A Compiler and Runtime for identifying Implicit Parallelism in Dataflow Programming

Gardiner, Thomas Andrew

August 24, 2012

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2012
Abstract

The Dataflow programming paradigm was introduced to allow the programmer to define their applications in the form of data-dependency graphs (or workflows). This representation brought with it a rich set of opportunities for application-level optimisations that are not available to languages that use the traditional imperative programming model.

The aim of this dissertation was to show that describing applications in this way can not only allow a compiler to effortlessly produce a highly scalable, parallelised executable, but also that this method of representing a program is intuitive and leads to a flexible and extensible codebase.

The approach taken is to design and implement a domain-specific language (called Flow) for defining coarse-grain dataflow programs. A custom compiler and run-time engine were built that automatically extract implicit parallelism from the application; scaling to large multi-core processor hardware.

A Geophysical science application for oil exploration was ported to the Flow language, and shown to have good strong-scaling properties.
## Contents

1 Introduction ......................................................... 1
   1.1 Proposed solution ............................................. 2
      1.1.1 Graphs ..................................................... 2
      1.1.2 Flow ..................................................... 4
   1.2 Implicit parallelism ........................................... 5
   1.3 Minimal Path Detection ......................................... 6
   1.4 Features ....................................................... 6
   1.5 Overview ...................................................... 7

2 Background and Literature Review ................................. 9
   2.1 Von Neumann .................................................. 9
   2.2 Dataflow .................................................... 10
   2.3 Dataflow Architectures ....................................... 10
   2.4 Coarse-grain Dataflow ...................................... 11
   2.5 Domain Specific Language ................................... 12
   2.6 Summary ..................................................... 13

3 Flow .............................................................. 14
   3.1 Fundamental concepts ......................................... 14
   3.2 Dependency graph ............................................. 16
   3.3 Instantiation .................................................. 18
   3.4 Parallel loops ................................................ 20
   3.5 Cyclic dependencies .......................................... 21
   3.6 Control flow branching ...................................... 23
   3.7 Design justifications ......................................... 24
   3.8 Summary ..................................................... 27

4 Abstract Syntax Tree ................................................ 29
   4.1 Syntax and Semantics .......................................... 29
   4.2 Lexer .......................................................... 30
   4.3 Parser ........................................................ 31
   4.4 AST .......................................................... 32
   4.5 Summary ..................................................... 33

5 Flow Integrated Development Environment ........................ 35
## 5.1 Syntax Highlighting

5.2 Incremental Compilation

5.3 Error messages

5.4 Summary

### 6 Internal Representation

6.1 Minimal Path Discovery

6.1.1 Compile-time optimisation

6.1.2 Runtime optimisation

6.1.3 MPD Summary

6.2 Code Generation

6.3 Summary

### 7 Run-time Engine

7.1 The API

7.2 Run-time Engine Design

7.2.1 Work stealing

7.2.2 Scheduling

7.2.3 Distributed-memory load balancing strategies

7.2.4 Proposed algorithm

7.3 Summary

### 8 Results and Evaluation

8.1 Efficiently extracting scalable parallelism

8.1.1 Experimental setup

8.1.2 Scheduler overhead

8.1.3 Scalability

8.2 Implementing real-life problems

8.3 Summary

### 9 Conclusions

9.1 What was implemented?

9.2 Future work

### A Using Flow in practice

### B Flow grammar
List of Tables

8.1 Serial impl vs equivalent Flow program . . . . . . . . . . . . . . . . . . 64
8.2 Task timings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
## List of Figures

1.1 A simple serial workflow ........................................ 3
1.2 Graphs A has two instances of graph B, which itself has an instance of graph C .................................................. 3
1.3 The use of inner graphs is equivalent to embedding them directly . . 4
1.4 Outline of implementation ........................................ 7
3.1 Flow ........................................................................ 14
3.2 A simple dependency graph in Flow .............................. 17
3.3 A do-until loop example ............................................ 22
4.1 AST ........................................................................ 29
4.2 Lexer and Parser in action .......................................... 30
4.3 Assignment represented as an AST ................................ 33
5.1 Shows syntax highlighting of Flow sourcecode in the Eclipse IDE 36
5.2 Shows a syntax error underlined and indicated in the Problems window 37
5.3 Shows a semantics error underlined and indicated in the Problems window 38
6.1 IR and Code Gen ..................................................... 40
6.2 Minimum set of tasks needed to compute output hook y are highlighted 41
6.3 Task M is redundant ................................................ 42
6.4 In the left instance of graph B, not all its output hooks are used ...... 43
6.5 The output hooks are annotated with the names of the hooks they help compute .................................................. 43
6.6 One of B’s inputs is a constant ..................................... 46
7.1 Code Gen and Runtime Engine .................................... 49
7.2 The main classes that represent the Flow API .................. 51
7.3 Each thread has access to its own local queue and others’ shared queues 54
7.4 Schedulers and workers communicate using two task queues .... 55
7.5 A thread’s subtasks will always eventually come back to it for aggregation. Each vertical line represents an executing thread and the rectangles are tasks. ........................................ 56
7.6 Workload grows/shrinks from/to bottom left corner of distributed-memory mesh topology .......................... 57
7.7 Single scheduler thread per CPU node communicates with neighbouring schedulers .................................................. 58

8.1 A string of tasks is executed in serial for every integer passed to the input hook. In this example the input hook consists of 3 integers, but this graph is dynamically built at run-time, so could have an arbitrary number of concurrently executing strings of tasks .......................... 62

8.2 Shows what proportion of runtime the worker spends idle, for varying task lengths .................................................. 64

8.3 Shows how completion time varies, with problem-size-per-worker kept constant ..................................................... 65

8.4 Shows how time spent busy varies, with total-problem-size kept constant .................................................. 66

8.5 Shows how completion time varies, with total-problem-size kept constant .................................................. 67

8.6 Shows actual vs ideal completion time, given varying number of worker threads .................................................. 70

9.1 Summary of implementation .................................................. 73
Acknowledgements

I would like to give a special thank you to my supervisor, Ally Hume, for taking me on as his MSc dissertation student; and giving me the freedom to pursue my own project idea. His guidance and advice throughout this project has been invaluable.

I would also like to thank Magnus Hagdorn for allowing me to use his Geophysical science oil exploration code as a proof-of-concept example for my project. And I would like to thank all the staff and students at EPCC who have helped and supported me during this time.
Chapter 1

Introduction

In a world where keeping up with Moore’s law is becoming increasingly challenging [32] and multi-core processors have become the norm [1], adding parallelism to applications should no longer be just an after-thought. Without parallelism you are not only missing out on the opportunity for significant speedup of your application, but are wasting precious hardware resources by letting CPU cores sit idle.

The problem is that, due to their inherent non-determinism, writing correct parallel programs is significantly more difficult than writing serial programs. Multiple threads are executing concurrently, only synchronising their behaviour when explicitly asked to. This makes testing for correctness more time-consuming and challenging, as the number of different ways in which threads may interact on different runs is potentially unbounded. It also makes debugging more difficult, as bugs are not consistently reproducible.

New parallel languages, to assist in writing parallel programs, do exist, such as Chapel [2], X10 [3] and other PGAS languages [4], however these are either still in development or are yet to gain widespread popularity [5]. A few of the parallel languages that have taken off in the mainstream [6] include Clojure [7], Go [8] and Erlang [9]. These languages provide a number of useful constructs for expressing parallelism, such as immutable objects for thread safety and explicit message-passing for communication. The problem is that they still require the programmer to convert their mindset from thinking in serial, to thinking in parallel.

The aim of this dissertation project is to show that programs can be described (i.e. programmed) in such a way that the programmer does not need to think in parallel; but large amounts of scalable parallelism can still be implied. By using implicit rather than explicit parallelism the programmer can focus on writing correct serial programs, and let the compiler worry about multi-threading, scalability and thread-safety. More specifically, the aim of the project is to show that describing programs in the form of a data-dependency graph (or workflow) is not only intuitive, but also presents the characteristics we just described (i.e. it exposes implicit parallelism). The project will do this by designing, implementing and using a domain specific language to describe ap-
applications as a dependency graph, and implementing a compiler and run-time engine to execute these applications in parallel.

1.1 Proposed solution

The proposed solution is to allow the programmer to use their existing C/C++/Fortran subroutines (or functions) and use a domain-specific language (DSL) to connect them together to form an application. This DSL will be designed specifically for this purpose, and we shall call it Flow.

1.1.1 Graphs

We will refer to the subroutines/functions that Flow connects together as tasks. Tasks are the smallest unit of execution for the Flow language and Flow does not try to extract parallelism at a lower level than a task. Flow is a DSL, but could also accurately be described as a "Coordination Language" [30]. The purpose of Flow is to coordinate and manage when and where each task in the application is executed.

Tasks (i.e. subroutines/functions written by the programmer) in Flow conceptually have inputs and outputs, and these we shall refer to as hooks. Each hook will have a datatype (e.g. int, bool, string, MyCustomType, etc.) and once an input or output hook has been assigned a value it must not be changed (Flow assumes they are immutable). Flow guarantees that all input hooks will have a value assigned to them before the task is executed. And a task must promise to assign values to all its output hooks by the time it has finished.

Flow connects all its tasks together by representing them as a data-dependency graph. It is called a "data-dependency" graph because the graph describes which data a task depends on (i.e. data it needs before it can be executed). More specifically, it describes which task(s) create this data and which output hook(s) of those tasks hold the data. For brevity we shall just refer to these data-dependency graphs as graphs.

Figure 1.1 illustrates a simple example of a Flow graph. It shows that, rather than directly connecting tasks together, they are connected via their hooks. Task A has an output hook, x, which is connected to task B’s input hook, a. This tells Flow that once task A has completed, it should assign the value of hook x to the value of hook a (B.a := A.x). As hook a is the only input dependency of task B, Flow can then execute task B. If there were more dependencies it would have to wait until all input values are available.

Flow effectively boils down to connecting tasks together in this way, but to allow more complex relationships to be described there are a number of other language constructs that provide additional semantics (such as loops). These will be described in detail.
later in chapter 3, but for now there is just one more construct worth introducing. This construct is *inner graphs*.

Inner graphs allow a programmer to break up their main graph into reusable sub-graphs. There is effectively no difference between a graph and an inner graph, it is just that when one graph is referenced from inside another, we call it an inner graph. Figure 1.2 illustrates the use of an inner graph B, used twice inside graph A. Graph B itself uses graph C as an inner graph. Note that this is entirely equivalent to creating a single graph like the one shown in Figure 1.3. Note: D, E and F are tasks (not inner graphs).

With the introduction of inner graphs, Flow now still just connects hooks together, but the hooks can belong to either tasks or graphs. To help in explanations later we shall use the term *node* as an umbrella term to mean anything that has input and output hooks (i.e. tasks and graphs).

In summary, graphs describe the data-dependency between nodes by connecting their
Figure 1.3: The use of inner graphs is equivalent to embedding them directly.

flow and output hooks together. Nodes can either be tasks (i.e. subroutines written in C++/C/Fortran) or references to other graphs (inner graphs). Inner graphs are just syntactic sugar for embedding graphs within graphs. And finally, in order to execute a node you must first provide values for all its inputs, and when it completes it will have assigned values to all its outputs.

1.1.2 Flow

Flow is the name given to the language, but we shall also use this name to refer to the Flow compiler, as well as the Flow runtime engine. The Flow compiler compiles Flow sourcecode into C++ sourcecode. This generated sourcecode, together with a purpose-built C++ runtime engine library (discussed in chapter 7), will make up the majority of the application logic. The only significant part that the programmer has to do (after writing the application in Flow) is implement the actual tasks1 (as subroutines written in C++/C/Fortran). Once these are implemented, any general-purpose C++ compiler can be used to compile the sourcecode into an application executable2.

The Flow runtime is just a name for the logic that controls (at run-time) how the application is executed (i.e. which tasks are executed and where). This logic is all part of the compiled executable, but we make this distinction, between compiler and runtime, in order to allow us to distinguish the decisions that can be made statically (from the Flow sourcecode alone) from the decisions that must be made dynamically (i.e. they require run-time information).

Importantly, the decisions on how many tasks to create and execute is all decided at run-time, as it completely depends on the volume of the input data. And similarly, with where to run a task (i.e. which CPU core or node to run it on), this is also decided.

1or use existing code
2if using Fortran routines a Fortran linker is also needed
at run-time, as it depends on the hardware resources available and on how busy these resources are.

As an aside: To run the application on separate distributed-memory CPU nodes\(^3\), the application executable should be started as a separate process on each CPU node. An environment variable can be used to inform each process of the host and port of the root node, and this root node has the job of making all the nodes aware of each other.

### 1.2 Implicit parallelism

Parallelism is the technique of having different computations within an application run concurrently. The purpose of parallelism is to reduce latency in an application by trying to make full use of the hardware resources available. This could be in the form of instruction-level parallelism where instructions are pipelined so they use different hardware functional units (e.g. FPU, ALU, etc.) concurrently. Alternatively it could be application-level parallelism, using real independent threads of execution. This could be for the purpose of making use of multiple processing cores, or it could simple be to provide work for a processor to do while it is waiting for high-latency I/O operations to complete.

The type of parallelism we will be discussing is application-level parallelism; the type that has separate threads of execution. Within application-level parallelism, this could either be explicit or implicit. Explicit parallelism is where the application programmer explicitly writes code to create and start new threads of execution (or creates a thread-pool and submits tasks to it). The alternative is implicit parallelism, where the programmer simply specifies the behaviour of the program, and it is the job of the compiler and/or run-time to identify opportunities for parallelism and to schedule new threads.

One of the aims of this dissertation is to write a compiler and run-time engine to identify implicit parallelism in Flow graphs. This implicit parallelism will appear in two forms: the first is through independent tasks and the second is through parallel loops. Independent tasks are defined as any tasks that are not directly or indirectly dependent on each other. Dependencies are explicitly defined by the programmer in Flow, so two tasks can be considered independent if there is no direct path from one task to the other, by following data-dependencies in one direction. Parallel loops, on the other hand, are defined using the Flow `foreach` construct (described in chapter 3). This works very much like a normal for-loop in other languages, but every iteration is enforced to be independent of every other (by the Flow language), so by definition can be computed in parallel.

\(^3\)Throughout this dissertation the terms CPU node and CPU core will be used regularly. It is assumed that a distributed-memory computer is made up of multiple CPU nodes (connected via an interconnect) and that each CPU node will consist of one or more CPU cores. Each core is effectively its own processor, with its own scheduling unit and other functional units.
1.3 Minimal Path Detection

As just mentioned, tasks are dependent on each other if there is a direct dependency path between them. What this means is that, for a task $T$, you can very easily identify all the tasks that $T$ is dependent on, by traversing the dependency graph upwards (i.e. following the input dependencies). The same can be done for a top-level output hook (i.e. to find all the tasks that must be computed in order to evaluate a given output hook). In Figure 1.3 only task E needs to be computed to evaluate A’s right-hand output hook.

Minimal Path Detection (MPD) is the name we have given to exactly this technique. MPD aims to find the minimum set of tasks that must be computed in order to evaluate a given set of output hooks. The advantage of this optimisation is that programmers can re-use graphs within graphs in their Flow programs, and if they do not use all the outputs of a particular graph, they do not have to incur the processing cost of calculating any redundant (unnecessary) tasks within that inner graph.

This provides a large benefit over imperative programming languages, such as C or Fortran, where if you call a library subroutine, but are only interested in one of the outputs, you still incur the full overhead of calling that subroutine. One could argue that, in that case, that library subroutine should have been designed better, and perhaps should have been split into multiple subroutines; but it is not always easy for a library writer to guess what all the use-cases of their library will be. By using MPD, Flow programmers can write fully-featured graphs, and then let the compiler and run-time worry about optimising it for different use-cases.

1.4 Features

While only a subset of the features listed below have been implemented in this dissertation project; the potential features of the Flow coordination language are enumerated here, as a basis for motivating the longer term aims of this research, and encouraging future work:

- automatically scale to available hardware: the number of independent tasks in Flow grows with the size of the input data, and because Flow instantiates these tasks as self-contained executables, they can easily be spread out across the hardware resources available

- programmers can use the best tool for the job: Flow could potentially handle tasks from any programming language, so a single application could use Fortran for one task, Python for another and Matlab for a third.

- fault tolerance: every task is a self-contained executable, so Flow could easily support fault tolerance, where if a CPU node fails, the tasks from that node could be rescheduled elsewhere
• bugs quickly reproducible: tasks are self-contained, so if a task throws an exception, Flow could dump the set of inputs for that task to file. The programmer could then reproduce the error using those inputs, without having to re-run the whole application

• writing tests becomes trivial: for the same reason as the previous point; tasks can be tested in isolation

• automatic memory management: in languages like C and C++, allocation and deallocation of dynamic memory is an important, and often error-prone, job; but because Flow has all the information it needs to identify when a piece of data is no longer needed (i.e. when all the tasks that depended on it have completed), it could manage all this automatically.

1.5 Overview

Figure 1.4 provides a clear overview of what will be achieved in this dissertation. We will come back to this breakdown as we progress through each chapter.

The next few paragraphs will give a short taster of what each chapter will cover. One will notice that each chapter relates specifically to a square in figure 1.4.

*Flow* is the DSL that was designed specifically for the purpose of describing applications as graphs. Its design will be discussed in chapter 3.

The *AST* is an Abstract Syntax Tree that is an in-memory tree-based representation of Flow sourcecode. ASTs are useful for identifying syntax and semantics errors in an input program. ASTs and the syntax and semantics of the Flow grammar will be discussed in chapter 4.

The *IR* is the Flow Internal Representation that is also an in-memory representation of Flow sourcecode. The advantage of this representation is that it allows implicit parallelism, and other forms of optimisation, to easily be identified. If there are no errors in the AST, the compiler converts it into the Flow IR. The design of this IR will be discussed in chapter 6.
The *Code Generation* stage is where the IR is traversed by the compiler and C++ is generated. This code provides the logic for the application at run-time; indicating the dependencies between different tasks. The code generation step will be discussed partly in chapter 6 and partly in chapter 7.

The *run-time engine* is the C++ library that was designed and built for the purpose of executing a Flow graph as efficiently as possible. Its aim is to make best use of the hardware resources available, extracting as much parallelism as is possible given the provided input data and the graph structure. The design of the run-time engine will also be discussed in chapter 7.

However, before all this, chapter 2 will outline previous research conducted in this field; discussing how and why others have tackled similar problems.
Chapter 2

Background and Literature Review

The concept of Dataflow was first proposed by Jack Dennis in 1970 [27] and has been developed in many different ways since then. It was pursued as an innovative alternative to the classic Von Neumann hardware model and promised the potential of “unrivalled implicit parallelism” [49].

2.1 Von Neumann

The Von Neumann model [47] is the conceptual model that we think of whenever we are writing programs in imperative programming languages such as C or Fortran. Its two key features are the program counter and the global updateable memory store. The program counter moves forward, executing one instruction at a time, and the global memory store is a central place that all instructions have access to. As a programmer this model makes it very easy to enforce correctness, as the instructions just need to be placed in the right order, and sharing data is simple as it is globally available.

The problem is that some instructions take longer than others; for example, loading data from memory can take hundreds of clock cycles [10], while arithmetic may just use one. This means that the processor is left idle for hundreds of clock cycles waiting for the result to come back. Instruction level parallelism techniques [22], such as out-of-order execution [39] (re-orders instructions to try to fill the gap), can be used to reduce this problem, but finding hundreds of independent instructions for every memory load is not always possible.

An approach to filling these gaps, that works very well [44], is the use of explicit parallelism. This is where a number of different threads are explicitly started by the programmer’s code. This provides a lot more independent instructions for the processor to schedule, making better use of available hardware resources. The problem is that issues such as thread-safety and deadlock make writing programs like this significantly more complicated than writing serial programs. Ideally these complications would be handled by the compiler.
2.2 Dataflow

The dataflow paradigm was introduced for exactly this reason. The programmer only has to describe the data-dependencies between nodes (e.g. instructions) and the compiler does the rest. Johnston et al. [35] provide a good explanation of how dataflow works: dataflow programs are described using acyclic directed graphs, with nodes connected by arcs. Data flows along these directional arcs, with the arcs working like unbounded first-in-first-out queues. When a node has data on its input arcs, it will, at some undefined time, take the first tokens from its input arcs, perform its operation and place new tokens on its output arcs. Nodes are functional, meaning that operations have no side-effects (explained further in chapter 3.

By representing programs in this way, instructions can be executed as soon as they have tokens on their input arcs, rather than having to wait for a program counter to reach them. This provides a lot of opportunities for implicit parallelism [36]. The absence of side-effects is also an advantage [17] as it makes it easier to control data locality (making sure data is physically close to the thread that needs it). The main disadvantage, however, is that the token matching process (identifying when a node has input tokens for processing) is unfortunately more expensive than just moving an instruction counter [35].

2.3 Dataflow Architectures

Silc et al. published a thorough discussion of how dataflow models and architectures have developed since their introduction in 1970 [43]. There has been a large variety of implementations, but they can be categorised into a few main models. We will discuss these here.

The first model, the one proposed by Dennis [28], is the “Single-token-per-arc” dataflow model. This model represented programs using a static graph of nodes, connected together via arcs. It allowed just one token per arc, and a node (i.e instruction) was considered executable when all its input arcs have a token on them. This model has simple and straightforward semantics, but the single-token-per-arc restriction meant there was limited parallelism to be gained. Parallelism was present in the sense that different independent nodes could be computed concurrently, but if multiple data items needed to be processed by the same node, they would have to queue up to be processed as a pipeline. Nonetheless, a number of implementations were constructed [28] [24] [25].

To overcome the limitations of the single-token-per-arc model, Watson and Gurd [48] proposed the tagged-token dataflow model. This model added a tag (or colour) to each token which allowed groups of tokens to be differentiated from other groups of tokens. This meant that, instead of just holding a single token, arcs could hold a whole bag of tokens, ready to be processed. The execution model changes so that a node is now considered executable when tokens with the same tag are present on all input arcs. This
allows nodes with different sets of tokens to be processed concurrently. A number of architectures were also constructed for this [48] [19] [51].

The main problem with the tagged-token model was how to efficiently store and access the potentially unbounded bags of tokens. The approach used by most modern dataflow architectures [43] is the one suggested by Papdopoulos [40], whereby a separate memory frame is held for each tag type and these are accessed directly using a known offset from a frame pointer. This does however bound the amount of available concurrency to a constant maximum number of frames.

Another issue with the dataflow model as described is that it makes poor use of register and cache-reuse. The model tries to make as many nodes available for concurrent processing as possible, but there is nothing to enforce that a thread should prioritise executing nodes, for which it already has the tokens cached, over any other node. Ideally a thread should avoid having to access main memory as much as possible. A solution to this, implemented by a number of architectures [41] [31], is to identify, at run-time, subgraphs of a dataflow program that exhibit "a low degree of parallelism" [43], and transform these into sequential threads. This allows a particular thread to follow a set of tokens all the way through a subgraph, without having to read and write these tokens back and forth from main memory.

2.4 Coarse-grain Dataflow

After many iterations and many implementation of the dataflow architecture it was inevitably concluded that, while large amounts of parallelism did provide a lot of potential for performance, the parallelism achieved through dataflow was just too fine-grained [46]. The overhead of the dataflow token matching process, and the advances in instruction-level parallelism (ILP) for Von Neumann models, meant that dataflow was not competitive in performance [35]. The solution to this was coarse-grain dataflow, i.e. "dataflow/Von Neumann hybrid" [34]. This used dataflow to describe the high-level application, but instead of a single instructions at each node, each node held a whole sequence of instructions (e.g. a subroutine). This moved the responsibility of choosing the level of granularity from the compiler over to the programmer. Having larger granularity meant benefiting from both the ILP techniques of Von Neumann, and the implicit parallelism of dataflow. And importantly, the parallelism was coarse-grained enough that the overhead of token matching became insignificant.

With fine-grain dataflow being left behind, more recent efforts no longer focus on implementing dataflow at the hardware-level, but instead use it for identifying high levels of coarse-grain parallelism at the software level. Two recent efforts include Flow-based programming [38] and PaPy [23] and these will be discussed here.

Flow-based programming (FBP) [38] works as a graph of asynchronously communicating processes (like the Actor model [18]). Every node in the graph is a process and these nodes send streams of tokens to each other via defined channels. There is no concept of
an executable node, as all nodes are executing continuously, processing tokens as they arrive. Tokens can be grouped together by using begin and end tokens in the stream. This is a radically different approach to dataflow compared to those we have seen so far; delegating the problem of token matching to the programmer. It can perform well if your application resembles a graph of pipelines, but does not load-balance well if all nodes are not equally busy all of the time. Implementations are available for writing the nodes in C, C# or Java [37].

PaPy [23] (i.e. Parallel pipelines in Python) slightly more closely resembles the dataflow model we have seen implemented in hardware, but it does so in software. It is implemented as a parallel library written in the Python programming language. The programmer uses library calls to create and connect nodes together, where nodes represent tasks written in Python. A worker-pool (i.e. thread pool) design pattern is used to process the nodes, and parallelism is available both at the node and data-item level (i.e. can process multiple data items on each graph node concurrently). Nodes are submitted to the worker-pool as soon as all their inputs are available.

The major difference between these software implementations of dataflow and the hardware implementations from the past is that FBP and PaPy use library calls, in existing programming languages, to build up their graphs, while the hardware implementation typically used a Domain Specific Language (DSL) to describe theirs [49]. We will argue, and aim to show in this dissertation, that the use of a DSL for representing the graph provides significant advantages over the former.

### 2.5 Domain Specific Language

There are many recent implementations of new programming languages for defining parallel programs [2] [3] [4] [7] [8] [9]. The problem, however, is that there are already a vast amount of invaluable software libraries already written in existing serial programming languages, and having to rewrite them all would be time-consuming and error-prone. Gelernter et al. [30] argue that a better approach to parallel programming is to implement applications using a language-independent coordination notation (i.e. "a coordination language"). This does not replace your currently implemented libraries, but instead coordinates the communication between them and controls how and when they are used. Such a DSL would provide a consistent notation for describing the high-level layout of an application, even if the underlying procedural language used to implement the nodes changes.

It is such a coordination language that this dissertation project has implemented, providing a DSL that makes expressing a dataflow graph clear and concise. Van Deursen et al. [45] describe the advantages and disadvantages of implementing a DSL. They explain that, compared to general-purpose programming languages, a DSL is "a much better solution than a generic approach, but for a smaller set of problems". The small set of problems Flow aims to solve are problems that can easily be expressed in the
form of a workflow (data-dependency graph).

Another major advantage of a DSL is that the problem can be expressed in a much higher level of abstraction, meaning that domain experts could potentially understand, validate, modify and even develop applications in such a language [45], without needing to be confident software developers. The disadvantage, however, is that getting the scope right can be quite difficult: having a high level of abstraction, without losing too much expressivity. This last problem of getting the DSL right will be addressed in the DSL design chapter next.

2.6 Summary

The concept of data-flow programming, which provides the inspiration for Flow, has been around since the early 1970s. Many hardware implementations of this paradigm were designed and built due to the potential for high degrees of parallelism that it promised. In the end it was found that fine-grained parallelism came with too high a cost in the form of token-matching overhead. To overcome this, coarse-grain overhead was introduced. This has been explored in many forms, but never as a DSL coordination language implemented in software. It is this approach that this dissertation has pursued.
Chapter 3

Flow

In this section we will discuss the design decisions that were made for the Domain Specific Language (DSL) implemented in this project. We will start by describing the language as it is in its final revision, but will finish by explaining why it was designed in this way and the reasons for some of the changes that were made along the way. The DSL is called Flow.

Figure 3.1 shows which stage of the implementation we are discussing.

3.1 Fundamental concepts

Imperative programming languages, such as C, Fortran or Java, provide step-by-step instructions on how the processor should compute the desired result. The statements are executed in order, and each statement will update the state of the running program.

In contrast, Declarative programming languages describe what computations need to be performed, rather than how they should be performed. This provides freedom to the compiler and/or runtime to decide how and when to compute the necessary computations. It is important to realise that Flow is a declarative DSL, meaning that the order in which statements are specified in its sourcecode do not in any way enforce the order in
which they must be executed at runtime.

Another important concept to understand is referential transparency. A subroutine is referentially transparent when:

- it has no side-effects, and;
- it is pure.

It is important to understand this concept, as Flow assumes that all tasks are referentially transparent, and could not safely perform its optimisations without this assumption.

Having no side-effects means that no matter how many times a subroutine is called, if its output(s) (e.g. return value) are not used, the state of the program will not change (i.e. the subroutine call could be removed and the program behaviour would be unaffected). If a subroutine meets the requirement of having no side-effects, its only purpose is to calculate its output(s) from the provided inputs (the values given to its parameters).

One ambiguity that is worth clarifying is regarding the use of the term output(s) (rather than result or return-value) in the previous paragraph. In Flow every external subroutine can have multiple inputs and multiple outputs. In order to achieve this, when calling C++/C/Fortran subroutines: references (or pointers) to the variables that will hold the output values are passed into the subroutines as input parameters (rather than relying on the return value). This means the subroutine can assign values directly to their output variables, rather than trying to pass multiple values out through the single return value.

To meet the other requirements of referential transparency, a subroutine is pure if, given a set of inputs, the result is always the same. This implies that once we have called such a subroutine with a given set of inputs, we could, in theory, cache the result, and next time we are presented with the same set of inputs, we would not need to call the subroutine, but could instead just look up and retrieve the result from the cache.

Flow is a DSL that allows you to call external subroutines written in C++, C and Fortran. Flow assumes that any such subroutines (i.e. tasks) meet the requirements of referential transparency (as described above). This allows the Flow compiler to reuse results where needed and even skip subroutine calls if it finds them unnecessary.

A simple example that demonstrates these points is shown in Listing 3.1. The mySubroutine C function does not meet the requirements of referential transparency:

- Its first line is fine as it uses an input to calculate the value for an output.
- The second line, however, is illegal as it has side-effects: the input value is modified. This could pose a problem if that input is reused elsewhere.
- The third line is also illegal as its use of a mutable global variable means the function is not pure.

This subroutine is not pure as, given the same inputs (i.e. same values for input1 and input2), it could produce different output values because output2 depends on a global
variable that may change between invocations. This could be fixed by passing the `mutableGlobalVariable` in as an input instead.

Listing 3.1: mySubroutine is not referentially transparent

```c
void mySubroutine ( int *input1 , float *input2 ,
        int *output1 , float *output2 )
{
    // legal
    *output1 = *input1 * 2;
    // illegal – has side effects
    *input1 = 8;
    // illegal – is not pure
    *output2 = *input2 + mutableGlobalVariable;
}

int main ( )
{
    int input1 = 5, output1 ;
    float input2 = 7.0 , output2 ;
    mySubroutine (&input1 , &input2 , &output1 , &output2 );
    mutableGlobalVariable ++;
    printf ( "out1:%d\nout2:%f\n" , output1 , output2 );
    return 0 ;
}
```

### 3.2 Dependency graph

We have so far discussed how Flow is a declarative language and that it can call external subroutines (written in C++, C and Fortran). We know that the order of statements in Flow do not enforce an order on when a statement is executed, so what does? The answer is: the dependency graph.

Figure 3.2 shows a simple example of a dependency graph that can be described in Flow. Absolutely everything in Flow is a node, so even the more complicated constructs can be thought of in this way. The diagram shows five nodes (A-E), each of which have one or two inputs (a and b) and one or two outputs (x and y). The dependency graph shows that in order to calculate a value for A’s output, x, we need to compute node E. E in turn depends on two inputs, a and b, that we have specified come from the single outputs from nodes C and D respectively. These depend on B, which in turn depends on A’s input value, a.

Listing 3.2 shows how you might describe the same graph in the Flow language. This code snippet shows the `definition` of "A". The first line shows that "A" is a graph that
Figure 3.2: A simple dependency graph in Flow
takes a single integer (called $a$) as input, and has a single floating point number (called $x$) as output. Parameters declared inside parentheses ('(' and ')') are inputs and those declared inside square brackets ('[ ' and ' ]') are outputs.

Listing 3.2: A simple graph described using Flow

```plaintext
graph A (a : int) [x : float] {
    node nodeB = B(a = a);
    node nodeC = C(a = nodeB.x);
    node nodeD = D(a = nodeB.y);
    node nodeE = E(a = nodeC.x, b = nodeD.y);
    x = e.x;
}
```

The body of the graph is shown inside curly braces ('{ ' and ' }') and describes how the input(s) are connected to the output(s). They are connected via a chain of instantiated nodes.

At this point it is worth clarifying the terminology we will be using. Everything is a node and a graph connects instances of nodes together. We will start by looking at two types of node: graphs and tasks; but there is also a foreach node, a do-while node and a switch node (discussed shortly). All node types can be instantiated within a graph, and instances of nodes are then connected together. Graphs and tasks must be defined and given a name before they can be instantiated. Listing 3.2 shows the definition of graph "A", and the instantiation of a "B" node, a "C" node, a "D" node and an "E" node. Elsewhere these nodes have been defined (i.e. given definitions), but without looking at their definitions we cannot tell whether they are graphs or tasks (because the syntax for instantiating graphs and tasks is the same). The instances of the nodes have been given names: nodeB, nodeC, nodeD and nodeE, respectively.

Listing 3.3: Two instances of node A

```plaintext
node myNode = A(a = 5);
node yourNode = A(a = 7);
myResult = myNode.x;
yourResult = yourNode.x;
```

Listing 3.3 shows an example of how we could have multiple instances of node A in a graph; instantiated with different input values. These would be using the same node definition, but the outputs may be different as the input values are not the same.

### 3.3 Instantiation

When you instantiate a node you are creating a new instance of that node, providing it with its inputs, and asking it to evaluate all its outputs. How it does this depends on the

---

18
type of node it is: e.g. if it is a task it will execute the underlying subroutine, while if it
is a graph it will traverse the graph, instantiating nodes as it encounters them.

Instantiating a node means that its inputs have been given values and that, once instan-
tiated, all outputs will have values. Importantly, once instantiated this node cannot have
its values changed and can be treated just like a simple object with immutable prop-
erties (i.e. fields). In Listing 3.3, \textit{myNode} could be treated just like an instance of an
immutable object that has two properties, \textit{a} and \textit{x}; even though, behind the scenes there
has actually been a lot of processing required to evaluate the value of \textit{x}.

Note that this represents a critical contrast to how traditional Dataflow languages treat
nodes. In Dataflow languages nodes are connected in the same manner as here, but the
behaviour is more like a pipeline [49] or factory assembly-line. In traditional dataflow
you pass an ordered list of tokens into a node and it will process them, spitting out new
tokens at the other end which are in turn sent on to other nodes. This is not how Flow
works; if you have multiple items that need processing, you instantiate multiple nodes
to handle them.

\begin{verbatim}
Listing 3.4: Shows Person node being used as an object
graph C (a : string) [x : Person] 
{ 
    node person = Person(name = a);
    x = person;
}
\end{verbatim}

Just to give a concrete example of how nodes can be treated as objects, Listing 3.4
shows how an instantiated node can be used as an output from a graph. The Person
node has a single input, \textit{name}, which is a string of characters. It is not shown here, but
the Person node may have outputs such as address or date of birth, which it may be
populating from a database or other source, based on the provided name. Alternatively
it may have no outputs, in which case no processing is required when instantiating the
Person, and really it is just more like a Struct in C or C++ (because you have to set all
its fields yourself).

\begin{verbatim}
Listing 3.5: Defining two subroutines
native B (a:int) [x:string, y:float] : c;
native D (x:float) [x:int] : fortran;
\end{verbatim}

We have seen examples of how nodes can be defined as graphs, but we have also men-
tioned that they can be defined as tasks (i.e. subroutines written in other languages).
Listing 3.5 shows an example of how "B" and "D" might be defined if they were tasks,
rather than graphs. The \textit{native} keyword tells the Flow compiler that they are imple-
mented natively in another programming language, rather than as a graph. The next
part of the definition is the same as for a graph, with "B" having an input integer and
two outputs; one of which is a string of characters and the other a floating point number.
The last part of the prototype, just before the semi-colon, tells Flow which language the developer has implemented them in. Here, node "B" will be implemented as a C function and node "D" will be implemented as a Fortran subroutine. This means that the Flow compiler will automatically generate these interfaces in the appropriate language, to be implemented by the developer (i.e. they will be placed in a generated header file).

So how does this define the order in which statements should be executed, or more accurately, instantiated? The answer is that the only constraint is that no node can be instantiated before all its dependencies have been instantiated. So given the graph from Listing 3.2, nodeB must be instantiated before nodeC and nodeD, and nodeE cannot be instantiated until after nodeC and nodeD. In the source code nodeC appears before nodeD, but there is no reason why nodeD could not be instantiated before, or in parallel with nodeC (as they are not dependent on each other).

### 3.4 Parallel loops

So far we have discussed how everything in Flow is a node; have seen examples of how graphs and tasks can be defined; and have seen how nodes can be instantiated and linked together to form a dependency graph. Importantly, every node is referentially transparent and all instantiated nodes are immutable. This makes the program very easy to reason over, as there are no side-effects to worry about and each variable will only ever have one value; but what happens when we want to aggregate multiple results?

Listing 3.6: Summing an array of integers in C

```c
int i, total = 0;
for (i = 0; i < NO_OF_ITEMS; ++i)
{
    total += items[i];
}
```

This is best demonstrated with an example. Say we have a list of numbers that we want to sum. In an imperative language such as C we might write a loop like that shown in Listing 3.6 where each of the integers in the `items` array are added to the `total`. There are two ways in which this is different in Flow: firstly, Flow would like to expose the parallelism available in this loop (especially if there was some processing occurring inside each iteration), and secondly, nodes are not mutable in Flow, so the `total` could not be a node.

Listing 3.7 shows how you would define such a loop in Flow. It is shown within a graph too, just to demonstrate how a list of items is declared in Flow: a * before the type (i.e. *int) means this is a list of the declared type. A list of lists of floats would be declared as **float.

---

1This syntax was chosen arbitrarily, and in retrospect could have been different (and should perhaps be changed in the future)
As always, everything in Flow is a node, and thus, the foreach loop is an instance of a foreach node. It has a single implicit input (the list of items) and in this example it has a single output (the total). Each iteration can be performed concurrently and the instance (called loop) is fully instantiated once every iteration has completed. Once instantiated, the node is immutable and its output (the total) will hold the sum of all the items. The final line assigns the total to the output of the Sum graph.

There are currently three aggregate operators supported in Flow, as demonstrated in Listing 3.8. The first adds all the results together, the second multiplies them, and the third appends them all in a list. The first two can only be applied to integers and floating point numbers, while the third must be applied to a list. The potential for other operators, as well as custom operators, is vast, but for the sake of simplicity Flow is currently limited to just these three.

### 3.5 Cyclic dependencies

The foreach loop allows you to process multiple independent items in parallel, however sometimes you do not know how many iterations you need. Dependency graphs are directional, but there are times when you want to express a circular dependency, for example when you want to repeat an operation until a chosen condition is met.

Figure 3.3 shows the control flow of such a loop. It takes some initial value, \( a \), and performs an update step (e.g. tries to come up with a solution) until the result is within...
some acceptable tolerance. Once that is the case, we take that value and output it as value $x$. A typical example is any kind of guided search algorithm where we need to perform multiple iterations until a solution is reached.

Listing 3.9: Do-until loop in Flow

```plaintext
node cycle = do [ init x:int = a, noOfIterations:int ]
{
    node result = UpdateStep(value = x);
    next x = result.updatedValue;
    noOfIterations += 1;
}
until (result.isWithinTolerance);
```

An example of how this can be implemented in Flow is shown in Listing 3.9. If you ignore the outputs declared after the `do` keyword, you will notice that this is just a do-until loop which is effectively a do-while loop with the conditional check negated. Do-until just mean that the loop condition decides when to exit the loop, rather than when to repeat the loop. Do-until was implemented, rather than do-while, as the majority of use-cases that were found involved a stopping condition (rather than a loop condition). Unlike the foreach loop, these iterations are not performed in parallel. There may be parallelism exposed within the loop body, but the construct itself does not imply opportunities for parallelism. It purely exists for control flow.

Note that the stopping condition here is the `isWithinTolerance` output of the UpdateStep node. This output must be of type `bool` which represents a boolean (true/false) value.
Also notice that \textit{result} was declared inside the loop body but is used in the stopping condition. Flow has the same scoping rules as most other languages: specifying that a variable is only visible within the block in which it was declared; meaning that \textit{result} is not visible outside of this loop. The slight exception, which may cause confusion, is that the stopping condition, despite being outside of the curly braces, is considered part of the same scope as the loop body. This is necessary due to the immutability constraints mentioned earlier. The stopping condition must be able to change, as otherwise we would loop forever.

Looking back at the outputs declared on the do-until loop shown in Listing 3.9, we can see that there are two: an integer, \textit{x}, that has been initialised to the value of \textit{a}, and another integer, \textit{noOfIterations}. Like the foreach loop, the do-until loop is also represented as a node, meaning that once the loop condition is satisfied, \textit{cycle} will be an immutable instance with the aforementioned two outputs.

The \textit{noOfIterations} output has been included here purely to demonstrate that aggregate outputs, like those seen on the foreach loop, can be added to do-until loops in exactly the same way. We are just incrementing this output every iteration so that it contains a count of how many iterations were required to find our solution.

The \textit{x} output, on the other hand, is declared differently to how we have seen before. It is prefixed by the \textit{init} keyword to signify that \textit{a} is just its "initial" value and that we will be updating it each iteration. Unlike an aggregate output, we are not combining values to build the result, but are instead directly replacing it each time. To make it even clearer that we are updating a value (which normally isn’t allowed in the immutable world of Flow), the \textit{next} keyword is used to prefix the line where \textit{x} is updated. This was introduced to make it clear that this update to \textit{x} will not apply until the next iteration. So even if \textit{x} was used on the last line of the loop, it would still only have its current value (not its next value). Note: notice that \textit{init} outputs can be read as well as written inside a loop, whereas aggregate outputs cannot.

\section{Control flow branching}

The final construct in the Flow DSL is the switch statement node. This works much as you would expect, allowing you to take different code paths based on the value of an input. It differs slightly from traditional switch statements, in that you can’t drop through from one case to another, so only one case is chosen; and this \textit{case} then has the task of assigning values to the node’s output(s).

This is most easily understood with an example, as shown in Listing 3.10. This shows the instantiation of a switch node that has a single output: \textit{age}. There are two cases defined: one for when the person’s name is "John" and one for when his name is "Peter", and a third \textit{default} case catches all other values of \textit{person.name}. Every single \textit{case} block must assign values to all outputs, so if \textit{person.name} has value "Mary", \textit{switchNode} will be instantiated as a node whose output, \textit{age}, will have value 0.
Listing 3.10: Switch statement in Flow

```java
node switchNode = switch (person.name) [age : int]
{
    "John": { age = 26; }
    "Peter": { age = 32; }
    default: { age = 0; }
} ;
personsAge = switchNode.age;
```

Listing 3.11: The equivalent of a Flow switch in C

```c
int age;
if (equals(person.name, "John"))
{
    age = 26;
}
else if (equals(person.name, "Peter"))
{
    age = 32;
}
else
{
    age = 0;
}
personsAge = age;
```

Just to clarify it even further, the equivalent implementation of Listing 3.10, written in C, is shown in Listing 3.11. Notice that it is actually the equivalent to a set of if-else blocks, rather than the C switch statement. This is firstly because the logic more closely matches that of an if-else statement, but also because C switch statements do not support switching on strings. The Flow switch statement supports strings, integers and boolean values. Future work could potentially include ranges of values and lists of choices. Ranges would be particularly useful to allow handling of floating point numbers.

### 3.7 Design justifications

Designing the Flow DSL took many iterations, including trial and error by trying to represent real-life problem domains in this language. It is, however, beyond the scope of this dissertation to go over all the iterations that were performed. Instead the discussion shall focus on the reasons why this final design was the one chosen.

The real-life problem domain that initially inspired the design of Flow was a simple credit derivatives pricing model [26], for calculating a set of metrics for a set of trades. The details are unimportant for this discussion, but the reason why this model was
interesting was that the choice of which tasks needed computing strongly depended on
the input data: Each trade is for a particular portfolio and a portfolio represents a set of
credit curves\(^2\). A trade also has a currency, which implies a particular yield curve\(^3\). On
top of this, each trade has a product type and this, combined with the metric, decides
which calculator to use. The calculator then takes the inputs (the credit curves and yield
curve) and outputs a value for the metric.

Listing 3.12: An example Credit Derivatives application written in an earlier version of
Flow

```java
/* a simplified high level Credit derivatives model */
graph CreditModel ( books:*Book, metrics:*Metric )
  [ bookMetrics:*BookMetric ]
  {
    foreach (book in books) [ bookMetrics:*BookMetric ]
      {
        foreach (trade in book.trades) [bookSum:float]
          {
            def prt = Portfolio ( name = trade.name );
            foreach (entity in prt.entities) [ curves:*Curve ]
              {
                def curve = CreditCurves ( entity = entity );
                curves = curve;
              }
            def yc = YieldCurve ( currency = trade.currency );
            foreach (metric in metrics)
              [ metricResults:*MetricResult ]
                {
                  def findCalc = FindCalculator ( metric = metric,
                                               product = trade.product );
                  def calc = Calculate ( calculator = findCalc.name,
                                         yieldCurve = yc, creditCurves = curves );
                  metricResults = MetricResult ( metric = metric,
                                           val = calc.result );
                }
          /* combine the metrics into a single valuation */
          def val = Valuation ( results = metricResults );
          bookSum += agg.valuation ;
        }
  }
/* these are two different bookMetrics variables */
bookMetrics = bookMetrics ;
```

\(^2\)Credit curves indicate the probability of a company/entity in the portfolio defaulting on its debt.
\(^3\)A yield curve holds predicted interest rates over time.
The point is that it has a lot of dependencies, a lot of opportunities for parallelism and potentially lots of redundant (unnecessary) calculations if the problem is approached naively.

Listing 3.13: An example of a geophysical oil exploration code written in the latest version of Flow

```plaintext
graph Inverse (misfit: float, inputs: ForwardInput, model: FiniteElementModel) [result: float]
{
    /*
    * Each iteration will be performed in parallel
    */
    node forloop = foreach(input in inputs)
        [oneDFields: float]
    {
        node prim = ComputePrimaryFields(
            freq = input.freq, srcPos = input.srcPos);
        oneDFields :=prim.result;
    };
    
    /*
    * Each iteration will be performed in serial
    */
    node doLoop = do
        [init mModel: FiniteElementModel = model]
    {
        node total = ComputeTotalFields(
            fields1d = forloop.oneDFields, model = mModel);
        node deriv = ComputeDeriv(totals = total.response);
        node update = UpdateModel(
            deriv = deriv.d, currentModel = mModel);
        node misfits = ComputeMisfits(
            model = update.newModel, maxMisfit = misfit);
        next mModel = update.newModel;
    } until (misfits.smallEnough);
    
    /*
    * Assign final result to output hook
    */
    node resultNode = FinalResult(model = doLoop.mModel); result = resultNode.output;
}
```
Listing 3.12 shows the Credit Derivatives model described in an early version of Flow. The syntax has some clear differences compared to the final version of Flow, but the general structure should still be recognisable. The main difference is that in the final version everything has to be explicitly instantiated as a node (using the keyword "node"), whereas in this previous version this is all a bit more subtle. Loops just look like normal loops and nodes are defined using the def keyword. The reason this was changed was because an important concept in any dataflow language is that there is no such thing as a program counter and it is just data being passed between nodes. It was therefore decided to try and make this clearer by explicitly using the word node (instead of def) and to make everything (even loops) a named node instance.

The other real-life problem domain that was tackled was a Geophysical science application for identifying pockets of oil under the ocean floor [11]. This was a live project being worked on by Magnus Hagdorn\(^4\) in which he was splitting his existing implementation into tasks and was looking for a workflow engine to run it on. This presented itself as a perfect example application for testing the viability of Flow.

Unfortunately, due to the need for an NDA (non-disclosure agreement), usable code was not available until over half way through the dissertation, so the design of the Flow DSL had to rely on a very rough overview of what the application would look like. Listing 3.13 shows this draft version of the application, described using Flow.

A useful observation is that the application has an optimisation step whereby it repeatedly tries to improve the model until the model represents the input data well enough. This was the reason why the do-until construct was included in Flow.

The final point to make about the Flow DSL design is regarding the decision to have nodes as immutable instances, rather than using the conventional dataflow model of passing streams of tokens between nodes (i.e. treating nodes like pipelines). The reason for this decision was purely to aid in understanding and reasoning over the application. The problem with the pipeline paradigm is that if a node has multiple inputs you have to have special syntax and logic to enforce that the right set of tokens are processed by the node at the same time. Such a paradigm could work well if, for example, you have a live feed of input data and want the graph to be re-evaluated every time its inputs change. However, for Flow, the target applications were all one-off evaluations given a set of input data. The Flow model also more closely resembles an Object-Oriented model, which is familiar to many programmers.

### 3.8 Summary

The domain-specific language, Flow, has been designed specifically for the purpose of describing a data-dependency graph of tasks. It assumes that all tasks are referentially transparent (i.e. pure, with no side-effects) and that they will assign values to all their

\(^4\)http://www.geos.ed.ac.uk/homes/mhagdorn/
outputs upon execution. As opposed to traditional dataflow models, that pass streams of
tokens between nodes, Flow uses an instantiation model, whereby each node instance
can only be given a single set of inputs, after which it is immutable.

The Flow language uses tasks (i.e. native subroutines/functions) as its smallest building
block. Graph nodes and Foreach nodes provide opportunities for vast amounts of im-
licit parallelism to be extracted at run-time. Do-Until nodes and Switch nodes provide
high-level control flow.
Chapter 4

Abstract Syntax Tree

Once we have designed a new programming language (in our case: Flow), programmers can write applications in this language. However, for them to be of any use, they need to be able to compile them and execute them. The first stage of compilation is to transform the source code into an in-memory AST representation (i.e. an Abstract Syntax Tree) in order to check it for errors (i.e. syntactic and semantic errors). The stage of the project that this represents is shown in Figure 4.1.

In this chapter we will discuss how a software tool called ANTLR was used to generate the code to convert source code, written in the Flow language, into an abstract syntax tree (AST) that our Flow compiler can work with. We will also discuss how syntax and semantics errors are identified by ANTLR.

4.1 Syntax and Semantics

The first step in building any programming language is to convert the source code (usually stored in a text file) into an in-memory internal representation that the language’s compiler can understand and navigate. In order to be able to do that, the source code has to abide by a strict set of rules that specify which words and symbols can appear where in the document and how these can be combined. If they abide by these rules we
can derive meaning from this code and build our internal representation to represent this meaning. The rules we are referring to are commonly known as syntax and semantics.

A language abides by the syntax (i.e. is syntactically correct) if it has followed the grammatical rules of the language, that specify what makes a word and how these words can be combined in the language. On the other hand, a language abides by the semantics (i.e. is semantically correct) if the combination of those words also has some sensible meaning. For example, in English the sentence "Carrots run deceptively" is syntactically correct as it is in the form: subject-verb-adverb; however the sentence arguably does not make sense, so would not be considered semantically correct.

ANTLR \[12\] stands for ANother Tool for Language Recognition and is a very useful piece of software for building domain specific languages. You provide it with a description of your language’s grammar and it will automatically generate a Lexer and a Parser for you, in one of its many supported programming languages (C, Python, Java, etc.). A Lexer is a tool that takes your source code (in the form of an input stream) and splits it into a stream of valid tokens, e.g. keywords, symbols and identifiers that make sense in your language. The Parser then takes these tokens, checks that their use is syntactically correct, and if so builds an Abstract Syntax Tree (AST) that you can further process. An AST \[42\] is a standard tree-based in-memory representation that can easily be navigated in order to check the semantics of the source code. Figure 4.2 demonstrates the Lexer and Parser in action. The Lexer takes in sourcecode and spits out tokens. The Parser takes in tokens and spits out an AST.

4.2 Lexer

So far we have mentioned how you can provide ANTLR with a description of your DSL grammar and it will automatically generate code to, among other things, check the syntax for you. The way you describe your grammar to ANTLR is actually through its own domain specific language. ANTLR has a DSL for describing DSLs, which interestingly enough was implemented in a previous version of ANTLR \[42\].
The first piece of information to tell ANTLR is what it should consider to be a token in the Flow DSL. Listing 4.1 shows how this is done in the ANTLR DSL. As can be seen, the only tokens we are interested in are actually just the graph ids, variable ids, numbers and strings. Everything else, such as keywords and curly braces, etc. are purely there to give structure to the language and remove ambiguity. Once we have parsed the source code and we have an AST, we are no longer interested in those; but we are interested in what graphs are called, which variables we are referring to and which constants to assign to our nodes.

As shown in Listing 4.1, a graph name must start with an uppercase letter, followed by zero or more ANY_CHARS, while a variable must start with a lowercase. Numbers are represented by one or more digits and may have a floating point part to it. Meanwhile string constants start and end with speech marks and have zero or more non-speech-mark characters in between.

### 4.3 Parser

So far we have told ANTLR about some of the tokens that the Lexer should be recognising. What we have yet to tell it is what structural tokens, such as keywords and curly braces, look like. As will become clear when we look at this next part of the grammar, while this part of the grammar is aimed at the Parser, it also includes all the structural tokens that should also be recognised by the Lexer. The ANTLR DSL has it split this way because, unlike the tokens we have specified so far, structural tokens are actually relevant to both the Lexer and the Parser. The Lexer needs to know about them in order to turn them into tokens, and the Parser needs to know about them in order to check that the order of the tokens it is seeing is syntactically correct.

Listing 4.2: Telling ANTLR how graphs are defined in Flow

```antlr
graph: 'graph' GRAPH_ID interface graphBody;
interface: '(' declarations ')' ( '[' declarations ']' )? ;
graphBody: '{' statement* '}' ;
declarations: ( declaration ( ',' declaration )* ) ;
declaration: VAR_ID ':' type ;
```

Listing 4.2 shows a snippet of how the structure of the Flow language is described in the ANTLR DSL. The first line says that graphs are defined using the keyword ‘graph’, followed by a graph id (see Listing 4.1), followed by its interface, followed by its body.
The interface specifies the graph’s inputs and outputs (as stated on line two of Listing 4.2), and the graph’s body consists of zero or more statements (as stated on line three). Notice that the structural tokens (‘(’, ‘)’, ‘[’, ‘]’, ‘{’ and ‘}’, etc.) are included here, meaning that the Parser generated by ANTLR will output an error if it does not see them in the appropriate places.

4.4 AST

The previous section discussed how the Parser takes the stream of tokens from the Lexer and checks that they match our grammar. If it finds that the tokens do not it will output helpful error messages with a line number and position on that line. All this logic is automatically generated by ANTLR when we give it our Flow grammar. This next section will address what the Parser produces as output if it does not find any errors. Its output is an Abstract Syntax Tree (i.e. AST).

An AST is a directed acyclic tree structure where each vertex holds some information and may have one or more child vertices (we are using the term vertex/vertices here rather than node/nodes to avoid confusion with nodes from Flow). This structure ends up looking like a rooted tree where a vertex’s children provide more information about that vertex. This is best demonstrated with an example.

Listing 4.3: Telling ANTLR how assignments are defined in Flow

```plaintext
assignment: VAR_ID '=' rValue ';' ;
rValue: constant | instantiation | varRef | foreach
    | doWhile | switchStatement ;
constant: 'true' | 'false' | NUMBER | STRING ;
```

Take a simple Flow assignment statement: age = 65;. Listing 4.3 shows the grammar for a Flow assignment. Given this description, ANTLR does not have enough information to know how you want the AST to be built. To overcome this, ANTLR provides additional syntax (called rewrite rules) to tell the ANTLR Parser what to do. Listing 4.4 shows the adjustments that had to be made to Listing 4.3.

To add rewrite rules to your ANTLR grammar you simply add a "->" arrow and express on the right of this how the tokens you have matched should be rewritten in the AST. So, taking our statement "age = 65;", the rewrite rules for "assignment" say: create a new vertex, called N_ASSIGN, make VAR_ID its first child and expand rValue to be its remaining children. This can be seen in Figure 4.3. N_ASSIGN has been created as a placeholder for the root of the tree, and the token representing VAR_ID has been placed as its first child ("age"). We will continue with our example to see how the right hand side of this tree is constructed.

---

1Remember: inputs are declared between parentheses, while outputs are declared between square brackets
Now all that is left is to rewrite the rValue, which only consists of a single token: "65". If we follow the "rValue" rule in Listing 4.4 you can see that we have not added any specific rewrite rules for rValues, so we can just follow the "constant" rule. This shows that if the token is a number (which 65 is), it should be rewritten with an N_NUMBER vertex as its root and the token holding the number should be its only child. The result of this is shown in Figure 4.3.

This was a quick overview of how rewrite rules can be used to define how the AST should be built by your ANTLR Parser. There is a little bit more to it than this and the details are described in great detail in the official ANTLR book [42]. This example just showed the rewrite rules for a simple assignment statement. For this project the rewrite rules were added to the whole grammar and this can be seen in the Flow.g file, included in Appendix B.

4.5 Summary

A tool called ANTLR provides a lot of help for converting Flow sourcecode into an in-memory AST. The Flow grammar was defined using ANTLR’s DSL, and from this, ANTLR generated a Flow Lexer and a Flow Parser. Not only do they convert Flow
sourcecode to an AST, but they also output helpful error messages if there are any syntactic errors. In the next chapter we will discuss how semantic errors were identified and shown to the user. This will also include some examples.
Chapter 5

Flow Integrated Development Environment

An IDE (Integrated Development Environment) for a programming language usually comes in the form of an advanced text editor with support for common programming tasks. An example of such is the Eclipse CDT[16], which is an IDE for developing C++ programs. This IDE provides syntax highlighting, automatic compilation, auto-complete and debugging, among many other features. These helpful features aid in the process of successful application development.

The advantage of the Eclipse IDE is that it is built to be extensible and provides a lot of built-in functionality to make adding support for new programming languages easy. For this reason, instead of building the Flow compiler as a stand-alone application, it was instead built as a plug-in/extension to the Eclipse IDE.

The features of this will be described here.

5.1 Syntax Highlighting

The most basic feature of any IDE is syntax highlighting, as it groups keywords and certain identifiers into different colours and thus allows the programmer to quickly scan and interpret the structure of a sourcecode document. Figure 5.1 shows a screenshot of the syntax highlighting that was implemented for Flow sourcecode files. Keywords are shown in one colour, graphs in another, and constants (e.g. strings) in a third.

5.2 Incremental Compilation

One of the main motivations for integrating with Eclipse for the development environment was the support it provides for incremental compilation. Incremental compilation
is the technique of splitting a compiled application into *compilation units* so that when a change is made to the source code, only a few compilation units (not the whole application) has to be recompiled every time.

Eclipse provides an API for being notified when a resource (e.g. a sourcecode file) has changed. This means that an ideal compilation unit for Flow is a Flow graph (i.e. the result of compiling a single Flow sourcecode file). By splitting the compilation in this way, when a Flow sourcecode file is modified by the programmer, only its graph has to be recompiled (not all the other graphs too). That is, unless the *signature* of the graph is changed.

Listing 5.1: A graph signature

```plaintext
graph A (a:*int, str:*string) [x:int, y:int]
```

The *signature* of a graph is just the top-level inputs and outputs that define a graph. Listing 5.1 shows an example of what makes up a graph’s signature. If this part of the graph changes (e.g. an additional input hook is added or an output hook is removed) then the graphs that are using this graph (as inner graphs) will also need to be recompiled. If, however, the signature stays the same, and just the body of the graph is changed, those graphs that are using it do not need to know about it, as the way they use will not be affected.

Logic for performing these checks was implemented and the dependencies between
5.3 Error messages

The job of the compiler is first and foremost to check that the syntax and semantics of the defined graph are correct and that the compiler can understand what the programmer is trying to describe. If the compiler does not do this successfully, the result of the next phase; converting the description into an IR (internal representation); can produce undefined and erroneous behaviour, making it very difficult for the programmer to find where their mistake is located. Therefore, not only should a compiler not allow such semantically incorrect programs to be compiled, but it should help the programmer understand where they may have gone wrong.

The ANTLR-generated code (the Flow Parser and Flow Lexer) handles all the syntax checking, notifying the programmer of the exact position in the document where the grammar rules are not being followed. This comes with a helpful description of what it expects to find at this position. An example of such a syntactic error is shown in

Figure 5.2: Shows a syntax error underlined and indicated in the Problems window

compilation units was managed using an Eclipse metadata store (a directory of metadata files). This allowed the programmer to write Flow sourcecode files in the Eclipse text editor, and upon saving, all the necessary compilation units would automatically be recompiled. If no errors are found during the compilation stage then the appropriate C++ source and header files are generated. If there are errors, however, then these are displayed inline in the source document.
The semantics checking, on the other hand, had to be implemented by hand using the AST as input, but followed the same approach; providing detailed context-sensitive error messages at the exact line and position where the error was found. This feature not only makes programming in Flow easy and painless, it also made the job of testing the compiler simple and efficient. Whenever bugs were found, several test cases could quickly be created in the form of Flow sourcecode documents; and the error messages allowed you to instantly identify exactly how the compiler was interpreting them.

Checking for semantic errors involved traversing the AST, and building up a context-sensitive picture of which variables are available where, what their types are, and what fields (i.e. input/output hooks) these types contain. An example of the type of error messages that this can identify is shown in Figure 5.3. Here the code has been changed to reference an output hook called ‘outputs’, but there is no such hook in the PrimaryFields definition. The Problems window (at the bottom of the screenshot) shows the detailed error message that is produced.

### 5.4 Summary

The Flow compiler was built as part of an Eclipse IDE plugin and provides a number of useful features for the developer. These features are syntax highlighting, automatic incremental compilation (upon saving of a Flow sourcecode file) and in-line syntax and
Syntax errors are automatically generated by the ANTLR-generated code, however the semantics errors have to be derived by traversing the AST and deriving the program semantics.
Chapter 6

Internal Representation

When a programmer defines a program written in any programming language they use a textual representation defined in a source file. A compiler can then transform this source file into an executable that can be run on a processor. In the case of Flow, the DSL compiler transforms the DSL source into a C++ source file, which is later compiled by a C++ compiler into an executable. However, for any of these compilers to perform their transformation, they first need to convert their source file into an internal (in-memory) representation that they can traverse and interpret. This internal representation (or IR) should represent the program in a way that allows the necessary transformation to most easily be performed and to allow the set of desired optimisations to easily be identified.

Compilers like GCC even have multiple internal representations to allow a range of optimisations to be performed [13]. For Flow we have the AST (as described in the previous chapter), but this is not useful for the purpose of identifying parallelism and performing optimisations; so we convert this AST into what we call the Flow IR. The design and implementation of this custom IR will be discussed in this chapter.

Designing the IR involved going through a number of iterations before a design was formed that met all the requirements. The requirements for the IR revolve around being able to easily achieve the two main goals. These goals were as follows:

- minimal path discovery (MPD): identifying the smallest set of tasks that need to

![Figure 6.1: IR and Code Gen](image)
be computed in order to get the desired outputs

- extracting implicit parallelism: identifying which tasks are independent of each other, so that they can be performed concurrently. This is actually done at runtime, but we need an IR that makes it easy to generate the code.

As the strategy for tackling these two goals should effectively decide what the structure of the IR should be, we need to understand the issues that affect these two goals. The next two sections will therefore discuss these goals and the decisions that were made on how to implement them. We will finish in summary by indicating what the final structure of the IR looks like.

Figure 6.1 shows how this chapter fits into the overall project.

6.1 Minimal Path Discovery

Minimal path discovery (MPD) is an optimisation step that involves removing any redundant calculations from a graph. This cannot be performed at a program-level by the C++ compiler as it requires a number of assumptions to be made about the relationship between different tasks/functions. C++ has no restrictions to prevent different functions from changing global state and even allows the use of pointer arithmetic to access and change data that it does not own. Flow on the other hand works on the assumption that every task will only read and write data that has been passed to it; and only data that has been defined as output data will be modified (input data should remain untouched). These assumptions allow the compiler to identify exactly which tasks are
needed to compute a particular output. Figure 6.2 shows the set of tasks that are needed to compute output hook $y$.

A question at this point might be why this optimisation is needed, as why would a programmer add tasks to a graph if they are not interested in the output. There are two main reasons. The first is by accident, often through some form of refactoring and editing of code, where an output from a graph is removed as it is no longer needed, but the programmer forgets to remove its dependencies (Figure 6.3 shows an example of this, where node M is redundant). The second, and hopefully the more common instance, is where functionality is reused. This is where a graph has multiple outputs and in most cases we are interested in all of them, but there exist some use cases where we only want the result of a few. In this case an ideal solution is to have the compiler/runtime skip any redundant tasks, rather than the programmer having to write special case logic for every possibility. Not only does this save time and hide complexity for the programmer, but it also encourages good software engineering practice through code reuse [20].

### 6.1.1 Compile-time optimisation

The idea behind MPD is simple enough: you identify the output nodes of the graph that you are interested in the results for, then follow their dependencies backwards up the graph. Every node that you meet must be computed. Every node that you do not is redundant and can therefore be skipped. The complexity comes, however, when you consider inner graphs.

As we follow the dependencies up the graph we may find that in some cases we are interested in the result of all outputs of an inner graph, while in others we only need one.
Figure 6.4: In the left instance of graph B, not all its output hooks are used

Figure 6.5: The output hooks are annotated with the names of the hooks they help compute
Figure 6.4 shows an example of this, where the graph contains two instances of inner graph B, but does not use all its outputs in both cases. Instead of computing everything both times we can apply MPD to the inner graphs too to further remove redundant calculations. The problem is that the simplest solution, applying this process recursively whenever we encountered an inner graph, has unbounded memory requirements as the nesting can be arbitrarily deep. This would not be appropriate as the compiler should be able to handle any type (or size or nesting) of graph.

The other problem is that for each unique usage of an inner graph, the compiler would have to generate a separate graph permutation so that it can be referenced from the graph using it. This means $2^n - 1$ potential combinations for a graph with n outputs, i.e. a graph with 4 outputs could potentially be used in 15 different ways, and the compiler would have to generate code for each of these. In practice it may be rare that any particular graph is used in more than 2 or 3 different ways, but even so, it would mean that when a programmer changes how they use a graph, it is not just the graph that they are editing that must be recompiled, but also the inner graph that it is using. This could recursively lead to many other inner-inner graphs also needing recompiling. Effectively it would mean that you can never have pre-compiled libraries in Flow, but would have to compile them all from source.

The decision therefore was to restrict the compiler to performing MPD at the graph level (i.e. traversing the graph upwards assuming all outputs) and defer inner graph MPD until runtime.

### 6.1.2 Runtime optimisation

Graph level MPD is useful and can eliminate redundant tasks if they are not assisting in computing the outputs. Applying this further onto inner (i.e. nested) graphs is where the real strength lies, however, as this can eliminate large chunks of unnecessary processing. As discussed, the runtime engine should perform this optimisation step, however any help the compiler can provide to reduce the runtime overhead (of performing MPD) is valuable.

MPD involves traversing the graph backwards (starting at the outputs), whereas executing a graph involves traversing it forwards (starting at the inputs). Therefore if the compiler can perform some pre-processing of the graph so that the runtime only has to traverse the graph in one direction, this may halve the run-time overhead of graph traversal. Effectively, as the runtime traverses the graph forwards, it should be able to decide at each node whether that node is needed to calculate the graph-level outputs that have been requested.

To allow the runtime to do this, the compiler can add extra information to each graph node, specifying which graph-level outputs that node is a dependency of. More specifically, it can add information to each output hook on each node, so that the runtime not only knows whether the node is needed, but also whether all its outputs are needed. Figure 6.5 illustrates this (Note: this figure will be explained in more detail in a moment).
The runtime engine should start by processing the top-level (root) graph as normal, executing every task it encounters while it traverses the graph from the inputs downwards. It can do this because it knows that all outputs are required, and that any graph-level redundant tasks will have been removed by the compiler. When it reaches an inner graph, however, it now has to make a note of which output hooks of that graph are being used. If it is all of them, then it can also traverse and execute this graph as normal; but if some of them are not used then it must treat this inner graph slightly differently.

So, if not all of the inner graph’s outputs are needed, then as the runtime engine traverses inside the inner graph, for each node it encounters, it can look at the output hooks for that node, and if none of those hooks are needed (i.e. the annotations do not indicate that they are needed to compute the outputs we require), the node can be ignored (together with all its dependents). If on the other hand some of them are needed, then it does not skip the node; executing it if it is a task, or processing it as already described if it is an inner graph. Here is an example: looking again at Figure 6.5, assume this has been used as an inner graph, and we have found that we only need output $y$. The run-time engine will first reach task B. Looking at its outputs we can see that at least one of its output hooks help compute $y$, so we execute B. We then reach C, but its outputs do not help compute $y$, so we skip it. Then finally we reach task D, and this is needed so we execute it.

### 6.1.3 MPD Summary

Minimal Path Discovery (MPD) is a technique for identifying and removing redundant tasks from a graph. The decision was made to have the compiler perform graph-level MPD and leave inner-graph MPD to the run-time. The compiler, however, would help the run-time by tagging the output hooks of nodes with the names of the graph-level output hooks that they help to compute.

It should be mentioned here that, due to time-constraints, only the graph-level MPD was actually implemented in this dissertation project. For future work, however, the run-time MPD could remove significantly more redundant tasks.

The needs of MPD suggest that a graph-based representation would be ideal for the Flow IR.

### 6.2 Code Generation

Code generation is where we take the compiler’s internal representation, traverse it, and output C++ code that will represent the logic that was described in the Flow source code. In our case, the code, when run, should build a dependency graph at runtime that the run-time engine can use to schedule tasks correctly (and in parallel). For this the IR we have described in the previous section (i.e. a graph-based IR) also exactly meets the needs of the code generator.
The most intuitive approach to code generation seemed to be to traverse the graph forwards (from input hooks, downwards), generating code for each node as it was traversed. This, however, proved to be quite difficult as, in C++, functions and variables have to be defined/declared before they are used, whereas when generating code we do not know exactly which functions/variables we are going to need until we need them.

To overcome this it was actually found that generating the code in reverse was significantly more straightforward. Whenever a node was reached, while traversing backwards through the IR, the compiler can work out which functions and inputs it needs, generate code to call those functions with those input variables, and then make a note to itself that it needs to generate the required functions/variables later. Then once it has finished generating code for the C++ function it is currently generating, it can look up which other functions it needs to generate and start generating them.

Traversing the graph in reverse, actually also provides two additional advantages. By traversing the graph backwards, it also allows the compiler to combine the code generation with the graph-level MPD step, in a single traversal of the IR. This speeds up the compilation phase and also means less repeated logic in the compiler’s codebase.

The final advantage is that it also deals with node inputs that do not come from the outer graph’s inputs (e.g. constants). An example is shown in Figure 6.6 where B gets one of its inputs from a constant value: 5. Generating code by naively traversing the graph forwards would have missed code for this constant. By traversing backwards, this is dealt with trivially.

6.3 Summary

The requirements of the IR design are firstly that it easily allows redundant tasks to be identified (using MPD), and secondly that it makes the task of code generation easier. As was shown when discussing the needs of both of these goals, a graph representation
is actually ideal.

The nodes have hooks and the hooks are connected to other hooks. These connections allow the compiler to traverse the graph in both directions (both forwards and in reverse), although it was found that reverse traversal (upwards, starting from the root graph’s output hooks) was the best approach for achieving both goals.

The compiler was implemented to firstly traverse the AST and convert it into a graph (i.e. the IR)\(^1\). Secondly, this graph is then traversed in reverse, outputting C++ code as it goes along, only following nodes that graph-level MPD deems necessary.

\(^1\)This was non-trivial, but beyond the scope of this dissertation to discuss here
Chapter 7

Run-time Engine

The Flow run-time engine is the piece of software that contains all the logic for traversing and executing graphs at run-time. Its job is to efficiently execute the necessary tasks in the graph, using the provided input data. Specifically, its requirements are as follows:

- no task should be started before its dependencies have completed.
- if there is no dependency between two tasks they can be executed concurrently.
- if a task has no dependencies it can be executed at any time.
- if a task has one or more dependencies, the appropriate input data should be passed to it upon execution.

Meanwhile, desirable properties of the run-time engine are as follows:

- aim to maximise CPU resource usage, in order to reduce application completion time.
- extract parallelism where possible (down to the granularity of a task) in order to be as scalable as possible
- scale appropriately on different hardware environments (anything from a single dual-core laptop to a large cloud cluster or supercomputer)

The Flow compiler is a source-to-source compiler, transforming Flow source code into C++ code. Writing code to generate code, however, is a time-consuming and convoluted process. To simplify this process, instead of having the compiler generate all the code, a C++ API was designed to provide an interface between the generated code and a natively implemented C++ run-time engine. What this meant was that the only code the compiler needed to generate was the code to connect the nodes and hooks together and to link tasks to their associated subroutines.

The advantage of having such a clean API between the graph and the run-time engine is that the run-time engine can easily be replaced with different implementations depend-
ing on the setting. For example, if you are using a shared-memory machine your implementation does not have to worry about inter-process communication and the varying costs of communicating with different threads. Whereas on a distributed-memory cloud or supercomputer one can take advantage of vast amounts of memory and CPU cores. One could also potentially write implementations for different operating systems, e.g. Windows vs Linux.

In this chapter we will start by discussing the API that was designed to cleanly split the generated graph code from the implemented run-time engine. This will provide an understanding of how the run-time engine interacts with the generated code, but will also provide an insight into what code the compiler had to generate. After this, the remainder of this chapter will focus on the design of the run-time engine itself. There are a number of options for how this could have been implemented, and we will discuss some of these here. We will finish by suggesting an optimal approach.

Figure 7.1 indicates where this chapter fits into the overall implementation.

7.1 The API

The role of the Flow API is to provide a clear interface between the run-time engine and the compiler’s generated code. Listing 7.1 shows how this API can be used. The Graph class is the user facing part of the Flow API. It is the role of the compiler-generated code to instantiate this graph (and connect it to all its nodes), and it is the role of the run-time engine to traverse and execute it.

The ParallelSharedMemoryScheduler class is the implementation of the run-time engine that was implemented in this dissertation. Its design will be discussed shortly. The idea is that this run-time engine can easily be replaced with any other implementation, as long as that implementation is able to traverse a Graph (notice how the graph is passed to the scheduler via its execute method). The buildGraphA function is part of the code that was generated by the compiler, and in this case generates an instance of the Graph class that represents graph A. The compiler will generate such a function for every graph that the programmer defines (in Flow), so any graph can be used as the root node to execute. All a programmer needs to do to execute a graph is call the
build function to generate an instance, assign values to any inputs of that graph (note that \texttt{A\_obj} is a struct that the compiler has generated), and then pass that instance to the run-time engine.

Listing 7.1: An example of how to run Flow code

```cpp
#include "A\_generated.hpp"

int main()
{
    // instantiate the run-time engine
    int noOfCores = 8;
    ParallelSharedMemoryScheduler scheduler(noOfCores);

    // call compiler-generated code to create the root node
    Graph* graphA = buildGraphA();

    // assign values to any inputs
    A\_obj* a = (A\_obj*)graphA->getObj();
    a->input = 53;

    // now pass the graph to the runtime engine for execution
    scheduler.execute(a);

    // now read out and print the output(s)
    cout << "Result:" << a->output << endl;
}
```

The aim of the run-time engine is to traverse and execute the graph that represents the application. And the aim of the compiler is to generate code that builds the graph. For this reason, the Flow API is a set of nodes (e.g. graphs and tasks) and hooks.

Everything in Flow is a node (e.g. tasks, graphs, foreach loops, etc.) and every node has input and output hooks. The API therefore reflects this. There are two top-level classes: AbstractNode and Hook. AbstractNode is the abstract base class for all nodes and therefore holds input and output Hooks. The Hook class has an owner (i.e. the AbstractNode it belongs to) as well as a dependency (for traversing up the graph), and 0 or more dependents (for traversing down the graph). Figure 7.2 illustrates the full class hierarchy.

As this hierarchy shows, there are two main types of AbstractNode: AbstractGraph and Task; and the run-time engine can differentiate between them by using the \texttt{isTask} method of their abstract base class (AbstractNode). The Task class, as expected, represents a task, and provides an \texttt{execute} method for the run-time engine to call. This will call the underlying subroutine that this task represents.

AbstractGraph is the abstract base class for all other nodes and has a \texttt{build} method for
the run-time engine to call. The run-time engine does not need to know what type of AbstractGraph it is dealing with; just that after build has been called, it can use the graph’s hooks to traverse inside it. In other words, when build is called, a compiler-generated function will be called that instantiates all the nodes that represent this inner graph, and connects them all together via their hooks.

All the main language constructs (Graph, Foreach loop, DoUntil loop and Switch statement) are represented as concrete implementations of the abstract class AbstractGraph. They each have their own implementations of the build method that construct a graph of nodes to represent their logic. Note that due to time-constraints only the Graph and Foreach subclasses were implemented, but DoUntil and Switch should be straightforward to implement in the future.

Listing 7.2: Showing how the runtime engine can assign values from one node to another

```c++
void* objA = nodeA->getObj();
void* objB = nodeB->getObj();
void* A_x = nodeA->getHook("x")->getter()->get(objA);
nodeB->getHook("a")->getter()->set(objB, A_x);
```

There are three more methods from Figure 7.2 that we have so far neglected to mention. These are getObj, getter and setter. These are all present to allow the run-time engine to move data between nodes. The role of the run-time engine is not only to traverse nodes and execute tasks, but also to pass data between them. This is because
connections between hooks represent data-dependencies, so when a task completes, its output values will need to be extracted and passed to the tasks that depend on it.

Listing 7.2 shows a C++ code example of how the run-time engine can extract the value of node A’s x value and pass it to node B’s a value. Notice that the run-time engine can simply pass around void pointers (i.e. anonymous object pointers), as it does not need to know what the underlying object type is. All the casting and extracting of data is performed by compiler-generated code (e.g. the Getter and Setter objects returned by their respective methods).

Listing 7.3: A graph definition

```cpp
graph A (a:int, str:string) [x:int, y:int] {
...
}
```

Listing 7.4: Some of the code generated by the compiler

```cpp
struct A_obj_struct {
    List a;
    String str;
    int x, y;
};
typedef struct A_obj_struct A_obj;

// the setter for A’s x hook will call this function
void A_set_x (void *obj_, void *val_) {
    ((A_obj*)obj_)->x = *((int*)val_);
}

// the getter for A’s x hook will call this function
void* A_get_x (void *obj_) {
    return &((A_obj*)obj_)->x;
}
```

For completeness we will show a quick example of the type of code the compiler is generating behind the scenes (although understanding this is not necessary to implement a run-time engine). Take, as an example, a graph that has been defined as shown in Listing 7.3. This has two inputs and two outputs. Among other things, the compiler would generate a struct to hold this data, and getters and setters for each of the inputs and outputs. Listing 7.4 shows some of the code that would be generated. Notice that, to make life easier for the run-time engine, the struct contains both the inputs and the
outputs. When the run-time engine calls the `getObj` method, an instance of the `A_obj` struct is what it will receive a pointer to.

This overview has provided a high-level summary of the C++ API that was designed, and also an insight into what code had to be generated by the compiler. Designing an API to separate the compiler from the run-time engine meant that the same graph could be run with different implementations of the run-time engine. A serial implementation of the run-time engine was implemented first in order to test the correctness of the API and the compiler-generated code. This serial version will not be discussed here, but what will be discussed is the design and implementation of the parallel run-time engine. This we will look at now.

### 7.2 Run-time Engine Design

There are many different approaches to adding parallelism to an application. Examples include distributing the tasks across $n$ threads at compile-time, or having a single thread that controls the whole application and submits tasks to a threadpool. These are both simple and effective solutions for many problems\(^1\), but they do not meet the needs of Flow. The aim of Flow is to dynamically take advantage of the hardware resources available, by extracting as much parallelism as the application (and its input data) provide. This means that firstly it should not have to rely on compile-time information for how many threads to distribute data across (this should be based on the number of CPU cores available). And it should be able to scale to an arbitrarily large system (with hundreds or thousands of CPU cores and nodes), so having a single centralised controller would quickly become a bottleneck.

In the following subsections we will present a number of promising published approaches/algorithms from the literature, discuss if and how they could be applicable for the Flow run-time engine, and finish by proposing an optimal algorithm.

#### 7.2.1 Work stealing

Work stealing is a popular algorithm for dealing with load balancing in thread-pool style task farms \([21]\). The general idea is that there are a number of threads, who each have their own task queue. They pop tasks from the top of their queue, execute them and then return to fetch the next task. If a thread runs out of tasks (or in some implementations, when it runs low on tasks \([50]\)) it will seek out another (victim) thread and take (i.e. steal) tasks from that thread’s task queue. This is work stealing.

Dinan et al. \([29]\) have demonstrated a work-stealing algorithm successfully scaling up to 8,192 processors. They describe a number of techniques that helped them to achieve good performance and reduce lock contention.

\(^1\)MPI and some parts of OpenMP are examples of these, respectively
Figure 7.3: Each thread has access to its own local queue and others’ shared queues

Their first suggestion was that a thread’s task queue should be divided into shared and local portions. Figure 7.3 illustrates this. The local portion should only be visible to the owning thread and thus requires no lock for thread safety. The shared portion should be available to anyone, so any stealing threads (thieves) must obtain a lock before accessing items from a victim’s task queue. The owning thread regularly checks the size of its shared portion, and whenever the size ratio between the two becomes too large, it will move tasks from the local portion to the shared portion, or vice-versa. Dinan et al.’s algorithm is designed so that the jobs of checking the size of the shared portion, and moving tasks into it from the local portion, can both be safely performed without obtaining a lock.

Their second suggestion is that when threads steal tasks from a victim thread they should take half of the tasks present in the shared portion of the queue (rather than some fixed number). This reduces how regularly a thief will have to come back to steal more tasks, but also leaves the victim thread with enough tasks so it is less likely to have to steal more tasks for itself later.

Their final suggestion is that the locking on the shared portion of a task queue be implemented using spin-locks rather than sleep locks. The difference between these two is in how they deal with the situation when a thread tries to obtain a lock but another thread is already using it. The sleep lock will send the thread to sleep until the lock becomes available, thus freeing up that CPU core to perform other tasks. The spin-lock, on the other hand, will keep the thread active, looping around and repeatedly trying to obtain the lock. They describe two advantages to the latter.

The first is that obtaining a spinlock is much faster for a local thread than a remote thread, meaning that the (local) victim thread will implicitly get priority (this is not the case for a sleep lock) and thus will be able to process more tasks and therefore have more chances to create more tasks that the (remote) thieves can steal. The second is that it tackles the problem of thief threads waiting patiently for a lock, just to find that when they do get the lock, all the tasks have been taken by another thread. The spin-lock allows these threads to perform a lock-free check of the shared-portion size between each loop, and if they find that the queue has depleted, they can exit early and go in search of another victim. This leads to less wasted (idle) time [29].
7.2.2 Scheduling

The additional challenge for Flow is that a Flow graph does not just define a static set of tasks that can simply be distributed evenly among threads. Which tasks, and how many tasks will be created depends on the input data at run-time; so can be vastly different on each run. Tasks also need to be able to generate other tasks, and more difficultly, results from multiple tasks need to be able to be aggregated together.

Heinis et al. [33] present a "workflow execution engine" designed to dynamically handle workflows with variable and unpredictable work loads. Instead of just having worker threads, it divides its threads into navigator threads and dispatcher threads. To avoid mixing terminology, we will refer to these as scheduler and worker threads, respectively. Worker threads, like those in the previous section, execute tasks from a queue. Scheduler threads, on the other hand, take completed tasks and use the result of these to decide whether to schedule (i.e. create) more tasks. The scheduler threads are aware of the dependencies between tasks in the workflow, while worker threads just blindly execute tasks that they are given.

The communication of these threads is done through two global containers (that Heinis et al. call "tuple spaces"). One contains scheduled tasks while the other contains completed tasks (This is illustrated in Figure 7.4). Whenever a scheduler creates new tasks it will add them to the scheduled tasks container. Worker threads will then take a task from that container, execute it, and place the task, together with its result, back in the completed tasks container. These will get picked up by the schedulers and the cycle will continue.

As they explain, the advantage of their solution is that, as the ratio of scheduled and completed tasks changes, schedulers can dynamically switch themselves to the role of worker, and vice versa. This tries to reduce the amount of time any one thread is left starved of work to do. The disadvantage of the solution, however, is that, while their paper does also describe local caching of tasks, it still involves a centralised location for holding the scheduled and completed tasks. By having such a single location that
7.2.3 Distributed-memory load balancing strategies

In order to build a scalable task farm we need to look at algorithms that take a more distributed approach to task management. The disadvantage of distributed approaches is that no single thread will have an up-to-date picture of the whole system at all times. And even if one did try to obtain such a picture (by for example communicating with every other thread), the picture will often be out-of-date before any informed decisions can be made with it. The critical advantage, however, is that you can continue to add additional hardware resources (additional multi-core CPU nodes) to a system and there will be no single communication bottleneck, as all the new threads will only be communicating with threads/resources local to themselves. This should lead to a highly scalable run-time engine.

Willebeek-LeMair et al. [50] have conducted such a survey, evaluating "strategies for dynamic load-balancing" on distributed-memory, multi-core computers. They discuss the results of conducting performance experiments on five dynamic load-balancing strategies, all of which use distributed control. As shall be discussed, a strategy called Receiver Initiated Diffusion (RID) shows the highest potential for the Flow use-case.

A Flow application always starts from a single root node (or graph) that specifies the inputs to the application and the desired outputs. Within that graph it has various constructs, but what it boils down to is executable tasks and embedded inner graphs. These

![Diagram of a task farm](image.png)

Figure 7.5: A thread’s subtasks will always eventually come back to it for aggregation. Each vertical line represents an executing thread and the rectangles are tasks.

...all threads need to communicate with, the amount of contention for this resource will increase linearly with the number of threads and is unlikely to scale to large distributed-memory systems.
inner graphs can contain tasks and inner graphs of their own, all of which may need executing and traversing, but eventually these will all complete, their results aggregated, and what will be left is the single outer root node, with all its outputs resolved. What this means is that this single graph node, that may expand to have hundreds if not thousands of concurrently running tasks, will always start, and eventually finish, with just a single running thread. Figure 7.5 illustrates this, showing that any thread that has created tasks will always give itself at least one of its subtasks (so will usually stay busy) and the completed work will always come back to it, as it is in the best position to perform the aggregation (as it knows where all the tasks are).

With such an application, that always grows and shrinks from the same point, an ideal distributed-memory load-balancing strategy is one that tries to allocate any new tasks to threads that are physically as close as possible to the thread that created them. This is because if a thread created them, then it is also likely to want to access their output data later. As Figure 7.6 shows, if your physical CPU nodes are for example connected in a mesh topology, and the initial worker thread is in the bottom-left corner, then an ideal distributed-memory load-balancing strategy would grow from this corner and then shrink back down to it later.

Willebeek-LeMair et al. describe five strategies; two of which they found to be particularly good for applications "with a high measure of locality (local communication dependencies between tasks)". These are SID and RID, meaning Sender and Receiver Initiated Diffusion, respectively. RID is based on SID, with the major difference being that instead of threads sending tasks to neighbouring threads when their task queue is too big, in RID it is the thread with too few tasks that initiates the request for tasks from its neighbours (if its task queue is too small). In other words RID is based on work-stealing, while SID is based on work-sharing. The authors found that RID was the approach that "could most easily be scaled" and is "best suited for a broader range of systems."

The way that RID works is that there are worker threads running on separate CPU nodes, each holding a queue of tasks. Each thread will work on its local tasks, but will regularly poll its neighbouring threads to find out how many tasks they have in their queue. The thread calculates the average number of tasks that its neighbours have, and if this number is higher than the size of its own task queue by some threshold, it will initiate a load re-balance. A load re-balance means requesting an appropriate number

Figure 7.6: Workload grows/shrinks from/to bottom left corner of distributed-memory mesh topology
of tasks from each of its neighbours so that at the end of the re-balance all its neighbours will have the same number of tasks as itself (give or take 1).

More formally, a worker thread will initiate a load re-balance if it finds that:

$$\frac{1}{n} \left( \sum_{i=1}^{n} \text{noOfTasks}(i) \right) > \text{noOfLocalTasks} + \text{threshold}$$  \hspace{1cm} (7.1)

where a worker thread has $n$ neighbours. And during a load balance, the total number of tasks to request from neighbour $x$ can be estimated as:

$$\text{noOfTasks}(x) - \frac{1}{n+1} \left( \sum_{i=1}^{n} \text{noOfTasks}(i) + \text{noOfLocalTasks} \right)$$  \hspace{1cm} (7.2)

This strategy matches the needs of Flow quite well by only migrating tasks to neighbouring CPU nodes. It also does not over-burden the busier CPU nodes with the overhead of migrating tasks, as the majority of load re-balancing will be initiated by less busy CPU nodes.

### 7.2.4 Proposed algorithm

Given the above discussion, the proposed load balancing strategy, for implementing the Flow run-time engine, is a hybrid between the distributed RID algorithm and a simplified, shared-memory version of the worker/scheduler algorithm. The idea is that in each shared-memory CPU node there will be a single scheduler that handles completed tasks and the creation of new tasks. The rest of the cores on the node are then given the role of worker; performing any tasks the scheduler creates. Meanwhile, every scheduler is
regularly communicating with the schedulers of neighbouring CPU nodes, load balancing as described in the RID algorithm. This is illustrated in Figure 7.7, where the grids represent shared-memory CPU nodes, and the black squares are the scheduler threads, communicating with neighbouring schedulers (note: schedulers will always place new tasks that they create on their own queue first).

The main reason for choosing this hybrid approach is that it meets the needs of the Flow runtime but is also relatively simple to understand and implement. RID provides the advantages of work-stealing, which as we discussed earlier, provides scalability of up to thousands of concurrent threads. While an intra-node worker/scheduler strategy provides enough of a centralised controller for the handling of task generation and aggregation to be straightforward.

One difference in the way that the scheduler/worker strategy has been proposed to the way it is described by Heinis et al. is that their design had multiple schedulers instead of just one. While this did provide the advantage that the workers could convert themselves to schedulers if the tasks were being completed faster than new tasks could be created, it was one of our initial assumptions that individual tasks would be substantial (i.e. coarse-grain) enough that their run-time would easily overshadow the overhead of task creation. As we shall see later (in chapter 8) when we discuss the implementation results, this is certainly the case.

The other advantage of this approach is that it provides a clear distinction between shared-memory and distributed-memory logic. This is important as the communication latency between threads on different CPU nodes is significantly larger than that of threads on the same node. Had we, for example, chosen to use the RID strategy at the thread-level, rather than at the CPU node level, tasks would have been just as likely to migrate between nodes, as not, meaning that some Flow applications might be unlucky and have a large number of their task migrations occurring inter-node. By using the hybrid model we have proposed, the majority of inter-thread communication is occurring in shared-memory, while any inter-node communication occurs irregularly and in large chunks (between schedulers). This leaves us in the nice position of being bandwidth-bound rather than latency-bound for inter-node communication.

Due to time constraints, only the shared-memory part of this run-time engine library was implemented, but for future work it should be relatively straightforward to extend the library to include the RID algorithm. As will be discussed in the next chapter, the performance of this library is good, and the only constraint on scalability is the size of the shared-memory CPU node.

7.3 Summary

In this chapter we started by discussing how the Flow API was designed to provide a clean interface between the compiler and the run-time engine. An overview of the main classes and methods was given, together with examples of how the run-time engine can
navigate a graph and execute its tasks. A quick insight was also provided into the kind of code the compiler has to generate.

The remainder of the chapter focused on approaches and algorithms for implementing the run-time engine. A hybrid approach for running on a distributed-memory system was proposed, but due to time-constraints, only the shared-memory part of this implementation was implemented. While this implementation will never be able to scale as highly as a full distributed-memory implementation, it should be able to take full advantage of a large multi-core shared-memory machine. In the results and evaluation, discussed in the next chapter, this is what we show.
Chapter 8

Results and Evaluation

The aim of this dissertation project was to show that describing programs in the form of data-dependency graphs is not only intuitive, but also provides opportunities for scalable parallelism to be efficiently identified and utilised by a compiler and run-time engine. Importantly, the programmer should not have to express any explicit parallelism, but the dependencies (or lack of) between tasks should provide all the information necessary to correctly identify which tasks can be run concurrently.

There are two sides to the question of whether this aim was achieved. The first is whether this representation provides enough information for a compiler and runtime to extract scalable parallelism, and if so, can it do so efficiently. The second is whether the restrictive nature of a data-dependency graph representation is expressive enough to allow the intricacies of real-life problems to be described. These will be addressed in the following two sections.

8.1 Efficiently extracting scalable parallelism

The first question that the dissertation wanted to address was whether scalable parallelism can be extracted from a dependency-graph representation of an application, and if so, can it be done so efficiently. The question of scalability relates to how much additional performance (i.e. reduction in completion time) can be achieved by providing additional hardware resources (i.e. CPU cores). Meanwhile, the question of efficiency relates to how high the run-time overhead of this parallelism identification process is.

8.1.1 Experimental setup

In order to accurately test and measure this scalability and overhead, a benchmark application was created with a number of benchmark parameters that allowed different scenarios to be tested. The general setup was a graph, like the one shown in Figure 8.1,
Figure 8.1: A string of tasks is executed in serial for every integer passed to the input hook. In this example the input hook consists of 3 integers, but this graph is dynamically built at run-time, so could have an arbitrary number of concurrently executing strings of tasks.
that took a collection of integers as inputs (assume 3 integers, for the purpose of the diagram). For each integer in the input, a linear string of tasks was executed. And at the end their results were aggregated and the result assigned to the graph output hook.

Every task represented the same subroutine, but the size of the task workload was able to be altered. The task did not need to produce any real result, but it was important that it kept the CPU core busy (so it could not just be implemented to sleep for \( n \) seconds). The subroutine was implemented to initialise a local variable-sized array, and then sum up its values. This result was then printed to \texttt{stdout}; just to ensure that the C++ compiler would not optimise it out. The benchmark parameters were as follows (three of which are shown in Figure 8.1):

- \textit{task-workload}: the size of the variable-sized array in the tasks’ subroutine
- \textit{tasks-per-input}: the length of the linear string of tasks that is executed for each input integer
- \textit{no-of-inputs}: the number of input integers
- \textit{no-of-workers}: the number of threads allocated to the role of worker

These experiments were conducted on an Amazon EC2 [14] Extra Large Instance [15]. This consisted of 4 virtual dual-core processors and 15 GB of main memory. As one core is needed to run the scheduler, and the machine has 8 virtual cores, a maximum of 7 workers was used in these experiments. The runtime engine is also only a shared-memory implementation, so only one Amazon EC2 instance was used.

### 8.1.2 Scheduler overhead

The first experiment was to measure the overhead of the Flow scheduling process. In order to do this a linear Flow graph was directly compared against the equivalent program written to call the Task subroutines directly (without the Flow run-time). In other words, a C++ program was written that called the Task subroutine 20 times in row. To match this, the benchmark parameters were set as follows:

- \textit{tasks-per-input}: 20
- \textit{no-of-inputs}: 1
- \textit{no-of-workers}: 1

Table 8.1 shows how serial completion time compares to the Flow completion time as we varied \textit{task-workload}. What this shows is that the difference between the two is consistently around 5 milliseconds, which equates to around 250 microseconds per task. This is a useful statistic, as it shows that the overhead of the flow run-time is around a quarter of a millisecond per task, suggesting that task granularity only needs to be a few milliseconds for Flow to be efficient.
Table 8.1: Serial impl vs equivalent Flow program

<table>
<thead>
<tr>
<th>task-workload</th>
<th>serial (ms)</th>
<th>Flow (ms)</th>
<th>diff (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>27</td>
<td>31</td>
<td>4</td>
</tr>
<tr>
<td>3000</td>
<td>940</td>
<td>944</td>
<td>4</td>
</tr>
<tr>
<td>5000</td>
<td>2610</td>
<td>2616</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 8.2: Shows what proportion of runtime the worker spends idle, for varying task lengths
Figure 8.2 illustrates this more clearly, showing what proportion of total run-time the worker was left idle; due to waiting for the scheduler to provide it with its next task. This shows very promising results, showing that as long as tasks are written to be at least 10ms in length, then the overhead of the Flow run-time is less than 2%. It is important to note that this is only for a single worker. We shall look at how this overhead scales next.

8.1.3 Scalability

The previous sub-section showed that the efficiency of the Flow runtime engine is good, with around a quarter of a millisecond of overhead per task. The other experiments of interest are to investigate how this overhead is affected by scaling in both resource and problem size. All of these experiments used 100 tasks-per-input.

The first experiment was designed to test weak scaling, where we look at how performance is affected by increasing both the size of the resource (in this case, the no-of-workers) and the problem size (i.e. the no-of-inputs) linearly. What this means is that problem size per worker is kept constant, which is a weaker requirement than strong scaling (discussed shortly). An ideal result, meaning perfect weak scaling, would be for runtime (i.e. completion time) to remain constant.

As shown in Figure 8.3, this is not far off. There is some variation due to noise, but up to 6 workers the completion time is effectively flat. An interesting observation is that all of them tail off in performance at 7 workers. The exact same observation can be made when running on a 4-core machine, but at 3 workers. This is because the Amazon
machines have background processes that can affect the run-time. As long as at least one core is available to perform these background processes, their noise does not affect the results.

The second experiment tests strong scaling. This is where we look at how performance is affected by once again varying the no-of-workers, but keeping problem size constant. i.e. to what extent can we reduce the completion time of a fixed problem by providing additional hardware resources. The no-of-inputs is fixed at 4, so at any point in time a maximum of 4 tasks will be available to be executed in parallel.

Figure 8.4 shows exactly what we would expect. With up to four workers the average time spent idle by any given worker is effectively 0% as there is always a task for the worker to perform and the task-workload is big enough to make the scheduling overhead insignificant (task-workload=2500, which equates to about 7ms per task). As we increase the number of workers beyond four, the average idle time per worker increases linearly, as there is only enough parallelism in the application for four workers to be busy at any point in time.

Figure 8.5 shows the same data, but in terms of time (rather than as a percentage). The Idle and Active times add up to the Total time as any worker can only be in any of those two states. We see the same convergence between Active and Idle beyond four workers, but a positive observation is that total run-time stays relatively flat after this point; meaning that the overhead of having additional idle workers does not have a significant negative effect on performance.

It is also positive to see that total time decreases logarithmically, up to 4 workers, as we
double the number of workers. This shows that on a logithmic scale performance increases linearly, which indicates perfect strong scaling. This is however what we would expect, given the pure parallelism expressed in the benchmark. The more interesting question, is how does the Flow run-time perform on real-life solutions.

8.2 Implementing real-life problems

There are many ways in which a data-dependency graph can be described, but as discussed earlier, the decision was taken to design and implement a domain specific language (DSL) specifically for this task. This was to make the job of implementing the compiler and runtime as easy as possible, as this allowed full control of the expressivity of the language. By doing so, any necessary assumptions could be (and are) enforced, and the number of different combinations of operators the compiler and run-time engine have to be able to handle could be restricted. The challenge, however, was to still have enough expressivity that real-life problems could successfully be implemented.

In order to ensure that enough expressivity was available, the design of the DSL was done with two real-life problems in mind. The first was a high-level model for calculating the price of Credit Derivatives trades in an investment bank. This was the original motivation for this project and was helpful for deciding on the concept for the language. Unfortunately this was too large a project to tackle in this dissertation. The second was a Geophysical science application for conducting oil exploration. This presented itself
as ideal, as the owner of this project, Magnus Hagdorn\(^1\), was in the process of splitting this program into independent tasks and would just need a workflow framework, like Flow, to coordinate the processing of its tasks.

Listing 8.1: The geophysical model implemented as a Flow graph

```c
graph Forward (input: int) [output: int]
{
  node setup = Setup(
    input = input, config = "test/halfspace.cfg");
  node freqs = foreach(freq in setup.freqs) [result: int]
  {
    node wavs = foreach(waveNo in setup.waveNos)
      [result: int]
      {
        node srcPoss = foreach(srcPos in setup.srcPoss)
          [result: int]
        {
          node primFields = PrimaryFields(waveNo = waveNo,
            srcPos = srcPos, freq = freq);
          node scatter = ScatterFields(
            primary = primFields.output,
            waveNo = waveNo,
            srcPos = srcPos,
            freq = freq);
          result += scatter.result;
        }
        result += srcPoss.result;
      }
      node response =
        Response(input = wavs.result, freq = freq);
      result += response.output;
  }
  output = freqs.result;
}
```

Confirming the language would provide enough expressivity to implement the Geophysical workflow was difficult during the design phase as the NDA (non-disclosure agreement) did not come through until half way through the project; and Magnus had not finished splitting out his program into independent tasks until around the same time. This meant that the DSL design was put together with only a very high-level idea of what the final workflow would look like.

In the end, Magnus had successfully implemented a large subset of his application (i.e. the Forward computation [11]) as separate tasks. This unfortunately did not include

\(^1\)http://www.geos.ed.ac.uk/homes/mhagdorn/
the model optimisation step which would have provided opportunities to use the \textit{do-until} construct, but it did present graph parallelism (i.e. independent tasks) and loop parallelism (i.e. multiple data items that require the same computation to be performed). The nodes were implemented as separate executable programs, but they did require some data to be passed between them\footnote{The frequencies, wave numbers and source positions.}, and there were explicit dependencies between them, so some could not be run until others had completed.

The application is not strictly what Flow was designed for, as Flow was designed for calling subroutines (rather than programs) and assumes that those subroutines have no side effects (whereas these programs were reading and writing files to/from a shared filesystem). However, the constructs are present in Flow to allow such an application to still be successfully implemented. The graph was implemented by implementing the tasks as C++ functions that executed the associated external program, and the dependencies between tasks were enforced by passing an arbitrary value between them (even though this value was not always used). Listing 8.1 shows the Flow graph that was implemented.

As can be seen, the Setup stage generates a set of frequency ids, wave numbers and source positions. For each combination of these a PrimaryFields calculation is performed, and a ScatterFields computation is performed using this primary fields output. A Response calculation then performs an aggregation step for each frequency. The actual aggregation is performed on files inside the Response program, but Flow aggregation (+=) is used to enforce the right dependencies.

The application was tested with 8, 10 and 12 worker threads. This was simply a matter of passing a different number as a command-line argument (see Appendix A). Magnus confirmed that the output results looked correct. As can be seen from the size of the sourcecode in Listing 8.1, writing this graph in Flow took no longer than a few minutes, including time used to adjust the graph to get the dependencies right. The sourcecode is small, yet describes a relatively complicated set of dependencies; and once Magnus finishes his full application, this graph can be used as is by referencing it as an inner graph.

Implementing the C++ subroutines to call Magnus’ external programs took under an hour, and once written they did not need to be modified, even when changes were made to the graph. The advanced compiler error messages also meant that any mistakes (e.g. misspelled variable names or missing inputs) were instantly identified. At no point did we have to check the compiler-generated code to identify a problem.

At the very inner loop there were 480 independent tasks to compute (10 source positions × 24 wave numbers × 2 frequencies). Table 8.2 shows the breakdown of how long each task took to execute in isolation, how many times each task was invoked, and what the accumulative total time would have been if they were all run in serial. Clearly PrimaryFields is where the vast majority of the workload is.

Given the information in the table, the expected runtimes with ideal (i.e. perfect) scaling
Table 8.2: Task timings

<table>
<thead>
<tr>
<th>Task</th>
<th>Execution time (ms)</th>
<th>Invocations</th>
<th>Accumulative (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>2600</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>PrimaryFields</td>
<td>24700</td>
<td>480</td>
<td>11,856</td>
</tr>
<tr>
<td>ScatterFields</td>
<td>1100</td>
<td>480</td>
<td>528</td>
</tr>
<tr>
<td>Response</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12,386</td>
</tr>
</tbody>
</table>

Figure 8.6: Shows actual vs ideal completion time, given varying number of worker threads
can be computed (i.e. how the completion-time should vary, in a perfectly efficient system, as the number of worker threads varies). This is shown in Figure 8.6 against the (admittedly limited) actual times that were recorded. No system will display perfectly efficient scaling, as there are always overheads such as thread management and, in the case of Magnus’ workflow, process startup costs for each task; however despite this (for the data that we have) the actual performance is very close (within 10%) of ideal.

The speedup results are as impressive as one could have hoped for (7.5 times on 8 cores), but not unexpected. The tackled graph problem, although real, is still relatively trivial, and embarrassingly parallel. Every expensive task (i.e. the PrimaryFields tasks) is independent of every other, and the reduction step (Response) is insignificant. These results merely confirm that the dynamic load-balancing of the run-time engine is working well; and the correctness of the results confirm that the dependencies are being respected, meaning thread-safety is not an issue. The more important question is whether Flow was expressive enough to represent a real-life problem.

The answer is yes, as the implementation of this graph shows, however unfortunately not all the features of Flow were used. There were no redundant tasks to be removed by MPD; although this is easily tested by adding a redundant task; and this works well. One of the key features of Flow (and dataflow in general) is that data should be passed along arcs between nodes, rather than accessing a shared file system, or local memory. This was shown to a limited extent, with the wave numbers, frequencies and source positions being passed to the PrimaryFields and ScatterFields nodes, but these were simple integers, being passed across shared memory. An interesting analysis for the future will be to see whether this becomes a bottleneck when larger constructs are passed across a distributed-memory cluster. The Do-until and Switch functionality were also not tested, and for future work it would be interesting to see what kinds of overhead they introduce.

The most promising finding, was a qualitative one, finding how quickly a real-life application can be linked together using Flow. Although Magnus was unfamiliar with the Flow language he was able to understand the notation and point out where the graph was not representing the right logic. Being able to provide such a high-level overview of the application, in just a single page of code, also means that it is quick and easy to get an understanding of how the whole application fits together.

8.3 Summary

This Evaluation chapter has addressed the two main questions of the dissertation:

- can the Flow compiler and run-time engine efficiently extract scalable parallelism from an application described as a data-dependency graph
- can real-life applications be expressed in this form
Regarding the first, the results have been very positive, showing that where parallelism was available, Flow was able to use it to speed up the application; only limited by the number of CPU cores. Due to the fact that only a shared-memory implementation of the run-time engine was completed, this CPU core limitation is unfortunately significant, as on Amazon Web services we were only able to scale to 7 cores (1 was reserved for the scheduler). Potentially this model should be able to scale up to thousands of currently running threads, and being able to test and profile this should have provided some very interesting results.

Regarding the second question; the answer is yes, but not conclusive. The problem tackled was arguably trivial and was embarrassingly parallel. A more complex and interesting real-life workflow would, in retrospect, have given us a lot more confidence in the effectiveness of Flow. Unfortunately, trying to implement a large, complex application, while at the same time implementing a compiler and run-time engine, was beyond the scope of this dissertation, given the time-frame.

As mentioned, the answer to the second question is inconclusive as there were many features that would have been nice to prove, but were not. It is argued that inner-graph MPD has a lot of potential to positively impact the completion-time of an application, but without a complex graph (and run-time support for MPD) this feature remains un-evaluated. A more complex graph may also have placed more overhead on the scheduler, which could have proven to be a bottleneck. In the examples we have shown, the scheduler has never been an issue. Analysis of the cost of movement of data between CPU nodes would also have been an interesting test, and a greater variety of workflows would perhaps have identified some serious limitations in the Flow language’s feature set.
Chapter 9

Conclusions

In retrospect, the aims for this project were quite ambitious for a 16-week project. Many parts of this project could easily have been justified as a project in their own right. Despite this, however, the amount completed was not inconsequential and an end-to-end system has been implemented.

Figure 9.1 provides a visual representation of what was achieved.

9.1 What was implemented?

The plan was to firstly to design a domain specific language to allow programmers to represent how their data should be passed between tasks (i.e. functions/subroutines) in order to compute the desired outputs. A compiler and associated run-time engine was needed to take this DSL representation of a program and schedule those tasks in the right order. Not only this, but the data should also be passed correctly between tasks and it should be passed in a format that they expect (via a predefined interface). This is to allow any task to be combined with any other task, without the programmer having to write any additional plumbing code.

With such an implementation in place it was hypothesised that a number of optimisa-
tions could be performed, above and beyond those of traditional imperative-language compilers. These were not to replace the optimisations of those compilers, but instead just to add to them. Therefore the DSL compiler should generate optimised code in an existing language, in this case C++, for which there are many mature compilers already available to provide further optimisation.

These additional optimisations were firstly in the form of Minimal Path Discovery (i.e. trying to find the smallest set of tasks that need to be computed to achieve the desired result) and secondly in the form of automatic (and thread-safe) implicit parallelism. The plan was for the latter to expose as much parallelism as possible by assuming that every task in the program can be run in parallel unless there is an explicit direct or indirect data-dependency between them. This parallelism would only be restricted by the physical limit to how much compute power is available at runtime (i.e. no. of CPU cores).

Finally, once the compiler and run-time engine were available, it was planned to show the effectiveness of this DSL by using it to implement a workflow-type application and having it run in parallel. An existing geophysical sciences application, provided such an opportunity, and the DSL was to be designed with this implementation in mind.

As described so far, this was all successfully implemented during this dissertation project and works as an end-to-end system. In fact, in one respect the implementation went beyond what was arguably required, and was implemented as an Eclipse CDT [16] text editor plugin. This provides syntax highlighting, automatic compilation and underlining of any tokens that cause compilation errors. A programmer can write a flow source file in Eclipse and upon saving it will automatically compile that file, either showing any compilation errors inline, or automatically generating the necessary C++ source and header files. As an aside, it will also recompile any other Flow source files that depend on that file (if necessary).

Once the C++ source files have been generated, the only work left for the programmer to do is to initialise the scheduler from a main function and to implement the tasks themselves. An example of this is given in Appendix A. All the other plumbing and thread-management has been done for you. The function signatures for your subroutines (i.e. tasks) have conveniently been placed in the generated header file (flow-name_generated.hpp), so you just have to add a body to these. Compile the application with a C++ compiler of your choice, and you now have a fully functioning, thread-safe, parallel executable.

9.2 Future work

For future work there are a number of potential avenues that could be explored. Many of these have already been mentioned or discussed throughout this dissertation.

Using Flow to implement more complicated/complex workflows would provide for interesting analysis, and may identify limitations in the Flow language as it stands. The
Do-until loop and Switch node would be useful constructs to add support for in the parallel run-time engine, and the run-time engine could also be extended with the proposed distributed-memory logic (from chapter 7). This would allow programmers to build Flow applications that scale to hundreds, if not thousands, of CPU cores.

If more complicated workflows were to be implemented (that include inner graphs), the potential speedup that could be achieved through run-time-MPD removing redundant tasks (as described in chapter 6), may introduce a whole new level of application-level optimisation, not currently seen in any other language. It should also be trivial to add the planned Fortran subroutine support to the Flow compiler.

Apart from these, there was a list of potential features outlined in section 1.4 that data-flow applications are in a good position to take advantage of. These included automatic memory management and fault-tolerance; both of which would be essential for handling any large-scale applications.
Appendix A

Using Flow in practice

Listing A.1: The main function and one of the tasks that had to be written for Magnus’ application

```cpp
#include "Forward_generated.hpp"

Graph* buildGraphForward()
{
    Graph *forward = new Graph("forward",
                                &newForward, &buildForward);
    forward->addInput("input",
                       &Forward_set_input, &Forward_get_input);
    forward->addOutput("output",
                        &Forward_set_output, &Forward_get_output);
    return forward;
}

int main(int argc, char *argv[])
{
    int noOfWorkers = 2;
    if (argc > 1)
    {
        char *end;
        noOfWorkers = strtol(argv[1], &end, 10);
    }
    cout << "No. of workers: " << noOfWorkers << endl;
    ParallelSharedMemoryScheduler scheduler(noOfWorkers+1);
    Graph *forward = buildGraphForward();
    int id = 0;
    Forward_set_input(forward->getObj(), &id);
    scheduler.execute(*forward);
```
```c
void executeForward_Setup(void *obj) {
    Forward_Setup_obj* ptr = (Forward_Setup_obj*) obj;

    // execute setup programs
    std::system("SetupRSI25D");
    std::system("genModel1D_");
    std::system("genMesh25D_");
    std::system("genModel2D_");

    // generate frequencies, waveNos and srcPositions
    for (int i = 1; i <= 2; ++i) {
        ptr->freqs.add(&i, sizeof(int));
    }
    for (int i = 1; i <= 24; ++i) {
        ptr->waveNos.add(&i, sizeof(int));
    }
    for (int i = 1; i <= 10; ++i) {
        ptr->srcPoss.add(&i, sizeof(int));
    }
}
...
```
Appendix B

Flow grammar

Listing B.1: The ANTLR grammar file that describes the Flow grammar

/*
 *  one graph per file */
flow: nativeGraph* graph EOF
     -> ^(N_FLOW graph nativeGraph*);

/*  graph MyGraph (...) [...] [...] */
graph: 'graph' GRAPH_ID interfaceBody graphBody
      -> ^(N_GRAPH GRAPH_ID interfaceBody? graphBody?);
nativeGraph:
   'native' GRAPH_ID interfaceBody ':' VAR_ID ';
      -> ^(N_NATIVE
           GRAPH_ID
           interfaceBody?
           ^(N_LANGUAGE VAR_ID)
           );

/*  height: float, mother:Person, name:string */
declarations: ( declaration (',', declaration)* )?
      -> ^(N_DECLS declaration*));
declaration: VAR_ID ':' type
      -> ^(NDECL VAR_ID type);
/*  init myResult:int = 7 */
initDeclaration: 'init' declaration eq='=' rValue
doDeclarations: (doDeclaration (',') doDeclaration)*?
    -> ^(N_DO_DECLS doDeclaration*);
doDeclaration: declaration | initDeclaration;

statement: assignment ';'; -> assignment
   | definition ';'; -> definition
   | combine ';'; -> combine
   | next ';'; -> next;

node myVal = Person( name="anna" ); /*
definition: 'node' assignment
   -> ^(N_DEF assignment);
*/
next val = true; /*
next: 'next' assignment
   -> ^(N_NEXT assignment);
/* sum += myVal.result */
combine: VAR_ID combineOperator rValue
   -> ^(N_COMBINE ^(combineOperator VAR_ID rValue));
combineOperator: '+='|'*'|'.'='|'...'=';

/* e.g. MyType( input1 = myObj.val ) */
instantiation: graphRef '(' assignments ')' ';
   -> ^(N_INST graphRef assignments);

/* foreach (val in values) [sum:float] { ... } */
foreach: 'foreach' '(' VAR_ID 'in' varRef ')' '[ declarations ]' graphBody
   -> ^(N_FOREACH VAR_ID varRef graphBody? declarations);

dowhile: 'do' '[' doDeclarations ']' graphBody 'until' '(' varRef ')
   -> ^(N_DO varRef graphBody doDeclarations);

/* switch (varRef) [...] [ expr: [...] default: [...] ] */
switchStatement: 'switch' '(' varRef ')' '[ declarations ]' '{ switchBody '
   -> ^(N_SWITCH varRef declarations switchBody);
switchBody: (caseStatement)* defaultStatement;
caseStatement: constant colon=':' graphBody
   -> ^(N_CASE $ colon constant graphBody);
defaultStatement: 'default' colon=':' graphBody
   -> ^(N_CASE $ colon 'default' graphBody);

Lexer rules (capital letters)
for creating the individual tokens.
Note: if you want no spaces, it must be one token.

GRAPH_ID: NOT_I ANY_CHAR* | 'I' LOWERCASE ANY_CHAR*;
INTERFACE_ID: 'I' UPPERCASE ANY_CHAR*;
VAR_ID: LOWERCASE ANY_CHAR*;
NUMBER: DIGIT+ ('.' DIGIT+)? ;
STRING: '"' (~('"'))* '"' ? ;
WHITE_SPACE: (SPACE | NEW_LINE | TAB)+
   { $channel=HIDDEN; };

80
COMMENT: '/*' (options \{greedy=false;\} : .)* '*/'

{\$channel=HIDDEN;};

/* Fragments should only be used by other lexer rules */
fragment DIGIT: '0'..'9';
fragment LOWERCASE: 'a'..'z';
fragment UPPERCASE: 'A'..'Z';
fragment NOT_I: 'A'..'H' | 'J'..'Z';
fragment ANY_CHAR: DIGIT | LOWERCASE | UPPERCASE;
fragment NEW_LINE: '\n' | '\r' | '\r\n';
fragment SPACE: ' ';
fragment TAB: '\t';
Bibliography


