnotRec}) \]

\[ \text{emptyAnnot } :: \text{AnnotEnv} \]
\[ \text{addAnnot } :: \text{AnnotEnv } \rightarrow \text{Node } \rightarrow \text{AnnotRec } \rightarrow \text{AnnotEnv} \]
\[ \text{updateAnnot } :: \text{AnnotEnv } \rightarrow \text{Node } \rightarrow \text{AnnotRec } \rightarrow \text{AnnotEnv} \]
\[ (\forall) :: \text{MetaAnnot } \rightarrow \text{MetaAnnot } \rightarrow \text{Boolean} \]
\[ \text{Y } \lor . = \text{Y} \]
\[ . \lor \text{Y} = \text{Y} \]
\[ ? \lor ? = ? \]
\[ . \lor . = \text{N} \]

\[ \text{VisitEnv} = \text{Nodes } \rightarrow \text{Boolean} \]
\[ \text{emptyVisit } :: \text{VisitEnv} \]
\[ \text{addVisit } :: \text{VisitEnv } \rightarrow \text{Node } \rightarrow \text{VisitEnv} \]

Figure 5.4: Meta-annotation extended definitions for programs

The meta-annotations describe where the \textbf{moreover} clauses are nested. Us-
ing this information the simplifier can target certain expressions to extract their annotations.

To place meta-annotations in a program the abstract program definition from Chapter 3 has to be extended. Also, now that a program can contain more than one `moreover` clause, they need to be explicitly represented. These extensions are shown in Figure 5.4. Firstly, the Term type is extended with a new "Moreover" form. It is an explicit representation of a `moreover` clause. The first argument is the expression the `moreover` clause is attached to, the second argument is the annotation expression itself.

Secondly, a new term type, MTerm, is defined that adds a status flag to a Term. The status, MetaAnnot, has three possible values:

- Y: this expression contains a `moreover` clause
- N: this expression does not contain a `moreover` clause
- ?: meta-annotation not yet known

Finally, a new program type, MMemory, is defined in the same way as before to be a memory consisting of the new extended terms. As all names are unique within a program, a definition can be looked up by finding its name in a memory. If the name refers to a lambda variable, then it has no associated definition and the node returned by lookup is undefined. This was not an issue with the Basic Caliban simplifier, as names were only looked up during evaluation when lambda variables were instantiated before the evaluator reached them.

Figure 5.5 shows a function for setting the meta-annotations for programs. Basically, a term is meta-annotated if one of its subterms is meta-annotated. The simplifier can then use the meta-annotations as a "route map" to locate the `moreover` clauses and evaluate them. The MA must be careful about mutually recursive definitions as these could cause the scheme to loop whilst determining the meta annotation of a term. To avoid looping the scheme keeps a track of where it has been (the visit structure). If a node as already been visited, then the scheme does not descend into it, allowing its current meta annotation to be used by the caller.
- Meta-annotate an expression $\mathcal{MA} :: \text{Node} \rightarrow \text{MMemory} \rightarrow \text{Visit} \rightarrow \text{MMemory}$

$\mathcal{MA} \ n \ m \ v =$

if $v \ n$

= case $(m \ n)$ of

otherwise

$(?, \ \text{Const} \ c) \Rightarrow \text{assign} \ m \ n \ (\text{N}, \ \text{Const} \ c)$

$(?, \ \text{Var} \ v) \Rightarrow \text{assign} \ m \ n \ (\text{N}, \ \text{Var} \ v)$

otherwise

$\Rightarrow \text{assign} \ m' \ n \ (\text{maFlag} \ (m' \ o), \ \text{Var} \ v)$

where

$(o, \ \text{type}) = \text{lookup} \ v \ m$

$m' = \mathcal{MA} \ m \ m' \ v'$

$(?, \ \lambda \ v \ b) \Rightarrow \text{assign} \ m' \ n \ (\text{maFlag} \ (m' \ b), \ \lambda \ v \ b)$

where

$m' = \mathcal{MA} \ b \ m' \ v'$

$(?, \ \text{App} \ f \ a) \Rightarrow \text{assign} \ m'' \ n \ ((\text{maFlag} \ (m'' \ f)) \ \lor (\text{maFlag} \ (m'' \ a)), \ \text{App} \ f \ a)$

where

$m' = \mathcal{MA} \ f \ m' \ v' \ v''$

$m'' = \mathcal{MA} \ a \ m' \ v''$

$(?, \ \text{If} \ a \ b \ c) \Rightarrow \text{assign} \ m''' \ n \ ((\text{maFlag} \ (m''' \ a)) \ \lor (\text{maFlag} \ (m''' \ b)) \ \lor (\text{maFlag} \ (m''' \ c)), \ \text{If} \ a \ b \ c)$

where

$m' = \mathcal{MA} \ a \ m' \ v' \ v'' \ v'''$

$m'' = \mathcal{MA} \ b \ m' \ v' \ v'' \ v'''$

$m''' = \mathcal{MA} \ c \ m'' \ v' \ v'' \ v'''$

$(?, \ \text{Constr} \ \text{tag} \ [x_1, \ ..., \ x_k])$

$\Rightarrow \text{assign} \ m_k \ n \ ((\lor_{1 \leq i \leq k} \text{maFlag} \ (m_k \ x_i)), \ \text{Constr} \ \text{tag} \ [x_1, \ ..., \ x_k])$

where

$m_1 = \mathcal{MA} \ x_1 \ m' \ v' \ v''$;

$m_k = \mathcal{MA} \ x_k \ m_{k-1} \ v' \ v''$

$(?, \ \text{Letrec} \ [v_1 = x_1, \ ..., \ v_k = x_k] \ \text{body})$

$\Rightarrow \text{assign} \ m_{k+1} \ n \ ((\lor_{1 \leq i \leq k} \text{maFlag} \ (m_{k+1} \ x_i)) \ \lor \text{maFlag} \ m_{k+1} \ \text{body}, \ \text{Letrec} \ [v_1 = x_1, \ ..., \ \text{body}]$

where

$m_1 = \mathcal{MA} \ x_1 \ m' \ v' \ v''$;

$m_k = \mathcal{MA} \ x_k \ m_{k-1} \ v'$;

$m_{k+1} = \mathcal{MA} \ \text{body} \ m_k \ v' \ v''$

$(?, \ \text{Moreover} \ e \ a) \Rightarrow \text{assign} \ m' \ n \ (\text{Y}, \ \text{Moreover} \ e \ a)$

where

$m' = \mathcal{MA} \ e \ m' \ v' \ v''$

$(?, \ \text{x}) \Rightarrow \text{match} \ \text{x}$

where

$m' = \mathcal{MA} \ e \ m' \ v' \ v''$

$\text{maFlag} :: \ \text{MTerm} \ \text{Node} \rightarrow \ \text{State}$

$\text{maFlag} \ (s, \ _) = s$

Figure 5.5: A scheme to meta-annotate a program
To aid the searching simplification process, another structure is kept: the annotation environment. It stores the uncovered annotations for each term visited by the simplification process. Its construction is similar to a Memory, in that it is built as a function from term names to annotations. The annotation environment function also returns a Success value to determine if it located an annotation for the expression it was passed.

Note that annotations are built in the same namespace as the program, they are part of the program. This means that the same range of node names is used for both program and annotation.

Figure 5.6 shows the updated versions of schemes $A$ and $B$. There are three changes that are needed to support multiple moreover simplification. Firstly, the functions act on annotated memories rather than plain memories. Secondly, an annotation environment needs to be threaded around. Finally, the placed streams must not be evaluated to WHNF when they are first met. This means there is no call to $E$ in the $B$ schema. Evaluating an expression to WHNF removes any moreover clauses that are attached to expressions that are evaluated because $E$ treats moreover like an identity function. The moreover clauses must be maintained until the $S$ scheme removes them and they are entered into the annotation environment. If this does not happen, parallelism can be lost.

To implement multiple moreover a new simplification schema, $S$, is introduced. Figures 5.7 and 5.8 show the schema. It takes a term from which an annotation needs to be extracted, the current version of the memory and an environment that describes the annotations extracted from all the terms examined so far. It returns a new version of the memory and annotation environment, a success value (to determine if it was blocked in the process of finding the annotation) and the name of the annotation expression that it extracted. Around this scheme is wrapped the $S'$ scheme that adds to and checks the annotation environment. When a term is examined for the first time, an entry consisting of the NoPlace annotation is made for the term, so that if the same term is searched recursively whilst it is initially being searched then a null assertion will result. This stops the scheme looping and also stops it from building cyclic annotations.
\( \mathcal{A} :: \text{Node} \rightarrow \text{MMemory} \rightarrow \text{AnnotEnv} \rightarrow (\text{MMemory}, \text{AnnotEnv}) \)

\[
\mathcal{A} \ n \ m \ \alpha = \text{case} \ (m \ n) \ \text{of} \\
(\text{Constr NoPlace}) \Rightarrow (m, \alpha) \\
(\text{Constr Bundle streams}) \Rightarrow (m', \alpha) \quad \text{if } r = S \\
\quad \Rightarrow \text{FAIL} \quad \text{otherwise} \\
\quad \quad \quad \text{where} \\
\quad \quad \quad (m', r) = \mathcal{B} \text{ streams } m \\
(\text{Constr And } [x, y]) \Rightarrow \mathcal{A} \ y \ m' \ \alpha' \\
\quad \quad \quad \text{where} \\
\quad \quad \quad (m', \alpha') = \mathcal{A} \times m \ \alpha \\
(\text{Constr Arc } [x, y]) = (m, \alpha) \\
(\text{Ind } x) \Rightarrow \mathcal{A} \times m \ \alpha \\
\text{annot} \Rightarrow (m^4, \alpha^\prime) \quad \text{if } r \ & \ r' \ & \ r'' = S \\
\quad \Rightarrow \text{FAIL} \quad \text{otherwise} \\
\quad \quad \quad \text{where} \\
\quad \quad \quad (m', \alpha', r, \text{an}) = S' \text{ annot } m \\
\quad \quad \quad (m'', r') = \mathcal{E} \text{ annot } m' \\
\quad \quad \quad (m'', \alpha'') = \mathcal{A} \text{ annot } m'' \ \alpha \\
\quad \quad \quad (\text{Ind } \text{an'}') = m'' \ n \\
\quad \quad \quad m^4 = \text{assign } m'' \ n \ (N, \text{Constr And } [\text{an}, \text{an'}])
\]

\( \mathcal{B} :: \text{Node} \rightarrow \text{MMemory} \rightarrow (\text{MMemory}, \text{Success}) \)

\[
\mathcal{B} \ n \ m = \text{case} \ (m \ n) \ \text{of} \\
(\text{Constr Nil}) \Rightarrow (m, S) \\
(\text{Constr Cons } [h, t]) \Rightarrow \mathcal{B} \ t \ m \\
(\text{Ind } x) \Rightarrow \mathcal{B} \times m \\
x \Rightarrow \mathcal{B} \times m' \quad \text{if } r = S \\
\Rightarrow (m', F) \quad \text{otherwise} \\
\quad \quad \quad \text{where} \\
\quad \quad \quad (m', r) = \mathcal{E} \times m
\]

Figure 5.6: Updated versions of \( \mathcal{A} \) and \( \mathcal{B} \) for multiple moreover
\[ S' \colon \text{Node} \to \text{MMemory} \to \text{AnnotEnv} \to (\text{MMemory}, \text{AnnotEnv}, \text{Success}, \text{Node}) \]

\[ S' \text{ n } m \alpha = (m, \alpha, S, a) \]

\[ (m', \alpha'', S, an) \quad \text{if } r = S \]

\[ (r, a) = \alpha n \]

\[ \alpha' = \text{add}_\text{Annot} \alpha n \text{ noplace} \]

\[ (m', \alpha'', S, an) = S \text{ n } m \alpha' \]

\[ \alpha'' = \text{update}_\text{Annot} \alpha'' n \text{ an} \]

\[ S \colon \text{Node} \to \text{MMemory} \to \text{AnnotEnv} \to (\text{MMemory}, \text{AnnotEnv}, \text{Success}, \text{Node}) \]

\[ S \text{ b } m \alpha = \text{case } (m \text{ n}) \text{ of} \]

\[ (?, x) \Rightarrow S' \text{ x } m' \alpha \]

\[ \text{where} \]

\[ m' = \mathcal{M}A \text{ x } m \text{ emptyVisit} \]

\[ (\_ , \text{Const } c) \Rightarrow (m, \alpha, S, \text{noplace}) \]

\[ (Y, \text{Var } v) \Rightarrow S' \text{ n } m' \alpha \]

\[ \text{where} \]

\[ (m, \_ ) = \text{lookup } v \text{ p} \]

\[ m' = \text{assign } m \text{ n } (Y, (\text{Ind } m)) \]

\[ (N, \text{Var } v) \Rightarrow (m, \alpha, S, \text{noplace}) \]

\[ (\_ , \lambda \text{ v } b) \Rightarrow (m, \alpha, S, \text{noplace}) \]

\[ (\_ , \text{App } f a) \Rightarrow \text{case } (m \text{ f}) \text{ of} \]

\[ (Y, \lambda \text{ v } b) \Rightarrow \mathcal{F} f \text{ n } a \text{ m } \alpha \]

\[ (Y, \text{func}) \Rightarrow (m', \alpha', \text{F}, \text{an}) \quad \text{if } r = F \]

\[ = (m'', \alpha, r', \text{annot}) \quad \text{otherwise} \]

\[ \text{where} \]

\[ (m', \alpha', r, \text{an}) = S' \text{ f } m \alpha \]

\[ (m'', \alpha', r', \text{an'}) = S' \text{ n } m' \alpha' \]

\[ (m'', \text{annot}) = \text{new } m'' (N, (\text{Constr And } [\text{an}, \text{an'}])) \]

\[ (N, \text{func}) \Rightarrow \text{case } (m \text{ a}) \text{ of} \]

\[ (Y, \text{arg}) \Rightarrow S' \text{ a } m \alpha \]

\[ (N, \text{arg}) \Rightarrow (m, \alpha, S, \text{noplace}) \]

\[ (N, \text{Constr tag args}) \Rightarrow (m, \alpha, S, \text{noplace}) \]

\[ (Y, \text{Constr tag } [x_1, \ldots, x_k]) \]

\[ = (m'_{k}, \alpha_k, \text{rs}, \text{annot}_k) \]

\[ \text{where} \]

\[ (m_1, \alpha_1, r_1, \text{an}_1) = S' x_i m \alpha \]

\[ \ldots \]

\[ (m_k, \alpha_k, r_k, \text{an}_k) = S' x_k m_{k-1} \alpha_{k-1} \]

\[ \text{rs} = \bigwedge_{1 \leq i \leq k} r_i \]

\[ (m'_2, \text{annot}_2) = \text{new } p_k (N, (\text{Constr And } [\text{an}_1, \text{an}_2]) \]

\[ \ldots \]

\[ (m'_{k}, \text{annot}_k) = \text{new } p'_{k-1} (N, (\text{Constr And } [\text{annot}_{k-1}, \text{an}_k]) \]

\[ \ldots \]

Figure 5.7: The searching simplifier scheme (part a)
\[ S \beta m \alpha = \text{case} \ (m \ n) \ of \]

\[
\begin{align*}
&: \\
(Y, \text{If } a \ b \ c) \Rightarrow (m^4, \alpha'', r' \land r'', \text{annot}) & \text{if } (((\text{maFlag } m \ b) \lor (\text{maFlag } m \ c)) \land r = S) \\
\Rightarrow (m', \alpha', r, \text{an}) & \text{otherwise} \\
\text{where} \\
(m', \alpha', r, \text{an}) = S' \ a \ m \\
(m'', r') = \mathcal{E} \ a \ m \\
\text{consequent} = \text{if } (\text{isTrue } m'' \ a) \ \text{then } b \ \text{else } c \\
(m''', \alpha'', r', \text{an'}) = S' \ \text{consequent } m'', \ c' \\
(m^4, \text{annot}) = \text{new } m''' \ (N, \text{Constr And } [\text{an}, \text{an'}])
\end{align*}
\]

(N, If a b c) \Rightarrow (m, \alpha, S, \text{noplace})

(Y, Letrec: \( [v_1 = x_1, \ldots, v_k = x_k] \) body)
\Rightarrow S' \ \text{body} \ m_{k+2} \ \alpha

\text{where}

\[
\begin{align*}
\text{env} &= [(v_1, x_1), \ldots, (v_k, x_k)] \\
m_1 &= I \ x_1 \ m \ \text{env} \\
&: \\
(m_k &= I \ x_k \ m_{k-1} \ \text{env}) \\
m_{k+1} &= I \ \text{body} \ m_k \ \text{env} \\
m_{k+2} &= \text{assign } m_{k+1} \ n (Y, (\text{Ind body}))
\end{align*}
\]

(N, Letrec: defs body) \Rightarrow (m, \alpha, S, \text{noplace})
\[
(\_ \ \text{Ind } x) \Rightarrow S' \times m \ \alpha
\]
\[
(\_ \ \text{Moreover e a}) \Rightarrow (m^4, \alpha'', S, n) \quad \text{if } r = S \\
\Rightarrow (m''', \alpha'', F, \text{noplace}) & \text{otherwise} \\
\text{where} \\
(m', \alpha') = A \ a \ m \ \alpha \\
m'' = \text{assign } m' \ n (Y, \text{Ind e}) \\
(m''', \alpha'', r, \text{an}) = S' \ e \ m'' \ \alpha' \\
(m^4, \text{an}) = \text{new } m''' \ (N, \text{Constr And } [a, \text{an}])
\]

(\_ \ _) \Rightarrow (m, \alpha, S, \text{noplace})

Figure 5.8: The searching simplifier scheme (part b)
- The top level driving scheme
\[ \mathcal{T} :: \text{Node} \rightarrow \text{MMemory} \rightarrow (\text{Node, MMemory}) \]
\[ \mathcal{T} n m = (\text{an, } m') \]
where
\[ m' = \mathcal{M.A} \text{ n } m \text{ emptyVisit} \]
\[ (m'', a, r, \text{an}) = \mathcal{S'} \text{ n } m \text{ emptyAnnot} \]
\[ m''' = \mathcal{R.A} \text{ an } m'' \]

- Remove aliasing from annotated streams
\[ \mathcal{R.A} :: \text{Node} \rightarrow \text{MMemory} \rightarrow \text{MMemory} \]
\[ \mathcal{R.A} n m = \text{case } (m n) \text{ of} \]
\[ (\_ \text{ Constr NoPlace}) \Rightarrow m \]
\[ (\_ \text{ Constr Bundle } [x_1, ..., x_k]) \Rightarrow m_k \]
where
\[ (m_1, \_ ) = \mathcal{E} x_1 m \]
\[ \vdots \]
\[ (m_k, \_ ) = \mathcal{E} x_k m_{k-1} \]
\[ (\_ \text{ Constr Arc } [a, b]) \Rightarrow \mathcal{R.A} b m' \]
where
\[ m' = \mathcal{R.A} a m \]
\[ (\_ \text{ Constr And } [a, b]) \Rightarrow \mathcal{R.A} b m' \]
where
\[ m' = \mathcal{R.A} a m \]

Figure 5.9: Schemes to drive the searching simplification
Once $S'$’s call to $S$ returns with an annotation, $S'$ places the returned annotation into the annotation environment for use by further searches.

The schemes assume an initial memory node called “noplace” that points to the “(N, Constr NoPlace)” term, this saves building a new NoPlace node each time the rules bottom out.

The $S$ schema is a modified version of the $E$ (WHNF) schema. It is called $S$ because it searches out the annotations held within the program, with the help of the meta-annotations defined in Figure 5.5. If an expression does not contain an annotation then it is not evaluated at simplification time unless it is needed to help extract an annotation from elsewhere. Functions that contain annotations can only yield them when applied, so the rule for lambda expressions returns nothing regardless of whether the body is annotated or not. Function applications fall into three categories. Firstly those that contain no annotations; they just return. Secondly, those that have the annotation in an argument position: here the function application does not need to be evaluated, only the argument needs to be searched. Finally, if the function does contain an annotation it has to be applied to release the annotation. The four rules for application perform these steps by running down the application spine until either a meta-annotated function is found or there are no more meta-annotations. When a meta-annotated function is found it is applied in much the same way as in normal WHNF evaluation, using the $F'$ scheme, but instead of evaluating the result of the application with $E$ as $F$ does, $F'$ uses $S'$. Also, when copying nodes, $C$ should make any new nodes have a meta-annotation of ? and then call $MA$ on the copied graph to update mark expressions which contain annotations after the instantiation. Copying must also maintain the uniqueness of names by renaming any names that are defined in the copied section of graph.

Next come constructors. The schema shows that when some of the arguments to an aggregate data item are meta-annotated then they should be searched and the results of the searches Anded together. Conditionals are very important for Caliban programs as they select between different placement choices. If a conditional is meta-annotated then there is an annotation nested in at least one
of the subexpressions. If one of the consequent expressions is meta-annotated then the boolean expression needs to be evaluated so that a decision about which consequent expression to be used can be made. In this case the conditional is making a choice between two different placement decisions. Unfortunately, evaluating an expression using $E$ removes any nested $\text{moreover}$ clauses, thus losing some of the annotation. The solution is to search the boolean expression to extract any nested annotations before evaluating it. Once the boolean is fully evaluated, the correct consequent can also be searched. If it is just the boolean expression that is meta-annotated then it is all that needs to be searched and the conditional can be left alone.

In Caliban, $\text{letrec}$s are used merely to name nodes in the program graph. The body is instantiated with the local definitions and then searched. Note that the $T$ scheme has to clear the meta annotations for all the nodes that it visits during the instantiation so that if meta annotated expressions are instantiated into an expression, the simplifier will meta annotate the expression again (using $M.A$) and thus notice the instantiated annotation.

Finally when a $\text{moreover}$ clause is found the annotation is evaluated to ANF using $A$ and the expression it is annotating is searched. The resulting annotations are joined together and returned.

A final pair of schemas is used to drive the whole process. $T$ takes in a fresh MMemory and returns the fully simplified program together with the name of the annotation. Up until now, the schemes have evaluated placed streams to WHNF when they have been found in an annotation. This will no longer work, as the $E$ evaluator will remove any $\text{moreover}$ clauses it finds as $\text{moreover}$ is equivalent to the identity function for the WHNF evaluator. By delaying the WHNF evaluation until all the annotations have been uncovered this problem is removed. The $S$ schema does not require the streams to the evaluated to WHNF; it is the following phase, network extraction, that needs it. Network extraction needs the streams to be in WHNF because that removes any aliasing problems. Figure 5.6 shows a version of $B$ that does not evaluate streams to WHNF. This is complemented by the $R.A$ scheme (remove aliasing) which “post processes” the
annotation by evaluating each placed stream to WHNF.

\[
in x y = a \text{ moreover Bundle } [a] \\
\quad \text{where} \\
\quad a = f x y \\
out x = b \text{ moreover Bundle } [b] \\
\quad \text{where} \\
\quad b = g x \\
main ls = res \\
\quad \text{where} \\
\quad res = \text{ out loop} \\
\quad \text{loop} = \text{ in ls res}
\]

Figure 5.10: A sample Caliban program using two NFOs

To demonstrate how searching simplification works a small example is given. Figure 5.10 shows a program that uses two NFOs to build a process network. The NFOs take some argument streams and place the computations that result.

To show the sample simplification the following notation will be used:

- Semantic brackets are used to enclose pieces of syntax rather than using the abstract notation used to describe the rules.

- Application nodes are explicitly shown as infix @ symbols.

- If nodes need naming, then a subscripted number is used. This is the same as the node name used to distinguish terms in the abstract description.

- Important meta-annotations are shown as underlinings.

- Only expression arguments to the schemas are shown; \( m \) and \( a \) are assumed.

- The annotation environment starts off empty, items are added to it along the way. Only interesting additions are shown.
The program is converted into an intermediate form and meta-annotated as follows:

\[ \text{in} = \lambda x. \lambda y. \text{letrec } a = f \times y \]
\[ \text{in a moreover Bundle } [a] \]
\[ \text{out} = \lambda x. \text{letrec } b = g \times x \]
\[ \text{in b moreover Bundle } [b] \]
\[ \text{main} = \lambda s. \text{letrec } \text{res} = \text{out} @_3 \text{loop}_4 \]
\[ \text{loop } = \text{in}_5 @_0 \text{ls}_7 @_8 \text{res}_0 \]
\[ \text{in } \text{res}_1 \]

What follows is a trace of the simplifier similar to the one in Chapter 3. To make the trace clearer the precise lazy reduction order has not been used for the \( E \) schema.

\[
\begin{array}{l}
S' [\text{res}_1] \\
A \quad S' [\text{out}_2 @_3 \text{loop}_4] \\
\quad S' [\lambda x_0. \text{letrec } ...] \\
B \quad \leftarrow \text{NoPlace} \\
\quad F' [\lambda x_{10} ...] 3 [\text{loop}_4] \\
C \quad S' [\text{letrec } r = g_{11} @_{12} \text{loop}_4 \text{in } r_{13} \text{ moreover}_{14} \text{ Bundle}_{15} [r]] \\
D \quad S' [r_{13} \text{ moreover}_{14} \text{ Bundle}_{15} [r]] \\
\quad A [\text{Bundle}_{15} [r]] \\
\quad \vdots \\
\quad \leftarrow \text{Bundle}_{15} [r] \\
S' [r_{13}] \\
S' [g @_{12} \text{loop}_4] \\
\quad \alpha 12 = \text{NoPlace} \\
S' [\text{loop}_4] \\
S' [\text{in}_5 @_0 \text{ls}_7 @_8 \text{res}_0] \\
\quad \alpha 8 = \text{NoPlace} \\
\quad \vdots \\
\quad S' [\text{in}_5] \\
\quad S' [\lambda x. \lambda y. \text{letrec } ...] \\
\quad \leftarrow \text{NoPlace} \\
\quad F' [\lambda x. \lambda y. \text{letrec } ...] 8 [\text{res}] \\
\quad S' [\text{letrec } a = (f_{19} @_{20} \text{ls}_7 @_{21} \text{res}_0) \text{ in } a \text{ moreover}_{30} \text{ Bundle}_{22} [a]] \\
E \quad S' [a \text{ moreover}_{30} \text{ Bundle}_{22} [a]] \\
\quad \alpha 30 = \text{NoPlace} \\
\quad A [\text{Bundle}_{22} [a]] \\
\quad \vdots \\
\quad \leftarrow \text{Bundle}_{22} [a] \\
S' [a]
\end{array}
\]
The trace shows several of the mechanisms of the new schemes in action. On
the line marked A, the annotation environment record for node 3 is made to be
\texttt{NoPlace}. Later on in the trace, this will be used to stop a circular annotation
being formed. On line B, the \texttt{out} function search returns \texttt{NoPlace} because the
annotation is inside the function. The function application is then performed
using \texttt{F'}. Remember that the function body will be copied and meta-annotated:
line C shows that the definition of \texttt{r} is now meta-annotated because it uses \texttt{loop}.
Line D shows the first \texttt{moreover} clause being uncovered. The annotation is
evaluated and the annotated expression is searched. This subsearch reveals the
annotation for \texttt{loop} on line E. The search of the expression for this \texttt{moreover}
clause in turn reveals the annotation for \texttt{res} (which is what the top level call
to \texttt{S'} is looking for). This is because the expression definitions are mutually
recursive. Line F shows how the annotation environment is used to avoid looping
in these situations. The node for \texttt{res} is already in the annotation environment
with annotation \texttt{NoPlace} as a value. This annotation is returned as the result of
the second call to \texttt{S'} on \texttt{res} so avoiding any circular annotations. Finally on line
G, note how the annotation environment is updated with the annotation for \texttt{res}
when \texttt{S'} returns.

### 5.1.3 Phasing

The previous chapter showed the need for some mechanism to fold networks
on top of each other. This is useful, because if computations have differing
placements it is difficult to write annotation code to match those placements.
Sometimes the items that need bundling are difficult to get into a form which allows them to be bundled. For example, one way to match up a farm and a tree computation would be to build a data structure that represented each of the two structures (and allowed access to the streams that form them) and then write a function to match them reasonably well. With the With connective the programmer only needs to join the annotations (representing the subnetworks) together using the new connective.

\[
\begin{align*}
    a \text{ With } b & \equiv b \text{ With } a \quad (5.10) \\
    a \text{ And } b \text{ With } c & \equiv a \text{ And } (b \text{ With } c) \quad (5.11) \\
    (a \text{ And } b) \text{ With } (a \text{ And } c) & \equiv a \text{ And } (b \text{ With } c) \quad (5.12)
\end{align*}
\]

Figure 5.11: Equivalences for the With connective

Figure 5.11 shows a set of equivalences for the new annotation connective. With has higher priority than And, equivalence 5.11. Most importantly though, equivalence 5.12, if a node appears on the both sides of the With, then it is not included in the phasing. In other words, a network cannot be folded onto itself.

**Aspects of an implementation of With**

The idea behind phasing is to reuse the resources of one subnetwork to implement another, therefore reducing the resource requirement of the whole system. Specifically, when two subnetworks are phased they share processing elements rather than being implemented on disjoint sets of processors.

A simple implementation of phasing pairs nodes from the smaller subnetwork with nodes from the larger network. This can be done using Bundle. A pair of nodes, one from each subnetwork, is picked and bundled together to produce one node in the final computation.

Implementing phasing therefore reduces to the problem of discovering a good mapping from one network to the other. An optimal mapping algorithm would be able to produce a mapping that would minimise the execution time of the computation as a whole. This is impossible, even with extensive analysis, as
it is equivalent to the general scheduling problem. Instead a reasonable set of heuristics has to be used to produce the mapping.

Before examining some heuristics, certain properties of phases will be presented. If two subnetworks really are phased then their execution will naturally fall in disjoint sections of a timeline. This means that by placing the networks on the same set of processors there is no time penalty as they should not interact. Also, it is possible to gain by removing some of the communication that would have taken place between the phases by placing producer/consumer pairs from the phases onto the same processor. These producer/consumer pairs, where one is in one phase and the other in the other phase, form the interface between two phases. An example of disjoint phases is shown in Figure 5.12. Phase A computes a value (or a series of values), but phase B cannot start work until all of A’s output is ready. In this case the two networks can be placed on top of each other without bad interactions.

![Figure 5.12: Two phases](image)

Communication can be in either direction between two phases. Sometimes it is in both directions, for example A and B could be coroutining computations passing their results back and forth. An example of the latter program structure will appear in the next chapter, when an iterative Jacobi solver is implemented.

In its loosest sense, phasing is all about reducing the number of processors a program needs to execute, or increasing the computational work of processors so that they are not lying idle for considerable periods of time. If two disjoint computations are phased, even if they run at the same time, the computational pattern is likely to be dissimilar and so increase the likelihood that a processor has some work to do at any one time. This effect is known as parallel slackness. The more work a processor has to perform, the more likely it is that it can find
work to do when the currently executing task blocks or finishes. Therefore, the more work a processor is asked to do, the better the processor utilisation and efficiency.

There are many possible mappings between two process networks. To help select one of them, here are two heuristics that identify aspects of the performance of the resulting process network.

1. Minimise communication

2. Minimise fan-in/out

One way to minimise communication is to place producer/consumer pairs on the same processor, this means aligning the interfaces of the phases. This is the only place where this gain can be made as internal nodes from each phase do not communicate; remember only one node from each phase is placed on a particular processor.

Minimising the fan-in/out is a way of preserving locality of communication. When a fan-in/out becomes too large for a network architecture to support, the parallel computer has to route messages to their destinations via other nodes in the system. This has the effect of increasing the communication distance. Even on machines that are relatively insusceptible to message distance, due to wormhole routing for example, reducing message distance also helps in reducing network contention. To reduce fan-in/out, producer/consumer pairs from one phase are mapped onto a similarly communicating pair in the other phase to reduce the need for new communication links.

There is an assumption that the computational load of each node in a phase is similar. In other words there is no point in placing two nodes from a single phase onto the same processor. If the computational load of each processor is \( x \) in the largest network and \( y \) for each node of the smaller network, then the total computational load for each processor in the resulting fused network is at most \( x + y \). If both networks are required to compute at different times then they will not interfere and therefore give an efficiency increase because they are using fewer processors in the same time to complete the same computation.
A subnet network mapping algorithm

There are many possible schemes that could be used to apply the heuristics mentioned above to the problem of pairing nodes from different networks together. The obvious solution is to use a generate and test algorithm — all solutions are generated and then tested against the heuristics, the solution with the highest score being chosen. This method has a particularly bad complexity order, having to generate and test $m P_n$ (permutations of $n$ objects into $m$ places, $\frac{m!}{(m-n)!}$) if $n$ is the size of the smaller network, and $m$ of the larger. Even for a modest number of processors, this leads to a huge search space. E.g. with just 10 processors in each phase there are a possible 3.6 million solutions.

What is needed is a way of limiting the number of solutions that need to be tested. One simple method is only to generate solutions that reduce interface producer/consumer communication. In other words, only generate solutions in which all the producer and consumers pairs across the phase boundary are coalesced. If there is a single boundary producer/consumer pair for the phases then the number of solutions generated for our 10 process networks would be $9 P_9$, which is 360000. The more interface links, the better the complexity becomes. With 5 boundary producer/consumer pairs the number of solutions generated is $5 P_5 \times 5 P_5$ which is 14400.

In general terms, the worst case for the number of solutions that need to be tested is given in Figure 5.13.

The full algorithm for phase matching A With B is therefore as follows:

1. **Commonality:** Factor out any common nodes to both side of the With.

2. **Direction:** Decide on a direction of mapping by placing the smaller network on top of the larger.

3. **Interface mappings:** Generate all possible interface mappings that cut the number of communication arcs.

4. **Solutions:** Generate all solutions for each partial mapping from the previous step.
5. Calculate: For each solution, calculate the number of *introductions*, i.e. the number of new arcs between nodes of the old base network that are needed to satisfy the joined network.

6. Choose: Select a solution that best fits needs by minimising the number of introductions.

Next, a small example will be used to demonstrate the algorithm at work. The process network:

is being phased, the broken vertical line shows the divide between the phases. Each node is labeled with a letter. As phase A contains four nodes, whilst B only contains three the network for B will be folded on top of A. The second step of the algorithm is to recognise the interface mappings and fix them. As there is only one interface link it is simply fixed. Next, a tree of solutions for each of the other nodes is built:
It simply assigns all the possible mappings from the remainder of the nodes in $B$ to all the rest of the nodes in $A$. This leaves six solutions to the mapping, each solution is labelled at the leaf. For example, solution 3 maps node $e$ to $c$, $f$ to $b$ and $g$ to $a$.

Finally, the cut and introduction values are computed for each putative solution:

<table>
<thead>
<tr>
<th>solution</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>introductions</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

The introduction markings noted for solution two, for example, come from $f$ being mapped onto $a$ thus inducing two new communication arcs, one between $c$ and $a$ and the other between $d$ and $a$.

Solutions 4 and 6 are the right ones to choose, because they do not introduce any new communications. It can be seen that the chosen solutions are similar in structure, they both wrap the $efg$ ring around the similar $bcd$ ring in the other phase.

A greedy optimisation

The time complexity of the above algorithm is very bad because it selects the best solution out a set of possible solutions. This set of possible solutions can be very large, and therefore take a long time to generate and a large amount of space to store. A greedy algorithm attempts to build a single solution by choosing the best option at each step. In this way, the time taken to generate a solution is considerably reduced, although the solution may not be as good as the one found by a search of the solution space.
The greedy version of the algorithm starts by mapping all interface nodes together as this is a requirement of phasing. If there is a choice of interface nodes to map to, it maps to the one with the highest number of connections.

Then, starting from one of the mapped interface nodes, the network to be mapped is traced, mapping each as yet unmapped node onto the destination network. The following rules are applied for each individual mapping, with the first matching rule being used:

- Map the node to a neighbour of where its neighbour has been mapped
- Map onto the free node with the largest number of neighbours

The first rule tries to ensure that locality is maintained in the mapped network. This reduces the fan-in/out of the resulting network and therefore helps to maintain the locality. The second rule is used when there is no obvious choice of node to map to. As this is a greedy algorithm, the best possible node is chosen to map to.

As with all greedy algorithms, they do not necessarily produce the best results due to their local (single mapping at a time) view of the problem.

The greedy algorithm would produce the same results for the example given earlier in this section. Although it could also produce a solution that mapped the efg ring onto cba which would introduce a new communication between a and c. The solution found depends on the order in which the graph is traversed.

5.1.4 Integrating multiple moreover clauses and phasing

In Chapter 4, section 4.4.2, it was shown that multiple moreover transformations and phasing do not work well together. This is because multiple moreover use And to concatenate annotations by default, whereas to phase two computations the With connective is needed. Therefore it is impossible to phase annotations that are generated by subexpressions’ moreover clauses. To solve this problem a new annotation called Annot has been introduced. It returns the annotation built for the execution of its argument expression. When an expression’s annotation is
explicitly used by Annot it is not included in the multiple moreover composite annotation.

Adding Annot to the system is a simple matter of extending the Placement type as follows:

\[
\text{data Placement} = \text{NoPlace} \mid \text{Bundle [Stream]} \mid \text{Arc Stream Stream}
\]

\[
\mid \text{Placement And Placement} \mid \text{Annot a}
\]

The effect of using Annot will be to return the aggregated annotation built up by the $S$ schemes for the argument expression. It also needs to ensure that the annotation is not also automatically aggregated into the general annotation by the normal multiple moreover scheme. This can be achieved by extending the record for each expression in the annotation environment. Instead of just holding the annotation, it holds two versions of the annotation ($i,a$). The first version, $i$, is an indirection node that points to the annotation. The second, $a$, is the annotation itself. The idea is to use $i$ whenever the annotation is returned as part of the normal automatic aggregation process. When Annot is used to extract an annotation, then the $a$ version is returned. The reason for this is that when Annot is used, it can update the $i$ value into a NoPlace so that instances of the annotation that were used due to normal aggregation are no longer accessible.

To implement this scheme, two extensions to the $S'$ and $A$ schemes are needed. These are shown in Figure 5.14.

To show how the Annot-extended simplifier works, the example program from Figure 5.10 is extended to include calls to Annot and then re-simplified. The extended program is in Figure 5.15. Here is the intermediate form of this program, together with the initial meta-annotations:
– Extended annotation record
AnnotRec = (Node, Node)

$S' :: \text{Node} \rightarrow \text{MMemory} \rightarrow \text{AnnotEnv} \rightarrow (\text{MMemory}, \text{AnnotEnv}, \text{Success}, \text{Node})$

$S' \ n \ m \ \alpha = (m, \alpha, S, i) \quad \text{if } r = S$
$(m'', \alpha'', S, \text{an}) \quad \text{otherwise}$

where
$(r, (i,a)) = \alpha \ n$
$\alpha' = \text{add}_\text{annot} \alpha \ n \ (\text{noplace, noplace})$
$(m', \alpha'', s, \text{an}) = S \ n \ m \ \alpha'$
$(m'', \text{ind}) = \text{new} m' \ (N, \text{Ind an})$
$\alpha''' = \text{update}_\text{annot} \alpha'' \ n \ (\text{ind, an})$

$A :: \text{Node} \rightarrow \text{MMemory} \rightarrow \text{AnnotEnv} \rightarrow (\text{MMemory}, \text{AnnotEnv})$

$A \ n \ m \ \alpha = \text{case} \ (m \ n) \ \text{of}$
$(\text{Constr NoPlace}) \Rightarrow (m, \alpha) \quad \text{if } r = S$
$(\text{Constr Bundle streams}) \Rightarrow (m', \alpha) \quad \text{otherwise}$

where
$(m', \alpha) = B \ \text{streams} \ m$

$(\text{Constr And} \ [x, y]) \Rightarrow A \ y \ m' \ \alpha'$

where
$(m', \alpha') = A \ x \ m \ \alpha$

$(\text{Constr Arc} \ [x, y]) = (m, \alpha)$
$(\text{Constr Annot} \ x) \Rightarrow (m'''', \alpha')$

where
$(m'', \alpha', r, \text{an}) = S' \ x \ m \ \alpha$
$(i, a) = \alpha' \ x$
$m''' = \text{assign} \ m' \ i \ \text{noplace}$
$m''' = \text{assign} \ m'' \ x \ (N, \text{Ind a})$

$(\text{Ind x}) \Rightarrow A \ x \ m \ \alpha$
$\text{annot} \Rightarrow A \ \text{annot} \ m' \ \alpha \quad \text{if } r = S$
$\Rightarrow \text{FAIL} \quad \text{otherwise}$

where
$(m', r) = E \ \text{annot} \ m$

Figure 5.14: Changes to $S'$ and $A$ required for Annot
in x y = a moreover Bundle [a] where a = f x y
out x = b moreover Bundle [b] where b = g x
main ls = res moreover Annot res With Annot loop where res = out loop loop = in | res

Figure 5.15: A sample Caliban program using two phased NFOs

in = λx. λy. letrec a = f x y in a moreover Bundle [a]
out = λx. letrec b = g x in b moreover Bundle [b]
main = λls. letrec res = out₂ @₃ loop₁ loop = in₅ @₆ ls₇ @₈ res₉ in res moreover₃₀ Annot res With₃₁ Annot loop

Note that the annotation environment now takes a tuple of values for each entry. The first is known as the indirection value and the second the annotation. Two further pieces of notation are used in this example:

- \( n \) represents the node named \( n \). It is like a wildcard pattern.

- \( \nabla n \) is an indirection node named \( n \). These only appear in the indirection field of the annotation environment entries. They always indirect to the second field of the entry, i.e. the annotation entry itself.

With this new notation, the annotation entry \((\nabla₅, \alpha₆)\) is an indirection node, 5, redirecting to node 6.

The simplifier trace now looks like this:

\[
\begin{align*}
A & \rightsquigarrow [\text{res₂ moreover₃₀ Annot res With₃₁ Annot loop}] \\
\alpha & 50 = (\text{NoPlace, NoPlace}) \\
\end{align*}
\]
\[ A \left[ \text{Annot res} \right]\]
\[ S' \left[ \text{res}_1 \right]\]
\[ S' \left[ \text{out}_2 \; \text{ls}_3 \text{loop}_4 \right] \quad \alpha 3 = (\text{NoPlace}, \text{NoPlace}) \]
\[ S' \left[ \text{out}_2 \right] \]
\[ S' \left[ \lambda x_{10}. \; \text{letrec} \ldots \right] \]
\[ \leftarrow \text{NoPlace} \]
\[ F' \left[ \lambda x_{10} \ldots \right] 3 \left[ \text{loop}_4 \right] \]
\[ S' \left[ \text{letrec} \; r = g_{11} \; \text{ls}_{12} \; \text{loop}_4 \; \text{in} \; r_{13} \; \text{moreover}_{14} \; \text{Bundle}_{15} \; [r] \right] \]
\[ S' \left[ r_{13} \; \text{moreover}_{14} \; \text{Bundle}_{15} \; [r] \right] \]
\[ \alpha 14 = (\text{NoPlace}, \text{NoPlace}) \]
\[ A \left[ \text{Bundle}_{15} \; [r] \right] \]
\[ \vdots \]
\[ \leftarrow \text{Bundle}_{15} \; [r] \]
\[ S' \left[ r_{13} \right] \]
\[ S' \left[ g \; \text{ls}_{12} \; \text{loop}_4 \right] \quad \alpha 12 = (\text{NoPlace}, \text{NoPlace}) \]
\[ S' \left[ \text{ls}_5 \right] \]
\[ S' \left[ \text{ls}_7 \right] \]
\[ S' \left[ \text{ls}_8 \right] \]
\[ S' \left[ \text{ls}_9 \right] \]
\[ S' \left[ \text{out}_2 \; \text{ls}_3 \; \text{loop}_4 \right] \]
\[ \alpha 8 = (\text{NoPlace}, \text{NoPlace}) \]

\[ B \]
\[ \text{3 found in } \alpha, \text{ returning } 3's \text{ indirection entry} \]
\[ \leftarrow \text{NoPlace} \]
\[ \leftarrow \alpha_{60} \quad \alpha 30 = (\forall_{60}, \text{Bundle}_{22} \; [a]) \]
\[ \leftarrow \alpha_{66} \quad \alpha 8 = (\forall_{66}, \alpha_{60}) \]
\[ \leftarrow \alpha_{61} \quad \alpha 12 = (\forall_{61}, \alpha_{66}) \]
\[ \leftarrow \alpha_{62} \quad \alpha 14 = (\forall_{62}, \alpha_{61} \text{ And } \text{Bundle}_{15} \; [r]) \]

\[ C \]
\[ \leftarrow \alpha_{62} \quad \alpha 3 = (\text{NoPlace}_{61}, \alpha_{62}) \]
\[ A \left[ \text{Annot loop} \right]\]
\[ S' \left[ \text{ls}_5 \; \text{ls}_7 \; \text{ls}_8 \; \text{res}_9 \right] \]
\[ 8 \text{ found in } \alpha, \text{ returning } 8's \text{ indirection entry} \]
\[ \leftarrow \alpha_{66} \]
\[ \leftarrow \alpha_{60} \quad \alpha 8 = (\text{NoPlace}_{66}, \alpha_{60}) \]
\[
D \leftarrow \varnothing_{51} \text{ given that } (\varnothing_{60} \text{ With}_{51} \varnothing_{62}) \\
S' \leftarrow \varnothing_{[\text{res}]}
\]

\[
E \leftarrow 3 \text{ found in } \alpha, \text{ returning } 3 \text{'s entry} \\
\leftarrow \text{NoPlace}_{64} \\
\leftarrow \varnothing_{69} \quad \alpha_{50} = (\varnothing_{69}, \text{NoPlace}_{64} \text{ And } \varnothing_{51})
\]

There are several new mechanisms in the new simplification rules. Firstly, line \(A\) shows the annotation environment being updated with a blank entry for the node named 50. Line \(B\) shows the annotation environment being used to cut the search. Note how the indirection part of the entry is returned as the result. Line \(C\) shows how the rule for \textit{Annot} in \(A\) cuts the indirection entry in an annotation environment entry by making it into a \texttt{NoPlace} node. From now on, any references to node 64 will be \texttt{NoPlace} and not annotation for \texttt{res}. Also, the actual annotation is returned for use by the caller. This is demonstrated on line \(E\); the lookup returns \texttt{NoPlace} rather than the annotation for \texttt{res}. Finally, line \(D\) shows how the memory is updated so that the \textit{Annot} calls are replaced by the actual annotations.

The final annotation environment produced by the program is:

<table>
<thead>
<tr>
<th>Node</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>( (\varnothing_{69}), \texttt{NoPlace}<em>{64} \text{ And } \varnothing</em>{51})</td>
</tr>
<tr>
<td>8</td>
<td>( \texttt{NoPlace}<em>{66}, \varnothing</em>{60} )</td>
</tr>
<tr>
<td>3</td>
<td>( \texttt{NoPlace}<em>{64}, \varnothing</em>{62} )</td>
</tr>
<tr>
<td>14</td>
<td>( (\varnothing_{62}), \varnothing_{61} \text{ And Bundle}_{15} [r] )</td>
</tr>
<tr>
<td>12</td>
<td>( (\varnothing_{61}), \varnothing_{66} )</td>
</tr>
<tr>
<td>30</td>
<td>( (\varnothing_{60}), \texttt{Bundle}_{22} [a] )</td>
</tr>
</tbody>
</table>

If the annotation is traced from node 69, which is what is returned by the top call to \(S'\) and the result is simplified to remove \texttt{NoPlaces} then

\[
\text{Bundle}_{22} [a] \text{ With}_{51} \text{Bundle}_{15} [r]
\]

is the result, which is what the programmer wanted.
5.2 The Advanced Caliban Runtime System

So far only changes to the compiler frontend have been discussed. In this section the runtime system issues for Advanced Caliban are presented. Firstly the abstract requirements for a node in the final process graph are listed, then a more concrete notion of how they may be satisfied is given. Next the sequential runtime system on which Caliban is based is presented, followed by the hardware architecture of the AP1000. Finally, details of the implementation are presented.

5.2.1 The software architecture

This chapter has already provided a brief description of the new runtime system interface adopted by Advanced Caliban. This section will elaborate on the requirements of the runtime system and show how the chosen interface can meet those requirements.

Firstly some terminology; a process is a computation that generates a single stream, a node is a processor on the target machine.

The runtime system must provide/support the following:

- A way of denoting a process in the functional context
  
  This means that the interface must allow the calling functional program to describe what each process does.

- A way of mapping several processes onto one node
  
  The processes themselves have to be allocated onto nodes.

- A way of communicating a stream of values between processes
  
  Caliban processes on different nodes only communicate using streams. The runtime system must provide a way of sending individual stream values from a producer process to a consumer process.

- Allow overlapping of computation between a producer and a consumer
  
  Where a producer and consumer pair of processes do not reside on the same node the runtime system must allow them to compute in parallel.
- **Manage buffering of input and output data on node**
  No data items are allowed to be lost by the system. All communicated elements must be buffered by the consuming process until they are used.

- **Allow sharing of all computations on a particular node**
  When two processes are placed on the same node, their common subexpressions must be shared, without recomputation.

The model that Caliban uses to achieve parallelism is that the program consists of a graph that has to be evaluated. This graph is partitioned into possibly overlapping, but not subsuming, partitions. Each of these partitions becomes a process at runtime. The interfaces between partitions are the streams, Figure 5.16 shows the overlapping nature of the graph partitions. Each triangle represents the graph of a stream computation. The overlapped sections are shared subexpressions that each sharing stream computes locally. If two processes are placed on the same node then the results of these local subexpressions are shared between them without recomputation. Also, by extension, the stream output of co-habiting processes is shared naturally through the heap rather than by message passing.

![Figure 5.16: Partitions of the program graph](image)

When there was only one process per node, as in Basic Caliban, the implementation of a node was very simple. Now that more than one stream can be computed on a node the runtime system on the node has to share the CPU resources between the processes. No process must be starved of CPU time indefinitely as this could lead to a partial result, with processes waiting for data from the starved process. There are several ways in which this liveness property can
be assured. The most obvious is to use pre-emptive multitasking. A time slice is
given to each process as it starts to execute, either a timer interrupts the process
when the time is up, or the process yields at some natural break in computation
(e.g. a message send or receive). When the process yields it is placed at the
back of the queue to wait for its turn again and the next process in line is run
for its time slot. An alternative way to implement a completely fair scheme is
to use some resource other than time to ensure that all processes are given the
cpu and not indefinitely delayed, e.g. the number of function calls or backward
jumps (forward jumps alone cannot form a loop) could trigger a rescheduling.
This method is completely fair: an infinitely looping process cannot block other
processes from running because the running process is only given a finite amount
of a resource (time, function calls etc) before it is deselected.

A second approach is to assume that the programmer is not going to write
programs that loop. When this is the case it is safe to use natural break schedul-
ing alone. This means only rescheduling the processes when the current process
has finished a defined piece of work, for example when it sends a message. Al-
though this does not provide semantic correctness for all programs, it is simpler
to implement and costs less for the runtime system as some mechanism is needed
to count the resource usage of the process.

To achieve parallelism with a lazy language the demand for values has to be
modified. If the natural lazy demand is used then no parallelism exists as values
are not demanded until they are needed, therefore causing a co-routining effect.
To achieve parallelism in Caliban the demand for elements is stimulated early,
before they are required. Only elements demanded across node boundaries have
their demand profile changed, i.e. only elements sent down streams. Elements
are demanded one place before they are used so that when the consumer comes
to use the value, it is ready on the consumer for use. At that point the next
element is demanded so that it can be ready for use. The one-ahead demand is
an arbitrary choice, to create parallelism. Using a greater compute ahead would
overlap more communication time at the expense of using more heap space and
possibly delaying needed work for longer.
5.2.2 FCG

The runtime system is based on FCG, a code generator and runtime system for lazy functional languages written by Koen Langendoen. It was written as a backend for the FAST compiler that is being used by the rest of the Caliban compiler and so was a natural system to base the parallel implementation on. Fortunately, FAST/FCG is competitive with current state of the art lazy functional systems, beating the performance of many leading compilers [59, 58]. FCG was originally produced to experiment with the FRATS evaluation strategy for shared-memory parallel computers [60, 57].

Figure 5.17 shows the construction of the FCG system. Functional C is the output format of the FAST compiler, this is converted into FCG’s internal representation, Koala, and then a restricted C program is produced. All the code for the program is placed in one large C function in the final output as FCG maintains its own stack and uses jumps to implement function calls and tail recursion more efficiently than a normal C compiler.

![Figure 5.17: The layout of the backend compiler](image)

FCG is a good starting point because it already has support for shared memory parallel evaluation, including:

- locks on shared nodes
- easy task creation
- task queue
- thread suspension facilities

FCG also uses an efficient tagging mechanism to aid the garbage collector and evaluator. The runtime system relies on the program being type correct.
The type of an object is not determinable at runtime. The garbage collector and evaluator can tell the layout of the heap cells by using the two least significant bits of the object address for tagging information. To retrieve an address for an object from its pointer, the two least significant bits of the pointer need to be masked out the first time it is used in a particular context. Objects are a minimum of two words long, therefore the least significant bits would not be used in addressing as all the cells are word aligned. Figure 5.18 shows the tag and cell layouts. If the least significant bit of the pointer is set then the pointer is itself the data item, encoding its value in the remaining 31 bits. All functions know the type of their arguments, so the type of the basic value does not need to be stored. Well known constants such as [], TRUE, FALSE and all characters are stored using the Cons tag (xxx010) but with the value of the pointer being lower than the lowest allowed heap address. Aggregate data (packs, arrays etc) is stored like a curried function application, the pointer has a xxx010 tag and the first field of the cell has a yyy01 tag with the arity of the function it describes set to 0. The number of arguments in the node, ss, then describes the number of arguments in the cell. If there are $n$ arguments to the aggregate data then the cell will be $n + 1$ words long. Arrays are treated differently because the ss field can only encode numbers up to 31 (5 bits). The array bounds are placed as the second and third cells of the word with an escape value as the ss field of the first word. For arrays with $n$ elements, the cell is $n + 3$ words long. Figure 5.19 shows the cell layout for aggregate data.

Tags on the pointers to a shared application cannot be updated after the application has been evaluated because the tags are stored locally, at each point of reference rather than at the object. Instead, FCG uses indirection nodes to maintain sharing. Indirections are stored as a special function, in other words, when an application is evaluated the function code is changed to the indirection function and the argument is updated to the result of the application. Indirection nodes are short circuited during evaluation so that they are only used once for each site at which an indirected value is examined. Indirection nodes are deleted during garbage collection.
<table>
<thead>
<tr>
<th>Pointer Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx1 Basic</td>
<td>Basic data, e.g. integers, floating point numbers etc. Encoded in the pointer itself. 31 bits available for data.</td>
</tr>
<tr>
<td>xxx10 Cons</td>
<td>Data constructor cell. The first field provides extra tag information.</td>
</tr>
<tr>
<td>yyy00 list</td>
<td>List construction. The tail is the first element of the cell, and the head the second. When the tail is tagged as yyy10 then it is another cons cell, when it is tagged as yyy00 then it is an apply node.</td>
</tr>
<tr>
<td>yyy10 curried</td>
<td>The node is a curried function identifier, together with a number of arguments (fewer than the functions arity). yyy (=cccccccccaass) contains three fields describing the curry.</td>
</tr>
<tr>
<td>ccccccc 20 bits function id</td>
<td></td>
</tr>
<tr>
<td>aa 5 bits function arity</td>
<td></td>
</tr>
<tr>
<td>ss 5 bits number of args in node</td>
<td></td>
</tr>
<tr>
<td>xxx00 AP Application node. Again, the first field’s header is used to distinguish between two types:</td>
<td></td>
</tr>
<tr>
<td>yyy00 application</td>
<td>The first field holds a pointer to another AP node (yyy00) or to a curried function application (yyy10). The second field is the argument, which can be of any type.</td>
</tr>
<tr>
<td>yyy10 suspension</td>
<td>Encoded just like a curried function application, except that the number of resident arguments is the same as the function arity. If the arity field is 0 then the suspension is a CAF.</td>
</tr>
</tbody>
</table>

Figure 5.18: FCG’s cell layout

```
Cons cell
| Tail | Head |

Constructor
<table>
<thead>
<tr>
<th>Tag and size</th>
<th>data 1</th>
<th>data 2</th>
</tr>
</thead>
</table>

Array
<table>
<thead>
<tr>
<th>Size escape</th>
<th>lower</th>
<th>upper</th>
<th>data 1</th>
<th>data 2</th>
</tr>
</thead>
</table>

Figure 5.19: The cell layout of aggregate data in FCG
FCG uses several instruction level optimisations that are described in Chapter 5 of [58]. These include using registers as much as possible for stack manipulation and arguments/results for function calls. FCG also relies on the simple optimisations performed by normal C compilers, e.g. removing unused variables and unneeded assignment statements. The optimisations lead to very good performance figures, outperforming Clean (0.8.1), LML (0.998) and Glasgow Haskell (0.10).

5.2.3 The AP1000

The implementation of Advanced Caliban is targeted to the Fujitsu AP1000 distributed memory parallel computer. This section describes the machine and its software. It should be noted that few of the special features of the machine are used in the implementation, so it would be easy to port the implementation to other platforms provided that they satisfy a small number of criteria, which are:

- any processor can send a message to any other processor,
- message transmission is reliable,
- messages between two processors arrive in the order they are sent,
- sending is non-blocking and
- there is a non-blocking test to see if messages are waiting.

The computer is run by the Imperial College/Fujitsu Parallel Computing Center, IFPC. They provide access to parallel computing for academic community. Similar machines are sited in Australia, at the Australian National University, and in Japan.

The AP1000 was originally developed as a research machine to investigate the application of distributed memory computing to various scientific problems. During development several interesting subsystems, such as distributed disks/video
units and vector processors were added. These make the machine extremely flexible for general purpose use.

The section is split into two parts, first a hardware description is given. This is followed by information about the software features provided.

**AP1000 hardware architecture**

The Fujitsu AP1000 [49] is a distributed memory parallel computer whose architecture is shown in Figure 5.20. It is split into a host computer (standard Unix computer) and an array of 128 cell computers. Each cell consists of a SPARC-1 processor, 16MB of Random Access Memory and an autonomous message controller. The memories of the processors are completely disjoint, requiring programmed message passing to exchange information. Each processor’s dedicated autonomous message controller deals with all message transfer without interrupting the processor. The cells are connected by three communication networks:

- **T-Net**
  
The basic message passing network is a 2D torus, linking each processor to its four neighbours.

- **B-Net**
  
  Built on top of the mesh of cells is a broadcast network. Lines of cells are
connected by buses and the bus controllers are connected in a ring. The host computer is connected to the bus controller ring as if it were a bus controller.

- **S-Net**
  
  Built as a tree on top of the B-Net, the synchronisation network provides fast method to synchronise all cells.

The torus network uses wormhole routing techniques to improve message time and throughput. With wormhole routing, the message transfer time does not change linearly with the number of intermediate cells involved in the transfer. This is unlike a store and forward style network, as was used in the transputer system of chapter 3, where the message transfer time is a multiple of the time to transfer to one neighbour.

Wormhole routing, therefore, seems to have reduced the need for locality as communicating processes can be placed far apart with only a small overhead. This is unfortunately not wholly true. Wormhole routing requires that a channel from source to destination be kept open during the duration of the message transfer. Whilst this channel is open no other transfer can use the links involved in the currently operating transfer, therefore the pending operation is blocked until the channels are freed. Contention is the new limiting factor for communication. The more links that a message has to traverse, the more likely it is that the message will hit contended links. Of course, it is sometimes possible to organise the global communication pattern in such a way that no contention is created even when all nodes are communicating simultaneously. For example, in an exchange step of a mesh computation all processors can send data one step in the same direction without experiencing contention.

**AP1000 software support**

Each cell on the AP1000 runs a small operating system, providing a few simple capabilities (e.g. a clock, non-preemptively scheduled processes etc). Access to the communication and parallel programming facilities is via a set of libraries
linked with the application code. The structure of an AP1000 parallel application
is a set of program binaries; one for the host and one or more for the cells.
The host program is responsible for configuring the cells and interfacing the
application with the user. For the configuration, the host specifies the number of
cells that the application needs. Each cell is then assigned one or more binaries to
run. Finally the binaries are broadcast onto the cells using the B-Net. This means
that the load time is proportional to the number of different cell programs in
use, rather than the number of cells in use. The host version of the programming
libraries therefore contains all the initialisation routines as well as communication
routines for the operations that the host can take part in.

Each cell participating in the program executes one or more tasks. The cells
each have an ID, as do each of the tasks running on the cells. To reference a
particular task on a particular cell, both the cell and task IDs are needed.

Inter cell messages have the following properties:

- Message send is asynchronous
  The message send routines return before the message has reached its final
destination or been received. This is because the call simply hands control
of the message operation to the local message controller that then handles
the operation autonomously.

- Message receive can be either synchronous or asynchronous
  The libraries include both synchronous (the call blocks until a suitable
message is available) and asynchronous (the call returns with a message if
there is one waiting, otherwise is returns signalling that no message was
waiting) forms of message receive.

- Messages are tagged
  Each message is given a tag value, chosen by the programmer. This is used
to define what the message means to the receiver.

- Message passing is reliable
  The user is guaranteed that all messages will arrive at their destination
unless an error is signalled.

- Messages arrive in order
  The user is also guaranteed that all messages sent from a particular source to a particular destination will arrive at the destination in the order in which they were sent.

- Automatic buffering
  The system automatically buffers incoming messages into a circular buffer. Buffer slots are only reused when the message they belonged to has been received. If the buffer is full, all future messages to the cell are blocked until some of the messages have been received from the buffer, thus freeing space. Once space has been freed, the pending message will be buffered as normal.

One important facility that the cell’s operating system does not provide is any form of software interrupt. For example, the user cannot set an alarm to interrupt the current task after a pre-specified time, or when a message arrives.

5.2.4 The Non-Preemptive Runtime System

This section describes the implementation of the non-preemptive version of the Caliban runtime system. The additions required for the preemptively scheduled system are given later. The implementation comes in two halves, the host side and the cell side. The majority of the system is captured by the implementation of the interface functions: setup, connect and readstream. Each of these functions will be described in the following subsections, followed by further details of parts of the system not covered by the basic interface functions.

The functions setup and connect split the work of procnet in two:

<table>
<thead>
<tr>
<th>procnet</th>
<th>Now done by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build IO connections/control</td>
<td>setup</td>
</tr>
<tr>
<td>Link IO with evaluation</td>
<td>connect</td>
</tr>
<tr>
<td>Start evaluation of stream(s)</td>
<td>setup</td>
</tr>
</tbody>
</table>
The only route for information from the front end compiler to the runtime system is through the source code. This is because we use a standard functional language compiler between the outer phases. Therefore the calls to the Caliban primitives must encode all the information needed by the runtime system to organise the parallelism.

The same code is run on the host and all the cells, different versions of the Caliban runtime system primitives differentiate the behaviour between host and cell.

**Setup**

The task of `setup` is to initialise IO and evaluation control data structures and start any computations that are needed. It is the only function that is implemented on both the host and the cells. It has to be called at the top level application of the program, i.e. the call to `setup` will be evaluated first. From Figure 5.21 it can be seen that `setup` takes a `BundleList` and a `WiringList` as its arguments. The definition given in the figure is purely to provide the program with a sequential meaning. When run in parallel `setup` performs the actions given above.

Figure 5.21 shows the data structures maintained by the runtime system to manage the communication and computation. One `out_rec` is kept for each output stream, and one `in_rec` is kept for each input stream. The output record stores information about which output port it is associated with, how much data has been output so far, a record for each consumer of the data, the data already sent and a suspension containing the currently unrequested tail of the stream (if it is valid; if the stream is finished then the suspension is not valid).

The input record stores information about each of the streams needed by computations on the local cell. The `PortID` of the producer and a flag saying whether the stream has finished or not are needed. As will be seen later (5.2.4) only the latest value from the producer need be stored as earlier values reside in the heap. When a request for a data item is sent the `request` field is set to `FALSE`, this means that the data field is empty. Any attempts made by the
Chapter 5. Implementation of Advanced Caliban

/* Managing output streams */
struct out_rec {
    int stream;  /* Output port */
    int nel;     /* Number of els output so far */
    int cons;    /* Number of consumers */
    List consumers; /* Our consumers */
    int demand;  /* The number of consumers waiting for the next
                  * element to be generated by us. */
    List data;   /* Data generated already */
    List sizes;  /* Message sizes, relate to each data message */
    int finished; /* Has the stream finished? */
    word susp;   /* The unevaluated tail */
    task *the_task; /* The task struct for this stream */
};

/* A consumer */
struct consumer_rec {
    int nel;     /* Number of items requested by this consumer so far */
    int demand;  /* Is this consumer waiting for the next item? */
    int procid;  /* Location of this consumer */
};

/* Managing input streams */
struct in_rec {
    int procid; /* Location of source */
    int stream; /* */
    int finished; /* Has the stream ended? */
    int request; /* Has the last requested item been received? */
    char *data; /* Data from latest request reply */
    int datasize; /* Size of data from latest reply */
    List waiting; /* tasks waiting for the next result */
};

Figure 5.21: The data structures used by the runtime system to manage communication and computation
program to read the requested data are suspended onto the waiting list. When
the data arrives it is placed into the data field and the flag set to TRUE. The
suspended tasks can then be released.

The host uses a simplified version of the data structures as there can only
be one input stream from the process network and the single output stream (the
user input being fed to the cells) does not need to be evaluated. When the host
calls setup it evaluates the BundleList and the WiringList then works out how
many processors are needed to run the parallel program. It allocates the AP1000
cells needed and loads them with copies of the cell binary; which is the same
program object code, but linked with the cell version of the runtime system.
Once the loading has taken place the cells run independently. In the meantime
the host goes into a IO loop, receiving messages from the cells. It will either
receive data messages to be displayed to the user as the result of the program, or
it will receive requests for the user input which it can satisfy immediately. When
the final value in the output stream from the cells has been received, the host
broadcasts a termination message to all the cells to halt any further computation.
When all the cells are finished the host can also finish.

The cell version of the runtime system is considerably more complicated.
As with the host, the cells call setup first, evaluating the BundleList and the
WiringList. It is here that the host/cell functionalities diverge. The cells work
out which of the bundles they are responsible for by indexing into the BundleList
using their cell ID. An output record for each of the streams in the bundle is
created, with a pointer to the expression for each stream placed in the susp field.
Then the WiringList is examined. From it, the cell can work out which other
cells need access to the streams computed locally and can therefore construct
consumer records for each of them. Also, it can construct input records for each
stream that the locally computed streams need access to. It then sends requests
for the first data item of each of the streams it needs to import. Finally it calls
the cell runtime system control loop. Note that none of the suspensions for the
streams to be computed are evaluated — that will happen straight away in the
control loop, when the initial request messages from other cells are received.
Stream message protocol

Streams of data are used to connect cells together. Each element of a stream is communicated separately. To tell the producer that there is a demand for the next element of a particular stream, the consumer sends a request packet to the producer. The producer then computes the required value and sends the reply back to consumer. When a stream finishes the producer sends an “End” message to the consumers instead of a data message. As this stands, this would not produce any parallelism as the requests would not be sent until the value is actually to be used, therefore the consumer would have to block whilst the producer computes the value. The producer and consumer would co-routine with one or the other standing idle. To get parallelism it is necessary to stimulate the demand for values early. Initially, the Caliban system requests a value one ahead of the value currently in use. So a request for value \( n \) is sent out just before value \( n - 1 \) is used.

Cell control loop

The final action of setup is to hand control of the cell to the control loop. Before this loop is described in detail, one more structure needs to be described.

As the cell could be computing several streams at once there may be more than one expression to be evaluated at a particular point in time. Each of the streams has a thread of control associated with it, each thread of control has a task structure that describes the current state of execution of the thread. The runtime system uses a task queue to maintain all the pending tasks. Task structures are placed onto the queue when they need to be evaluated. Eventually they will be removed from the queue and evaluated.

The control loop controls the cell completely, it follows a simple cycle of dealing with messages, evaluating a task and repeating. Pseudo code for the loop is given in Figure 5.22. It is important to deal with messages as soon as possible so as to keep the system as live as possible. If messages are left for a long period without being dealt with then other cells may be stalled waiting for
data. To alleviate this as much as possible for the non-preemptively scheduled system, all pending messages are dealt with before any computation takes place in a particular cycle.

The inner loop deals with each message that is waiting. If the message is some data that has been requested then it is unpacked into the task's input record and any tasks that are waiting for it are placed back in the task queue ready to execute. If the incoming message is a request for data then the request is noted in the output record structure, and in the record associated with the requestor. Then, if not already in the queue or evaluated, the suspension for that data value is placed in the task queue. If the data is already available it is sent straight away. Note that packed versions of each of the data items are held in the output records so that the values do not need to be unpacked into messages for each destination. If the waiting message is a stop signal from the host, then the computation as a whole is finished and the cell can exit straight away.

After all the messages have been dealt with, the next task in the queue is evaluated. The evaluation of stream items is split into two, firstly the cons cell is produced using a weak head normal form (WHNF) evaluator. Once this is complete, the actual item is evaluated to normal form ready for sending. This split is made to reduce the size of the computation before the next yield point, i.e. when the incoming message queue is examined and another thread is allowed to execute.

A flag in the task structure defines which kind of evaluation to perform. After WHNF evaluation (i.e. the cons cell has been produced) the head's suspension is placed back into the queue and the tail is stored in the output record, ready for when it is requested. If, instead, NIL is the result of the WHNF evaluation, then the stream is noted as finished and “End” messages are sent to all waiting consumers.

After full evaluation the data item is ready for sending. The AP1000 can only send messages consisting of contiguous areas of memory, so the data item has to be packed into a contiguous array. This packed array is stored for future requests of the value so that the packing does not need to be repeated. Finally, copies of
loop
  loop
    if message not waiting
      break (leaves inner loop)
    Receive message
    case message.tag
      Data:
        Unpack the data and place it in \texttt{in\_rec}
        Place any tasks waiting for the data on the task queue
      Request:
        Note request in output structures
        If not already in the task queue, enqueue the
        task structure for the requested value
        If reply is already available, send it
      End:
        Note that stream has finished
        Wake all tasks that are suspended on this value
    Stop:
      exit
  end case
end loop

if task queue empty
  continue (continue with next iteration)
  susp = head of task queue
  if susp.type == \texttt{WHNF}
    evaluate susp to \texttt{WHNF}
  else
    evaluate susp to normal form
  end if
if susp finished evaluating
  if susp.type == \texttt{WHNF}
    if susp.expr == \texttt{NIL}
      Record the end of the stream
      Send "End" message to all consumers waiting
    else
      Place tail suspension into this task's output record
      Place head suspension into the task queue, with an evaluation type of Full
    end if
  else
    Pack result into the output record
    Send data messages to all all waiting cells
  end if
end if
end loop

Figure 5.22: Pseudo code for the control loop
the message are sent to all the consumers that are waiting.

Cell memory

Because each cell works completely independently, not sharing any memory, a cell can garbage collect when it likes. Caliban uses a standard two space collector. The only modification needed is to extend the root set to include each of the suspended tasks and the suspensions in the output records.

The garbage collector is also modified to recognise calls to readstream so that when a reference to a stream is collected the producer of that stream can be notified. This allows the producer to do two things:

- garbage collect any values waiting to be sent to the consumer that has just collected the related readstream, and
- possibly garbage collect the thread if no consumers are left it.

Connect

The interprocessor communication links are managed by connect. It acts as a distributed switch or environment patcher. A call to connect sits between references to a stream and the definition of the stream to allow the system to choose whether the stream is evaluated locally or fetched from a remote processor. The running program will not see a difference between local computation and remote fetching of the stream.

As identical code is run on all cells, information about where each stream is to be executed needs to be available to the call to connect so that the correct access to the stream can be made. The arguments to connect are the actual stream that the call is defining, the PortID of where that stream should be executed (remember the PortID defines a processor and an output port number) and the WiringList that defines the process network’s interconnection.

A short program is given to help demonstrate how connect works:
let
  a = connect a' (1,0) wiringList
  a' = ...
  b = ... a ...
  ...
in ...

Here, the expression named b is referencing the stream a. The actual definition of a is the expression named a' and all references to the stream have been indirected through the call to connect. This call states that a' should be computed on cell 1, and exported from port 0 of cell 1.

Say cell x is evaluating b which references a (and indirectly a'). The first time it needs a it will evaluate the connect call. The call to connect is only evaluated once for this cell, because all other references to a will share the result of the first evaluation of a.

If cell x is in fact cell 1, then a' should be evaluated locally. The local processor ID matches the processor ID in the PortID of the connect call so it just returns its first argument, the actual stream expression, to be computed locally.

If, on the other hand, the local cell ID does not match the processor ID of the call’s PortID then a has been imported. All that is needed to import the stream is the input port number on the local processor through which the stream will be fed. All references to the same imported stream on this cell will use the same port. To find the input port number, connect matches the PortID given to the left hand sides of the Connections in the wiring list. This produces a list of connections that involve a input reference to the stream in question. It then extracts the single reference, as there is exactly one, whose input cell ID is the same as the local cell ID. In this case the Connection will look like this: ((1,0),(x,y)), where (1,0) is the PortID of the remote stream and x is the local processor ID. The value y is the local input port number that is being looked for. With this information, connect can return (readstream y 1) as its result.
Readstream

The runtime system itself autonomously deals with the transfer of stream items between cells, readstream interfaces with the local runtime system to introduce the stream elements into the local program’s heap and to stimulate demand of a remote item. It takes two arguments, first is the local port number of the stream needed. The second is the position in the stream of the value wanted. When readstream is called it examines the input record associated with the port, which is its first argument, to determine if the item it is requesting is already locally available. Remember, stream items are pre-fetched; demand is placed for elements of a stream before it is known that they are needed. If the item is available then a request is sent off to the producer for the next item and the current item is unpacked into the heap, the input record is updated and the call returns with the following result:

\[
\text{element : (readstream } y (n+1))
\]

In other words a cons cell is created with the stream element and another call to readstream to fetch the tail. If the input record showed that the stream had ended then nil is returned.

If, on the other hand, the item being waited for is not yet available (the producer is still computing it, it is in transit or it is waiting in the message input queue of the processor), then the current task suspends itself, waiting on the input record. At this point the system returns to the control loop and any newly pending messages are dealt with before another task is scheduled. When the requested element finally arrives at the consumer, the runtime system places it in the input record and places the suspended task back on the run queue, ready to continue running with the newly arrived value. Finally, the re-awakened task will continue from the point of suspension and the call will succeed.

Node locking

Because several tasks are evaluating a shared program graph, graph locking needs to be employed so that the tasks do not interfere with each other. When an
evaluator enters a function application it marks it as locked by changing the function name to a special designated lock function. Should the task then block and another task try to evaluate the same application node, the second task will block as well, waiting for the application to be unlocked by the first task when it resumes.

If this were not done, each task that entered the function application block would evaluate a different copy of the result and therefore perform extra work. This would have a disastrous effect on performance.

As a result of this locking, it is interesting to note that only one task enters the `readstream` call for a particular element of an input stream. All the others are blocked at the application (or earlier) waiting for the leading task to complete the call and unlock the application node. This means that `readstream` does not need to support several tasks reading a particular value. Also, only one task can be queued off an input waiting for a stream item.

If a task finds a locked expression, it is guaranteed that the expression contains a blocked `readstream` call. This is because the only reason why an evaluator relinquishes control of the processor before finishing its work is if it becomes blocked waiting for a result from a remote processor. Blocking is therefore an efficient implementation strategy as the task could not have evaluated the blocked subexpression as it too would have become blocked waiting for the `readstream` result. Relinquishing the processor in this manner also allows the runtime system to receive the message that both tasks are waiting for should it have arrived.

**Extending compute ahead**

The default amount of compute ahead provided by the system is one. That is, as the consumer reads an element it requests the next element from the producer. It is possible to increase the amount of compute ahead by allowing the producer to evaluate the next item before it has received a request for it. This means that as soon as the request arrives it can be satisfied out of the producer’s result store. To help ensure that this extra, speculative work does not interfere with work that has actually been requested, the speculative work is only scheduled if the current
work queue is empty.

5.2.5 Extensions for preemption

The system described above suffers from fairness problems: it is possible for a stream computation on a cell to block all other stream computations on that cell indefinitely. This may seem like a problem, except that programmers tend not to write programs that loop... at least not deliberately! It is possible to extend the runtime system to be completely fair. The solution is to allow tasks to be interrupted after a period even though they have not finished evaluating. This is preemption — taking the ultimate control of when to hand over the CPU from the task and giving it the runtime system.

To implement a preemptive system some form of strictly increasing property needs to be “consumed” by the executing task, e.g. time, number of function calls etc. There must be no way in which the task can execute indefinitely without consuming some of the property, if this were not true then it would be possible for the task to maintain control over the CPU for ever.

The most obvious way of controlling preemption is to limit the amount of time which a task can execute before it is suspended. To implement this a software timer interrupt is needed to take control after the allotted time. Unfortunately the AP1000 does not provide such a service in the cell operating system so this cannot be done. Interestingly, the MPI standard does not include software interrupts either [69], if interrupts were to be used in the runtime system, it could not be written in portable MPI.

A second possibility is to increment and test a counter at a strategic point in the executing instruction sequence, e.g. whenever a function is called. If the count exceeds a certain number then the computation is suspended. To ensure that this is correct, a site for the increment/test has to be found such that it is regularly executed. The disadvantage of this system is that the program pays a price for the timing code even when tasks do not overrun their limits.

With preemptive scheduling there would two reasons why a task will give up
the processor:

1. blocking due to a readstream and

2. running out of its time (resource) slice.

If a task runs out of its time slice, it will leave part of the evaluation graph locked. This means that tasks attempting to evaluate a locked expression will block, even though there is no reason to (the subexpression could be evaluated fully). The second task will therefore lose some of its time slice.

Unlocking the locked graph when a task is descheduled is not really a proposition as it could become very expensive. Not only would the unlock be done, but also the graph would need to be relocked when the task was rescheduled. Finally the task would also have to deal with situation where it is rescheduled to find that some other task has evaluated the graph it was in the process of evaluating.

A better solution would be to have the task that finds the locked expression block on it, but reschedule the task that holds the lock for the remainder of the blocking task’s time slice.

Both of these solutions to a critical problem cause extra work to be performed due to their management. It is not clear that preemption gives any performance improvements, although this would have to be tested by experimentation.

5.3 Conclusions

This chapter has shown how the Advanced Caliban system can be implemented. The implementation is divided into two halves. First the front end handles the translation of the user’s application code into a form that explicitly exposes all the parallelism in one place. This is done by a series of program transformations:

1. Simplification

An extended simplification process was presented. Multiple annotations are coalesced into one annotation whilst the program is partially evaluated. Abstracted forms of annotations are evaluated, leaving an explicit network description.
2. **Network extraction**
   A new network extraction algorithm was shown. It allows multiple streams to be placed onto each processor. A new interface to the runtime system is used to allow the program to be partitioned without excessive program transformation.

3. **Phasing**
   An algorithm for mapping two subnetworks onto the same set of processors is presented. As with many mapping algorithms in this field, the complexity in the worst case is poor. Random mappings provide an escape route to allow large networks to be phased.

   Contained in these phases are the techniques for compiling the new features found in Advanced Caliban — multiple `moreover` clauses, multiple output nodes (Bundle), phased subnetworks (With) and indirect annotation referencing (Annot).

   Secondly, the implementation of the runtime system is presented. This is a complete reimplementation from the Basic Caliban runtime system, based around the new parallelism interface. The new interface consists of three functions:

   1. **setup**
      Used to build the internal data structures and tasks required to run each cell. It also starts off the computation by requesting the initial data from all the cells that supply the node.

   2. **connect**
      Implements a distributed switch to define where streams are computed. All streams are defined in terms of a call to `connect`. When the stream is meant to be computed locally its value is just returned, when it is to be computed remotely a stream reading suspension is returned.

   3. **readstream**
      This function imports stream values that have been requested from remote producers into the local heap.
The runtime system is structured as a multi-threaded graph reducer, similar to a shared memory parallel graph reducer.

All of the compiler has been implemented except for the multiple moreover and phasing extensions.

In the following chapter, several example application programs are presented along with some performance figures for the Caliban system.
Chapter 6

Using Advanced Caliban

During previous chapters, small example programs have been used to motivate and demonstrate various language features. In this chapter larger examples of Caliban programming are discussed in detail, both from the application viewpoint and from the Caliban system viewpoint. Performance results for the programs are presented, together with discussion of the general performance of the system. Finally some conclusions about Caliban as a language are also drawn.

Two applications are presented. Firstly a ray tracer, based closely on the one already seen in Chapter 2. The second is an application from the world of numerical problems; Jacobi Relaxation. It is a standard algorithm for solving large systems of simultaneous equations numerically. It is also representative of a large class of similar numerical methods. Unfortunately due to problems with the Advanced Caliban runtime system, there are no performance results for the Jacobi Relaxation application.

6.1 Raytracing

Chapter 2 showed a simple ray tracing program implemented firstly in a sequential functional language and then in Basic Caliban. Two types of parallelism were demonstrated: farm and pipeline parallelism. In the former case, the basic sequential program merely had one map operation changed into a semantically equivalent farm operation. The second, pipeline, case was more complicated as
it involved transforming the program to change the computation order.

The language demands made by the raytracer are very limited. In both the parallel versions presented there is only ever one output task on a processor and there is no situation where phasing could be used to any advantage. The computations have a very simple structure as there is no dependence between one computation and the next. The only way in which Advanced Caliban can improve implementation of the programs is by removing the annotation plumbing needed by the farm and parInsert operations.

Figure 6.1 shows the simple raytracer implemented in Advanced Caliban. The only difference between this program and the original Basic Caliban version is that the annotation is not explicitly plumbed back to the top level. This version is much cleaner than the original.

```caliban
rayTrace scene viewpoint = farm N impact rays
  where
  rays = generateRays viewpoint
  impact ray = fold earlier impacts
    where
    impacts = map (hit ray) scene

farm :: Int -> (a->a) -> [a] -> [a]

farm n func input = farmed moreover fan farmed slaves
  where
    farmed = unpartition slaves
    slaves = map (map func) jobs
    jobs = partition n input
```

Figure 6.1: Farmed raytracing in Advanced Caliban

Similarly, Figure 6.2 shows the pipeline raytracer in Advanced Caliban. The only differences between it and the original (Figure 2.22) are that the annotation plumbing has gone and that instead of placing streams with Node they are placed with Bundle. Notice how there is a moreover clause for each equation in the definition of parInsert, this mirrors exactly what the original plumbing version did. Where a tuple that added to the annotation was returned, now a moreover
clause is used. Also, remember that any fold operator can be used to collate the
final answers from the working processors — this is where the parallelism in this
version comes from.

\[
\text{rayTrace scene viewpoint} = \text{parInsert} (\text{map2 earlier}) \llimps
\]
\[
\text{where}
\]
\[
\llimps = \text{map impForObject scene}
\]
\[
\text{impForObject obj} = \text{map} (λ\text{ray} . \text{hit ray obj}) \text{rays}
\]
\[
\text{rays} = \text{generateRays viewpoint}
\]

\[
\text{parInsert} :: ([a] → [a] → [a]) → ([a] → [a])
\]
\[
\text{parInsert} f [s] = s \text{ moreover Bundle } [s]
\]
\[
\text{parInsert} f (s:ss)
\]
\[
\quad = \text{appl moreover Bundle } [\text{appl}] \text{ And Arc appl next}
\]
\[
\text{where}
\]
\[
\text{appl} = f s \text{ next}
\]
\[
\text{next} = \text{parInsert} f s:s
\]

Figure 6.2: A pipeline raytracer in Advanced Caliban

6.1.1 Raytracer performance

This section presents performance figures from running the two versions of the
program given above.

A note about performance data

Presenting performance information for parallel programs has always been a con-
tentious area.

Basic parallel program performance results can be presented in two ways:

1. Number of the processors used against execution time or

2. Number of the processors used against speedup achieved, where speedup is
defined as \( \frac{T(1)}{T(N)} \), the execution time for one processor over the time for \( N \) processors.
These styles of presentation lead to two different views of parallel program performance — real performance and processor efficiency. Which view is more important depends on what task is being performed. An algorithm designer probably works with efficiency, whereas a software engineer building a real system will want to know how many processors are needed to reach a certain speed or to solve a certain problem in a given time.

There are differing views about which programs should have their performance measured. Should a single algorithm be scaled for the various processor configurations being tested, or should different algorithms be used to achieve the best performance for each particular ensemble? The latter route would mean writing a different version of each program for each processor configuration. For example, the fastest sequential raytracer is probably not one that compares each ray with each object in the database, it would divide the scene database so that it can be pruned during the ray comparisons. The work presented here follows the former route as it is mostly concerned with the Caliban system rather than the problems being solved.

An important part of a program’s performance is not only the execution time for a fixed problem size with varying number of processors used, but also how the problem size changes the execution graph. Because of this, each problem will have several different problem size graphs to allow comparison.

In the following section the main style of graph presented will show the number of processors against time. This is in the belief that execution time is the most important aspect of a parallel program.

Each processor/time graph line will also include a “perfect speedup” line. This is calculated by dividing the execution time for one processor by the number of processors at a particular point. This measure is useful in determining what overheads are introduced by the parallel program and the Caliban system. Unfortunately this does not distinguish between the algorithmic overheads of the parallelisation, (e.g. inherent recomputation), and system overheads, (e.g. message handling times). These effects can be hard to separate. For example, a program may behave badly when used with a large number of processors because
of the evaluation order of Caliban; processors may sit idle for long periods of time. This could be considered an algorithmic problem — a different evaluation order could be imposed by rewriting the program, or it could be considered a system problem — if the system were to use a different order, better results would be obtained. Either way, the program still suffers from a problem and this is shown up by comparison to the “perfect speedup” line.

**Farm raytracers**

The initial set of runs show how the execution time of the program varies with the number of slaves used to compute the result. Figure 6.3 shows the execution time and the speedup graphs used for several different problem sizes. The problem sizes are 20×20, 40×40 and 100×100 image size. These graphs all use a 20 object scene.

The performance of the raytracer seems to bottom out between 15 and 20 processors. At first glance it looks as if it may be something to do with the number of objects in the rendered scene (also 20). However this cannot be the case as each slave is still rendering the whole scene, albeit for a smaller number of rays.

To show why exact linear speedup is not achieved for higher processor numbers an execution-time graph can be used. The x-axis is a timeline, each processor has a separate horizontal plot above the timeline. The lines show time when the processor is busy working, gaps show when it is idle because it is waiting for input or waiting for work to do. Figures 6.4–6.6 show three execution-time graphs for the raytracer with 20 objects and a 20×20 scene. The three plots are for different slave numbers.

The plots show that as the number of slaves increases, so the collector (processor 1) is spending more and more time working. In figure 6.4 (only two slaves) the collector process spends a significant proportion (about 95%) of its time idle, this indicates that it is easily able to service the two slaves. In figure 6.5, the collector is idle for less time (about 75%). It is starting to introduce delays into the farm (shown by gaps appearing in the slave plots). This is because it gathers the
Figure 6.3: Simple raytracer performance figures for three problem sizes
Figure 6.4: Execution-time plot of the 2 slave static raytracer

Figure 6.5: Execution-time plot of the 15 slave static raytracer
results in a round-robin manner. Therefore, when a slave delivers a result, that slave has to wait until the other slaves have delivered their results before it can deliver its next result. If the time taken by the collector to perform one complete round of result collections is similar to the time taken to compute one element then delays can start occurring due to unforeseen events like a cell performing a garbage collection or the network causing message delays. In figure 6.6, the time taken for a complete round of result collections is larger than the computation time, so each slave is idle for significant periods of time (about 60%).

The problem with this implementation of the raytracer is that the cost of communication is very high compared with the cost of computation for each job. If multiple rays were to be sent as a single job then the grain size of the jobs would be increased and therefore the ratio of communication time to computation time would improve. Figure 6.7 shows a version of the simple raytracer that has a variable grain size control. Block is used to divide the rays into contiguous blocks of size $M$.

With this version of the raytracer, a problem becomes immediately apparent. When the simplifier evaluates each stream to WHNF to remove aliasing it has
to evaluate the first cons cell of each slave’s output. In order to do this it must evaluate the first \(N\) blocks of rays. This could become quite expensive to evaluate at compile time. To get round this, a dummy value that can be evaluated easily can be prepended to the stream to stop the simplifier evaluating the ray blocks. The function \texttt{red} (meaning stop) and its associated \texttt{unred} do this. Here they are applied to the output of each slave. This area is discussed further at section 6.3.2.

\[
\text{rayTrace scene viewpoint} = \text{unblock results}
\]

\[
\text{where}
\]

\[
\text{results} = \text{farm N impact'} (\text{block M rays})
\]

\[
\text{rays} = \text{generateRays viewpoint}
\]

\[
\text{impact'} \text{ rays} = \text{map impact rays}
\]

\[
\text{impact ray} = \text{fold earlier impacts}
\]

\[
\text{where}
\]

\[
\text{impacts} = \text{map (hit ray) scene}
\]

\[
\text{farm} :: \text{Int} \rightarrow (a \rightarrow a) \rightarrow [a] \rightarrow [a]
\]

\[
\text{farm n func input} = \text{farmed moreover fan farmed slaves}
\]

\[
\text{where}
\]

\[
\text{farmed} = \text{unpartition (map unred slaves)}
\]

\[
\text{slaves} = \text{map red (map (map func) jobs)}
\]

\[
\text{jobs} = \text{partition n input}
\]

\[
\text{red} :: \text{Stream} \rightarrow \text{Stream}
\]

\[
\text{red s} = (\text{CHAR '!'}) : s
\]

\[
\text{unred} :: \text{Stream} \rightarrow \text{Stream}
\]

\[
\text{unred} = \text{tail}
\]

Figure 6.7: Farmed raytracing with variable grain size control

Figure 6.8 shows the effect of grain size on the computation time of the 100×100 image with 10 slaves problem. The graph shows erratic behaviour, with the execution time increasing. The general increase is caused by the \texttt{red} function. At program startup each slave uses its compute ahead to compute the first value it is to deliver, because each slave has had \texttt{red} applied to it there is no work to do to evaluate the first item as it is a constant. The slaves send their first result to the collector. The collector at this point is trying to evaluate the \texttt{unpartition} call which entails getting the first impact from the first slave.
Unfortunately, the first result from the first slave is thrown away by the call to `unred` before `unpartition` sees it, so the collector sits waiting for the next result from the slave. This means that it sits waiting while the first slave evaluates its first real impact. All the other slaves also have to wait as they will not receive a request for their second item (their first real impact) until their dummy value has been thrown away by the collector. This causes a knock on delay for each slave. Once all the dummy values are out of the system the farm behaves as normal.

Figure 6.9 shows an execution-time plot for a 10 slave raytracer, with the work packet size of 160 rays.

![Execution-time plot for a 10 slave raytracer, 100x100 scene.](image)

Figure 6.8: Different job sizes for a 10 slave farm raytracer, 100x100 scene.

The problem lies in the fact that the dummy value is taking the compute ahead for the first item. If the system used a greater compute ahead then real values would be computed during the dead time. In general the more compute ahead a program uses the more overlapped the computation becomes and the higher the parallel slackness. The experiment above is repeated with a compute ahead of two (Figure 6.10).

The plot shows that all the dead time at the start of the run is taken up, thus reducing the run time by about 10 seconds (50%). Not only has the dead
Figure 6.9: Execution-time for a 10 slave farm raytracer, 100x100 scene, 160 ray work packet.

Figure 6.10: Execution-time for a 10 slave farm raytracer, 100x100 scene, 160 ray work packet and compute ahead of two.
time been taken up, but the normal running has improved by removing any waits during the computation too.

If the extra compute ahead is applied to the normal farm experiment then run time is also improved. Figure 6.11 shows a 35 processor farm with different degrees of compute ahead. For this problem, there is no gain from going beyond a two element compute ahead.

![Graph showing different degrees of compute-ahead for a 35 processor farm raytracer](image)

Figure 6.11: Different degrees of compute-ahead for a 35 processor farm raytracer (100x100)

Figure 6.12 shows a 100×100 farm using a compute ahead of two. The percentage of time spent in the evaluator increases to above 95%, because the slave can get on with the next value before previous one has been collected. As all the results of each slave are to be used the work is guaranteed to be needed and therefore useful.

It has been shown that a compute ahead of two improves the performance of the farm raytracer. From investigations of blocking factors given that compute ahead it has been shown that a blocking factor of 40 is good. Combining these two values, Figure 6.13 shows the farm experiment as a whole.

When combining the extra compute ahead and a good blocking size the plot
follows the linear speedup line quite closely.

The pipeline raytracers

The pipeline raytracer shown in Figure 6.2 forms a pipeline with as many stages as there are objects in the scene to render. For the purposes of experimentation more flexibility is needed. Figure 6.14 shows an extended pipeline raytracer that allows the number of stages to be varied by changing the number of objects assigned to each stage. As each stage can now render more than one object it is necessary for impact data for each ray to be folded to produce the single best impact for that ray. This means that impact selection now happens in two places, once at the very top level of the program where impacts from each pipeline stage are merged, and once on each pipeline stage where the best impact for that objects on that stage is produced.

Figure 6.15 shows the results of running the pipeline raytracer for different numbers of pipeline stages. The speedups obtained do not closely match the perfect speedup line, unlike the farm results obtained earlier. This is because computations in a pipeline are tightly coupled to each other. A delay in one part
Figure 6.13: A static farm with compute ahead of two and blocking factor of 40.
rayTrace scene viewpoint = impacts
    where
    impacts = parInsert (map2 earlier) llimps
    llimps = map (rayTrace rays) subScenes
    rayTrace rs ss = map (\r. fold earlier (map (hit r) ss)) rs
    subScenes = block N scene
    rays = generateRays viewpoint

Figure 6.14: Pipeline farm with control over the number stages

of the pipeline causes delays elsewhere: a pipeline runs at the speed of its slowest stage. In the 20 processor case, the percentage of time spent in the evaluator is as low as 50%. Even if the processors were working at full efficiency, there would still be a difference between the actual time and perfect speedup time. This is to do with the fact that the rays are being generated on each processor rather than being passed along the pipeline. As the problem size increases so does the time spent garbage collecting on each cell. The collections also cause further delays in the pipeline.

As with the farm version of the raytracer, grain size can be an issue. Figure 6.16 shows a new version of the pipeline raytracer where the rays are grouped into blocks to be passed down the pipe.

Running this raytracer with four processors and varying the size of the work packets on a 40×40 image gives the graph in Figure 6.17. It can be seen from this graph that when the work packet contains only one ray, the execution time is greater than in the non-blocked case. This is because of the overhead in building the single ray work packet and then dismantling it for evaluation. As soon as two rays are put into the packets, the execution time improves. A minimum is reached around the 10-20 ray packet size and then the results get worse again. The reason for this is that parallelism is being lost as the packet size gets bigger. As the packet size increases so does the time to fill and empty the pipeline. Given a fixed problem size, this means that the percentage of time when the pipeline is not full (during the filling emptying stages) increases, leading to observed
Figure 6.15: The basic pipeline raytracer for three problem sizes
blockedRayTrace scene viewpoint
    = unblock blImpacts
    where
    blImpacts = parInsert (map2 blockedEarlier) llimps
    blockedEarlier a b = map2 earlier a b
    llimps = map (map rayTrace rayBlocks) subScenes
    rayTrace rs ss = map (λr. fold earlier (map (hit r) ss)) rs
    rayBlocks = block M rays
    subScenes = block N scene
    rays = generateRays viewpoint

Figure 6.16: The pipeline raytracer with grain size control

inefficiencies. Therefore determining the best packet size is a juggling act between communication overhead reduction and the amount of parallelism.

Figure 6.17: Different job sizes for a 4 stage pipeline raytracer, 40×40 scene.

As with the farm example, the pipeline was run with varying compute aheads. Figure 6.18 shows the result for a 5 processor pipeline and a 40×40 image. Again, the main improvement comes by increasing the compute ahead to two.

Taking the best of the results from the previous two graphs and combining
Figure 6.18: Different compute ahead values for a 40×40 scene pipeline raytracer. These results in the run times given in Figure 6.19. The gap between linear speedup and the results obtained has improved, although not has much as would be desired.

Figure 6.19: A 40×40 scene pipeline raytracer, with compute ahead of two and block size of 10.
6.2 Jacobi Relaxation

One application area where parallelism can be put to great use is the solution of large-scale numerical problems. Typically this involves solving large systems of simultaneous equations. The equations can be represented as a multidimensional array of coefficients. The algorithms iterate some function over the array to generate better and better approximations to the solution.

Each generation is usually dependent on the previous generation’s values, thus sequentialising the production of the each new approximation to the final result. This causes synchronisation between the evaluation of generations. But, as the arrays can be very large there is scope for data parallelism within the computation of each generation. A further synchronising effect is produced by the control mechanism needed to determine if the computation has reached the desired accuracy.

This section presents an implementation of one such numerical algorithm — Jacobi Relaxation. Firstly the problem is specified, then a sequential functional solution is developed. From the sequential version a naive parallelisation is produced. Finally, a more realistic parallel version is presented.

6.2.1 The problem

Here is an example of using the Laplace Equation, motivated from [61]. Consider a two dimensional conducting metal sheet with a series of constant voltages applied along each edge. The problem is to determine the pattern of voltages that is formed on the sheet by these fixed edge voltages. This can be solved by using Laplace’s equation for all the internal points on the sheet.

If \( v(x,y) \) is a function that defines the resulting voltage at a point \((x,y)\) on the sheet then Laplace’s equation is as follows:

\[
\nabla^2 v = \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} = 0
\]

This is a linear partial differential equation. One can solve this class of equa-
tions numerically by using a two-dimensional grid of points representing the metal sheet in the problem. To increase the accuracy of the solution more points can be used. As the number of points increases so does the amount of computation required to calculate the result. There is, therefore, a trade off between accuracy and computation time.

As it stands, equation 6.1 cannot be solved directly. Instead the differential equation has to be manipulated into a difference equation. In [55], it is shown that this manipulation is done using a Taylor series expansion with some simplification. The manipulations lead to the following formulation of the equation:

\[ v(x, y) = \frac{v(x + h, y) + v(x, y + h) + v(x - h, y) + v(x, y - h)}{4} \]  

(6.2)

Here, \( h \) is the mesh size. When the sheet is represented as an array, \( h \) can be made equal to one so that the values \( v(x + h, y) \) refer to the neighbouring positions of a point in the array.

From equation 6.2, it can be seen that there exists a very simple relationship between the value for a particular point and the values of the neighbours, i.e. the value is the average of its neighbours. Therefore to compute a value, the values of the neighbours are needed. This relationship is described by the stencil given in Figure 6.20.

![Figure 6.20: Requirement stencil for the transformed Laplace equation](image)

Jacobi Relaxation consists of computing a series of closer and closer approxi-
mations to the final result as defined by equation 6.2. To compute the series of approximations, the following equation is applied to generate each new generation:

\[
v_{i+1}(x, y) = \frac{v_i(x + h, y) + v_i(x, y + h) + v_i(x - h, y) + v_i(x, y - h)}{4}
\]  

(6.3)

where \(v_i(x, y)\) is the voltage at the point \((x, y)\) at generation \(i\). The problem remains of where to find the initial values for each grid location. The edge values are set by the problem, and are therefore known. All the other grid values can start off as arbitrary values, e.g. zero.

Through successive computation steps the values of \(v_i\) will converge towards the correct result as defined by equation 6.2. Some accuracy variable is needed to determine when to stop iterating. Methods for convergence testing will be discussed later.

6.2.2 Local neighbourhood operations

A local neighbourhood operation, LNO, is an array operation where the computation of a new element requires data only from its local neighbours and the previous value of the element itself. Figure 6.21 gives declarative definitions for 1- and 2-dimensional array LNO functions. These definitions do not specify what happens at the edges, there are two options:

1. toroidal edges, or
2. fixed edge values.

The choice of toroidal or fixed edges is problem dependent. With the first option, the array is made to wrap around so that with \texttt{lno1D}, for example, the value at index 1 of the result is \((f \ x_n \ x_1 \ x_2)\) and the value at the \(n^{th}\) index of the result is \((f \ x_{n-1} \ x_n \ x_1)\). This is generalised for the 2D version of the function by wrapping in all four directions.

The second option is to provide the LNO function with some fixed values that are used for the edge values. One way to represent this is to make the array one
element larger at each edge to accommodate the fixed edge values. With this the indices would run from 0 to $n + 1$, but the LNO call would only compute the values from 1 to $n$ for the resultant array.

\[
\text{lno1D :: } (a \rightarrow a \rightarrow a \rightarrow b) \rightarrow \text{Array Int a} \rightarrow \text{Array Int b}
\]
\[
lno1D \ f \ (x_1, x_2, \ldots, x_n) = \langle \ldots, f \ x_{i-1} \ x_i \ x_{i+1}, \ldots \rangle
\]

\[
\text{lno2D :: } (a \rightarrow a \rightarrow a \rightarrow a \rightarrow a \rightarrow b) \rightarrow \text{Array (Int,Int) a} \rightarrow \text{Array (Int,Int) b}
\]
\[
lno2D \ f \ \begin{bmatrix} x_{[1,1]}, & x_{[1,2]}, & \ldots, & x_{[1,n]} \\ \vdots & \vdots & \ddots & \vdots \\ x_{[m,1]}, & x_{[m,2]}, & \ldots, & x_{[m,n]} \end{bmatrix} = \begin{bmatrix} \ldots \\ f \ x_{[i,j-1]} \ x_{[i,j+1]} \ x_{[i+1,j]} \ x_{[i-1,j]} \ x_{[i,j]} \ \ldots \end{bmatrix}
\]

Figure 6.21: LNO functions for 1D and 2D arrays

A second way to implement the fixed edge values approach is to define a structure called a patch. It is an array together with the edge values. Figure 6.22 shows the Haskell definition for a patched 2D LNO function. Haskell arrays are all one dimensional, so the bounds of the 2D array need to be explicitly stored in the patch. The function \text{mkarray2d} builds a new array of the desired size where each element is an application of the supplied function to a tuple containing the element’s coordinates. Thus, the definition of \text{data}, in the body of \text{lno}, is a call to \text{mkarray2d} that uses \text{buildApp} to build an application of \text{f} for each element in the input array. The arguments to these applications are the current values of the neighbours and the element itself. To do this, \text{buildApp} uses \text{pick}, which either selects a value from the data in the patch or from one of the edges as appropriate.

### 6.2.3 Sequential Jacobi Relaxation

From looking at the stencil, Figure 6.20, and the defining equation, equation 6.3, it is obvious that Jacobi Relaxation is an example of an LNO. The problem description necessitates that the version of LNO that uses fixed edge values is required. Figure 6.23 shows how a simple sequential version of the algorithm can be implemented. The top function, \text{jacobi}, requires a patch containing the initial value of each point on the plane together with the values of the all the
type Patch a = ((Int, Int), Body a, (Edge a, Edge a, Edge a, Edge a))
type Body a = Array (Int, Int) a
type Edge a = Array Int a
type LnoFunc a = a → a → a → a → a → a

lno :: (LnoFunc a) → (Patch a) → (Patch a)
lno f p = (bds, data, edges)
  where
    data = mkarray2d bds (buildApp f p)
    (bds, _edges) = p

buildApp :: (LnoFunc a) → (Patch a) → (Int, Int) → a
buildApp f p (x, y) = f (pick p x (y-1)) (pick p x (y+1))
  (pick p (x+1) y) (pick (x-1) y)
  (pick p x y)

Figure 6.22: A 2D LNO in Haskell

elements. It produces a list of patches that are successively closer and closer approximations to the solution of the system.

jacobi :: (Patch Float) → [Patch Float]
jacobi p = generations
  where
    generations = iterate (lno average) p
    average n s e w o = (n+s+e+w)/4

iterate :: (a → a) → a → [a]
iterate f x = x : (iterate f (f x))

Figure 6.23: A simple sequential implementation of Jacobi iteration

This solution does not take into account the convergence of the system. To do this, a separate filter can be applied to the output of jacobi which computes the residuals for each generation and can therefore decide when the computation has proceeded far enough. Because the language is non-strict, only those generations demanded by the convergence test will actually be evaluated.

One example of a residual function may be to compute the percentage change
of each element position from one generation to the next. The residual value for the whole generation would then be the largest change computed for that generation. When the largest change has decreased to below a certain threshold then computation can cease. This method is implemented in Figure 6.24.

```plaintext
convergenceJacobi :: (Patch Float) → Float → (Patch Float)
convergenceJacobi init limit = selectList solutions control
  where
    solutions = jacob init
    control = map (limit >) convs
    convs = convergences solutions

convergences :: [Patch Float] → [Float]
convergences patches = nmap convergence patches

convergence :: (Patch Float) → (Patch Float) → Float
convergence (bs, a, _) (_ b, _) = arrayfold max 0.0 cs
  where
    cs = mkarray2d bs cpoint
    cpoint (x, y) = percChange a!(x,y) b!(x,y)

percChange :: Float → Float → Float
percChange a b = \frac{100(a-b)}{a}

nmap :: (a → a → b) → [a] → [b]
nmap f [] = []
nmap f [a] = []
nmap f (a:b:rest) = (f a b) : (nmap f (b:rest))

selectList :: [a] → [Bool] → a
selectList (v:vs) (False:controls) = selectList vs controls
selectList (v:vs) (True:controls) = v
```

Figure 6.24: Adding convergence to the sequential Jacobi iteration

The program works by generating a list of boolean values, one for each generation, that define whether the solutions have reached the required convergence or not. As each False value is generated the corresponding solution is thrown away. When the solutions reach the right accuracy, a True value is produced by convergence and the corresponding generation of the solution is returned.
This is not the most efficient solution possible for the problem, but it is close to the specification. One source of inefficiency is that the \textit{lno} function is called once for each generation, this means that each of the neighbour selections is done for each element for each generation. The program can be transformed into a different form by making the elements of a patch into streams (lists) of values. Each list represents the values that an element has from generation to generation.

This is a data type transformation. With the basic version of Jacobi Relaxation, the type of the generated data was [Patch Float]. In making each element into a stream of generations, the type of the generated data becomes Patch [Float]. All the functions that previously acted on the first type can now be transformed easily to the second. The transformations follow naturally from the changed shape of the central data structure.

Figure 6.25 is the transformed streamed Jacobi Relaxation solver. It will be used in the next section as the basis for a parallel version of the program. What the transformation has done is to push the \texttt{map} inside the data structure: from the patch level to the element level. The rest of the computation structure stays the same, \texttt{max} is still being \texttt{arrayfolded} over some representation of the residuals for the system, only now each element of a residual array is a stream of residuals for that particular location in the input patch.

### 6.2.4 Parallel Jacobi Relaxation

From the defining equations, it is clear that generations cannot be computed in parallel. Each value in a generation refers to values of the previous generation. There is, however, intra-generation parallelism. Each value within a generation can be computed in parallel as its computation will only refer to values from the previous generation.

To parallelise a program with Caliban it has to be divided into stream producing processes. Each process can then be placed and computed in parallel. Figure 6.25 shows a version of the program that generates a stream for each element position in the plane that is being solved. Each stream is the sequence of
convergenceJacobiS :: (Patch Float) → Float → (Patch Float)
convergenceJacobiS init limit = selectListS solutions control
    where
    solutions = jacobiS init
    control = map (limit >) convs
    convs = convergenceS solutions

jacobiS :: (Patch Float) → Patch [Float]
jacobiS init = result
    where
    result = join init generations
    generations = lno (map5 average) result

convergeS :: Patch [Float] → [Float]
convergeS (∅, data, ∅) = arrayfold (map2 max) (repeat 0.0) residuals
    where
    residuals = arraymap (nmap percChange) data

join :: (Patch a) → (Patch [a]) → (Patch [a])
join ((x,y), initData, (n, s, e, w)) (∅, computedData, ∅) = ((x,y), streamData, (streamN, streamS, streamE, streamW))
    where
    streamData = mkarray2d (x,y) (joinEl initData computedData)
    joinEl i c (x, y) = i!(x,y) : c!(x,y)
    streamN = mkarray x (streamIt n)
    streamS = mkarray x (streamIt s)
    streamE = mkarray y (streamIt e)
    streamW = mkarray y (streamIt w)
    streamIt a i = repeat (a!i)

selectListS :: (Patch [a]) → [Bool] → (Patch a)
selectListS (bounds, data, (n,s,e,w)) (c.cs) = (bounds, heads, edgeHeads) if c
    otherwise
    selectListS (bounds, tails, edgeTails) cs
    where
    heads = ahead data
    edgeHeads = (ahead n, ahead s, ahead e, ahead w)
    tails = atail data
    edgeTails = (atail n, atail s, atail e, atail w)
    ahead = arraymap head
    atail = arraymap tail

map5 :: (a→b→c→d→e→f) → [a] →[b] →[c] →[d] →[e] →[f]
map5 f [] [] [] [] [] = []
map5 f (a1:a2) (a2:a3) (a3:a4) (a4:a5) (a5:as5)
    = (f a1 a2 a3 a4 a5) : (map5 f as1 as2 as3 as4 as5)


Figure 6.25: A streamed sequential version of the Jacobi iteration
approximations of the result for its position, referring directly to its neighbouring streams to gather the values required for each new generation. This means that each of the streams can be placed to produce a parallel program. Figure 6.26 shows this naive parallel implementation of the algorithm.

\[
\text{jacobiS :: (Patch Float) } \rightarrow \text{ Patch [Float]} \\
\text{jacobiS init = result moreover annotation }
\]

\[
\text{where}
\]

\[
\text{result = join init generations}
\]

\[
\text{generations = lno (map5 average) result}
\]

\[
(\_ \_ \_ \_ \_ ) = \text{ lno place result}
\]

\[
\text{annotation = arrayfold (And) NoPlace places}
\]

\[
\text{place :: LnoFunc [Float] Placement}
\]

\[
\text{place n s e w o = fan o [n, s, e, w] And Bundle [o]}
\]

Figure 6.26: A naive parallel implementation of Jacobi Iteration

This implementation is naive for several reasons:

1. The grain of the processes is very small. For the current electrical potentials example, each process does three additions and one division for each generation. It then needs to communicate four values with the neighbours. The communication time would swamp the computation time.

2. A large number of processors are needed. Even a modest 100\times100 problem needs 10000 processors! Techniques for processor reduction from Chapter 4 can be used, i.e. bundling sets of streams together to form larger computational units.

3. Data exchange has a small grain size too. Even with bundled processes, the exchange of data between processors is wasteful. Each element will still be transferred independently, resulting in poor message passing performance \((n \text{ sets of message overhead, where } n \text{ is the number of elements that need to be communicated})\).

4. The work performed by the Caliban compiler to extract the process network is proportional to the problem size, not proportional to the amount of
parallelism the program represents. This is the most important concern. As the problem is scaled, so the compile time will scale. If it takes one second to partially evaluate and extract each stream required for the computation, then a mesh of $1000 \times 1000$ streams would take over 270 hours to compile! This is always true, no matter how many processors the streams are bundled into. This is because each stream is separated and then bundled together to form larger computations.

A method is needed of describing the parallelism that is only has complex as the parallelism that it describes.

**Patch based parallel implementation**

In common with imperative solutions to parallelising Jacobi Iteration, an approach is to divide the plane up into patches that are computed by different processors. Figure 6.27 shows how the plane can be divided up into patches and how those patches will interact in the final implementation. Remember that each element of the plane needs only its north, south, east and west neighbours to compute its next approximation. The elements on the edge of each patch, therefore need elements from a neighbouring patch. Therefore each patch has a set of four edges that it exchanges with its neighbours between each generation of computation.

There are two notions of a patch that are needed. One represents the patch before computation, when the edges stored in the patch are those of the neighbours — this is an *unfolded* patch. The second is the version after computation and just before edge exchange where the edges are those of the patch itself. This is called a *folded* patch. After each computation step the folded patches are unfolded by communicating the edges with their neighbours. At this point the next computation step can proceed. Figure 6.28 shows how a patch is unfolded to allow computation.

Each processor will now be represented by a patch. A new type of patch needs to be introduced, it is a *streamed patch*. Instead of there being a single
6.2. Jacobi Relaxation

Full plane, consisting of several patches.

Neighbouring patches exchange edges.

Figure 6.27: Dividing the plane into patches and patch communication
Chapter 6. Using Advanced Caliban

Figure 6.28: Unfolding a patch ready for computation
array data held and set of edge arrays, the streamed patch has a list of data arrays and lists of edge arrays. Each element of the data array lists represents a successive approximation to the correct data array for its patch, in much the same way as the streamed single element LNO worked. Unfortunately the patch definition already available cannot be instantiated to produce the data structure needed. This is because it is defined to store arrays of objects, not lists of arrays of objects. Lists of patches do not have the correct properties either as they do not allow the separation of edges from the data arrays. With the streamed patch, the data array and the edges can be placed separately so that only the remote edge required by a processor needs to be imported and not the whole patch. This reduces the communication overhead of the program.

Figure 6.29 shows the extended LNO definitions for folded patches and for streamed patches. The folded patch definition is identical in structure to the unfolded version. The only difference lies in interpretation. The same is true for the streamed version of both types of patch. The streamed patches consist of a stream (list) of patch bodies (representing successive approximations to the final result) and streams of edges. These edges are either edges from the neighbour or from the patch itself, depending on which version of the streamed patch is used. The structure of slno is very similar to that of lno, except that the provided function is mapped over the incoming generations provided by the input streamed patch. The incoming generations are converted into a stream of patches by spToPatches. The provided function is then mapped over the stream using buildBody. The definition of buildBody is the same as the array construction computation for lno. As this is an unfolded LNO operation, the edges are returned untouched. The folded version of slno proceeds by mapping an extraction function over the streamed patch produced by a call to slno.

Using the streamed version of LNO the parallel Jacobi Iteration program is shown in Figure 6.30. In this program each processor is represented as a streamed patch. The data and edge streams of each patch are placed and bundled together to form one node. The whole computation is then represented as a patch of these streamed patches. The initial input to the program, an unfolded patch of
– A folded patch
  type FPatch a = Patch a

– Streamed versions of patches
  type SPatch a = ((Int, Int), [Body a], ([Edge a], [Edge a], [Edge a], [Edge a]))
  type SFPatch a = SPatch a

– Streamed LNO slno :: Lnofunc a b → SPatch a → SPatch b
  slno f input = (bds, inter, edges)
  where
    inter = map (buildBody bds f) (spToPatches input)
    (bds, data, edges) = p

buildBody :: (Int, Int) → Lnofunc a b → Patch a → Body b
buildBody bds f init = mkarray2d bds (buildApp f init)

spToPatches :: SPatch Float → [Patch Float]
spToPatches (bds, b, (n, s, e, w)) = patches
  where
    patches = map5 (buildP bds) b n s e w
    buildP bds b n s e w = (bds, b, (n, s, e, w))

– Perform a streamed LNO, returning a folded patch.
foldSLno :: (Lnofunc a b) → (SPatch a) → (SFPatch b)
foldSLno f input = ((x, y), data, (n, s, e, w))
  where
    ((x, y), data, _) = slno f input
    n = map (mkarray (1, x) (extract 1 1 1 0)) data
    s = map (mkarray (1, x) (extract 1 y 1 0)) data
    e = map (mkarray (1, y) (extract 1 x 1 0)) data
    w = map (mkarray (1, y) (extract 1 1 1 0)) data

– Extract the i\textth value of a range of values from a 2D array
extract :: Int → Int → (Body a) → Int → a
extract xinit yinit xskip yskip dat i = data !! (xinit + ((i-1) × xskip), yinit + ((i-1) × yskip))

Figure 6.29: Streamed patch LNO
numbers, is converted into a patch of patches of numbers. The edge values to the whole problem, i.e. those values that are fixed in this example, are made into pseudo-streamed patches where only the relevant edge is defined.

Just as in the sequential stream version, the top computation forms a loop; the LNO computation refers to the final result patch and the join, that generates the final result, refers to the result of the LNO. This recursive reference is used to start the whole computation off and to allow each processor access to the generation before the one it is computing.

Note that the top level LNO structure disappears during compile time simplification, leaving a set of communicating processes. The outer patch structure is not used in the resulting program, having been partially evaluated away.

It is the representation of these patches that is important. With the naive solution the patches were not explicitly represented, rather their constituent parts were assembled to form logical patches. This meant that the Caliban compiler had to do a disproportionate amount of work, compared to the amount of parallelism extracted, to build the process network. With the patched solution, only the top level patch structure and the join are evaluated at compile time. The top level patch is only as large as the amount of parallelism that the program represents, so the compiler’s work is now in proportion to the amount of parallelism extracted and not the problem size.

Another effect of the new program is to vectorise the edge exchanges. An edge is now communicated with a single message, rather than \( n \) single element messages. This reduces the communication overhead.

The edge communication comes about naturally as a result of the folding and unfolding operations. The \texttt{unfold} operation in \texttt{stream} connects all the edge streams to their respective consumers. The \texttt{map (edge ...)} in \texttt{slno} does the folding step, to construct the edges after each computation step.

\textbf{Parallel convergence}

So far, the patched parallel program does not include the convergence computation. This can be added as a filter, in the same way as it was for the other
parallelJacobi :: Patch (Patch Float) → Patch (SFPatch Float)
parallelJacobi init = res moreover annotation
  where
  res = join init comp
  comp = lno streamf res
  annotation = arrayfold (And) NoPlace places
  (anno places, _) = lno placePatch res

streamf :: LnoFunc (SF Patch Float) (SF Patch Float)
streamf n s e w o = foldSLno average newo
  where
  newo = unfold n s e w o

unfold :: LnoFunc (SF Patch Float) (SPatch Float)
unfold (anno s) (anno s)
(anno n, anno s)
(anno w)
(anno e, anno s)
(dbs, data, _) = (dbs, data, [s,n,w,e])

placePatch :: LnoFunc (Patch (SPatch Float)) (Patch Placement)
placePatch (anno s) (anno s)
(anno n, anno s)
(anno w)
(anno e, anno s)
(dbs, data, [on,os,oe,ow]) = fan data [n,s,e,w] And
  Bundle [data, on, os, oe, ow]

Figure 6.30: Parallel Jacobi Iteration using streamed patches
versions of the program. Figure 6.31 shows the code required to add convergence testing to the patched parallel program.

```haskell
convergenceJacobi :: (Patch Float) → Float → (Patch Float)
convergenceJacobi init limit = display selected moreover finalAnnotation
  where
    res = join init comp
    comp = lno streamf res
    (res selected, _) = lno (selectControl control) res
    control = map (limit >) residuals
    residuals = arrayfold (map 2 max) (repeat 0.0) convergences
    (convergences, _) = lno (patchConv percChange max 0.0) res

    finalAnnotation = ((computation Annot With selectAnnot)
                           With convergenceAnnot) And controlAnnot
    computationAnnot = arrayfold (And) NoPlace places
    (places, _) = lno placePatch res
    controlAnnot = Bundle [control]
    selectAnnot = arrayfold (And) NoPlace (arraymap node selected)
    convergenceAnnot = arrayfold (And) NoPlace (arraymap node convergences)

    selectControl :: [Bool] → LnoFunc (SPatch Float) [Float]
    selectControl control selected (o, _) = isit control o
      where
        isit (False :rest) (d : dat) = isit rest dat
        isit (True :rest) (d : dat) = d

    patchConv :: (a → a → b) → (b → b → b) → b → LnoFunc (SPatch a) → [b]
    patchConv f j b selected (o, _) = nmap (convPatch f j b o) bds

    convPatch :: (a → a → b) → (b → b → b) → b → (Int, Int) → Body a → Body a → b
    convPatch f j b bds d1 d2 = arrayfold j b convs
      where
        convs = mkarray2d bds convEl
        convEl i = f (d1!i) (d2!i)
```

Figure 6.31: Adding convergence to the patched parallel Jacobi Relaxation program

The convergence computation consists of two fold operations; one to fold the residual values of each of the points on a particular processor patch into a single value for that patch, and another to fold the patch residual values into a convergence value for whole plane. The patch residual computations should be placed
on the same processor as the patch that they are examining, otherwise the high overhead of copying each patch array to another processor for each generation would be incurred. To do this, the process networks for the computation and convergence are phased together. Phasing guarantees that, when possible, interface nodes of the phases will be mapped to the same place. Each node in the convergence graph refers to only one node in the computation graph, so the convergence graph nodes will be automatically placed on top of the corresponding node in the computation graph. It is, of course, possible to express this mapping explicitly, but using phasing is easier and less error prone.

The streams of residual values from each patch are folded to produce a stream of overall values for the plane. This is converted into a stream of control values which is then used to select the patch bodies that need to be returned as the final result of the computation as a whole. Again, the selection process needs access to the patch data and should be located on the same processor otherwise excessive communication would occur. The same technique for phasing the selection process network with the patch processors is used as was used by the convergence network. Again, this could be expressed explicitly if the programmer wanted to.

Finally, the process that converts the convergence values to a control value is placed as a separate process. This process now has a fan-in and fan-out that is the same as the number of processor patches, which could be quite large if a large number of processor patches are being used. The control process needs to use all the residual streams being generated by all the patches. Similarly, all the selection processes on the patches have to use the control stream and therefore import it.

If the large fan-in/out becomes a problem it can be reduced by using a tree structure to merge or partition the data. In the case of the convergence folding, simply replacing the \texttt{arrayfold} call in the definition of \texttt{residuals} by \texttt{pararrayfold} would have the desired effect. Figure 6.32 shows a definition of \texttt{pararrayfold}. 
pararrayfold :: ([a] → [b] → [b]) → [b] → Array Int [a] → [b]
pararrayfold op base arr = paf 1
  where
    paf i = comp moreover
      Bundle [comp] And Arc comp next    if i ≤ b
        = base otherwise
    where
    comp = op (arr!i) next
    next = paf (i+1)

Figure 6.32: Parallel folding of arrays of streams

A broadcast NFO

The fan-out case is different. It is equivalent to a broadcast of the control stream to the processor patches. To reduce the fan-out, the broadcast could use a tree structure, similar to the fold tree built to gather the convergence data. A second approach could snake the control stream around the places where it is used. In order to avoid references to a stream consuming data from the original source, a copy of the stream needs to be made and referenced in place of the original stream. Copying a stream is simply achieved by mapping the identity function, id, over it.

The broadcast NFO can be used to reduce the fan-out needed, by strategically copying the stream and placing the copies on different processors to the original data. A sample implementation of the broadcast NFO is given in Figure 6.33. In general the broadcast NFO returns a number of copies of the stream that is being broadcast. The consumer of the stream must choose one of the copies to read instead of the original stream. To get the effect desired by the implementor of the broadcast NFO, each of the consumers must choose different copies of the input. The example implementation supplied snakes the stream around the consumers. With a more careful implementation one of the copy processes could be avoided by having the first consumer read the original stream.

With the broadcast NFO the program has to be updated to select one of the copies of the control stream. This change is shown in Figure 6.34. The
Chapter 6. Using Advanced Caliban

broadcasting using a snake

```haskell
bcast :: Stream → Int → Array Int Stream
bcast s n = barray
  where
    barray = mkarray (1,n) (prev barray)
    prev a i = copy moreover Bundle [copy]
      where
        copy = map id (a!(i-1))
```

Figure 6.33: Broadcasting using a snake

copies of the control stream are tupled with the patches so that the selection function can use the correct copy of the stream when selecting the result. Finally the network generated by the broadcast NFO is phased with the computation network. Because there should be a one-to-one mapping of copies of the control with the processor patches, the phasing algorithm will map the copy processes onto the consumer’s processor.

Appendix A shows the final version of the Jacobi Relaxation program. Figure 6.35 shows the final, unphased network that is generated by the program. It shows the three different network topologies. In the middle is the computation plane. Onto this plane are mapped the edge, residual and selection computations. These computations are all naturally structured like the plane and so map easily.

On top of this is built some form of combining network that takes the residual values and combines them to generate a control stream which is piped round to the final broadcast network. The broadcast network somehow distributes the control information to the processors on the computation plane. In the final pro-
convergenceJacobi :: (Patch Float) → Float → (Patch Float)
convergenceJacobi init limit = display result moreover finalAnnotation
  where
  :.
  (_, selected, _) = arraymap selectControl' cres
  cres = arrayzip tupleup dat bc
  bc = bcast (x×y) control
  ((x,y), dat, _) = res
  tupleup x y = (x,y)
  :.
  finalAnnotation = (((computationAnnot With ...) ...) )
  With controlAnnot
  controlAnnot = Annot bc

selectControl' :: (SPatch a, [Bool]) → [Body a]
selectControl' ((_, o, _), control) = isit control o
  where
  isit (False:rest) (d:dat) = isit rest dat
  isit (True:rest) (d:dat) = [d]

Figure 6.34: Using the broadcast mechanism to distribute control information
gram, the three networks are phased together to create a process network with the same shape as the patch plane — the convergence and broadcast pyramids in the diagram collapse onto the processor plane.

![Diagram of the process network](image)

**Figure 6.35**: The full, unphased process network generated by the Jacobi Relaxation program

### 6.2.5 Performance of Parallel Jacobi Relaxation

Due to problems with the Advance Caliban runtime system, it has been impossible to produce performance results for the Parallel Jacobi Relaxation application.

### 6.3 Advanced Caliban — An Analysis

In this section, some conclusions about the way Caliban programs are developed and constructed are presented. Some of the problems associated with Caliban
will also be described.

6.3.1 An Advanced Caliban overview

Caliban is an annotation language that describes how a functional program is partitioned into a set of threads and how those threads are scheduled onto a set of processors. Because the annotation language is built into the source language it can share and have full access to the data structures and results of the application that it is annotating.

Communication between threads is via streams. A stream is a list of normal form values. Each thread is a partition of the application program that produces a single value; Caliban requires that this value be a stream. Any thread that is placed on a different processor from another thread and requires that thread’s stream must import that stream. To import a stream its constituent values are communicated, one at a time, from the source processor to the destination processor. All the threads that require the stream in question that are located on the destination processor can share the communicated data. Threads that coexist on the same processor share all subexpressions through normal evaluation; this includes directly sharing streams between threads.

Between processors there is no communication other than via placed streams. This means that any subexpressions used by both producer and consumer that are on different processors is recomputed on each processor.

Caliban, where possible, attempts to maintain the semantics of the base language, in this case non-strict semantics. This means not evaluating expressions until there is a need for the value, and then only evaluating them as a far as needed. While this may be an advantage for programmers as it increases the likelihood of termination and allows more programs to the represented, it is a hindrance to parallel programming. If values are only computed at the point of need then no parallelism exists. Parallelism is only gained by evaluating expressions before they are needed so that the consumer of the value does not need to wait for the value before continuing.
To alleviate this, Caliban modifies the evaluation order for streams, where those streams are communicated between processors. Firstly, the values communicated are evaluated to normal form rather than weak-head normal form. Furthermore, the evaluation of streams is performed one element ahead of where the current demand from the consumer is. Once a compute-ahead value has been computed, it is sent to the consuming processor so that it is ready when needed. Streams that link threads on the same processor are evaluated lazily as normal.

6.3.2 Program Structure

To parallelise a program with Caliban, it has to be transformed into a set of stream-typed expressions that can then be placed by an annotation. This process divides the program into two levels. The top level is a coordination structure that only exists at compile time. For example, in the patched Jacobi Relaxation program the patch structure (the patch that contained the streamed patches and the streamed patches themselves) only exists at compile time. Once compiled, the structure is replaced by a mesh of dependent closures representing each thread. The structure disappears during Simplification as the annotation produced depends on information that is only obtainable by evaluating the top level patch and streamed patches. Specifically, the evaluation happens when each stream is reduced to WHNF. Once this has happened there is no useful remnant of the original patch structure. The second program level is those parts of the program that exist and are used at run time.

The process of restructuring the program to have a set of stream based expressions is similar to a data type transformation [53] in that the data of the program need to be transformed into a set of streams (lists). In the Jacobi Relaxation case, the transformation performed converted the patch into a patch of streamed patches. The streamed patch was introduced as a data structure that merely contained, in a useful form, the stream definitions for each logical processor patch. It was used as a coordination device and didn’t serve any useful computational role.
A similar transformation happens in the farm case. Initially the raytracing program consists of a list of jobs to be computed (rays entering the raytracer and leaving and impact data). This is transformed into a list of lists of jobs. Again, this is just a datatype transformation. The outer list structure is a coordination structure that is “evaluated away” at compile time to leave a set of threads that evaluate the inner lists of jobs.

In the case of the pipeline raytracer the transformation is more subtle. The database of objects is stored as a list. In the simple case, a thread is constructed for each object in the database. This makes the database list structure into a coordination structure, therefore it will disappear at compile time. With the pipeline version that uses groups of objects per thread, the transformation is similar to the farmed raytracer, except that the objects are partitioned rather than the rays.

From this it can be seen that parallelism is gained by migrating software into the coordination layer. The migration process may involve transformation or not depending on the structure of the application program. The main requirement of this transformation is to coerce the types of the expressions that will become processes. These expressions must become stream typed. This is not just because the backend type system requires it, it is because by casting the expressions as streams, a reduction order is defined for them. If arbitrary expressions could be placed as parallel processes then the notion of reduction order would be hard to define.

The stream is a useful data type as it has an obvious linear reduction order, that is to say, the elements of the stream are evaluated in order, each one fully evaluated before the next one is demanded. Other data types do not possess this simple semantics. Branching structures are difficult as a decision has to be made about which branch to evaluate first. For example, a tree can be evaluated depth-first or breadth-first. Neither order is obviously more useful than the other, therefore both orders need to be accommodated somehow.
Chapter 6. Using Advanced Caliban

Close language levels

Traditionally in configuration languages, there has been a clear separation between the configuration language and the application language. Each language worked in its own world, with a clearly defined interface between the two. Caliban, on the other hand, has a rich, bi-directional, interface. The annotation language is the same as the application language, they share all mechanisms and data. This is very good from the point of view of software reuse. Common annotation forms and algorithmic structures can be encoded as NFOs and stored in libraries for later reuse. NFOs have two forms, those that describe process network shapes, e.g. fan, and those that build process networks and the computations from the computational components, e.g. farm.

Not only can specific and planned reuse take place, but any higher order function, HOF, available to the base language can be used to build annotations. This opens up a second form of reuse, “code for free”. Two examples of this type of reuse have been seen in this thesis; firstly map has frequently been used to build annotations and processes and secondly lno was used in the Jacobi Iteration example from this chapter. This example is particularly interesting; just by changing the function parameter to lno, it can be made to build the computation or the annotation.

Finally, NFOs can be collected together and placed in the modules. In fact, at the source level, Caliban is implemented as a module that defines that annotation type and several support NFOs.

As well as the ease of reuse, the close relationship between the coordination and application language layers helps in developing applications. Annotations have full access to all the parts of the application that are needed, e.g. using information from the application to decide on the degree of parallelism to be extracted. Also, the annotation and algorithm could be affected by any static data that the program has, e.g. in the case of a raytracer, the scene database.

Not only that, but the coordination layer can affect the application layer. For example, an annotation and the application could agree on an algorithm and an
annotation at simplification time, in other words code for selecting the algorithm and building the annotation could be run at simplification time.

**Problems with the symbiotic relationship**

Unfortunately there are problems with the close relationship between the language levels. These relate to the simplifier’s lack of understanding of the program as it is presented. The simplifier aims to partially evaluate the program until the annotation is completely uncovered and all NFOs are expanded. The problems arise because the simplifier does not know what constitutes part of the annotation (what is in the coordination layer) and what is part of the application code. The solution adopted up until now is to evaluate all stream references in the annotation to WHNF. This has two desirable effects:

1. all vestiges of the annotation are guaranteed to have been removed, and
2. each stream is then easily identifiable (using the address of its closure) and two streams can be compared for identity.

This does lead to the problem of the simplifier over evaluating the application at compile time. The following simple program fragment shows the situation:

```haskell
s1 moreover pipe ls
where
ls = [s1, s2, s3]
s1 = f a
s2 = g b
...
```

The first stage of simplification leads to pipe being expanded to:

```
Bundle [head ls] And pipe (tail ls)
```

and some time later, when the bundled stream is evaluated to WHNF, to:

```
Bundle [s1] And ...
```
At this point the evaluation could stop as $s_1$ (the head of $ls$) is probably application level code. Without this knowledge though, the simplifier must proceed until $s_1$ has been evaluated to WHNF to allow matching of streams to take place in the network extraction phase. Evaluating the expression to WHNF could entail a lot of work that the programmer probably wished to be performed at runtime, not compile time (e.g. see page 212).

There are several solutions to this problem that could be adopted. The first is a programming level fix, which can be used straight away, but is ultimately unsatisfactory. It is to define a function $\text{red}$ (meaning “stop”) that takes a stream and conses an element on the front of it. A second function $\text{unred}$ has the opposite effect, i.e. removing the element. Each time a stream is defined that could suffer from the over evaluation problem, $\text{red}$ can be used to bring it to WHNF prematurely. All the consumers of this stream now need to apply $\text{unred}$ to it before using it.

This scheme is simple and requires no extra technology. Unfortunately it is not very clean. Not only does the definition of an affected stream have to be changed, but also each consumer must be adjusted. It also leads to the compute-ahead problem observed during the raytracer development.

The general solution to the problem is to get the simplifier to think that a stream expression is in WHNF without evaluating it to WHNF. The previous solution does this by adding a vacuous cons cell onto the front of the stream, thereby actually making the stream into a WHNF expression. The disadvantage that all the consumers of the stream have to know about it. A better solution would be to add an annotation to the simplifier which defines an expression to be already in WHNF. This way the simplifier would not evaluate the stream expression, also the stream value is not affected so consumers are unaware of any change and need not be touched.

A “meta-function” $\text{whnf}$ could then be defined that acts like the identity function, but stops the evaluation of its argument at compile time. Such a definition would look like: (note the underlining is the WHNF annotation)
\texttt{whnf \texttt{x = x}}

So the previous example could be handled as follows:

\begin{verbatim}
\begin{verbatim}
s_1 moreover pipe \texttt{ls
where
\texttt{ls = [s_1, s_2, s_3]}
s_1 = \texttt{whnf (f \ a)}
\end{verbatim}
\end{verbatim}

Note that \texttt{whnf} is applied to the definition of the stream, not its uses.

This annotation is the converse of the strict annotation that can be found in some sequential lazy functional programming languages [51]. Strict is usually defined as meta-function, like \texttt{whnf} above, that acts like the identity function but actually forces the full evaluation of the expression to normal form rather than just WHNF. It is used to improve efficiency.

**Demand profile problems**

The final problem to be covered in this section is that of the demand profile of stream expressions whilst the program is running. In a fully lazy sequential system, nothing is evaluated until it is consumed. To maintain the lazy flavour of Caliban, the consumer controls the demand of a stream. The system implements a compute ahead strategy, but ultimately the consumer controls the evaluation of a stream. The compute ahead is needed because laziness is inherently non-parallel. There are two approaches to dealing with this, the first is to analyse the program to determine which subexpressions will eventually be evaluated by the program anyhow and allow them to be evaluated early. The second approach is to evaluate subexpressions speculatively in the hope that some will be be used and therefore a saving in time will have been made.

Caliban falls between the two approaches. The assumption with Caliban is that some prefix of a communicated stream is going to be used by the computation of that stream’s consumer. Therefore it is reasonably safe to evaluate slightly
ahead of the demand placed on the stream by the consumer. The more compute-ahead used, the more risk that the computation could be wasted. Also, there is more risk that the evaluator on the producer processor will run out of memory.

The effects of the consumer based demand profile have already been seen with the farm NFO defined in Chapter 2 (page 72). If the incoming work is divided into contiguous blocks then the computation of each block is sequentialised. This is because after the system has demanded the first item of each slave’s output the only point of demand left is the consumer of the farm. It will demand the second item of the farm’s output, which will come from the first slave (as did the first item). Subsequent items will also be demands from the first slave, leaving the remaining slaves idle. This will continue until all the items from the first slave have been exhausted when the demand will move to the second slave, and so on. It must be ensured that the demand is maintained evenly across the set of slave processors. A round-robin style of work allocation ensures this.

Another example of unexpected demand profile occurred during the development of the Jacobi Iteration codes. The pre-convergence testing version of the program returned the results of a single chosen patch after each time step. This meant that instead of the process network starting work on all the patches at once the demand for computation spreads across the network, starting from the processor that is delivering the results, like a wavefront. It is like the reverse of a pipeline fill, the furthest processor from the demand source starts last as the demand takes $n$-hops to reach it. This then leads to delays in delivering results to neighbouring processors which means that the delays are propagated to the processor delivering the result.

By adding the convergence and selection computations each cell in the mesh has the demand applied for the following generation at the same time. This has a strictifying effect on the computation as a whole. Now, all the cells compute each generation at the same time, rather than the computation rippling across the plane.
Static process network restriction

In its current form Caliban is restricted to building static process networks at compile time. This is because of the way in which Caliban introduces parallelism. An annotation can affect any stream that is in its scope. The compiler has full access to the whole annotation and therefore knows the full extent of the parallelisation, i.e. which streams are placed and how they are bundled. At run time, the information is not known. With a naive dynamic Caliban a stream could be used at one point, and then later in the computation an annotation could be uncovered that places that stream. What then becomes of the stream is unclear. Should it be migrated to a parallel processor (a messy and expensive operation) or should its annotation be ignored as part of the stream has already been evaluated. The static Caliban would have uncovered this annotation at compile time and therefore known that the stream should be placed.

One way to achieve a form of dynamic Caliban would be through compilation techniques that package up subnetworks such that they are instantiated as a whole when a certain condition is met — effectively allowing the construction of phases to be dynamic. Much work is needed to examine the feasibility of this approach, which is beyond the scope of this thesis.

Program development

One final, and very important aspect of Caliban is the ease with which programs can be developed and prototyped. Caliban programs can be developed in any Haskell-based functional language system, such as Gofer by Mark Jones [51]. With a sequential functional language, the declarative correctness of the program can be confirmed before it is run on the parallel machine. Caliban syntax can be simulated on a sequential system by just providing the Caliban module that defines the annotation and stream types.
6.4 Conclusions

This chapter has demonstrated application development for Caliban. Two applications were developed and some performance figures presented. In the raytracer case, it was shown firstly that as the number of slaves/pipeline stages increases, so does the performance. But there comes a cutoff point where further increases lead to less of an improvement, even to a degradation in performance. Also, it was shown that increased work packet size (increased granularity) improves performance, again up to a certain point, when degree of parallelism issues take over and reduce performance. Finally, extra compute ahead is shown to make a big difference to performance, especially for farm style computations.

Stepwise development was demonstrated with the Jacobi Iteration example. Each feature was added to the program, finishing with a program that uses most of the elements of Caliban introduced in the previous chapter.

The chapter finished with a discussion on the strengths and weaknesses of Advanced Caliban. This discussion highlighted some areas for further work which will be discussed in the next chapter.
Chapter 7

Conclusions

This chapter starts with a summary of the work presented in previous chapters. It continues with a discussion of related work. Next, the contribution and results of the thesis are reviewed, followed by some ideas for future work. Finally, there are a few concluding words.

7.1 Summary of thesis

In Chapter 2 the basis of this thesis was presented: parallel programming software and methodologies are lagging behind parallel hardware. We can build machines that are capable through parallelism of amazing performance, but it is still very difficult to write programs for them. Even moderate parallelism cannot automatically be extracted from any but the simplest of programs. There is also no global model of parallel computing, making portability a difficult and relevant problem.

The literature survey showed the following more specific points:

- Parallel computing offers a way forward in the search for higher performance computing.

- Distributed memory architecture parallel computers provide the platform of choice to achieve the parallelism required.
Communications locality is of far less importance than it used to be, in terms of performance, due to cut-through routing techniques.

Spark based systems do not provide a high level view of parallel programming.

Skeletons are too restrictive in their approach to parallel programming.

Configuration languages don’t offer enough interaction between configuration and application.

Chapter 2 went on to present Basic Caliban, a simple language (originally introduced in [53]) that allows a programmer to partition a lazy functional program into concurrent processes. Caliban is an annotation language that is represented in the base functional language. Inter-process communication is provided by streams (head strict, lazy lists). Each process in the parallel program is actually a stream producing subexpression of the original program. The annotation names which subexpressions are wanted as parallel tasks and allows the programmer to indicate any anticipated dependencies between two tasks.

The “Process Placement Rule” defines what is computed by each parallel task. It says that all non-placed subexpressions of a stream are evaluated on the local processor. This allows the programmer to explicitly express replication of computation if this improves the performance.

Common annotation structures (and therefore common process network structures) can be captured as NFOs. These allow libraries of expertise to be built and therefore aids software reuse: a major prerequisite for software engineering methodologies. Programs containing applications of NFOs are partially evaluated at compile time to expose the full structure of the process network that they encode.

Finally the chapter presented a simple raytracing program, initially written as a farm computation. Using step-by-step transformation techniques, this program was transformed into an equivalent pipeline program. This showed the power of purely declarative programming languages.
The chapter showed that Caliban allows the programmer to take a more global view of the parallel computation represented by the program. This is true in two respects; firstly, Caliban can express the communication between parallel processes and secondly, NFOs allow groups of processes to be dealt with en masse.

Chapter 3 went on to present an implementation of the ideas presented in Chapter 2. The implementation was divided into three major sections. The simplification stage partially evaluates the program to reveal the full structure of the process network represented by the provided annotation. The evaluator uses term graph rewriting as its basis to ensure that all sharing is maintained during the evaluation process. The simplifier also has the ability to evaluate programs in the presence of partial information. To notate this, a set of formal schemas were presented.

Network extraction, the second major section of the implementation, transforms an annotated program into a set of mutually recursive functions, each one representing the computation of a process in the intended process network. The network extractor implements the process placement rule, acting as a static scheduler for the system. Some optimisations to the results of the extractor were presented.

The final section of the implementation was the runtime system. The interface to the runtime system consists of a single parallelism providing function called \texttt{procnet}. It allows a process network to be described, where each node is defined using a function from streams to stream and the arcs are described by a wiring diagram, connecting processes’ inputs and outputs. Each process has only one output stream, which each consumer being supplied by a separate lightweight task on the producer processor. One element compute ahead is used to allow parallel processing.

The following chapter looked at the restrictions of Basic Caliban and proposed a set of extensions that create a new language: Advanced Caliban. The extensions were as follows:
• Multiple output processes

One of the major restrictions of Basic Caliban was that a process could only produce a single stream. This meant that a large number of programs could not be encoded efficiently. Without multiple output processes replication of computation had to be used to perform such as tasks as stream splitting (dividing a stream into several substreams, as used in, for example, a farm computation). The solution was to introduce the Bundle annotation that placed two computations on the same processor. Bundle subsumes Node as a singleton Bundle is equivalent to a Node assertion. Several examples motivating the extension were presented.

Bundle enables better scheduling.

• Multiple moreover clauses

Having just a single moreover clause meant that any annotation formed inside a computation NFO had to be plumbed out explicitly. This reduced the elegance of the language and meant that NFOs became less general. The introduction of multiple moreover allows programs to be much cleaner and allows NFOs to be more general. It also allows normal HOFs to be used to build Caliban computations rather than having to copy the HOF and add annotation plumbing to it. The other major advantage was to improve encapsulation. An NFO builds and deploys its own annotation. The calling code for an NFO needn’t know about its placement decisions.

Multiple moreover allow more software reuse.

• Phasing

The main thrust behind this feature is to aid the programmer in producing programs that better utilise the target machine. With collapses two process networks on top of each other. If the two subnetworks form distinct phases of the computation then a better processor utilisation will be achieved by reusing the processors for each phase. The feature can also be used elsewhere to schedule two dissimilar networks together, especially where there is no obvious or simple mapping. The algorithms suggested
map communicating nodes together where possible to reduce communication overheads. This is useful as it allows layered networks to be easily joined, e.g. in the Jacobi Iteration example in chapter 6 where the main computation and residual computation networks are fused. This joining of producer and consumer is not always desirable as it can lead to a loss of parallelism.

Phasing improves process network efficiency.

The goals of phasing and multiple moreover clauses are not totally concordant. Thus a meta-function, Annot, had to be introduced to allow sub process networks to be phased rather than joined in the normal manner.

Chapter 5 went on to present an implementation of Advanced Caliban. The new implementation was based on the original front end and a new back end (FCG) and target machine (AP1000). The simplification schemas from chapter 3 were extended to cope with multiple moreover clauses. This involved considerable reworking of the original schemas to allow annotations to be automatically plumbed back to the top level. The result is a searching simplifier. It uses annotations in the program graph to direct it towards any nested annotations and an annotation environment to keep track of annotations already found. To implement the Annot meta-function required the annotation environment to be extended to a double indirection environment to avoid reuse of annotations that have been explicitly used by Annot. Secondly, each placed stream is evaluated to WHNF during a second phase of the algorithm. This separation of evaluation times is needed to ensure that nested annotations are not lost.

Network extraction was completely changed for the new implementation. A much improved algorithm was developed that suffered from none of the problems of the original version, and had many advantages. The new algorithm is based on using an indirection mechanism to control where each placed stream is computed. If a placed stream is reduced, first the indirection mechanism decides if the reducer is on the correct processor for the stream, if so the closure for the stream is returned and evaluation continues as normal. If the reducer is not
on the right processor, then a readstream closure is returned and the stream is fetched from a remote processor. This scheme not only simplifies the network extraction process, but also allows multiple streams to cohabit a processor easily.

Phasing is implemented as another compiler phase after network extraction. Its job is to join two process networks, scheduling them onto the same set of processors. The smaller network is mapped on top of the larger, each node of the smaller being bundled with one in the larger. It is assumed that each node in a subnetwork is computationally balanced, therefore the scheduling problem is simplified. A heuristic algorithm is presented that attempts to reduce communication and reuse already existing routes between processors. By reusing links the fan-in/out is kept to a minimum wherever possible.

A runtime system based on the FCG functional language abstract machine and the Fujitsu AP1000 distributed memory parallel computer was described. A non-preemptive version is described as it mostly implements the intended semantics and is simpler and more efficient. Extensions for preemption are also described.

Chapter 6 presents two medium sized applications. Several versions of a simple raytracer are shown. Using performance graphs and execution-time graphs the effects of degree of parallelism and grain size are demonstrated. Both farm and pipeline versions are presented, each with different processor numbers, problems sizes and work packet granularities. In the case of the farm version, collector saturation is demonstrated.

Jacobi Relaxation is also presented as a complete implementation cycle from the problem specification to implementation. Powerful use of the ApplyLNO HOF is demonstrated as it describes the shape and style of the computation. Each part of the computation is added incrementally and its process network is phased with the existing network.

Chapter 6 concludes with an analysis of Advanced Caliban. It starts with a summary of the language in its current state. The notion that Caliban consists of two interwoven language levels (coordination and application levels) is developed. Parallelism is achieved by transforming code from the application level into the
coordination level. This is followed by a discussion of some of the weaknesses of the Caliban system relating to the simplification and runtime demand profiles. Together with these issues, several solutions are presented.

7.2 Related Work

This section compares this work with other work in the same sphere. Several other language systems are examined, as well as work on applications.

7.2.1 HPF

HPF’s approach to parallel programming is via partitioning of data. Once the partition has been set the computation can follow its data using the Owner Computes rule.

Caliban also partitions data, however in a functional language, the notions of data and computation are very similar. Data is not acted upon by a program (that would require mutability), rather it is copied and transformed. When data is partitioned, computation automatically follows. To define the placement of subexpressions Caliban uses the Process Placement Rule.

One difference between Caliban and HPF can be seen in how they provide the parallelism. HPF is a data parallel language based on arrays. It achieves parallelism by partitioning arrays of data and using multiple processors to compute the results of the partitions. Caliban is essentially a process parallel language based on streams. Parallelism is achieved by breaking the computation down into stream producing units and linking them together. HPF partitions the container whereas Caliban uses the container to link the parallel processes, where the container is an array or a stream depending on the language. This difference is due to the different paradigms (data/process parallel) that the two languages inhabit. With a data parallel language the data item (e.g. an array) is partitioned, but it is acted upon as a whole. In the process parallel language the data is partitioned (e.g. into lists of items) and each partition is explicitly acted upon.
Both languages provide static (compile time) partitioning and scheduling. The partition of a program is fixed by the programmer at compile time, although HPF allows programmed repartitioning though the \texttt{REDISTRIBUTE} directive and the use of dummy arguments in subprograms. HPF provides a purely regular set of partitioning capabilities whereas Caliban allows arbitrary partitions of the program to be constructed.

\subsection{Skeletons}

Caliban NFOs can provide much of the functionality of a skeleton system:

- an NFO can capture an algorithm and implement it in parallel,
- NFOs (subnetworks) can be composed together with their interfaces matching and
- NFOs have declarative as well as operational readings

What is missing is the ability to choose an implementation of an NFO based on the environment in which it is being called. This could be added to Caliban by writing a layer of software (in Caliban) that does the selection of NFO implementation according to the environment. This code would be executed by the simplifier and would result in the tuned parallel program given the parameters of the target machine (cf P$^3$L).

Skeleton systems provide a fixed repertoire of algorithms from which the programmer can chose. Caliban allows the system designer to provide a library of NFOs to implement common algorithms and the programmer to add to this library should there not be an NFO or set of NFOs that meet the requirements of the particular program being developed.

\subsection{Coordination/Configuration languages}

Caliban is very similar to configuration/coordination languages such as Delirium, PCN and Darwin. It can be made to mirror a configuration programming language directly by defining components as functions that take a group of input
streams and produce a group of output streams ([Stream] → [Stream]). Once set up like this, where clauses can be used to link the components together and annotations used to describe what type of parallelism is required. The similarities even extend to being able to build new components out of combinations of other components.

There is a very strong similarity between Caliban and Delirium. Both are functional, both are embedding (the coordination code sits around the application code rather than inside it) and both use data dependencies to express coordination. Delirium uses transforms to map data explicitly from one stage of the computation to the next, whereas Caliban uses the natural data dependencies present in the program. For both systems, migrating code in and out of the coordination layer allows the programmer to change the degree of parallelism available in an application. Both systems express the application’s parallelism using the base language’s facilities, however Caliban interprets these at compile time whereas Delirium leaves this until run time.

PCN’s template mechanism is very similar to Caliban’s NFOs. Templates are higher order functions that capture a parallel programming structure. The user passes functions and data into the template to define its exact behaviour. Again, much of the scheduling work in PCN is carried out at run time, allowing for dynamic configurations. Caliban does all this work at compile time, allowing only static configurations.

Using programmed parallelism Caliban, like PCN and Delirium, can control large sets of coordination items.

### 7.2.4 Jacobi iteration

An interesting paper by Hartel and Vree discusses the construction of an iterative Jacobi solver [42]. He starts by specifying the problem in a Prolog-like, logic language and then uses a transformation to convert the logic specification into an non-backtracking solution that can be directly implemented using a functional language. It is interesting to note that the solution gained is very similar to the
implementation found in Chapter 6 of this thesis. The salient point being that both implementations use cyclic functional expressions (resulting in a process network with cycles for the Caliban program) to implement the iterative nature of the solver.

Caliban programming involves the use of cycles wherever iteration would have taken place in an imperative system. If these are not immediately obvious to the programmer then perhaps Hartel’s method of specification and transformation could be used to construct programs.

7.3 Contribution of this Thesis

This thesis has helped to map out another part of the parallel programming spectrum. It has introduced the notion of simplification — the process of partially evaluating a program to extract information out of it to be used later in the compilation process. Initially, when there was just a single moreover clause, simplification was simple. With the introduction of multiple moreover clauses the process became much more complicated. The complete description of the process is contained in Chapter 5.

There are two implementations of network extraction. The original method was similar, in principle, to lambda-lifting. The later and considerably more flexible method was based on variable renaming and environment manipulation. Both methods form part of the original material presented.

Phasing is introduced as a simple and flexible method of processor reuse. A simple algorithm for matching processors between two subnetworks has been presented.

A limited implementation of Advanced Caliban has been produced. Using this, two sample applications and some performance results have been presented. The case studies are used as the basis for an analysis of the Caliban which examines the strengths and weaknesses of the language.
7.4 Future Work

There are many topics that could be researched further. This thesis has documented a complete cycle in the development of Caliban. Another cycle could be performed, this would entail a complete implementation of Advanced Caliban and more application case studies.

It has been shown that Caliban suffers from a lack of precise evaluation control, making it difficult to specify certain programs. Enhancement in this area would be most useful. There are two sides to this work. Firstly at simplification time, better evaluation control is needed to avoid evaluating certain expressions at compile time. Solutions to this problem, such as the WHNF annotation, have already been discussed in Chapter 6. Secondly, runtime control of evaluation would be highly desirable. Again, Chapter 6 showed the problems caused by limited compute-ahead and demand driven evaluation. Solutions could involve allowing the specification of the compute-ahead of an arc (its parallel slackness or closeness of coupling) or changing the evaluation order of specific arcs by making them data driven rather than demand driven.

Developing a performance model for Caliban would be useful for programmers to plan their approach to parallelising an application using Caliban. There are difficulties with this work as Caliban does not place single work packages, rather it places processes that deliver a series of sub-results over time. The exact timing of evaluation and therefore the computational load of a Caliban process depends on the timing of the demand for that processes results. Also, it is possible that work is duplicated amongst a set of processors. This will affect the performance model considerably. Some form of heuristic model based on computation loads could be a useful tool for programmers.

An extension of performance models would be to develop scheduling algorithms to map Caliban process networks to physical processors. At the moment a one to one mapping of logical nodes to physical nodes is used. By placing another mapping/scheduling layer into the system, decomposition and scheduling become decoupled problems. Because of the static nature of Caliban it is
more amenable to automatic scheduling than other languages. Of course, a programmed approach would be still be possible.

At present Caliban uses streams to enable inter-process communication. However, there is nothing special in the use of streams as process connectors other than the fact that they closely model arcs in a process network. Other data types such as trees (discussed earlier) or bags could be experimented with. Bags would provide a useful mechanism for introducing a limited degree of non-determinism into a program. This could usefully be used to implement dynamic farm structures for example.

One final area that needs to be investigated further is the ability of Caliban to express at least some limited forms of dynamic process networks. Presently Caliban is inherently limited to expressing static networks as the whole network needs to be expanded at compile time. Approaches could be investigated where an annotation describes an overall shape of a network without explicitly instantiating the network at compile time. This would be useful for applications that use dynamic structures such as divide and conquer or data based decomposition techniques (where the data is not known until run time).

### 7.5 Final Thoughts

This thesis is about Caliban and its development. What started with a simple language used to elaborate ideas on functional programming techniques has developed into a powerful parallel programming language with novel features. This thesis has shown that Caliban provides is a viable approach to parallel programming for certain problems (those with static communication requirements).

However, Caliban is not without its problems. Section 6.3 presented an analysis of Advanced Caliban that included a discussion of some of its failings. Further work is needed to address these problems and produce solutions.

Along the way novel implementation techniques for the language have been developed. These include a particularly novel partial evaluation algorithm to uncover and normalise a program’s annotation. Although the language places
unusual demands on the compiler, it is still possible to implement it effectively.
Appendix A

Jacobi Relaxation in Advanced Caliban

This is the final version of the Jacobi Relaxation program, written in Advanced Caliban. The program is based on a patched data representation:

- Simple element based patch
  type Patch a = ((Int, Int), Body a, (Edge a, Edge a, Edge a, Edge a))

- Folded patch
  type FPatch a = Patch a

- Streamed patch
  type SPatch a = ((Int, Int), [Body a], ([Edge a], [Edge a], [Edge a], [Edge a]))

- Folded streamed patch
  type SFPatch a = SPatch a

- A 2D array of elements
  type Body a = Array (Int,Int) a

- A 1D array of elements
  type Edge a = Array Int a

- An LNO function type
  type LnoFunc a b = a → a → a → a → a → b

The top level program is called parallelJacobi:

```
parallelJacobi :: (Patch Float) → Float → (Patch Float)
```
parallelJacobi init limit
  = display selected moreover finalAnnotation
    where
    - The main cyclic computation
    res = join init comp
    comp = lno streamf res
    ((x,y), dat, _) = res

    - The residual and selection mechanism
    (_ selected, _) = arraymap selectControl’ cres
    cres = arrayzip tupleup dat bc
    bc = bcast (x*y) control
    tupleup x y = (x,y)
    control = map (limit >) residuals
    residuals = arrayfold (map2 max) (repeat 0.0) convergences
    (_ convergences, _) = lno (patchConv percChange max 0.0) res

    - The annotation
    finalAnnotation = (((computationAnnot With selectAnnot)
      With convergenceAnnot) With controlAnnot)
    computationAnnot = arrayfold (And) NoPlace places
    (_ places, _) = lno placePatch res
    controlAnnot = Annot bc
    selectAnnot = arrayfold (And) NoPlace (arraymap node selected)
    convergenceAnnot = arrayfold (And) NoPlace (arraymap node convergences)
    node a = Bundle [a]

join :: (Patch a) → (Patch [a]) → (Patch [a])
join ((x,y), initData, (n, s, e, w)) (res, computedData, res)
  = ((x,y), streamData, (streamN, streamS, streamE, streamW))
    where
    streamData = mkarray2d (x,y) (joinEl initData computedData)
    joinEl i c (x, y) = i!(x,y) : c!(x,y)
    streamN = mkarray x (streamLt n)
    streamS = mkarray x (streamLt s)
    streamE = mkarray y (streamLt e)
    streamW = mkarray y (streamLt w)
    streamLt a i = repeat (a!)

placePatch :: LnoFunc (Patch (SPatch Float)) (Patch Placement)
placePatch (n, s, e, w) = fan data [n,s,e,w] And Bundle [data,on,os,oe,ow]
There are three sections to this function definition. The top section defines the basic LNO computation. The middle section defines the selection and convergence computations. The final section defines the annotation used to place the computations.

The selection and convergence computations are supported by the following functions:

- Select a generation of the body of a streamed patch as the return result.
  selectControl' :: (SPatch a, [Bool]) → [Body a]
  selectControl' ((_, o, _), control) = isit control o
  where
  isit (False:rest) (d:dat) = isit rest dat
  isit (True:rest) (d:dat) = [d]

- Calculate the convergence values for a streamed patch.
  patchConv :: (a → a → b) → (b → b → b) → b → LnoFunc (SPatch a) [b]
  patchConv f j b ds, o, _ = nmap (convPatch f j b ds) o

- Calculate the convergence values for a pair of patch bodies.
  convPatch :: (a → a → b) → (b → b → b) → b → (Int, Int) → (Body a) → (Body a) → b
  convPatch f j b ds d1 d2 = arrayfold j b convs
  where
  convs = mkarray2d ds convEl
  convEl i = f (d1!i) (d2!i)

- Calculate the percentage change between two numbers.
  percChange :: Float → Float → Float
  percChange a b = \[\frac{100(a-b)}{a}\]

- Apply a function to neighbouring items of a list.
  nmap :: (a → a → b) → [a] → [b]
  nmap f [] = []
  nmap f [a] = []
  nmap f (a:b:rest) = (f a b) : (nmap f (b:rest))

The LNO functions are defined as follows:
- A streamed patch function to calculate the average across of the plain.

```haskell
streamf :: LnoFunc (SFPatch Float) (SFPatch Float)
streamf n s e w o = foldSLno average newo
    where
    newo = unfold n s e w o
```

```haskell
unfold :: LnoFunc (SFPatch Float) (SPatch Float)
unfold (n s e w o) = foldSLno average new o
    where
    new o = unfold n s e w o
```

- Element based LNO.

```haskell
lno :: (LnoFunc a b) -> (Patch a) -> (Patch b)
lno f p = (bds, data, edges)
    where
    data = mkarray2d bds (buildApp f p)
    (bds, _edges) = p
```

- Apply an LNO function to its arguments from the provided patch.

```haskell
buildApp :: (LnoFunc a b) -> (Patch a) -> (Int, Int) -> b
buildApp f p (x, y) = f (pick p x (y-1)) (pick p x (y+1))
    where
    (pick p x y) = p
```

- Select an element from the local data or from one of the boundary edges.

```haskell
pick :: (Patch a) -> Int -> Int -> a
pick ((xt, yt), arr, (n,s,e,w)) x y
    = arr !! (x,y) if inrange (xt, yt) x y
    = e!y if x > xt
    = s!x if y > yt
    = w!y if x < 1
    = n!x otherwise
    where
    inrange (xt, yt) x y = ((x≤xt) & (x≥1)) & ((y≤yt) & (y≥1))
```

- Stream based LNO.

```haskell
slno :: (LnoFunc a b) -> (SPatch a) -> (SPatch b)
slno f input = (bds, data, edges)
    where
    data = map (buildBody bds f) (spToPatches input)
    (bds, data, edges) = p
```
Apply an LNO function to a patch, returning a new body.

buildBody :: (Int, Int) → (LnoFunc a b) → (Patch a) → (Body b)
buildBody bds f init = mkarray2d bds (buildApp f init)

Convert a streamed patch into a list of patches.

spToPatches :: (SPatch Float) → [Patch Float]
spToPatches (bds, bs, (ns, ss, es, ws)) = patches
  where
    patches = map5 (buildP bds) bs ns ss es ws
    buildP bds b ns e w = (bds, b, (n,s,e,w))

Perform a streamed LNO, returning a folded patch.

foldSLno :: (LnoFunc a b) → (SPatch a) → (SFPatch b)
foldSLno f input = ((x,y), data, (n,s,e,w))
  where
    ((x,y), data, _) = slno f input
    n = map (mkarray (1,x) (extract 1 1 0)) data
    s = map (mkarray (1,x) (extract 1 y 0)) data
    e = map (mkarray (1,y) (extract x 1 0)) data
    w = map (mkarray (1,y) (extract 1 1 0)) data

Extract the i'th value of a range of values from a 2D array

extract :: (Int) → (Body a) → (Array a)
ex = data !! ((1+i) × xskip), yinit+((i-1) × yskip))

The types of array functions:

- 1D array subscripting

  (Array a) ! x → a

- 2D array subscripting

  (Array a) !! (x,y) → a

- Map for arrays

  arraymap :: (a → b) → (Array a) → (Array b)

- Fold for arrays

  arrayfold :: (a → b → b) → b → (Array a) → b
Bibliography


